ПЕДАГОГИЧЕСКОЕ ОБРАЗОВАНИЕ

Д. Ш. Матрос
Г. Б. Поднебесова

ТЕОРИЯ АЛГОРИТМОВ

ПЕДАГОГИЧЕСКОЕ ОБРАЗОВАНИЕ

Д. Ш. Матрос
Г. Б. Поднебесова

ТЕОРИЯ АЛГОРИТМОВ

Учебник

Рекомендовано
УМО по специальностям
педагогического образования
в качестве учебника для студентов
высших учебных заведений,
обучающихся по специальности
050202.65(030100) — информатика

Москва
БИНОМ. Лаборатория знаний
2008

УДК 570
ББК 22.12
М33

Матрос Д. Ш.

М33 Теория алгоритмов : учебник / Д. Ш. Матрос, Г. Б. Поднебесова. — М. : БИНОМ. Лаборатория знаний, 2008. — 202 с. : ил. — (Педагогическое образование).
ISBN 978-5-94774-226-8

Учебник по курсу «Теория алгоритмов» для педагогических вузов по специальности «Информатика», полностью соответствующий стандарту.

Изложение имеет четкую логическую структуру и охватывает следующие темы: понятие алгоритма, машина Тьюринга, примитивно-рекурсивные функции, нормальные алгоритмы, вычислимость и разрешимость, сложность вычислений, NP-полные задачи. Каждая тема сопровождается тестовыми заданиями и упражнениями.

Для студентов и преподавателей педагогических вузов, учителей общеобразовательных школ.

УДК 570
ББК 22.12

По вопросам приобретения обращаться:

«БИНОМ. Лаборатория знаний» (499) 157-52-72, e-mail: Lbz@aha.ru

http://www.Lbz.ru

ISBN 978-5-94774-226-8

© Матрос Д. Ш., Поднебесова Г. Б., 2008
© БИНОМ. Лаборатория знаний, 2008

Оглавление

Предисловие 6

Глава I. Предварительные обсуждения 9
1.1. Неформальное понятие алгоритма 9
1.1.1. Основные требования к алгоритмам 12
1.1.2. Блок-схемы алгоритмов 17
1.1.3. Подходы к уточнению понятия алгоритма 19
1.2. Предварительные определения 21
1.2.1. Множества и функции 21
1.2.2. Функции от натуральных чисел 23
1.2.3. Отношения и предикаты. Логические обозначения 23
1.3. Алгоритм как программа для компьютера 24
Тестовые задания 27

Глава II. Машина Тьюринга 30
2.1. Основные определения 31
2.2. Операции над машинами Тьюринга 36
2.3. Универсальная машина Тьюринга 43
2.4. Тезис Тьюринга 45
2.5. Проблема остановки 46
2.6. Машина фон Неймана 48
Упражнения 49
Тестовые задания 52

Глава III. Рекурсивные функции 55
3.1. Примитивно-рекурсивные функции 55
3.2. Примитивно-рекурсивные операторы 60
3.3. Функции Аккермана 63
3.4. Частично-рекурсивные функции. Тезис Чёрча 64
Упражнения 67
Тестовые задания 68

Глава IV. Нормальные алгоритмы Маркова 72
4.1. Нормальные алгоритмы 72
4.2. Операции над алгоритмами Маркова. Принцип нормализации 79
Упражнения 86
Тестовые задания 88

Глава V. Машина с неограниченными регистрами 91
5.1. Основные определения 91
5.2. МНР-вычислимые функции 96
5.3. Порождение вычислимых функций 101
5.3.1. Соединение программ 102
5.3.2. Подстановка 105
5.3.3. Рекурсия 107
5.3.4. Минимизация 109
5.3.5. Развилка и повторение 111
5.4. Тезис Чёрча 113
Упражнения 115
Тестовые задания 117

Глава VI. Вычислимость и разрешимость 121
6.1. Эквивалентность различных теорий алгоритмов . 122
6.2. Нумерация алгоритмов 125
6.2.1. Нумерация программ 125
6.2.2. Нумерация вычислимых функций 128
6.3. Теоремы параметризации 132
6.4. Универсальный алгоритм 134
6.5. Неразрешимые проблемы в теории вычислимости 135
6.6. Разрешимые и перечислимые множества 141
6.7. Теорема Райса 145
Тестовые задания 147

Глава VII. Эффективные операции на множестве частичных функций 150
7.1. Рекурсивные операторы 150
7.2. Эффективные операции на вычислимых функциях 152
7.3. Первая теорема о рекурсии 154
7.4. Приложение к семантике языков программирования 156
7.5. Вторая теорема о рекурсии 159
Тестовые задания 161

Глава VIII. Сложность вычисления 163
8.1. Меры сложности 163
8.2. Теорема об ускорении. 167
8.3. Элементарные функции 170
Тестовые задания 173

Глава IX. Введение в теорию NP-полных задач 175
9.1. Формальные языки и грамматики 175
9.1.1. Основные понятия 176
9.1.2. Грамматики с фразовой структурой 178
9.1.3. Иерархия Хомского 179
9.2. Задачи распознавания, языки и кодирование . . . 181
9.3. Детерминированные машины Тьюринга и класс P 184
9.4. Недетерминированные вычисления и класс NP 185
9.5. Полиномиальная сводимость и NP-полные задачи 188
9.6. Примеры NP-полных задач 191
Тестовые задания 194

Литература 196
Предметный указатель 198
Обозначения 201

Предисловие

В соответствии со стандартами высшего профессионального образования второго поколения по специальности 030100 (Информатика в педагогических университетах и институтах) введен новый предмет «Теория алгоритмов».

Данный учебник содержит материал, полностью соответствующий требованиям государственного образовательного стандарта по этому предмету и реализующий основные цели:

- познакомить читателя с максимально широким кругом понятий теории алгоритмов. Количество определяемых понятий и специальных терминов превышает количество понятий, обсуждаемых более детально. Тем самым у студентов формируется терминологический запас, необходимый для самостоятельного изучения специальной математической программистской литературы;
- сообщить читателю необходимые конкретные сведения из теории алгоритмов, представляемые стандартом. Разбор доказательств приведенных утверждений, ответы на тестовые задания и выполнение упражнений позволяют студенту овладеть методами, наиболее употребляемыми при решении практических задач, и понять основные идеи современной теории алгоритмов;
- получить запас алгоритмически неразрешимых задач (проблем). Изучение таких задач необходимо программисту для уменьшения трудозатрат на «изобретение велосипеда» и обогащения его навыками конструирования алгоритмов.

Реализация этих целей позволит будущему учителю информатики понять важность основных положений теории алгоритмов для современной практики, осознать, что без их

глубокого знания невозможно построение современного программного обеспечения, необходимого для научных и прикладных исследований.

Сформулированные цели связаны также и с тем, что, к сожалению, даже многие учителя информатики и программисты не могут объяснить, например, почему за последние пятьдесят лет практически ничего не изменилось в процессе отладки программ, что особенно контрастирует с колоссальным прогрессом в области собственно программирования.

В последнее время теория алгоритмов — активно развивающаяся область, лежащая на стыке математики и информатики. Полученные результаты нашли отражение в ряде монографий. В первую очередь это вопросы, связанные со сложностью алгоритмов, и теория так называемых NP-полных задач. Но учебной литературы для педагогических вузов, которая освещала бы все эти проблемы (от классических построений до результатов последних лет), нет.

Все это привело к следующей структуре книги.

В первой главе вводится неформальное понятие алгоритма, формулируются основные требования к нему, напоминаются основные определения из других разделов математики. Рассматривается алгоритм как программа для компьютера и структурная теорема Бома–Джакопини.

Следующие четыре главы посвящены наиболее принципиальным подходам при введении понятия алгоритма (машины Тьюринга, рекурсивные функции, нормальные алгоритмы Маркова, машины с неограниченными регистрами (МНР)). Теоретическая строгость изложения везде доводится до интерпретации полученных результатов для практики современного программирования. Более того, показано, что с помощью любой из предложенных теорий можно «запрограммировать» любую реализованную на компьютере задачу, а вот доказать несуществование компьютерной программы для решения задачи можно только с помощью какой-то из этих теорий. Такой методологический подход очень важен для будущих учителей.

В шестой главе показывается эквивалентность различных теорий алгоритмов и доказывается алгоритмическая неразрешимость некоторых проблем в теории вычислимости. При этом подчеркивается, что основная причина — общность в постановке тех или иных задач. То есть при определенной конкретизации задачи становятся вполне решаемыми, о чем и свидетельствует опыт программирования на компьютере.

Седьмая глава посвящена теоретическим основам и приложениям к языкам программирования рекурсивных операторов.

В восьмой главе рассматривается вопрос вычислимости функции за какое-то определенное время. Знаменитая теорема Блюма об ускорении показывает, что «естественный» путь определения сложности функции через сложность реализующей ее программы невозможен. В связи с этим вводятся специальные классы сложности и рассматривается класс хорошо знакомых нам со школы элементарных функций.

В заключительной главе дается введение в теорию NP-полных задач. Приводится большое количество примеров и формулируется одна из важнейших проблем современной математики — как соотносятся между собой классы P и NP.

Каждая глава заканчивается контрольным тестом и упражнениями. Как показывает опыт, выполнение упражнений позволяет лучше усвоить пройденный материал, а тестовые задания дают возможность организовать рейтинговую систему в процессе обучения предмету. В книге конец упражнения обозначен символом $\triangle$, а знак $\square$ указывает на завершение доказательства теоремы.

При изложении авторы следовали монографиям С. К. Клини, О. П. Кузнецова и Г. М. Адельсон-Вельского, Н. Катленда, А. И. Мальцева, А. А. Маркова и Н. И. Нагорного, М. Гэри и Д. Джонсона и др. Этими работами мы и рекомендуем пользоваться для дальнейшего изучения материала.

Данная книга предназначена для будущих учителей информатики, которые уже прошли такие предметы, как высшая математика, математическая логика, дискретная математика, теория вероятностей, программное обеспечение, программирование. Поэтому предполагается, что читатель не испытает затруднений в понимании математических текстов.

Авторы будут рады любым замечаниям и предложениям, которые можно направлять по адресу:

454080, г.Челябинск, пр. Ленина, 69,

Челябинский государственный педагогический университет, факультет информатики.

E-mail: matros@cspu.ru

Глава I

Предварительные обсуждения

Данная глава посвящена изложению предварительных результатов, которые на неформальном уровне вводят нас в круг необходимых проблем.

В первом пункте дается интуитивное понятие алгоритма, излагаются основные требования к нему. В дальнейшем, при различных уточнениях этого понятия, мы будем показывать, что все предлагаемые конструкции удовлетворяют сформулированным требованиям. Подробно рассматриваются блок-схемы как один из самых распространенных и наглядных способов представления алгоритмов. Описываются основные идеи различных подходов к уточнению понятия алгоритма, которые будут рассмотрены в последующих главах.

Во втором пункте вводятся, скорее напоминаются, основные определения из других математических дисциплин, которые будут использоваться в дальнейшем.

В третьем пункте алгоритм рассматривается как программа для компьютера. Из структурной теоремы следует, что необходимо показать возможность реализации каждой из трех базовых структур при любом уточнении понятия алгоритма. Отсюда будет следовать возможность реализации любой программы для компьютера средствами модели, уточняющей понятие алгоритма.

1.1. Неформальное понятие алгоритма

С *алгоритмами*, т. е. эффективными процедурами [8], однозначно приводящими к результату, математика имела дело всегда. Школьные методы умножения «столбиком» и деления «углом», метод исключения неизвестных при решении системы линейных уравнений, правило дифференцирования сложных функций, способ построения треугольника по трем заданным сторонам — все это алгоритмы. Однако

пока математика имела дело в основном с числами и вычислениями и понятие алгоритма отождествлялось с понятием метода вычисления, потребности в изучении самого этого понятия не возникало. Традиции организации вычислений складывались веками и стали составной частью общей научной культуры в той же степени, что и элементарные навыки логического мышления. Все многообразие вычислений комбинировалось из 10–15 четко определенных операций арифметики, тригонометрии и анализа. Поэтому понятие метода вычисления считалось изначально ясным и не нуждалось в специальных исследованиях.

До середины XIX в. единственной областью математики, работавшей с нечисловыми объектами, была геометрия, и как раз она, не имея возможности опираться на вычислительную интуицию человека, резко отличалась от остальной математики повышенными требованиями к строгости своих рассуждений. До сих пор любой современный семиклассник, для которого математика — это мир вычислений, мучительно привыкает к понятиям доказательства и математического построения и никак не может понять, зачем доказывать равенство отрезков, когда проще измерить, и зачем строить перпендикуляр с помощью циркуля и линейки, когда есть угольник с «готовым» прямым углом или транспортир.

Такое же мучительное привыкание к новым, более жестким требованиям строгости началось в математике во второй половине XIX века. Оно стимулировалось в основном математикой нечисловых объектов — открытием неэвклидовой геометрии, появлением абстрактных алгебраических теорий типа теорий групп и т. д. Одним из решающих обстоятельств, приведших к пересмотру оснований математики, т. е. принципов, лежащих в основе математических рассуждений, явилось создание Г. Кантором теории множеств. Довольно быстро стало ясно, что понятия теории множеств в силу своей общности лежат в основе всего здания математики. Однако почти столь же быстро было показано, что некоторые кажущиеся вполне естественные рассуждения в рамках этой теории приводят к неразрешимым противоречиям — парадоксам теории множеств (более подробно см. [6]). Все это потребовало точного изучения принципов математических рассуждений (до сих пор казавшихся интуитивно ясными) математическими же свойствами. Возникла особая отрасль математики — основание математики, или метаматематика.

Опыт парадоксов теории множеств научил математику крайне осторожно обращаться с бесконечностью и по возможности даже о бесконечности рассуждать с помощью финитных методов. Сущность финитного подхода заключается в том, что он допускает только конечный набор действий над конечным числом объектов. Выявление того, какие объекты и действия над ними следует считать точно определенными, какими свойствами и возможностями обладают комбинации элементарных действий, что можно и чего нельзя сделать с их помощью — все это стало предметом теории алгоритмов и формальных систем, которая первоначально возникла в рамках математики и стала важнейшей ее частью. Главным внутриматематическим приложением теории алгоритмов явилось доказательство невозможности алгоритмического (т. е. точного и однозначного) решения некоторых математических проблем. Такие доказательства (да и точные формулировки доказываемых утверждений) неосуществимы без точного понятия алгоритма.

Пока техника использовала вычислительные методы, эти высокие проблемы чистой математики ее мало интересовали. В технику термин «алгоритм» пришел вместе с кибернетикой. Если понятие метода вычисления не нуждалось в пояснениях, то понятие процесса управления пришлось вырабатывать практически заново. Понадобилось осознавать, каким требованиям должна удовлетворять последовательность действий (или ее описание), чтобы считаться конструктивно заданной, т. е. иметь право называться алгоритмом. В этом осознании огромную помощь инженерной интуиции оказала практика использования вычислительных машин, сделавшая понятие алгоритма ощутимой реальностью. С точки зрения современной практики *алгоритм — это программа*, а критерием алгоритмичности процесса является возможность его запрограммировать. Именно благодаря этой реальности алгоритма, а также потому, что подход инженера к математическим методам всегда был конструктивным, понятие алгоритма в технике за короткий срок стало необычайно популярным (быть может, даже больше, чем в самой математике).

Однако у всякой популярности есть свои издержки, в повседневной практике слово «алгоритм» употребляется слишком широко, теряя зачастую свой точный смысл. Приблизительные описания понятия «алгоритм» (вроде того, которое приведено в первой фразе этого пункта) часто принимаются за точные определения. В результате за алго-

ритм зачастую выдается любая инструкция, разбитая на шаги. Появляются такие дикие словосочетания, как «алгоритм изобретения» (а ведь наличие «алгоритма изобретения» означало бы конец изобретательства как творческой деятельности).

Ясное представление о том, что такое алгоритм, важно, конечно, не только для правильного словоупотребления. Оно важно и при разработке конкретных алгоритмов, особенно когда имеется в виду их последующее программирование. Чтобы ориентироваться в море алгоритмов, необходимо уметь сравнивать различные алгоритмы решения одних и тех же задач, причем не только по качеству решений, но и по характеристикам самих алгоритмов (числу действий, расходу памяти и т. д.). Такое сравнение невозможно без введения точного языка для обсуждения всех этих вопросов; иначе говоря, сами алгоритмы должны стать такими же предметами точного исследования, как и те объекты, для работы с которыми они предназначены.

Строгое исследование основных понятий теории алгоритмов начнется в следующей главе. Прежде чем приступить к нему, обсудим на неформальном уровне некоторые основные принципы, по которым строятся алгоритмы, и выясним, что же именно в понятии алгоритма нуждается в уточнении.

1.1.1. Основные требования к алгоритмам

Отметим несколько общих черт алгоритмов, ясно вырисовывающихся из вышесказанного и часто признающихся характерными [7,11] для понятия алгоритма:

a) алгоритм — это процесс последовательного построения величин, идущий в дискретном времени таким образом, что в начальный момент задается исходная конечная система величин, а в каждый следующий момент система величин получается по определенному закону (программе) из системы величин, имевшихся в предыдущий момент времени (*дискретность алгоритма*);

b) система величин, получаемых в какой-то (не начальный) момент времени, однозначно определяется системой величин, полученных в предшествующие моменты времени (*детерминированность алгоритма*);

c) закон получения последующей системы величин из предшествующей должен быть простым и локальным (*элементарность шагов алгоритма*), типичный пример множества элементарных действий — система команд ЭВМ;
d) если способ получения последующей величины из какой-нибудь заданной величины не дает результата, то должно быть указано, что надо считать результатом алгоритма (*результативность алгоритма*);
e) начальная система величин может выбираться из некоторого потенциально бесконечного множества (*массовость алгоритма*).

Понятие алгоритма, в какой-то мере определяемое этими требованиями, конечно, нестрогое: в формулировках требований встречаются слова «способ», «величина», «простой», «локальный», точный смысл которых не установлен. В дальнейшем это нестрогое понятие алгоритма будет называться **непосредственным** или **интуитивным понятием алгоритма**.

Сделаем несколько комментариев к описанным требованиям. Любой алгоритм применяется к исходным данным и выдает результат. В привычных технических терминах это означает, что алгоритм имеет входы и выходы. Кроме того, в ходе работы алгоритма появляются промежуточные результаты, которые используются в дальнейшем. Таким образом, каждый алгоритм имеет дело с данными — входными, промежуточными и выходными. Поскольку мы собираемся уточнять понятие алгоритма, нужно уточнить и понятие данных, т. е. указать, каким требованиям должны удовлетворять объекты, чтобы алгоритмы могли с ними работать.

Ясно, что эти объекты должны быть четко определены и отличимы как друг от друга, так и от «необъектов». Во многих случаях хорошо понятно, что это значит: к таким алгоритмическим объектам относятся числа, векторы, матрицы смежностей графов, формулы. Изображения (например, рисунок графа) представляются менее естественными в качестве алгоритмических объектов. Если говорить о графе, то дело даже не в том, что в рисунке больше несущественных деталей и два человека один и тот же граф изобразят по-разному (в конце концов, разные матрицы смежности тоже могут задавать один и тот же граф с точностью до изоморфизма), а в том, что матрица смежности легко разбивается на элементы, причем из элементов всего двух видов (нулей и

единиц) строят матрицы любых графов, тогда как разбить на элементы рисунок гораздо труднее. Наконец, с такими объектами, как хорошая книга или осмысленное утверждение, с которыми легко управляется любой человек (но каждый по-своему!), алгоритм работать не сможет, пока они не будут описаны как данные с помощью других, более подходящих объектов.

Вместо того чтобы пытаться дать общее словесное определение четкой определенности объекта, в теории алгоритмов фиксируют конкретные конечные наборы исходных объектов (называемых элементарными) и конечный набор средств построения других объектов из элементарных. Набор элементарных объектов образует конечный *алфавит* исходных символов (цифр, букв и т. д.), из которых строятся другие объекты.

Типичным средством построения являются индуктивные определения, указывающие, как строить новые объекты из уже построенных. Простейшее индуктивное определение — это определение некоторого множества слов, классическим примером которого служит определение идентификатора в ПАСКАЛе: идентификатор — это либо буква, либо идентификатор, к которому приписана справа буква или цифра. Слова конечной длины в конечных алфавитах (в частности, числа) — самый простой тип алгоритмических данных, а число символов в слове (длина слова) — естественная единица измерения объема обрабатываемой информации. Более сложный случай алгоритмических объектов — формулы. Они также определяются индуктивно и также являются словами в конечном алфавите, однако не каждое слово в этом алфавите является формулой. В этом случае обычно основным алгоритмам предшествуют вспомогательные, которые проверяют, удовлетворяют ли исходные данные нужным требованиям. Такая проверка называется синтаксическим анализом.

Данные для своего размещения требуют *памяти*. Память обычно считается однородной и дискретной, т. е. состоит из одинаковых ячеек, причем каждая ячейка может содержать один символ алфавита данных. Таким образом, единицы измерения объема данных и памяти согласованы. При этом память может быть бесконечной. Вопрос о том, нужна ли отдельная память для каждого из трех видов данных (входных, выходных и промежуточных), решается по-разному.

Требование результативности, в частности, предполагает, что всякий, кто предъявляет алгоритм решения некоторой задачи, например вычисления функции $f(x)$, обязан показать, что алгоритм останавливается после конечного числа шагов (как говорят, сходится) для любого x из области задания f. Однако проверить результативность (сходимость) гораздо труднее, чем требования, изложенные в других пунктах. В отличие от них сходимость обычно не удается установить простым просмотром описания алгоритма; общего же метода проверки сходимости, пригодного для любого алгоритма A и любых входных данных x, как будет показано далее, вообще не существует.

Если алгоритм используется для вычисления значений числовой функции, то эта функция называется *эффективно вычислимой*, или *алгоритмически вычислимой*, или просто *вычислимой*.

Следует различать:

1) описание алгоритма (инструкцию или программу);
2) механизм реализации алгоритма (например, компьютер), включающий средства пуска, остановки, реализации элементарных шагов, выдачи результатов и обеспечения детерминированности, т. е. управления ходом вычисления;
3) процесс реализации алгоритма, т. е. последовательность шагов, которая будет порождена при применении алгоритма к конкретным данным.

Будем предполагать, что описание алгоритма и механизм его реализации конечны (память, как уже говорилось, может быть бесконечной, но она не включается в механизм). Требование конечности процесса реализации совпадает с требованием результативности.

Пример 1. Рассмотрим следующую задачу: дана последовательность P из n положительных чисел (n — конечное, но произвольное число); требуется упорядочить их, т. е. построить последовательность R, в которой эти же числа расположены в порядке возрастания. Почти сразу же приходит в голову следующий простой способ ее решения: просматриваем P и находим в ней наименьшее число; вычеркиваем его из P и выписываем его в качестве первого числа R; снова обращаемся к P и находим в ней наименьшее число; приписываем его справа к полученной части R и так далее, до тех пор, пока из P не будут вычеркнуты все числа.

Возникает естественный вопрос: что значит «и так далее»? Для большей ясности перепишем описание способа решения в более четкой форме, разбив его на шаги и указав переходы между шагами.

Шаг 1. Ищем в P наименьшее число.

Шаг 2. Найденное число приписываем справа к R (в начальный момент R пуста) и вычеркиваем его из P.

Шаг 3. Если в P нет чисел, то переходим к шагу 4. В противном случае переходим к шагу 1.

Шаг 4. Конец. Результатом считать последовательность R, построенную к данному моменту. △

Большинство читателей сочтет такое описание достаточно ясным (и даже излишне формальным), чтобы, пользуясь им, однозначно получить нужный результат. Однако это впечатление ясности опирается на некоторые неявные предположения, к правильности которых мы привыкли, но которые нетрудно нарушить. Например, что значит «дана последовательность чисел»? Является ли таковой последовательность $\sqrt[7]{3}$, $\sqrt[5]{2}$, $(1{,}2)^{\pi}$? Очевидно, да, однако в нашем описании ничего не сказано, как найти наименьшее число среди таких чисел. В нем вообще не говорится о том, как искать наименьшие числа, по-видимому, предполагается, что речь идет о числах, представленных в виде десятичных дробей, и что известно, как их сравнивать.

Итак, необходимо уточнить формы представления данных. При этом нельзя просто заявить, что допустимо любое представление чисел. Ведь для каждого представления существует свой алфавит (который, помимо цифр, может включать запятые, скобки, знаки операций и функций и т. д.) и свой способ сравнения чисел (например, перевод в десятичную дробь), тогда как конечность алфавита требует фиксировать его заранее, а конечность описания алгоритма позволяет включить в него лишь заранее фиксированное число способов сравнения. Фиксация представления чисел в виде десятичных дробей также не решает всех проблем. Сравнение 10–20-разрядных чисел уже не может считаться элементарным действием: сразу нельзя сказать, какое из чисел больше: 90811557001,15 или 32899901467,0048. В машинных алгоритмах само представление числа еще требует дальнейшего уточнения: нужно, во-первых, ограничить число разрядов (цифр) в числе, так как от этого зависит, сколько ячеек памяти будет занимать число, а во-вторых, договориться о способе размещения десятичной запятой в числе,

т. е. выбрать представление в виде числа с фиксированной или плавающей запятой, поскольку способы обработки этих двух представлений различны. Наконец, любой, кто имел дело с программированием, отметит, что на шаге 1 требуется узнать две вещи: само наименьшее число (чтобы записать его в R) и его место в P, т. е. его адрес в той части памяти, где хранится P (чтобы вычеркнуть его из P), а следовательно, нужно иметь средства указания этого адреса.

Таким образом, даже в этом простом примере несложный анализ показывает, что описанию, которое выглядит вполне ясным, еще далеко до алгоритма. Мы столкнулись здесь с необходимостью уточнить почти все основные характеристики алгоритма, которые отмечались ранее: алфавит данных и форму их представления, память и размещение в ней элементов P и R, элементарные шаги (поскольку шаг 1 явно не элементарен). Кроме того, становится ясным, что выбор механизма реализации (скажем, человека или компьютера) будет влиять и на сам характер уточнения: у человека требования к памяти, представлению данных и к элементарности шагов гораздо более слабые и «укрупненные», отдельные незначительные детали он может уточнить сам.

Пожалуй, только два требования к алгоритмам в приведенном описании выполнены в достаточной мере (они-то и создают впечатление ясности). Довольно очевидна сходимость алгоритма: после выполнения шагов 1 и 2 либо работа заканчивается, либо из P вычеркивается одно число; поэтому ровно после n выполнений шагов 1 и 2 из P будут вычеркнуты все числа и алгоритм остановится, а R будет результатом. Кроме того, не вызывает сомнения детерминированность: после каждого шага ясно, что делать дальше, если учесть, что здесь и в дальнейшем используется общепринятое соглашение — если шаг не содержит указаний о дальнейшем переходе, то после него выполняем шаг, следующий за ним в описании. Поскольку использованные в примере средства обеспечения детерминированности носят довольно общий характер, остановимся на них несколько подробнее.

1.1.2. Блок-схемы алгоритмов

Связи между шагами можно изобразить в виде графа. Для примера из предыдущего пункта граф изображен на рис. 1.1.

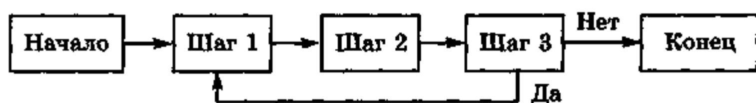


Рис. 1.1

Такой граф, в котором вершинам соответствуют шаги, а ребрам — переходы между шагами, называется *блок-схемой* алгоритма. Его вершины могут быть двух видов: вершины, из которых выходит одно ребро (их называют *операторами*), и вершины, из которых выходят два ребра (их называют *логическими условиями*, или *предикатами*). Кроме того, имеется единственный оператор конца, из которого не выходит ни одного ребра, и единственный оператор начала. Важной особенностью блок-схемы является то, что связи, которые она описывает, не зависят от того, являются шаги элементарными или представляют собой самостоятельные алгоритмы или, как говорят в программировании, блоки (по существу, шаг 1 таковым и является). Возможность «разблочивать» алгоритм хорошо известна и широко используется в программировании: большой алгоритм разбивается на блоки, которые можно раздать для программирования разным лицам. Для отдельного блока неважно, как устроены другие блоки; для программирования блока достаточно знать, где лежит вся исходная информация, какова форма ее представления, что должен делать блок и куда записать результат. С помощью блок-схем можно, наоборот, несколько алгоритмов, рассматриваемых как блоки, связать в один большой алгоритм. В частности, если алгоритм A_1, вычисляющий функцию $f_1(x)$, соединен с алгоритмом A_2, вычисляющим функцию $f_2(x)$, и при этом исходными данными для A_2 служит результат A_1, то полученная блок-схема задает алгоритм, вычисляющий $f_2(f_1(x))$, т. е. композицию функций f_1 и f_2 (рис. 1.2). Такое соединение алгоритмов называется *композицией* алгоритмов.

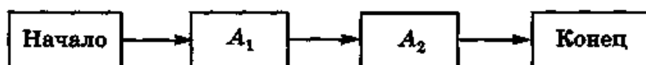


Рис. 1.2

На блок-схеме хорошо видна разница между описанием алгоритма и процессом его реализации. Описание — это граф, процесс реализации — это путь в графе. Различные

пути в одном и том же графе возникают при различных данных, которые создают разные логические условия в точках разветвления. Отсутствие сходимости означает, что в процессе вычисления не появляется условий, ведущих к концу, и процесс идет по бесконечному пути (зацикливается).

При всей наглядности языка блок-схем не следует, однако, переоценивать его возможности. Блок-схема отражает связи лишь по управлению (что делать в следующий момент, т. е. какому блоку передать управление), а не по информации (где этому блоку брать исходные данные). Например, рис. 1.2 при сделанной оговорке относительно данных изображает вычисление $f_2(f_1(x))$, однако он же мог изображать последовательное вычисление двух независимых функций $f_1(5)$ и $f_2(100)$, порядок которых несуществен. Блок-схемы не содержат сведений ни о данных, ни о памяти, ни об используемом наборе элементарных шагов. В частности, надо иметь в виду, что если в блок-схеме нет циклов, это еще не значит, что нет циклов в алгоритме (они могут быть в каком-нибудь неэлементарном блоке). По существу, блок-схемы — это не язык, а очень удобное средство для описания детерминированности алгоритма, и оно будет неоднократно использоваться в дальнейшем.

1.1.3. Подходы к уточнению понятия алгоритма

Ранее были сформулированы основные требования к алгоритмам. Однако понятия, использованные в этих формулировках (такие, как ясность, четкость, элементарность), сами нуждаются в уточнении. Очевидно, что их словесные определения будут содержать новые понятия, которые снова потребуют уточнения, и т. д.

Поэтому в теории алгоритмов принимается другой подход: выбирается конечный набор исходных объектов, которые объявляются элементарными, и конечный набор способов построения из них новых объектов. Этот подход был уже использован при обсуждении вопроса о данных: уточнением понятия «данные» в дальнейшем будем считать множества слов в конечных алфавитах. Для уточнения детерминизма будут использоваться либо блок-схемы и эквивалентные им словесные описания (аналогичные приведенным в примере 1), либо описание механизма реализации алгоритма. Кроме того, нужно зафиксировать набор элементарных шагов и договориться об организации памяти.

После того как это будет сделано, получится конкретная алгоритмическая модель.

Алгоритмические модели, рассматриваемые в этой книге, претендуют на право считаться формализацией понятия «алгоритм». Это значит, что они должны быть универсальными, т. е. допускать описание любых алгоритмов. Поэтому может возникнуть естественное возражение против предлагаемого подхода: не приведет ли выбор конкретных средств к потере общности формализации? Если иметь в виду основные цели, стоявшие при создании теории алгоритмов,— универсальность и связанную с ней возможность говорить в рамках какой-либо модели о свойствах алгоритмов вообще, то это возражение снимается следующим образом.

Во-первых, доказывается сводимость одних моделей к другим, т. е. показывается, что всякий алгоритм, описанный средствами одной модели, может быть описан средствами другой. Во-вторых, благодаря взаимной сводимости моделей в теории алгоритмов удалось выработать инвариантную по отношению к моделям систему понятий, позволяющую говорить о свойствах алгоритмов независимо от того, какая формализация алгоритма выбрана. Эта система понятий основана на понятии вычислимой функции, т. е. функции, для вычисления которой существует алгоритм. Общее понятие вычислимости и его свойства будут более подробно рассмотрены в дальнейшем.

Тем не менее, хотя общность формализации в конкретной модели не теряется, различный выбор исходных средств приводит к моделям разного вида. Можно выделить три основных типа универсальных алгоритмических моделей, различающихся исходными эвристическими соображениями относительно того, что такое алгоритм.

Первый тип связывает понятие алгоритма с наиболее традиционными понятиями математики — вычислениями и числовыми функциями. Наиболее развитая и изученная модель этого типа — **рекурсивные функции** — является исторически первой формализацией понятия алгоритма.

Второй тип основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять в каждый отдельный момент лишь весьма примитивные операции. Такое представление не оставляет сомнений в однозначности алгоритма и элементарности его шагов. Кроме того, эвристика этих моделей близка к компьютеру. Основной теоретической моделью этого типа (созданной

в 30-х годах — раньше самых первых ЭВМ!) является **машина Тьюринга.**

Третий тип алгоритмических моделей — это преобразования слов в произвольных алфавитах, в которых элементарными операциями являются подстановки, т. е. замены части слова (подслова) другим словом. Преимущества этого типа — в его максимальной абстрактности и возможности применить понятие алгоритма к объектам произвольной (не обязательно числовой) природы. Впрочем, как будет ясно из дальнейшего, модели второго и третьего типа довольно близки (их взаимная сводимость доказывается просто) и отличаются в основном эвристическими акцентами. Пример моделей этого типа — **нормальные алгоритмы Маркова.**

В дополнение к этим основным типам универсальных моделей мы изложим еще подход, основанный на так называемых **машинах с неограниченными регистрами (МНР),** известными также под названием **адресных машин (RAM).** Для читателей, соприкасающихся с программированием, изложение на базе МНР особенно привлекательно ввиду близости этих машин к реальным компьютерам и удобного представления их программ в виде блок-схем.

1.2. Предварительные определения

1.2.1. Множества и функции

Для обозначения множеств будем использовать прописные буквы A, B, C, D, ... Пустое множество обозначается символом $\varnothing$. Стандартный символ $\mathbb{N}$ обозначает множество натуральных чисел $\{0, 1, 2, ...\}$, $\mathbb{N}^+$ обозначает множество положительных натуральных чисел $\{1, 2, 3, ...\}$, а $\mathbb{Z}$ — множество целых чисел.

Упорядоченная пара элементов x, y обозначается (x, y); таким образом, вообще говоря, $(x, y) \neq (y, x)$.

Декартовым произведением множеств A и B называется множество

$$\{(x, y) \mid x \in A, y \in B\},$$

и обозначается оно $A \times B$.

Запись $(x_1, x_2, ..., x_n)$ обозначает упорядоченный n-набор (набор из n элементов $x_1, ..., x_n$); этот набор обозначается одной жирной буквой $\mathbf{x}$, т. е. $\mathbf{x} = (x_1, x_2, ..., x_n)$. Если

$A_1, A_2, \ldots, A_n$ суть множества, то их декартово произведение $A_1 \times A_2 \times \ldots \times A_n$ обозначает множество

$$\{(x_1, \ldots, x_n) \mid x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n\}.$$

Произведение $A \times A \times \ldots \times A$ (n раз) сокращенно обозначают как A^n, а A^1 означает просто A.

Областью определения функции f называется множество $\{x \mid f(x)$ определено$\}$ и обозначается $Dom(f)$.

Множество $\{f(x) \mid x \in Dom(f)\}$ называется *множеством значений*, или *образом* (range) функции f, и обозначается $Ran(f)$.

Если A и B — множества, то будем говорить, что f есть функция из A в B, если $Dom(f) \subset A$ и $Ran(f) \subset B$, и писать $f: A \to B$.

Функция называется *инъективной*, если из $x, y \in Dom(f)$ и $x \neq y$ следует, что $f(x) \neq f(y)$. Для инъективной функции f через f^{-1} обозначается функция, *обратная* к f, то есть такая единственная функция g, что $Dom(g) = Ran(f)$ и $g(f(x)) = x$ для всех $x \in Dom(f)$.

Функция f из A в B называется *сюръективной*, если $Ran(f) = B$.

Инъективную (сюръективную) функцию $f: A \to B$ будем называть просто *инъекцией* из A в B (*сюръекцией* A на B). Функция, являющаяся одновременно инъекцией и сюръекцией, называется *биекцией*.

Ограничением функции f на множестве X называется функция с областью определения $X \cap Dom(f)$, значение которой для каждого $x \in X \cap Dom(f)$ равно $f(x)$; обозначается $f \mid X$. $Ran(f \mid x)$ обозначается через $f(x)$.

Если Y — множество, то *прообразом* Y относительно f называется множество $f^{-1}(Y) = \{x \mid f(x) \in Y\}$.

Если f, g — функции, то будем говорить, что g *продолжает* f, если $Dom(f) \subseteq Dom(g)$ и $f(x) = g(x)$ для всех $x \in Dom(f)$; обозначается $f = g \mid Dom(f)$. Это отношение функций f, g записывается как $f \subseteq g$.

Композиция двух функций f и g есть функция с областью определения $\{x \mid x \in Dom(g)$ и $g(x) \in Dom(f)\}$, значение которой, когда она определена, равно $f(g(x))$. Эту функцию обозначают через $f \circ g$.

Через $f_\varnothing$ обозначим нигде не определенную функцию, т. е. $Dom(f_\varnothing) = Ran(f_\varnothing) = \varnothing$. Очевидно, что $f_\varnothing = g \mid \varnothing$ для любой функции g.

Пусть $\alpha(x)$ и $\beta(x)$ — выражения, включающие переменные $x = (x_1, \ldots, x_n)$. Тогда запись $\alpha(x) \simeq \beta(x)$ означает, что для каждого x выражения $\alpha(x)$ и $\beta(x)$ либо одновременно определены и равны, либо оба не определены.

1.2.2. Функции от натуральных чисел

Преимущественно в этой книге мы будем иметь дело с функциями от натуральных чисел, т. е. с функциями из $\mathbb{N}^n$ в $\mathbb{N}$ для разных n, большей частью для $n = 1$ или 2.

Функция f из $\mathbb{N}^n$ в $\mathbb{N}$ называется n-местной функцией. Значение f на наборе $(x_1, \ldots, x_n) \in Dom(f)$ записывается как $f(x_1, \ldots, x_n)$ или $f(x)$, если $x = (x_1, \ldots, x_n)$. Подчеркнем, что для нас слово функция означает *частичную* функцию, т. е. функцию из $\mathbb{N}^n$ в $\mathbb{N}$, область определения которой не обязательно совпадает с $\mathbb{N}^n$. Но если требуется, мы будем специально писать «частичная функция», чтобы подчеркнуть ее возможную «не всюду определенность». *Тотальной* функцией из $\mathbb{N}^n$ в $\mathbb{N}$ называется такая функция, область определения которой совпадает с $\mathbb{N}^n$.

1.2.3. Отношения и предикаты. Логические обозначения

Если A — множество, то свойство $M(x_1, \ldots, x_n)$, выполняющееся (или истинное) на некоторых наборах из A^n и не выполняющееся (или ложное) на всех других наборах из A^n, называется *n-местным отношением*, или *предикатом* на A.

Бинарное отношение R на множестве A называется *отношением эквивалентности*, если для всех x, y, $z \in A$ имеет место:

1) рефлексивность, т. е. $R(x, x)$;
2) симметричность, т. е. $R(x, y)$ влечет $R(y, x)$;
3) транзитивность, т. е. если $R(x, y)$ и $R(y, z)$, то $R(x, z)$.

Класс эквивалентности x есть множество $\{y \mid R(x, y)\}$, т. е. множество, состоящее из всех элементов, эквивалентных x.

Бинарное отношение R на множестве A называется *частичным порядком*, если для всех x, y, $z \in A$ имеет место:

1) иррефлексивность, т. е. не $R(x, x)$;
2) транзитивность, т. е. если $R(x, y)$ и $R(y, z)$, то $R(x, z)$.

Частичный порядок обычно обозначают символом $<$ и записывают $x < y$. Часто определяют частичный порядок, вводя сначала предикат $\leq$ (обозначающий $<$ или $=$) со свойствами:

1) $x \leq x$;
2) если $x \leq y$ и $y \leq x$, то $x = y$;
3) отношение $\leq$ транзитивно;

а затем определяя $x < y$ как $x \leq y$ и $x \neq y$.

В дальнейшем мы будем употреблять стандартные логические отношения:

$\equiv$ — эквивалентно по определению;
$\Rightarrow$ — логическое следование;
$\Leftrightarrow$ — тогда и только тогда;
$\forall$ — для всех;
$\exists$ — существует.

1.3. Алгоритм как программа для компьютера

Как было отмечено выше, для современной практики алгоритм — это программа для компьютера, а алгоритмичность процесса определяется возможностью его запрограммировать. Более того, многие практики довольно скептически относятся к основным теоретическим построениям, предпочитая все решать напрямую — программированием на том или ином языке.

Относясь с пониманием к такому подходу, мы, однако, покажем его ограниченность в следующем смысле:

- все, что можно запрограммировать на любом алгоритмическом языке, можно представить в изложенных ниже теоретических построениях;
- только теоретические построения, приводящие к формализации понятия «алгоритм», показывают, какие задачи вообще нельзя запрограммировать.

Для обоснования первого положения нам необходимо сначала разобраться, что такое программа.

В соответствии со структурной теоремой К. Бома и Г. Джакопини (Bohm C., Jacopini G.), всякая программа может быть построена с использованием только трех структур: следование, развилка и повторение. Для изображения этих структур на схемах введем понятие функционального блока, который используют для обозначения действия по обработке информации (Хьюз Дж., Мичтом Дж.; Фролов Г. Д., Кузнецов Э. И.).

Функциональный блок:

1) содержит единственный вход;
2) содержит единственный выход;
3) не содержит бесполезных (недостижимых) фрагментов;
4) не содержит бесконечных циклов.

На схеме функциональный блок обозначается в виде прямоугольника, имеющего один вход и один выход, внутри которого записываются действия по обработке информации (рис. 1.3, S — производимая обработка).

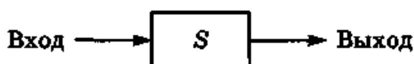


Рис. 1.3. Функциональный блок

Следование. Эта структура означает, что управление передается от одного функционального блока к следующему и изображается в виде последовательности функциональных блоков, соединенных стрелками (рис. 1.4). Стрелки показывают последовательность выполнения действий по обработке информации. Сложное действие, изображенное в виде одного функционального блока, может быть представлено как последовательность более простых действий, то есть как следование. Обратно, последовательность функциональных блоков может быть заменена одним функциональным блоком.

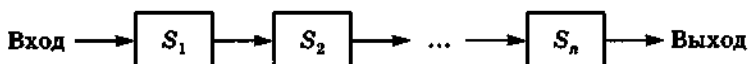


Рис. 1.4. Схема следования

Развилка. Эта структура служит для выбора одной из двух альтернатив, то есть одного из двух возможных функциональных блоков в зависимости от некоторого условия (рис. 1.5). При выполнении этой структуры сначала проверяется значение предиката P. В случае истинного значения

(T) управление передается на выполнение функционального блока S_1. В противном случае (F) управление передается на выполнение функционального блока S_2.

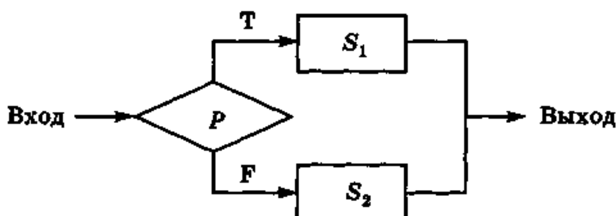


Рис. 1.5. Схема развилки

Повторение. Эта структура используется для многократно повторяющегося действия (рис. 1.6). Исполнение начинается с проверки значения предиката P. Если оно истинно, то управление передается на выполнение функционального блока S, после чего происходит возврат вновь на проверку значения предиката P. Действие выполняется до тех пор, пока значение предиката P не станет ложным. Тогда происходит выход из цикла.

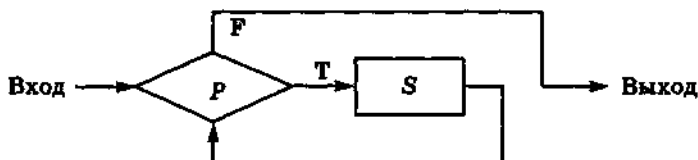


Рис. 1.6. Схема повторения

Каждая из рассмотренных структур удовлетворяет определению функционального блока. В силу этого развилка или повторение может рассматриваться как обобщенный функциональный блок. Таким образом, в структурах «развилка» или «повторение» функциональные блоки сами могут быть конструкциями такого же типа, поэтому возможны вложенные конструкции, например вложенные развилки, вложенные циклы и т. д.

Сделаем важное замечание. Рассмотренную выше структуру повторения называют ЦИКЛ-ПОКА (цикл с предусловием), т. к. условие (предикат) P проверяется до выполнения функционального блока S. На практике часто используются также ЦИКЛ-ДО (цикл с постусловием), в котором

условие проверяется после выполнения функционального блока S (рис. 1.7).

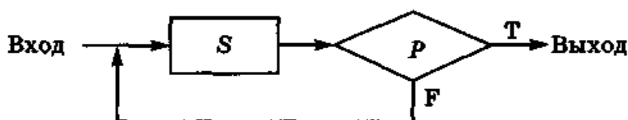


Рис. 1.7. Схема ЦИКЛ-ДО

В этом случае выполнение функционального блока S производится до тех пор, пока предикат P не станет истинным. Структура ЦИКЛ-ДО эквивалентна такому следованию из функционального блока S и ЦИКЛ-ПОКА (рис. 1.8):

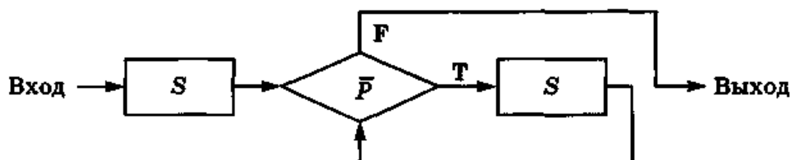


Рис. 1.8. Структура, эквивалентная структуре ЦИКЛ-ДО

Поскольку выход из структуры ЦИКЛ-ПОКА происходит при ложном значении предиката, то необходимо проверить отрицание предиката P, т. е. $\overline{P}$.

В дальнейшем нам необходимо будет показать, что с помощью предлагаемых теоретических построений можно представить все три базовые структуры: следование, развилку, повторение. Тогда по структурной теореме первое положение будет обосновано.

Как уже отмечалось, без точного определения понятия алгоритма мы не можем сказать о задаче, для решения которой нам не известен алгоритм, существует ли он вообще или просто мы его не знаем.

Тестовые задания

1. Множество $\{x \mid f(x) \text{ определено}\}$ — это:
   a) множество значений функции f;
   b) область определения функции f;
   c) ограничение функции f.

2. Если из $x, y \in Dom(f)$ и $x \neq y$ следует $f(x) \neq f(y)$, то функция называется:
   a) инъективной;
   b) сюръективной;
   c) биективной.

3. Функция f из A в B называется сюръективной, если
   a) $Dom(f) = A$;
   b) $Ran(f) = B$;
   c) $Dom(f) = Ran(f)$.

4. Поставить в соответствие. Функция f называется:
   a) инъективной;
   b) сюръективной;

   1) если из $x, y \in Dom(f)$ и $x \neq y$ следует $f(x) \neq f(y)$;
   2) если $Ran(f) = B$;
   3) если $Dom(g) = Ran(f)$ и $g(f(x)) = x$.

5. Множество $f^{-1}(Y) = \{x \mid f(x) \in Y\}$ называется:
   a) прообразом Y относительно f;
   b) ограничением функции f на множестве Y;
   c) продолжением функции f.

6. Функция f из $\mathbb{N}^n$ в $\mathbb{N}$, область определения которой не обязательно совпадает с $\mathbb{N}^n$, называется ...
   a) n-местной функцией;
   b) частичной функцией;
   c) тотальной функцией.

7. Поставить в соответствие. Если для всех $x, y, z \in A$ имеет место:
   a) иррефлексивность;
   b) транзитивность,

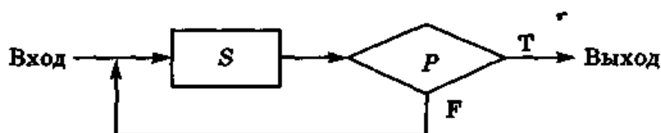
   то R называется:
   1) отношением эквивалентности;
   2) отношением порядка;
   3) частичным порядком;
   4) n-местным отношением.

8. Поставить в соответствие:
   a) следование;
   b) развилка;
   c) повторение;

   1) служит для выбора одной из двух альтернатив;

2) управление передается от одного функционального блока к следующему;
3) управление передается от одного функционального блока к другому и обратно;
4) используется для многократно повторяющегося действия.

9. Какая структура изображена на схеме?



a) цикл-до;
b) цикл-пока;
c) структура, эквивалентная структуре цикл-пока;
d) развилка.

Глава II

Машина Тьюринга

Данная глава посвящена первой формальной математической модели понятия «алгоритм» — машине Тьюринга. В первом пункте вводятся все необходимые определения и показывается, что машина Тьюринга удовлетворяет необходимым свойствам неформального понятия «алгоритм», которые подробно рассмотрены в п. 1.1.1.

Во втором пункте доказывается, что применение основных (в «компьютерном» смысле) операций — композиция, развилка, цикл — над машинами Тьюринга снова приводит к машинам Тьюринга. То есть все, что можно запрограммировать, можно реализовать с помощью машины Тьюринга.

Следующий пункт посвящен универсальной машине Тьюринга, то есть машине, аргументом у которой выступает машина Тьюринга. Таким образом, обоснованна возможность построения универсальной вычислительной машины. Этот строгий математический результат был технически реализован через десять лет после его доказательства.

Сформулированный в четвертом пункте тезис Тьюринга позволяет всюду в дальнейшем заменить нестрогое определение алгоритма из первой главы строгим — машина Тьюринга. Это дает нам возможность уже в следующем пункте привести пример первой алгоритмически неразрешимой проблемы — проблемы остановки. Там же разъясняется важность полученного результата для современной компьютерной практики. При этом необходимо отметить, что получение этого и подобных ему результатов без необходимой математической формализации принципиально невозможно.

В последнем пункте представлено некоторое обобщение машины Тьюринга.

2.1. Основные определения

Машина Тьюринга состоит из:

1) управляющего устройства, которое может находиться в одном из состояний, образующих конечное множество $Q = \{q_1, \ldots, q_n\}$;
2) бесконечной ленты, разбитой на ячейки, в каждой из которых может быть задан один из символов конечного алфавита $A = \{a_1, \ldots, a_m\}$;
3) устройства обращения к ленте, т. е. считывающей и пишущей головки, которая в каждый момент времени обозревает ячейку ленты и в зависимости от символа в этой ячейке и состояния управляющего устройства:
   a) записывает в ячейку символ (быть может, совпадающий с прежним или пустой, т. е. стирает символ);
   b) сдвигается на ячейку влево или вправо или остается на месте;
   c) переходит в новое состояние (или остается в прежнем).

Среди состояний управляющего устройства выделены начальное состояние q_1 и заключительное состояние q_z. В начальном состоянии машина находится перед началом работы, а попав в заключительное состояние q_z, останавливается.

Рассмотрим теперь, как выполняются основные требования a)–e) к алгоритмам из п. 1.1.1 в построенной нами алгоритмической модели — машине Тьюринга.

Данные машины Тьюринга — это слова в алфавите ленты.

Память машины Тьюринга — это конечное множество состояний (внутренняя память) и лента (внешняя память). Лента бесконечна в обе стороны, однако в начальный момент времени только конечное число ячеек ленты заполнено непустыми символами, остальные ячейки пусты, т. е. содержат пустой символ λ (пробел). Из характера работы машины следует, что в любой последующий момент времени лишь конечный отрезок ленты будет заполнен символами.

Детерминированность машины определяется тем, что для любого внутреннего состояния q_i и символа a_j однозначно заданы:

a) следующее состояние q'_i;
b) символ a'_j, который нужно записать вместо a_j в ту же ячейку (стирание символа будем понимать как запись пустого символа λ);

c) направление сдвига головки d_k, обозначаемое одним из трех символов: L (влево), R (вправо), E (на месте).

Любую машину Тьюринга можно описать либо системой правил (команд), имеющих вид

$$q_i a_j \to q'_i a'_j d_k, \tag{1}$$

либо таблицей, строкам которой соответствуют состояния, столбцам — входные символы, а на пересечении строки q_i и столбца a_j записана тройка символов $q'_i a'_j d_k$, либо блок-схемой, которую будем называть диаграммой переходов.

В диаграмме переходов состояниям соответствуют вершины, а правилу вида (1) — ребро, ведущее из q_i в q'_i, на котором написано $a_j \to a'_j d_k$. Условие однозначности требует, чтобы для любого j и любого $i \neq z$ в системе команд имелась только одна команда, аналогичная (1), с левой частью $q_i a_j$. Состояние q_z в левых частях команд не встречается. На диаграмме переходов это выражается условием, что из каждой вершины, кроме q_z, выходит ровно m ребер (m — число символов в алфавите ленты), причем на всех ребрах, выходящих из одной вершины, левые части различны, а в вершине q_z нет выходящих ребер. В дальнейшем будем опускать символы q'_i и a'_j, если $q'_i = q_i$, $a'_j = a_j$.

Приведенное описание также показывает **дискретность** машины Тьюринга.

Элементарные шаги машины — это считывание и запись символов, сдвиг на ячейку влево или вправо, а также переход управляющего устройства в следующее состояние.

Результатом работы машины Тьюринга является слово на ленте после остановки машины. Случай, когда машина не останавливается, будет рассмотрен ниже.

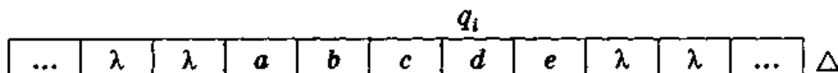
Возможность выбора в качестве начальной системы любого слова в алфавите ленты обеспечивает **массовость** машины Тьюринга.

Таким образом, построенная нами машина Тьюринга — конкретная алгоритмическая модель, претендующая на право формализации понятия «алгоритм».

Полное состояние машины Тьюринга, по которому однозначно можно определить ее дальнейшее поведение, определяется ее внутренним состоянием, состоянием ленты (т. е. словом, записанным на ленте) и положением головки на ленте. Полное состояние будем называть *конфигурацией*, или *машинным словом*, и обозначать тройкой $\alpha_1\ q_i\ \alpha_2$, где

q_i — текущее внутреннее состояние, α_1 — слово слева от головки, а α_2 — слово, образованное символом, обозреваемым головкой, и словом справа от него, причем слева от α_1 и справа от α_2 нет пустых символов.

Пример 1. Конфигурация с внутренним состоянием q_i, в которой на ленте записано *abcde*, а головка обозревает d, запишется как $abcq_ide$.



...	λ	λ	a	b	c	d	e	λ	λ	...

△

Стандартной начальной конфигурацией назовем конфигурацию вида $q_1\alpha$, т. е. конфигурацию, содержащую начальное состояние, в которой головка обозревает крайний левый символ слова, написанного на ленте.

Стандартной заключительной конфигурацией назовем конфигурацию вида $q_z\alpha$.

Ко всякой не заключительной конфигурации K машины T применима ровно одна команда вида (1), которая переводит конфигурацию K в K'. Обозначим это $K \to K'$. Если для K_1 и K_n существует последовательность конфигураций $K_1, K_2, \ldots, K_n$, такая, что $K_1 \to K_2 \to \ldots \to K_n$, то обозначим это $K_1 \Rightarrow K_n$.

Пример 2. Пусть в системе команд машины T имеются команды $q_2a_5 \to q_3a_4R$ и $q_3a_1 \to q_4a_2L$, тогда $q_2a_5a_1a_2 \to a_4q_3a_1a_2 \to q_4a_4a_2a_2$ и, следовательно, $q_2a_5a_1a_2 \Rightarrow q_4a_4a_2a_2$. △

Последовательность конфигураций $K_1 \to K_2 \to K_3 \to K_4 \to \ldots$ однозначно определяется исходной конфигурацией K_1 и полностью описывает работу машины T, начиная с K_1. Она конечна, если в ней встретится заключительная конфигурация, и бесконечна в противном случае.

Пример 3. Машина с алфавитом $A = \{1, \lambda\}$, состояниями $\{q_1, q_2\}$ и системой команд $q_11 \to q_11R$, $q_1\lambda \to q_11R$ при любой начальной конфигурации будет работать бесконечно, заполняя единицами всю ленту вправо от начальной точки. △

Если $\alpha_1q_1\alpha_2 \Rightarrow \beta_1q_z\beta_2$, то будем говорить, что машина T перерабатывает слово $\alpha_1\alpha_2$ в слово $\beta_1\beta_2$, и обозначать это $T(\alpha_1\alpha_2) = \beta_1\beta_2$. Запись $T(\alpha)$ будем употреблять и в смысле обозначения машины T с исходными значениями α.

Уточним, как будут представляться данные. В нашем (простейшем) случае исходные, промежуточные данные и результат будут записываться на ленте в алфавите A^*. Кроме этого, выделяется специальный символ $\lambda \notin A^*$, и тогда $A = A^* \cup \{\lambda\}$.

Запись на ленте словарного вектора $(\alpha_1, ..., \alpha_k)$ назовем *правильной*, если она имеет вид $\alpha_1\lambda\alpha_2\lambda...\alpha_{k-1}\lambda\alpha_k$ либо $\alpha_1*\alpha_2*...*\alpha_k$, где $*$ — символ-разделитель, $* \notin A^*$.

Пусть f — функция, отображающая множество векторов над A^* в себя. Машина T *вычисляет функцию* f, если:

1) для любых векторов V, W, таких, что $f(V) = W$, $q_1V^* \Rightarrow q_zW^*$, где V^*, W^* — правильные записи V и W соответственно;
2) для любого вектора V, такого, что $f(V)$ не определено, машина T, запущенная в стандартной начальной конфигурации q_1V^*, работает бесконечно.

Если для функции f существует машина T, которая ее вычисляет, то f называется *вычислимой по Тьюрингу*.

С другой стороны, всякой вычисляющей машине Тьюринга можно поставить в соответствие вычислимую ею функцию.

Две машины Тьюринга с одинаковым алфавитом A^* будем называть *эквивалентными*, если они вычисляют одну и ту же функцию.

Определения, связанные с вычислением функций, заданных на словарных векторах, имеют в виду переработку нечисловых объектов. Однако нам понадобятся числовые функции, а точнее, функции отображающие $\mathbb{N}$ в $\mathbb{N}$. Договоримся представлять числа в единичном (унарном) коде, т. е. для всех числовых функций $A = \{1, *, \lambda\}$ и число x представляется словом $1...1 = 1^x$, состоящим из x единиц.

Числовая функция $f(x_1, ..., x_n)$ вычислима по Тьюрингу, если существует машина T, такая, что $q_1 1^{x_1} * 1^{x_2} * ... * 1^{x_n} \Rightarrow$ $\Rightarrow q_z 1^y$, когда $f(x_1, ..., x_n) = y$, и работает бесконечно, начиная с $q_1 1^{x_1} * 1^{x_2} * ... * 1^{x_n}$, когда $f(x_1, ..., x_n)$ не определена.

Пример 4. Сложение. Во введенном раннее представлении чисел сложить числа a и b — это значит слово $1^a * 1^b$ переработать в слово 1^{a+b}, т. е. удалить разделитель $*$ и сдвинуть одно из слагаемых, скажем первое, к другому. Это осуществляет машина T_+ с четырьмя состояниями и следую-

щей системой команд (первая команда введена для случая, когда $a = 0$ и исходное слово имеет вид $*1^b$):

$q_1* \to q_z\lambda R$
$q_1 1 \to q_2\lambda R$
$q_2 1 \to q_2 1R$
$q_2* \to q_3 1L$
$q_3 1 \to q_3 1L$
$q_3\lambda \to q_z\lambda R.$

В этой системе команд перечислены не все возможные сочетания состояний машины и символов ленты, опущены те из них, которые при стандартной начальной конфигурации никогда не встретятся. Опускать ненужные команды будем и в дальнейшем, в таблице это будет отмечено прочерками. Диаграмма переходов T_+ приведена на рисунке 2.1; заключительное состояние отмечено двойным кружком. △

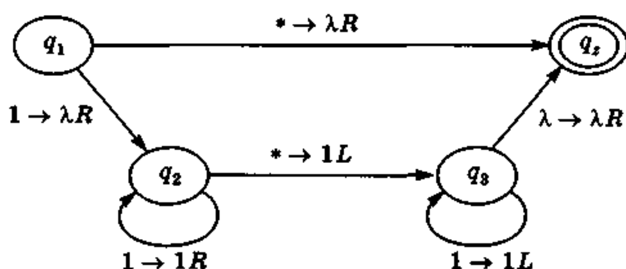


Рис. 2.1. Диаграмма переходов машины T_+

Пример 5. Копирование слова, т. е. переработка слова α в $\alpha * \alpha$. Для чисел эту задачу решает машина $T_{коп}$ с системой команд, представленной в таблице 2.1.

Таблица 2.1

	1	λ	*	0
q_1	$q_2 0R$	$q_z\lambda R$	$q_1 {*} L$	$q_1 1L$
q_2	$q_2 1R$	$q_3 {*} R$	$q_3 {*} R$	—
q_3	$q_3 1R$	$q_4 1L$	—	—
q_4	$q_4 1L$	—	$q_4 {*} L$	$q_1 0R$

При первом проходе в состоянии q_2 ставится маркер (символ *) справа от исходного числа 1^a. Далее при каждом проходе исходного числа машина $T_{коп}$ заменяет левую еди-

ницу числа нулем и пишет (в состоянии q_3) одну единицу справа от 1^a в ближайшую пустую клетку. Таким образом, копия 1^a строится за a проходов. После записи очередной единицы машина переходит в состояние q_4, которое передвигает головку влево до ближайшего нуля, после чего машина переходит в q_1, и цикл повторяется. Он прерывается, когда q_1 обнаруживает на ленте не единицу, а маркер, тогда головка возвращается влево в свое исходное положение, заменяя по дороге все нули единицами. Диаграмма переходов $T_{\text{коп}}$ дана на рисунке 2.2. △

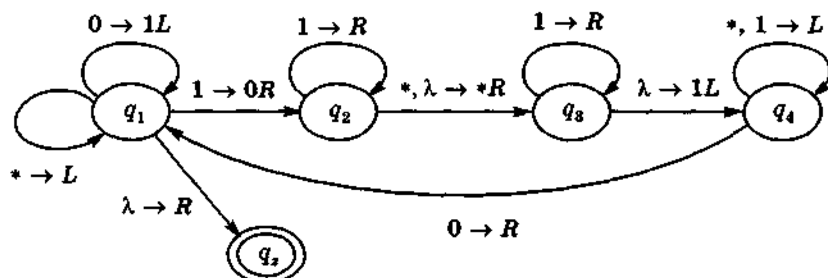


Рис. 2.2. Диаграмма переходов машины $T_{\text{коп}}$

На этой диаграмме, а также на последующих, приняты следующие сокращения:

1) если из q_i в q_j ведут два ребра с одинаковой правой частью, то они объединяются в одно ребро, на котором левые части записаны через запятую;
2) если символ на ленте не изменяется, то он в правой части команды не пишется.

Отметим, что на петле в q_4 использованы одновременно оба сокращения.

2.2. Операции над машинами Тьюринга

Для понимания того, как конкретная машина решает данную задачу, возникает необходимость в содержательных пояснениях, аналогичных приведенным выше для машины $T_{\text{коп}}$. Для того чтобы сделать эти пояснения более формальными и точными, мы будем использовать блок-схемы и некоторые операции над машинами Тьюринга.

Теорема 1. Если функции $f_1(x)$ и $f_2(y)$ вычислимы по Тьюрингу, то их композиция $g(x) = f_2(f_1(x))$ также вычислима по Тьюрингу.

Доказательство. Пусть T_1 — машина, вычисляющая f_1, а T_2 — машина, вычисляющая f_2 и множества их состояний соответственно $Q_1 = \{q_{11}, \ldots, q_{1n_1}\}$ и $Q_2 = \{q_{21}, \ldots, q_{2n_2}\}$. Построим диаграмму переходов машины T из диаграмм T_1 и T_2 следующим образом: отождествим начальную вершину q_{21} диаграммы машины T_2 с конечной вершиной q_{1z} диаграммы машины T_1 (для систем команд это равносильно тому, что систему команд T_2 приписываем к системе команд T_1 и при этом q_{1z} в командах T_1 заменяем на q_{21}). Получим диаграмму с $n_1 + n_2 - 1$ состояниями. Начальным состоянием машины T объявим q_{11}, а заключительным — q_{2z}. Для простоты обозначений будем считать f_1 и f_2 числовыми функциями одной переменной.

Пусть $f_2(f_1(x))$ определена. Тогда $T_1(1^x) = 1^{f_1(x)}$ и $q_{11}1^x \Rightarrow$ $\Rightarrow q_{1z}1^{f_1(x)}$. Машина T пройдет ту же последовательность конфигураций с той разницей, что вместо $q_{1z}1^{f_1(x)}$ она перейдет в $q_{21}1^{f_1(x)}$. Эта конфигурация является стандартной начальной конфигурацией для машины T_2, поэтому $q_{21}1^{f_1(x)} \Rightarrow q_{2z}1^{f_2(f_1(x))}$. Но так как все команды T_2 содержатся в T, то $q_{11}1^x \Rightarrow$ $\Rightarrow q_{21}1^{f_1(x)} \Rightarrow q_{2z}1^{f_2(f_1(x))}$ и, следовательно, $T(1^x) = 1^{f_2(f_1(x))}$.

Если же $f_2(f_1(x))$ не определена, то T_1 или T_2 не остановится, а значит, и машина T не остановится. Следовательно, машина T вычисляет $f_2(f_1(x))$. □

Построенную таким образом машину T будем называть *композицией машин* T_1 и T_2 и обозначать $T_2(T_1)$, а также изображать блок-схемой (рис. 2.3).

Рис. 2.3. Блок-схема композиции $T_2(T_1)$

Эта блок-схема всегда предполагает, что исходными данными машины T_2 являются результаты T_1. При этом они уже «готовы к употреблению», так как благодаря вычислимости (которая существенна при композиции) заключительная конфигурация T_1 легко превращается в стандартную начальную конфигурацию T_2.

Так как машина Тьюринга вектор над A^* воспринимает как слово в алфавите $A^* \cup \{*\}$, определение композиции и теорема 1 остаются в силе, если T_1 и T_2 вычисляют функции от нескольких переменных. Важно лишь, чтобы данные для T_2 были в обусловленном виде подготовлены машиной T_1. Это хорошо видно на следующем примере.

Пример 6. Машина, диаграмма которой приведена на рис. 2.4, — это машина $T_+(T_{\text{коп}})$. Она вычисляет функцию $f(x) = 2x$ для $x \neq 0$. При этом машина $T_{\text{коп}}$ строит двухкомпонентный вектор, а T_+ вычисляет функцию от двух переменных. △

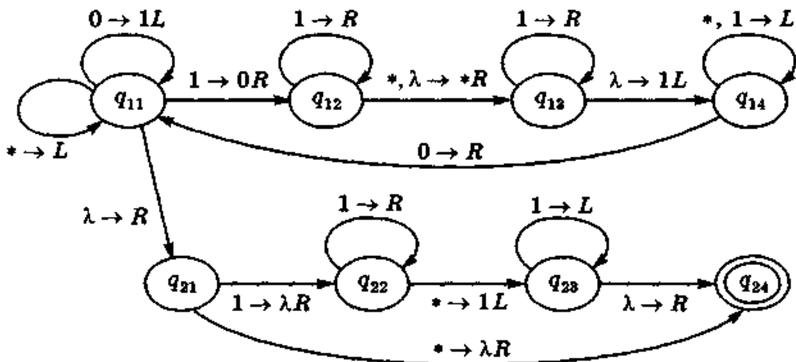


Рис. 2.4. Диаграмма переходов машины $T_+(T_{\text{коп}})$

Для удобства последующих построений установим следующий важный факт.

Теорема 2. Любая функция, вычислимая по Тьюрингу, вычислима на машине Тьюринга с правой полулентой. Иначе говоря, для любой машины Тьюринга T существует эквивалентная ей машина T_R с правой полулентой (аналогичная теорема формулируется для левой полуленты).

Доказательство можно найти, например, в [16].

Рассмотрим теперь вычисление предикатов на машинах Тьюринга.

Говорят, что машина T *вычисляет предикат* $P(\alpha)$ (α — слово в A^*), если $T(\alpha) = \omega$, где $\omega = \text{T}$, когда $P(\alpha)$ истинно, и $\omega = \text{F}$, когда $P(\alpha)$ ложно. Если $P(\alpha)$ не определен, то машина T не останавливается.

При обычном вычислении предиката уничтожается α, что не всегда удобно, особенно если после T должна работать

другая машина. Поэтому введем понятие вычисления с восстановлением.

Говорят, что машина T *вычисляет* $P(\alpha)$ *с восстановлением*, если $T(\alpha) = \omega\alpha$.

Лемма. Если существует машина T, вычисляющая $P(\alpha)$, то существует и $\tilde{T}$, вычисляющая $P(\alpha)$ с восстановлением.

Доказательство. Вычисление с восстановлением можно представить следующей последовательностью конфигураций:

$$q_1\alpha \Rightarrow q_{n_1}\alpha*\alpha \Rightarrow \alpha*q_{n_2}\alpha \Rightarrow \alpha*q_{n_3}\omega \Rightarrow q_{n_4}\omega\alpha.$$

Первая часть этой последовательности реализуется машиной $T_{\text{коп}}$; вторая — простым сдвигом головки до маркера *; третья — машиной T_R, вычисляющей $P(\alpha)$ на правой полуленте (одного копирования мало для восстановления, так как копию может испортить основная машина T, нужна машина, не заходящая влево от α); четвертая — переносом ω в крайнее левое положение. Машина $\tilde{T}$ является композицией указанных четырех машин. □

Пример 7. Машина с диаграммой на рис. 2.5 вычисляет предикат «α — четное число»: головка достигает конца числа в состоянии q_2, если число единиц четно, и в состоянии q_3, если число единиц нечетно, после чего она перемещается в исходное положение в состоянии q_4 либо q_5 и печатает T или F соответственно.

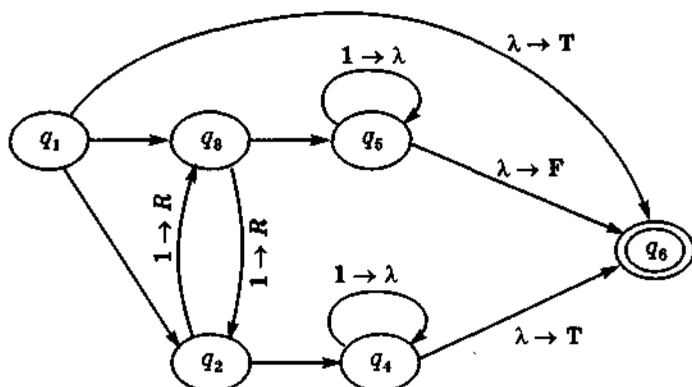


Рис. 2.5. Диаграмма переходов машины «α — четное число»

Для того чтобы этот предикат вычислялся с восстановлением, достаточно в петлях q_4 и q_5 не стирать, а сохранять единицы, то есть заменить команды $1 \to \lambda\, L$ на команды $1 \to L$. △

Так как машина Тьюринга с алфавитом $A^* = \{\text{T}, \text{F}\}$ и командами $q_1\text{T} \to q_z\text{F}E$ и $q_1\text{F} \to q_z\text{T}E$ вычисляет отрицание логической переменной, то из вычислимости всюду определенного предиката $P(\alpha)$ следует вычислимость $\overline{P}(\alpha)$.

Пусть функция $f(\alpha)$ задана описанием: «если $P(\alpha)$ истинно, то $f(\alpha) = g_1(\alpha)$, иначе $f(\alpha) = g_2(\alpha)$». (Под «иначе» имеется в виду «если $P(\alpha)$ ложно»; если $P(\alpha)$ не определено, то $f(\alpha)$ также не определено.) Функция $f(\alpha)$ называется *развилкой* или *условным переходом* к $g_1(\alpha)$ и $g_2(\alpha)$ по условию $P(\alpha)$.

Теорема 3. Если функции $g_1(\alpha)$, $g_2(\alpha)$ и предикат $P(\alpha)$ вычислимы по Тьюрингу, то развилка $g_1(\alpha)$ и $g_2(\alpha)$ по $P(\alpha)$ также вычислима.

Доказательство. Пусть T_1 — машина с состояниями $q_{11}, q_{12}, \ldots, q_{1n_1}$ и системой команд Σ_1, вычисляющая g_1, T_2 — машина с состояниями $q_{21}, q_{22}, \ldots, q_{2n_2}$ и системой команд Σ_2, вычисляющая g_2; T_p вычисляет с восстановлением $P(\alpha)$. Тогда машина T, вычисляющая развилку g_1 и g_2 по P, — это композиция T_p и машины T_3, система команд Σ_3 которой имеет следующий вид:

$$\Sigma_3 = \Sigma_1 \cup \Sigma_2 \cup \{q_{31}\text{T} \to q_{11}\lambda R, q_{31}\text{F} \to q_{21}\lambda R, q_{1z} \to q_{2z}E\}.$$

Первые две из новых команд передают управление системе команд Σ_1 или Σ_2 в зависимости от значения предиката $P(\alpha)$. Третья команда введена для того, чтобы T_3 имела одно заключительное состояние q_{2z}. Отсутствие символа ленты в этой команде означает, что она выполняется при любом символе. □

Построенную в теореме 3 машину T будем называть развилкой машин T_1 и T_2 по условию T_p и обозначать $T_{\text{раз}}(T_1, T_2, T_p)$, а также изображать блок-схемой (рис. 2.6).

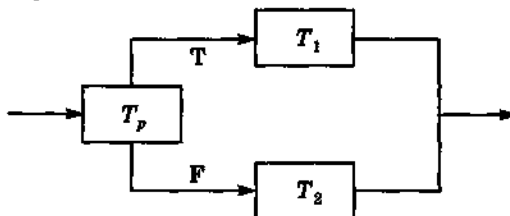


Рис. 2.6. Блок-схема развилки машин $T_{\text{раз}}(T_1, T_2, T_p)$

Пусть функция $f(\alpha)$ задана описанием: «пока истинно $P(\alpha)$, вычислять $g_1(\alpha)$, иначе $f(\alpha) = g_2(\alpha)$». Функция $f(\alpha)$ называется *повторением* или *циклом* от $g_1(\alpha)$ и $g_2(\alpha)$ по условию $P(\alpha)$.

Теорема 4. Если функции $g_1(\alpha)$, $g_2(\alpha)$ и предикат $P(\alpha)$ вычислимы по Тьюрингу, то цикл $g_1(\alpha)$ и $g_2(\alpha)$ по $P(\alpha)$ также вычислим.

Доказательство. Сохраняя обозначения из доказательства предыдущей теоремы, введем машину T, вычисляющую цикл g_1 и g_2 по P, — это композиция $T_{раз}$ и машины T_3, система команд Σ_3 которой имеет следующий вид:

$$\Sigma_3 = \Sigma_1 \cup \Sigma_2 \cup \{q_{31}\text{T} \to q_{11}\lambda R,\ q_{31}\text{F} \to q_{21}\lambda R,\ q_{1z} \to q_{p1}E\}.$$

Первые две из новых команд передают управление системам команд Σ_1 или Σ_2 в зависимости от значений предиката $P(\alpha)$. Третья команда отождествляет начальную вершину q_{p1} диаграммы машины $T_{раз}$ с конечной вершиной q_{1z} диаграммы машины T_1 (то есть после вычисления $g_1(\alpha)$ мы переходим к новой проверке условия $P(\alpha)$). Конечное состояние машины T есть состояние q_{2z}. □

Построенную в теореме 4 машину T будем называть циклом машин T_1 и T_2 по условию T_p и обозначать $T_{цикл}(T_1, T_2, T_p)$, а также изображать блок-схемой (рис. 2.7).

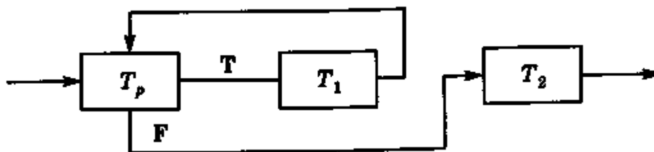


Рис. 2.7. Блок-схема цикла машин $T_{цикл}(T_1, T_2, T_p)$

Пример 8. В примере 4 было описано построение машины T_+ для сложения двух чисел. На рис. 2.8 приведена диаграмма машины T_{++} для сложения n чисел ($n = 1, 2, \ldots$). Цикл из состояний q_1, q_2, q_3 — это немного модифицированная и «зацикленная» машина T_+, в которой заключительное состояние совмещено с начальным. Сумма, полученная в очередной итерации цикла, является первым слагаемым следующего цикла. Состояние q_4 реализует развилку. В нем проверяется условие — есть ли второе слагаемое. Если да (о чем говорит наличие маркера *), то происходит переход

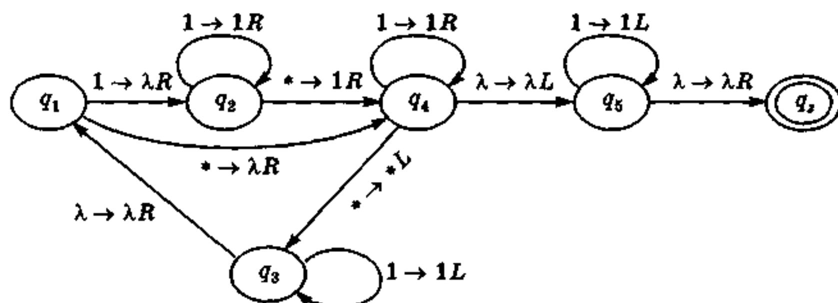


Рис. 2.8. Диаграмма переходов машины T_{++}

к следующему циклу; если нет (о чем говорит λ после единиц), то машина выходит из цикла. △

Благодаря вычислимости композиции, развилки и цикла, словесные описания и язык блок-схем можно сделать достаточно точным языком описания работы машин Тьюринга. Каждый блок — это множество состояний, в котором выделены начальное и заключительное состояния и система команд. Переход к блоку — это обязательно переход в его начальное состояние. Машина Тьюринга, описываемая блок-схемой, — это объединение состояний и команд, содержащихся во всех блоках. В частности, блоком может быть одно состояние. Блоки, вычисляющие предикаты, обозначим буквой P.

Пример 9. На рис. 2.9 приведена блок-схема машины Тьюринга T_*, осуществляющая умножение двух чисел: $T_*(1^a * 1^b) = 1^{ab}$. Ее заключительное состояние — q_{03}. Блоки реализуют следующие указания и вычисления:

P_1 — вычислить с восстановлением предикат «оба сомножителя больше нуля»;

T_1 — стереть все непустые символы справа;

T_2 — установить головку у ячейки, следующей (вправо) за маркером *, маркер * заменить на ~;

$T_{коп}$ — см. пример 5, в котором команду $q_1\lambda \to q_z R$ надо заменить на команду $q_1{\sim} \to q_z R$; эта машина $(a - 1)$ раз копирует 1^b; после i-го цикла она останавливается в конфигурации $1^{a-i-1}\,({\sim}1^b)^{i-1}{\sim}\,q_1 1^b * 1^b$;

T_3 — вернуть головку к крайнему слева непустому символу. После $(a - 1)$-го цикла этим символом окажется ~ (или *, если $a = 1$), и тогда происходит выход из цикла и переход к $\tilde{T}_{++}$;

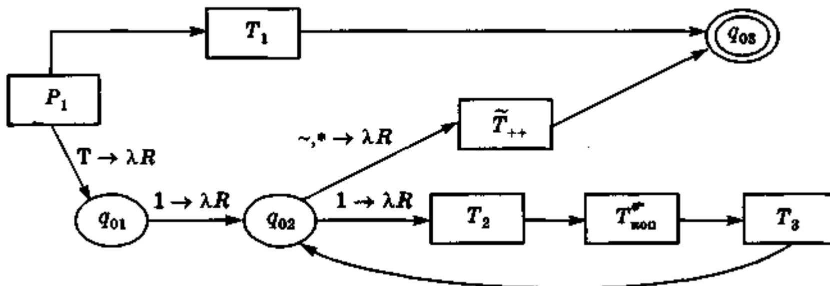


Рис. 2.9. Блок-схема машины *T*.

$\widetilde{T}_{++}$ работает как T_{++} с той разницей, что числа, которые она суммирует, разделены двумя видами маркеров: ~ и *. Для этой цели ко всем командам с маркером * в левой части добавлены такие же команды для маркера ~. △

Другие важные примеры читатель может найти в [14, 16].

2.3. Универсальная машина Тьюринга

Систему команд машины Тьюринга можно интерпретировать и как описание работы конкретного механизма, и как программу, т. е. совокупность предписаний, однозначно приводящих к результату.

При разборе примеров мы невольно принимаем вторую интерпретацию, выступая в роли механизма, который способен воспроизвести работу любой машины Тьюринга. Эта способность связана с существованием алгоритма воспроизведения работы машины Тьюринга по заданной программе, то есть системе команд. Словесное описание такого алгоритма дать нетрудно, и его основное циклически повторяющееся действие состоит в следующем.

Для текущей конфигурации $\alpha_1 a_k q_i a_j \alpha_2$ найти в системе команд команду с левой частью $q_i a_j$. Если правая часть этой команды имеет вид $q'_i a'_j R$, то заменить в текущей конфигурации $a_k q_i a_j$ на $a_k a'_j q'_j$, если правая часть имеет вид $q'_i a'_j L$, то заменить $a_k q_i a_j$ на $q'_i a_k a'_j$, если же правая часть имеет вид $q'_i a'_j E$, то заменить $a_k q_i a_j$ на $a_k q'_i a'_j$.

Как уже говорилось в первой главе, словесное описание алгоритма может быть неточным и нуждаться в формализации. Так как в качестве такой формализации понятия алгоритма сейчас обсуждается машина Тьюринга, то естественно поставить задачу построения машины Тьюринга, реализующей описанный алгоритм воспроизведения.

Для машин Тьюринга, вычисляющих функции от одной переменной, формулировка этой задачи такова: построить машину Тьюринга U, вычисляющую функцию от двух переменных и такую, что для любой машины T с системой команд Σ_T $U(\Sigma_T, \alpha) = T(\alpha)$, если $T(\alpha)$ определена (или останавливается), и $U(\Sigma_T, \alpha)$ не останавливается, если $T(\alpha)$ не останавливается.

Любую машину U, обладающую указанным свойством, будем называть *универсальной машиной Тьюринга.*

Нетрудно обобщить эту формулировку для любого числа переменных.

Теорема 5. Универсальная машина Тьюринга существует.

Доказательство этой теоремы мы приводить не будем, но отметим, что имеется ее конструктивное доказательство (см., например, [8]), в котором описывается построение искомой универсальной машины Тьюринга.

Отметим следующий важный для нас момент. При построении машин Тьюринга в примерах (как, впрочем, и в рекомендуемом доказательстве) мы не жалели ни символов ленты, ни состояний, стремясь к наглядности построений. Нетрудно показать, а программист, привыкший к двоичному кодированию, легко в это поверит, что можно построить машину Тьюринга U всего с двумя символами на ленте. К. Шеннон построил универсальную машину всего с двумя состояниями, что совсем не очевидно. В то же время показано (Б. Боброу и М. Минский), что универсальная машина с двумя состояниями и двумя символами невозможна. Вообще в определенных пределах уменьшение числа символов U ведет к увеличению числа состояний, и наоборот (сводка результатов о минимальных универсальных машинах Тьюринга приведена в [14]).

Существование универсальной машины Тьюринга означает, что систему команд Σ_T любой машины T можно интерпретировать двояко, а именно как:

- описание работы конкретного устройства машины T;
- программу для универсальной машины U.

Для современного информатика это обстоятельство вполне естественно: любой алгоритм может быть реализован либо аппаратно (построением соответствующей схемы), либо программно (написание программы для универсального компьютера). Однако важно сознавать, что сама идея универсального алгоритмического устройства совершенно не связана с развитием современных технических средств его реализации (лампы, полупроводники, БИС и т. д.) и является не техническим, а математическим фактом, который описывается в абстрактных математических терминах, не зависящих от технических средств, и к тому же опирающимся на крайне малое количество весьма элементарных исходных понятий. Характерно, что основополагающие работы по теории алгоритмов (в частности, работы Тьюринга) появились в 30-х годах XX века, до создания первых ЭВМ.

Указанная двоякая интерпретация сохраняет на абстрактном уровне основные плюсы и минусы двух вариантов инженерной реализации. Конкретная машина T работает гораздо быстрее; кроме того, управляющее устройство машины U довольно громоздко, т. е. велико число состояний и команд. Однако его сложность постоянна, и, будучи раз построено, оно годится для реализации сколь угодно больших алгоритмов, понадобится лишь большой объем ленты, которую естественно считать более дешевой и более просто устроенной, чем управляющее устройство. Кроме того, при смене алгоритма не понадобится строить новых устройств, нужно лишь написать новую программу.

2.4. Тезис Тьюринга

До сих пор нам удавалось для всех процедур, претендующих на алгоритмичность, то есть конструктивных процедур, строить реализующие их машины Тьюринга. Более того, из теорем 1, 3, 4 (п. 2.2) и теоремы Бома–Джакопини (п. 1.3) следует, что любая программа (в смысле программы для компьютера) может быть реализована с помощью машины Тьюринга.

Будет ли это удаваться всегда? Утвердительный ответ на этот вопрос содержится в тезисе Тьюринга, который формулируется так: ***всякий алгоритм может быть реализован машиной Тьюринга.***

Доказать тезис Тьюринга нельзя, поскольку само понятие алгоритма (или эффективной процедуры) является не-

точным. Это не теорема и не постулат математической теории, а утверждение, которое связывает теорию с теми объектами, для описания которых она создана.

По своему характеру тезис Тьюринга напоминает гипотезы физики об адекватности математических моделей физическим явлениям и процессам. Подтверждением тезиса Тьюринга является, во-первых, математическая практика, а во-вторых, то обстоятельство (уже отмечавшееся в п. 1.1.3), что описание алгоритма в терминах любой другой известной алгоритмической модели может быть сведено к его описанию в виде машины Тьюринга.

Тезис Тьюринга позволяет, с одной стороны, заменить неточное утверждение о существовании эффективных процедур (алгоритмов) точными утверждениями о существовании машин Тьюринга, а с другой стороны, утверждение о несуществовании машины Тьюринга для решения какой-то проблемы истолковать как утверждение о несуществовании алгоритма ее решения вообще.

2.5. Проблема остановки

В числе общих требований, предъявляемых к алгоритмам (п. 1.1.1), есть и требование результативности. Его можно сформулировать так: нужно построить алгоритм B, такой, что для любого алгоритма A $B(A, \alpha) = \mathrm{T}$, если $A(\alpha)$ дает результат, и $B(A, \alpha) = \mathrm{F}$, если $A(\alpha)$ не дает результат.

Эта задача называется **проблемой остановки**.

Теорема 6. Не существует машины Тьюринга T_0, решающей проблему остановки для произвольной машины Тьюринга T.

Доказательство. Предположим, что машина T_0 существует. Для определенности будем считать, что маркером между Σ_T и α на ленте машины T_0 служит $*$. Построим машину $T_1(\Sigma_T) = T_0(T_{\text{коп}}(\Sigma_T))$. Исходными данными машины T_1 является система команд любой машины T. Запись Σ_T на ленте машина T_1 преобразует в $\Sigma_T * \Sigma_T$, а затем работает как машина T_0. Таким образом, T_1 также решает проблему остановки для любой машины T, но только в том случае, когда на ленте T в качестве данных α написана ее собственная система команд Σ_T. Иначе говоря, $T_1(\Sigma_T) = \mathrm{T}$, если машина $T(\Sigma_T)$ останавливается, и $T_1(\Sigma_T) = \mathrm{F}$, если машина $T(\Sigma_T)$ не останавливается.

Пусть q_{1n} — заключительное состояние T_1. Добавим к системе команд T_1 одно состояние $q_{1,n+1}$, объявив его заключительным, и m команд (m — число символов T_1) q_{1n} F $\to$ $q_{1,n+1}$ E, q_{1n} $a_j \to q_{1n}$ R для любого a_j, кроме F. Получим машину T_1', которая останавливается, если T не останавливается, и не останавливается, если T останавливается.

Запишем теперь на ленте T_1' ее собственную систему команд $\Sigma_{T_1'}$. Когда T_1' останавливается, она не останавливается, и не останавливается, если она останавливается. Очевидно, такая машина T_1' невозможна. □

В силу тезиса Тьюринга невозможность построения машины Тьюринга означаем отсутствие алгоритма решения данной проблемы. Поэтому полученная теорема дает первый пример алгоритмически неразрешимой проблемы, а именно, алгоритмически неразрешимой оказывается проблема остановки для машины Тьюринга, то есть проблема определения результативности алгоритма.

В частности, полученный результат показывает невозможность создания программы для компьютера, решающей указанную проблему. Такой результат принципиально невозможно получить, оставаясь в рамках программирования для компьютера (п. 1.3), о чем мы уже говорили в первой главе.

При истолковании утверждений, связанных с алгоритмической неразрешимостью, следует иметь в виду следующее важное обстоятельство. В таких утверждениях речь идет об отсутствии единого алгоритма, решающего данную проблему; при этом вовсе не исключается возможность решения этой проблемы в частных случаях, но различными средствами для каждого случая. В частности, теорема не исключает того, что для отдельных классов машин Тьюринга проблема остановки может быть решена. Например, она решается для всех машин, приведенных в примерах п. 2.2. Однако существуют конкретные машины Тьюринга (например, любая универсальная машина) с неразрешимой проблемой остановки. Поэтому неразрешимость общей проблемы остановки вовсе не снимает необходимость доказывать сходимость предлагаемых алгоритмов, а лишь показывает, что поиск таких доказательств нельзя полностью автоматизировать.

Неразрешимость проблемы остановки можно интерпретировать как несуществование общего алгоритма для отладки программ, точнее, алгоритма, который по тексту любой программы и ее входным данным определял бы, зацик-

лится ли программа на этих данных или нет. Если учесть сделанное ранее замечание, такая интерпретация не противоречит тому эмпирическому факту, что большинство программ в конце концов удается отладить, то есть установить наличие зацикливания, найти и устранить его причину. При этом решающую роль играют опыт и искусство программиста.

2.6. Машина фон Неймана

Рассмотрим одно обобщение машины Тьюринга — машину фон Неймана.

По принципу обработки информации это вычислительное устройство существенно отличается от машины Тьюринга.

Как мы знаем на каждом такте в машине Тьюринга происходит преобразование информации только в одной ячейке, на которую в этот момент указывает головка, остальные ячейки «бездействуют», хотя часто имеется возможность работать параллельно. Простейшее решение в этом случае — использование нескольких машин Тьюринга с общей для них внешней памятью (лентой). Но такой подход не всегда допустим — при обращении к одной и той же ячейке памяти могут возникнуть конфликты.

В машине фон Неймана число одновременно обрабатываемых ячеек может неограниченно расти, оставаясь в каждый момент времени конечным.

Элемент Неймана (ЭН) — это устройство, которое на каждом такте пребывает в одном из конечного числа состояний $r_i \in R$, образующих его алфавит. ЭН имеет два входных канала (левый и правый); по каждому из них на такте t также поступает по одному состоянию из R (рис. 2.10).

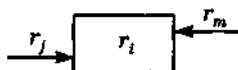


Рис. 2.10. Элемент Неймана

Элемент реализует функцию $r_i^{t+1} = \Psi(r_i^t, r_j^t, r_m^t)$, то есть в такте $t + 1$ переходит в состояние r_i^{t+1}, определяемое его состоянием в текущий момент времени t и значениями, поступившими по входным каналам.

Состояния элементов Неймана в момент времени t определяют *конфигурацию* машины Неймана (рис. 2.11) в момент t: $k(t)$.

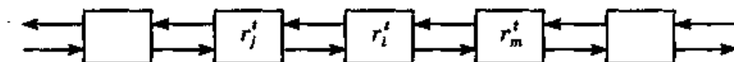


Рис. 2.11. Структура машины Неймана

Функционирование машины Неймана — это переход от конфигурации $k(t)$ к конфигурациям $k(t + 1)$, $k(t + 2)$, ...

За один такт свое состояние может менять большое число элементов Неймана, что фактически приводит к параллельной обработке информации.

Таким образом, машина фон Неймана может служить теоретической моделью систем параллельных вычислений.

Упражнения

1. Дан алфавит $A = \{1, \lambda\}$ и состояния $\{q_1, q_z\}$. Построить зацикливающуюся машину Тьюринга и ее диаграмму.
2. Построить диаграмму переходов машины Тьюринга T_+ для сложения двух заданных натуральных чисел, $A = \{1, \lambda\}$.
3. Построить машину Тьюринга для получения следующего натурального числа в унарной системе, т. е. вычислить функцию $f(x) = x + 1$, $A = \{1, \lambda\}$.
4. Имеется машина Тьюринга с алфавитом $A = \{1, \lambda\}$ и внутренними состояниями $Q = \{q_1, q_z\}$ со следующей системой команд:

A \ Q	q_1
λ	$q_z 1$
1	$q_1 1 R$

Определить, в какое слово переработает машина каждое из следующих слов, если она находится в начальном состоянии q_1 и обозревает указанную ячейку:

a) $1\lambda11\lambda\lambda11$ (обозревается ячейка 4, считая слева);

b) $11\lambda111\lambda1$ (обозревается ячейка 2);

c) $1\lambda\lambda111$ (обозревается ячейка 3).

Ответ изобразить схематически в виде последовательности конфигураций, возникающих на ленте на каждом такте работы машины.

5. Имеется машина Тьюринга с алфавитом $A = \{0, 1, \lambda\}$ и внутренними состояниями $Q = \{q_1, q_2, q_z\}$ со следующей системой команд:

A \ Q	q_1	q_2
λ	q_z1	$q_1\lambda L$
1	q_20R	q_11R
0	q_11R	q_20L

Определить, в какое слово переработает машина слово 1101λ01, если она находится в начальном состоянии q_1 и обозревает 4-ую ячейку слева. Ответ изобразить схематически в виде последовательности конфигураций, возникающих на ленте на каждом такте работы машины.

6. Дано слово *bcadc* в алфавите $A = \{a, b, c, d, \lambda\}$. Построить систему команд машины Тьюринга T, преобразующей данное слово в слово *bcdcc*.

7. Построить диаграмму переходов машины T из упражнения 6.

8. По диаграмме (рис. 2.8) построить систему команд машины Тьюринга T_{++} для суммирования n натуральных чисел.

9. Построить машину Тьюринга T_{++} для суммирования n натуральных чисел другим способом.

10. Построить диаграмму переходов машины Тьюринга T_{-}, которая вычисляет разность двух натуральных чисел.

11. Построить машину Тьюринга, которая правильно вычисляет функцию $f(x) = x + 1$ в десятичной системе счисления.

12. Построить машину Тьюринга, выполняющую правый сдвиг (все символы сдвигаются на одну ячейку вправо): $q_1 01^x 0 \underset{B^+}{\Rightarrow} 01^x q_z 0$.

13. Построить машину Тьюринга, выполняющую транспозицию двух чисел: $01^x q_1 01^y 0 \underset{B}{\Rightarrow} 01^y q_z 01^x 0$.

14. Дан алфавит $A = \{+, -, 0, 1, \lambda\}$ и внутренние состояния $Q = \{q_1, q_2, \ldots, q_z\}$. Построить машину Тьюринга, проверяющую, является ли исходная последовательность арифметическим выражением.

15. Дан алфавит $A = \{a, b, c, \ldots, z, \lambda\}$ и внутренние состояния $Q = \{q_1, q_2, \ldots, q_z\}$. Построить машину Тьюринга для циклического сдвига последнего элемента, например, если дана последовательность *dcba* получить *adcb*.

16. Дан алфавит $A = \{(,), \lambda\}$ и внутренние состояния $Q = \{q_1, q_2, \ldots, q_z\}$. Построить машину Тьюринга, проверяющую верно ли в исходной последовательности расставлены скобки.

17. Дан алфавит $A = \{a, b, c, \ldots, z, \lambda\}$ и внутренние состояния $Q = \{q_1, q_2, \ldots, q_z\}$. Построить машину Тьюринга для транспозиции элементов, например, если дана последовательность *abcd*, то заменить ее на *dcba*.

18. Построить машину Тьюринга, которая вычитает единицу из числа в десятичной системе счисления.

19. Построить машину Тьюринга, вычисляющую следующую функцию:
$f(x_1, x_2, \ldots, x_n) = x_m, (m \le n)$.

20. Дана конечная последовательность единиц, вписанных в ячейки без пропусков. Построить машину Тьюринга, которая записывала бы в десятичной системе счисления количество этих единиц, т. е. пересчитывала набор этих единиц.

21. Построить машину Тьюринга, которая выполняет умножение двух чисел в унарной системе счисления.

22. Даны два набора единиц, разделенные *. Построить машину Тьюринга, которая выбирала бы больший из этих наборов, а меньший стирала.

23. Построить машину Тьюринга, которая выполняет деление на два в унарной системе счисления.

24. Дан алфавит $A = \{a, b, c, \ldots, z, \lambda\}$ и внутренние состояния $Q = \{q_1, q_2, \ldots, q_z\}$. Построить машину Тьюринга, которая подсчитывает количество букв *a* в заданной последовательности.

Тестовые задания

1. Машина Тьюринга — это:
   a) ее полное состояние;
   b) устройство, представленное в виде бесконечной ленты, управляющего устройства и головки;
   c) набор команд, определяющих ее состояние в каждый конкретный момент.
2. Система правил $q_i a_j \to q'_i a'_j d_k$ описывает:
   a) конфигурацию;
   b) полное состояние;
   c) последовательность шагов.
3. Натуральные числа в машине Тьюринга представляются:
   a) в унарном коде;
   b) в двоичном коде;
   c) в троичном коде.
4. Можно ли построить универсальную машину Тьюринга?
   a) да;
   b) нет.
5. Можно ли построить универсальную машину Тьюринга с двумя символами на ленте и двумя состояниями?
   a) да;
   b) нет.
6. Смысл проблемы остановки (с точки зрения программирования) заключается в следующем:
   a) существует алгоритм воспроизведения работы по заданному алгоритму;
   b) не существует алгоритма, который бы по номеру алгоритма определял результат;
   c) не существует общего алгоритма для отладки программ;
   d) нет верного ответа.
7. Какой алгоритм реализован следующей машиной Тьюринга (в унарной системе)?

   $q_1* \to q_z*E$
   $q_1 1 \to q_2 \lambda R$
   $q_2 1 \to q_2 1R$
   $q_2* \to q_3 1L$
   $q_3 1 \to q_3 1L$
   $q_3 \lambda \to q_z \lambda R$

   a) вычисление функции $f(x) = x + 1$;
   b) вычитание двух чисел;

c) сложение двух чисел;
d) вычисление функции $f(x) = x - 1$.

8. Какая из машин Тьюринга вычисляет функцию $f(x) = x + 1$ (в унарной системе)?
a) $q_1 1 \to q_z \lambda R$
b) $q_1 1 \to q_1 1L,\ q_1 \lambda \to q_z 1E$
c) $q_1 1 \to q_2 1L,\ q_2 \lambda \to q_1 1E$

9. Внутренняя память машины Тьюринга — это:
a) лента;
b) конечное множество состояний;
c) нет верного ответа.

10. Внешняя память машины Тьюринга — это:
a) лента;
b) конечное множество состояний;
c) нет верного ответа.

11. Машинное слово, обозначаемое $\alpha_1 q_i \alpha_2$, называется:
a) системой команд;
b) конфигурацией или полным состоянием;
c) нет верного ответа.

12. Поставить в соответствие:
a) a_j;
b) q_i;
c) d_k;
d) q_1;
e) q_z;
1) направление сдвига головки;
2) машинное слово;
3) состояние управляющего устройства;
4) символ из алфавита A;
5) конечное состояние управляющего устройства;
6) начальное состояние управляющего устройства;
7) текущая конфигурация.

13. Что получится в результате выполнения следующей последовательности команд?
$q_1 1 \to q_1 1R$
$q_1 \lambda \to q_1 1R$
a) машина остановится, поставив вторую 1;
b) машина остановится, все стирая;
c) машина не остановится, заполняя ленту 1.

14. В текущей конфигурации $\alpha_1 a_k q_i a_j \alpha_2$ выполнена команда $q_i a_j \to q'_i a'_j L$. Какая конфигурация получится в результате выполнения данной команды?

a) $\alpha_1 a_k a'_j q'_i \alpha_2$;
b) $\alpha_1 q'_i a_k a'_j \alpha_2$;
c) $\alpha_1 a_k q'_i a'_j \alpha_2$.

15. Функция f называется вычислимой по Тьюрингу, если:
   a) для f существует правильная система команд;
   b) для f существует машина T, которая ее вычисляет;
   c) нет верного ответа.

16. Поставить в соответствие:
   a) слова в алфавите ленты;
   b) конечное множество состояний и лента;
   c) считывание и запись символов, сдвиг на ячейку влево или вправо, а также переход управляющего устройства в следующее состояние;
   1) детерминированность машины;
   2) данные машины Тьюринга;
   3) память машины Тьюринга;
   4) элементарные шаги машины.

17. Что является исходными данными машины T_2 для композиции машин $T_2(T_1)$?
   a) система команд машины T_1;
   b) система команд машины T_2;
   c) результат работы машины T_1;
   d) результат работы машины T_2.

18. Числовая функция $f(x_1, \ldots, x_n)$ вычислима по Тьюрингу, если существует машина T, такая, что
   $q_1 1^{x_1} * 1^{x_2} * \ldots * 1^{x_n} \Rightarrow q_z 1^y$, когда
   a) $f(x_1, \ldots, x_n)$ не определена;
   b) $f(x_1, \ldots, x_n) = y$;
   c) $f(x_1, \ldots, x_n) = 0$.

19. Пусть функция $f(\alpha)$ задана описанием: «если $P(\alpha)$ истинно, то $f(\alpha) = g_1(\alpha)$, иначе $f(\alpha) = g_2(\alpha)$». Тогда $f(\alpha)$ называется:
   a) композицией;
   b) развилкой;
   c) повторением.

20. Пусть функция $f(\alpha)$ задана описанием: «пока истинно $P(\alpha)$, вычислять $g_1(\alpha)$, иначе $f(\alpha) = g_2(\alpha)$». Тогда $f(\alpha)$ называется:
   a) композицией;
   b) развилкой;
   c) повторением.

Глава III

Рекурсивные функции

Данная глава посвящена исторически первой алгоритмической модели — рекурсивным функциям. В первом пункте дается конструктивное определение примитивно-рекурсивных функций, приводятся многочисленные примеры. Здесь же вводятся примитивно-рекурсивные отношения и предикаты.

Во втором пункте рассматриваются примитивно-рекурсивные операторы и дается определение оператора минимизации, который позволяет, в частности, вводить обратные функции.

В третьем пункте строится вычислимая функция, которая, тем не менее, не является примитивно-рекурсивной — функция Аккермана. Возникает необходимость расширения средств построения вычислимых функций.

Поэтому в четвертом пункте даются определения частично-рекурсивных и общерекурсивных функций. Формулируется тезис Чёрча (аналог тезиса Тьюринга) и показывается, что любая программа для компьютера может быть реализована частично-рекурсивной функцией.

3.1. Примитивно-рекурсивные функции

В первой главе мы ввели понятие эффективно или алгоритмически вычислимой функции. Возникает вопрос: какие функции алгоритмически вычислимы? Как описать такие алгоритмически вычислимые функции?

Исследование этих вопросов привело к созданию в 30-х годах прошлого века теории рекурсивных функций. В этой теории, как и вообще в теории алгоритмов, принят конструктивный (финитный) подход, основной чертой которого является то, что все множество исследуемых объектов (в данном случае функций) строится из некоторого базиса

с помощью простых операций, эффективная вычислимость которых достаточно очевидна. Операции над функциями будем в дальнейшем называть операторами.

В базис включим:

- константу 0;
- функцию следования $x' = x + 1$;
- функцию проекции $U_m^n(x_1, x_2, \dots, x_n) = x_m \quad (m \le n)$.

Мощным средством получения новых функций из уже имеющихся является суперпозиция, т. е. подстановка функций в функцию. Придадим ей стандартный вид.

Оператором суперпозиции S_m^n называется подстановка в функцию от m переменных m функций от n одних и тех же переменных. Суперпозиция дает новую функцию от n переменных.

Пример 1.
Для функций $h(y_1, \dots, y_m)$, $g_1(x_1, \dots, x_n)$, ..., $g_m(x_1, \dots, x_n)$

$$S_m^n(h, g_1, \dots, g_m) = h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n)) = $$
$$= f(x_1, \dots, x_n). \quad \triangle$$

Это определение порождает семейство операторов суперпозиции $\{S_m^n\}$. Благодаря функциям проекции, стандартизация суперпозиции не уменьшает ее возможностей: любую подстановку функций в функцию можно выразить через S_m^n, U_m^n.

Пример 2. Функцию $f(x_1, x_2) = h(g_1(x_1, x_2), g_2(x_1))$ можно записать в следующем стандартном виде:

$$f(x_1, x_2) = S_2^2(h(x_1, x_2), g_1(x_1, x_2),$$
$$S_2^1(U_1^2(x_1, x_2), g_2(x_1), g_3(x_1)),$$

где g_3 — любая функция от x_1. $\triangle$

Таким образом, если заданы функции U_m^n и операторы S_m^n, то можно считать заданными всевозможные операторы подстановки функций в функции, а также переименования, перестановки и отождествления переменных.

Пример 3.

$$f(x_2, x_1, x_3, \dots, x_n) = f(U_2^2(x_1, x_2), U_1^2(x_1, x_2), x_3, \dots, x_n);$$
$$f(x_1, x_1, x_3, \dots, x_n) = f(U_1^2(x_1, x_2), U_1^2(x_1, x_2), x_3, \dots, x_n). \quad \triangle$$

Нам понадобится еще одно семейство операторов — операторы примитивной рекурсии.

Оператор примитивной рекурсии R_n определяет $(n + 1)$-местную функцию f через n-местную функцию g и $(n + 2)$-местную функцию h так:

$$\begin{cases} f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n); \\ f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)). \end{cases} \quad (1)$$

Тот факт, что функция f определена схемой (1), выражается равенством $f(x_1, \ldots, x_n, y) = R_n(g, h)$. Эта схема определяет f рекурсивно не только через другие функции, но и через значения f в предшествующих точках: значение f в точке $y + 1$ зависит от значения f в точке y. Для вычисления $f(x_1, \ldots, x_n, k)$ понадобится $k + 1$ вычислений по схеме (1) для $y = 0, 1, \ldots, k$. Существенным в операторе примитивной рекурсии является то, что независимо от числа переменных в f рекурсия ведется только по одной переменной y; остальные n переменных $x_1, \ldots, x_n$ на момент применения схемы (1) зафиксированы и играют роль параметров.

Функция называется *примитивно-рекурсивной*, если она может быть получена из константы 0, функции следования x' и функции проекции U_m^n с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии.

Этому определению можно придать более формальный индуктивный вид.

1) Функции 0, x' и U_m^n для всех натуральных n, m, где $m \le n$, являются примитивно-рекурсивными.
2) Если $g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)$, $h(y_1, \ldots, y_m)$ примитивно-рекурсивные функции, то $S_m^n(h, g_1, \ldots, g_m)$ примитивно-рекурсивная функция для любых натуральных n, m.
3) Если $g(x_1, \ldots, x_n)$ и $h(x_1, \ldots, x_n, y, z)$ примитивно-рекурсивные функции, то $R_n(g, h)$ примитивно-рекурсивная функция.
4) Других примитивно-рекурсивных функций нет.

Из такого индуктивного описания нетрудно извлечь процедуру, порождающую все примитивно-рекурсивные функции.

Пример 4. Сложение $f_+(x, y) = x + y$ примитивно-рекурсивно:

$f_+(x, 0) = x = U_1^1(x);$

$f_+(x, y + 1) = f_+(x, y) + 1 = (f_+(x, y))'.$

Таким образом, $f_+(x, y) = R_1(U_1^1(x), h(x, y, z))$, где

$h(x, y, z) = z' = z + 1$. △

Пример 5. Умножение $f_*(x, y) = xy$ примитивно-рекурсивно:

$f_*(x, 0) = 0$; $f_*(x, y + 1) = f_*(x, y) + x = f_+(x, f_*(x, y))$. △

Пример 6. Возведение в степень $f_{\exp}(x, y) = x^y$ примитивно-рекурсивно:

$f_{\exp}(x, 0) = 1$; $f_{\exp}(x, y + 1) = x^y \cdot x = f_*(x, f_{\exp}(x, y))$. △

Определим функцию $x \dot{-} y$ (арифметическое или урезанное вычитание):

$$x \dot{-} y = \begin{cases} x - y, & \text{если } x > y, \\ 0, & \text{в противном случае.} \end{cases}$$

Пример 7. Примитивно-рекурсивными являются следующие функции, связанные с вычитанием:

1) $\bar{f}(x) = x \dot{-} 1$, определяемая схемой:

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

2) $f_-(x, y) = x \dot{-} y$, определяемая схемой:

$f_-(x, 0) = x$;

$f_-(x, y + 1) = x \dot{-} (y + 1) = (x \dot{-} y) \dot{-} 1 = f_-(x, y) \dot{-} 1 =$
$= \bar{f}(f_-(x, y))$;

3) $f(x, y) = |x - y| = x \dot{-} y + y \dot{-} x = f_+(f_-(x, y), f_-(y, x))$;

4)
$$sg(x) = \begin{cases} 0, & \text{если } x = 0, \\ 1, & \text{если } x \neq 0. \end{cases}$$

Схема этой функции имеет вид:

$sg(0) = 0$; $sg(x + 1) = 1$;

5) $\min(x, y) = x \dot{-} (x \dot{-} y) = f_-(x, f_-(x, y))$;

6) $\max(x, y) = y + (x \dot{-} y) = f_+(y, f_-(x, y))$. △

Пример 8. Примитивно-рекурсивными являются следующие функции, связанные с делением ($\overline{sg}(x) = 1 \dot{-} sg(x)$ — отрицание функции сигнум):

1) $r(x, y)$ — остаток от деления y на x:

$r(x, 0) = 0$;

$r(x, y + 1) = (r(x, y) + 1) \cdot sg(|x - (r(x, y) + 1)|)$.

Смысл второй строчки определения в следующем: если $(y + 1)$ не делится нацело на x, то $sg(|\, x - (r(x, y) + 1)\,|) = 1$ и $r(x, y + 1) = r(x, y) + 1$; если же $(y + 1)$ делится на x, то $sg(|\, x - (r(x, y) + 1)\,|) = 0$, а значит, и $r(x, y + 1) = 0$.

2) $q(x, y) = [y/x]$ — целая часть дроби y/x:

$q(x, 0) = 0;$

$q(x, y + 1) = q(x, y) + \overline{sg}(|\, x - (r(x, y) + 1)\,|).$

Второе слагаемое, как и в случае $r(x, y)$, зависит от делимости $(y + 1)$ на x. Если $(y + 1)$ делится нацело на x, то $q(x, y + 1)$ на единицу больше, чем $q(x, y)$, если нет, то $q(x, y + 1) = q(x, y)$. △

В рекурсивных описаниях функций примера 8 видны логические действия: проверка условия (делимости $y + 1$ на x) и выбор дальнейшего хода вычисления в зависимости от истинности условия. Займемся теперь логическими операциями более подробно.

Из функций примера 7 легко получается примитивная рекурсивность «арифметизированных» логических функций, то есть числовых функций, которые на множестве $\{0, 1\}$ ведут себя как логические функции. Действительно, если $x, y \in \{0, 1\}$, то

$$\begin{aligned} &\bar{x} = 1 \dot{-} x; \\ &x \vee y = \max(x, y); \qquad (2) \\ &x \,\&\, y = \min(x, y). \end{aligned}$$

Как известно из курса дискретной математики, множество функций (2) обладает свойством функциональной полноты. Отсюда и из того, что суперпозиция является примитивно-рекурсивным оператором (это в следующем пункте), следует примитивная рекурсивность всех логических функций.

Отношение $R(x_1, \ldots, x_n)$ называют *примитивно-рекурсивным*, если примитивно-рекурсивна его характеристическая функция χ_R:

$$\chi_R(x_1, \ldots, x_n) = \begin{cases} 1, & \text{если } R(x_1, \ldots, x_n) \text{ выполняется,} \\ 0, & \text{в противном случае.} \end{cases}$$

Ввиду взаимнооднозначного соответствия между отношениями и предикатами χ_R будет характеристической функцией и для соответствующего предиката.

Предикат называют *примитивно-рекурсивным*, если его характеристическая функция примитивно-рекурсивна.

В силу (2), если предикаты $P_1, \ldots, P_k$ примитивно-рекурсивны, то и предикат, полученный из них с помощью логических операций, примитивно-рекурсивен.

Пример 9. Следующие предикаты и отношения примитивно-рекурсивны:

1) $Pd_n(x)$ — предикат «x делится на n»,
$\chi_{Pd_n}(x) = \overline{sg}(r(n, x))$;
2) $Pd_{n,m}(x)$ — предикат «x делится на m и n»,
$Pd_{n,m}(x) = Pd_m(x) \,\&\, Pd_n(x)$;
3) отношение $x_1 > x_2$,
$\chi_>(x_1, x_2) = sg(x_1 \dot{-} x_2)$;
4) если функции $f(x)$, $g(x)$ примитивно-рекурсивны, то предикат «$f(x) = g(x)$» также примитивно-рекурсивен, его характеристическая функция $\chi(x) = \overline{sg}(|\, f(x) - g(x)\,|)$. △

3.2. Примитивно-рекурсивные операторы

Оператор называется *примитивно-рекурсивным* (ПР-оператором), если он сохраняет примитивную рекурсивность функции.

Операторы S^n_m и R_n являются ПР-операторами по определению. В п. 2.2 мы встречались с оператором условного перехода (развилкой), обозначим его тут B, который по функциям $q_1(x_1, \ldots, x_n)$, $q_2(x_1, \ldots, x_n)$ и предикату $P(x_1, \ldots, x_n)$ строит функцию $f(x_1, \ldots, x_n) = B(q_1, q_2, P)$:

$$f(x_1, \ldots, x_n) = \begin{cases} q_1(x_1, \ldots, x_n), \text{ если } P(x_1, \ldots, x_n) \text{ истинно}, \\ q_2(x_1, \ldots, x_n), \text{ если } P(x_1, \ldots, x_n) \text{ ложно}. \end{cases} \quad (3)$$

Примитивная рекурсивность B видна из следующего соотношения, эквивалентного (3):

$$f(x_1, \ldots, x_n) = g_1(x_1, \ldots, x_n)\, \chi_p(x_1, \ldots, x_n) + \\ + g_2(x_1, \ldots, x_n)\, \chi_{\bar{p}}(x_1, \ldots, x_n).$$

Пусть $f(x_1, \ldots, x_n, y)$ — функция от $(n + 1)$-й переменной. Хорошо известные операции суммирования $\sum\limits_{y<z}$ и умножения $\prod\limits_{y<z}$ по переменной y с пределом z — это операторы,

которые из функции $f(x_1, \ldots, x_n, y)$ порождают новые функции:

$$q(x_1, \ldots, x_n, z) = \sum_{y<z} f(x_1, \ldots, x_n, y),$$

$$h(x_1, \ldots, x_n, z) = \prod_{y<z} f(x_1, \ldots, x_n, y).$$

Они примитивно-рекурсивны (если f — примитивно-рекурсивна) в силу следующих соотношений:

$g(x_1, \ldots, x_n, 0) = 0$ (по определению);

$g(x_1, \ldots, x_n, z + 1) = g(x_1, \ldots, x_n, z) + f(x_1, \ldots, x_n, z)$;

$h(x_1, \ldots, x_n, 0) = 1$ (по определению);

$h(x_1, \ldots, x_n, z + 1) = h(x_1, \ldots, x_n, z) \cdot f(x_1, \ldots, x_n, z)$.

Теперь легко показать, что операторы $\sum_{y=k}^{z}$ и $\prod_{y=k}^{z}$ также примитивно-рекурсивны.

Ограниченный оператор наименьшего числа (μ-оператор) или *ограниченный оператор минимизации*, который применяется к предикатам, определяется так:

$$\mu y_{y \le z} P(x_1, \ldots, x_n, y) = \begin{cases} \text{наименьшему } y \le z, \text{ такому,} \\ \text{что } P(x_1, \ldots, x_n, y) \text{ истинно,} \\ \text{если такое } y \text{ существует;} \\ z \text{ в противном случае.} \end{cases}$$

Из предиката $P(x_1, \ldots, x_n, y)$ с помощью оператора $\mu y_{y \le z}$ получается функция $f(x_1, \ldots, x_n, z)$. Второй случай в определении μ добавлен для того, чтобы f была всюду определена.

Пример 10. Пусть $P(x_1, x_2, y) = Pd_{x_1, x_2}(y)$ (пример 9 из п. 3.1). Тогда $\mu y_{y \le z} P(x_1, x_2, y)$ равен наименьшему общему кратному (НОК) x_1 и x_2, если $z \ge \text{НОК}(x_1, x_2)$, и равен z, если $z \le \text{НОК}(x_1, x_2)$. △

Ограниченный μ-оператор примитивно-рекурсивен:

$$\mu y_{y \le z} P(x_1, \ldots, x_n, y) = \sum_{i=0}^{z} \prod_{j=0}^{i} (1 - \chi_P(x_1, \ldots, x_n, j)).$$

Ограничение суммирования до величины z в ограниченном μ-операторе дает гарантию окончания вычислений, поскольку оно оценивает сверху число вычислений предиката P. В дальнейшем будет показано, что неограниченный μ-оператор не является примитивно-рекурсивным.

μ-оператор (как ограниченный, так и неограниченный) является удобным средством для построения обратных функций. Действительно, функция $g(x) = \mu y(f(y) = x)$ («наименьший y, такой, что $f(y) = x$») является обратной к функции $f(x)$.

Поэтому в применении к одноместным функциям μ-оператор иногда называют оператором обращения.

Пример 11. Определим функцию $[z/x]$ как функцию, обратную умножению (целая часть является обратной функцией только тогда, когда z делится нацело на x; поэтому в описании используется не предикат равенства, а предикат $>$):

$$[z/x] = \mu y_{y \le z}(x(y + 1) > z). \qquad \triangle$$

Пример 12. Функция $[\sqrt{x}]$ примитивно-рекурсивна:

$$[\sqrt{x}] = \mu y_{y \le z}((y + 1)^2 > x). \qquad \triangle$$

Пример 13. Целая часть логарифма по любому целому основанию k примитивно-рекурсивна:

$$[\log_k x] = \mu y_{y \le z}(k^{y+1} > x). \qquad \triangle$$

Подведем некоторые итоги. Из простейших функций — константы 0, функции $x + 1$ и функции проекции U — с помощью операторов суперпозиции и примитивной рекурсии было получено огромное разнообразие функций, включая основные функции арифметики, алгебры и анализа (с поправкой на целочисленность). Тем самым выяснено, что эти функции имеют примитивно-рекурсивное описание, которое однозначно определяет процедуру их вычисления. Следовательно, их естественно отнести к классу вычислимых функций.

Сделаем два замечания. Во-первых, все примитивно-рекурсивные функции всюду определены. Это следует из того, что простейшие функции всюду определены, а операторы S_m^n и R_n это свойство сохраняют. Во-вторых, строго говоря, мы имеем дело не с функциями, а с их примитивно-рекурсивными описаниями. Эти описания также можно разбить на классы эквивалентности, отнеся в один класс все описания, задающие одну и ту же функцию. Однако задача распознавания эквивалентности примитивно-рекурсивных описаний, как будет показано далее, алгоритмически неразрешима.

3.3. Функции Аккермана

После проведенных в предыдущих пунктах рассуждений возникает естественный вопрос: все ли вычислимые функции можно описать как примитивно-рекурсивные? Чтобы показать, что ответ на этот вопрос является отрицательным, построим пример вычислимой функции, не являющейся примитивно-рекурсивной. Идея примера в том, чтобы построить последовательность функций, каждая из которых растет существенно быстрее предыдущей, и сконструировать с ее помощью функцию, которая растет быстрее любой примитивно-рекурсивной функции.

Итак, мы хотим найти по возможности простые, но очень быстро растущие функции. Опыт показывает, что произведение растет быстрее суммы, а степень быстрее произведения. Называя сложение, умножение и возведение в степень действиями 0-й, 1-й и 2-й ступени и вводя для них в целях единообразия обозначения

$$P_0(a,\ x) = a + x,\quad P_1(a,\ x) = ax,\quad P_2(a,\ x) = a^x,$$

приходим к знакомой всем идее о продолжении этой последовательности путем введения действий высших ступеней. При этом действие высшей ступени должно возникать из действия предыдущей ступени так же, как умножение возникает из сложения, возведение в степень из умножения. Функции P_0, P_1, P_2 связаны следующими соотношениями:

$$P_1(a,\ x+1) = P_0(a,\ P_1(a,\ x));\ P_1(a,\ 1) = a;\ P_1(a,\ 0) = 0;$$

$$P_2(a,\ x+1) = P_1(a,\ P_2(a,\ x));\ P_2(a,\ 1) = a;\ P_2(a,\ 0) = 1. \quad (4)$$

Продолжим эту цепочку, полагая по определению для $n = 2,\ 3,\ \ldots$

$$\begin{aligned} &P_{n+1}(a,\ 0) = 1; \\ &P_{n+1}(a,\ 1) = a; \qquad (5) \\ &P_{n+1}(a,\ x+1) = P_n(a,\ P_{n+1}(a,\ x)). \end{aligned}$$

Ясно, что соотношения (4) вытекают из (5) и соотношения $P_1(a,\ 0) = 0$. Растут функции P_n крайне быстро. Например, при $n = 3$ имеем:

$$P_3(a,\ 0) = 1;\ P_3(a,\ 1) = a;\ P_3(a,\ 2) = a^a;\ P_3(a,\ 3) = a^{a^a}.$$

Введем новые функции

$$B(n,\ x) = P_n(2,\ x),\ A(x) = B(x,\ x).$$

Функции $B(n, x)$ называют *функциями Аккермана*, а функцию $A(x)$ — *диагональной функцией Аккермана*.

Для функций $B(n, x)$ из (5) вытекают следующие тождества:

$$\begin{cases} B(0, x) = 2 + x, \\ B(n+1, 0) = sg(n), \\ B(n+1, x+1) = B(n, B(n+1, x)). \end{cases} \tag{6}$$

Эти соотношения позволяют вычислять значения функций $B(n, x)$ и, следовательно, $A(x)$, причем для вычисления функций в данной точке нужно обратиться к значению функции в предшествующей точке — совсем как в схеме примитивной рекурсии. Однако здесь рекурсия ведется сразу по двум переменным, и это существенно усложняет характер упорядочения точек и нахождение значения функции в предшествующей точке.

Теорема. Функция Аккермана $A(x)$ растет быстрее, чем любая примитивно-рекурсивная функция, и, следовательно, не является примитивно-рекурсивной. Более точно, для любой одноместной примитивно-рекурсивной функции $f(x)$ найдется такое n, что для любого $x \geq n$, $A(x) > f(x)$.

С доказательством можно ознакомиться, например, в [10].

Так как функция Аккермана тотальная (т. е всюду определенная) вычислимая функция, и так как она не примитивно-рекурсивна, то мы должны отказаться от идеи, что понятие примитивно-рекурсивной функции является точным формальным аналогом неформального понятия алгоритма.

3.4. Частично-рекурсивные функции. Тезис Чёрча

Пример функции Аккермана $A(x)$ показывает, что средства построения вычислимых функций нуждаются в расширении. Можно показать, что операторы кратной рекурсии (т. е. по нескольким переменным одновременно) не дают желаемого замыкания класса всех вычислимых функций. Более подходящим для этой цели оказывается неограниченный μ-оператор (в дальнейшем прилагательное «неограниченный» будем опускать).

Функция называется *частично-рекурсивной*, если она может быть построена из простейших функций 0, $x + 1$, U_m^n с помощью конечного числа применений операторов суперпозиции, примитивной рекурсии и μ-оператора.

По определению μ-оператор применяется к предикатам. Поскольку в теории рекурсивных функций истинность предиката $P(x)$ всегда связана со справедливостью некоторого равенства (например, $\chi_P(x) = 0$), и, наоборот, всякое равенство является предикатом от содержащихся в нем переменных, то μ-оператору можно придать стандартную форму, например:

$$f(x_1, \ldots, x_n) = \mu y(g(x_1, \ldots, x_{n-1}, y) = x_n)$$

или

$$f(x_1, \ldots, x_n) = \mu y(g(x_1, \ldots, x_n, y) = 0).$$

Будучи применен к вычислимой функции, μ-оператор снова дает вычислимую функцию. Действительно, для вычисления функции $f(x_1, \ldots, x_n) = \mu y(g(x_1, \ldots, x_n, y) = 0)$ на наборе $(x_1, \ldots, x_n)$ существует простая процедура: вычисляем g на наборах $(x_1, \ldots, x_n, 0)$, $(x_1, \ldots, x_n, 1)$, ... до тех пор, пока не получим значения нуль. Однако, в отличие от рассмотренных ранее процедур, она может не привести к результату: это произойдет в случае, когда на данном наборе $(x_1, \ldots, x_n)$ уравнение $g(x_1, \ldots, x_n, y) = 0$ не имеет решения. В этом случае функция $g(x_1, \ldots, x_n, y)$ считается неопределенной. Например, обратная к $x + 1$ функция $x - 1 = \mu y(y + 1 = x)$ не определена при $x = 0$. Заметим, что механизм возникновения неопределенности здесь такой же, как и при вычислениях на машине Тьюринга: в случае неопределенности процесс вычисления не останавливается.

Таким образом, среди рекурсивных функций появляются не полностью определенные, т. е. *частичные функции*. В случае когда функция g, к которой применяется μ-оператор, сама является частичной, функция $f(x_1, \ldots, x_n) = \mu y(g(x_1, \ldots, x_n, y) = 0)$ вычисляется с учетом следующего условия: если для $y = 0, 1, \ldots, i - 1$ функция $g(x_1, \ldots, x_n, y) \neq 0$, а $g(x_1, \ldots, x_n, i)$ не определена, то и $f(x_1, \ldots, x_n)$ не определена. Например, функция $f(x) = \mu y[y - x = 5]$ при $x = 1$ не определена (хотя уравнение $y - 1 = 5$ имеет решение $y = 6$), так как функция $y - 1 - 5$ не определена при $y = 0$.

Частично-рекурсивная функция называется *общерекурсивной*, если она всюду определена.

Пусть функция $f(x_1, ..., x_n)$ задана следующим образом: «пока $P(x_1, ..., x_n)$ истинно, вычислять $q_1(x_1, ..., x_n)$, иначе вычислять $q_2(x_1, ..., x_n)$». Функция $f(x_1, ..., x_n)$ называется повторением или циклом от $q_1(x_1, ..., x_n)$, $q_2(x_1, ..., x_n)$ по условию $P(x_1, ..., x_n)$ и обозначается $f(x_1, ..., x_n) = D(q_1, q_2, P)$.

Можно показать, что если q_1, q_2 и P примитивно-рекурсивны, то $f = D(q_1, q_2, P)$ частично-рекурсивна.

Объединяя этот результат с примитивной рекурсивностью операторов развилки B (п. 3.2) и S^n_m, а также на основании теоремы Бома–Джакопини (п. 1.3) мы получаем, что любая программа (в смысле программы для компьютера) может быть реализована частично-рекурсивной функцией.

Рассмотрим теперь, как выполняются основные требования к алгоритмам а)–е) из п. 1.1.1 в построенной нами алгоритмической модели — частично-рекурсивные функции.

Детерминированность определяется полной определенностью в вычислении простейших функций 0, $x + 1$, U^n_m, а также полной заданностью в действиях операторов суперпозиции, примитивной рекурсии и μ-оператора. Все соответствующие шаги подробно и однозначно описаны при их определении. Там же показана **элементарность** каждого шага и **дискретность** вычислений.

Результатом является значение частично-рекурсивной функции, которое вычисляется в процессе реализации описанных операторов. Если функция не определена, то процесс вычислений не останавливается.

Возможность выбора в качестве аргумента любого натурального числа обеспечивает **массовость** частично-рекурсивных функций.

Понятие частично-рекурсивной функции оказалось исчерпывающей формализацией понятия вычислимой функции (в частности, функция Аккермана общерекурсивна [10]).

Это обстоятельство выражено в виде тезиса Чёрча: ***всякая функция, вычислимая некоторым алгоритмом, частично-рекурсивна.***

Комментарии к этому тезису полностью совпадают с комментариями к тезису Тьюринга (см. п. 2.4).

Упражнения

1. Доказать, что если функция $f(x_1, x_2, \ldots, x_n)$ примитивно-рекурсивна, то следующие функции примитивно-рекурсивны:

   a) $g(x_1, x_2, x_3, \ldots, x_n) = f(x_2, x_1, x_3, \ldots, x_n)$ — перестановка аргументов;

   b) $\psi(x_1, x_2, \ldots, x_n) = f(x_2, \ldots, x_n, x_1)$ — циклическая перестановка аргументов;

   c) $h(x_1, \ldots, x_n, x_{n+1}) = f(x_1, \ldots, x_n)$ — введение фиктивного аргумента;

   d) $\varphi(x_1, \ldots, x_{n-1}) = f(x_1, x_1, x_2, \ldots, x_{n-1})$ — отождествление аргументов.

2. Доказать, что следующие функции примитивно-рекурсивны:

   a) $f(x) = x + n$;

   b) $f(x, y) = x + y$;

   c) $f(x, y) = x \cdot y$;

   d) $f(x) = 10^x$ $(0^0 = 1)$;

   e) $f(x) = x!$ $(0! = 1)$;

   f) $f(x) = x \dot{-} 2$;

   g) $f(x) = 2^{x \dot{-} 1}$.

3. Какая функция получится с помощью схемы примитивной рекурсии (по данной схеме примитивной рекурсии восстановить функцию)?

   a) $f(x, 0) = x$,
   $f(x, y + 1) = x^{f(x, y)}$;

   b) $f(x, 0) = x$,
   $f(x, y + 1) = (f(x, y))^x$.

4. Доказать, что следующие функции примитивно-рекурсивны:

   a) $\min(x, y)$;

   b) $\max(x, y)$;

   c) $| x \dot{-} y |$.

5. Доказать, что следующие функции примитивно-рекурсивны:

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

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

c) $f(x, y) = 10^{y \dot{-} x}$;

d) $f(x, y) = y \dot{-} (y \dot{-} x)$.

6. Какая функция получится с помощью схемы примитивной рекурсии?

   a) $g(x, 0) = 2,\ g(x, y + 1) = g(x, y)^2$;

   b) $f(x, 0) = x,\ f(x, y + 1) = f(x, y) \cdot x$.

7. Пусть g — примитивно-рекурсивная функция. Доказать, что следующие функции примитивно-рекурсивны:

   a) $f(x_1, \ldots, x_n, x_{n+1}) = \sum\limits_{i=0}^{x_{n+1}} g(x_1, \ldots, x_n, i)$;

   b) $f(x_1, \ldots, x_n, y, z) = \begin{cases} \sum\limits_{i=y}^{z} g(x_1, \ldots, x_n, i), & \text{если } y \le z, \\ 0, & \text{если } y > z; \end{cases}$

   c) $f(x_1, \ldots, x_n, x_{n+1}) = \prod\limits_{i=0}^{x_{n+1}} g(x_1, \ldots, x_n, i)$;

   d) $f(x_1, \ldots, x_n, y, z) = \begin{cases} \prod\limits_{i=y}^{z} g(x_1, \ldots, x_n, i), & \text{если } y \le z, \\ 0, & \text{если } y > z. \end{cases}$

Тестовые задания

1. Среди требований к алгоритмам одно лишнее:
   a) простота;
   b) детерминированность;
   c) дискретность;
   d) результативность.

2. Среди перечисленных средств описания примитивно-рекурсивных функций одно лишнее:
   a) оператор минимизации;
   b) оператор суперпозиции;
   c) оператор примитивной рекурсии;
   d) константа 0;
   e) функция следования;
   f) функция проекции.

3. Частично-рекурсивная функция называется общерекурсивной, если она:
   a) всюду определена;
   b) может быть получена с помощью константы 0, функции следования и оператора проекции;
   c) все ответы верные.

4. Чему равно значение функции проекции $U_2^5(x_1, x_2, x_3, x_4, x_5)$?
   a) x_2;
   b) 2;
   c) 5;
   d) x_5.

5. Примитивно-рекурсивная функция $\bar{f}(x) = x \dot{-} 1$ определяется схемой:
   a) $\bar{f}(0) = 0,\ \bar{f}(1) = 0,\ \bar{f}(x + 1) = \bar{f}(x) + 1$;
   b) $\bar{f}(0) = 1,\ \bar{f}(1) = 0,\ \bar{f}(x + 1) = x$;
   c) $\bar{f}(0) = 0,\ \bar{f}(x + 1) = \bar{f}(x) - 1$.

6. Всякая эффективно вычислимая функция частично-рекурсивна. Это высказывание принадлежит:
   a) Чёрчу;
   b) Тьюрингу;
   c) Райсу.

7. Поставить в соответствие:
   a) μ;
   b) S_m^n ;
   c) R_n;
   d) x';
   e) U_m^n ;

   1) оператор минимизации;
   2) оператор суперпозиции;
   3) оператор примитивной рекурсии;
   4) функция следования;
   5) функция проекции;
   6) оператор тождества.

8. Верно ли, что оператор примитивной рекурсии R_n определяет n-местную функцию f через $(n+1)$-местную функцию g и $(n+2)$-местную функцию h?
   a) нет;
   b) да.

9. Какие значения m и n будут у оператора суперпозиции для функции $f(x_1, x_2) = h(g_1(x_1, x_2, x_3), g_2(x_1, x_2, x_3))$?
   a) S_2^2;
   b) R_2;
   c) S_2^3;
   d) S_3^2.

10. Поставить в соответствие:
   a) $h_1(x_1, x_2, x_3) = f(x_3, x_2, x_1)$;
   b) $h_2(x) = f(x, x)$;
   c) $h_3(x_1, x_2, x_3) = f(x_2, x_3)$;

   1) циклическая перестановка аргументов;
   2) отождествление;
   3) добавление фиктивных переменных;
   4) перестановка аргументов.

11. Числа натурального ряда можно получить с помощью:
   a) константы 0 и функции следования;
   b) функции следования и оператора проекции;
   c) константы 0, функции следования и оператора проекции;
   d) константы 0 и оператора минимизации.

12. Чему будет равно значение функции $f(x) = 2^{x-1}$ в нуле?
   a) $0'$;
   b) $1/2$;
   c) $0''$.

13. Примитивно-рекурсивная функция $f_{\exp}(x, y) = x^y$ определяется схемой:
   a) $f_{\exp}(x, 0) = 0'$, $f_{\exp}(x, y+1) = f_*(f_{\exp}(x, y), U_1^1(x))$;
   b) $f_{\exp}(x, 0) = 0$, $f_{\exp}(x, y+1) = f_*(f_{\exp}(x, y), U_1^1(x))$;
   c) $f_{\exp}(x, 0) = 0'$, $f_{\exp}(x, y+1) = f_*(f_{\exp}(x, y), 0')$;
   d) $f_{\exp}(x, 0) = 0'$, $f_{\exp}(x, y+1) = f_+(f_{\exp}(x, y), U_1^1(x))$.

14. Является ли функция Аккермана $A(x)$ примитивно-рекурсивной?
   a) да;
   b) нет.

15. Функции P_0, P_1 связаны следующими соотношениями:

$P_0(a, x) = a + x,$

$P_1(a, 1) = a,$

$P_1(a, x) = ax,$

$P_1(a, x + 1) = P_0(a, P_1(a, x)).$

Чему равно $P_1(2, 3)$?

a) 4;
b) 5;
c) 6.

16. Определенность в вычислении простейших функций 0, $x + 1$, U_m^n и полная заданность в действиях операторов суперпозиции, примитивной рекурсии и μ-оператора — это требование:

a) массовости;
b) детерминированности;
c) результативности.

Глава IV

Нормальные алгоритмы Маркова

В данной главе приводится еще одно уточнение понятия алгоритма — нормальные алгоритмы Маркова. Рассматривается большое количество примеров.

По аналогии с другими алгоритмическими моделями рассматриваются такие операции над алгоритмами Маркова, как композиция, соединение, разветвление и повторение.

В заключение формулируется принцип нормализации.

4.1. Нормальные алгоритмы

Теория нормальных алгоритмов разработана А. А. Марковым в конце 40-х годов прошлого века [11, 12]. Эти алгоритмы представляют собой правила по переработке слов в некоторых алфавитах.

Введем необходимые определения. *Алфавит* — это непустое множество. Элемент алфавита называется *буквой*. Последовательность букв данного алфавита называется *словом*. Пустое слово не имеет в своем составе ни одной буквы и обозначается Λ.

Если A и B — два алфавита, причем $A \subseteq B$ (A содержится в B), то алфавит B называется *расширением алфавита A*. Слова будем обозначать большими латинскими буквами: P, Q, R, ...; алгоритмы будем обозначать буквами M_1, M_2, ...

Процесс работы алгоритма над алфавитом A состоит в последовательном порождении слов в алфавите A. Пусть алгоритм M исходное слово P в алфавите A перерабатывает в слово Q. Будем обозначать это следующим образом:

$$M\colon P \Rightarrow Q. \qquad (1)$$

Процесс может закончиться порождением слова Q, которое мы назовем результатом работы алгоритма M над исходным словом. Говорят, что алгоритм M *применим к слову P*,

если процесс работы над этим словом заканчивается, то есть если имеется результат работы над данным словом как исходным.

В дальнейшем запись

$$!M(P) \qquad (2)$$

будет означать, что алгоритм M применим к слову P.

Пусть B — алфавит, а M_1 и M_2 — алгоритмы над алфавитом B. Будем говорить, что M_1 и M_2 эквивалентны относительно B, если для любых слов P и Q в алфавите B выполняются два условия:

1) если M_1: $P \Rightarrow Q$, то M_2: $P \Rightarrow Q$,
2) если M_2: $P \Rightarrow Q$, то M_1: $P \Rightarrow Q$.

Рассмотрим так называемые нормальные алгоритмы Маркова.

Марковской подстановкой называется операция над упорядоченной парой слов (P, Q), состоящая в следующем. В заданном слове R находят первое вхождение слова P (если оно есть) и, не изменяя остальных частей слова R, заменяют в нем это вхождение словом Q. Полученное слово называется результатом применения марковской подстановки (P, Q) к слову R. Если же нет вхождения P в слово 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 1. К слову R = *шрам* применим подстановку (*ра*, *ар*). В нашем случае P = *ра*, Q = *ар*, результатом будет слово *шарм*. △

Пример 2. В таблице рассматриваются примеры Марковских подстановок.

Преобразуемое слово	Марковская подстановка	Результат
541 551 678	(41 551, 00)	500 678
барабан	(ба, Λ)	рабан
шрам	(ра, ар)	шарм
функция	(Λ, β-)	β-функция
книга	(кн, Λ)	ига
мама	(Λ, Λ)	мама
дождь	(да, Т)	не применима

△

Запись $P \to Q$ называется формулой подстановки (P, Q). P называется левой частью, Q — правой частью в формуле

подстановки. Некоторые подстановки являются заключительными. Для обозначения таких подстановок будем использовать запись $P \to \cdot Q$, называя ее формулой заключительной подстановки.

Упорядоченный конечный список формул подстановок в алфавите A

$$\begin{cases} P_1 \to (\cdot)\, Q_1 \\ P_2 \to (\cdot)\, Q_2 \\ \dots \\ P_n \to (\cdot)\, Q_n \end{cases} \tag{3}$$

называется *схемой нормального алгоритма* в алфавите A. Запись точки в скобках означает, что она может стоять на этом месте, а может отсутствовать.

Нормальным алгоритмом Маркова в алфавите A называется следующее правило построения последовательности $\{R_i\}$ слов в алфавите A, исходя из данного слова R в этом алфавите. В качестве начального слова R_0 последовательности берется слово R. Пусть для некоторого $i \geq 0$ слово R_i построено и процесс построения рассматриваемой последовательности еще не завершился. Если при этом в схеме нормального алгоритма нет формул подстановки, левые части которых входили бы в R_i, то R_{i+1} полагают равным R_i, и процесс построения последовательности считается завершившимся. Если же в схеме имеются формулы с левыми частями, входящими в R_i, то в качестве R_{i+1} берется результат марковской подстановки первой из таких формул к слову R_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 R. Последний член S последовательности называется результатом применения нормального алгоритма к слову R. При этом говорят, что нормальный алгоритм перерабатывает R в S. Последовательность $\{R_i\}$ будем записывать так:

$$R_0 \to R_1 \to R_2 \to \dots \to R_{m-1} \to R_m, \text{ при } R_0 = R,\ R_m = S.$$

Нормальный алгоритм в алфавите A представляет собой предписание, однозначно определяемое указанными выше объектами и однозначно определяющее последовательность действий над исходным словом в алфавите A.

Всякий нормальный алгоритм M будет задаваться указанием следующих объектов: некоторого алфавита A, некоторой схемы Z и, если требуется, расширения алфавита, не имеющего общих букв с алфавитом A.

Пример 3. Построить нормальный алгоритм Маркова, заменив в алфавите $A = \{a, b, c\}$ все буквы a на c. Используем символ α для расширения алфавита A. $B = \alpha \cup A$. Схема Z нормального алгоритма будет иметь следующий вид:

$$\begin{cases} \alpha a \to c\alpha \\ \alpha b \to b\alpha \\ \alpha c \to c\alpha \\ \alpha \to \cdot\Lambda \\ \Lambda \to \alpha \end{cases}$$

Например:
$aacbab \to \alpha aacbab \to c\alpha acbab \to cc\alpha cbab \to ccc\alpha bab \to cccb\alpha ab \to cccbc\alpha b \to cccbcb\alpha \to cccbcb.$ △

Пример 4. Пусть $A = \{a, b\}$ — алфавит. Рассмотрим следующую схему нормального алгоритма в A:

$$\begin{cases} a \to \cdot\Lambda \\ b \to b \end{cases}$$

Этот алгоритм всякое слово P в алфавите A, содержащее хотя бы одно вхождение буквы a, перерабатывает в слово, получающееся из P вычеркиванием самого первого (левого) вхождения буквы a. Пустое слово он перерабатывает в пустое. Например, $aabab \Rightarrow abab$, $ab \Rightarrow b$, $ba \Rightarrow b$, $aa \Rightarrow a$, $bab \Rightarrow bb$.

Алгоритм неприменим к словам, которые содержат только букву b. △

Пример 5. Пусть $A = \{a_0, a_1, \ldots, a_n\}$ — алфавит. Рассмотрим схему

$$\begin{cases} a_0 \to \Lambda \\ a_1 \to \Lambda \\ \ldots \\ a_n \to \Lambda \\ \Lambda \to \cdot\Lambda \end{cases}$$

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

Например:

$a_5a_1a_2a_1a_4a_0 \to a_5a_1a_2a_1a_4 \to a_5a_2a_1a_4 \to a_5a_2a_4 \to a_5a_4 \to$
$a_5 \to \Lambda \to \Lambda.$ △

Пример 6. В алфавите $A = \{1\}$ схема $\Lambda \to \cdot 1$ определяет нормальный алгоритм, который к каждому слову в A (т. е. к числам, записанным в унарном виде, где 0 есть Λ) приписывает слева 1. Следовательно, алгоритм вычисляет функцию $f(x) = x + 1$. △

Пример 7. Дана функция:

$$\varphi_3(\underbrace{11\ldots1}_{n}) = \begin{cases} 1, \text{ если } n \text{ делится на } 3, \\ \Lambda, \text{ если } n \text{ не делится на } 3. \end{cases}$$

Схема нормального алгоритма в алфавите $A = \{1\}$, реализующего эту функцию, выглядит следующим образом:

$$\begin{cases} 111 \to \Lambda \\ 11 \to \cdot\Lambda \\ 1 \to \cdot\Lambda \\ \Lambda \to \cdot 1 \end{cases}$$

Этот алгоритм работает по такому принципу: пока число букв 1 в слове больше 3, алгоритм последовательно стирает по три буквы. Если число букв меньше 3, но больше 0, то оставшиеся 1 или 11 стираются заключительно; если слово пусто, то оно заключительно переводится в слово 1.

Например:

$111111111 \to 111111 \to 111 \to \Lambda \to 1;$
$11111111 \to 11111 \to 11 \to \Lambda.$ △

Сформулируем теперь точное определение вычислимости функций. Функция f, заданная на некотором множестве слов алфавита A, называется *нормально вычислимой*, если найдется такое расширение B данного алфавита A ($A \subseteq B$) и такой нормальный алгоритм в B, что каждое слово P в алфавите A из области определения функции f этот алгоритм перерабатывает в слово $f(P)$.

Таким образом, нормальные алгоритмы примеров 6 и 7 показывают, что функции $f(x) = x + 1$ и $\varphi_3(x)$ нормально вычислимы. Причем соответствующие нормальные алгоритмы удалось построить в том же самом алфавите A, т. е. расширять алфавит не потребовалось ($B = A$).

Рассмотрим примеры алгоритмов, в которых требуется расширение алфавита. Эти алгоритмы демонстрируют основные операции с числовыми и нечисловыми объектами.

Пример 8. Построим нормальный алгоритм для вычисления функции $f(x) = x + 1$ не в унарной системе (как в примере 6), а в десятичной. $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, а нормальный алгоритм будем строить в его расширении $B = A \cup \{a, b\}$. Схема этого нормального алгоритма следующая:

$0b \to \cdot 1$	$a0 \to 0a$	$0a \to 0b$
$1b \to \cdot 2$	$a1 \to 1a$	$1a \to 1b$
$2b \to \cdot 3$	$a2 \to 2a$	$2a \to 2b$
$3b \to \cdot 4$	$a3 \to 3a$	$3a \to 3b$
$4b \to \cdot 5$	$a4 \to 4a$	$4a \to 4b$
$5b \to \cdot 6$	$a5 \to 5a$	$5a \to 5b$
$6b \to \cdot 7$	$a6 \to 6a$	$6a \to 6b$
$7b \to \cdot 8$	$a7 \to 7a$	$7a \to 7b$
$8b \to \cdot 9$	$a8 \to 8a$	$8a \to 8b$
$9b \to b0$	$a9 \to 9a$	$9a \to 9b$
$b \to \cdot 1$		$\Lambda \to a$

Если применить алгоритм к пустому слову Λ, то на каждом шаге должна будет применяться самая последняя формула схемы. Получается бесконечный процесс:

$\Lambda \to a \to aa \to aaa \to \dots$.

Это означает, что к пустому слову данный алгоритм неприменим.

Если применить алгоритм к слову 32, то получим следующую последовательность слов:

$32 \to a32 \to 3a2 \to 32a \to 32b \to 33$.

Таким образом, слово 32 преобразовалось в слово 33. Применим алгоритм к слову 399:

$399 \to a399 \to 3a99 \to 39a9 \to 399a \to 399b \to 39b0 \to 3b00 \to 400$. △

Пример 9. Удваивающий алгоритм. Позволяет увеличить в два раза число, записанное в унарной системе.

Будем использовать алфавит $A = \{1\}$ и его расширение $B = A \cup \{\alpha, \beta\}$. Тогда схема нормального алгоритма будет выглядеть следующим образом:

$$\begin{cases} 11\beta \to 1\beta 1 \\ \alpha 1 \to 1\beta 1\alpha \\ \beta \to \Lambda \\ \alpha \to \cdot\Lambda \\ \Lambda \to \alpha \end{cases}$$

Применим этот алгоритм к слову 111. В результате должно получиться слово 111111.

$111 \to \underline{\alpha 1}11 \to 1\beta 1\underline{\alpha 1}1 \to 1\beta\underline{11\beta}1\alpha 1 \to 1\beta 1\beta 11\underline{\alpha 1} \to$
$1\beta 1\beta 1\underline{11\beta}1\alpha \to 1\beta 1\beta\underline{11\beta}11\alpha \to 1\underline{\beta}1\beta 1\beta 111\alpha \to$
$\underline{11\beta}1\beta 111\alpha \to 1\beta\underline{11\beta}111\alpha \to 1\underline{\beta}1\beta 1111\alpha \to \underline{11\beta}1111\alpha \to$
$1\underline{\beta}11111\alpha \to 111111\underline{\alpha} \to 111111$ △

Пример 10. Присоединяющий алгоритм. Приписывает любое слово в конец исходного.

Данный алгоритм присоединяет слово Q к любому слову из алфавита $A = \{a, b, c\}$, где Q — какое-либо слово в A, ε — буква, не входящая в A. Тогда расширение алфавита $B = A \cup \varepsilon$. Схема будет иметь вид:

$$\begin{cases} \varepsilon a \to a\varepsilon \\ \varepsilon b \to b\varepsilon \\ \varepsilon c \to c\varepsilon \\ \varepsilon \to \cdot Q \\ \Lambda \to \varepsilon \end{cases}$$

Применим этот алгоритм к слову aab:
$aab \to \varepsilon aab \to a\varepsilon ab \to aa\varepsilon b \to aab\varepsilon \to aabQ$. △

Пример 11. Переворачивающий алгоритм. Записывает слово в обратном порядке.

Ограничимся алфавитом $A = \{1, 2\}$, α, β — буквы, не входящие в A, т. е. $B = A \cup \{\alpha, \beta\}$. По мере увеличения алфавита алгоритм будет более громоздким. Схема нормального алгоритма будет иметь вид:

$$\begin{cases} \alpha\alpha \to \beta\alpha \\ \beta\alpha \to \beta \\ \beta 1 \to 1\beta \\ \beta 2 \to 2\beta \\ \beta \to \cdot\Lambda \\ \alpha 11 \to 1\alpha 1 \\ \alpha 12 \to 2\alpha 1 \\ \alpha 22 \to 2\alpha 2 \\ \alpha 21 \to 1\alpha 2 \\ \Lambda \to \alpha \end{cases}$$

Применим схему к слову 1112, в результате должно получиться слово 2111.

1112 → α1112 → 1α112 → 11α12 → 112α1 → α112α1 →
1α12α1 → 12α1α1 → α12α1α1 → 2α1α1α1 →
α2α1α1α1 → αα2α1α1α1 → βα2α1α1α1 → β2α1α1α1 →
2βα1α1α1 → 2β1α1α1 → 21βα1α1 → 21β1α1 →
211βα1 → 211β1 → 2111β → 2111 △

4.2. Операции над алгоритмами Маркова. Принцип нормализации

Пусть M_1 и M_2 — алгоритмы, а P — слово. Будем применять запись $M_1(P) \simeq M_2(P)$ для выражения того факта, что либо алгоритмы M_1 и M_2 оба неприменимы к слову P, либо оба применимы и при этом $M_1(P) = M_2(P)$. Назовем два алгоритма M_1 и M_2 над алфавитом A *вполне эквивалентными* относительно A, если для любого слова P в алфавите A выполняется $M_1(P) \simeq M_2(P)$.

Те же алгоритмы назовем *эквивалентными* относительно алфавита A, если $M_1(P) \simeq M_2(P)$ всякий раз, когда P есть слово в A, и хотя бы одно из слов $M_1(P)$ или $M_2(P)$ определено и является словом в A.

Пусть K обозначает алфавит {*, 1}, $\mathbb{N}$ — множество всех натуральных чисел и φ есть частичная эффективно вычислимая функция от n аргументов, т. е. функция, отображающая некоторое подмножество множества $\mathbb{N}^n$ в $\mathbb{N}$. Пусть A — алфавит, содержащий 1. Для всякого натурального n определим по индукции $\overline{0} = 1$, $\overline{n+1} = \overline{n}1$, таким образом $\overline{1} = 11$, $\overline{2} = 111$ и т. д. Слова $\overline{n}$ называются цифрами. Определим теперь нормальный алгоритм M схемой

$$\{ \Lambda \to \cdot 1$$

Для любого слова P в алфавите A имеем $M(P) = 1P$. В частности, $M(\overline{n}) = \overline{n+1}$ при любом натуральном n. Так, например, $\overline{(2,3,1)}$ обозначается словом 111*1111*11. Через M_φ обозначим соответствующий алгоритм в K, т. е. такой алгоритм, что $M_\varphi\left(\overline{(k_1, \ldots, k_n)}\right) = \overline{\varphi(k_1, \ldots, k_n)}$, если значение $\varphi(k_1, \ldots, k_n)$ определено, и M_φ не определено в противном случае. Предполагается также, что алгоритм M_φ неприменим к словам, отличным от слов вида $\overline{(k_1, \ldots, k_n)}$.

Назовем функцию φ *частично вычислимой по Маркову*, если существует нормальный алгоритм M над K, вполне эквивалентный M_φ относительно K.

Если функция φ определена всюду, то есть для любого набора из n натуральных чисел, и является частично вычислимой по Маркову, то мы назовем ее *вычислимой по Маркову*.

Будем говорить о нормальном алгоритме, что он замкнут, если его схема содержит формулу подстановки вида $\Lambda \to \cdot Q$.

Два алгоритма можно сочетать следующим образом. Предписано, исходя из произвольных начальных данных, сначала применить первый алгоритм, а затем к результату его работы — второй. Это предписание составляет новый алгоритм — композицию двух данных алгоритмов.

Покажем теперь, что композиция двух нормальных алгоритмов есть нормальный алгоритм. Пусть M_1 и M_2 — нормальные алгоритмы в алфавите A. Сопоставим каждой букве b этого алфавита новую букву $\bar{b}$, которую назовем двойником буквы b. Пусть $\bar{A}$ — алфавит, состоящий из всех двойников букв алфавита A. Выберем еще две буквы α и β, не принадлежащие $A \cup \bar{A}$. Обозначим через Z_{M_1} схему, полученную из схемы нормального алгоритма $M_1^{\cdot}$ ($M^{\cdot}$ — это нормальный алгоритм, схема которого получается из схемы M добавлением новой формулы подстановки $\Lambda \to \cdot\Lambda$ в конце) заменой в ней точки в каждой заключительной формуле подстановки буквой α, и обозначим через Z_{M_2} схему, которая получается путем замены в схеме алгоритма $M_2^{\cdot}$ всех букв алфавита A их двойниками, всех точек буквами β с последующей заменой всех формул подстановки вида $\Lambda \to Q$ и $\Lambda \to \cdot Q$ соответственно формулами подстановки $\alpha \to \alpha Q$ и $\alpha \to \alpha\beta Q$.

Зададим нормальный алгоритм M_3 в алфавите $B = A \cup \bar{A} \cup \{\alpha, \beta\}$ сокращенно записанной схемой:

$$\begin{cases} b\alpha \to \alpha b \\ \alpha b \to \alpha \bar{b} \\ \bar{b}c \to \bar{b}\bar{c} \\ \bar{b}\beta \to \beta\bar{b} \\ \beta\bar{b} \to \beta b \qquad (b, c \in A), \\ b\bar{c} \to bc \\ \alpha\beta \to \cdot\Lambda \\ Z_{M_1} \\ Z_{M_2} \end{cases}$$

где b, c пробегают алфавит A.

Нормальный алгоритм M_3 над алфавитом A таков, что для любого слова P в A $M_3(P) \simeq M_2(M_1(P))$. Этот нормальный алгоритм называется *композицией* алгоритмов M_1 и M_2.

Пусть M — некоторый нормальный алгоритм в алфавите A, B — некоторое расширение A. В начало схемы алгоритма M добавим всевозможные формулы подстановки вида $b \to b$, где b — буква из $B \setminus A$. Полученная таким образом схема определяет некоторый нормальный алгоритм M_B в алфавите B, который неприменим ни к какому слову, содержащему буквы из $B \setminus A$, и такой, что $M_B(P) \simeq M(P)$ для любого слова P в A. Алгоритм M_B вполне эквивалентен алгоритму M относительно алфавита A и называется *формальным распространением* алгоритма M на алфавит B.

Пусть даны нормальные алгоритмы M_1 и M_2 соответственно в алфавитах A_1 и A_2. Рассмотрим алфавит $A = A_1 \cup A_2$ и формальные распространения M_{1A} и M_{2A} алгоритмов M_1 и M_2 на алфавит A. Композиция M_3 алгоритмов M_{1A} и M_{2A} называется *нормальной композицией* алгоритмов M_1 и M_2 и обозначается символом $M_2 \circ M_1$. В общем случае под $M_n \circ \ldots \circ M_1$ будем понимать $M_n \circ (\ldots \circ M_3 \circ (M_2 \circ M_1)\ldots)$. M_3 является нормальным алгоритмом над A, причем $M_3(P) \simeq M_2(M_1(P))$ для любого слова P в алфавите A_1, и, кроме того, M_3 применим только к таким словам P в алфавите A, которые удовлетворяют условиям:

1) P есть слово в A_1;
2) M_1 применим к P;
3) M_2 применим к $M_1(P)$.

Предположим, что алфавит B является расширением алфавита A, и пусть P — произвольное слово в алфавите B. *Проекцией* P^A *слова* P *на алфавит* A называется слово, получающееся из P стиранием всех вхождений букв из $B \setminus A$. Сокращенно записанная схема

$\{\xi \to \Lambda \quad (\xi \in B \setminus A)$

задает нормальный *проектирующий алгоритм* $M_{B,A}$, т. е. $M_{B,A}(P) = P^A$ для любого слова P в B. Если M_1 — нормальный алгоритм в алфавите A, и B есть расширение A, то нормальный алгоритм M_2 в алфавите B, задаваемый схемой алгоритма M_1, назовем *естественным распространением* алгоритма M_1 на алфавит B. Очевидно, что $M_2(P) \simeq M_1(P)$ для любого слова P в A и, кроме того, $M_2(PQ) \simeq M_1(P)Q$ для любого слова P в A и слова Q в $B \setminus A$. Заметим, что естественное распространение ал-

горитма M_1 на B отличается от формального распространени M_1 на B, так как последнее неприменимо ни к какому слов содержащему буквы из $B\backslash A$.

Часто приходится рассматривать результаты работы дву или нескольких алгоритмов над одними и теми же исходны ми данными. В этом случае полезным может оказаться по строение системы этих результатов, которые могут служит исходными данными при работе какого-нибудь другого алго ритма.

Теорема 1. Пусть $M_1, \ldots, M_k$ — нормальные алгоритм и A — объединение их алфавитов. Тогда существует нор мальный алгоритм M над A, называемый *соединением* алго ритмов $M_1, \ldots, M_k$, такой, что

$$M(P) \simeq M_1^\#(P)M_2^\#(P)\ldots M_k^\#(P)$$

для любого слова P в алфавите A, где $M_i^\#$ — естественное рас пространение M_i на A.

Доказательство. Докажем теорему для $k = 2$, после чег индукцией по k легко может быть получен и общий случай.

Введем алфавит $\overline{A}$ двойников букв алфавита A. Положи $B = A \cup \overline{A}$. Пусть $\overline{M}_1$ — нормальный алгоритм, схема кото рого получается заменой каждой буквы схемы алгоритм M_1 ее двойником, и пусть $\overline{M}_1^\#$, $M_2^\#$ — естественные распро странения соответственно алгоритмов $\overline{M}_1$ и M_2 на B. Пуст $A = \{a_1, \ldots, a_n\}$. Существуют нормальные алгоритмы N_1 и N над B такие, что N_1 подставляет одновременно $a_1\overline{a}_1$ вместо a_1 $a_2\overline{a}_2$ вместо a_2 и т. д., а N_2 подставляет a_1 вместо $\overline{a}_1$, a_2 вмест $\overline{a}_2$ и т. д. Существуют, кроме того, нормальные алгоритм $N_{A\overline{A}}$ и $N_{\overline{A}A}$ такие, что $N_{A\overline{A}}(P) = P^A P^{\overline{A}}$ и $N_{\overline{A}A} = P^{\overline{A}} P^A$ Тогда, как легко проверить, композиция $M = N_2 \circ \overline{M}_1^\# \circ N_{\overline{A}A}$ $\circ\, M_2^\# \circ N_{A\overline{A}} \circ N_1$ обладает искомым свойством:

$$M_1 \simeq M_1^\#(P)M_2^\#(P)$$

для любого слова P в алфавите A. [

Следствие. Пусть $M_1, \ldots, M_k$ — нормальные алгоритмы со ответственно в алфавитах $A_1, \ldots, A_k$ и пусть $A = A_1 \cup \ldots \cup A_k$ Тогда существует нормальный алгоритм M над $A \cup \{*\}$ такой что $M(P) \simeq M_1^\#(P) * M_2^\#(P) * \ldots * M_k^\#(P)$ для любого слов P в A, где $M_i^\#$ — естественное распространение M_i на A.

(В частности, $M(P) \simeq M_1(P) * M_2(P) * \ldots * M_k(P)$ дл любого P в алфавите $A_1 \cap \ldots \cap A_k$.)

Доказательство. Существует такой нормальный алгоритм N в $A \cup \{*\}$, что $N(P) = *$ для любого слова P в A. Алгоритм N задан схемой:

$$\begin{cases} a \to \Lambda \\ \Lambda \to \cdot * \end{cases} \quad (a \in A).$$

Пусть на основании теоремы 1 M есть соединение алгоритмов M_1, N, M_2, N, ..., N, M_k. Тогда

$$M(P) \simeq M_1^{\#} * M_2^{\#}(P) * \ldots * M_k^{\#}(P)$$

для любого слова P в A, и, в частности, $M(P) \simeq M_1(P) * M_2(P) * \ldots * M_k(P)$ для любого слова P в пересечении алфавитов A_1, ..., A_k. □

Лемма. (1) Пусть M — нормальный алгоритм в алфавите A и α — произвольная буква, не принадлежащая A. Тогда существует нормальный алгоритм N над $A \cup \{\alpha\}$ такой, что для любого слова P в A

$$N(P) \simeq \begin{cases} \alpha P, \text{ если } M(P) = \Lambda, \\ P, \text{ если } M(P) \neq \Lambda, \end{cases}$$

и алгоритм N применим только к тем словам, к которым применим M.

(2) Если M_1 и M_2 — нормальные алгоритмы в алфавите A и α — буква, не принадлежащая A, то существует нормальный алгоритм M_3 над $A \cup \{\alpha\}$ такой, что $M_3(P) \simeq M_1(P)$ и $M_3(\alpha P) \simeq M_2(P)$ для любого слова P в A.

Доказательство. (1) Существует нормальный алгоритм N_1 над $A \cup \{\alpha\}$, перерабатывающий Λ в α и всякое непустое слово в алфавите $A \cup \{\alpha\}$ — в Λ. Такой алгоритм может быть задан следующей схемой:

$$\begin{cases} a \to \beta \\ \beta\beta \to \beta \\ \beta \to \cdot\Lambda \\ \Lambda \to \cdot\alpha \end{cases} \quad (a \in A \cup \{\alpha\}),$$

где β — буква, не принадлежащая алфавиту $A \cup \{\alpha\}$.

Пусть $N_2 = N_1 \circ M$. Для любого слова P в A если $M(P) = \Lambda$, то $N_2(P) = \alpha$, а если $M(P) \neq \Lambda$, то $N_2(P) = \Lambda$. Пусть S — тождественный нормальный алгоритм в A (со схемой $\Lambda \to \cdot\Lambda$), и пусть N есть соединение алгоритмов N_2 и S. Тогда если $M(P) = \Lambda$, то $N(P) = \alpha P$, и если $M(P) \neq \Lambda$, то $N(P) = P$.

(2) Введем алфавит $\overline{A}$ двойников букв алфавита A. Пусть $B = A \cup \overline{A} \cup \{\alpha, \beta\}$, где $\beta \notin A \cup \overline{A} \cup \{\alpha\}$. Если мы заменим в схе ме алгоритма $M_2^{\cdot}$ всякую букву алфавита A ее двойником, вс точки — буквой β, а в получившейся в результате этого схем заменим всякую формулу подстановки вида $\Lambda \to Q$, $\Lambda \to \beta Q$ со ответственно на $\alpha \to \alpha Q$, и $\alpha \to \alpha\beta Q$, то получим схему $Z_{\overline{M}_2}$. Пусть Z_{M_1} — схема алгоритма $M_1^{\cdot}$. Построим теперь схему:

$$\begin{cases} \alpha a \to \alpha\overline{a} \\ \overline{a}b \to \overline{a}\overline{b} \\ \overline{a}\beta \to \beta\overline{a} \\ \beta\overline{a} \to \beta a \quad (a \in A) \\ a\overline{b} \to ab \quad (a, b \in A) \\ \alpha\beta \to \cdot\Lambda \\ Z_{\overline{M}_2} \\ Z_{M_1^{\cdot}} \end{cases}$$

Определенный этой схемой нормальный алгоритм M над $A \cup \{\alpha\}$ есть искомый алгоритм, т. е. $M_3(P) \simeq M_1(P)$ $M_3(\alpha P) \simeq M_2(P)$ для всякого слова P в A. □

К исходным данным могут быть применены разные ал горитмы в зависимости от того, обладают эти данные ка ким-то свойством или нет. Таким образом мы получаем но вый алгоритм, который назовем *разветвлением* алгоритмо M_1 и M_2, управляемым алгоритмом M_3.

Теорема 2. Пусть M_1, M_2, M_3 — нормальные алгоритмы и A — объединение их алфавитов. Тогда существует нор мальный алгоритм M над A такой, что

$$M(P) \simeq \begin{cases} M_2(P), \text{ если } P \text{ есть слово в } A \text{ и } M_3(P) = \Lambda, \\ M_1(P), \text{ если } P \text{ есть слово в } A \text{ и } M_3(P) \neq \Lambda, \end{cases}$$

применимый к тем и только тем словам в A, к которым приме ним M_3.

Доказательство. Пусть M_{11}, M_{22}, M_{33} — формальны распространения соответственно алгоритмов M_1, M_2, M_3 н A, и пусть α — буква, не входящая в A. По лемме (1) сущест вует такой нормальный алгоритм N над $A \cup \{\alpha\}$, что

$$N(P) = \begin{cases} \alpha P, \text{ если } P \text{ есть слово в } A \text{ и } M_{33}(P) = \Lambda, \\ P, \quad \text{если } P \text{ есть слово в } A \text{ и } M_{33}(P) \neq \Lambda. \end{cases}$$

Кроме того, по лемме (2), существует нормальный алгоритм M_4 над $A \cup \{\alpha\}$ такой, что если P есть слово в A, то $M_4(P) \simeq M_{11}(P)$ и $M_4(\alpha P) \simeq M_{22}(P)$. Теперь остается положить $M = M_4 \circ N$. □

Пусть даны алгоритмы M_1 и M_3 в алфавите A и произвольное слово P_0 в A. Применим M_1 к P_0. Если в результате получится некоторое слово P_1, то применим M_3 к P_1. Если окажется, что $M_3(P_1) = \Lambda$, то процесс заканчивается, если же $M_3(P_1) \neq \Lambda$, то применим M_1 к P_1.

Если в результате получится некоторое слово P_2, то снова применим M_3 к P_2 и опять, если $M_3(P_2) = \Lambda$, то процесс останавливается, если $M_3(P_2) \neq \Lambda$, то применим M_1 к P_2 и т. д. Определенный таким образом алгоритм M_2 называется *повторением* алгоритма M_1, управляемым алгоритмом M_3. Очевидно, что $M_2(P_0) = Q$ тогда и только тогда, когда существует последовательность слов $P_0, P_1, \ldots, P_n$ $(n > 0)$ такая, что $P_n = Q$, $M_3(P_n) = \Lambda$, $P_i = M_1(P_{i-1})$ при $0 < i \leq n$ и $M_3(P_i) \neq \Lambda$ при $0 < i < n$.

Теорема 3. Пусть M_1 и M_3 — нормальные алгоритмы, A — объединение их алфавитов, M_{11} и M_{33} — формальные распространения соответственно M_1 и M_3 на A. Тогда существует нормальный алгоритм M_2 над A, являющийся повторением алгоритма M_{11}, управляемым алгоритмом M_{33}.

Доказательство. Теорему, очевидно, достаточно доказать для случая, когда алфавиты алгоритмов M_1 и M_3 совпадают и, следовательно, $M_{11} = M_1$, $M_3 = M_{33}$. Пусть буква α не входит в A. По лемме (1) существует такой нормальный алгоритм N над $B = A \cup \{\alpha\}$, что

$$N(P) = \begin{cases} \alpha P, \text{ если } P \text{ есть слово в } A \text{ и } M_3(P) = \Lambda, \\ P, \quad \text{если } P \text{ есть слово в } A \text{ и } M_3(P) \neq \Lambda. \end{cases}$$

Пусть $S^* = N \circ M_1$, S^* есть нормальный алгоритм в некотором расширении F алфавита B. Пусть буква β не входит в F. Рассмотрим следующую схему:

$$\begin{cases} \xi\beta \to \beta\xi \quad (\xi \in F) \\ \beta\alpha \to \cdot\alpha \\ \beta \to \Lambda \\ Z_{S^\beta} \end{cases}$$

где Z_{S^β} — схема, полученная из схемы алгоритма S^* путем замены в ней всех точек буквой β. Эта схема определяет некото-

рый нормальный алгоритм M, причем $M(P) = Q$ тогда и только тогда, когда существует такая последовательность слов P_0, ..., P_n, что $P = P_0$, $Q = P_n$, $P_i = S^*(P_{i-1})$ $(i \le n)$ и P_n — единственное в этой последовательности слово, начинающееся с буквы α. Пусть L — алгоритм, проектирующий алфавит F на алфавит $F \setminus \{\alpha\}$ (т. е. стирающий все входящие буквы α).

Теперь легко убедиться в том, что нормальный алгоритм $M_2 = L \circ M$ — искомый. □

Таким образом, нам удается для всех процедур, претендующих на алгоритмичность, построить реализующие их нормальные алгоритмы Маркова. Будет ли это удаваться всегда? Утвердительный ответ на этот вопрос содержится в принципе нормализации Маркова: ***всякий алгоритм может быть реализован нормальным алгоритмом Маркова.*** Или, что эквивалентно, ***всякий алгоритм нормализуем.***

Так же, как и тезисы Тьюринга и Чёрча, данный принцип не является теоремой, так как понятие алгоритма не является точным математическим понятием. Принцип нормализации фактически представляет собой вариант этих тезисов, относящийся к нормальным алгоритмам.

Относительно этого утверждения можно повторить все, что было сказано в п. 2.4 относительно тезиса Тьюринга.

Упражнения

1. Что получится в результате следующих Марковских подстановок в слово «апельсин»:
   a) (Λ, к);
   b) (пельс, спир);
   c) (ль, Λ)?
2. Построить нормальный алгоритм Маркова для вычисления функции $f(x) = x - 1$ $(x > 1)$ в унарной системе счисления.
3. Построить нормальный алгоритм Маркова для вычисления функции $f(x) = x - 1$ $(x > 1)$ в десятичной системе счисления.
4. Дано слово в алфавите $A = \{a, b, c\}$. Построить нормальный алгоритм Маркова, присоединяющий слово Q к данному слову.
5. Построить нормальный алгоритм Маркова, удваивающий слово в унарной системе счисления.

6. Построить нормальный алгоритм Маркова, который переворачивает любое заданное слово в алфавите $A = \{0, 1, 2, 3\}$.
7. Дан алфавит $A = \{a, b, c\}$. Построить нормальный алгоритм Маркова, заменяющий все a на c в некотором слове.
8. Построить нормальный алгоритм Маркова, который в слове из алфавита $\{a, b, c, d, e, f\}$ все вхождения последовательности abc заменяет на символ f.
9. Построить нормальный алгоритм Маркова, который в слове из алфавита $\{a, b, c, d, e, f\}$ все символы a заменяет на f, а все f — на af.
10. Дан алфавит $A = \{a, b, c\}$. Построить нормальный алгоритм Маркова, который удаляет все b в некотором слове.
11. Построить нормальный алгоритм Маркова, который в слове из алфавита $\{a, b, c, d, e, f\}$ удаляет все вхождения последовательности bc.
12. Построить нормальный алгоритм Маркова, который в любом слове в алфавите $A = \{a, b\}$ переносит все буквы a в начало слова.
13. Построить нормальный алгоритм Маркова для вычисления функции $f(x) = x - 1$ в троичной системе счисления.
14. Построить нормальный алгоритм Маркова для вычисления функции $f(x) = x + 1$ в троичной системе счисления.
15. Построить нормальный алгоритм Маркова для вычисления функции $f(x) = x \text{ Mod } 3$ в унарной системе счисления.
16. Построить нормальный алгоритм Маркова проверки четности числа, записанного в десятичной системе счисления.
17. Построить нормальный алгоритм Маркова проверки делимости числа, записанного в десятичной системе счисления, на 5.
18. Построить нормальный алгоритм Маркова для определения системы счисления (найти минимальное возможное основание), в которой записано натуральное число (предполагается, что основание не может превышать 16_{10}).

19. Построить нормальный алгоритм Маркова для перевода числа из двоичной системы счисления в восьмеричную.

20. Построить нормальный алгоритм Маркова для перевода числа из двоичной системы счисления в четверичную.

21. Построить нормальный алгоритм Маркова для перевода числа из восьмеричной системы счисления в двоичную.

22. Построить нормальный алгоритм Маркова, который в любом слове из алфавита $A = \{a, b, c, \ldots, z\}$ удваивает все буквы, стоящие на четных местах в исходном слове.

Тестовые задания

1. Какая из трех основных алгоритмических моделей занимается переработкой слов в произвольных алфавитах?
   a) нормальные алгоритмы Маркова;
   b) рекурсивные функции;
   c) машины Тьюринга.

2. Функция вычислима машиной Тьюринга тогда и только тогда, когда она частично-рекурсивна. Это высказывание:
   a) Тьюринга—Чёрча;
   b) Тьюринга;
   c) Чёрча.

3. Что получится в результате марковской подстановки (пано, рама) в слово «панорама»?
   a) рамарама;
   b) панопано;
   c) панорама;
   d) рама.

4. Восстановить правильную последовательность команд в схеме алгоритма, определяющего, делится число в унарной системе на 3 или не делится.
   a) $\Lambda \to \cdot 1$;
   b) $111 \to \Lambda$;
   c) $1 \to \cdot\Lambda$;
   d) $11 \to \cdot\Lambda$.

5. Если A и B — два алфавита, причем $A \subseteq B$, то:
   a) алфавит B называется расширением алфавита A;
   b) алфавит A называется расширением алфавита B;

c) алфавит B содержится в A;
d) нет верного ответа.

6. Поставить в соответствие:
a) (P, Q);
b) $P \to Q$;
c) $P \to \cdot Q$;
1) заключительная подстановка;
2) формула подстановки;
3) схема нормального алгоритма;
4) упорядоченная пара слов.

7. В процессе работы алгоритма получилась следующая последовательность:
$badca \to bdca \to bdc \to dc \to d \to \Lambda \to \cdot\Lambda$.
Выбрать соответствующую схему.

a) $\begin{cases} a \to \Lambda \\ b \to \Lambda \\ c \to \Lambda \\ d \to \Lambda \\ \Lambda \to \cdot\Lambda \end{cases}$ b) $\begin{cases} a \to a \\ b \to b \\ c \to c \\ d \to d \\ \Lambda \to \cdot\Lambda \end{cases}$ c) $\begin{cases} a \to \Lambda \\ b \to \Lambda \\ c \to \cdot\Lambda \\ d \to \Lambda \end{cases}$

8. Нормальный алгоритм определяет схема:
$$\begin{cases} \alpha 1 \to 11\alpha \\ \alpha \to \cdot\Lambda \\ \Lambda \to \alpha \end{cases}$$
Что получится в результате применения данного алгоритма к слову 11?
a) 1111;
b) 111111;
c) 11111;
d) 11.

9. Что получится в результате подстановки (Λ, ма) в слово «мапа»?
a) мамапа;
b) па;
c) мапама;
d) мапа.

10. Нормальный алгоритм определяет схема:
$$\begin{cases} a \to \cdot\Lambda \\ b \to b \end{cases}$$

Что получится в результате применения данного алгоритма к слову *abbbb*?

a) *bbbb*;
b) *aaaaa*;
c) *abbbbb*;
d) алгоритм неприменим к данному слову.

11. Операция над упорядоченной парой слов (P, Q) называется:

a) марковской подстановкой;
b) результатом применения марковской подстановки;
c) схемой нормального алгоритма.

12. Функция f, заданная на некотором множестве слов алфавита A, называется нормально вычислимой, если:

a) найдется такое расширение B данного алфавита A $(A \subseteq B)$ и такой нормальный алгоритм в B, что каждое слово P в алфавите A из области определения функции f этот алгоритм перерабатывает в слово $f(P)$;
b) найдется такой нормальный алгоритм в A, что каждое слово P в алфавите A из области определения функции f этот алгоритм перерабатывает в слово $f(P)$;
c) существует схема нормального алгоритма, вычисляющая эту функцию.

13. Верно ли, что функции $f(x) = x + 1$ и $\varphi_3(x)$ нормально вычислимы?

a) да;
b) нет.

14. Какое из высказываний не является истинным?

a) проблема определения общерекурсивности алгоритмов разрешима;
b) универсальный алгоритм существует;
c) не существует общего алгоритма для отладки программ;
d) частный случай алгоритмически неразрешимой проблемы разрешим.

Глава V

Машина с неограниченными регистрами

В данной главе излагается алгоритмическая модель, которая в наибольшей степени близка к современным компьютерам — машины с неограниченными регистрами [5]. Методика изложения аналогична принятой в этой книге и для других моделей.

В первом пункте даются все необходимые определения и показывается, что предложенная модель удовлетворяет требованиям к алгоритмическим моделям, изложенным в первой главе.

Во втором пункте вводится центральное понятие — МНР-вычислимые функции. На его основе строится понятие разрешимого предиката, что в дальнейшем дает возможность построить различные алгоритмические конструкции. Далее показывается, как с помощью кодирования можно распространить введенные понятия и на другие виды объектов, а не только на натуральные числа.

В следующем пункте показано, что операторы соединения, подстановки, рекурсии, минимизации, развилки и повторения, будучи примененными к МНР-вычислимым функциям, снова порождают МНР-вычислимые функции. Отсюда и из теоремы Бома—Джакопини, в частности, следует, что любая программа для компьютера может быть реализована с помощью МНР.

В последнем пункте сформулирован аналог тезисов Тьюринга, Чёрча, Маркова — тезис Чёрча для МНР.

5.1. Основные определения

Машина с неограниченными регистрами (МНР) состоит из:

1) ленты, содержащей бесконечное число регистров, обозначаемых через R_1, R_2, R_3, ..., каждый из которых в любой момент времени содержит некоторое нату-

ральное число. Число, содержащееся в R_n, мы будем обозначать через r_n;

2) программы P, состоящей из конечного списка команд. Команды бывают следующих четырех видов:
   a) команда обнуления $Z(n)$ — заставляет МНР заменить содержание R_n на 0; обозначается также $0 \to R_n$ или $r_n := 0$ (читается как r_n присваивается 0);
   b) команда прибавления единицы $S(n)$ — заставляет МНР увеличивать содержимое R_n на 1; обозначается также $r_n + 1 \to R_n$ или $r_n := r_n + 1$ (r_n присваивается $r_n + 1$);
   c) команда переадресации $T(m, n)$ — заставляет МНР заменить содержимое R_n числом r_m, содержащимся в R_m; обозначается также $r_m \to R_n$ или $r_n := r_m$ (r_n присваивается r_m);
   d) команда условного перехода $J(m, n, q)$ — заставляет МНР сравнивать содержимое регистров R_m и R_n, далее, если $r_m = r_n$, то МНР переходит к выполнению q-й команды программы P, если $r_m \neq r_n$, то МНР переходит к выполнению команды в P, следующей за командой условного перехода. Если условный переход невозможен ввиду того, что в P меньше чем q команд, то МНР прекращает работу.

Команды обнуления, прибавления единицы и переадресации называются *арифметическими*.

Рассмотрим теперь, как проводятся вычисления, т. е. как работает МНР. Чтобы производить вычисления, МНР должна быть снабжена программой $P = I_1 I_2 \ldots I_s$ и начальной конфигурацией, т. е. последовательностью $a_1, a_2, a_3, \ldots$ натуральных чисел в регистрах $R_1, R_2, R_3, \ldots$. МНР начинает работу с выполнения команды I_1. Пусть в некоторый момент вычисления МНР выполняет команду I_k. Тогда после ее выполнения МНР переходит к следующей команде в вычислении, определяемой так:

- если I_k есть арифметическая команда, то следующей командой в вычислении будет I_{k+1};
- если $I_k = J(m, n, q)$, то следующей командой в вычислении будет I_q, если $r_m = r_n$, и I_{k+1} в противном случае (r_m и r_n — текущее содержимое регистров R_m и R_n соответственно).

МНР продолжает работать таким образом так долго, как это окажется возможным. Вычисление останавливается тогда

и только тогда, когда нет следующей команды, т. е. когда МНР только что выполнила команду I_k, и следующая команда в вычислении есть I_v, где $v > s$. Это может произойти в следующих случаях:

a) если $k = s$ (выполнена последняя команда в P) и I_s — арифметическая команда;
b) если $I_k = J(m, n, q)$, $r_m = r_n$ и $q > s$;
c) если $I_k = J(m, n, q)$, $r_m \neq r_n$ и $k = s$.

Мы скажем тогда, что вычисление остановилось после выполнения команды I_k; *заключительная конфигурация* — это последовательность значений $r_1, r_2, \ldots, r_n$ содержимого регистров на этом шаге.

Пример 1. Рассмотрим следующую программу:
I_1 $J(1, 2, 6)$
I_2 $S(2)$
I_3 $S(3)$
I_4 $J(1, 2, 6)$
I_5 $J(1, 1, 2)$
I_6 $T(3, 1)$

Представим вычисления на МНР с такой программой при начальной конфигурации

R_1	R_2	R_3	R_4	R_5	
9	7	0	0	0	...

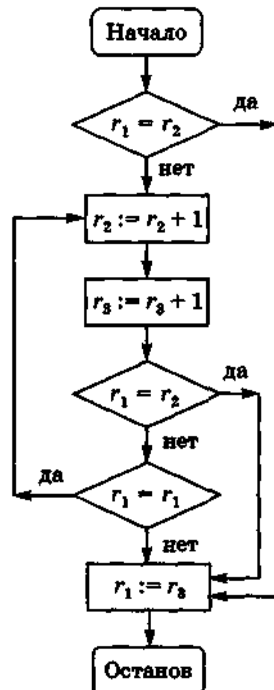
Рассмотрим процесс работы МНР по заданной программе, не задумываясь о том, какую функцию фактически она вычисляет. Ход вычисления можно представить, записывая последовательно сверху вниз конфигурации машины вместе со следующей командой, к которой она переходит на данном шаге.

	R_1	R_2	R_3	R_4	R_5	...	Следующая команда
Начальная конфигурация	9	7	0	0	0		I_1
	9	7	0	0	0		I_2 (т. к. $r_1 \neq r_2$)
	9	8	0	0	0		I_3
	9	8	1	0	0		I_4
	9	8	1	0	0		I_5 (т. к. $r_1 \neq r_2$)
	9	8	1	0	0		I_2 (т. к. $r_1 = r_1$)
	9	9	1	0	0		I_3
	9	9	2	0	0		I_4
	9	9	2	0	0		I_6 (т. к. $r_1 = r_2$)
Заключительная конфигурация	2	9	2	0	0		I_7: останов

Здесь вычисления прекращаются, поскольку в программе нет седьмой команды. △

Программу и ход вычисления удобно представить неформально с помощью диаграммы переходов или блок-схемы. Примем соглашение, что арифметические команды помещаются в прямоугольники, а условие проверки — в ромб.

Пример 2. Приведем блок-схему программы из примера 1.



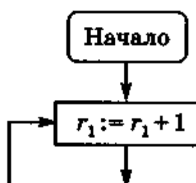
△

Обратим внимание на пятую команду. Ясно, что всегда $r_1 = r_1$, т. е. предложенная конструкция реализует *безусловный переход* на команду 2.

Составление блок-схемы полезно при написании программы в качестве промежуточного шага; дальнейшее преобразование блок-схемы в программу оказывается обычным шаблонным делом.

Бывают, конечно, вычисления, которые никогда не заканчиваются.

Пример 3. Никогда не заканчивается никакое вычисление по программе: $S(1)$, $J(1, 1, 1)$. Это вычисление представлено следующей блок-схемой:



△

Встречаются достаточно сложные примеры, когда вычисление может продолжаться бесконечно долго, но это всегда обусловлено повторением цикла при выполнении программы.

Пример 4. Вычисление по программе из примера 1 с начальной конфигурацией 2, 3, 0, 0 ... никогда не останавливается. △

Рассмотрим теперь, как выполняются основные требования к алгоритмам, изложенные в пункте 1.2, в построенной нами конкретной алгоритмической модели — МНР.

Данные МНР — это последовательности натуральных чисел, записанные на ленте. На ленту записываются исходные, промежуточные данные и окончательные результаты.

Память МНР — это лента (внешняя память) и конечное множество команд (внутренняя память). Лента бесконечна в одну сторону, однако в большинстве случаев в начальный момент времени содержимое только конечного числа регистров отлично от нуля. Из характера работы МНР следует, что в этом случае в любой последующий момент времени лишь конечный отрезок ленты будет заполнен ненулевыми числами. Поэтому важна не фактическая (как говорят в математике, актуальная) бесконечность ленты, а ее неограниченность, т. е. возможность писать на ней сколь угодно длинные, но конечные последовательности.

Элементарные шаги машины — это какая-то арифметическая команда и переход к выполнению следующей команды или (по команде условного перехода) сравнение содержимого двух регистров и соответствующий переход к выполнению необходимой команды.

Детерминированность МНР, т. е. последовательность ее шагов, определяется следующим образом. Для любой команды I_k и содержимого регистров r_1, r_2, r_3, ... однозначно заданы:

a) следующая к выполнению команда;

b) последовательность r_1', r_2', r_3', ..., которую нужно записать на ленту вместо r_1, r_2, r_3, ... (на самом деле может изменяться содержимое не более чем одного регистра).

Результатом работы МНР является заключительная конфигурация после остановки вычисления. Случай, когда МНР не останавливается, вернее, качественное объяснение этого факта будет дано ниже. Но уже сейчас интуитивно понятно, что в этом случае построенная МНР не приводит к искомому результату.

Таким образом, построенная нами МНР — это конкретная алгоритмическая модель, которая может претендовать на право считаться формализацией понятия алгоритм.

Введем еще некоторые обозначения. Пусть $a_1, a_2, a_3, \ldots$ — бесконечная последовательность элементов из $\mathbb{N}$, а P — программа.

Запись $P(a_1, a_2, a_3, \ldots)$ обозначает вычисление по программе P с начальной конфигурацией $a_1, a_2, a_3, \ldots$.

$P(a_1, a_2, a_3, \ldots)\downarrow$ означает, что вычисление $P(a_1, a_2, a_3, \ldots)$ в конце концов останавливается.

$P(a_1, a_2, a_3, \ldots)\uparrow$ означает, что вычисление $P(a_1, a_2, a_3, \ldots)$ никогда не останавливается.

Как мы уже отмечали, в большинстве случаев в начальных конфигурациях все a_i, кроме конечного их числа, равны 0. Поэтому удобным оказывается следующее обозначение. Пусть $a_1, a_2, \ldots, a_n$ — конечная последовательность натуральных чисел. Мы записываем:

$P(a_1, a_2, \ldots, a_n)$ — вычисление $P(a_1, a_2, \ldots, a_n, 0, 0, \ldots)$;

$P(a_1, a_2, \ldots, a_n)\downarrow$ означает, что $P(a_1, a_2, \ldots, a_n, 0, 0, \ldots)\downarrow$;

$P(a_1, a_2, \ldots, a_n)\uparrow$ означает, что $P(a_1, a_2, \ldots, a_n, 0, 0, \ldots)\uparrow$.

Если вычисление останавливается, то говорят, что оно сходится; а вычисление, которое никогда не останавливается, — расходится.

5.2. МНР-вычислимые функции

Предположим, что f — функция из $\mathbb{N}^n$ в $\mathbb{N}$ ($n \geq 1$). Что означает выражение: «функция f вычислима на МНР»? Естественно представить себе ответ в терминах вычисления значения $f(a_1, \ldots, a_n)$ по программе P с начальной конфигурацией $a_1, a_2, \ldots, a_n, 0, 0, \ldots$, т. е. мы рассматриваем вычисление вида $P(a_1, a_2, \ldots, a_n)$. При этом если $f(a_1, a_2, \ldots, a_n) = b$, то $P(a_1, a_2, \ldots, a_n)\downarrow$ и в каком-то регистре, например в R_1, должно находиться b, т. е. $r_1 = b$ в заключительной конфигурации. Если же при каком-то наборе $a_1, a_2, \ldots, a_n$

$f(a_1, a_2, \ldots, a_n)$ не определена, то естественно потребовать, чтобы $P(a_1, a_2, \ldots, a_n)\uparrow$. Таким образом мы приходим к следующим определениям.

Пусть f — частичная функция из $\mathbb{N}^n$ в $\mathbb{N}$. Предположим, что P — программа, а $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$.

Вычисление $P(a_1, a_2, \ldots, a_n)$ *сходится* к b, если $P(a_1, a_2, \ldots, a_n)\downarrow$ и в заключительной конфигурации в регистре R_1 находится b; это записывается как $P(a_1, a_2, \ldots, a_n)\downarrow b$.

МНР с программой P *МНР-вычисляет* f, если для всех $a_1, a_2, \ldots, a_n, b$ имеет место $P(a_1, a_2, \ldots, a_n)\downarrow b$ тогда и только тогда, когда $(a_1, a_2, \ldots, a_n) \in Dom(f)$ и $f(a_1, a_2, \ldots, a_n) = b$ (это, в частности, означает, что $P(a_1, a_2, \ldots, a_n)\downarrow$ тогда и только тогда, когда $(a_1, a_2, \ldots, a_n) \in Dom(f)$).

Функция f называется *МНР-вычислимой*, если существует такая МНР, которая МНР-вычисляет f.

Класс МНР-вычислимых функций обозначается через b, а класс n-местных МНР-вычислимых функций — через b_n.

Пример 5. $f(x, y) = x + y$. Мы получим $x + y$, прибавляя 1 к x (используя соответствующие команды) y раз. Вычисление $x + y$ начинается с начальной конфигурации $x, y, 0, 0, \ldots$; наша программа продолжает прибавление 1 к r_1, используя R_3 как счетчик числа прибавлений 1 к r_1. Типичной конфигурацией в процессе вычисления является:

R_1	R_2	R_3	R_4	R_5	
$x + k$	y	k	0	0	...

Программа спроектирована на останов, когда $k = y$, при этом в регистре R_1 оказывается число $x + y$, что и требовалось. Итак, нашу задачу решает МНР со следующей программой:

I_1 $J(3, 2, 5)$
I_2 $S(1)$
I_3 $S(3)$
I_4 $J(1, 1, 1)$

Таким образом, функция $f(x, y) = x + y$ оказывается МНР-вычислимой. △

Пример 6.

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

Так как мы ограничиваемся рассмотрением функций из $\mathbb{N}$ в $\mathbb{N}$, то это наилучшая аппроксимация функции $x - 1$. Предлагается следующая процедура. Для данной начальной конфигурации x, 0, 0, ... сначала проверить, верно ли, что $x = 0$; если да, то остановиться, в противном случае изменить содержание двух счетчиков, содержащих k и $k + 1$, начиная с $k = 0$. Типичной конфигурацией здесь будет:

R_1	R_2	R_3	R_4	R_5	
x	k	$k+1$	0	0	...

Проверяем, верно ли, что $x = k+1$; если да, то результатом вычисления служит k; в противном случае увеличиваем содержимое обоих счетчиков на единицу и проверяем снова. Итак, нашу задачу решает МНР со следующей программой:

I_1 $J(1, 4, 8)$
I_2 $S(3)$
I_3 $J(1, 3, 7)$
I_4 $S(2)$
I_5 $S(3)$
I_6 $J(1, 1, 3)$
I_7 $T(2, 1)$

Таким образом, функции $f(x) = x \dot{-} 1$ МНР-вычислима.

△

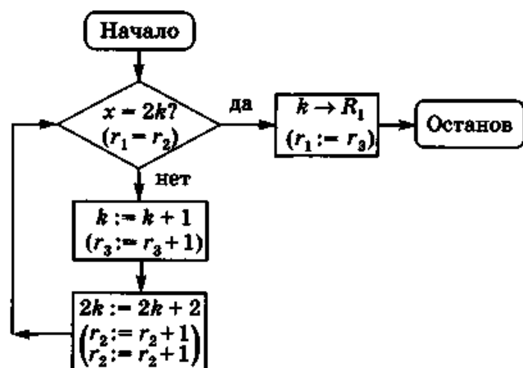
Пример 7.

$$f(x) = \begin{cases} \frac{1}{2}x, & \text{если } x \text{ четно,} \\ \text{не определена,} & \text{если } x \text{ нечетно.} \end{cases}$$

В этом примере $Dom(f)$ — это множество четных чисел, то есть наша МНР не должна останавливаться, если на входе дано нечетное число. Предлагается следующая процедура. Изменяется содержимое двух регистров, в одном из которых находится число k, в другом $2k$, где k пробегает ряд значений 0, 1, 2, ...; для последовательных значений k проверяем, верно ли, что $x = 2k$; если верно, то ответ — k; в противном случае увеличиваем k на единицу и повторяем предыдущую команду. Если x нечетно, то эта процедура, очевидно, никогда не закончится. Типичной конфигурацией будет

R_1	R_2	R_3	R_4	R_5	
x	$2k$	k	0	0	...

с исходным значением $k = 0$. Ниже представлена блок-схема этой процедуры и программа МНР, которая решает нашу задачу.



I_1 $J(1, 2, 6)$
I_2 $S(3)$
I_3 $S(2)$
I_4 $S(2)$
I_5 $J(1, 1, 1)$
I_6 $T(3, 1)$

Следовательно, функция $f(x)$ МНР-вычислима. △

Для каждой программы P и $n \geq 1$ результатом применения программы P к начальным конфигурациям вида a_1, a_2, ..., a_n, ... является единственная n-местная функция, вычисляемая программой P и обозначаемая как $f_P^{(n)}$. Из этого определения очевидно, что

$$f_P^{(n)}(a_1, a_2, \ldots, a_n) = \begin{cases} \text{единственное } b \text{, такое,} \\ \quad \text{что } P(a_1, a_2, \ldots, a_n) \downarrow b, \\ \quad \text{если } P(a_1, a_2, \ldots, a_n)\downarrow, \\ \text{не определена, если } P(a_1, a_2, \ldots, a_n)\uparrow. \end{cases}$$

В дальнейшем мы рассмотрим вопрос об определении $f_P^{(n)}$ для любой программы P.

Ясно, что конкретная функция может быть вычислима многими различными МНР; например, каждую программу МНР можно изменить, добавив команды типа $T(n, n)$. В терминах введенных нами обозначений: могут быть различные программы P_1 и P_2 с $f_{P_1}^{(n)} = f_{P_2}^{(n)}$ для некоторых (или всех) n. Позднее мы рассмотрим важную проблему, состоящую в выяснении того, вычисляют ли две программы одну и ту же функцию.

Предположим, что $M(x_1, x_2, \ldots, x_n)$ есть n-местный предикат на натуральных числах.

Характеристической функцией этого предиката $C_M(\boldsymbol{x})$ (здесь $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$) называется функция

$$C_M(\boldsymbol{x}) = \begin{cases} 1, \text{ если } M(\boldsymbol{x}) \text{ истинно,} \\ 0, \text{ если } M(\boldsymbol{x}) \text{ ложно.} \end{cases}$$

Предикат $M(\boldsymbol{x})$ *разрешим*, если функция C_M вычислима; $M(\boldsymbol{x})$ неразрешим, если функция C_M невычислима.

Пример 8. Следующие предикаты разрешимы:

a) «$x \neq y$».

Характеристическая функция определяется так:

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

Эту функцию вычисляет МНР со следующей программой:

I_1 $J(1, 2, 5)$
I_2 $Z(1)$
I_3 $S(1)$
I_4 $J(1, 1, 6)$
I_5 $Z(1)$

b) «$x = 0$».

Характеристическая функция определяется так:

$$g(x) = \begin{cases} 1, \text{ если } x = 0, \\ 0, \text{ если } x \neq 0. \end{cases}$$

Функцию $g(x)$ вычисляет МНР со следующей программой:

I_1 $J(1, 2, 3)$
I_2 $J(1, 1, 4)$
I_3 $S(2)$
I_4 $T(2, 1)$ △

Заметим, что, говоря о разрешимости (или неразрешимости), мы всюду имеем дело с вычислимостью (или невычислимостью) тотальных (т. е. всюду определенных) функций.

В контексте разрешимости свойства или предикаты иногда называются *проблемами*. Так, мы можем сказать, что проблема «$x \neq y$» разрешима. В дальнейшем мы займемся неразрешимыми проблемами.

Поскольку МНР работает только с натуральными числами, наше определение вычислимости и разрешимости применимо только к функциям и предикатам от натуральных чисел. Эти понятия легко распространяются и на другие виды объектов (целые числа, полиномы, матрицы и т. д.) с помощью кодирования.

Кодированием области D объектов называется биективное отображение α: $D \to \mathbb{N}$. Мы будем говорить, что объект $d \in D$ кодируется натуральным числом $\alpha(d)$.

Предположим, что $f\colon D \to D$; тогда f кодируется функцией $f^*\colon \mathbb{N} \to \mathbb{N}$, которая отображает код каждого объекта $d \in Dom(f)$ в код объекта $f(d)$. В явном виде это можно записать как $f^* = \alpha \circ f \circ \alpha^{-1}$.

Теперь можно распространить определение МНР-вычислимости на область D, полагая функцию f МНР-вычислимой, если f^* — МНР-вычислимая функция натуральных чисел.

Пример 9. Рассмотрим область $\mathbb{Z}$. Кодирование можно задать функцией α, где

$$\alpha(n) = \begin{cases} 2n, & \text{если } n \geq 0, \\ -2n - 1, & \text{если } n < 0. \end{cases}$$

Тогда α^{-1} задается так:

$$\alpha^{-1}(m) = \begin{cases} \frac{1}{2} m, & \text{если } m \text{ четно}, \\ -\frac{1}{2}(m+1), & \text{если } m \text{ нечетно}. \end{cases}$$

Теперь рассмотрим функцию $f(x) = x - 1$ на $\mathbb{Z}$; получаем $f^*\colon \mathbb{N} \to \mathbb{N}$, задаваемую так:

$$f^*(x) = \begin{cases} 1, & \text{если } x = 0, \\ x - 2, & \text{если } x \text{ четно (т. е. } x = \alpha(n), n > 0), \\ x + 2, & \text{если } x \text{ нечетно (т. е. } x = \alpha(n), n < 0). \end{cases}$$

Конструирование МНР, которая вычисляет f^*, является рутинным упражнением. Итак, $f(x) = x - 1$ вычислимая функция на $\mathbb{Z}$. △

5.3. Порождение вычислимых функций

В предыдущей главе для доказательства вычислимости функции мы в каждом конкретном случае строили соответствующую МНР. Это довольно трудоемкое и утомительное занятие.

Более перспективным выглядит следующий подход (вспомните введение рекурсивных функций!): выделить базис, т. е. некоторые функции, вычислимость которых очевидна, и методы комбинирования вычислимых функций, которые снова приводят к другим вычислимым функциям.

Очевидно, что к базису следует отнести все натуральные числа. Однако в прямом включении бесконечного множества $\mathbb{N}$ в базис нет необходимости. Достаточно одной константы 0 и функции следования $f(x) = x + 1$, чтобы получить весь натуральный ряд. Конкретные МНР очевидны:

0: программа $Z(1)$;

$x + 1$: программа $S(1)$.

Кроме того, в базис включим функцию проекции $U_i^n(x_1, \ldots, x_n) = x_i$, $n \geq 1$, $1 \leq i \leq n$. Ее вычислимость интуитивно очевидна: значение функции равно одному из значений ее аргумента. Соответствующая МНР строится достаточно просто:

U_i^n: программа $T(i, 1)$.

Итак, в базис мы включим три рассмотренные выше функции. Отметим один важный нюанс: можно показать, что команда переадресации избыточна в нашем определении МНР, т. е. для каждой команды переадресации $T(m, n)$ существует МНР без команды переадресации, которая для всякой конфигурации дает тот же результат, что и $T(m, n)$. Но функцию U_i^n исключить из базиса нельзя!

Перейдем теперь к рассмотрению методов комбинирования вычислимых функций.

5.3.1. Соединение программ

Рассмотрим некоторые технические средства, позволяющие строить программы, состоящие из нескольких других программ или подпрограмм.

Самый простой пример построения программы из нескольких — это соединение программ, состоящее в следующем: имеются программы P и Q и мы хотим написать программу, которая бы сначала выполняла P, а затем Q. Интуитивно хочется просто написать под командами программы P команды программы Q. Но тут возникают два технических момента: по управлению (управление должно передаваться от программы P точно к первой команде программы Q) и по памяти (программа Q должна использовать регистры, не затронутые вычислениями по программе P).

Решим сначала вопрос по управлению.

Пусть $P = I_1, I_2, \ldots, I_s$. Вычисление по программе P завершается, когда следующая команда в вычислении есть I_v,

для конечного $v > s$; в этом случае в нашей составной программе необходим переход к первой команде Q. Это произойдет автоматически, если $v = s + 1$. Поэтому потребуем, чтобы программа P останавливалась только тогда, когда следующей командой является I_{s+1}. Очевидно, что нарушение этого условия могут вызвать лишь команды условного перехода. Так мы и приходим к следующему определению.

Программа $P = I_1, I_2, \ldots, I_s$ имеет *стандартный вид*, если для всякой команды условного перехода $J(m, n, q)$ в P имеем $q \leq s + 1$.

Пример 10. Примеры 1, 3, 5–8 содержат программы, имеющие стандартный вид. △

Теперь ответим на вопрос о том, всякую ли программу можно привести к стандартному виду.

Теорема 1. Для любой программы P существует программа P^* стандартного вида, такая, что всякое вычисление по программе P^* идентично соответствующему вычислению по программе P за возможным исключением способа остановки. В частности, для любых $a_1, a_2, \ldots, a_n$ $P(a_1, a_2, \ldots, a_n){\downarrow}b$ тогда и только тогда, когда $P^*(a_1, a_2, \ldots, a_n){\downarrow}b$ и, следовательно, $f_P^{(n)} = f_{P^*}^{(n)}$ для всех $n > 0$.

Доказательство. Пусть $P = I_1, I_2, \ldots, I_s$. Чтобы получить из P программу P^*, достаточно просто изменить команды условного перехода так, чтобы все остановы происходили после условных переходов к команде I_{s+1}. Положим $P^* = I_1^*, I_2^*, \ldots, I_s^*$, где, если I_k не является командой условного перехода, то $I_k^* = I_k$, а если $I_k = J(m, n, q)$, то

$$I_k^* = \begin{cases} I_k, \text{ если } q \leq s+1, \\ J(m, n, s+1), \text{ если } q > s+1. \end{cases}$$

Очевидно, что P^* удовлетворяет требованиям теоремы. □

Пусть теперь программы P и Q имеют стандартный вид. Рассмотрим команды условного перехода в Q. Так как q-я команда в Q будет $(s + q)$-й командой в составной программе, то команда $J(m, n, q)$ в Q должна быть заменена на переход $J(m, n, s + q)$ в составной программе, чтобы сохранить смысл программы.

Теперь можно определить соединение или конкатенацию двух программ стандартного вида.

Пусть P и Q — программы стандартного вида длины s и t соответственно. *Соединением* или *конкатенацией* P и Q (в записи PQ) называется программа $I_1, I_2, \ldots, I_s, I_{s+1}, \ldots, I_{s+t}$, где $P = I_1, I_2, \ldots, I_s$, а команды $I_{s+1}, \ldots, I_{s+t}$ суть команды программы Q, у которой каждый условный переход $J(m, n, q)$ заменен на $J(m, n, s + q)$.

Очевидно, что при таком определении результат действия PQ будет тот, который нам необходим: каждое вычисление по программе PQ совпадает с соответствующим вычислением по программе P, за которым следует вычисление по программе Q, начатое в заключительной конфигурации программы P.

Перейдем к рассмотрению вопроса о памяти.

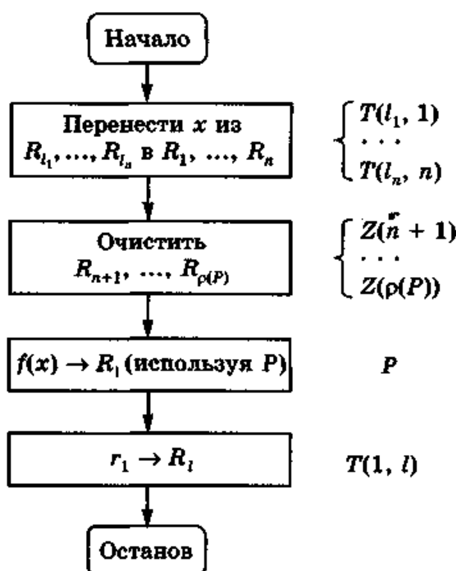
При написании программы Q часто бывает важно найти такие регистры, которые не затронуты вычислением по программе P. Особенно это необходимо, когда Q содержит P в качестве подпрограммы.

Так как программа P конечна, найдется наименьшее число u такое, что ни один регистр R_v, $v > u$, не затронут вычислением по программе P, т. е. если $Z(n)$, или $S(n)$, или $T(m, n)$, или $J(m, n, q)$ является командой P, то $m, n \le u$. То есть на протяжении всего вычисления по программе P содержимое регистров R_v, $v > u$, остается без изменений и не оказывает никакого воздействия на $r_1, \ldots, r_u$. Тогда при составлении новой программы Q можно использовать регистры R_v, $v > u$. Обозначим это число u через $\rho(P)$.

Введем одно обозначение, которое сильно упростит доказательства в этой главе. Пусть P — программа стандартного вида, вычисляющая функцию $f(x_1, \ldots, x_n)$. Когда P используется как подпрограмма большой программы, то

- входы $x_1, \ldots, x_n$ записываются в регистрах $R_{l_1}, \ldots, R_{l_n}$, а не в $R_1, \ldots, R_n$, как требуется в программе P;
- выход $f(x_1, \ldots, x_n)$, который потребуется в дальнейшем, помещается в регистр R_l, а не в R_1, как было условлено;
- рабочие регистры $R_{n+1}, \ldots, R_{\rho(P)}$ для P могут содержать разного рода нежелательную информацию.

Можно модифицировать P, принимая во внимание все эти детали.



Эту программу обозначим через $P[l_1, \ldots, l_n \to l]$. Она вычисляет $f(r_{l_1}, \ldots, r_{l_n})$ и помещает результат в R_l. Кроме того, программа затрагивает только регистры $R_1, \ldots, R_{\rho(P)}$ и R_l.

5.3.2. Подстановка

Обычный способ построения новых функций состоит в подстановке одних функций вместо аргументов в другие, иначе он известен под названием композиция или суперпозиция функций.

Сейчас мы покажем, что, применяя подстановку к вычислимым функциям, мы получаем снова вычислимые функции, т. е. класс b замкнут относительно операции подстановки.

Теорема 2. Пусть $f(y_1, \ldots, y_k)$, $g_1(x), \ldots, g_k(x)$, где $x = (x_1, \ldots, x_n)$, суть вычислимые функции. Тогда если $(g_1(x), \ldots, g_k(x)) \in Dom(f)$, то функция $h(x) = f(g_1(x), \ldots, g_k(x))$ вычислима.

Доказательство. Пусть программы $F, G_1, \ldots, G_k$ стандартного вида вычисляют функции $f, g_1, \ldots, g_k$ соответственно. Напишем программу H, которая осуществляет следующую естественную процедуру для вычисления функции h: «для данного x, используя программы $G_1, \ldots, G_k$, вычисляйте

последовательно $g_1(x)$, ..., $g_k(x)$, записывая их значения в порядке их вычисления. Затем примените программу F для вычисления $f(g_1(x), ..., g_k(x))$».

Осталось разобраться с памятью в том смысле, чтобы избежать потерь информации, которая потребуется на более поздних шагах, а именно x и те значения $g_i(x)$, которые уже получены. Полагая $m = \max(n, k, \rho(F), \rho(G_1), ..., \rho(G_k))$, мы начинаем с запоминания x в R_{m+1}, ..., R_{m+n}; регистры R_{m+n+1}, ..., R_{m+n+k} будут использованы для запоминания значений $g_i(x)$, когда они будут вычислены при $i = 1, ..., k$. Эти регистры совершенно не затрагиваются вычислениями по программам F, G_1, ..., G_k. Типичной конфигурацией при вычислении по программе H будет следующая:

$R_1 ... R_m$	$R_{m+1} ... R_{m+n}$	R_{m+n+1}	R_{m+n+2}	...	R_{m+n+k}			
0	x	$g_1(x)$	$g_2(x)$	...	$g_k(x)$	0	0	...

Тогда МНР, вычисляющая h, содержит следующую программу H:

$T(1, m+1)$
...
$T(n, m+n)$
$G_1[m+1, m+2, ..., m+n \to m+n+1]$
...
$G_k[m+1, m+2, ..., m+n \to m+n+k]$
$F[m+n+1, ..., m+n+k \to 1]$

Очевидно, что вычисление $H(x)$ будет заканчиваться тогда и только тогда, когда заканчивается каждое из вычислений $G_i(x)$, $1 \le i \le k$, и вычисление $F(g_1(x), ..., g_k(x))$. □

Новые функции можно получить перестановкой и отождествлением переменных или добавлением новых фиктивных переменных. Например, из функции $f(y_1, y_2)$ можно получить:

$h_1(x_1, x_2) = f(x_2, x_1)$ — перестановка,

$h_2(x) = f(x, x)$ — отождествление,

$h_3(x_1, x_2, x_3) = f(x_2, x_3)$ — добавление фиктивных переменных.

Покажем, что каждая из этих операций или их комбинаций преобразует вычислимые функции в вычислимые же функции.

Теорема 3. Пусть $f(y_1, \ldots, y_k)$ — вычислимая функция и $x_{i_1}, \ldots, x_{i_k}$ — последовательность из k переменных $x_1, \ldots, x_n$ (возможно с повторениями). Тогда функция $h(x_1, \ldots, x_n) = f(x_{i_1}, \ldots, x_{i_k})$ вычислима.

Доказательство. Полагая $\mathbf{x} = (x_1, \ldots, x_n)$, получаем функцию $h(\mathbf{x}) = f(U_{i_1}^n(\mathbf{x}), \ldots, U_{i_k}^n(\mathbf{x}))$. Так как U_i^n вычислимы, то по теореме 2 получаем, что $h(\mathbf{x})$ вычислима. □

Пример 11. Функция $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ вычислима. Это следует из того, что вычислима функция $x + y$ (пример из предыдущей главы), в которую подставляем $x_1 + x_2$ вместо x и x_3 вместо y. △

5.3.3. Рекурсия

Следующий способ построения новых функций состоит в определении каждого из ее значений в терминах ранее определенных значений и, возможно, использовании других уже определенных функций. Он нам известен под названием рекурсия. Напомним необходимое определение.

Пусть $f(\mathbf{x})$ и $g(\mathbf{x}, y, z)$, $\mathbf{x} = (x_1, \ldots, x_n)$, суть некоторые функции. Говорят, что функция $h(\mathbf{x}, y)$ определена с помощью *примитивной рекурсии* из функций f и g, если она задается следующим образом:

$h(\mathbf{x}, 0) = f(\mathbf{x})$;

$h(\mathbf{x}, y + 1) = g(\mathbf{x}, y, h(\mathbf{x}, y))$.

Легко видеть, что это определение корректно в том смысле, что мы можем найти любое нужное нам значение функции h. Действительно:

$h(\mathbf{x}, 0) = f(\mathbf{x})$,

$h(\mathbf{x}, 1) = g(\mathbf{x}, 0, h(\mathbf{x}, 0)) = g(\mathbf{x}, 0, f(\mathbf{x}))$,

$h(\mathbf{x}, 2) = g(\mathbf{x}, 1, h(\mathbf{x}, 1)) = g(\mathbf{x}, 1, g(\mathbf{x}, 0, f(\mathbf{x})))$,

$h(\mathbf{x}, 3) = g(\mathbf{x}, 2, h(\mathbf{x}, 2)) = g(\mathbf{x}, 1, g(\mathbf{x}, 1, g(\mathbf{x}, 0, f(\mathbf{x}))))$

и т. д.

Когда $n = 0$, т. е. параметры $\mathbf{x}$ не представлены, уравнения примитивной рекурсии принимают вид

$h(0) = a$, где $a \in \mathbb{N}$,

$h(y + 1) = g(y, h(y))$.

Покажем теперь, что класс b замкнут относительно операции примитивной рекурсии.

Теорема 4. Пусть $f(x)$ и $g(x, y, z)$ — вычислимые функции, $x = (x_1, ..., x_n)$; тогда функция $h(x, y)$, полученная из f и g с помощью примитивной рекурсии, вычислима.

Доказательство. Пусть F и G — программы стандартного вида, которые вычисляют функции $f(x)$ и $g(x, y, z)$. Построим программу H для функции $h(x, y)$, определенной примитивной рекурсией из f и g, следующим образом: при заданной начальной конфигурации $x_1, ..., x_n, y, 0, 0, ...$ программа H сначала вычисляет $h(x, 0)$ (применяя F), а затем если $y \neq 0$, то H применяет (многократно!) программу G для последовательного вычисления $h(x, 1), h(x, 2), ..., h(x, y)$ и затем останавливается.

Теперь разберемся с памятью. Пусть $m = \max(n + 2, \rho(F), \rho(G))$. Начнем с запоминания x, y в регистрах $R_{m+1}, ..., R_{m+n+1}$. Следующие два регистра используются, чтобы запоминать текущие значения чисел k и $h(x, k)$ для $k = 0, 1, ..., y$. Обозначая сумму $m + n$ через t, рассмотрим типичную конфигурацию процесса вычисления:

$R_1 \dots R_m$	$R_{m+1} \dots R_t$	R_{t+1}	R_{t+2}	R_{t+3}			
0	x	y	k	$h(x, k)$	0	0	...

Тогда МНР, вычисляющая h, содержит следующую программу H:

$T(1, m + 1)$
...
$T(n, m + n)$
$T(n + 1, m + n + 1)$
$F[1, 2, ..., n \to t + 3]$
I_q $J(t + 2, t + 1, p)$
$G[m + 1, m + 2, ..., m + n, t + 2, t + 3 \to t + 3]$
$S(t + 2)$
$J(1, 1, q)$
I_p $T(t + 3, 1)$.

Следовательно, функция h вычислима. □

В главе III приведены многочисленные примеры примитивно-рекурсивных функций. Очевидно, что все они, по доказанной теореме, МНР-вычислимы.

5.3.4. Минимизация

Существует третья уже знакомая нам операция, порождающая новые вычислимые функции — операция минимизации.

Пусть $f(x, y)$ — функция (не обязательно тотальная), и мы хотим определить функцию $g(x)$, положив

$g(x)$ = наименьшее y, такое, что $f(x, y) = 0$,

таким образом, чтобы из вычислимости f следовала бы вычислимость g.

Тут надо решить два вопроса. Во-первых, для некоторого x может не найтись y, для которого $f(x, y) = 0$. Во-вторых, предположим, что f вычислима, и рассмотрим следующий естественный алгоритм вычисления $g(x)$: «вычисляйте $f(x, 0)$, $f(x, 1)$, ... до тех пор, пока не найдете y, такой, что $f(x, y) = 0$». Эта процедура может не завершиться, если f не тотальна, даже если такой y существует; например, $f(x, 0)$ не определено, а $f(x, 1) = 0$.

Таким образом, мы приходим к следующему определению оператора минимизации μ, который дает вычислимые функции из вычислимых функций.

Для каждой функции $f(x, y)$

$$\mu y(f(x,y)=0) = \begin{cases} \text{наименьший } y \text{ такой, что} \\ \quad \text{a) } f(x,z) \text{ определено для всех } z \le y \text{ и} \\ \quad \text{b) } f(x,y) = 0, \\ \text{если такой } y \text{ существует;} \\ \\ \text{не определено в противном случае.} \end{cases}$$

$\mu y(\ldots)$ читается как «наименьший y, такой, что ...». Этот оператор называют μ-оператором.

Следующая теорема показывает, что класс b замкнут относительно операции минимизации.

Теорема 5. Пусть функция $f(x, y)$ вычислима. Тогда вычислима и функция $g(x) = \mu y(f(x, y) = 0)$.

Доказательство. Пусть $x = (x_1, \ldots, x_n)$ и F — программа стандартного вида, вычисляющая функцию $f(x, y)$. Пусть $m = \max(n + 1, \rho(F))$. Напишем программу G, которая реализует естественный алгоритм для g: «вычисляйте $f(x, k)$ для $k = 0, 1, 2, \ldots$ до тех пор, пока не найдете такое k, что $f(x, k) = 0$». Это значение k и будет требуемым выходом.

Значение x и текущее значение k запоминаются в регистрах $R_{m+1}, \ldots, R_{m+n+1}$ до вычисления $f(x, k)$; таким образом, типичной конфигурацией будет:

$R_1 \ldots R_m$	$R_{m+1} \ldots R_{m+n}$	R_{m+n+1}	R_{m+n+2}			
0	x	k	0	0	0	...

Заметим, что r_{m+n+2} всегда есть 0. Тогда МНР, вычисляющая g, содержит следующую программу:

$T(1, m + 1)$

...

$T(n, m + n)$

I_p $F[m + 1, m + 2, \ldots, m + n + 1 \to 1]$

$J(1, m + n + 2, q)$

$S(m + n + 1)$

$J(1, 1, p)$

I_q $T(m + n + 1, 1)$.

(I_p является первой командой подпрограммы $F[m + 1, m + 2, \ldots, m + n + 1 \to 1]$). □

Пример 12. Рассмотрим следующую функцию, получающуюся с помощью оператора минимизации:

$g(x, z) = \mu y(y + z - x = 0)$.

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

$y = 0,\ 2 + 0 = 2 \neq 7$;

$y = 1,\ 2 + 1 = 3 \neq 7$;

$y = 2,\ 2 + 2 = 4 \neq 7$;

$y = 3,\ 2 + 3 = 5 \neq 7$;

$y = 4,\ 2 + 4 = 6 \neq 7$;

$y = 5,\ 2 + 5 = 7$.

Таким образом, $g(7, 2) = 5$.

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

$y = 0,\ 4 + 0 = 4$;

$y = 1,\ 4 + 1 = 5$;

$y = 2,\ 4 + 2 = 6$;

...

Очевидно, что данный процесс будет продолжаться бесконечно. Следовательно, $g(3, 4)$ не определено.

Таким образом, $g(x, z) = x - z$. △

Пример 13. Аналогично с помощью оператора минимизации можно получить частичную функцию, выражающую частное от деления двух натуральных чисел:

$$\frac{x}{z} = g(x, z) = \mu y(zy - x = 0). \qquad \triangle$$

Пример 14. Пусть $f(x, y) = |x - y^2|$ и $g(x) = \mu y(f(x, y) = 0)$. Тогда

$$g(x) = \begin{cases} \sqrt{x}, \text{ если } x \text{ есть точный квадрат;} \\ \text{не определена в противном случае.} \end{cases}$$

причем $g(x)$ — нетотальная функция, порожденная из тотальной функции $f(x, y)$. △

Приведенный пример показывает, что, в отличие от подстановки и рекурсии, оператор минимизации может порождать нетотальные функции из тотальных, что существенно расширяет класс b.

5.3.5. Развилка и повторение

В первой главе говорилось, что для построения любых алгоритмов в смысле программирования для компьютеров достаточно трех функциональных блоков: следования, развилки и повторения. Выше мы показали, что класс b замкнут относительно следования. Теперь разберемся с двумя следующими блоками.

Для функций $g_1(x_1, ..., x_n)$, $g_2(x_1, ..., x_n)$ и предиката $P(x_1, ..., x_n)$ введем функцию $f(x_1, ..., x_n) = B(g_1, g_2, P)$, где

$$f(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n), \text{ если } P(x_1, \ldots, x_n) \text{ истинно,} \\ g_2(x_1, \ldots, x_n), \text{ если } P(x_1, \ldots, x_n) \text{ ложно.} \end{cases}$$

$B(g_1, g_2, P)$ называется оператором развилки или перехода.

Теорема 6. Пусть функции $g_1(x_1, ..., x_n)$, $g_2(x_1, ..., x_n)$, и предикат $P(x_1, ..., x_n)$ вычислимы; тогда вычислима и функция $f(x_1, ..., x_n) = B(g_1, g_2, P)$, т. е. класс b замкнут относительно оператора развилки.

Доказательство. Пусть $x = (x_1, ..., x_n)$ и G_1, G_2, C — программы стандартного вида, вычисляющие $g_1(x)$, $g_2(x)$ и $P(x)$ соответственно. Напишем программу, которая реализует естественный алгоритм для f: «вычислите $P(x)$, и если $r_1 = 1$ ($P(x)$ истинно), то вычислите g_1 (запустите G_1), если

же $r_1 = 0$ ($P(x)$ ложно), то вычислите g_2 (запустите G_2)». В обоих случаях искомое значение $f(x)$ будет находиться в R_1.

Обозначим $m = \max(n, \rho(G_1), \rho(G_2), \rho(C))$ и $t = m + n$. Тогда типичная конфигурация будет

$R_1...R_m$	$R_{m+1}...R_t$	R_{t+1}	R_{t+2}	R_{t+3}	
	x	$P(x)$	$f(x)$	0	...

Заметим, что r_{t+3} всегда есть 0. Тогда МНР, вычисляющая f, содержит следующую программу:

$T(1, m + 1)$
...
$T(n, m + n)$
$C[m + 1, ..., t \to t + 1]$
$J(t + 1, t + 3, q)$
$G_1[m + 1, ..., t \to t + 2]$
$J(1, 1, s)$
I_q $G_2[m + 1, ..., t \to t + 2]$
I_s $T(t + 2, 1)$

Здесь I_q — первая команда программы $G_2[m + 1, ..., t \to t + 2]$. □

Используя функции $g_1(x)$, $g_2(x)$ и предикат $P(x)$, построим функцию $h(x) = D(g_1, g_2, P)$, где $h(x)$ задана следующим образом: «пока $P(x)$ истинно, вычислять $g_1(x)$, иначе вычислить $g_2(x)$». $D(g_1, g_2, P)$ называется оператором повторения или цикла.

Теорема 7. Пусть функции $g_1(x)$, $g_2(x)$ и предикат $P(x)$ вычислимы, тогда вычислима и функция $h(x) = D(g_1, g_2, P)$, т. е. класс b замкнут относительно оператора повторения.

Доказательство. Пусть G_1, G_2, C — программы, как в теореме 6. Напишем программу, которая реализует естественный алгоритм для h: «вычислите $P(x)$, и если $P(x)$ истинно, то вычислите g_1, затем снова вычислите $P(x)$ и т. д., если же $P(x)$ ложно, то вычислите g_2». Искомое значение $f(x)$, очевидно, будет лежать в R_1.

Обозначим $m = \max(n, \rho(G_1), \rho(G_2), \rho(C))$, $t = m + n$. Тогда типичная конфигурация будет

$R_1...R_m$	$R_{m+1}...R_t$	R_{t+1}	R_{t+2}	R_{t+3}	
	x	$P(x)$	$g_1(x)$	0	...

Заметим, что r_{t+3} всегда есть 0. Тогда МНР, вычисляющая h, содержит следующую программу:

$T(1, m + 1)$
...
$T(n, t)$
$C[m + 1, ..., t \to t + 1]$
$J(t + 1, t + 3, q)$
$G_1[m + 1, ..., t \to t + 2]$
$J(1, 1, t + 1)$
I_q $G_2[m + 1, ..., t \to 1]$

где I_q — первая команда программы $G_2[m + 1, ..., t \to 1]$. □

Объединяя результаты этого пункта с замкнутостью класса b относительно следования, получаем на основании теоремы Бома—Джакопини, что любая программа (в смысле программы для компьютера) может быть реализована МНР.

5.4. Тезис Чёрча

До сих пор нам удавалось для всех процедур, претендующих на алгоритмичность, т. е. конструктивных процедур, строить реализующие их МНР. Будет ли это удаваться всегда? Утвердительный ответ на этот вопрос содержится в тезисе Чёрча, который формулируется так: ***всякий алгоритм может быть реализован МНР, или интуитивно и неформально определенный класс вычислимых частичных функций совпадает с классом b МНР-вычислимых функций.***

Во второй главе мы сделали некоторые комментарии к тезису Тьюринга. Все они справедливы и для этого тезиса Чёрча. Не повторяя их, сделаем еще ряд важных замечаний.

Подтверждением тезиса Чёрча является ряд свидетельств:

1. Фундаментальный результат: многие независимые варианты уточнения интуитивного понятия алгоритма привели к одному и тому же классу функций, которые мы обозначим через b.
2. Обширное семейство вычислимых функций, как было показано, принадлежит классу b; конкретные функции из примеров второй, третьей, четвертой и этой главы образуют исходную часть такого семейства, которое можно расширять до бесконечности методами, изложенными в них, и другими более сложными методами.

3. Реализация программы P, вычисляющей функцию на МНР, является очевидным примером алгоритма; таким образом прямо из определения класса b видно, что все функции в b вычислимы в неформальном смысле.
4. Никому не доводилось найти функцию, которую можно было бы признать вычислимой в неформальном смысле, не принадлежащую b.

Исходя из этих соображений и собственного опыта, большинство математиков приняли тезис Чёрча. В дальнейшем мы будем применять тезис Чёрча в следующем смысле.

Предположим, что мы располагаем неформально описанным алгоритмом для вычисления значения функции f. Такой алгоритм может быть описан на естественном языке, или с помощью диаграмм, или в полуформальных математических терминах, или с помощью любых других средств, однозначно определяющих, как эффективно вычислить значения f там, где они определены, за конечный отрезок времени. В такой ситуации нам может понадобиться доказательство МНР-вычислимости f. Сделать это мы можем двумя способами.

Способ 1. Написать программу, которая МНР-вычисляет функцию f (и доказать, что она действительно это делает), или косвенным образом доказать, что такая программа существует. Это можно сделать, например, применив методы этой главы. Полное и формальное доказательство того, что функция f МНР-вычислима, может быть долгим и теоретически трудным процессом.

Способ 2. Дать неформальное (хотя и строгое) доказательство того, что данный неформальный алгоритм действительно вычисляет f. Затем аппелировать к тезису Чёрча и сделать заключение об МНР-вычислимости функции f.

В дальнейшем мы будем пользоваться способом 2 как строгим и надежным методом доказательства.

Пример 15. Пусть P является МНР-программой; определим функцию f следующим образом:

$$f(x, y, t) = \begin{cases} 1, & \text{если } P(x) \downarrow y \text{ после не более} \\ & \text{чем } t \text{ шагов вычисления } P(x); \\ 0 & \text{в противном случае.} \end{cases}$$

Неформальный алгоритм вычисления f состоит в следующем: «для заданной тройки (x, y, t) моделируйте вычисления $P(x)$ (например, на листе бумаги), выполняя t шагов $P(x)$, если только это вычисление не остановится раньше. Если $P(x)$ остановится после t или менее шагов и число y будет в регистре R_1, то $f(x, y, t) = 1$. В противном случае (т. е. если $P(x)$ останавливается за t или менее шагов с числом, отличающимся от y в R_1, или если $P(x)$ не останавливается после t шагов) $f(x, y, t) = 0$».

Моделирование $P(x)$ для не более чем t шагов является, очевидно, механической процедурой, которую можно выполнить за конечное время. Таким образом, функция f эффективно вычислима. Следовательно, по тезису Чёрча f МНР-вычислима. △

Пример 16. Пусть f и g — одноместные эффективно вычислимые функции. Определим функцию h:

$$h(x) = \begin{cases} 1, \text{ если } x \in Dom(f) \text{ или } x \in Dom(g), \\ \text{не определена в противном случае.} \end{cases}$$

Алгоритм для вычисления h можно описать в терминах алгоритмов, заданных для вычисления f и g следующим образом: «при заданном x запускайте алгоритмы для вычисления $f(x)$ и $g(x)$ одновременно. (Представьте себе двух исполнителей или две машины, работающие одновременно (параллельно), или одного исполнителя, который выполняет поочередно по одному шагу каждого алгоритма.) Если хоть одно из этих вычислений остановится, остановите это вычисление в целом и положите $h(x) = 1$. В противном случае, продолжайте неограниченно долго». △

Упражнения

1. Дана программа для МНР, вычисляющая функцию $f(x, y) = x + y$. Исполнить ее для чисел 3 и 2.

R_1	R_2	R_3	R_4	R_5	
$x+k$	y	k	0	0	...

I_1 $J(3, 2, 5)$
I_2 $S(1)$
I_3 $S(3)$
I_4 $J(1, 1, 1)$

2. Дана программа для МНР, вычисляющая функцию $f(x) = \begin{cases} \frac{1}{2}x, \text{ если } x \text{ четно,} \\ \text{не определена, если } x \text{ нечетно.} \end{cases}$

Исполнить ее для числа 6.

R_1	R_2	R_3	R_4	R_5	
x	$2k$	k	0	0	...

I_1 $J(1, 2, 6)$
I_2 $S(3)$
I_3 $S(2)$
I_4 $S(2)$
I_5 $J(1, 1, 1)$
I_6 $T(3, 1)$

3. Построить МНР, вычисляющую функцию:

a) $f(x, y) = x - y$;
b) $f(x, y) = x \cdot y$;
c) $f(x, y) = x/y$;
d) $f(x) = \begin{cases} \frac{1}{3}x, \text{ если } x \text{ делится на 3,} \\ \text{не определена в противном случае;} \end{cases}$
e) $f(x) = 2x$;
f) $f(x) = x - 2$, если $x > 2$;
g) $f(x) = x + 5$;
h) $y = x \text{ div } 3$;
i) $y = ax$;
j) $y = 2x - 1$;
k) $y = ax + b$;
l) $y = x^2$;
m) $y = b - ax$;
n) $y = ax - b$;
o) $y = x^3$;
p) $y = 2^x$;
q) $y = x/a$;
r) $y = 3^x$;
s) $y = x \bmod 3$;
t) $y = a - 2x$.

4. Построить МНР, вычисляющую среднее арифметическое двух чисел.

5. Построить МНР, выполняющую проверку на нечетность.

6. Построить МНР, выполняющую проверку деления на 5.

Тестовые задания

1. Какую из команд не имеет машина с неограниченными регистрами (МНР)?
   a) цикл;
   b) условный переход;
   c) прибавление единицы;
   d) переадресация.

2. Поставить в соответствие командам МНР их назначение:
   a) $Z(n)$;
   b) $S(n)$;
   c) $T(m, n)$;
   d) $J(m, n, q)$;

   1) заменяет содержимое R_n на 0;
   2) увеличивает содержимое R_n на 1;
   3) заменяет содержимое R_m на число r_n из R_n;
   4) сравнивает содержимое R_n и R_m и осуществляет переход на команду с номером q;
   5) заменяет содержимое R_n на число r_m из R_m.

3. Если $r_m = r_n$ для команды $J(m, n, q)$, то
   a) будет осуществлено вычисление команды с номером q;
   b) будет осуществлен переход на следующую команду;
   c) нет верного ответа.

4. Какая команда МНР не относится к арифметическим командам?
   a) условный переход;
   b) обнуление;
   c) прибавление единицы;
   d) переадресация;
   e) нет верного ответа.

5. Что получится в результате выполнения следующей МНР-программы
   I_1 $J(3, 2, 5)$
   I_2 $S(1)$
   I_3 $S(3)$
   I_4 $J(1, 1, 1)$
   если в R_1 содержится число 4, а в R_2 число 2?
   a) 6;
   b) 8;
   c) 2.

6. Внутренняя память МНР — это:
   a) конечное множество состояний;
   b) лента;
   c) нет верного ответа.

7. Запись $P(a_1, a_2, \ldots, a_n)$ обозначает:
   a) что вычисление P никогда не останавливается;
   b) что вычисление P останавливается;
   c) нет верного ответа.

8. Поставить в соответствие:
   a) функция f называется МНР-вычислимой;
   b) вычисление $P(a_1, a_2, \ldots, a_n)$ сходится к b;
   c) МНР (машина с неограниченными регистрами) с программой P МНР-вычисляет f;
   1) если для всех $a_1, a_2, \ldots, a_n, b$ имеет место $P(a_1, a_2, \ldots, a_n)\downarrow b$;
   2) если $P(a_1, a_2, \ldots, a_n)\downarrow$ и в заключительной конфигурации в регистре R_1 находится b;
   3) если существует такая МНР, которая МНР-вычисляет f.

9. Какую из команд имеет машина с неограниченными регистрами (МНР)?
   a) условный переход;
   b) цикл;
   c) присваивание;
   d) нет верного ответа.

10. Какая команда МНР выполняет замену содержимого R_n на число r_m из R_m
   a) $J(m, n, q)$;
   b) $Z(n)$;
   c) $S(n)$;
   d) нет верного ответа.

11. После команды $J(m, n, q)$ будет выполняться команда с номером q, если:
   a) $r_m = r_n$;
   b) $r_m \neq r_n$;
   c) нет верного ответа.

12. Какая из команд МНР относится к арифметическим командам?
   a) прибавление единицы;
   b) условный переход;
   c) нет верного ответа.

13. В программе вычисления функции

$$f(x) = \begin{cases} x/2, \text{ если } x \text{ четно}, \\ \text{не определена, если } x \text{ нечетно} \end{cases}$$

восстановить правильную последовательность команд:
a) $J(1, 2, 6)$
b) $T(3, 1)$
c) $S(2)$
d) $S(2)$
e) $S(3)$
f) $J(1, 1, 1)$

14. Что получится в результате выполнения следующей МНР-программы
I_1 $T(1, 3)$
I_2 $S(5)$
I_3 $J(2, 5, 9)$
I_4 $Z(4)$
I_5 $J(1, 4, 2)$
I_6 $S(3)$
I_7 $S(4)$
I_8 $J(1, 1, 5)$
I_9 $T(3, 1)$
если в R_1 содержится число 6, в R_2 число 3, а в R_3, R_4 и R_5 нули?
a) 18;
b) 3;
c) 9;
d) 2.

15. Внешняя память МНР — это:
a) лента;
b) конечное множество команд;
c) конечное множество состояний.

16. Вычисление $P(a_1, a_2, \ldots, a_n)$ сходится к b, если:
a) $P(a_1, a_2, \ldots, a_n)\downarrow$ и в заключительной конфигурации в регистре R_1 находится b;
b) существует такая МНР, которая МНР-вычисляет f;
c) для всех $a_1, a_2, \ldots, a_n, b$ имеет место $P(a_1, a_2, \ldots, a_n)\downarrow b$.

17. Для любой команды I_k и содержимого регистров r_1, r_2, r_3, ... однозначно заданы:

– следующая к выполнению команда;
– последовательность r_1', r_2', r_3', ..., которую нужно записать на ленту вместо r_1, r_2, r_3, ...

Это —
a) детерминированность МНР;
b) элементарные шаги машины;
c) дискретность машины;
d) нет верного ответа.

18. Что является результатом работы МНР?
a) заключительная конфигурация после остановки вычисления;
b) какая-то команда и переход к выполнению следующей команды;
c) последовательность ее шагов;
d) нет верного ответа.

19. Поставить в соответствие:
a) данные МНР;
b) память МНР;
c) элементарные шаги машины;

1) последовательность натуральных чисел, записанная на ленте,
2) последовательность шагов машины;
3) лента и конечное множество команд;
4) какая-то арифметическая команда и переход к выполнению следующей команды.

20. Последовательность a_1, a_2, a_3, ... натуральных чисел в регистрах R_1, R_2, R_3, ... — это
a) начальная конфигурация;
b) заключительная конфигурация;
c) нет верного ответа.

Глава VI

Вычислимость и разрешимость

Если в предыдущих главах мы изложили четыре независимых подхода к построению алгоритмической модели, то результаты данной главы не зависят от конкретной модели.

В первом пункте сформулирован фундаментальный результат — все приведенные ранее алгоритмические модели эквивалентны. Это дает нам возможность всюду в дальнейшем работать просто с алгоритмами, не акцентируя внимания на его конкретной реализации. Более того, теперь в каждом конкретном случае мы будем выбирать ту модель, которая нам наиболее удобна.

В следующем пункте сначала доказывается счетность множества всех алгоритмов, а затем строится прямая нумерация команд, программ, вычислимых функций. В качестве такой нумерации выбрана гёделева нумерация, хотя отмечается возможность и другого подхода. Выбор в качестве конкретной модели МНР позволяет сделать эту нумерацию достаточно наглядной. Тут же разбирается канторова диагональ как один из важнейших методов доказательства.

Приведенные в третьем пункте теоремы о параметризации служат мощным инструментом в дальнейшем изложении.

В четвертом пункте обобщается универсальная машина Тьюринга — строится универсальный алгоритм и рассматриваются его основные свойства.

Следующий пункт посвящен изложению неразрешимых проблем в теории вычислений. Объясняется роль понятия частичной определенности в теории алгоритмов. Для каждой неразрешимой проблемы изложена ее интерпретация для практики программирования на компьютере.

В шестом пункте вводятся понятия разрешимого и перечислимого множества, исследуются их основные свойства и связь между ними.

Последний пункт посвящен теореме Райса, которая служит своеобразным обобщением многих неразрешимых проблем. Ее интерпретация для нужд практического программирования позволяет объяснить многие особенности программного обеспечения современного компьютера.

6.1. Эквивалентность различных теорий алгоритмов

Мы познакомились с несколькими теориями, каждая из которых уточняет понятие алгоритма. Возникает вопрос, как связаны эти теории между собой? Ответ на него дает следующая фундаментальная теорема.

Теорема 1. Следующие классы функций (заданных на натуральных числах и принимающих натуральные значения) совпадают:

а) класс всех функций, вычислимых по Тьюрингу;
б) класс всех частично-рекурсивных функций;
в) класс всех нормально вычислимых функций;
г) класс всех МНР-вычислимых функций.

Это уже не тезис, а математическая теорема, которая может быть строго доказана (мы этого доказательства приводить не будем).

Уясним смысл и значение приведенной теоремы. Она означает, что теории машин Тьюринга, нормальных алгоритмов Маркова, рекурсивных функций и МНР равносильны. В разное время в разных странах ученые независимо друг от друга, изучая интуитивные понятия алгоритма и вычислимости, создали теории, описывающие данные понятия. И эти теории оказались равносильными. Если бы один их этих классов оказался шире какого-либо другого класса, то соответствующий тезис Чёрча, Маркова или Тьюринга был бы опровергнут. Например, если бы класс нормально вычислимых функций оказался шире класса частично-рекурсивных функций, то существовала бы нормально вычислимая, но не частично-рекурсивная функция. В силу ее нормальной вычислимости она была бы алгоритмически вычислима в интуитивном понимании алгоритма, и утверждение о том, что она не является частично-рекурсивной, опровергало бы тезис Чёрча. Но сформулированная теорема справедлива, и таких функций не существует, что служит еще од-

ним косвенным подтверждением истинности тезисов Тьюринга, Маркова и Чёрча. Отметим, что существуют еще и другие теории алгоритмов, и для всех них также доказана их равносильность с рассматриваемыми теориями.

Особого внимания заслуживает тезис Тьюринга. В силу специфики определения машины Тьюринга он обладает высшей степенью убедительности. В самом деле, работа машины Тьюринга адекватно описывает поведение математика-вычислителя, занятого выполнением данного ему предписания: находясь в определенном «умонастроении» и концентрируя свое внимание на определенной текстовой информации, вычислитель предписанным ему образом изменяет эту информацию, «смещает» центр своего внимания и переходит в новое «умонастроение». То же самое способна делать и машина Тьюринга. То обстоятельство, что ее шаги носят более локальный характер, чем действия реального вычислителя, на самом деле не слишком качественно: каждый «интегральный» шаг работы вычислителя может быть смоделирован серией «локальных» шагов машины Тьюринга. Приведенные соображения с учетом эквивалентности машин Тьюринга другим теориям (по теореме из этого параграфа) являются, быть может, самым сильным аргументом в пользу тезисов Тьюринга, Чёрча, Маркова. Разберемся немного с историей этого вопроса.

Впервые принцип, утверждающий пригодность некоторых конкретных уточнений понятия алгоритма, был сформулирован А. Чёрчем во второй половине тридцатых годов XX века и относился к λ-определимым функциям [6].

Чуть позже А. Тьюринг заложил основы теории машин Тьюринга и сформулировал свой принцип.

В конце 40-х начале 50-х годов XX века А. А. Марков разработал теорию нормальных алгоритмов (или алгорифмов, как называл их создатель теории).

Внимательный читатель уже обратил внимание, что машины с неограниченными регистрами (Дж. Шепердсон и X. Стерджис, 1963 год) реализуют идеи частично-рекурсивных функций и машин Тьюринга. Этот подход гораздо ближе к современным алгоритмам, чем остальные. Поэтому, принимая во внимание, что работа адресована в основном студентам-информатикам, этот подход в дальнейшем будет использоваться чаще, чем другие.

Обычно тезис Чёрча—Тьюринга—Маркова называют просто тезисом Чёрча, подчеркивая этим, кем впервые был сформулирован этот результат.

Итак, всякий алгоритм, описанный в терминах, например, частично-рекурсивных функций, можно реализовать машиной Тьюринга, нормальным алгоритмом Маркова или МНР, и наоборот. Отсюда следует, что любые утверждения о существовании или несуществовании алгоритмов, сделанные в одной из теорий, верны и в другой. В сочетании с тезисом Чёрча это означает, что такие утверждения можно формулировать для алгоритмов вообще, не фиксируя конкретную модель и используя результаты всех теорий.

Таким образом, возможно изложение теории алгоритмов, инвариантное по отношению к способу формирования понятия «алгоритм», — при любой формализации основные свойства алгоритмов остаются теми же самыми. (Это верно, когда речь идет о существовании или не существовании алгоритмов; характеристиках тех качеств алгоритмов, которые, вообще говоря, инвариантны по отношению к выбранной формализации.) Основные понятия такой инвариантной теории — это **алгоритм** (рекурсивное описание функции, система команд машины Тьюринга, схема нормального алгоритма Маркова, система команд МНР) и **вычислимая функция**.

Эквивалентность утверждения «функция f вычислима» и «существует алгоритм, вычисляющий функцию f» иногда приводит к смешению понятий алгоритма и вычислимой функции. Различие между ними — это различие между функцией и способом ее задания. Например, примитивно-рекурсивная функция может иметь много примитивно-рекурсивных описаний. Примитивно-рекурсивные описания также можно разбить на классы эквивалентности, отнеся в один класс все описания, задающие одну и ту же функцию. Однако задача распознавания эквивалентности примитивно-рекурсивных описаний алгоритмически неразрешима.

Обратим внимание на прикладную сторону излагаемых здесь теорий. Ввиду инвариантности основных результатов общей теории алгоритмов, их прикладное значение никак не связано с тем, насколько близки к практике используемые в них теоретические модели алгоритмов. Например, машины Тьюринга весьма далеки от современных компьютеров, а рекурсивные функции — от языков программирования, прежде всего из-за предельной скромности используемых средств. Однако именно скромность средств, во-первых, делает эти

модели чрезвычайно удобным языком доказательств, во-вторых, позволяет понять, без чего нельзя обойтись, а без чего можно и какой ценой, т. е. отличать удобства от принципиальных возможностей. Иначе говоря, прикладное значение рассматриваемых моделей заключается в том, что с их помощью удобно строить теорию, верную для любых алгоритмических моделей, в том числе и для сколь угодно близких к практике.

6.2. Нумерация алгоритмов

То, что множество всех алгоритмов счетно, можно объяснить и без обращения к конкретным алгоритмическим моделям: любой алгоритм можно задать конечным описанием (например, в алфавите знаков, используемых при наборе математических книг), а множество всех конечных слов в фиксированном алфавите счетно.

Счетность множества алгоритмов означает наличие взаимно однозначного соответствия между алгоритмами и числами натурального ряда, то есть функции типа $\varphi : \mathbb{N} \to A^*$, где A^* — множество алгоритмов.

Такая функция $\varphi(n) = A$, $n \in \mathbb{N}$, $A \in A^*$, называется *нумерацией алгоритмов*, а ее аргумент n — номером алгоритма A при нумерации φ.

Из взаимной однозначности отображения φ следует существование обратной функции $\varphi^{-1}(A_n) = n$, восстанавливающей по описанию алгоритма A_n его номер n.

Далее, используя конкретную алгоритмическую модель — МНР, мы сделаем описанные общие результаты более наглядными и введем конкретную нумерацию.

6.2.1. Нумерация программ

Множество X *счетно*, если существует биекция f: $X \to \mathbb{N}$.

Перечислением или *нумерацией множества* X называется сюръективное отображение g: $\mathbb{N} \to X$, часто его представляют следующей записью: $X = \{x_0, x_1, \ldots\}$, где $x_n = g(n)$. Если функция $g(n)$ инъективна, то говорят о перечислении или нумерации без повторений.

Пусть X — множество конечных объектов (например, множество команд или множество программ); X называется

эффективно счетным, если существует биекция $f\colon X \to \mathbb{N}$, такая, что обе функции f и f^{-1} эффективно вычислимы.

Теорема 2. Следующие множества являются эффективно счетными:

a) $\mathbb{N}\times\mathbb{N}$;

b) $\mathbb{N}^{+}\times\mathbb{N}^{+}\times\mathbb{N}^{+}$;

c) $\bigcup\limits_{k>0}\mathbb{N}^{k}$ — множество всех конечных последовательностей натуральных чисел.

Доказательство.

a) Биекция $\pi\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ определяется равенством $\pi(m, n) = 2^{m}(2n + 1) - 1$. Из определения очевидно, что π^{-1} можно задать как $\pi^{-1}(x) = (\pi_1(x), \pi_2(x))$, где π_1 и π_2 — вычислимые функции, определяемые следующими равенствами:

π_1 = минимальной степени 2 в двоичном разложении числа $x + 1$;

$$\pi_2 = \frac{1}{2}\left(\frac{x+1}{2^{\pi_1(x)}} - 1\right).$$

b) Для явного задания биекции $\zeta\colon \mathbb{N}^{+}\times\mathbb{N}^{+}\times\mathbb{N}^{+} \to \mathbb{N}$ используем введенную выше функцию π:

$$\zeta(m, n, q) = \pi(\pi(m - 1, n - 1), q - 1).$$

Тогда

$$\zeta^{-1}(x) = (\pi_1(\pi_1(x)) + 1,\ \pi_2(\pi_1(x)) + 1,\ \pi_2(x) + 1).$$

Поскольку функции π, π_1, π_2 эффективно вычислимы, то эффективно вычислимы функции ζ и ζ^{-1}.

c) Биекция $\tau\colon \bigcup\limits_{k>0}\mathbb{N}^{k} \to \mathbb{N}$ задается тождеством:

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \ldots + 2^{a_1+a_2+\ldots+a_k+k-1} - 1.$$

Очевидно, что функция τ эффективно вычислима. Чтобы установить биективность τ и вычислить $\tau^{-1}(x)$, используем тот факт, что у каждого натурального числа имеется ровно одно представление в двоичной позиционной системе. Таким образом, для данного x можно эффективно и единственным образом указать числа $k \geq 1$ и $0 \leq b_1 < b_2 < \ldots < b_k$, такие, что $x + 1 = 2^{b_1} + 2^{b_2} + \ldots + 2^{b_k}$, откуда получаем $\tau^{-1}(x) = (a_1, \ldots, a_k)$, где $a_1 = b_1$, а $a_{i+1} = b_{i+1} - b_i - 1$ $(1 \leq i < k)$. □

Обозначим множество всех команд МНР через $\mathcal{L}$, а множество всех программ через $\mathcal{P}$. Каждая программа состоит из конечного списка команд, поэтому сначала мы рассмотрим множество $\mathcal{L}$, а затем $\mathcal{P}$.

Теорема 3. Множество $\mathcal{L}$ эффективно счетно.

Доказательство. Явно определим биекцию $\beta\colon \mathcal{L} \to \mathbb{N}$, которая отображает команды четырех типов в натуральные числа вида $4u$, $4u + 1$, $4u + 2$, $4u + 3$ соответственно. Используем функции π и ξ, определенные при доказательстве теоремы 2.

$\beta(Z(n)) = 4(n - 1)$,
$\beta(S(n)) = 4(n - 1) + 1$,
$\beta(T(m, n)) = 4\pi(m - 1, n - 1) + 2$,
$\beta(J(m, n, q)) = 4\xi(m, n, q) + 3$.

Это явное определение показывает эффективную вычислимость функции β. Для вычисления $\beta^{-1}(x)$ найдем сначала такие числа u и r, что $x = 4u + r$ при $0 \le r \le 3$. Значение r указывает тип команды $\beta^{-1}(x)$, а u позволяет найти конкретную команду данного типа, а именно:

если $r = 0$, то $\beta^{-1}(x) = Z(u + 1)$;
если $r = 1$, то $\beta^{-1}(x) = S(u + 1)$;
если $r = 2$, то $\beta^{-1}(x) = T(\pi_1(u) + 1,\ \pi_2(u) + 1)$;
если $r = 3$, то $\beta^{-1}(x) = J(m, n, q)$, где $(m, n, q) = \xi^{-1}(u)$.

Следовательно, функция эффективно вычислима. □

Теорема 4. Множество $\mathcal{P}$ эффективно счетно.

Доказательство. Определим биекцию $\gamma\colon \mathcal{P} \to \mathbb{N}$ следующим образом, используя биекции τ и β теорем 2 и 3:

если $P = I_1, I_2, \ldots, I_s$, то $\gamma(P) = \tau(\beta(I_1), \ldots, \beta(I_s))$.

Так как τ и β — биекции, то и γ является биекцией, а эффективная вычислимость функций τ, β и обратных к ним функций обеспечивает эффективную вычислимость γ и γ^{-1}. □

Число $\gamma(P)$ называется *кодовым номером*, или *гёделевым номером* программы P, или просто *номером* P.

Термин гёделев номер стал применяться после знаменитых работ К. Гёделя, впервые осуществившего идею кодирования нечисловых объектов натуральными числами в 1931 году.

Пусть P_n — программа с номером $n = \gamma(P_n)$, $P_n = \gamma^{-1}(n)$, будем называть P_n n-й программой. По построению γ неравенство $m \neq n$ влечет за собой различие программ P_m и P_n, хотя они могут вычислять одну и ту же функцию.

Для дальнейших результатов крайне важно, что функции γ и γ^{-1} эффективно вычислимы, т. е.:

a) по данной программе P можно эффективно найти ее номер $\gamma(P)$;

b) по данному номеру n можно эффективно найти программу $P_n = \gamma^{-1}(n)$.

Пример 1.

a) Пусть P является программой $T(1, 3)$, $S(4)$, $Z(6)$. Вычислим $\gamma(P)$.

$\beta(T(1, 3)) = 4\pi(0, 2) + 2 = 4(2^0(2 \cdot 2 + 1) - 1) + 2 = 18$,

$\beta(S(4)) = 4 \cdot 3 + 1 = 13$,

$\beta(Z(6)) = 4 \cdot 5 = 20$.

Следовательно, $\gamma(P) = 2^{18}+2^{32}+2^{53}-1 = 9007203549970431$.

b) Пусть $n = 4127$, найдем P_{4127}. $4127 = 2^5 + 2^{12} - 1$; следовательно, P_{4127} является программой с двумя командами I_1 и I_2, где

$\beta(I_1) = 5 = 4 \cdot 1 + 1$.

$\beta(I_2) = 12 - 5 - 1 = 6 = 4 \cdot 1 + 2 = 4\pi(1, 0) + 2$.

Согласно определению β, $I_1 = S(2)$, а $I_2 = T(2, 1)$, и, значит, P_{4127} есть $S(2)$, $T(2, 1)$. △

Существует, конечно, много других возможных эффективных биекций из $\mathcal{P}$ в $\mathbb{N}$; наш выбор деталей определения γ был в достаточной степени произвольный. Снова подчеркнем, что существенна эффективная вычислимость γ и γ^{-1}. Для излагаемой далее теории подходит всякая другая биекция, удовлетворяющая этому условию.

Однако нам надо зафиксировать какую-нибудь конкретную нумерацию программ, и мы выбираем нумерацию, задаваемую функцией γ из теоремы 4, т. е., например, всюду в дальнейшем P_{4127} всегда означает программу $S(2)$, $T(2, 1)$.

6.2.2. Нумерация вычислимых функций

Исследуя нашу фиксированную нумерацию программ, можно перенумеровать вычислимые функции, а также их области определения и множества значений. Но сначала введем важные обозначения.

Для каждого $a \in \mathbb{N}$ и $n \geq 1$

$\Phi_a^{(n)}$ — n-местная функция, вычислимая по программе P_a ($f_{P_a}^{(n)}$ в обозначениях 5.2);

$W_a^{(n)}$ — область определения функции $\Phi_a^{(n)}$,

$$W_a^{(n)} = \{(x_1, \ldots, x_n) | P_a(x_1, \ldots, x_n)\downarrow\};$$

$E_a^{(n)}$ — множество значений функции $\Phi_a^{(n)}$.

В последующих главах мы в основном будем иметь дело с одноместными функциями и для удобства опустим верхний индекс. Так, будем писать Φ_a вместо $\Phi_a^{(1)}$, W_a вместо $W_a^{(1)}$, E_a вместо $E_a^{(1)}$.

Пример 2. Пусть $a = 4127$. Из предыдущего параграфа мы знаем, что программа P_{4127} есть $S(2)$, $T(2, 1)$. Следовательно, $\Phi_{4127}(x) = 1$ для всех x, а $\Phi_{4127}^{(n)}(x_1, \ldots, x_n) = x_2 + 1$, если $n > 1$. Таким образом, $W_{4127} = \mathbb{N}$, $E_{4127} = \{1\}$ при $n = 1$ и $W_{4127}^{(n)} = \mathbb{N}^n$, $E_{4127}^{(n)} = \mathbb{N}^+$, если $n > 1$. △

До сих пор мы отталкивались от некоторой программы P_a. Теперь рассмотрим вычислимые функции.

Пусть f — одноместная вычислимая функция. Тогда должна найтись вычисляющая f программа P, такая, что $f = \Phi_a$, где $a = \gamma(P)$. Будем говорить, что a есть некоторый индекс f.

Поскольку для каждой функции существует много различных вычисляющих ее программ, нельзя сказать, что a — единственный индекс f; в действительности у каждой вычислимой функции бесконечно много индексов.

Отсюда следует, что каждая одноместная вычислимая функция представлена в перечислении Φ_0, Φ_1, Φ_2, ... и что это перечисление с повторениями. Аналогичное утверждение относится также к n-местным функциям и их перечислениям. Обозначим через b_n множество всех n-местных вычислимых функций.

Теорема 5. Множество b_n счетно.

Доказательство. Используя перечисление $\Phi_0^{(n)}$, $\Phi_1^{(n)}$, $\Phi_2^{(n)}$, ..., в котором есть повторения, построим перечисление без повторений. Пусть

$$f(0) = 0,\ f(m+1) = \mu z(\Phi_z^{(n)} \neq \Phi_{f(0)}^{(n)}, \ldots, \Phi_{f(m)}^{(n)}).$$

Тогда $\Phi^{(n)}_{f(0)}, \Phi^{(n)}_{f(1)}, \Phi^{(n)}_{f(2)}, \ldots$ есть перечисление b_n без повторений. □

Отметим, что мы не требуем, чтобы функция f, определенная в этом доказательстве, была вычислимой. Тем не менее, можно построить всюду определенную вычислимую функцию, такую, что $\Phi^{(n)}_{h(0)}, \Phi^{(n)}_{h(1)}, \ldots$ окажется перечислением b_n без повторений [19].

Следствие. Множество b счетно.

Доказательство. Так как $b = \bigcup\limits_{n\geq 1} b_n$, то счетность b вытекает из того, что объединение счетного числа счетных множеств есть счетное множество.

Для каждого n положим, что f_n равна функции, использованной в теореме 5 для перечисления b_n без повторений. Пусть π — биекция $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ из теоремы 2. Определим θ: $b \to \mathbb{N}$ равенством $\theta(\Phi^{(n)}_{f_n(m)}) = \pi(m, n-1)$. Очевидно, что θ — биекция. □

Следующая теорема показывает, что существует невычислимая функция. Идея доказательства столь же важна, как и результат.

Теорема 6. Существует невычислимая всюду определенная (или тотальная) функция.

Доказательство. Мы построим тотальную функцию f, одновременно отличающуюся от каждой функции в перечислении $\Phi_0, \Phi_1, \Phi_2, \ldots$ класса b_1.

Положим

$$f(n) = \begin{cases} \Phi_n(n) + 1, \text{ если значение } \Phi_n(n) \text{ определено,} \\ 0, \text{ если значение } \Phi_n(n) \text{ не определено.} \end{cases}$$

Заметим, что мы так определили f, что для каждого n функция f отличается от Φ_n в точке n:

- если $\Phi_n(n)$ определено, то $f(n) = \Phi_n(n) + 1 \neq \Phi_n(n)$;
- если $\Phi_n(n)$ не определено, то f отличается от Φ_n тем, что значение $f(n)$ определено.

Так как f отличается от каждой одноместной вычислимой функции Φ_n, то f не представлена в перечислении класса b_1, т. е. не является вычислимой. Очевидно, что f — тотальная функция. □

Метод построения функции f в теореме 6 является примером диагональной конструкции, открытой Г. Кантором. Многие читатели знакомы с этим методом по его примене-

нию в доказательстве несчетности множества действительных чисел. Лежащая в его основе идея применима к огромному числу ситуаций и является центральной в доказательствах многих результатов, относящихся к вычислимости и разрешимости.

Объясним, почему выбран термин «диагональный». Для этого опять рассмотрим построение функции f в теореме 6. Полную информацию о функциях Φ_0, Φ_1, Φ_2, ... можно представить следующей бесконечной таблицей.

Таблица 6.1

	0	1	2	3	...
Φ_0	$\Phi_0(0)$	$\Phi_0(1)$	$\Phi_0(2)$	$\Phi_0(3)$	...
Φ_1	$\Phi_1(0)$	$\Phi_1(1)$	$\Phi_1(2)$	$\Phi_1(3)$	...
Φ_2	$\Phi_2(0)$	$\Phi_2(1)$	$\Phi_2(2)$	$\Phi_2(3)$	...
Φ_3	$\Phi_3(0)$	$\Phi_3(1)$	$\Phi_3(2)$	$\Phi_3(3)$	...
...	...	...	...	...	...

Считаем, что в том случае, когда значение $\Phi_n(m)$ не определено, в таблице записано «не определено».

Для построения функции f отбираются диагональные элементы таблицы (они обведены овалами) $\Phi_0(0)$, $\Phi_1(1)$, $\Phi_2(2)$, ... и систематически изменяются так, что для каждого n значение $f(n)$ отличается от $\Phi_n(n)$. Заметим, что значения $f(n)$ можно выбирать сравнительно свободно, надо только обеспечить отличие $f(n)$ от $\Phi_n(n)$. Таким образом, например,

$$g(n) = \begin{cases} \Phi_n(n) + 27^n, \text{ если } \Phi_n(n) \text{ определена,} \\ n^2, \text{ если } \Phi_n(n) \text{ не определена,} \end{cases}$$

является другой невычислимой тотальной функцией.

Если говорить о множествах, то вопрос состоит в том, принадлежит ли n множеству или нет. Проиллюстрируем диагональную конструкцию на примере множеств.

Пример 3. Предположим, что A_0, A_1, A_2, ... есть перечисление всех подмножеств множества $\mathbb{N}$. Определим новое множество B, применяя диагональный принцип: $\forall n \in \mathbb{N}$ если $n \notin A_n$, то включаем его в множество B, если $n \in A_n$, то n не включаем B. Очевидно, что $B \notin A_n$ для всякого n. Следовательно, множество всех подмножеств $\mathbb{N}$ несчетно. $\triangle$

6.3. Теоремы параметризации

Предположим, что $f(x, y)$ — вычислимая функция (не обязательно тотальная). Тогда для каждого фиксированного значения переменной $x = a$ функция f приводит к одноместной вычислимой функции g_a, где $g_a(y) = f(a, y)$. Поскольку g_a вычислима, то у нее есть индекс e, такой, что $f(a, y) = \Phi_e(y)$.

Следующая теорема показывает, что индекс e можно эффективно найти по a.

Теорема 7 (***s-m-n-теорема, простая форма***). Предположим, что $f(x, y)$ — вычислимая функция. Существует тотальная вычислимая функция $k(x)$, такая, что $f(x, y) = \Phi_{k(x)}(y)$.

Доказательство. Для каждого фиксированного a через $k(a)$ обозначим кодовое число программы Q_a, которая, исходя из данной начальной конфигурации

R_1					
y	0	0	0	0	...

(1)

вычисляет $f(a, y)$.

Пусть F — программа, вычисляющая функцию f. Тогда Q_a получается из F приписыванием спереди команд, преобразующих конфигурацию (1) в

R_1	R_2				
a	y	0	0	0	...

Таким образом, определим Q_a как следующую программу:

$$\begin{array}{l} T(1, 2) \\ Z(1) \\ a \text{ раз} \begin{cases} S(1) \\ \ldots \\ S(1) \end{cases} \\ F \end{array}$$

Поскольку F фиксировано и ввиду эффективности нашей нумерации программ γ, функция k оказывается эффективно вычислимой. Следовательно, по тезису Чёрча k — вычислимая функция. По построению $\Phi_{k(a)} = f(a, y)$ для каждого a. □

S-m-n-теорему иногда называют **теоремой параметризации**, так как она показывает, что индекс вычислимой функции (такой, как g_a в приведенном рассуждении) можно эффективно найти по параметру (такому, как a), от которого он эффективно зависит.

Пример 4. Пусть $f(x, y) = y^x$. По теореме 7 существует тотальная вычислимая функция k такая, что $\Phi_{k(x)}(y) = y^x$. Следовательно, для каждого фиксированного n величина $k(n)$ является индексом функции y^n. △

Пример 5. Пусть

$$f(x, y) = \begin{cases} y, \text{ если } y \text{ кратно } x, \\ \text{не определена в противном случае.} \end{cases}$$

Функция f вычислима. Обозначим через k вычислимую функцию, такую, что $\Phi_{k(x)}(y) = f(x, y)$. Теперь для каждого фиксированного n значение $\Phi_{k(n)}(y)$ определено тогда и только тогда, когда y кратно n, и тогда и только тогда, когда y принадлежит множеству значений $\Phi_{k(n)}$, т. е. $W_{k(n)} = n\mathbb{N}$ (множество чисел, кратных n) $= E_{k(n)}$. Таким образом, мы получили индексацию последовательности множеств $(n\mathbb{N})$ как областей определения вычислимых функций и множеств значений вычислимых функций. △

Очевидный способ обобщения теоремы 7 состоит в замене единственных переменных x, y на m, n-наборы $\mathbf{x}$ и $\mathbf{y}$ соответственно. Можно также отразить то обстоятельство, что функция k, определенная в доказательстве теоремы 7, эффективно зависит от конкретной программы для исходной функции f. Вместо фиксированной вычислимой функции $f(x, y)$ мы рассмотрим вычислимую функцию общего вида $\Phi_e^{(m+n)}(\mathbf{x}, \mathbf{y})$ и проблему нахождения для каждого индекса e числа z, такого, что $\Phi_e^{(m+n)}(\mathbf{x}, \mathbf{y}) = \Phi_z^{(n)}(\mathbf{y})$.

Теорема 8 (***s-m-n-теорема***). Для всяких натуральных чисел m, $n \geq 1$ существует тотальная вычислимая $(m + 1)$-местная функция $S_n^m(e, \mathbf{x})$, такая, что

$$\Phi_e^{(m+n)}(\mathbf{x}, \mathbf{y}) = \Phi_{S_n^m(e, \mathbf{x})}^{(n)}(\mathbf{y}).$$

Доказательство. Обобщим доказательство теоремы 7. Для каждого $1 \leq i \leq m$ определим $Q(i, x_i)$ как подпрограмму

$$\begin{array}{l} Z(i) \\ \left.\begin{array}{l} S(i) \\ \dots \\ S(i) \end{array}\right\} x_i \text{ раз,} \end{array}$$

которая заменяет текущее содержимое R_i на x_i. Тогда для фиксированных m, n определим $S_n^m(e, x)$ как кодовое число следующей программы:

$$\left.\begin{array}{l} T(n, m+n) \\ \dots \\ T(2, m+2) \\ T(1, m+1) \\ Q(1, x_1) \\ Q(2, x_2) \\ \dots \\ Q(m, x_m) \end{array}\right\}$$

Эта часть программы преобразует любую конфигурацию вида

R_1		R_n			
y_1	...	y_n	0	0	...

в конфигурацию

R_1		R_m	R_{m+1}		R_{m+n}		
x_1	...	x_m	y_1	...	y_n	0	...

P_e

Из этого явного определения и эффективности γ и γ^{-1} получаем эффективную вычислимость функции S_n^m, а, следовательно, по тезису Чёрча и ее вычислимость. □

6.4. Универсальный алгоритм

Существование нумерации алгоритмов позволяет работать с ними как с числами. Это особенно удобно при исследовании алгоритмов над алгоритмами. Такие алгоритмы уже рассматривались при построении универсальной машины Тьюринга и в связи с проблемой остановки. Полученные результаты можно сформулировать в инвариантном виде.

Рассмотрим функцию $\psi(x, y)$, заданную соотношением $\psi(x, y) = \Phi_x(y)$. То есть единственная функция ψ реализует все одноместные вычислимые функции Φ_0, Φ_1, ... в том смысле, что для любого фиксированного m функция g, заданная как $g(y) = \psi(m, y)$, является в точности функцией Φ_m. Таким образом, мы описали ψ как универсальную функцию для одноместных вычислимых функций. Введем следующее определение.

Универсальной функцией для n-местных вычислимых функций называется $(n+1)$-местная функция $\psi_{\cup}^{(n)}$, определенная соотношением

$$\psi_{\cup}^{(n)}(l, x_1, \ldots, x_n) = \Phi_l^{(n)}(x_1, \ldots, x_n).$$

Вместо $\psi_{\cup}^{(1)}$ мы будем писать $\psi_{\cup}$.

Теорема 9. Для каждого n универсальная функция $\psi_{\cup}^{(n)}$ вычислима.

Идея доказательства. Для конкретной нумерации γ' теорема доказывается построением универсальной машины Тьюринга и теоремой 1. Для любой другой вычислимой нумерации γ'' можно выбрать один из двух путей:

a) построить новую универсальную машину Тьюринга, работающую непосредственно с этой нумерацией;
b) построить алгоритм перевода γ'' в γ'.

В частности, из теоремы следует, что любая программа P, вычисляющая $\psi_{\cup}^{(n)}$, может вычислять все другие функции, и P подходит название универсальной программы, а $\psi_{\cup}^{(n)}$ — универсальной функции.

По существу, вычислимая нумерация служит языком программирования для универсального алгоритма. Путь «a» — это построение новой машины для каждого нового языка, путь «b» — построение транслятора для прежней машины. Переформулируем теорему в общем виде.

Теорема 10. Существует универсальный алгоритм $U(y, x_1, \ldots, x_n)$, такой, что для любого алгоритма A с номером $\gamma(A)$ $U(\gamma(A), x_1, \ldots, x_n) = A(x_1, \ldots, x_n)$ или $U(y, x_2, \ldots, x_n) = A_y(x_1, \ldots, x_n)$.

6.5. Неразрешимые проблемы в теории вычислимости

Решение задачи перечисления всех алгоритмов достаточно ясно, по крайне мере в принципе. Могло бы показаться, что перечисление примитивно-рекурсивных или общерекурсивных функций окажется более легким делом. Однако это не так.

Теорема 11. *Для любого перечисления множества всюду определенных вычислимых (т. е. общерекурсивных) функций существует общерекурсивная функция, не входящая в это перечисление.*

Доказательство. Пусть γ — перечисление множества общерекурсивных функций, порождающее их последовательность $f_0, f_1, f_2, \ldots$. Введем функцию $g(x) = f_x(x) + 1$. Так как f_x общерекурсивна, то и g общерекурсивна. Если $g(x)$ попадает в перечисление γ, то она имеет некоторый номер x_g и, следовательно, $g(x) = f_{x_g}(x)$. Тогда в точке $x = x_g$ по определению $g(x_g) = f_{x_g}(x_g) + 1$, а по перечислению $g(x_g) = f_{x_g}(x_g)$. Получаем противоречие, откуда следует, что g не входит в перечисление, порождаемое γ. □

Отметим, что в приведенном доказательстве используется диагональный метод Г. Кантора.

Естественно, возникает желание построить алгоритм, который бы выделил общерекурсивные функции из множества вычислимых функций. Но и тут нас подстерегает неудача.

Теорема 12. *Проблема определения общерекурсивности функций неразрешима, то есть неразрешима проблема «Φ_x всюду определена».*

Доказательство. Пусть g — характеристическая функция этой проблемы, т. е.

$$g(x) = \begin{cases} 1, \text{ если } \Phi_x \text{ тотальна,} \\ 0, \text{ если } \Phi_x \text{ нетотальна.} \end{cases}$$

Нам надо показать, что функция $g(x)$ невычислима. Используя диагональный метод, зададим функцию f следующим образом:

$$f(x) = \begin{cases} \Phi_x(x) + 1, & \text{если } \Phi_x \text{ тотальна,} \\ 0, & \text{если } \Phi_x \text{ нетотальна.} \end{cases}$$

Очевидно, что f — тотальная и отличается от каждой вычислимой функции Φ_x. Теперь, применяя g и универсальную функцию $\psi_\cup$, заменяем f следующим образом:

$$f(x) = \begin{cases} \psi_\cup(x, x) + 1, & \text{если } g(x) = 1, \\ 0, & \text{если } g(x) = 0. \end{cases}$$

Предположим, что функция g вычислима. Так как $\psi_{\cup}$ вычислима, то по тезису Чёрча вычислима и функция f, что приводит к противоречию. Следовательно, g невычислима. □

Рассмотрим так называемую **проблему самоприменимости** машин Тьюринга, которая состоит в следующем: остановится ли машина Тьюринга T, если на ленте машины T написана ее собственная система команд? В инвариантном виде соответствующее утверждение формулируется так.

Теорема 13. Проблема «$x \in W_x$» (или, что эквивалентно, «функция $\Phi_x(x)$ определена», или «$P_x(x)\downarrow$», или «функция $\psi_{\cup}(x, x)$ определена») неразрешима.

Доказательство. Характеристическая функция этой проблемы задается следующей формулой:

$$f(x) = \begin{cases} 1, \text{ если } x \in W_x, \\ 0, \text{ если } x \notin W_x. \end{cases}$$

Проведем доказательство методом от противного. Пусть функция f вычислима. Определим функцию g следующим образом:

$$g(x) = \begin{cases} 0, & \text{если } x \notin W_x \text{ (т. е. если } f(x) = 0), \\ \text{не определена}, & \text{если } x \in W_x \text{ (т. е. если } f(x) = 1). \end{cases}$$

Поскольку функция f вычислима, то по тезису Чёрча вычислима и функция g. Значит, найдется такое m, что $g = \Phi_m$. Отсюда следует, что если $m \in W_m$, то $m \in Dom(g)$ и по определению функции g $m \notin W_m$. Если же $m \notin W_m$, то $m \notin Dom(g)$ и $m \in W_m$. Получаем противоречие, а значит, функция f не является вычислимой, и, следовательно, проблема «$x \in W_x$» неразрешима. □

Заметим, что эта теорема вовсе не утверждает, что мы не можем для любого конкретного числа a сказать, будет ли определено значение $\Phi_a(a)$. Для некоторых чисел это очень просто. Например, если мы напишем программу P, вычисляющую тотальную функцию, и $P = P_a$, то мы сразу знаем, что значение $\Phi_a(a)$ определено. Теорема утверждает, что не существует единого *общего* метода решения вопроса о том, будет ли $\Phi_x(x)$ определена; другими словами, не существует метода, работающего при всех x.

Теорема 14 (***проблема остановки***). Проблема «$\Phi_x(y)$ определена» (или в эквивалентной форме «$P_x(y)\downarrow$», или «$y \in W_x$»] неразрешима.

Доказательство. Рассмотрим характеристическую функцию проблемы «функция $\Phi_x(y)$ определена», которая имеет следующий вид:

$$g(x, y) = \begin{cases} 1, \text{ если } \Phi_x(y) \text{ определена,} \\ 0, \text{ если } \Phi_x(y) \text{ не определена.} \end{cases}$$

Если бы функция g была вычислима, то вычислимой была бы и функция $f(x) = g(x, x)$, но f — характеристическая функция проблемы «$x \in W_x$» и в силу теоремы 13 она невычислима. Значит, функция g невычислима, и проблема «$\Phi_x(y)$ определена» неразрешима. □

Этот результат был подробно прокомментирован во второй главе на примере машины Тьюринга.

Отметим, что самоприменимость — частный случай проблемы остановки и именно поэтому теорему 13 нельзя получить из теоремы 14 подстановкой x вместо y в $\Phi_x(y)$; частный случай алгоритмически неразрешимой проблемы может оказаться и разрешимым.

Теоремы 11, 12 и 14 проливают свет на роль понятия частичной определенности в теории алгоритмов. Еще в первой главе среди требований к алгоритмам говорилось о желательности его результативности. Первым ударом по этому требованию была неразрешимость проблемы остановки. Возникает желание либо вообще убрать частичные алгоритмы из общей теории алгоритмов (скажем, не считать их алгоритмами), либо ввести стандартный метод доопределения частичных алгоритмов. Однако ни первое, ни второе эффективными методами сделать нельзя.

Первая идея не годится потому, что в силу теоремы 12 мы знаем, что нет эффективного способа распознать частичные алгоритмы среди множества всех алгоритмов, и, следовательно, предполагаемый отбор невозможен.

Вторая идея также имеет не менее убедительные опровержения.

Теорема 15. Существует такая частично-рекурсивная функция f, что никакая общерекурсивная функция g не является ее доопределением.

Доказательство. Определим f следующим образом:

$$f(x) = \begin{cases} \Phi_x(x) + 1, & \text{если } \Phi_x(x) \text{ определена,} \\ \text{не определена,} & \text{если } \Phi_x(x) \text{ не определена.} \end{cases}$$

Очевидно, что $f(x)$ вычислима. Пусть теперь g — произвольная общерекурсивная функция и x_g — ее номер, $g(x) = \Phi_{x_g}(x)$. Так как g всюду определена, то $g(x_g) = \Phi_{x_g}(x_g)$ определена и, следовательно, $f(x_g) = \Phi_{x_g}(x_g) + 1 = g(x_g) + 1$. Таким образом, для любой общерекурсивной функции g выполняется неравенство $f(x_g) \neq g(x_g)$. □

Следовательно, существуют частичные алгоритмы, которые нельзя доопределить до всюду определенных алгоритмов.

Наконец, еще одна идея: построить язык, описывающий все всюду определенные алгоритмы и только их, — не может осуществиться потому, что описания в этом языке можно упорядочить (например, лексикографически), поэтому наличие такого языка означало бы существование полного перечисления всех всюду определенных функций, что противоречит теореме 11.

Таким образом, при формулировке общего понятия алгоритма неизбежно возникает дилемма — либо определение алгоритма должно быть достаточно общим, чтобы в число объектов, удовлетворяющих этому определению, заведомо вошли все объекты, которые естественно считать алгоритмами, либо сохранится требование об обязательной результативности алгоритма.

В первом случае этому определению будут отвечать частичные алгоритмы, и избавиться от них конструктивными методами нельзя. Во втором случае никакую процедуру нельзя называть алгоритмом до тех пор, пока для нее не будет решена проблема остановки, а единого метода решения этой проблемы не существует.

В общей теории алгоритмов используется первый вариант. Впрочем, еще раз отметим, что, когда речь идет об алгоритме, решающем данную задачу, теория алгоритмов обязательно требует сходимости (результативности), и только в случае, когда алгоритм решения всюду определен, соответствующая задача считается разрешимой.

Рассмотренная в теореме 13 неразрешимая проблема «$x \in W_x$» важна по нескольким причинам. Одна из них состоит в том, что неразрешимость многих проблем можно доказать, показав, что они по крайней мере не проще, чем эта. Этот процесс называется **сведением** одной проблемы к дру-

гой. Именно таким путем мы без большого труда доказали, что проблема остановки (теорема 14) неразрешима.

Для сведения проблемы «$x \in W_x$» к другой проблеме часто используется *s-m-n*-теорема, как показывает, например, доказательство следующего результата.

Теорема 16. Проблема «$\Phi_x = 0$» неразрешима.

Доказательство. Рассмотрим функцию f, определенную формулой:

$$f(x, y) = \begin{cases} 0, & \text{если } x \in W_x, \\ \text{не определена}, & \text{если } x \notin W_x. \end{cases}$$

Эту функцию мы ввели для того, чтобы далее воспользоваться *s-m-n*-теоремой. Тем самым мы рассматриваем x как параметр, и нас интересует функция $g_x(y) = f(x, y)$. При этом мы выбрали f так, чтобы $g_x = 0 \Leftrightarrow x \in W_x$.

Тезис Чёрча показывает, что функция f вычислима. Тогда по *s-m-n*-теореме существует тотальная вычислимая функция $k(x)$ такая, что $f(x, y) = \Phi_{k(x)}(y)$. Таким образом, по определению $x \in W_x \Leftrightarrow \Phi_{k(x)} = 0$.

Следовательно, мы свели проблему «$x \in W_x$» к проблеме «$\Phi_x = 0$». Так как первая из них неразрешима, то неразрешима и вторая. □

Эта теорема показывает, что в области проверки правильности компьютерных программ имеются принципиальные ограничения: не существует общего эффективного метода проверки того, будет ли программа вычислять нулевую функцию. Несколько изменив доказательство теоремы 16, можно убедиться в том, что аналогичное утверждение справедливо и для любой другой конкретной вычислимой функции, то есть имеет место следующая теорема.

Теорема 17. Проблема «$\Phi_x = g$», где g — любая фиксированная вычислимая функция, неразрешима.

Более того, неразрешим вопрос о том, вычисляют ли две программы одну и ту же одноместную функцию. Это показывает следующая теорема.

Теорема 18. Проблема «$\Phi_x = \Phi_y$» неразрешима.

Доказательство. Пусть c — такое число, что $\Phi_c = 0$. Если $f(x, y)$ — характеристическая функция проблемы «$\Phi_x = \Phi_y$», то функция $g(x) = f(x, c)$ — характеристическая функция проблемы «$\Phi_x = 0$». По теореме 16 функция g невычислима, так что невычислима и функция f. □

6.6. Разрешимые и перечислимые множества

Множество M называется *разрешимым* (или *рекурсивным*), если существует алгоритм A_M, который по любому объекту a дает ответ, принадлежит a множеству M или нет. Алгоритм A_M называется разрешающим алгоритмом для M.

Дадим более формальное эквивалентное определение. Множество M называется разрешимым, если оно обладает общерекурсивной характеристической функцией, то есть вычислимой всюду определенной функцией χ_M такой, что

$$\chi_M(a) = \begin{cases} 1, \text{ если } a \in M, \\ 0, \text{ если } a \notin M. \end{cases}$$

Очевидно, что пересечение, объединение и разность разрешимых множеств также разрешимы. Любое конечное множество разрешимо.

Множество M называется *перечислимым* (или *рекурсивно-перечислимым*), если оно является областью значений некоторой общерекурсивной функции, то есть существует общерекурсивная функция $\psi_M(x)$, такая, что $a \in M$ тогда и только тогда, когда $a = \psi_M(x)$ для некоторого x.

Функция ψ_M называется перечисляющей для множества M; соответственно алгоритм, вычисляющий ψ_M, называется перечисляющим и порождающим для M.

Пример 6. Множество квадратов натуральных чисел $M = \{a \mid a = x^2\}$ перечислимо (оно просто задано перечисляющей функцией $a = x^2$) и разрешимо. В качестве разрешающего алгоритма можно предложить, например, следующий: данное число a сравниваем последовательно с 0, 1^2, 2^2 и так далее до тех пор, пока не появится такое число n^2, что либо $a = n^2$, либо $a < n^2$. В первом случае $a \in M$, во втором $a \notin M$. △

Пример 7. Пусть имеется процедура вычисления цифр разложения числа π в бесконечную десятичную дробь: $\pi \approx 3{,}1415926536\ldots$. По мере вычисления будем образовывать из последовательно стоящих цифр трехразрядные числа: 314, 159, 265 … . Множество таких чисел обозначим M_π. Перечисляющую его процедуру мы уже описали при определении множества M_π; таким образом для функции ψ_π имеем: $\psi_\pi(0) = 314$, $\psi_\pi(1) = 159$, $\psi_\pi(2) = 265$ и т. д. Некоторые трой-

ки могут повторяться, поэтому ψ_π — это пример функции, перечисляющей с повторениями. Пока неизвестно никакого разрешающего алгоритма для M_π, несмотря на то, что очевидна конечность M_π. Но это вовсе не означает, что M_π неразрешимо; оно может оказаться и разрешимым (например, через достаточно большое количество шагов перечисляющей процедуры окажется, что выписаны все возможные тройки от 000 до 999). △

Пример 8. Из индуктивных определений, как правило, нетрудно извлечь порождающую процедуру. Иногда она содержится в них явно, как, скажем, в определении $(n + 1)! = n!(n + 1)$, задающем и перечисляющую функцию $\varphi(n) = n!$, и способ ее вычисления.

Иногда, чтобы получить из такого определения порождающую функцию, нужно ввести дополнительные соглашения. Например, индуктивное определение в языке программирования Паскаль «идентификатор — это либо буква, либо идентификатор, к которому справа приписана буква или цифра» порождает все возможные идентификаторы. Но чтобы придать этому процессу порождения вид перечисляющей функции, нужно ввести порядок перечисления. Если в качестве такого порядка принять лексикографический и договориться, что буквы в этом порядке предшествуют цифрам, то в алфавите из 26 латинских букв и 10 цифр $\varphi(0) = a$, $\varphi(25) = z$, $\varphi(61) = az$, $\varphi(64) = a2$ и т. д. △

Пример 9. Построенная нами функция γ (см. п. 6.2) нумерует программы, т. е. устанавливает взаимно однозначное соответствие между программами и числами. Поэтому можно сказать, что γ является перечисляющей функцией без повторений для множества всех программ (а вследствие тезиса Чёрча и алгоритмов). △

Важность введенных понятий разрешимости и перечислимости прежде всего в том, что благодаря им становятся точными такие понятия, как «конструктивный способ задания множества» и «множество, заданное эффективно». В частности, так как язык множеств является универсальным языком математики, и всякому утверждению можно придать вид утверждения о множествах, язык разрешимых и перечислимых множеств является универсальным языком для утверждений о существовании (или отсутствии) алгоритмов решения математических проблем.

Пример 10. Теорема об алгоритмической неразрешимости проблемы остановки утверждает, что множество пар (x, y), таких, что алгоритм A_x при данных y даст результат, неразрешимо. Теорема об определении общерекурсивности алгоритма утверждает, что множество всех общерекурсивных функций неразрешимо. △

Итак, эффективно заданное множество — это множество, обладающее разрешающей или перечисляющей функцией. Равносильны ли эти два типа задания? В одну сторону ответ почти очевиден.

Теорема 19. Если непустое множество M разрешимо, то оно перечислимо в алфавите A.

Доказательство. Пусть $\alpha_1, \alpha_2, \ldots, \alpha_n, \ldots$ — перечисление всех слов в алфавите A (например, в лексикографическом порядке), β — некоторое слово из M. Перечисляющую функцию $\psi_M(n)$ для M определим так:

$$\psi_M(0) = \beta;$$

$$\psi_M(n+1) = \begin{cases} \psi_M(n), & \text{если } \alpha_{n+1} \notin M, \\ \alpha_{n+1}, & \text{если } \alpha_{n+1} \in M. \end{cases}$$

Ясно, что $\psi_M(n)$ вычислима и всюду определена. □

Обращение теоремы 19 неверно.

Теорема 20. Существует множество M, которое перечислимо, но неразрешимо.

Доказательство. Докажем, что таковым является множество $M = \{x \mid A_x(x) \text{ определено}\}$, то есть множество номеров самоприменимых алгоритмов. Из теоремы 13 следует, что M неразрешимо. Построим перечислимую функцию ψ_M для M. Для определенности будем считать, что алгоритмы — это машины Тьюринга.

Введем предикат $S(x, m)$ — истинный, если машина с номером x останавливается за m шагов. Очевидно, что этот предикат вычислим и всюду определен: для его вычисления надо запустить машину T_x при данных x и дать ей проработать m шагов. Предикат $S(x, m)$ можно представить в виде бесконечной квадратной матрицы (см. таблицу 6.2). Упорядочим теперь клетки этой матрицы: (0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), ..., то есть сначала клетки с суммой координат 0, затем клетки с суммой 1 и т. д. Тогда каждая клетка получит номер, соответствующий ее порядку в этой последовательности.

Таблица 6.2

x	m			
	0	1	2	...
0	$S(0,0)$	$S(0,1)$	$S(0,2)$	...
1	$S(1,0)$	$S(1,1)$	$S(1,2)$	...
2	$S(2,0)$	$S(2,1)$	$S(2,2)$	...
...	...	...	...	...

Выберем какую-нибудь заведомо самоприменимую машину Тьюринга (например, любую всюду определенную машину из примеров главы II), вычислим ее номер во введенной нумерации и обозначим его c. Определим теперь алгоритм вычисления функции $\psi_M(n)$: находим клетку (i, j) с номером n; если $S(i, j) = \mathrm{T}$, то $\psi_M(n) = i$, если $S(i, j) = \mathrm{F}$, то $\psi_M(n) = c$. Так как $\psi_M(n) = i$ — это номер машины T_i, останавливающейся при данных i через j шагов, то область значений ψ_M содержит только номера самоприменимых машин. С другой стороны, всякая самоприменимая машина T_x останавливается через конечное число k шагов. Но тогда $\psi_M(n') = x$, где n' — это номер клетки (x, k).

Таким образом, область значений ψ_M совпадает с множеством номеров самоприменимых алгоритмов. Кроме того, ввиду общерекурсивности предиката S функция ψ_M также общерекурсивна. Следовательно, она является перечисляющей функцией для M. ☐

Согласно теореме 20, перечислимость — более слабый вид эффективности; хотя перечисляющая процедура и задает эффективно список элементов множества M, однако поиск данного элемента a в этом списке (всегда бесконечном, но, быть может, с повторяющимися элементами) может оказаться неэффективным: это неопределенно долгий процесс, который в конечном счете остановится, если $a \in M$, но не остановится, если $a \notin M$. Поэтому список элементов M, заданный перечисляющей функцией, сам по себе не гарантирует разрешающей процедуры для M. Теорема 20 дает пример, когда ее просто не существует.

В одном важном случае перечислимость гарантирует разрешимость.

Теорема 21. Множество M разрешимо тогда и только тогда, когда M и $\overline{M}$ перечислимы.

Доказательство. Если M разрешимо, то $\overline{M}$ также разрешимо, так как $\psi_{\overline{M}} = 1 - \chi_M$. Но тогда перечислимость M и $\overline{M}$ следует из теоремы 19. Пусть теперь M и $\overline{M}$ перечислимы функциями ψ_M и $\psi_{\overline{M}}$ соответственно. Но тогда для любого элемента a его поиск в объединенном списке $\psi_M(0)$, $\psi_{\overline{M}}(0)$, $\psi_M(1)$, $\psi_{\overline{M}}(1)$, ... обязательно увенчается успехом, так как либо $a \in M$, либо $a \in \overline{M}$. Такой поиск и есть разрешающая процедура для M. □

Сопоставление теорем 20 и 21 показывает, что дополнение к перечислимому множеству может оказаться неперечислимым. В частности, множество несамоприменимых алгоритмов неперечислимо. Такая «несимметрия» самоприменимости и несамоприменимости объясняется отмеченной ранее несимметрией положительного и отрицательного ответа при поиске заданного элемента в бесконечном списке. Например, если алгоритм A_x самоприменим, то, вычисляя $A_x(x)$, мы это когда-нибудь узнаем; если же A_x несамоприменим, то об этом нельзя узнать, вычисляя $A_x(x)$.

Приведем без доказательства интересный результат, связывающий понятия перечислимости и вычислимости (доказательство можно найти, например, в [10]).

Графиком n-местной частичной функции $f(x_1, ..., x_n)$ называется совокупность последовательностей $(x_1, ..., x_n, y)$ натуральных чисел, удовлетворяющих соотношению $f(x_1, ..., x_n) = y$.

В частности, график нигде не определенной функции есть пустое множество.

Теорема 22 (*о графике*). *Для того чтобы частичная функция f была частично-рекурсивной необходимо и достаточно, чтобы график был рекурсивно перечислим.*

6.7. Теорема Райса

Просматривая накопленный запас алгоритмически неразрешимых проблем, нетрудно заметить, что почти все они так или иначе связаны с самоприменимостью — довольно экзотической ситуацией, когда алгоритм работает с собственным описанием. Можно решить, что так как понятие самоприменимости далеко от алгоритмической практики, то неразрешимость в этой практике также никогда не встретится. К сожалению, это совсем не так.

Теорема 23 (*Райс*). Никакое нетривиальное свойство вычислимых функций не является алгоритмически разрешимым.

(Для доказательства эту теорему удобнее сформулировать в менее эффектном, но более точном виде: пусть A — любой класс вычислимых одноместных функций, нетривиальный в том смысле, что $A \subset B$, $A \neq \varnothing$, $A \neq B$. Тогда проблема «$\Phi_x \in A$» неразрешима.)

Доказательство. Пусть нигде не определенная функция $f_0(y) \notin A$. Выберем некоторую функцию $g(y) \in A$. Рассмотрим функцию $f(x, y)$, определенную так:

$$f(x, y) = \begin{cases} g(y), & \text{если } x \in W_x, \\ f_0(y), & \text{если } x \notin W_x. \end{cases}$$

По s-m-n-теореме получаем тотальную вычислимую функцию $k(x)$, такую, что $f(x, y) = \Phi_{k(x)}(y)$. Таким образом, получаем, что

$x \in W_x \Rightarrow \Phi_{k(x)} = g$, т. е. $\Phi_{k(x)} \in A$,

$x \notin W_x \Rightarrow \Phi_{k(x)} = f_0$, т. е. $\Phi_{k(x)} \notin A$.

Таким образом, проблема «$x \in W_x$» сведена к проблеме «$\Phi_{k(x)} \in A$». Для случая $f_0 \in A$ надо выбрать $g \notin A$, и далее аналогично вышеизложенному. □

Из теоремы Райса следует, что по номеру вычислимой функции нельзя узнать, является ли эта функция постоянной, периодической, ограниченной и т. д.

Для того чтобы разобраться в смысле теоремы Райса, надо, прежде всего, вспомнить, что номер x функции f — это номер алгоритма программы P_x, вычисляющей f; в свою очередь, по номеру алгоритма однозначно восстанавливается его описание, и разным номерам соответствуют разные алгоритмы. Для функций это неверно: при $x \neq y$ f_x и f_y могут быть одной и той же функцией.

В теореме Райса участвуют и алгоритмы, и функции, и их следует четко различать. Класс A — это класс (или свойство) функций; в то же время «f_x» означает «функция, вычислимая алгоритмом P_x». Таким образом, смысл теоремы Райса в том, что по описанию алгоритма ничего нельзя узнать о свойствах функции, которую он вычисляет. В частности, оказывается неразрешимой проблема эквивалентности алгоритмов (теорема 18 легко следует из теоремы Райса).

Тестовые задания

1. Впервые принцип, утверждающий пригодность некоторых конкретных уточнений понятия алгоритма, был сформулирован:
   1) А. Тьюрингом;
   2) А. Чёрчем;
   3) А. Марковым.

2. Задача распознавания эквивалентности примитивно-рекурсивных описаний:
   1) алгоритмически разрешима;
   2) алгоритмически неразрешима.

3. Пусть X — множество конечных объектов; X называется эффективно счетным, если существует:
   1) сюръективное отображение g: $\mathbb{N} \to X$, где $x_n = g(n)$;
   2) биекция f: $X \to \mathbb{N}$, такая, что обе функции f и f^{-1} эффективно вычислимы;
   3) биекция f: $\mathbb{N} \to X$, такая, что функция f эффективно вычислима.

4. Невычислимая всюду определенная функция:
   1) существует;
   2) не существует.

5. *Индекс вычислимой функции можно эффективно найти по параметру, от которого он эффективно зависит.* Это —
   1) s-m-n-теорема;
   2) теорема Райса;
   3) теорема о нумерации.

6. Поставить в соответствие. Для каждого $a \in \mathbb{N}$ и $n \geq 1$:
   1) $\Phi_a^{(n)}$;
   2) $W_a^{(n)}$;
   3) $E_a^{(n)}$;

   a) область определения $\Phi_a^{(n)} = \{(x_1, \ldots, x_n) \mid P_a(x_1, \ldots, x_n)\downarrow\}$;
   b) n-местная функция, вычислимая по программе $P_a = f_{P_a}^{(n)}$;
   c) одноместная вычислимая функция, представленная в перечислении Φ_0, Φ_1, Φ_2, ... ;
   d) множество значений функции $\Phi_a^{(n)}$.

7. Для любого перечисления множества всюду определенных вычислимых функций:
   1) не существует общерекурсивной функции, не входящей в это перечисление;
   2) существует общерекурсивная функция, не входящая в это перечисление;
   3) нет верного ответа.

8. Поставить в соответствие:
   1) проблема определения общерекурсивности алгоритмов неразрешима;
   2) проблема самоприменимости алгоритмов неразрешима;
   3) проблема определения результативности алгоритма неразрешима;
   a) т. е. проблема «$x \in W_x$» неразрешима;
   b) т. е. проблема «$\Phi_x = g$», где g — любая фиксированная вычислимая функция, неразрешима;
   c) т. е. проблема «$\Phi_x(y)$ определена» неразрешима;
   d) т. е. проблема «Φ_x всюду определена» неразрешима.

9. Выбрать верное объяснение теоремы: «Проблема «$\Phi_x = g$», где g — любая фиксированная вычислимая функция, неразрешима».
   Не существует общего метода проверки того:
   1) будет ли программа вычислять нулевую функцию;
   2) вычисляют ли две программы одну и ту же одноместную функцию;
   3) вычисляет ли программа конкретную функцию.

10. Если существует общерекурсивная функция $\psi_M(x)$, такая, что $a \in M$ тогда и только тогда, когда $a = \psi_M(x)$ для некоторого x, то множество M называется:
    1) разрешимым;
    2) разрешающим;
    3) перечислимым.

11. Перечислимое, но не разрешимое множество:
    1) существует;
    2) не существует.

12. Выбрать неверное утверждение.
    Из теоремы Райса следует, что:
    1) по номеру вычислимой функции f ничего нельзя узнать о свойствах этой функции;

2) *по синтаксису* программы ничего нельзя узнать о ее семантике;
3) не существует общего алгоритма для отладки программ.

13. Множество квадратов натуральных чисел $M = \{a \mid a = x_2\}$:
1) перечислимо, но не разрешимо;
2) перечислимо и разрешимо;
3) разрешимо, но не перечислимо.

Глава VII

Эффективные операции на множестве частичных функций

Развивая изложенную выше теорию, мы введем пон: эффективного или вычислимого оператора на множе всех частичных функций. Затем покажем, что опера сложения, вычитания, умножения и другие обычно опр ляемые (или стандартные) операции являются эффекти ми, если их рассматривать как операции над индексам трагиваемых объектов.

Изложенные в этой главе две теоремы о рекурсии им очень важные приложения к семантике языков програ рования. Именно на этот аспект и обращается серьезное мание.

7.1. Рекурсивные операторы

Обозначим через F_n ($n \geq 1$) класс всех частичных ф ций из $\mathbb{N}^n$ в $\mathbb{N}$. Словом *оператор* мы будем обозначать ф цию Φ: $F_m \to F_n$. Будем рассматривать лишь тотально о деленные операторы Φ, то есть такие операторы, обл определения которых совпадает с классом F_m.

Основная проблема при попытке дать определение п тию вычислимого (или эффективного) оператора Φ: F_1 - состоит в том, что как вводимая функция f, так и выв мая функция $\Phi(f)$ могут являться бесконечными объек и, следовательно, не могут быть заданы за конечное вр В то же время, согласно нашему интуитивному предста нию об эффективных процессах, мы должны каким-то с зом уметь проводить их за конечное время.

Пример 1. Чтобы разобраться, как можно преодо эту трудность, рассмотрим следующие операторы из F_1 в

$\Phi_1(f) = 2f$;

$\Phi_2(f) = g$, где $g(x) = \sum_{y \leq x} f(y)$.

Эти операторы достаточно просты и записаны в явном виде. Покажем, почему эти операторы можно считать эффективными на интуитивном уровне. Пусть $f \in F_1$ и $g_1 = \Phi_1(f)$. Заметим, что любое конкретное значение $g_1(x)$ (если оно определено) может быть вычислено за конечное время по одному-единственному значению $f(x)$ функции f. В случае $g_2 = \Phi_2(f)$ для вычисления значения $g_2(x)$ (если оно определено) необходимо знать конечное число значений $f(0)$, $f(1)$, ..., $f(x)$. То есть *в обоих случаях любое определенное значение* выводимой функции может быть эффективно вычислено за конечное время при использовании *лишь конечной информации* о вводимой функции f. △

Выявленную в примере 1 идею (работа с конечными объектами) реализуем на вводимом ниже определении рекурсивного оператора. Дадим сначала несколько вспомогательных определений.

Функция θ называется *конечной*, если ее область определения есть конечное множество.

Конечной частью функции f называется некоторая конечная функция θ, которая может быть продолжена до f (обозначается $\theta \subseteq f$).

Для удобства условимся, что всюду в этой главе буква θ обозначает конечную функцию. Закодируем каждую конечную функцию θ числом $\overline{\theta}$ следующим образом: набор из n чисел $x = (x_1, \ldots, x_n)$ кодируется числом

$$\langle x \rangle = p_1^{x_1+1} \cdot p_2^{x_2+1} \cdot \ldots \cdot p_n^{x_n+1},$$

где $p_1, p_2, \ldots, p_n$ — последовательные простые числа ($p_1 = 2$, $p_2 = 3$, ...). Для данной функции $\theta \in F_n$ положим

$$\overline{\theta} = \begin{cases} \prod\limits_{x \in Dom(\theta)} p_{\langle x \rangle}^{\theta(x)+1}, & \text{если } Dom(\theta) \neq \varnothing; \\ 0, & \text{если } Dom(\theta) = \varnothing \text{ (в этом случае } \theta = f_{\varnothing}). \end{cases}$$

Очевидно, что каждый набор $(x_1, \ldots, x_n)$ имеет свой уникальный код $\langle x \rangle$. Имеется простая эффективная процедура, с помощью которой можно для любого числа z определить, верно ли, что $z = \overline{\theta}$ для некоторой конечной функции θ, и если это так, то выяснить, принадлежит ли некоторое заданное x множеству $Dom(\theta)$. И если да, то вычислить значение $\theta(x)$. Закодировав таким образом каждую конечную функцию, мы теперь можем применить обычную теорию вычислимости.

Пусть $\Phi\colon F_m \to F_n$. Тогда Φ называется *рекурсивным оператором*, если существует вычислимая функция $\varphi(z, x)$, такая, что для всех $f \in F_m$, $x \in \mathbb{N}^n$, $y \in \mathbb{N}$ $\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\varphi(\overline{\theta}, x) = y$.

Пример 2. Оператор $\Phi(f) = 2f$ является рекурсивным оператором. Действительно, положим

$$\varphi(z, x) = \begin{cases} 2\theta(x), \text{ если } z = \overline{\theta} \text{ и } x \in Dom(\theta), \\ \text{не определена в остальных случаях.} \end{cases}$$

В силу тезиса Чёрча функция φ вычислима. Для любых f, x, y имеем:

$\Phi(f)(x) = y \Leftrightarrow x \in Dom(f)$ и $y = 2f(x) \Leftrightarrow$

существует $\theta \subseteq f$, для которой $x \in Dom(\theta)$ и $y = 2\theta(x) \Leftrightarrow$

существует $\theta \subseteq f$, для которой $\varphi(\overline{\theta}, x) = y$.

Отсюда следует, что Φ — рекурсивный оператор. △

Оператор $\Phi\colon F_m \to F_n$ называется *непрерывным*, если $\forall f \in F_m$ и $\forall x, y$ $\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\Phi(\theta)(x) = y$.

Оператор $\Phi\colon F_m \to F_n$ называется *монотонным*, если $\forall f, g \in F_m$ таких, что $f \subseteq g$, выполняется условие $\Phi(f) \subseteq \Phi(g)$.

Теорема 1. Любой рекурсивный оператор является непрерывным и монотонным.

Доказательство. Пусть $\Phi\colon F_m \to F_n$ — рекурсивный оператор, которому, по определению, соответствует вычислимая функция φ. Предположим, что $\Phi(f)(x) = y$, и пусть функция $\theta \subseteq f$ такова, что $\varphi(\overline{\theta}, x) = y$. Так как $\theta \subseteq \theta$, $\Phi(\theta)(x) = y$. Обратно, если $\theta \subseteq f$ и $\Phi(\theta)(x) = y$, то существует $\theta_1 \subseteq \theta$, такая, что $\varphi(\overline{\theta_1}, x) = y$, но тогда $\theta_1 \subseteq f$ и тем самым $\Phi(f)(x) = y$. Следовательно, оператор Φ непрерывен.

Пусть теперь $f \subseteq g$ и $\Phi(f)(x) = y$. Возьмем такую функцию $\theta \subseteq f$, что $\Phi(\theta)(x) = y$. Тогда $\theta \subseteq g$ и в силу непрерывности $\Phi(g)(x) = y$, то есть оператор Φ монотонен. □

7.2. Эффективные операции на вычислимых функциях

Известно много операций на вычислимых функциях или их областях, которые обычно определяются явно (произве-

дение, обращение вычислимых функций и т. д.). Можно ли считать эти операции эффективными? Ниже мы покажем для двух операций, что они являются эффективными, если рассматривать их как операции над индексами затрагиваемых объектов. Аналогично можно провести доказательства эффективности и других операций.

Теорема 2. Существует тотальная вычислимая функция $s(x, y)$, такая, что $\varphi_{s(x,y)} = \varphi_x \cdot \varphi_y$ $\forall x, y$.

Доказательство. Пусть $f(x, y, z) \simeq \varphi_x(z) \cdot \varphi_y(z) \simeq \simeq \psi_u(x, z) \cdot \psi(y, z)$. Таким образом, функция f вычислима и по s-m-n-теореме найдется тотальная вычислимая функция $s(x, y)$, такая, что $f(x, y, z) \simeq \varphi_{s(x,y)}(z)$, то есть $\varphi_{s(x,y)} = \varphi_x\varphi_y$. □

Полагая $g(x) = s(x, x)$, где s — функция из теоремы 2, имеем $(\varphi_x)^2 = \varphi_{g(x)}$.

Теорема 3. Существует тотальная вычислимая функция $s(x, y)$ такая, что $W_{s(x,y)} = W_x \cup W_y$.

Доказательство. Пусть

$$f(x, y, z) = \begin{cases} 1, \text{ если } z \in W_x \text{ или } z \in W_y, \\ \text{не определена в противном случае.} \end{cases}$$

Используя тезис Чёрча и вычислимость φ_x и φ_y, получаем вычислимость функции f, а значит, существование тотальной вычислимой функции $s(x, y)$, такой, что $f(x, y, z) \simeq \simeq \varphi_{s(x,y)}(z)$. Отсюда очевидно, что $W_{s(x,y)} = W_x \cup W_y$. □

Далее мы приведем без доказательства (их можно найти, например, в [5]) результаты Е. Майхилла и А. Шепердсона, которые показывают, что любой рекурсивный оператор, будучи ограничен областью вычислимых функций, дает на индексах эффективную операцию, а с другой стороны, таким образом возникают все эффективные операции на вычислимых функциях.

Теорема 4 (*Майхилл—Шепердсон, часть 1*).

Пусть Φ: $F_m \to F_n$ — рекурсивный оператор. Тогда существует тотальная вычислимая функция h, такая, что

$$\Phi(\varphi_e^{(m)}) = \varphi_{h(e)}^{(n)}, \; e \in \mathbb{N}.$$

Тотальная функция h: $\mathbb{N} \to \mathbb{N}$ называется *экстенсиональной*, если для любых натуральных чисел a, b из равенства $\varphi_a = \varphi_b$ следует $\varphi_{h(a)} = \varphi_{h(b)}$.

Отметим, что функция h, определяемая теоремой 4 для любого рекурсивного оператора Φ: $F_1 \to F_1$, является экстенсиональной.

Теорема 5 (***Майхилл—Шепердсон, часть 2***).

Пусть h — экстенсиональная тотальная вычислимая функция. Тогда существует единственный рекурсивный оператор Φ: $F_1 \to F_1$, такой, что $\Phi(\varphi_e) = \varphi_{h(e)}$ при всех $e \in \mathbb{N}$.

7.3. Первая теорема о рекурсии

Теорема 6 (***Клини, первая теорема о рекурсии***).

Пусть Φ: $F_m \to F_m$ — рекурсивный оператор. Тогда существует вычислимая функция f_Φ, которая является наименьшей неподвижной точкой для Φ, то есть

a) $\Phi(f_\Phi) = f_\Phi$;

b) если $\Phi(g) = g$, то $f_\Phi \subseteq g$.

Если функция f_Φ тотальна, то она является единственной неподвижной точкой Φ.

Доказательство. Используя свойства непрерывности и монотонности Φ, мы следующим образом построим наименьшую неподвижную точку f_Φ. Определим последовательность функций $\{f_n\}$, $n \in \mathbb{N}$:

$f_0 = f_\varnothing$ (функция с пустой областью определения);

$f_{n+1} = \Phi(f_n)$.

Тогда $f_0 = f_\varnothing \subseteq f_1$. Оператор Φ монотонный, значит, $\Phi(f_0) \subseteq \Phi(f_1)$, то есть $f_1 \subseteq f_2$. Таким образом получаем, что $f_n \subseteq f_{n+1}$ при всех n. Положим $f_\Phi = \bigcup\limits_{n \in \mathbb{N}} f_n$, то есть $f_\Phi(x) = y$ тогда и только тогда, когда $\exists n$ такое, что $f_n(x) = y$.

Покажем, что f_Φ есть неподвижная точка Φ. При всех n имеем $f_n \subseteq f_\Phi$, откуда $f_{n+1} = \Phi(f_n) \subseteq \Phi(f_\Phi)$, и, таким образом, $f_\Phi \subseteq \Phi(f_\Phi)$. Обратно, предположим, что $\Phi(f_\Phi) = y$; тогда существует конечная функция $\theta \subseteq f_\Phi$, такая, что $\Phi(\theta)(x) = y$. Возьмем n такое, что $\theta \subseteq f_n$. Тогда из непрерывности оператора следует, что $\Phi\,(f_n)(x) = y$, то есть $f_{n+1}(x) = y$. Отсюда $f_\Phi(x) = y$. Таким образом, $\Phi(f_\Phi) \subseteq f_\Phi$ и, значит, $\Phi(f_\Phi) = f_\Phi$.

Покажем теперь что f_Φ — наименьшая неподвижная точка Φ. Предположим, что $\Phi(g) = g$. Тогда $f_0 = f_\varnothing \subseteq g$ и по индукции мы получаем, что $f_n \subseteq g$ при всех n. Отсюда $f_\Phi \subseteq g$,

что и требовалось. Кроме того, если функция f_Φ тотальная, то $f_\Phi = g$, так что f_Φ — единственная неподвижная точка Ф.

Покажем, что функция f_Φ — вычислима. По теореме 5 существует тотальная вычислимая функция h такая, что $\Phi(\varphi_e) = \varphi_{h(e)}$ при всех e. Пусть e_0 — индекс f_0. Определим вычислимую функцию k соотношением

$k(0) = e_0$,

$k(n + 1) = h(k(n))$.

Тогда $f_n = \varphi_{k(n)}$ при всех n и, таким образо

$f_\Phi(x) = y \Leftrightarrow \exists n\ (\varphi_{k(n)}(x) = y)$.

Соотношение в правой части частично разрешимо, откуда и следует, что функция f_Φ вычислима. □

В этом доказательстве рекурсивность оператора Ф использовалась только для того, чтобы показать, что функция f_Φ вычислима. Первая же часть доказательства показывает, что любой непрерывный оператор имеет по крайней мере одну неподвижную точку.

Пример 3. Пусть Ф — рекурсивный оператор, задаваемый соотношениями

$\Phi(f)(0) = 1$;

$\Phi(f)(x + 1) \simeq f(x + 2)$.

Тогда наименьшей неподвижной точкой Ф будет

$$\begin{cases} f_\Phi(0) = 1, \\ f_\Phi(x+1) \text{ — не определено.} \end{cases}$$

Другие неподвижные точки Ф имеют вид

$$\begin{cases} f(0) = 1, \\ f(x+1) = a. \end{cases}$$ △

В предыдущих главах мы убедились, что определения, даваемые в терминах примитивной рекурсии, имеют смысл. В случае более сложных рекурсивных определений это не так очевидно — вполне можно представить себе, что функций, удовлетворяющих тому или иному определению, вообще не существует.

Именно здесь оказывается полезной первая теорема о рекурсии. Рекурсивные определения общего типа могут быть представлены уравнениями вида $f = \Phi(f)$, где Ф — некоторый рекурсивный оператор. Первая теорема о рекурсии показывает, что такое определение имеет смысл: всегда существует

даже вычислимая функция, которая ему удовлетворяет. Так как в математике требуется, чтобы определения описывали понятия однозначно, можно утверждать, что наше рекурсивное определение $f = \Phi(f)$ определяет наименьшую неподвижную точку оператора Φ. Таким образом, согласно первой теореме о рекурсии, класс вычислимых функций замкнут относительно рекурсивного определения общего типа.

7.4. Приложение к семантике языков программирования

В этом пункте мы познакомимся с тем, как первая теорема о рекурсии помогает решать проблему семантики программ. Изложение будет вестись в терминах некоторого общего, явно не описанного языка программирования L.

Основные символы языка L выбирают, имея в виду некоторый их конкретный смысл. Поэтому смысл будут иметь и более сложные выражения, построенные из них. Можно представить себе следующую простую программу вычисления функции. Пусть $g(x)$ есть некоторое выражение из L, такое, что если переменной x (возможно k-местной) придать некоторое конкретное значение a, то $g(a)$ может быть однозначно вычислено согласно правилам семантики языка L.

Если теперь взять в L символ функции f, который не входит в g, то

$$f(x) = g(x) \tag{1}$$

будет программой, реализующей функцию f_g, которая имеет следующий очевидный смысл: для любых чисел a значение $f_g(a)$ получается путем вычисления выражения $g(a)$ согласно правилам семантики языка L.

Допустим теперь, что g — такое выражение, в которое входит символ f. Будем в таком случае писать $g(f, x)$. Тогда программа (1) приобретет вид

$$f(x) = g(f, x). \tag{2}$$

Такие программы называются *рекурсивными*. Во многих случаях именно таким образом естественнее и экономнее описываются функции, которые мы хотим вычислить. Но в случае (2) смысл слова «программа» не вполне ясен. Основной вопрос для любой рекурсивной программы — как придать ей точный смысл? Существует два способа решения этого вопроса: вычислительный метод и метод неподвижной точки.

Вычислительный метод. Объясним его на примере. Рассмотрим рекурсивную программу:

$$f(x) = \begin{cases} 1, & \text{если } x = 0, \\ 2f(x-1), & \text{если } x > 0. \end{cases} \quad (3)$$

С помощью уравнения (3) можно формально вычислить значение $f(3)$: $f(3) \simeq 2 \cdot f(2) \simeq 2 \cdot 2 \cdot f(1) \simeq 2 \cdot 2 \cdot 2f(0) = 8$. Отсюда, если функция f_g определяется программой (3), то должно выполнятся равенство $f_g(3) = 8$.

Для более сложных рекурсивных программ может существовать несколько способов использования формального уравнения (2) в процессе вычисления. Рассмотрим, например, рекурсивную программу

$$f(x, y) = \begin{cases} 1, & \text{если } x = 0, \\ f(x-1, f(1,0)), & \text{если } x > 0. \end{cases} \quad (4)$$

Предположим, что мы хотим формально вычислить значение $f(1, 0)$. Имеем

$$f(1, 0) \simeq f(0, f(1, 0)). \quad (5)$$

В правой части f встречается дважды, и можно по-разному решить, какое из мест взять для подстановки $g(f, x)$.

Если взять левое f, то после соответствующих преобразований получим $f(1, 0) \simeq f(0, f(1, 0)) = 1$, так как $x = 0$.

Если же каждый раз заменять самый правый из символов f, то уравнение (5) дает

$$f(1, 0) \simeq f(0, f(1, 0)) \simeq f(0, f(0, f(1, 0))) \simeq$$
$$\simeq f(0, f(0, f(0, f(1, 0)))) \simeq \ldots$$

В этом случае значение $f(1, 0)$ формально вычислить не удается.

Правилом вычисления называется правило M, указывающее, как надо поступать, если в ходе вычислений возникла неоднозначность выбора возможных вариантов подстановки.

Для рекурсивной программы (4) мы сформулировали два правила: правило «крайнего левого» (LM) члена и правило «крайнего правого» (RM) члена. Возможны и другие правила. Для любого правила вычисления M рекурсивной программы (2) определим функцию $f_{g,M}$ следующим образом: $f_{g,M}(a)$ есть значение, полученное при формальном вычислении $f(a)$ с помощью правила M. Если при вычислении никакого значения не получено, то $f_{g,M}(a)$ считается неопределенным (для рекурсивной программы (4) $f_{g,LM}(1, 0) = 1$, а $f_{g,RM}(1, 0)$ — не определено).

Таким образом, каждое правило вычисления определяет метод реализации рекурсивной программы.

Приведенный выше пример показывает, что различные правила вычисления могут придать различный смысл каждой конкретной рекурсивной программе. Проблема, стоящая перед специалистом по ЭВМ, избравшим вычислительный метод, состоит в том, чтобы решить, какое из этих правил выбрать. Кроме того, для любого правила M возникает вопрос о том, в каком смысле функция $f_{g,M}$ удовлетворяет уравнению $f(x) = g(f, x)$.

Метод неподвижной точки, в котором используется первая теорема о рекурсии, дает возможность избежать этих вопросов.

Метод неподвижной точки. Выражение $g(f, x)$ языка L определяет оператор Φ: $F_n \to F_n$, такой, что

$$\Phi(f)(x) \simeq g(f, x)$$

для любой функции $f \in F_n$. Кроме того, в большей части языков программирования конечный и явный характер выражения $g(f, x)$ гарантирует, что оператор Φ рекурсивен. Тогда на основании первой теоремы о рекурсии заключаем, что Φ имеет вычислимую наименьшую неподвижную точку, которую мы можем обозначить через f_g. Таким образом, мы можем определить функцию, задаваемую программой (2), как f_g. Это логично, так как f_g — вычислимая функция, и, кроме того, $f_g(x) \simeq g(f_g, x)$.

Остается открытой проблема нахождения хороших практических процедур для реализации программы (2) в только что определенном смысле. Можно показать, что для любого правила вычисления M имеет место включение $f_{g,M} \subseteq f_g$. Кроме того, существуют правила вычисления M, для которых $f_{g,M} = f_g$ при всяком g. Любое из этих правил можно выбрать в качестве практического средства реализации рекурсивных программ. Отметим, что вычислительный метод и метод неподвижной точки скорее взаимно дополняют друг друга, нежели противоречат один другому: подход, основанный на неподвижной точке, с помощью первой теоремы о рекурсии дает теоретическое обоснование выбранного конкретного правила вычисления.

У метода неподвижной точки есть и другие преимущества: существует много способов доказательства корректности, эквивалентности и других свойств программ с семантикой неподвижной точки, и они могут быть строго обоснованы.

7.5. Вторая теорема о рекурсии

Первая теорема о рекурсии вместе с теоремой Майхилла—Шепердсона показывают, что для всякой экстенсиональной тотальной вычислимой функции f найдется число n, такое, что $\varphi_{f(n)} = \varphi_n$. Оказывается, что требование экстенсиональности не обязательно.

Теорема 7 (*вторая теорема о рекурсии*).
Пусть f — тотальная одноместная вычислимая функция. Тогда существует число n, такое, что $\varphi_{f(n)} = \varphi_n$.

Доказательство. По s-m-n-теореме существует тотальная вычислимая функция $s(x)$, такая, что для всех x

$$\varphi_{f(\varphi_x(x))}(y) \simeq \varphi_{s(x)}(y). \qquad (6)$$

Возьмем теперь какой-нибудь номер m функции s, то есть такое m, что $s = \varphi_m$. Переписывая (6), получаем

$$\varphi_{f(\varphi_x(x))}(y) \simeq \varphi_{\varphi_m(x)}(y).$$

Теперь, полагая $x = m$, а $\varphi_m(m) = n$ (левая часть равенства определена, так как функция $s = \varphi_m$ тотальна), получаем $\varphi_{f(n)}(y) = \varphi_n(y)$. □

Отметим, что функция f не индуцирует отображение $\varphi_x \to \varphi_{f(x)}$ на вычислимых функциях, для которого φ_n можно было бы назвать неподвижной точкой, то есть теорема 7 не является подлинной теоремой о неподвижной точке. Однако имеется индуцированное отображение программ, задаваемое равенством $f^*(P_x) = P_{f(x)}$. Требование неподвижной точки для f^* было бы слишком сильным — это должна была быть такая программа P_n, что $f^*(P_n)$ и P_n были бы тождественны, то есть $f(n) = n$. Но теорема 7 утверждает, что найдется такая программа P_n, что $f^*(P_n)$ и P_n дают одинаковый эффект в вычислении одноместных функций, то есть $\varphi_{f(n)} = \varphi_n$.

Всякое число n, такое, что $\varphi_{f(n)} = \varphi_n$ называют *неподвижной точкой*, или *значением неподвижной точки* для f.

В [6] можно найти различные обобщения теоремы 7. А мы рассмотрим некоторые следствия и применения второй теоремы о рекурсии.

Следствие 1. Если f — тотальная одноместная вычислимая функция, то существует такое число n, что $W_{f(n)} = W_n$ и $E_{f(n)} = E_n$.

Доказательство. Согласно теореме 7, существует число n, такое, что $\varphi_{f(n)} = \varphi_n$. Но если $\varphi_{f(n)} = \varphi_n$, то $W_{f(n)} = W_n$ и $E_{f(n)} = E_n$.□

Следствие 2. Если f — тотальная одноместная вычислимая функция, то существуют сколь угодно большие числа n, такие, что $\varphi_{f(n)} = \varphi_n$.

Доказательство. Фиксируем некоторое число k. Возьмем такое число c, что $\varphi_c \neq \varphi_0, \varphi_1, \ldots, \varphi_k$. Определим функцию g следующим образом:

$$g(x) = \begin{cases} c, & \text{если } x \leq k, \\ f(x), & \text{если } x > k. \end{cases}$$

Тогда функция g вычислима. Пусть n — неподвижная точка для g. Если $n \leq k$, то $\varphi_{g(n)} = \varphi_c \neq \varphi_n$, что невозможно. Значит, $n > k$. Тогда $f(n) = g(n)$ и n — неподвижная точка для f. □

Следствие 3. Пусть $f(x, y)$ — любая вычислимая функция. Тогда существует такой индекс e, что $\varphi_e(y) \simeq f(e, y)$.

Доказательство. Применим s-m-n-теорему, чтобы получить тотальную вычислимую функцию s, такую, что $\varphi_{s(k)}(y) \simeq f(x, y)$, а затем применим теорему 7, выбрав e в качестве неподвижной точки для s. □

Покажем теперь, что «естественная» нумерация вычислимых функций без повторений, которая была введена в главе VI, невычислима.

Теорема 8. Пусть f — тотальная возрастающая функция, такая, что

a) если $m \neq n$, то $\varphi_{f(m)} \neq \varphi_{f(n)}$;

b) $f(n)$ является наименьшим индексом $\varphi_{f(n)}$.

Тогда функция f невычислима.

Доказательство. Пусть f удовлетворяет условиям теоремы. Согласно a), функция $f(n)$ не может быть тождественной. Поэтому должно найтись число k, такое, что $f(n) > n$ при $n \geq k$, а тогда, согласно b), $\varphi_{f(n)} \neq \varphi_n$ при $n \geq k$. С другой стороны, если функция f вычислима, то, по следствию 2 теоремы 7, существует такое $n \geq k$, что $\varphi_{f(n)} = \varphi_n$, то есть мы получим противоречие. □

Пусть P — программа. Кодовое число $\gamma(P)$ можно рассматривать как описание P. Программу P можно было бы считать способной к *самовоспроизведению*, если для всякого

входа x вычисление $P(x)$ дает в результате собственное описание P, то есть $\gamma(P)$. Задача кажется невыполнимой, так как для построения самовоспроизводящейся программы P нам необходимо заранее знать $\gamma(P)$, а следовательно, и само P. Но ...

Теорема 9. Существует программа P, такая, что $P(x) \downarrow \gamma(P)$ для всех x, то есть программа P является самовоспроизводящейся.

Доказательство. Если положить $n = \gamma(P)$, то, по теореме 7, существует такое число n, что $\varphi_n(x) = n$ для всех x. Чтобы убедиться в этом, достаточно применить следствие 3 к функции $f(m, x) = x$. □

В программистских терминах этот результат можно сформулировать так: существует программа, которая при любом входе на выходе дает текст этой программы (подробнее, например, в [1]).

Тестовые задания

1. Пусть Φ: $F_m \to F_n$. Тогда Φ называется рекурсивным оператором, если:
   1) существует вычислимая функция $\varphi(z, x)$, такая, что для всех $f \in F_m$, $x \in \mathbb{N}^n$, $y \in \mathbb{N}$ $\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\varphi(\overline{\theta}, x) = y$;
   2) существует конечная функция $\theta \subseteq f$, такая, что $\Phi(\theta)(x) = y$;
   3) существует $\theta \subseteq f$, для которой $x \in Dom(\theta)$ и $y = 2\theta(x)$.

2. Поставить в соответствие. Оператор Φ: $F_m \to F_n$ называется:
   1) монотонным;
   2) непрерывным;

   a) если $\forall f, g \in F_m$ таких, что $f \subseteq g$, выполняется условие $\Phi(f) \subseteq \Phi(g)$;

   b) если существует вычислимая функция $\varphi(z, x)$, такая, что для всех $f \in F_m$, $x \in \mathbb{N}^n$, $y \in \mathbb{N}$ $\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\varphi(\overline{\theta}, x) = y$;

c) если $\forall f \in F_m$ и $\forall x, y$ $\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\Phi(\theta)(x) = y$.

3. Верно ли, что любой непрерывный оператор имеет по крайней мере одну неподвижную точку?
   1) да;
   2) нет.

4. Пусть g такое выражение, в которое входит символ f. Выбрать рекурсивную программу.
   1) $f(x) = g(f)(x)$;
   2) $f(x) = g(f, x)$;
   3) $f(x) = g(f(x))$.

5. По данной рекурсивной программе
   $$f(x) = \begin{cases} 1, & \text{если } x = 0, \\ 2f(x-1), & \text{если } x > 0, \end{cases}$$
   вычислить $f(2)$.
   1) 8;
   2) 4;
   3) 2.

6. Правило вычисления рекурсивной программы:
   1) придает смысл любой рекурсивной программе;
   2) определяет метод ее исполнения;
   3) все ответы верные.

7. Согласно первой теореме о рекурсии:
   1) любой рекурсивный оператор дает на индексах эффективную операцию;
   2) существует единственный рекурсивный оператор Φ, такой, что $\Phi(\varphi_e) = \varphi_{h(e)}$ при всех e;
   3) класс вычислимых функций замкнут относительно рекурсивного определения общего типа.

8. Всякое число n, такое, что $\varphi_{f(n)} = \varphi_n$, называют:
   1) неподвижной точкой;
   2) экстенсиональным;
   3) нет верного ответа.

9. Вычислительный метод и метод неподвижной точки:
   1) противоречат один другому;
   2) взаимодополняют друг друга;
   3) нет верного ответа.

Глава VIII

Сложность вычисления

В реальных вычислениях главный вопрос относительно функции f состоит не в том, вычислима ли f, а скорее в том, вычислима ли f практически. Иными словами, существует ли программа, вычисляющая функцию f за время (или в том объеме памяти), которым мы располагаем? Ответ отчасти зависит от мастерства в написании программ и возможностей компьютеров, но интуитивно понятно, что имеется дополнительный фактор, который можно назвать «внутренней сложностью функции f». Обсуждать подобные вопросы позволяет теория вычислительной сложности, элементам которой и посвящена эта глава.

Применяя подход, основанный на МНР, можно измерять время, требуемое для вычисления значения функции по конкретной программе в предположении, что каждый шаг МНР совершается за единицу времени. Таким образом, время вычисления является примером меры вычислительной сложности, отражающей сложность или эффективность используемой программы.

Сделаем одно важное замечание: в главе VI мы говорили о равносильности рассматриваемых подходов к понятию «алгоритм» и, следовательно, к равносильности соответствующих моделей вычислений в том смысле, что все они задают один и тот же класс функций. Но с точки зрения теории сложности, которая будет рассмотрена в этой главе, это далеко не так.

8.1. Меры сложности

Для каждой программы P через $t_P^{(n)}$ обозначим функцию

$$t_P^{(n)}(x) = \begin{cases} \text{число элементарных шагов, сделанных} \\ \text{при вычислении } f_P^{(n)}(x) \text{ по программе } P, \\ \text{если } f_P^{(n)}(x) \text{ определена,} \\ \\ \text{не определена в противном случае.} \end{cases}$$

Для удобства функцию $t_{P_e}^{(n)}$ для P_e будем обозначать через индекс программы: $t_e^{(n)}$. Как обычно, t_P обозначает $t_P^{(1)}$.

Семейство временных функций $t_e^{(n)}$ является примером **меры вычислительной сложности**. Некоторые простые, но важные свойства этих функций приводятся в следующей теореме, доказательство которой мы предлагаем провести читателю самостоятельно.

Теорема 1.

1) $Dom(t_e^{(n)}) = Dom(\Phi_e^{(n)})$ для всех n, $e \in \mathbb{N}$.

2) Для каждого n предикат $M(e, x, y) \equiv t_e^{(n)}(x) = y$ разрешим.

Отметим один интересный момент: второе свойство часто используется в теории сложности, и оно явно контрастирует с тем фактом, что $\Phi_e^{(n)}(x) = y$ является неразрешимым предикатом.

Часто в теории сложности рассматриваются свойства, справедливые для достаточно больших чисел n, но не обязательно для всех n. Так мы приходим к следующему определению.

Предикат $M(n)$ выполняется *для почти всех n* или *почти всюду* (п. в.), если $M(n)$ выполняется для всех n, кроме конечного множества (включая пустое) натуральных чисел (или эквивалентно, если существует n_0 такое, что $M(n)$ истинно для всех $n \geq n_0$).

Теперь можно установить существование произвольно сложных вычислимых функций.

Теорема 2. Пусть b — тотальная вычислимая функция. Существует тотальная вычислимая функция f, принимающая только значения 0 и 1, такая, что если e — индекс для f, то $t_e(n) > b(n)$ п. в.

Доказательство. Необходимую функцию f мы получим с помощью диагонального метода. Построение должно обеспечить следующее: если $t_i(m) \leq b(m)$ для бесконечно многих m, то f отличается от Φ_i хотя бы в одном из этих значений.

Определим f следующим образом. На каждом шаге n построения f мы либо определяем индекс i_n, либо за конечный промежуток времени решаем, что i_n не определено. Затем полагаем $f(n)$ отличающейся от $\Phi_{i_n}(n)$, если i_n определено.

Более подробно, предполагая, что значения $f(0), \ldots, f(n-1)$ определены, полагаем

$$i_n = \begin{cases} \mu i[i \le n,\ i \text{ отличается от всех ранее определенных} \\ \quad \text{значений } i_k \text{ и } t_i(n) \le b(n)], \text{если такое } i \text{ существует,} \\ \text{не определена в противном случае.} \end{cases}$$

$$f(n) = \begin{cases} 1, \text{ если } i_n \text{ определено и } \Phi_{i_n}(n) = 0, \\ 0, \text{ в противном случае.} \end{cases}$$

Существует конечная процедура, которая для данного i определяет, будет ли $t_i(m) \le b(m)$, так как

$$t_i(n) \le b(n) \Leftrightarrow \exists y \le b(n), \text{ такой, что } t_i(n) = y.$$

Предикат в правой части разрешим по второй части теоремы 1. Следовательно, существует эффективная процедура для решения вопроса о том, определено ли i_n, и если определено, то для нахождения этого числа. Кроме того, если i_n определено, то определена и функция $\Phi_{i_n}(n)$ в том смысле, что мы знаем ее номер. Следовательно, f — корректно определенная тотальная вычислимая функция. Значит, существует такое e, что $f = \Phi_e$.

По построению $\Phi_e(n) \ne \Phi_{i_n}(n)$, а следовательно, и $e \ne i_n$, если i_n определено. Покажем теперь, что если i является таким индексом, что $t_i(m) \le b(m)$ для бесконечно многих m, то $i = i_n$ для некоторого n, и, следовательно, $i \ne e$. Этого достаточно, чтобы показать, что $t_e(m) > b(m)$ для почти всех m.

Пусть $t_i(m) \le b(m)$ для бесконечно многих m. Пусть также

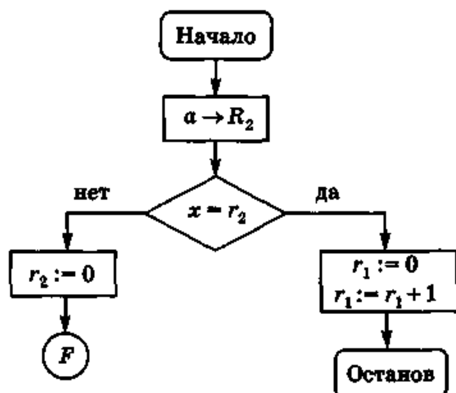
$$p = 1 + \max\{k \mid i_k \text{ определено и } i_k < i\},$$

если же нет ни одного определенного $i_k < i$, то положим $p = 0$. Возьмем такое n, что $n \ge i$, $n \ge p$ и $t_i(n) \le b(n)$.

Предположим, что $i \ne i_k$ для всех $k < n$. Тогда на n-м шаге построения функции f получаем: $i \le n$ и i отличается от всех ранее определенных i_k и $t_i(n) \le b(n)$. Значит, величина i_n определена и $i_n \le i$. Но поскольку $n \ge p$, то должно быть $i_n \ge i$. Следовательно, $i_n = i$. □

Вообще говоря, нельзя усилить эту теорему так, чтобы получить в заключение $t_e(n) > b(n)$ для всех n. Это связано с тем, что для каждой функции f всегда можно написать программу, быстро вычисляющую f на некотором конкретном значении аргумента a, просто внося значение $f(a)$ в программу.

Пусть, например, $f(a) = 1$. Пусть F — программа, вычисляющая f. Тогда программа F_1, основанная на приводимой блок-схеме, также вычисляет функцию f. Очевидно, что $t_{F_1}(a) = a + 3$ (так как действие $a \to R_2$ требует a шагов).



Таким образом, если b — такая вычислимая функция, что $b(x) > x + 3$ для некоторого x, то нельзя получить заключение теоремы 2 с $t_e(n) > b(n)$ для всех n.

Используя аналогичную идею, можно написать программу, которая быстро вычисляет f для любого данного конечного числа значений. Это показывает, что «$t_e(n) > b(n)$ п. в.» является наилучшим возможным заключением теоремы 2.

Можно ввести и другие естественные способы измерения сложности вычисления. Ограничиваясь, для простоты, одноместными функциями, приведем еще несколько примеров:

$$1.\quad \varphi_e(x) = \begin{cases} \text{число переходов, сделанных} \\ \text{при осуществлении } P_e(x), \text{ если } P_e(x)\downarrow, \\ \text{не определена в противном случае.} \end{cases}$$

Эта мера (мы полагаем здесь, что в команде перехода $J(m, n, p)$ переход к команде I_p происходит, если $r_m = r_n$; в противном случае перехода нет) тесно связана с числом петель, встречающихся при выполнении программы $P_e(x)$, которое в свою очередь связано с временем вычисления $t_e(x)$.

$$2.\quad \varphi_e(x) = \begin{cases} \text{максимальное число, содержащееся} \\ \text{в каком либо регистре машины за все время} \\ \text{вычисления } P_e(x), \text{ если } P_e(x)\downarrow, \\ \text{не определена в противном случае.} \end{cases}$$

Эта величина, очевидно, связана с объемом памяти, необходимым для того, чтобы провести вычисления $P_e(x)$ на реальном компьютере.

3. С подходом, основанным на машине Тьюринга, связаны еще две меры сложности:

a) число шагов в тьюринговом вычислении;

b) длина рабочей ленты, использованной в вычислении.

Можно показать, что для всех введенных выше мер сложности выполняется теорема 1, а значит, и теорема 2, так как при ее доказательстве мы использовали только свойства меры из теоремы 1. Такие результаты называют машинно-независимыми. Машинно-независимыми будут и следующие интересные результаты.

8.2. Теорема об ускорении

Итак, теперь мы можем определить вычислительную сложность программы. Попробуем ввести понятие сложности вычислимой функции f. Естественный путь выглядит так: выберем самую лучшую программу, вычисляющую эту функцию, и назовем сложностью функции f сложность этой выбранной программы.

Следующая теорема, приводимая нами без доказательства, показывает, что не существует наилучшей программы для вычисления функции f (доказательство можно найти, например, в [5]).

Теорема 3 (***Блюм, теорема об ускорении***). Пусть r — любая тотальная вычислимая функция. Существует тотальная вычислимая функция f, такая, что для любой программы P_i, вычисляющей f, существует другая программа P_k, также вычисляющая f и такая, что $r(t_k(x)) < t_i(x)$ п. в.

Если взять, например, $r(x) = 2x$, то получим $2t_k(x) < t_i(x)$, то есть программа P_k будет вычислять функцию f в два раза быстрее, чем программа P_i.

Вспоминая разные подходы к построению алгоритмической теории, теорему Блюма можно сформулировать так: существует такая вычислимая функция f, что для любой машины M_i, вычисляющей f, существует другая машина M_j, также вычисляющая f и причем быстрее, чем M_i.

Приведем забавное следствие теоремы об ускорении. Представим, что у нас есть МНР, которая делает 1 шаг за 1 секунду, заменим ее новой машиной, работающей в 100 раз быстрее. Тогда вычисление $P_k(x)$, требующее $t_k(x)$ секунд на старой машине, будет выполняться за $t_k(x)/100$ секунд на новой. Рассмотрим теперь функцию f, определяемую теоремой об ускорении с ускоряющим множителем 100. Предположим, что функция f вычисляется программой P_i на новой быстрой машине. По *теореме об ускорении существует программа* P_j *для* f, такая, что $100t_j(x) < t_i(x)$ почти всюду, то есть $t_j(x) < t_i(x)/100$. Таким образом, для почти всех x старая машина по программе P_j вычисляет f быстрее, чем новая машина по программе P_i. Отсюда мы делаем вывод, что по крайней мере для некоторых функций новая машина *не имеет* преимущества перед старой (для большинства входов)! Но «большинство» реальных вычислений всякой функции приходится выполнять для сравнительно небольших значений аргумента x. Если задано распределение вероятностей (или частот), с которыми встречаются разные значения x, то существует (или может быть эффективно найдена!) быстрейшая программа, вычисляющая f [15]. Отметим, что теорема об ускорении не является эффективной, то есть нет пути практического построения такой функции.

Вообще функция из теоремы об ускорении — это некоторая экзотика. Теорема Блюма доказывается построением с помощью техники диагонализации, и получающаяся функция не имеет никакого отношения к реальной практике вычислений. Но тем не менее, поскольку мы строим строгую теорию, то нужно искать другой подход к определению понятия сложности вычислимой функции.

Пусть b — любая тотальная вычислимая функция. С точки зрения сложности естественный класс функций составляют функции, у которых имеются программы с временем работы, не превосходящим функцию b. Мы приходим к следующему определению.

Пусть b — тотальная вычислимая функция. *Класс сложности* I_b функции b определяется следующим образом:

$$I_b = \{\Phi_e \mid \Phi_e \text{ тотальна и } t_e(x) \le b(x) \text{ п. в.}\} =$$
$$= \{f \mid f \text{ тотальна, вычислима и имеет программу } P_e, \text{ такую, что } t_e(x) \le b(x) \text{ п. в.}\}.$$

Очевидно, что аналогичные классы сложности можно определить и для других мер сложности.

Если b' — другая тотальная вычислимая функция такая, что $b'(x) \ge b(x)$ для всех x, то $I_{b'} \supseteq I_b$. В этом случае естест-

венно ожидать, что класс $I_{b'}$ содержит некоторые функции, не входящие в I_b, особенно если $b'(x)$ намного больше, чем $b(x)$. Следующая теорема показывает, что это не так: можно указать такие b и b', что b' значительно больше b (в любое число раз), но $I_{b'} = I_b$.

На самом деле эта теорема показывает, что функции b и b' можно выбрать так, что не существует временной функции сложности вычисления $t_e(x)$, которая лежит между $b(x)$ и $b'(x)$ для бесконечного множества разных x. Поэтому данная теорема называется теоремой о пробелах (или скачке).

Теорема 4 (***Бородин, теорема о пробелах***). Пусть r — тотальная вычислимая функция, для которой $r(x) \geq x$ при всех x. Тогда существует тотальная вычислимая функция b, такая, что

a) для каждого e и $x > e$ если $t_e(x)$ определено и $t_e(x) \geq b(x)$, то $t_e(x) > r(b(x))$;

b) $I_b = I_{r \circ b}$.

Доказательство. Определим неформально $b(x)$ следующим образом. Определим последовательность чисел $k_0 < k_1 < \ldots < k_x$ с помощью уравнений

$k_0 = 0$;

$k_{i+1} = r(k_i) + 1,\ 0 \leq i < x$.

Рассмотрим непересекающиеся промежутки $[k_i,\ r(k_i))$, $0 \leq i \leq x$. Существует $(x + 1)$ таких промежутков, а значит, по крайней мере один из них не содержит ни одного числа $t_e(x)$ для e, поскольку определено не больше, чем x таких чисел. Выберем i_x равным наименьшему i, такому, что $t_e(x) \notin [k_i, r(k_i))$ для всех $e < x$, и положим $b(x) = k_{i_x}$. Если теперь применить вторую часть теоремы 1, то получаем, что существует эффективная процедура для нахождения i_x (проверяем $t_e(x) \in [k_i, r(k_i))$ для разных e и i). Тогда, согласно тезису Чёрча, b — вычислимая функция. Если теперь предположить, что $x > e$ и $t_e(x) \geq b(x)$, то, по построению $b(x)$, мы имеем $t_e(x) \notin [b(x), r(b(x)))$, а значит, $t_e(x) > r(b(x))$ и мы доказали a).

Перейдем к пункту b). Очевидно, что $I_b \subseteq I_{r \circ b}$. Заметим, что если $f \in I_{r \circ b} \backslash I_b$, то f вычисляется по программе P_e с $t_e(x) \leq r(b(x))$ п. в., но $t_e(x) > b(x)$ для бесконечно многих x (в противном случае $f \in I_b$). Это, очевидно, противоречит пункту a). Следовательно, $I_b = I_{r \circ b}$. □

Легко видеть, что функция b в теореме может быть выбрана большей, чем любая заранее заданная вычислимая функция c, просто если положить $k_0 = c(x)$ вместо $k_0 = 0$ в доказательстве.

Из доказательства также видно, что теорема о пробелах машинно-независима.

8.3. Элементарные функции

В этом пункте мы введем класс элементарных функций как пример класса вычислимых функций, который очень просто может быть охарактеризован в терминах классов сложности, соответствующих времени вычисления. Как мы увидим, элементарные функции образуют широкий подкласс примитивно-рекурсивных функций. Они изучены довольно глубоко и представляют интерес сами по себе и помимо теории сложности.

Классом элементарных функций E называется наименьший класс, такой, что:

a) функции $x + 1, U_i^n$ $(1 \le i \le n)$, $x - y$, $x + y$, xy входят в E;

b) класс E замкнут относительно операции суперпозиции;

c) класс E замкнут относительно операций ограниченного суммирования и ограниченного произведения (т. е. если $f(x, z)$ принадлежит E, то и функции $\sum_{z<y} f(x, z)$ и $\prod_{z<y} f(x, z)$ принадлежат классу E).

Предикат $M(x)$ называется *элементарным*, если его характеристическая функция элементарна.

Грубо говоря, E является классом функций, которые можно получить итерацией операций обычной арифметики. Ясно, что элементарные функции вычислимы. Более того, согласно результатам третьей главы, они примитивно-рекурсивны. Следующая теорема позволяет построить некоторые примеры элементарных функций и предикатов.

Теорема 5. a) Класс E замкнут относительно ограниченной минимизации. b) Элементарные предикаты замкнуты относительно операций «не», «и», «или» и ограниченных кванторов «$\forall z < y$» и «$\exists z < y$».

Доказательство. Пусть функция $f(x, z)$ элементарна. Из результатов главы III следует, что

$$\mu z_{z<y}(f(x, z) = 0) = \sum_{v<y} \prod_{u<v} sg(f(x, u)).$$

Элементарность этой функции следует из элементарности sg, так как $sg(x) = x \dot- (x \dot- 1)$ и $1 = (x + 1) \dot- x$.

Случай b) очевиден. □

Можно показать, что практически все функции, перечисленные в теоремах и примерах III и V глав, элементарны.

Пример 1. Функция x^y — элементарна.

$$x^y = \prod_{i<y} x = \prod_{i<y} U_1^2(x, i).$$ △

Покажем теперь, что класс E замкнут относительно операции примитивной рекурсии, при условии, что мы заранее знаем элементарную границу для функции, определенной уравнением рекурсии.

Теорема 6. Пусть $f(x)$ и $g(x, y, z)$ элементарны, и пусть функция h определяется из f, g посредством

$h(x, 0) = f(x)$;

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

Предположим, что существует элементарная функция $b(x, y)$, такая, что $h(x, y) \le b(x, y)$ для всех x, y. Тогда функция h элементарна.

Доказательство. Фиксируем x, y. Тогда вычисление $h(x, y)$ обычным способом требует вычисления последовательности чисел $h(x, 0)$, $h(x, 1)$, ... , $h(x, y)$. Их можно закодировать одним числом s, где

$$s = 2^{h(x,0)} 3^{h(x,1)} \dots p_{y+1}^{h(x,y)} = \prod_{z \le y} p_{z+1}^{h(x,z)} \le \prod_{z \le y} p_{z+1}^{b(x,z)} = c(x, y),$$

где p_i — i-е простое число, $c(x, y)$ — элементарная функция. Обозначим через $(s)_i$ степень при p_i в разложении числа s. Ключевыми свойствами s являются:

a) $(s)_1 = h(x, 0) = f(x)$;

b) для $\forall z < y$ $(s)_{z+2} = h(x, z + 1) = g(x, z, (s)_{z+1})$;

c) $(s)_{y+1} = h(x, y)$.

Таким образом, получаем

$$h(x, y) = (\mu s_{s \le c(x,y)} [(s)_1 = f(x) \text{ и } \forall z < y \ (s)_{z+2} = g(x, z, (s)_{z+1})])_{y+1}.$$

Отсюда и из теоремы 5 следует, что функция h элементарна. □

Описанный в этой теореме способ определения функции называется ограниченной примитивной рекурсией.

Теперь можно сказать, что класс E замкнут относительно ограниченной рекурсии (другая формулировка теоремы 6). Следующее следствие этой теоремы показывает, что функции, вычислимые за элементарное время, элементарны.

Следствие. Предположим, что $b(x)$ — элементарная функция, а $\Phi_e^{(n)}$ — тотальная функция, такая, что $t_e^{(n)}(x) \le b(x)$ п. в. Тогда функция $\Phi_e^{(n)}$ элементарна.

Все конкретные примеры примитивно-рекурсивных функций из предыдущих глав оказались, как мы видели, элементарными. Единственное различие между этими классами состоит в том, что для класса E мы смогли доказать замкнутость относительно лишь ограниченной примитивной рекурсии. Как мы увидим ниже, это отличие принципиально: мы укажем примитивно-рекурсивную, но не элементарную функцию. Приведем эти результаты без доказательства.

Теорема 7. Если $f(x)$ — элементарная функция, то найдется такое число k, что для всех x

$$f(x) \le \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{\max(x)}}}}}}}_{k}$$

Следствие. Функция

$$f(x) \le \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{x}}}}}}}_{k}$$

примитивно-рекурсивна, но не элементарна.

И наконец, покажем связь между элементарными функциями и временем, за которое они могут быть вычислены.

Теорема 8. Если функция $f(x)$ элементарна, то существует вычисляющая ее программа P, такая, что $t_P^{(n)}(x)$ элементарна.

Обозначим через $b_k(z)$ число

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{z}}}}}}}_{k}.$$

Отметим, что $b_0(z) = z$, $b_1(z) = 2^z$ и, вообще, $b_{k+1}(z) = 2^{b_k(z)}$.

Теорема 9. Пусть $b(x)$ — тотальная функция и

$$I_b^* = \{\, g \mid g \text{ тотальна и } g = \Phi_e^{(n)} \text{ для некоторого } e,\ t_e^{(n)}(x) \le b(\max(x)) \text{ п. в.} \}.$$

Тотальная функция $f(x)$ элементарна тогда и только тогда, когда она вычислима за время, не большее b_k $(\max(x))$ для некоторого k, то есть

$$E = \bigcup_{k \geq 0} I^{*}_{b_k}.$$

Таким образом, мы полностью охарактеризовали элементарные функции как такие, которые можно вычислить за время

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2^{\max(x)}}}}}}\right\} k$$

для некоторого (зависящего от f) числа k. Со всеми доказательствами можно ознакомиться в [5].

Класс E содержит подавляющее большинство функций, встречающихся в практической математике. Он является первым естественным приложением к классу эффективно вычислимых функций, основанных на обычных операциях арифметики. В ряде работ [15] было обосновано, что E содержит все практически вычислимые функции.

Особое значение в плане практической вычислимости имеет класс функций P, вычислимых за время, ограниченное каким-либо полиномом от длины k-ичной записи ($k > 1$) аргумента, то есть объема входной информации. Изучением этого класса и совершенно нового класса NP (функции, вычислимые на недетерминированной машине Тьюринга за полиномиальное от длины входа время) мы займемся в следующей главе.

Тестовые задания

1. Мерой вычислительной сложности, отражающей сложность и эффективность используемой программы, является:
   1) время вычисления;
   2) длина кода программы;
   3) количество подпрограмм.

2. Чем является в неравенстве $t_e(n) > b(n)$ п. в. функция $t_e^{(n)}$?
   1) вычислимой функцией;
   2) семейством временных функций;
   3) нет верного ответа.

3. Предикат $M(n)$ выполняется для почти всех n или почти всюду (п. в.), если
   1) $M(n)$ является неразрешимым предикатом;
   2) если существует n_0, такое, что $M(n)$ истинно для всех $n \geq n_0$;
   3) $Dom(M(n)) = Dom(\Phi_e^{(n)})$ для всех n, e.
4. Существует ли тотальная вычислимая функция f, принимающая только значения 0, 1, такая, что если e — индекс для f, то $t_e(n) > b(n)$ п. в.?
   1) да;
   2) нет.
5. Теорема Блюма об ускорении показывает, что:
   1) не существует наилучшей программы для вычисления функции f;
   2) существует наилучшая программа для вычисления функции f;
   3) не существует вычислимой функции f для любой машины M_i.
6. Теорема о пробелах:
   1) машинно-зависима;
   2) машинно-независима.
7. Предикат $M(x)$ называется элементарным, если:
   1) принадлежит классу элементарных функций;
   2) его характеристическая функция элементарна;
   3) нет верного ответа.
8. Класс элементарных функций замкнут относительно ограниченной рекурсии?
   1) да;
   2) нет.

Глава IX

Введение в теорию NP-полных задач

В данной главе мы сначала рассматриваем формальные языки и грамматики. Затем вводим задачи распознавания, показываем, что хорошо известные читателю задачи, например дискретной математики, легко к ним сводятся.

Далее мы рассматриваем хорошо знакомую нам машину Тьюринга как детерминированную машину и вводим понятие недетерминированной машины Тьюринга за счет отказа от условия однозначности перехода в процессе работы машины. Определяются два основных класса задач распознавания: P — множество задач, для решения которых существует полиномиальная детерминированная машина Тьюринга, и NP — множество задач, для решения которых существует полиномиальная недетерминированная машина Тьюринга. Возникает одна из самых знаменитых современных нерешенных проблем: как соотносятся множества P и NP?

Мы вводим классы P и NP на основе машин Тьюринга. Однако отметим, что эти классы инвариантны относительно других рассмотренных нами моделей вычислений (и вообще, относительно любой «разумной» модели вычислений). Это исключительно важно в теории NP-полноты.

Выделение полиномиальных алгоритмов в специальный класс связанно с тезисом теории сложности: ***реально выполняемые алгоритмы отождествляются с полиномиальными алгоритмами***. Именно этот тезис и дает возможность применять теорию NP-полноты в практическом программировании.

В заключение приведены примеры шести самых известных NP-полных задач.

9.1. Формальные языки и грамматики

Вычислительный процесс может рассматриваться как преобразование входного множества строк в выходное мно-

жество строк. Такие процессы ведут себя совершенно определенным образом, и, следовательно, с ними можно обращаться как с математическими объектами.

9.1.1. Основные понятия

Буква (или *символ*) — это простой неделимый знак. Множество всех букв образует *алфавит*.

Пример 1

$A = \{a, b, c\}$, $B = \{0, 1\}$,

$C = \{\text{begin, end, if, then, else, ...}\}$,

$E = \{a, b, c, ..., x, y, z\}$.

Здесь мы можем рассматривать B как бинарный алфавит, C — как алфавит языка PASCAL (в котором слова типа begin не могут быть разделены), а E — как английский алфавит. △

Строка — это упорядоченная совокупность букв алфавита (например, алфавита A), и следовательно выглядит подобно элементам $A^n = A \times A \times \ldots \times A$. Однако строки принято записывать в виде $a_1 a_2 ... a_n$ в отличие от принятой записи для элементов декартова произведения $(a_1, a_2, ..., a_n)$. Буквы сами по себе также являются строками для случая $n = 1$. Будем допускать также случай, когда строка не имеет букв (*пустая строка*), и обозначать эту строку через Λ.

Множество всех строк над алфавитом A называют *замыканием* A и обозначают A^*, так что

$$A^* = \bigcup_{n=0}^{\infty} A^n, \qquad \text{где } A^0 = \{\Lambda\}.$$

Для удобства вводится также обозначение для множества всех непустых строк

$$A^+ = A^* \setminus \{\Lambda\}.$$

Язык L над алфавитом A — это заданное множество строк в A^*, поэтому $L \subseteq A^*$. Операции над строками индуцируют аналогичные операции на языках.

Слова могут составляться в предложения, а множество всех предложений, имеющих смысл, образует язык. Нас будут в основном интересовать искусственные языки, такие

как языки программирования или языки, описывающие правильные математические выражения, однако вначале рассмотрим случай английского языка. Возьмем предложение

The dog bit me.

Это предложение можно рассматривать двумя способами. Во-первых, изучать его как простую совокупность слов, каждое из которых является упорядоченной совокупностью букв; в этом случае предложение рассматривается синтаксически. Во-вторых, интерпретировать предложение, считая, что мы понимаем значения слов и их внутренние связи, тогда мы получаем семантику — значение предложения. Если мы произносим предложение, то оно влияет на нас своим воздействием — прагматика. В совокупности эти три области образуют семиотику языка.

Основным объектом нашего рассмотрения будет область синтаксиса.

Напомним, что для заданного алфавита A язык L является произвольным подмножеством множества A^*, однако произвольные подмножества представляют очень незначительный интерес. Мы хотим сосредоточить внимание на специальных языках, содержащих строки, которые благодаря внешней информации об их семантике считаются осмысленными или хорошо сконструированными.

Наиболее интересные языки бесконечны и, следовательно, не могут быть описаны явно. В этих случаях надо придумать способы порождения языка; грамматика G может рассматриваться как такая порождающая система. Сформулируем две основные задачи теории формальных языков:

1) порождение по заданной грамматике и связанным с ней языком предложения в языке;
2) определение по заданной грамматике принадлежности заданной строки языку.

Для того чтобы проверить, входит ли строка в L, надо знать как L порождается грамматикой G. Далее через $L(G)$ будем обозначать язык, порожденный грамматикой G. Процедура проверки вхождения строки α в язык $L(G)$ называется грамматическим разбором. Обычно правила грамматики подбирают таким образом, чтобы можно было применить известные процедуры грамматического разбора.

9.1.2. Грамматики с фразовой структурой

Грамматикой с фразовой структурой (ГФС) G называется алгебраическая структура, состоящая из упорядоченной четверки $\{N, T, P, S\}$, где:

- N и T — непустые конечные алфавиты *нетерминальных* и *терминальных символов* соответственно (называемых также нетерминалами и терминалами) таких, что $N \cap T = \varnothing$;
- P — конечное множество *продукций*, $P \subset V^{+} \times V^{*}$, где $V = N \cup T$ называется *словарем* грамматики G;
- $S \in N$ — *начальный символ* или *источник*.

Терминальный алфавит — это алфавит языка объекта. Нетерминальный алфавит — вспомогательные символы, используемые, как правило, для обозначения различных конструкций языка. Нетерминальные символы отсутствуют в конечном описании объекта.

Предполагая, что с имвол «$\rightarrow$» не содержится в V, соотношения $(\alpha, \beta) \in P$ обычно записывают в виде $\alpha \rightarrow \beta$.

Понятие продукции, которое также называют *правилом преобразования*, должно давать возможность заменять одну строку символов другой.

Пусть $\alpha, \beta \in V^{*}$; тогда β *прямо выводится* из α, если $\alpha = \gamma\sigma\delta$ и $\beta = \gamma\rho\delta$, где $\gamma, \delta, \rho \in V^{*}$, $\sigma \in V^{+}$ и $\sigma \rightarrow \rho \in P$. Этот факт будем записывать в виде $\alpha \Rightarrow \beta$; он может формально рассматриваться как преобразование строки α в строку β замещением подстроки σ на ρ.

Пусть α и β — слова над V, и существует конечная последовательность $\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = \beta$, тогда говорят, что α *порождает* β (записывается $\alpha \overset{*}{\Rightarrow} \beta$).

Строку α называют *сентенциальной формой*, если $S \overset{*}{\Rightarrow} \alpha$. Сентенциальная форма $\alpha \in T^{*}$ называется *предложением, порожденным* G. Таким образом *язык*, порожденный грамматикой G, есть $L(G) = \{\alpha\colon \alpha \in T^{*},\ S \overset{*}{\Rightarrow} \alpha\}$.

Применяя продукции к сентенциальным формам, можно действовать достаточно произвольно, поэтому возможно существование нескольких допустимых выводимых последовательностей для заданного выражения. Последовательность вывода, которая оперирует с самой левой из возмож-

ных подстрок, называется (левой) *канонической выводной последовательностью* предложения.

Пример 2. Пусть

$$G = (\{B\}, \{(,)\}, P, B),$$

где

$$P = \{P_1 = B \to (B),\ P_2 = B \to BB,\ P_3 = B \to ()\}.$$

Тогда предложение ()(()()) может быть выведено многими способами. Приведем четыре из них:

1) $B \Rightarrow BB \Rightarrow B(B) \Rightarrow ()(B) \Rightarrow ()(BB) \Rightarrow ()(()B) \Rightarrow ()(()())$;
2) $B \Rightarrow BB \Rightarrow B(B) \Rightarrow ()(B) \Rightarrow ()(BB) \Rightarrow ()(B()) \Rightarrow ()(()())$;
3) $B \Rightarrow BB \Rightarrow ()B \Rightarrow ()(B) \Rightarrow ()(BB) \Rightarrow ()(()B) \Rightarrow ()(()())$;
4) $B \Rightarrow BB \Rightarrow ()B \Rightarrow ()(B) \Rightarrow ()(BB) \Rightarrow ()(B()) \Rightarrow ()(()())$.

Есть ли канонический вывод среди приведенных выводов? △

9.1.3. Иерархия Хомского

Введенное понятие языка является достаточно общим описательным средством. Наложение различных ограничений на структуру грамматики $G = \{N, T, P, S\}$ позволяет построить более богатые теории языка, незначительно сократив его описательные возможности.

Грамматику с фразовой структурой без дополнительных ограничений называют *грамматикой Хомского типа* 0.

Если же все элементы P получаются из формы $\alpha \to \beta$, где $\alpha = \gamma_1 x \gamma_2$ и $\beta = \gamma_1 \delta \gamma_2$, γ_1, $\gamma_2 \in V^*$ и $x \in N$, $\delta \in V^+$, то говорят, что G является *контекстно-зависимой грамматикой* (КЗГ) или *грамматикой Хомского типа* 1. В этом определении строки γ_1 и γ_2 могут рассматриваться как контекст, в котором x может заменяться посредством δ.

Если подстановки могут быть выполнены без рассмотрения контекстов, тогда мы можем заменить «контексты» γ_1 и γ_2 пустой строкой Λ и получить более слабое ограничение: если $x \to \delta \in P$, то $x \in N$, $\delta \in V^+$. В этом случае грамматику G называют *контекстно-свободной* (КСГ) или *грамматикой Хомского типа* 2.

Наконец, если P состоит только из продукций вида $x \to \delta$, где $x \in N$, $\delta \in T \cup TN$ (т. е. правая часть представляет собой или один терминальный символ, или один терминальный символ, за которым следует один нетерминальный

символ). В этом случае грамматику G называют *регулярной* (РГ) или *грамматикой Хомского типа* 3.

Языки, порожденные каким-либо из этих типов грамматик, имеют аналогичные названия. Большинство практических языков является некоторыми расширениями контекстно-зависимых языков. Однако в практических вычислениях их первоначально рассматривают как контекстно-свободные или регулярные языки, применяя на последующих этапах семантические критерии для разрешения возможных неоднозначностей.

Пример 3. Построить грамматику для языка $\{x^n y^n z^n\colon n \in \mathbb{N}\}$ и определить ее тип.

Покажем, что порождающей грамматикой рассматриваемого языка является $G = \{N, T, P, S\}$, где $N = \{S, Y, Z\}$, $T = \{x, y, z\}$, $P = \{P_1 = S \to xSYZ,\ P_2 = S \to xYZ,\ P_3 = xY \to xy,\ P_4 = yY \to yy,\ P_5 = yZ \to yz,\ P_6 = ZY \to YZ,\ P_7 = zZ \to zz\}$.

Вначале заметим, что для любого $n \in \mathbb{N}$ мы можем получить

$$S \overset{(P_1)}{\overset{*}{\Rightarrow}} x^{n-1}S(YZ)^{n-1} \overset{(P_2)}{\Rightarrow} x^n(YZ)^n \overset{(P_6)}{\overset{*}{\Rightarrow}} x^nY^nZ^n \overset{(P_3)}{\Rightarrow} x^nyY^{n-1}Z^n \overset{(P_4)}{\overset{*}{\Rightarrow}}$$

$$\overset{(P_4)}{\overset{*}{\Rightarrow}} x^ny^nZ^n \overset{(P_5)}{\Rightarrow} x^ny^nzZ^{n-1} \overset{(P_7)}{\overset{*}{\Rightarrow}} x^ny^nz^n;$$

поэтому

$$\{x^ny^nz^n\colon n \in \mathbb{N}\} \subset L(G).$$

Теперь надо показать, что никакие другие строки не могут быть порождены G. Хотя возможны изменения в порядке применения правил (P_1), (P_2) и (P_6), любое предложение должно выводиться посредством сентенциальной формы $x^nYZ\alpha$, где α состоит из $n-1$ символов Y и $n-1$ символов Z. Для того чтобы получить строку над T, мы должны использовать правила (P_4), (P_5) и (P_7). При этом (P_7) может преобразовывать Z в z только в контексте zZ, а (P_5) делать то же преобразование только в контексте yZ. Таким образом, терминальный символ z не может появиться в строке раньше терминального символа y.

Аналогично, для замены Y на y при помощи правил (P_4) и (P_3) требуются контексты yY и xY соответственно. Следовательно, терминальный символ y не может появиться в строке раньше терминального символа x.

Таким образом, предложенный язык можно рассматривать как контекстно-зависимый. △

9.2. Задачи распознавания, языки и кодирование

Для удобства изложения теория NP-полных задач обычно строится для задач распознавания свойств. Такие задачи имеют только два возможных решения — «да» или «нет». Иначе говоря, задача распознавания П состоит из двух множеств:

D_Π — множество всех возможных индивидуальных задач;
Y_Π — множество индивидуальных задач с ответом «да» ($Y_\Pi \in D_\Pi$).

Как мы уже отмечали в первой главе, П — это массовая задача, которая содержит несколько параметров, конкретные значения которых неопределенны, а I — индивидуальная задача, полученная из массовой задачи П, когда всем параметрам задачи П присвоены конкретные значения.

Стандартная форма описания задач, которую мы будем использовать, состоит из двух частей:

1) описание **условия** задачи в терминах различных компонент — множеств, графов, функций, чисел и т. д.;

2) в терминах условия формируется **вопрос**, предполагающий один из двух ответов — «да» или «нет».

Это описание следующим образом определяет множества D_Π и Y_Π. Индивидуальная задача принадлежит D_Π тогда и только тогда, когда она может быть получена из стандартной формы описания подстановкой конкретных значений во все компоненты условия. Индивидуальная задача принадлежит Y_Π тогда и только тогда, когда ответом на вопрос задачи будет «да».

Пример 4. Сформулируем задачу распознавания, соответствующую известной из курса дискретной математики задаче о коммивояжере.

Условие. Заданы: конечное множество $C = \{c_1, c_2, \ldots, c_n\}$ «городов», «расстояние» $d(c_i, c_j) \in \mathbb{N}^+$ для каждой пары городов c_i, $c_j \in C$ и граница $B \in \mathbb{N}^+$.

Вопрос. Существует ли «маршрут», проходящий через все города из C, длина которого не превосходит B? Другими

словами, существует ли последовательность $\{c_{k_1}, c_{k_2}, \ldots, c_{k_n}\}$ элементов C, такая, что

$$\sum_{i=1}^{n-1} d(c_{k_i}, c_{k_{i+1}}) + d(c_{k_n}, c_{k_1}) \le B? \qquad \triangle$$

Читатель, конечно, знаком с задачей коммивояжера, но в оптимизационной формулировке (найти «маршрут» минимальной длины). Не представляет особого труда показать, как оптимизационная задача сводится к задаче распознавания.

Отметим важный момент. Так как значение целевой функции легко оценить, то задача распознавания не может быть сложнее соответствующей задачи оптимизации. То есть если для задачи о коммивояжере можно за какое-то время найти маршрут минимальной длины, то совершенно ясно, как практически за минимальное время решить соответствующую задачу распознавания: найти маршрут минимальной длины, вычислить его длину и сравнить ее с заданной границей B. Следовательно, выводы, которые будут получены в этой главе для задач распознавания, вполне применимы и к задачам оптимизации.

Перейдем к формальному описанию задач распознавания.

Как мы уже знаем, если множество Σ — алфавит, то Σ^* — множество всех конечных цепочек из символов алфавита Σ (слова), а любое подмножество $L \subseteq \Sigma^*$ — язык в алфавите Σ.

Установим теперь соответствие между задачами распознавания и языками с помощью схем кодирования, которые обычно применяются для представления индивидуальной задачи при ее решении на компьютере. *Схема кодирования e* задачи Π описывает каждую индивидуальную задачу из Π подходящим словом в некотором фиксированном алфавите Σ. Таким образом, задача Π и схема кодирования e задачи Π разбивают слова из Σ^* на три класса:

1) слова, не являющиеся кодами индивидуальных задач из Π;
2) слова, являющиеся кодами индивидуальных задач из Π с отрицательным ответом на вопрос;
3) слова, являющиеся кодами индивидуальных задач из Π с положительным ответом на вопрос.

Третий класс слов как раз и есть тот язык, который ставится в соответствие задаче П при кодировании e и обозначается через $L[\Pi, e]$.

$$L[\Pi, e] = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} \Sigma \text{ есть алфавит схемы кодирования } e, \\ x - \text{код индивидуальной задачи} \\ I \in Y_\Pi \text{ при схеме кодирования } e \end{array} \right\}$$

При применении формальной теории NP-полноты к задачам распознавания будем говорить, что некоторый результат имеет место для задачи П при схеме кодирования e, если этот результат имеет место для языка $L[\Pi, e]$.

Обычно схему кодирования считают «разумной», если она:

1) достаточно «сжатая» — то есть при кодировании индивидуальной задачи сохраняется ее естественная краткость;

2) допускает декодирование — то есть по данной компоненте условия задачи можно указать алгоритм с полиномиальным временем работы, который из любого заданного кода индивидуальной задачи позволяет извлечь описание этой компоненты.

Очевидно, что уточнение понятия «разумное кодирование» не является формальным определением, но нам этого будет достаточно для дальнейшей работы. Отметим, что нам вообще не известен удовлетворительный способ дать такое определение. Подробно об этом можно прочитать в [2].

Отметим, что если ограничиться только «разумными схемами кодирования», то наши результаты не будут зависеть от способа кодирования. В этом случае теряется связь с важнейшим формальным параметром «длина входа». Поэтому необходим и некоторый параметр, через который можно выразить временную сложность. Будем считать, что с каждой задачей распознавания связана не зависящая от схемы кодирования функция Length: $D_\Pi \to \mathbb{N}^+$, которая «полиномиально эквивалентна» длине кода индивидуальной задачи, получаемой при любой разумной схеме кодирования.

Полиномиальная эквивалентность понимается в следующем смысле: для любой разумной схемы кодирования e задачи П существуют два полинома p_1 и p_2, такие, что если $I \in D_\Pi$ и слово x есть код индивидуальной задачи I при кодировании e, то

$$\text{Length}[I] \le p_1(|x|) \text{ и } |x| \le p_2(\text{Length}[I]),$$

где $|x|$ — длина слова x.

Пример 5. В задаче КОММИВОЯЖЕР можно положить

$\text{Length}[I] = m + [\log_2 B] + \max\{[\log_2 d(c_i, c_j)] \mid c_i, c_j \in C\}$. △

Так как любые две разумные схемы кодирования задачи П дают полиноминально эквивалентные длины входов, то диапазон для выбора функции Length очень широк, а получаемые результаты будут верны для любой функции Length, удовлетворяющей сформированным выше условиям.

9.3. Детерминированные машины Тьюринга и класс Р

Во второй главе для формализации понятия алгоритм мы ввели машины Тьюринга, которые обладали свойством детерминированности. В дальнейшем мы их так и будем называть — детерминированные машины Тьюринга (ДМТ). Введем еще некоторые обозначения и соглашения, удобные для дальнейшего изложения.

Любая ДМТ имеет два заключительных состояния q_y и q_n (от Y — да, N — нет).

Будем говорить, что ДМТ T, имеющая входной алфавит Σ, *принимает* $x \in \Sigma^*$, если, будучи примененной к входу x, она оказывается в состоянии q_y.

Язык L_T, распознаваемый ДМТ T, задается следующим условием

$L_T = \{x \in \Sigma^* \mid T \text{ принимает } x\}$.

Соответствие между «распознаванием» языков и «решением» задач распознавания определяется следующим образом.

Будем говорить, что ДМТ T решает задачу распознавания П при кодировании e, если T останавливается на всех словах, составленных из букв входного алфавита, и $L_T = L[\Pi, e]$.

Время, требуемое ДМТ T для вычисления при входе x, есть число шагов, выполняемых до момента остановки.

Если ДМТ T останавливается на входах $x \in \Sigma^*$, то *временной сложностью* этой машины T называется функция T_T: $\mathbb{N}^+ \to \mathbb{N}^+$,

$$T_T(n) = \max\left\{ m = \left| \begin{matrix} \text{существует такое слово } x \in \Sigma^*, |x| = n, \\ \text{что вычисление машины } T \text{ на входе } x \\ \text{требует времени } m \end{matrix} \right. \right\}.$$

Детерминированная машина Тьюринга T называет *полиномиальной*, если существует такой полином p, что для всех $n \in \mathbb{N}^+$ $T_T(n) \le p(n)$.

Класс языков P определяется так:

$$\mathbf{P} = \left\{ L \left| \begin{matrix} \text{существует полиномиальная ДМТ,} \\ \text{для которой } L = L_T \end{matrix} \right. \right\}.$$

Будем говорить, что задача распознавания П принадлежит классу P при кодировании e, если $L[\Pi, e] \in P$, то есть существует полиномиальная ДМТ, которая решает задачу П при кодировании e.

Учитывая приведенные выше рассуждения об эквивалентности разумных схем кодирования, мы не будем приводить конкретные схемы кодирования, а будем просто говорить, что задача распознавания П принадлежит классу P.

9.4. Недетерминированные вычисления и класс NP

Недетерминированная машина Тьюринга (НДМТ) моделирует алгоритмы с некоторой «свободой выбора», причем нас интересует, сколько времени понадобится, если с выбором «всегда будет везти». По крайней мере, некоторые команды НДМТ при одной и той же истории ее работы и, значит, при одном и том же ее состоянии могут выполняться разными способами. Для каждого внутреннего состояния машины и читаемого с ленты символа эти способы заданы.

У НДМТ также имеется конечный набор внутренних состояний $(q_1, q_2, \ldots, q_n)$ и алфавит символов на ленте $\{a_1, a_2, \ldots, a_m\}$. Однако правила действий имеют вид списков:

$$q_i a_j \rightarrow \begin{cases} q_{i_1} a_{j_1} d_{k_1} \\ q_{i_2} a_{j_2} d_{k_2} \\ \ldots \\ q_{i_l} a_{j_l} d_{k_l} \end{cases},$$

где l — это максимальное число вариантов выполнения действия. Если для некоторых внутренних состояний q_i и читаемых символов a_j вариантов меньше, то последний будет дублироваться так, чтобы их стало l.

Перед выполнением очередного действия из одной машины возникает l машин. Записи на их лентах одинаковы — такие, какие были у их «предка», но с этого момента их пути расходятся: каждая машина выполняет свой вариант действия из списка. Процесс работы ДМТ может быть изображен в виде линейной последовательности действий, а процесс работы НДМТ — в виде дерева, вершинами которого являются выполняемые действия, а ребрами — переходы от одного действия к другому (рис. 9.1). Таким образом, она одновременно решает задачу разными способами, чтобы не упустить тот, при котором «везет».

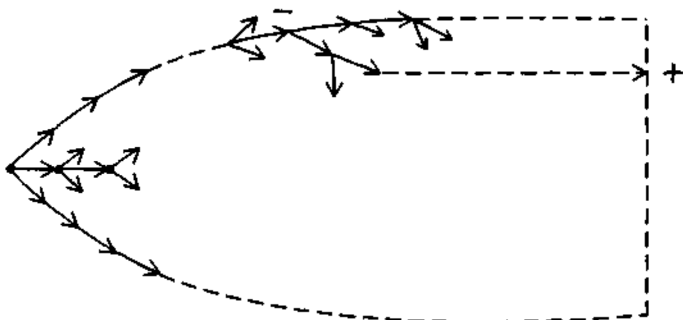


Рис. 9.1. Схема НДТМ

Если по некоторой цепи действия заканчиваются, то возможны два исхода: задача решена (управляющее устройство перешло в состояние q_y), или решение не найдено (управляющее устройство перешло в состояние q_n).

В первом случае все остальные машины, размножившиеся к этому моменту, тоже кончают работу и превращаются в одну машину. Тогда говорят, что вычисление было *принимающим*.

Во втором случае все остальные машины продолжают работу.

Если же по всем цепям произойдет окончание вычисления с отрицательными результатами, то все машины тоже сливаются в одну: задача не имеет решения. В этом случае говорят, что вычисление *непринимающее*.

Будем говорить, что НДМТ *принимает вход* x, если, по крайней мере, одно из ее вычислений на входе x является принимающим.

. *Язык, распознаваемый* НДМТ T, — это язык

$$L_T = \{x \in \Sigma^* \mid T \text{ принимает } x\}.$$

Время, требующееся НДМТ T для того, чтобы принять слово $x \in L_T$, — это минимальное число шагов, выполненных T до достижения заключительного состояния q_y, где минимум берется по всем принимающим вычислениям машины T на входе x.

Временная сложность НДМТ T — это функция T_T: $\mathbb{N}^+ \to \mathbb{N}^+$, определяемая следующим образом:

$$T_T(n) = \max\left(\{1\} \cup \left\{m \,\middle|\, \begin{matrix}\text{существует } x \in L_T, |x| = n, \text{ такое, что} \\ \text{время принятия } x \text{ машиной } T \text{ равно } m\end{matrix}\right\}\right).$$

Заметим, что временная сложность машины T зависит только от числа шагов, выполняемых в принимающих вычислениях, а если нет ни одного входа длины n, принимаемого машиной T, то мы полагаем $T(n)$ равным 1.

НДМТ называется НДМТ с *полиномиальным временем работы*, если найдется полином p, такой, что $T_T(n) < p(n)$ при всех $n > 1$.

Определим теперь класс NP так:

$$\text{NP} = \left\{L \,\middle|\, \begin{matrix}\text{существует НДТМ с полиномиальным} \\ \text{временем работы, такая, что } L_T = L\end{matrix}\right\}.$$

Будем говорить, что задача распознавания принадлежит классу NP при схеме кодирования e, если $L[\Pi, e] \in \text{NP}$.

Как и в случае класса P, мы будем просто говорить, что Π лежит в NP, не упоминая конкретную схему кодирования, когда ясно, что некоторая разумная схема кодирования задачи Π даст язык, принадлежащий NP.

Так как ДМТ можно считать частным случаем НДМТ с количеством разветвлений $l = 1$, все задачи класса P являются задачами класса NP, то есть $\text{P} \subseteq \text{NP}$.

Из проводившихся ранее рассуждений можно понять, что есть много причин считать это включение строгим, то есть считать, что P не совпадает с NP. Полиномиальные недетерминированные алгоритмы определенно оказываются более мощными, чем полиномиальные детерминированные алгоритмы, и неизвестны общие методы их превращения в детерминированные полиномиальные алгоритмы. Самый

сильный результат состоит в следующем (с доказательством можно ознакомиться в [2]).

Теорема 1. Если задача распознавания $\Pi \in NP$, то существует такой полином p, что Π может быть решена детерминированным алгоритмом с временной сложностью $O(2^{p(n)})$.

Таким образом, моделирование решения таких задач для ДМТ не удается найти быстрее, чем за экспоненциальное время. Это наводит на мысль, что полиномиальные недетерминированные алгоритмы являются более мощным средством, чем полиномиальные детерминированные алгоритмы. В самом деле, для многих частных задач класса NP, например КОММИВОЯЖЕР, не найдено полиномиального детерминированного алгоритма.

Поэтому широко распространено мнение, что $P \neq NP$, хотя доказательство этой гипотезы пока отсутствует.

9.5. Полиномиальная сводимость и NP-полные задачи

Если P не совпадает с NP, то различие между P и $NP \backslash P$ очень существенно. Все задачи из P могут быть решены полиномиальными алгоритмами, а все задачи из $NP \backslash P$ трудно-решаемы. Поэтому если $P \neq NP$, то для каждой конкретной задачи $\Pi \in NP$ важно знать, какая из этих двух возможностей реализуется.

Конечно, пока не доказано, что $P \neq NP$, нет никакой надежды показать, что некоторая конкретная задача принадлежит классу $NP \backslash P$. Поэтому цель теории NP-полных задач заключается в доказательстве более слабых результатов вида: «если $P \neq NP$, то $\Pi \in NP \backslash P$». Основная идея для этого подхода основана на понятии полиномиальной сводимости.

Будем говорить, что имеет место *полиномиальная сводимость* языка $L_1 \subseteq \Sigma_1^*$ к языку $L_2 \subseteq \Sigma_2^*$, если существует функция $f\colon \Sigma_1^* \to \Sigma_2^*$, удовлетворяющая условиям:

1) существует ДМТ, вычисляющая f с временной сложностью, ограниченной полиномом;
2) для любого $x \in \Sigma_1^*$ $x \in L_1 \Leftrightarrow f(x) \in L_2$.

Если L_1 полиноминально сводится к L_2, то будем писать $L_1 \Rightarrow L_2$ и говорить «L_1 сводится к L_2».

Лемма 1. Если $L_1 \Rightarrow L_2$, то из $L_2 \in$ P следует, что $L_1 \in$ P (эквивалентное утверждение: из $L_1 \notin$ P следует, что $L_2 \notin$ P).

Доказательство. Пусть Σ_1 и Σ_2 — алфавиты языков L_1 и L_2 соответственно, функция $f\colon \Sigma_1^* \to \Sigma_2^*$ осуществляет полиномиальную сводимость L_1 к L_2, T_f — полиномиальная ДМТ, вычисляющая f, и T_2 — полиномиальная ДМТ, распознающая L_2. Полиномиальная ДМТ, распознающая L_1, может быть получена следующей композицией машин T_2 и T_f: $T_2(T_f(x))$, где $x \in \Sigma_1^*$. Из теоремы 1 второй главы следует, что $T_2(T_f)$ — ДМТ. Так как $x \in L_1$ тогда и только тогда, когда $f(x) \in L_2$, то полученная ДМТ распознает L_1.

Покажем теперь, что эта ДМТ полиномиальная. Пусть p_f и p_2 — полиномы, ограничивающие время работы машины T_f и T_2 соответственно. Тогда $f(|x|) \le p_f(|x|)$, а время работы только что построенной ДМТ ограничено функцией $O(p_f(|x|) + p_2(p_f(|x|)))$, которая есть полином от $|x|$. □

Если Π_1 и Π_2 — задачи распознавания, а e_1 и e_2 — их схемы кодирования, будем писать $\Pi_1 \Rightarrow \Pi_2$ (относительно заданных схем кодирования), если имеет место полиномиальная сводимость языка $L[\Pi_1, e_1]$ к $L[\Pi_2, e_2]$.

Когда будет действовать стандартное предположение о «разумности» используемых схем кодирования, упоминание о конкретных схемах кодирования, как обычно, будет опускаться.

На уровне задач полиномиальная сводимость задачи распознавания Π_1 к задаче распознавания Π_2 означает наличие функции $f\colon D_{\Pi_1} \to D_{\Pi_2}$, удовлетворяющей условиям:

1) f вычисляется полиномиальным алгоритмом;

2) для всех $I \in D_\Pi$ $I \in Y_{\Pi_1} \Leftrightarrow f(I) \in Y_{\Pi_2}$.

Лемма 1 позволяет интерпретировать сводимость $\Pi_1 \Rightarrow \Pi_2$ как утверждение, что задача Π_2 «не проще» задачи Π_1. Введенное отношение полиномиальной сводимости особенно удобно, так как оно является транзитивным.

Лемма 2. Если $L_1 \Rightarrow L_2$ и $L_2 \Rightarrow L_3$, то $L_1 \Rightarrow L_3$.

Доказательство. Пусть Σ_1, Σ_2, Σ_3 — алфавиты языков L_1, L_2, L_3 соответственно, функция $f_1\colon \Sigma_1^* \to \Sigma_2^*$ реализует полиномиальную сводимость L_1 к L_2, а $f_2\colon \Sigma_2^* \to \Sigma_3^*$ — полино-

миальную сводимость L_2 к L_3. Тогда функция f: $\Sigma_1^* \to \Sigma_3^*$, где $f(x) = f_2(f_1(x))$ для всех $x \in \Sigma_1^*$, реализует искомую сводимость языка L_1 к языку L_3. Ее полиномиальность доказывается аналогично соответствующему доказательству леммы 1. □

Языки L_1 и L_2 (соответственно задачи распознавания Π_1 и Π_2) *полиноминально эквивалентны*, если они сводятся друг к другу, т. е. $L_1 \Rightarrow L_2$, и $L_2 \Rightarrow L_1$ (имеет место $\Pi_1 \Rightarrow \Pi_2$ и $\Pi_2 \Rightarrow \Pi_1$).

Из леммы 2 следует, что это отношение есть отношение эквивалентности, а отношение «$\Rightarrow$» определяет частичное упорядочение возникающих классов эквивалентности языков (задач распознавания).

Класс P — это «наименьший» относительно этого частичного порядка класс эквивалентности, и с вычислительной точки зрения его можно рассматривать как класс «самых легких» языков (задач распознавания).

Язык L называется NP-*полным*, если $L \in$ NP и любой другой язык $L' \in$ NP сводится к L.

Говоря неформально, задача распознавая Π называется NP-полной, если $\Pi \in$ NP и любая другая задача распознавания $\Pi' \in$ NP сводится к Π.

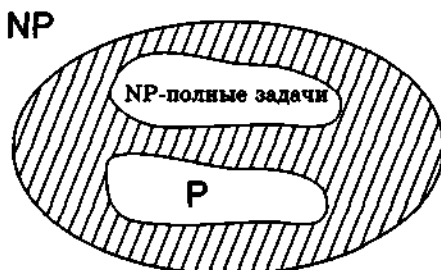


Рис. 9.2. Гипотетическая картина класса NP

Таким образом, лемма 1 позволяет отождествить NP-полные задачи с «самыми трудными» задачами из NP.

Если хотя бы одна NP-полная задача может быть решена за полиномиальное время, то и все задачи из NP также могут быть решены за полиномиальное время. Если хотя бы одна задача из NP труднорешаема, то и все NP-полные задачи труднорешаемы. Следовательно, любая NP-полная задача

П обладает свойством, которое сформулировано в начале этого пункта: если P ≠ NP, то П = NP\P. Точнее, П ∈ P тогда и только тогда, когда P = NP.

Можно показать [2, 4, 9], что если P ≠ NP, то существуют задачи из NP, неразрешимые за полиномиальное время и не являющиеся NP-полными. В предположении P ≠ NP на рис. 9.2 представлена картина класса NP.

Следующая лемма, которая немедленно следует из определения и транзитивности отношения ⇒, показывает, что вопрос о том, что любая задача из NP сводится к некоторому кандидату на NP-полную задачу, сильно упростился бы, если бы была известна, по крайней мере, одна NP-полная задача (доказательство оставляем читателю).

Лемма 3. Если L_1 и L_2 принадлежат классу NP, L_1 — это NP-полный язык и $L_1 \Rightarrow L_2$, то L_2 также NP-полный язык.

На уровне задач распознавания эта лемма указывает простой путь доказательства NP-полноты новой задачи П, если известна хотя бы одна NP-полная задача:

1) П ∈ NP;

2) какая-то известная NP-полная задача П′ сводится к П.

Однако, прежде чем можно будет воспользоваться этим методом доказательства, необходимо найти некоторую исходную NP-полную задачу.

9.6. Примеры NP-полных задач

Честь быть «первой» NP-полной задачей выпала на долю задачи распознавания из булевской логики.

ВЫПОЛНИМОСТЬ (ВЫП)

Условие. Задано множество булевских переменных $U = \{u_1, u_2, \ldots, u_m\}$ и конъюнкция C дизъюнкций над U.

Вопрос. Существует ли выполняющий набор значений истинности для C?

Пример 6. $U = \{u_1, u_2\}$, $C = (u_1 \vee \overline{u}_2)\ \&\ (\overline{u}_1 \vee u_2)$. Это индивидуальная задача из ВЫП, ответ на которую есть «да» (выполняющее задание значений истинности такое: $u_1 = u_2 =$ ИСТИНА). △

Пример 7. $U = \{u_1, u_2\}$, $C = (u_1 \vee \bar{u}_2) \& (u_1 \vee u_2) \& \bar{u}_1$. Это индивидуальная задача из ВЫП, ответ на которую есть «нет», поскольку C невыполнима. △

Сформируем теперь без доказательства фундаментальную теорему Кука [9].

Теорема 2 (*Кук*). Задача ВЫПОЛНИМОСТЬ есть NP-полная задача.

Опираясь на этот результат, мы можем теперь, используя указанный в предыдущем пункте путь доказательства (сводимость!), получить новые NP-полные задачи. Приведем 6 задач, которые чаще других используют в качестве «известных NP-полных задач» (все необходимые доказательства и много других NP-полных задач из различных разделов математики приведены в [2]).

3-ВЫПОЛНИМОСТЬ (3-ВЫП)

Условие. Дана конъюнкция $C = \{c_1, c_2, \ldots, c_m\}$ дизъюнкций на конечном множестве булевых переменных U, таких, что $|c_i| = 3$, $1 \le i \le m$.

Вопрос. Существует ли на U набор значений истинности, при котором выполняются все дизъюнкции из C?

ТРЕХМЕРНОЕ СОЧЕТАНИЕ (3-С)

Условие. Дано множество $M \subseteq W \times X \times Y$, где W, X, Y — непересекающиеся множества, содержащие одинаковое число элементов q.

Вопрос. Верно ли, что M содержит трехмерное сочетание, то есть подмножество $M' \subseteq M$, такое, что $|M'| = q$ и никакие два разных элемента M' не имеют ни одной равной координаты?

ВЕРШИННОЕ ПОКРЫТИЕ (ВП)

Условие. Дан граф $G = (V, E)$ и положительное целое число $K \le |V|$.

Вопрос. Имеется ли в графе G вершинное покрытие не более чем из K элементов, то есть такое подмножество $V' \subseteq V$, что $|V'| \le K$ 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 $J \le |V|$.

Вопрос. Верно ли, что G содержит некоторую клику мощности не менее J, то есть такое подмножество $V' \subseteq V$, что $|V'| \ge J$ 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, то есть такую последовательность $\langle v_1, v_2, \ldots, v_n \rangle$ вершин графа G, что $n = |V|$, $\{v_n, v_1\} \in E$ и $\{v_i, v_{i+1}\} \in E$ для всех i, $1 \le i < n$?

РАЗБИЕНИЕ

Условие. Заданы конечное множество A и «вес» $s(a) \in \mathbb{N}^+$ для каждого $a \in A$.

Вопрос. Существует ли подмножество $A' \subsetneq A$, такое, что

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) \ ?$$

Рис. 9.3. Диаграмма последовательности сведения задач

Одна из причин популярности этих шести задач состоит в том, что они содержались в исходном списке из 21 NP-полной задачи, приведенном в работе Карпа [4]. Естественно, что первая сводимость получена с помощью задачи ВЫПОЛНИМОСТЬ, так как некоторое время она была единственной известной NP-полной задачей. На рис. 9.3 приведена одна из возможных последовательностей сведения задач [2], которая может быть применена при доказательстве их NP-полноты.

Тестовые задания

1. Индивидуальная задача принадлежит D_{Π} тогда и только тогда, когда:
   1) ответом на вопрос задачи будет «да»;
   2) она может быть получена из стандартной формы описания подстановкой конкретных значений во все компоненты условия.

2. Пусть Σ — алфавит, Σ^* —множество всех конечных цепочек, составленных из символов алфавита Σ. Любое подмножество $L \subseteq \Sigma^*$ называется:
   1) словом;
   2) языком;
   3) предложением.

3. Выбрать неверное утверждение.
   Схема кодирования считается «разумной», если она:
   1) допускает декодирование;
   2) полиномиально эквивалентна длине кода индивидуальной задачи;
   3) достаточно «сжатая».

4. Если существует такой полином p, что для всех $n \in \mathbb{N}^+$, $T_T(n) \leq p(n)$, то детерминированная машина Тьюринга называется:
   1) полиномиально эквивалентной;
   2) полиномиальной;
   3) эквивалентной.

5. Машина Тьюринга, имеющаяся конечный набор внутренних состояний $\{q_1, q_2, \ldots, q_n\}$, алфавит символов на ленте $\{a_1, a_2, \ldots, a_m\}$ и правила действий в виде списков, называется:
   1) детерминированной;
   2) недетерминированной;
   3) полиномиальной.

6. Процесс работы ДМТ может быть изображен в виде дерева, вершинами которого являются выполняемые действия, а ребрами — переходы от одного действия к другому?
   1) да;
   2) нет.

7. Если по некоторой цепи действия НДМТ заканчиваются и все остальные машины, размножившиеся к этому мо-

менту, тоже кончают работу и превращаются в одну машину, то

1) управляющее устройство перешло в состояние q_y;
2) управляющее устройство перешло в состояние q_n.

8. НДТМ принимает вход x, если:
   1) все вычисления на входе x являются принимающими;
   2) только одно из вычислений на входе x является принимающим;
   3) нет верного ответа.

9. Время, требующееся НДМТ T для того, чтобы принять слово $x \in L_T$, — это:
   1) минимальное число шагов, выполненных T до достижения заключительного состояния q_y;
   2) минимальное число шагов, выполненных T до достижения заключительного состояния q_n;
   3) максимальное число шагов, выполненных T до достижения заключительного состояния q_y.

10. Если $L_1 \Rightarrow L_2$, и $L_2 \Rightarrow L_1$, то языки L_1 и L_2
    1) полиномиально сводимы;
    2) полиномиально эквивалентны;
    3) полиномиально зависимы.

11. Если $L \in$ NP и любой другой язык $L' \in$ NP сводится к L, то
    1) язык L полиномиально эквивалентен языку L';
    2) языки L и L' находятся в отношении эквивалентности друг к другу;
    3) язык L называется NP-полным.

Литература

1. *Верещагин Н. К.* Вычислимые функции / Н. К. Верещагин, А. Шень. — М. : МЦНМО, 1999. — 176 с.
2. *Гэри М.* Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. — М. : Мир, 1982. — 416 с.
3. *Игошин В. И.* Математическая логика и теория алгоритмов / В. И. Игошин. — Саратов, 1991.
4. *Карп Р. М.* Сводимость комбинаторных проблем / Р. М. Карп // Кибернетический сборник. — М. : Мир, 1975. — Вып. 12. — С. 16–38.
5. *Катленд Н.* Вычислимость. Введение в теорию рекурсивных функций / Н. Катленд. — М. : Мир, 1983.
6. *Клини С. К.* Введение в метаматематику / С. К. Клини. — М. : Издательство иностр. лит., 1957.
7. *Колмогоров А. Н.* К определению алгоритма / А. Н. Колмогоров, В. А. Успенский // Успехи математических наук. — 1958. — №4(13). — С. 3–28.
8. *Кузнецов О. П.* Дискретная математика для инженера / О. П. Кузнецов, Г. Н. Адельсон-Вельский. — М. : Энергоатомиздат, 1988.
9. *Кук С. А.* Сложность процедур вывода теорем / С. А. Кук // Кибернетический сборник. — М. : Мир, 1975. — Вып. 12. — С. 5–15.
10. *Мальцев А. И.* Алгоритмы и рекурсивные функции / А. И. Мальцев. — М. : Наука, 1965.
11. *Марков А. А.* Теория алгорифмов / А. А. Марков [Тр. матем. ин-та АН СССР]. — 1954. — Т. 42.
12. *Марков А. А.* Теория алгорифмов / А. А. Марков, Н. И. Нагорный. — М. : Наука, 1984.
13. *Мендельсон Э.* Введение в математическую логику / Э. Мендельсон. — М. : 1976.
14. *Минский М.* Вычисления и автоматы / М. Минский. — М. : Мир, 1971.

15. *Роджерс Х.* Теория рекурсивных функций и эффективная вычислимость / Х. Роджерс. — М. : Мир, 1972.
16. *Трахтенброт Б. А.* Алгоритмы и вычислительные автоматы / Б. А. Трахтенброт. — М. : Советское радио, 1974.
17. *Фролов Г. Д.* Элементы информатики / Г. Д. Фролов, Э. И. Кузнецов. — М. : Высшая школа, 1989.
18. *Хьюз Дж.* Структурный подход к программированию / Дж. Хьюз, Дж. Мичтом. — М. : Мир, 1980.
19. *Friedberg R. M.* Three Theorems on Recursive Enumeration / R. M. Friedberg // Jour Symbolic Logic. — 1958. — № 23. — p. 309–316.
20. *Bohm C.* Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules. / C. Bohm, G. Jacopini. // Communications of the ACM. — May 1966. — Vol. 9. — p. 366–371.

Предметный указатель

Алгоритм 9
- перечисляющий 141
- *проектирующий* 81
- разрешающий 141
- универсальный 135
Алгоритма
- блок-схема 18
- детерминированность 12
- дискретность 12
- естественное распространение 81
- интуитивное понятие 13
- массовость 13
- результативность 13
- формальное распространение 81
- элементарность шагов 13
Алгоритмов
- композиция 18, 81
- нормальная композиция 81
- нумерация 125
- разветвление 84
- соединение 82
- эквивалентность 79
Алфавит 14, 72, 176
Алфавита расширение 72

Блок функциональный 25

Вычисление
- непринимающее 186
- принимающее 186
- сходится 97

Гёделев номер программы 127

Грамматика 177
- контекстно-свободная 179
- *регулярная* 180
- с фазовой структурой 178

Декартово произведение 21
Диагональ Кантора 130

Задача
- NP-полная 190
- о коммивояжере 181
- распознавания свойств 181
Запись на ленте правильная 34

Класс
- NP 187
- P 185
- сложности 168
- элементарных функций 170
Кодирование области 100
Кодовый номер программы 127
Команды МНР 92
- - арифметические 92

Маркова
- нормальный алгоритм 74
- принцип нормализации 86
Марковская подстановка 73
Машина с неограниченными регистрами (МНР) 91
- фон Неймана 48
Меры вычислительной сложности 164, 166
Множества
- нумерация 125
- перечисление 125

Множество
- перечислимое 141
- разрешимое 141
- счетное 125
- эффективно счетное 126

Оператор 56, 150
- минимизации неограниченный 64, 109
- - ограниченный 61
- монотонный 152
- наименьшего числа ограниченный 61
- непрерывный 152
- повторения 112
- примитивно-рекурсивный 60
- примитивной рекурсии 57
- развилки 111
- рекурсивный 152
- суперпозиции 56
Отношение
- примитивно-рекурсивное 59
- частичного порядка 24
- эквивалентности 23

Почти всюду 164
Правило вычисления 157
Предикат 23
- примитивно-рекурсивный 60
- разрешимый 99
- элементарный 170
Проблема
- остановки 46, 138
- самоприменимости 137
Проблемы неразрешимые 135
Программ
- нумерация 125, 127
- подстановка 105
- соединение 102, 104
Программа рекурсивная 156
Программы стандартный вид 103

Рекурсия примитивная 107
- - ограниченная 171

Сводимость полиномиальная 188
Сентенциальная форма 178
Сложность временная 163
- - ДМТ 184
- - НДМТ 187
Структура
- повторения 26
- развилки 25
- следования 25
Схема кодирования 182
- нормального алгоритма 74

Тезис
- Тьюринга 45
- Чёрча 66, 113
Теорема
- *s-m-n* простая 132
- - полная 133
- Блюма об ускорении 167
- Бома–Джакопини 25
- Бородина о пробелах 169
- Клини о рекурсии первая 154
- - - - вторая 159
- Майхилла-Шепердсона 153, 154
- о графике 145
- параметризации 133
- Райса 146
Точка неподвижная 154, 159
Тьюринга
- машин
- - композиция 37
- - развилка 40
- - цикл 41
- машина 31
- - вычисляет предикат 38
- - вычисляет с восстановлением 39
- - вычисляет функцию 34
- - детерминированная 31, 184
- - - полиномиальная 185
- - нетерминированная 185
- - - полиномиальная 187
- - универсальная 44
- машины эквивалентные 34

Функции
- индекс 129

– конечная часть 151
Функций нумерация 128
Функция
– Аккермана 64
– биективная 22
– вычислимая алгоритмически 15
– вычислимая по Маркову 80
– вычислимая по Тьюрингу 34
– инъективная 22
– конечная 151
– нормально вычислимая по Маркову 76
– МНР-вычислимая 97
– общерекурсивная 66
– перечисляющая 141
– примитивно-рекурсивная 57
– проекции 56, 102
– следования 56, 102
– сюръективная 22
– тотальная 23
– универсальная 135
– характеристическая предиката 59, 99
– частичная 23, 65
– частично вычислимая по Маркову 79
– частично-рекурсивная 65
– экстенсиональная 153
Хомского иерархия 179

Эквивалентность полиномиальная 183
Элемент Неймана 48

Язык 176
– NP-полный 190
Языки полиномиально эквивалентные 190

Обозначения

	Значение	Раздел
(x, y)	упорядоченная пара элементов x и y	1.2.1
$A \times B$	декартово произведение множеств A и B	1.2.1
$Dom(f)$	область определения функции f	1.2.1
$Ran(f)$	множество значений функции f	1.2.1
$f_\varnothing$	нигде не определенная функция	1.2.1
$\mathbb{N}$	множество натуральных чисел	1.2.1
$\mathbb{N}^+$	множество положительных натуральных чисел	1.2.1
$\mathbb{Z}$	множество целых чисел	1.2.1
$\mathbf{x}$	упорядоченный n-набор $(x_1, x_2, ..., x_n)$	1.2.1
x'	функция следования	3.1, 5.1
U_m^n	функция проекции	3.1, 5.1
S_m^n	оператор суперпозиции	3.1
$R_n(g, h)$	оператор примитивной рекурсии	3.1
μ	оператор минимизации	3.2, 5.3.4
R_n	регистр МНР с номером n	5.1
r_n	число, содержащееся в регистре МНР с номером n	5.1
$Z(n)$	МНР-команда обнуления	5.1
$S(n)$	МНР-команда прибавления единицы	5.1
$T(m, n)$	МНР-команда переадресации	5.1
$J(m, n, q)$	МНР-команда условного перехода	5.1
$P(a_1, ..., a_n)$	вычисление МНР по программе P с начальной конфигурацией $a_1, ..., a_n, 0, 0, ...$	5.1

$P(a_1, ..., a_n)\downarrow$	МНР-вычисление $P(a_1, ..., a_n)$ в конце концов останавливается	5.1
$P(a_1, ..., a_n)\uparrow$	МНР-вычисление $P(a_1, ..., a_n)$ никогда не останавливается	5.1
$P(a_1, ..., a_n)\downarrow b$	МНР-вычисление $P(a_1, ..., a_n)$ сходится и в заключительной конфигурации $r_1 = b$	5.2
b	класс одноместных МНР-вычислимых функций	5.2
b_n	класс n-местных МНР-вычислимых функций	5.2
$f_P^{(n)}$	n-местная функция, вычислимая МНР с программой P	5.2
$P[l_1, ..., l_n \to l]$	программа МНР вычисляет $f(r_{l_1}, r_{l_2}, \ldots, r_{l_n})$ и помещает результат в R_l	5.3.1
$\rho(P)$	количество регистров МНР, которое затрагивается при работе программы P	5.3.1
$\mathcal{L}$	множество всех команд МНР	6.2.1
$\mathcal{P}$	множество всех программ МНР	6.2.1
$\Phi_a^{(n)}$	n-местная функция, вычислимая по программе P_a	6.2.2
$W_a^{(n)}$	область определения функции $\Phi_a^{(n)}$	6.2.2
$E_a^{(n)}$	множество значений функции $\Phi_a^{(n)}$	6.2.2
$\langle x \rangle$	код набора $x = (x_1, \ldots, x_n)$	7.1
I_b	класс сложности функции b	8.2
E	класс элементарных функций	8.3
P	класс задач, для решения которых существует полиномиальная детерминированная машина Тьюринга	9.3
NP	класс задач, для решения которых существует полиномиальная недетерминированная машина Тьюринга	9.4

Учебное издание

Серия: Педагогическое образование

Матрос Дмитрий Шаевич
Поднебесова Галина Борисовна

Теория алгоритмов
Учебник

Ведущий редактор *Е. Костомарова*
Художник *С. Инфантэ*
Художественный редактор *О. Лапко*
Корректор *Е. Клитина*
Компьютерная верстка *В. Носенко*

Подписано в печать 29.05.08. Формат 60×90/16
Бумага офсетная. Усл. печ. л. 13,0
Тираж 2000 экз. Заказ 3257.

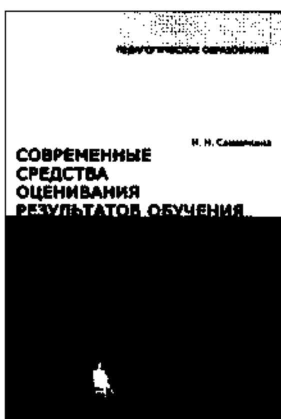
Издательство «БИНОМ. Лаборатория знаний»
Адрес для переписки: 125167, Москва, проезд Аэропорта, 3
Телефон: (499)157-52-72. E-mail: Lbz@aha.ru
http://www.Lbz.ru

Отпечатано в производственной фирме «Полиграфист»
160001, г. Вологда, ул. Челюскинцев, 3.

ЛИТЕРАТУРА ДЛЯ ВУЗОВ

■ ПЕДАГОГИКА

ИМЕЕТСЯ В ПРОДАЖЕ



Самылкина Н. Н. ***Современные средства оценивания результатов обучения*** */ Н. Н. Самылкина. — 2007. — 172 с. : ил. — (Педагогическое образование).*

Книга представляет собой курс лекций для новой общепрофессиональной дисциплины «Современные средства оценивания результатов обучения» для подготовки будущих учителей информатики в области теории и практики педагогических измерений с использованием компьютерных технологий. Она также может быть полезна при изучении раздела «Использование современных информационных и коммуникационных технологий в учебном процессе» и курса «Психолого-педагогическая диагностика на основе компьютерного тестирования» для учителей информатики.

Для студентов педагогических вузов, учителей, методистов и администрации образовательных учреждений, а также для аспирантов и соискателей при подготовке ими диссертаций (при обработке результатов педагогического эксперимента).

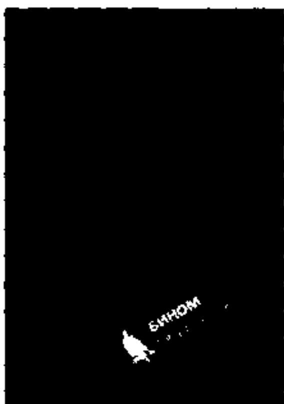
ИЗДАТЕЛЬСТВО
«БИНОМ
Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: Lbz@aha.ru, http://www.Lbz.ru
Оптовые поставки:
(495) 174-7616, 171-1954, 170-6674

ЛИТЕРАТУРА ДЛЯ ВУЗОВ

■ ПЕДАГОГИКА

ИМЕЕТСЯ В ПРОДАЖЕ



Филатова Л. О. ***Развитие преемственности школьного и вузовского образования в условиях введения профильного обучения в старшем звене средней школы*** */ Л. О. Филатова. — 2006. — 192 с. : ил.*

В монографии рассматриваются сущность и компоненты преемственности общего среднего и высшего профессионального образования. Анализируются пути и возможности развития преемственности этих двух компонентов системы непрерывного образования в условиях начавшейся модернизации российского образования. Раскрыты роль и перспективы изменения представлений о целях и ценностях образования, структуры и содержания образования на старшей ступени школы, совершенствования организационных форм и методов обучения, создания новой образовательной среды на базе средств информационных и коммуникационных технологий в повышении эффективности преемственности школьного и вузовского образования.

Для преподавателей и методистов.

ИЗДАТЕЛЬСТВО

«БИНОМ
Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: Lbz@aha.ru, http://www.Lbz.ru
Оптовые поставки:
(495) 174-7616, 171-1954, 170-6674

ИНФОРМАТИЗАЦИЯ ОБРАЗОВАНИЯ

ИМЕЕТСЯ В ПРОДАЖЕ

Цветкова М. С. ***Модели комплексной информатизации общего образования*** */ М. С. Цветкова, Э. С. Ратобыльская, Г. Д. Дылян. — 2007. — 119 с. : ил.*

Книга является результатом научного исследования проблем управления процессами информатизации системы общего среднего образования и может служить пособием для практических работников всех уровней управления системой образования. Основная идея пособия — комплексный подход при моделировании и осуществлении процессов информатизации школьного образования.

Для управленческих работников органов управления образованием, учреждений образования, специалистов, использующих в своей работе информационные компьютерные технологии.

ИЗДАТЕЛЬСТВО
«БИНОМ
Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: Lbz@aha.ru, http://www.Lbz.ru
Оптовые поставки:
(495) 174-7616, 171-1954, 170-6674

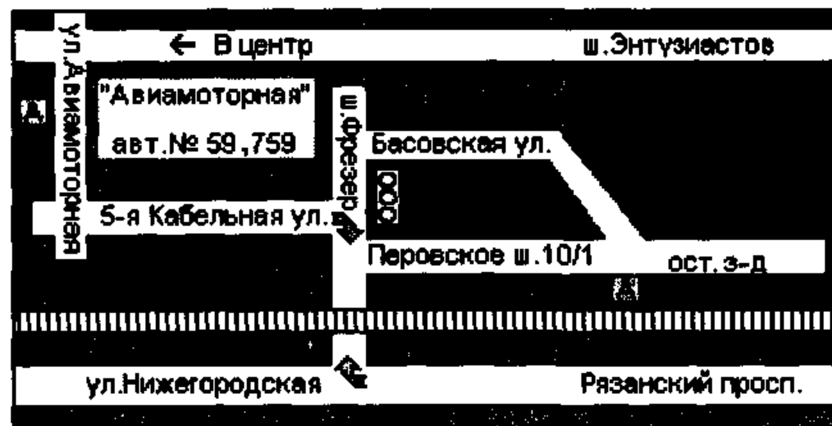
По вопросам оптовых поставок литературы обращаться:

ЗАО «Торговый Дом «БИНОМ»

109202, Москва, Перовское ш., 10/1, магазин «Книги».

Телефоны:
(495) 171-19-54
(495) 170-66-74
(495) 174-76-16

e-mail:
tdbinom@Lbz.ru



Торговый Дом «БИНОМ» предлагает вашему вниманию широкий спектр **учебной, специальной** и **справочной** литературы ведущих издательств, в частности:

- БИНОМ. Лаборатория знаний
- Интуит.ru
- МИР
- МЦНМО
- Руссо
- Техносфера
- Физматлит
- Эком

Представлены следующие тематики:

- математика
- информатика
- компьютерная литература
- физика
- химия
- биология
- медицина
- инженерные специальности
- экономика и бизнес

Подписывайтесь в каталоге агентства «Роспечать»
Посещайте *сайт www.infojournal.ru*

Ежемесячный научно-методический журнал
«ИНФОРМАТИКА И ОБРАЗОВАНИЕ»

Учредители — *Российская академия образования,*
издательство «Образование и Информатика».

Издается с 1986 года как предметный журнал для учителей информатики.

ОСНОВНЫЕ РУБРИКИ ЖУРНАЛА:

ИКТ В ОБРАЗОВАНИИ — *аналитические статьи и документы по информатизации образования, нормативные материалы о преподавании информатики и информатизации образования. Статьи ведущих ученых РАО, педагогических вузов и научных институтов.*

ИНФОРМАТИЗАЦИЯ ШКОЛЫ — методические материалы и нормативные документы для заместителя директора школы по информатизации.

ЗАРУБЕЖНЫЙ ОПЫТ — статьи зарубежных авторов об использовании ИКТ в образовании.

МЕТОДИКА — методические материалы по всем темам пропедевтического, базового и профильного курсов информатики (I — XI классы), дидактические материалы, авторские методики, а также общие психолого-педагогические аспекты преподавания информатики в школе, ПТУ, ССУЗе и вузе.

ИКТ В ПРЕДМЕТАХ — рубрика для учителей-предметников, использующих ИКТ в своей деятельности.

ЗАДАЧИ — разнообразные задачи различных уровней сложности с разбором решений и методическими рекомендациями.

ПЕДАГОГИЧЕСКИЙ ОПЫТ — методические разработки лучших учителей и преподавателей информатики, творческие находки и опыт учителей, использующих ИКТ на уроках.

ТОЧКА ЗРЕНИЯ — дискуссионные материалы по проблемам информатизации образования и преподаванию курса информатики.

ПОДПИСНЫЕ ИНДЕКСЫ ЖУРНАЛА:

в каталоге агентства «Роспечать»: 70423 — для индивидуальных подписчиков;
73176 — для предприятий и организаций;
в объединенном каталоге «Пресса России» — 26097.

ПРИЛОЖЕНИЕ К ЖУРНАЛУ «ИНФОРМАТИКА И ОБРАЗОВАНИЕ» «ИНФОРМАТИКА В ШКОЛЕ»

Периодичность издания — 8 раз в год.

Подписные индексы:
в каталоге агентства «Роспечать»: 81407 — для индивидуальных подписчиков;
81408 — для предприятий и организаций;
в объединенном каталоге «Пресса России» — 45751.

Адрес для писем: 127051, Москва, а/я 163,
ООО «Образование и Информатика»
Телефон: (495) 139-56-89, *факс: (495) 497-67-96*,
e-mail: readinfo@mtu-net.ru, info@infojournal.ru
Сайт в *Интернете: www.infojournal.ru*

Перед вами книга из серии
ПЕДАГОГИЧЕСКОЕ ОБРАЗОВАНИЕ

МАТРОС ДМИТРИЙ ШАЕВИЧ
Доктор педагогических наук, профессор, заведующий кафедрой информатики и методики преподавания информатики, декан факультета информатики Челябинского государственного педагогического университета.
Сфера научных интересов — теоретическая информатика, информатизация образования.

ПОДНЕБЕСОВА ГАЛИНА БОРИСОВНА
Кандидат педагогических наук, доцент, заместитель декана по учебной работе факультета информатики Челябинского государственного педагогического университета.
Сфера научных интересов — теоретическая информатика, модернизация обучения информатике в вузе.

Учебник по курсу «Теория алгоритмов» для педагогических вузов по специальности «Информатика» полностью соответствует стандарту.

Изложение имеет четкую логическую структуру и охватывает следующие темы: понятие алгоритма, машина Тьюринга, примитивно-рекурсивные функции, нормальные алгоритмы, вычислимость и разрешимость, сложность вычислений, NP-полные задачи. Каждая тема сопровождается тестовыми заданиями и упражнениями.

Для студентов и преподавателей педагогических вузов, а также учителей общеобразовательных школ.

ISBN 978-5-94774-226-8
9 785947 742268