

У Ч Е Б Н И К / Д Л Я В У З О В

Ю. Г. Карпов

ТЕОРИЯ АВТОМАТОВ

Допущено Министерством образования Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавров «Информатика и вычислительная техника» и по специальности «Вычислительные машины, комплексы, системы и сети» направления подготовки дипломированных специалистов «Информатика и вычислительная техника»



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2003

Рецензенты:

Крук Е. А., доктор технических наук, зав. кафедрой «Безопасность информационных систем»
Санкт-Петербургского Государственного университета аэрокосмического приборостроения

Соколов Б. В., доктор технических наук, профессор кафедры «Компьютерная математика и программирование»
Санкт-Петербургского Государственного университета аэрокосмического приборостроения

К26 **Теория автоматов** / Ю. Г. Карпов — СПб.: Питер, 2003. — 208 с.: ил.

ISBN 5-318-00537-3

Эта книга служит формированию знаний и умений, которые образуют теоретический фундамент, необходимый для корректной постановки и решения проблем в области информатики, для осознания целей и ограничений при создании вычислительных структур, алгоритмов и программ обработки информации.

В этом учебнике практическое использование моделей не является частной иллюстрацией теоретических результатов — наоборот, автор постарался практические проблемы проектирования и анализа систем сделать отправной точкой, а формальный аппарат — средством систематического решения этих проблем. В каждом разделе книги большое внимание уделено вопросам абстрагирования и адекватной интерпретации и реализации результатов аналитических преобразований.

Усвоение рассмотренных в книге моделей теоретической информатики, способов их анализа и синтеза должно создать основу, позволяющую читателю воспринимать и усваивать многие другие общетехнические и специальные дисциплины по информационным технологиям, вычислительным средствам и системам, инструментарию и методам проектирования программных систем, входящим в программу высшей школы.

Книга допущена Министерством образования Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавров «Информатика и вычислительная техника» и по специальности «Вычислительные машины, комплексы, системы и сети» направления подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 22.181.3я7

УДК 519.713(075)

Содержание

Введение	6
От издательства	7
ГЛАВА 1	
Конечные функциональные преобразователи	8
Постановка проблемы	8
Булевы функции	11
Свойства булевых функций	17
Нормальные формы представления булевых функций	14
Преобразование в нормальную форму	15
Реализация булевых функций	16
Минимизация булевых функций	17
Функциональная полнота	22
Алгебра Жегалкина и линейные функции	24
Замкнутые классы булевых функций	26
Теорема Поста	27
Формы представления булевых функций	32
Семантические деревья	32
Бинарные диаграммы решений — БДР	32
Булевые алгебры	34
Пороговая логика	36
Контрольные задания	37
ГЛАВА 2	
Введение в математическую логику	40
Формальные модели	40
Логика высказываний	44
Формулировка и доказательство теорем	48
Проверка доказательных рассуждений	50
Силлогизмы	52

Логическое следствие	54
Основная теорема логического вывода	57
Приведение к нормальным формам	57
Метод резолюции	58
Другие методы	61
Адекватность логики высказываний	61
Основы логики предикатов и логического вывода	62
Предикаты	62
Свободные и связанные переменные	65
Интерпретации	65
Равносильности логики предикатов	66
Ограниченные предикаты	68
Логический вывод в логике предикатов	71
Скулемовская стандартная форма	71
Алгоритм унификации	73
Логическое программирование	78
Логический вывод в ПРОЛОГЕ	79
Стратегии вывода	80
Применение логического вывода для анализа схем	84
Экспертные системы	85
Контрольные задания	88

ГЛАВА 3

Конечные автоматы	95
Автоматное преобразование информации	96
Реализация КА	99
Эквивалентность КА: теорема Мура	102
Минимизация КА	106
Автоматы Мили и Мура	110
Примеры КА	112
Триггеры	112
Электронные часы	112
Схема управления микрокалькулятором	114
Команды операционной системы UNIX	118
Реактивные системы	118
Протокол PAR передачи сообщений в сетях	119
Протокол выбора лидера в распределенной системе	121
Визуальный формализм представления моделей реактивных систем: Statecharts	124
Протокол IEEE 802.12	125
Автомат, регулирующий пешеходный переход	126
Графы переходов при спецификации и анализе параллельных программ	127
Проблема умножения: алгоритм, который не может выполнить КА	131

Алгебраическая структурная теория конечных автоматов	132
Кодирование внутренних состояний	
конечного автомата	133
Разбиения и частично упорядоченные множества	136
Универсальные алгебры и конгруэнции	139
Последовательная декомпозиция конечных автоматов	142
Параллельная декомпозиция конечных автоматов	143
Алгоритм поиска конгруэнций конечного автомата	143
Контрольные задания	146
ГЛАВА 4	
Автоматные языки	151
Языки	152
Грамматики. Автоматные грамматики и языки	153
Лемма о накачке	158
Эквивалентность и минимизация конечноавтоматных распознавателей	160
Недетерминированные конечно-автоматные распознаватели	163
Синтаксические диаграммы. Связь синтаксических диаграмм и автоматных языков	167
Трансляторы автоматных языков	168
Транслятор языка LINUR	170
Транслятор языка EXPR	172
Регулярные множества и регулярные выражения	174
Регулярные множества	174
Регулярные выражения	174
Теорема Клини	176
Контрольные задания	183
ГЛАВА 5	
Машины Тьюринга	187
Формальные модели алгоритмов	188
Понятие алгоритма	188
Формализация понятия алгоритма	188
Машина Тьюринга	190
Примеры машин Тьюринга	193
Свойства машины Тьюринга	194
Реализация машины Тьюринга	196
Универсальная машина Тьюринга	198
Алгоритмически неразрешимые проблемы	198
Частично разрешимые проблемы	202
Контрольные задания	202
Литература	204

Введение

Книга представляет собой учебник по курсу «Теория автоматов» — одному из важнейших компонентов федерального государственного образовательного стандарта бакалавров по направлению «Информатика и вычислительная техника». Этот курс является основной составной частью дисциплины «Теоретическая информатика», входящей в естественнонаучный цикл дисциплин российской высшей школы. Курс служит формированию знаний и умений, которые образуют теоретический фундамент, необходимый для корректной постановки и решения проблем в области информатики, для осознания целей и ограничений при создании вычислительных структур, алгоритмов и программ обработки информации.

Использование науки в инженерной практике при создании новых объектов состоит в построении и использовании математических моделей — абстракций, отражающих интересующие исследователя свойства реальности. Предмет данной книги составляет аппарат основных моделей в области информатики: булева алгебра, конечные автоматы, формальные языки, машины Тьюринга, классическая логика и т. д. Здесь изучаются их свойства, преимущества и ограничения, примеры применения. Следуя глубоко верной мысли о том, что нет ничего более практического, чем хорошая теория, в данной книге эти модели рассматриваются не с точки зрения их абстрактных свойств как таковых; они изучаются как средство решения практических, инженерных задач.

Исследования в области теории автоматов начались в середине 50-х годов прошлого века. Несмотря на свою простоту, модель конечного автомата оказалась чрезвычайно удобной в огромном числе приложений не только в информатике, но и во многих других областях инженерной деятельности. Большой интерес к этой теории объясняется именно широкими возможностями ее применения. Без преувеличения можно сказать, что теория автоматов является одним из фундаментальных блоков современной теоретической и практической информатики. Наряду с классическими приложениями теории автоматов, такими как проектирование встроенных систем логического управления, обработка текстов и построение компиляторов искусственных языков, в последнее время появились новые, нетрадиционные области применения этой теории — спецификация и верификация

систем взаимодействующих процессов (в частности, протоколов коммуникации), языки описания документов и объектно-ориентированных программных систем, оптимизация логических программ и другие. Для одного из наиболее обещающего из новых важных приложений теории автоматов, а именно для верификация систем процессов, необходимы знания из логики и теории логического вывода. Это является одной из причин включения соответствующего материала в книгу.

В данной книге практическое использование моделей является даже не частной иллюстрацией теоретических результатов — наоборот, автор постарался практические проблемы проектирования и анализа систем сделать отправной точкой, а формальный аппарат — средством систематического решения этих проблем. В каждом разделе книги большое внимание удалено вопросам абстрагирования (построения адекватной модели), а также адекватной интерпретации и реализации результатов аналитических преобразований.

Усвоение рассмотренных в книге моделей теоретической информатики, способов их анализа и синтеза должно обеспечить основу, позволяющую читателю воспринимать и усваивать многие другие общетехнические и специальные дисциплины по информационным технологиям, вычислительным средствам и системам, инструментарию и методам проектирования программных систем, входящим в программу высшей школы.

В результате освоения данного курса читатель на основе ясного понимания соответствующих теоретических разделов должен УМЕТЬ:

- реализовать простейший вид преобразования информации — конечное функциональное отображение — в произвольном базисе логических функций;
- выделять в доказательных рассуждениях естественного языка логическую структуру, факты, посылки и следствия; строить схемы формальных доказательств и проверять их правильность;
- интерпретировать программы на языке логического программирования ПРОЛОГ;
- разрабатывать дискретные системы управления на базе модели конечного автомата, доводя их до программной и аппаратной реализации;
- использовать алгебру регулярных выражений и синтаксические диаграммы для задания формальных языков, строить трансляторы автоматных языков.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательства <http://www.piter.com>.

ГЛАВА 1 Конечные функциональные преобразователи

Целью главы является ознакомление читателя с систематическим подходом к построению комбинационных схем — простейших преобразователей информации, реализующих функциональное отображение конечных множеств, и с математическими основами этого подхода: теорией двоичных (булевых) функций и ее обобщением — булевой алгеброй. В результате изучения материала этой главы читатель должен освоить:

- основы теории булевых функций, свойства и методы преобразования таких функций;
- теорию базисов булевых функций и методы представления функций в различных базисах;
- простейшие методы минимизации булевых функций;
- методы представления булевых функций.

Постановка проблемы

В главе рассматриваются проблемы построения простейших дискретных преобразователей информации. Пусть A — некоторое множество элементов информации, представленных тем или иным образом, B — другое множество элементов информации, а Φ — функция преобразования. Преобразователь информации можно представить себе схематично как устройство, реализующее отображение $\Phi: A \rightarrow B$ одного множества на другое (рис. 1.1, *a*). Рассмотрим возможность систематического построения таких преобразователей. Очевидно, что трудно предложить какое-то решение в общем случае, когда множества A и B имеют произвольную природу, а о самом отображении Φ ничего не известно. Однако, если множества A и B явля-

ются конечными (то есть преобразователь, который мы хотим построить, является «конечным функциональным преобразователем») и дискретными (то есть преобразование осуществляется в дискретные моменты времени), существует систематический метод решения этой проблемы. Этот метод состоит в том, что элементы множеств А и В предварительно кодируют двоичными кодами и строят преобразование одного множества двоичных векторов в другое (рис. 1.1, б). Именно этим мы и будем заниматься в данной главе.

Двоичное кодирование состоит во взаимном однозначном сопоставлении всем элементам конечного множества некоторых двоичных векторов одной и той же длины. При таком подходе проблема реализации преобразователя Φ сводится к построению трех преобразователей: кодировщика $K: A \rightarrow X$, собственно функционального преобразователя $F: X \rightarrow Y$ и декодировщика $D: Y \rightarrow B$, причем эти отображения должны быть выбраны так, что $K \circ F \circ D = \Phi$ (рис. 1.1, б). Рассмотрим проблемы построения этих трех преобразователей поочередно.

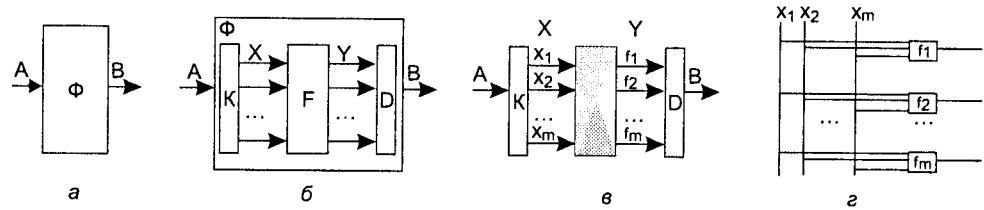


Рис. 1.1. Реализация конечного функционального преобразователя

Число двоичных векторов длины n равно 2^n , и если все элементы множества S задекодировать двоичными векторами одной и той же длины n , то для однозначности кодирования n должно быть не меньше, чем $\log_2|S|$. Пусть m и n — длины двоичных векторов для кодирования соответственно множеств А и В входных и выходных информационных элементов конечного функционального преобразователя Φ . Тогда кодирование — это построение некоторого отображения $\{0,1\}^n \rightarrow \{0,1\}^m$, декодирование — построение отображения $D: \{0,1\}^m \rightarrow B$, а функциональное отображение F сопоставляет каждому вектору из $\{0,1\}^m$ некоторый вектор из $\{0,1\}^n$, причем так, чтобы $K \circ F \circ D = \Phi$.

Пример 1.1

Пусть множество $A = \{a1, a2, a3, a4, a5\}$, а множество $B = \{b1, b2, b3\}$ и отображение Φ задается табл. 1.1.

Таблица 1.1. Отображение $\Phi: A \rightarrow B$

A	B
a1	b2
a2	b3
a3	b2
a4	b1
a5	b2

Понятно, что в данном примере n должно быть не меньше 3, m должно быть не меньше 2. Мы можем выбрать произвольно функции кодирования и декодирования (например, представленные на рис. 1.2 в виде таблиц I и II). Отображение F строим таким образом, чтобы соотношение $K \bullet F \bullet D = \Phi$ выполнялось.

I		II		III		IV	
$K: A \rightarrow \{0,1\}^3$		$F: \{0,1\}^3 \rightarrow \{0,1\}^2$		$D: \{0,1\}^2 \rightarrow B$		$\Phi: A \rightarrow B$	
A	$\{0,1\}^3$	$\{0,1\}^3$	$\{0,1\}^2$	$\{0,1\}^2$	B	A	B
a1	010	000	--	00	b1	a1	b2
a2	110	001	10	01	--	a2	b3
a3	001	010	10	10	b2	a3	b2
a4	100	011	10	11	b3	a4	b1
a5	011	100	00			a5	b2
		101	--				
		110	11				
		111	--				

Рис. 1.2. Отображение F

Устройство кодирования должно преобразовывать элементы информации множества A заданной природы в двоичные векторы (представленные тем или иным удобным нам способом), причем преобразование это должно быть взаимно однозначно. Аналогично, устройство декодирования должно преобразовывать двоичные векторы (представленные тем или иным способом) в элементы информации множества B. Проблема построения конечного функционального преобразователя теперь сводится к реализации произвольного преобразователя $F: \{0,1\}^m \rightarrow \{0,1\}^n$, который может быть задан, например, таблично, поскольку исходное множество A конечно. Проблема построения произвольного преобразователя $F: \{0,1\}^m \rightarrow \{0,1\}^n$ имеет более четкую математическую формулировку, однако решить ее непросто. Для упрощения решения этой проблемы удобен следующий простой прием: вместо одной функции $F: \{0,1\}^m \rightarrow \{0,1\}^n$ построим n функций $f_i: \{0,1\}^m \rightarrow \{0,1\}$ так, что реализация совокупности этих более простых функций даст искомый преобразователь F. Этот прием представлен на рис. 1.1, в. На рис. 1.1, г функции f_i выделены явно.

Определение 1.1. Функции вида $f: \{0,1\}^m \rightarrow \{0,1\}$, сопоставляющие двоичным векторам двоичные значения, называются двоичными (или булевыми) функциями.

Булевые функции определены на конечном множестве аргументов, каждый из которых принимает только два значения, 0 и 1. Поэтому булевые функции можно представить табличей, перечисляющей для каждого набора значений аргументов значение функции.

Если мы сможем эффективно реализовывать любые двоичные функции, при построении конечных функциональных преобразователей останутся только проблемы кодирования и декодирования, которые должны решаться всякий раз по-своему при каждом специфическом представлении входной и выходной информации.

Булевы функции

Несмотря на то что булевых функций от любого заданного числа m двоичных переменных конечное число, их слишком много, чтобы иметь запас преобразователей для любых встретившихся отображений. Поэтому **основной задачей теории булевых функций является разработка систематического метода построения сложных функций из более простых**. Этот метод основан на изучении свойств булевых функций. Рассмотрим сначала все возможные функции от одной двоичной переменной. Их четыре, две из них — константы (0 и 1), одна — тождественная функция и только одна — функция отрицания (функция НЕ) — является нетривиальной.

p	$\neg p$
0	1
1	0

Очевидно, что число различных булевых функций от m переменных равно 2 в степени 2^m . При $m = 2$ это число 16 , то есть всего функций от двух переменных 16 , однако интересных из них меньше.

Таблица 1.2

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \oplus q$	$p \Leftrightarrow q$	$p \mid q$	$p \downarrow q$
0	0	0	0	1	0	1	1	1
0	1	0	1	1	1	0	1	0
1	0	0	1	0	1	0	1	0
1	1	1	1	1	0	1	0	0

Функция $p \wedge q$ называется функцией **И**, или **конъюнкцией** своих аргументов. Знак конъюнкции часто опускают, иногда в качестве знака конъюнкции используют точку, или $\&$. Функция $p \vee q$ называется функцией **ИЛИ**, или **дизъюнкцией**. Функцию $p \Rightarrow q$ называют **импликацией** или функцией логического следования. Функцию $p \oplus q$ — это функция **сложения по модулю два** (в англоязычной литературе ее называют **XOR** — eXclusive OR). Функция $p \Leftrightarrow q$ называется функцией **эквивалентности**. Знак эквивалентности иногда обозначают \equiv . Функция $p \downarrow q$ имеет название **стрелка Пирса** или функция **НЕ-ИЛИ**. Функцию $p \mid q$ называют функцией **штиrix Шеффера**, или функцией **НЕ-И**.

Суперпозиция двоичных функций может быть записана как формула, которую называют логической формулой.

Пример 1.2

Логическая формула $f = ((\neg p) \vee q) \oplus (p \wedge q \wedge (p \vee r))$ задает функцию от трех переменных как суперпозицию функций одной и двух переменных.

Для уменьшения числа скобок вводятся приоритеты операций. Наиболее приоритетна функция отрицания. Затем идет конъюнкция, после нее — дизъюнкция. Все другие функции имеют равный приоритет, меньший, чем у дизъюнкции. Очевид-

но, что скобками можно установить желаемый порядок операций. Вышеприведенная формула может быть эквивалентно записана так: $f = \neg p \vee q \oplus r q(p \vee r)$. Отметим, что используют и другое распределение приоритетов, в частности, полагая импликацию менее приоритетной, чем все другие функции.

Логическая формула дает возможность построить соответствующий функциональный преобразователь, если мы имеем «элементарные» или «базисные» преобразователи. Для реализации преобразователя f примера 1.2 необходимо иметь элементы, реализующие отрицание, дизъюнкцию, конъюнкцию и сложение по модулю два (см. рис. 1.3).

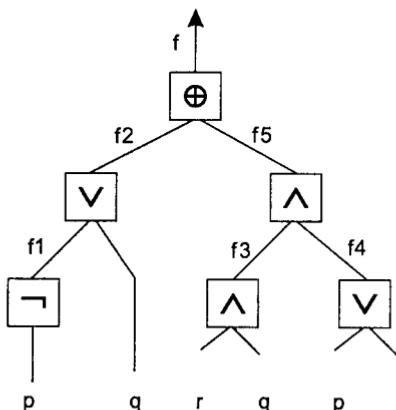


Рис. 1.3. Синтаксическая структура формулы $f = \neg p \vee q \oplus r q(p \vee r)$ примера 1.2

Очевидным образом по формуле можно построить табличное представление функции f .

Таблица 1.3

p	q	r	$f_1 = \neg p$	$f_2 = f_1 \vee q$	$f_3 = r \wedge q$	$f_4 = p \vee r$	$f_5 = f_3 \wedge f_4$	$f = f_2 \oplus f_5$
0	0	0	1	1	0	0	0	1
0	0	1	1	1	0	1	0	1
0	1	0	1	1	0	0	0	1
0	1	1	1	1	1	1	1	0
1	0	0	0	0	0	1	0	0
1	0	1	0	0	0	1	0	0
1	1	0	0	1	0	1	0	1
1	1	1	0	1	1	1	1	0

Очевидно, что суперпозицией нескольких простых булевых функций можно построить более сложную функцию, в частности булеву функцию от большего числа переменных. Поставим вопрос: можно ли суперпозицией фиксированного набора функций представить *любую* булеву функцию от *любого* числа переменных? Удивительно, но ответ на этот вопрос положителен.

Теорема 1.1. Любая двоичная функция может быть представлена как суперпозиция только трех функций: И, ИЛИ и НЕ.

Доказательство этой основной теоремы теории булевых функций основано на свойствах функций И, ИЛИ и НЕ. Их мы сейчас и рассмотрим.

Свойства булевых функций

Ниже приводится несколько полезных свойств булевых функций, которые можно легко проверить с помощью таблиц истинности.

Таблица 1.4

$p \vee q = q \vee p$	$pq = qp$	Коммутативность
$(p \vee q) \vee r = p \vee (q \vee r)$	$(pq)r = p(qr)$	Ассоциативность
$(p \vee q)r = pr \vee qr$	$(pq) \vee r = (p \vee r)(q \vee r)$	Дистрибутивность
$\neg(p \vee q) = \neg p \neg q$	$\neg(pq) = \neg p \vee \neg q$	Законы де Моргана
$p \neg p = 0$	$p \vee \neg p = 1$	Законы исключенного третьего
$(p \vee q)p = p$	$(pq) \vee p = p$	Законы поглощения
$p \wedge 0 = 0$	$p \vee 0 = p$	Свойства нуля
$p \wedge 1 = p$	$p \vee 1 = 1$	Свойства единицы
$\neg \neg p = p$	$\neg 1 = 0; \neg 0 = 1$	Свойства отрицания
$p \Leftrightarrow q = (p \Rightarrow q)(q \Rightarrow p)$	$p \Rightarrow q = \neg p \vee q$	Свойства импликации
$p \oplus q = \neg pq \vee p \neg q$	$1 \oplus p = \neg p$	Свойства сложения по модулю два

Две булевые функции Φ и F равны (или равносильны), если их значения совпадают на всех наборах аргументов (обозначать это будем $\Phi = F$). Очевидно, что если булевые функции Φ и F равносильны, то $\Phi \Leftrightarrow F$ принимает единичные значения на всех наборах аргументов.

Доказательство теоремы 1.1 основано на следующей лемме (вспомогательном утверждении).

Лемма 1.1. «О дизъюнктивном разложении». Любая булева функция $f(x_1 \dots x_m)$ от m переменных может быть представлена так:

$$f(x_1, \dots, x_i, \dots, x_m) = \neg x_i f(x_1, \dots, 0, \dots, x_m) \vee x_i f(x_1, \dots, 1, \dots, x_m).$$

Доказательство. При $x_i = 0$ правая часть принимает значение

$$\neg 0 \wedge f(x_1, \dots, 0, \dots, x_m) \vee 0 \wedge f(x_1, \dots, 1, \dots, x_m),$$

что после использования свойств булевых функций элементарно приводится к $f(x_1, \dots, 0, \dots, x_m)$. Аналогично, при $x_i = 1$ правая часть приводится к $f(x_1, \dots, 1, \dots, x_m)$.

Доказательство теоремы 1.1. Разложим функцию $f(x_1, \dots, x_m)$ последовательно по переменным x_1, x_2, \dots, x_m :

$$\begin{aligned} f(x_1, \dots, x_m) &= \neg x_1 f(0, x_2, \dots, x_m) \vee x_1 f(1, x_2, \dots, x_m) = \\ &= \neg x_1 \neg x_2 f(0, 0, x_3, \dots, x_m) \vee \neg x_1 x_2 f(0, 1, x_3, \dots, x_m) \vee \\ &\quad \vee x_1 \neg x_2 f(1, 0, x_3, \dots, x_m) \vee x_1 x_2 f(1, 1, \dots, x_m) = \dots \\ &= \neg x_1 \neg x_2 \dots \neg x_m f(0, 0, \dots, 0) \vee \neg x_1 \neg x_2 \dots x_m f(0, 0, \dots, 1) \vee \dots \vee x_1 x_2 \dots x_m f(1, 1, \dots, 1). \end{aligned}$$

Таким образом, любую булеву функцию можно представить как дизъюнкцию термов, каждый из которых представляет собой конъюнкцию всех переменных (с отрицаниями или без них) и значений этой функции на соответствующем конкретном наборе значений переменных. Обозначим для $\sigma \in \{0,1\}$, $x^\sigma = \neg x$ при $\sigma = 0$ и $x^\sigma = x$ при $\sigma = 1$. Тогда окончательная форма представления любой булевой функции будет иметь вид:

$$f(x_1, \dots, x_m) = \bigvee_{\sigma_1 \in \{0,1\}} x_1^{\sigma_1} x_2^{\sigma_2} \dots x_m^{\sigma_m} f(\sigma_1, \sigma_2, \dots, \sigma_m).$$

Поскольку значения функции на каждом конкретном наборе равны либо 0, либо 1, то в формуле останутся только такие термы, которые соответствуют наборам переменных, на которых функция равна единице:

$$f(x_1, \dots, x_m) = \bigvee_{f(\sigma_1, \sigma_2, \dots, \sigma_m)=1} x_1^{\sigma_1} x_2^{\sigma_2} \dots x_m^{\sigma_m}.$$

В соответствии с этой теоремой функция $f(p, q, r) = \neg p \vee q \oplus qr(p \vee r)$ примера 1.2 на основании ее табличного представления (табл. 1.3) имеет следующий вид (как суперпозиция функций И, ИЛИ, НЕ):

$$f(p, q, r) = \neg p \neg q \neg r \vee \neg p \neg q r \vee \neg p q \neg r \vee p q \neg r.$$

Хотя эта запись чуть более сложна, чем запись исходной функции, ее неоспоримое преимущество в том, что она выражена через суперпозицию **только** функций И, ИЛИ, НЕ. Более того, такое представление для двоичных функций единственно (с точностью до перестановок термов).

Определение 1.2. *Функционально полным набором (или базисом) будем называть такое множество двоичных функций, суперпозицией которых могут быть выражены любые булевые функции.*

Очевидно, множество булевых функций {И, ИЛИ, НЕ} является базисом: теорема 1.1 говорит о том, что их суперпозицией можно представить любую булеву функцию от любого числа аргументов. Базис {И, ИЛИ, НЕ} называется базисом Буля.

Нормальные формы представления булевых функций

Определение 1.3. *Представление булевой функции в форме дизъюнкции:*

$$f(x_1, \dots, x_n) = K_1 \vee K_2 \vee \dots \vee K_n, n \geq 1,$$

где каждый терм K_i (или конъюнкт) представляет собой конъюнкцию взятых с отрицаниями или без них двоичных переменных функции, называется дизъюнктивной нормальной формой этой функции (или ДНФ). Если каждый конъюнкт содержит в точности по одной все (взятые с отрицаниями или без них) двоичные переменные функции, ДНФ называется совершенной дизъюнктивной нормальной формой этой функции (или СДНФ).

Теорема 1.1 гарантирует, что в СДНФ может быть представлена любая булева функция за исключением тождественного нуля.

Теорема 1.2. *Представление булевых функций в СДНФ единственно.*

Следствием этой теоремы является то, что две функции можно сравнивать по их СДНФ.

Пример 1.3

Примерами ДНФ функций являются:

$$\Phi_1(p, q, r) = \neg p \neg q \neg r \vee \neg p \neg q r \vee \neg p q \neg r \vee p q \neg r,$$

$$\Phi_2(p, q, r) = \neg p \neg q \vee p \neg r \neg q \vee \neg p \neg r \vee \neg p q,$$

$$\Phi_3(p, q, r) = \neg p \neg q \vee \neg p \neg q r \vee q \neg r.$$

Первая функция – это СДНФ (и, следовательно, ДНФ) функции $f(p, q, r)$ примера 1.2, вторая содержит терм, куда p входит дважды: с отрицанием и без него. Все три функции равны: все эти формулы представляют одну и ту же функцию, заданную первоначально формулой $\neg p \vee q \oplus qr(p \vee r)$.

Существуют другие нормальные представления булевой функции в виде конъюнкции дизъюнкций, полностью симметричные СДНФ и ДНФ.

Определение 1.4. Представление булевой функции в форме конъюнкции:

$$f(x_1, \dots, x_m) = D_1 \wedge D_2 \wedge \dots \wedge D_n, n \geq 1,$$

где каждый терм D_i представляет собой дизъюнкцию (взятых с отрицаниями или без них) каких-либо двоичных переменных функции (дизъюнктов), называется конъюнктивной нормальной формой этой функции (или КНФ). Если каждый дизъюнкт КНФ содержит в точности по одной все (взятые с отрицаниями или без них) двоичные переменные функции, то такая КНФ называется совершенной конъюнктивной нормальной формой этой функции (или СКНФ).

Теорема 1.3. Любая булева функция (не равная 1) может быть представлена в СКНФ, причем представление любой функции в СКНФ единственно.

Доказательство теоремы может быть проведено аналогично доказательству теоремы 1.1 на основании следующей леммы Шеннона о конъюнктивном разложении.

Лемма 1.2. (Клод Шенон) Любая булева функция $f(x_1, \dots, x_i, \dots, x_m)$ от m переменных может быть представлена так:

$$f(x_1, \dots, x_i, \dots, x_m) = (\neg x_i \vee f(x_1, \dots, 1, \dots, x_m)) \wedge (x_i \vee f(x_q, \dots, 0, \dots, x_m)).$$

Проверить справедливость леммы 1.2 можно простой подстановкой возможных значений переменной x_i . Доказательство леммы и теоремы остается для самостоятельной работы читателя.

Преобразование в нормальную форму

Всякая аналитическая запись функции может быть преобразована в нормальную форму. Приведем систематическую процедуру преобразования функции в нормальную форму с использованием свойств двоичных функций из табл. 1.4.

Шаг 1. Используем свойства

$$p \Leftrightarrow q = (p \Rightarrow q)(q \Rightarrow p), p \Rightarrow q = \neg p \vee q, p \oplus q = \neg p q \vee p \neg q$$

для того, чтобы устраниТЬ операции $\Leftrightarrow, \Rightarrow, \oplus$, оставив только операции И, ИЛИ, НЕ.

Шаг 2. Используем свойства отрицания и законы де Моргана:

$$\neg(p \vee q) = \neg p \neg q, \neg(pq) = \neg p \vee \neg q,$$

чтобы каждая операция отрицания относилась только к одной переменной.

Шаг 3. Используем свойства дистрибутивности и другие свойства, чтобы получить нормальную форму. Заметим, что для получения СКНФ часто нужно использовать следующее свойство дистрибутивности: $(pq) \vee r = (p \vee r)(q \vee r)$.

Пример 1.4

Преобразуем аналитически функцию $f(p, q, r) = \neg p \vee q \oplus rq(p \vee r)$ примера 1.2 в СКНФ.

Применяя шаги процедуры, получим:

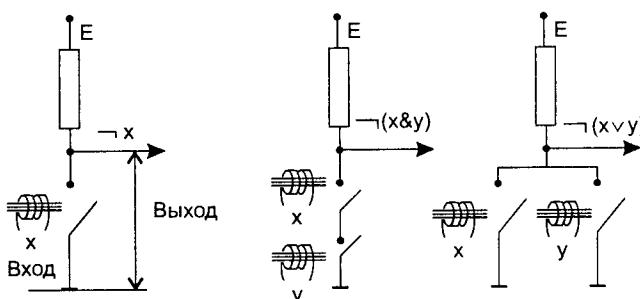
$$\begin{aligned} f(p, q, r) &= \neg p \vee q \oplus rq(p \vee r) = (\neg(\neg p \vee q))rq(p \vee r) \vee (\neg p \vee q) \neg(rq(p \vee r)) = \\ &= p \neg qrq(p \vee r) \vee (\neg p \vee q)(\neg r \vee \neg q \vee (\neg p \neg r)) = (\neg p \vee q)(\neg r \vee \neg q \vee (\neg p \neg r)) = \\ &= (\neg p \vee q)(\neg r \vee \neg q \vee \neg p)(\neg r \vee \neg q \vee \neg r) = (\neg p \vee q)(\neg q \vee \neg r). \end{aligned}$$

Таким образом, мы получили КНФ функции $f(p, q, r)$. Чтобы получить ее СКНФ, нужно каждый дизъюнкты, в котором не хватает какой-либо переменной, повторить дважды: с этой переменной и с ее отрицанием:

$$f(p, q, r) = (\neg p \vee q \vee r)(\neg p \vee q \vee \neg r)(p \vee \neg q \vee \neg r)(\neg p \vee \neg q \vee \neg r).$$

Реализация булевых функций

Основная проблема в реализации двоичных функций — это проблема адекватного представления дискретных значений 0 и 1 с помощью физических величин, имеющих обычно непрерывный диапазон изменений. В электрической цепи значения 0 и 1 удобно представлять соответственно низким и высоким уровнями тока или напряжения. Наиболее естественно использовать для этого ключ-реле. Следующий рисунок показывает, как с помощью электромеханического реле представить все три базовые двоичные функции.



Рассмотрим первую схему. Подача на вход x высокого потенциала приведет к замыканию ключа и, следовательно, к падению напряжения на выходе схемы. При низком входном потенциале x ключ разомкнут, и на выходе схемы сохраняется высокое напряжение E . Очевидно, что если высокое напряжение считать единицей, а низкое — нулем, то эта схема реализует отрицание. В современных схемах роль электронного ключа без механических деталей выполняет так называемый

вентиль, или транзисторный переход. Совокупность вентилей изготавляется на кремниевой подложке с высокой степенью интеграции. В одном блоке — чипе — может помещаться схема из 10^6 и более транзисторных переходов. Современная технология позволяет с использованием только одного вида вентилей изготавливать на одной подложке крупные узлы вычислительной техники: процессоры, память и т. п. Возможность автоматизированного построения в едином технологическом цикле логических схем огромной сложности стала главной причиной последних революционных изменений в области всей вычислительной техники.

Элементарные схемы функций базиса Буля изображаются так.

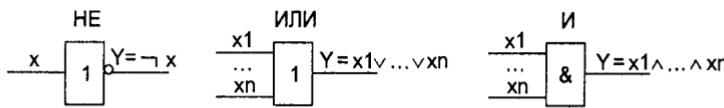


Рис. 1.4. Изображение схем, реализующих функции базиса Буля

Минимизация булевых функций

Сложность схемы, реализующей булеву функцию, определяется сложностью ее аналитической записи. Поскольку одну и ту же булеву функцию можно представить различными формулами, то задача выбора наиболее простой формулы, задающей булеву функцию, непосредственно ведет к наиболее простой схеме, то есть, в частности, к экономии материалов, объема, веса и энергопотребления схемы. Минимальной формой булевой функции в некотором базисе можно считать такую, которая содержит минимальное число суперпозиций функций базиса, допуская скобки. Однако построить эффективный алгоритм такой минимизации с получением минимальной скобочной формы трудно.

Рассмотрим более простую задачу минимизации при синтезе комбинационных схем, при которой ищется не минимальная скобочная форма функции, а ее минимальная ДНФ. Для этой задачи существуют простые эффективные алгоритмы. Рассмотрим два из них.

Метод Квайна. Минимизируемая функция представляется в СДНФ, и к ней применяются все возможные операции неполного склеивания

$$Kx \vee K\neg x = K \vee Kx \vee K\neg x,$$

а затем поглощения $K \vee Kx = K$, и эта пара этапов применяется многократно.

Карта Карно. Это двумерная табличная форма представления булевой функции, позволяющая в графической наглядной форме легко отыскать минимальные ДНФ логических функций. Каждой клетке в таблице сопоставляется терм СДНФ минимизируемой функции, причем так, что любым осям симметрии таблицы соответствуют зоны, взаимно инверсные по какой-либо переменной. Такое расположение клеток в таблице позволяет легко определить склеивающиеся термы СДНФ (отличающиеся знаком инверсии только одной переменной): они располагаются в таблице симметрично. В таблице (рис. 1.5, б) представлена карта Карно для функ-

ции двух переменных — импликации. Все четыре клетки соответствуют всем возможным конъюнкциям СДНФ функции 2 переменных. Единичные значения функции показывают те термы, которые присутствуют в СДНФ этой функции. Расположения термов для карты Карно функции 2 переменных (рис. 1.5, а) таково, что соседние клетки соответствуют склеивающимся термам, отличающимся только одной переменной: в один конъюнкт эта переменная входит без отрицания, а в другой — с отрицанием. В соответствии с рис. 1.5, б импликацию можно представить минимальной ДНФ: $\neg x \vee y$.

$y \quad \neg y$ $x \quad \boxed{\begin{array}{ c c } \hline xy & xy \\ \hline \neg xy & \neg xy \\ \hline \end{array}}$	$y \quad \neg y$ $x \quad \boxed{\begin{array}{ c c } \hline 1 & 0 \\ \hline 1 & 1 \\ \hline \end{array}}$
<i>a</i>	<i>b</i>

Рис. 1.5. Кarta Карно для импликации $x \Rightarrow y$

На рис. 1.6, а представлена карта Карно функции $f(p, q, r) = \neg p \vee q \oplus qr (\neg p \vee r)$ примера 1.2 на основании ее табличного представления (табл. 1.3). Ее минимальная ДНФ из рис. 1.6, а равна $q \neg r \vee \neg p \neg q$.

$q \quad \neg q$ $p \quad \boxed{\begin{array}{ c c c c } \hline 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array}}$	$y \quad \neg y$ $x \quad \boxed{\begin{array}{ c c c c } \hline 1 & 0 & - & 1 \\ \hline 1 & 1 & 1 & - \\ \hline 0 & 1 & - & 0 \\ \hline - & - & 0 & 0 \\ \hline \end{array}}$
<i>a</i>	<i>b</i>

Рис. 1.6. Карты Карно функций 3 и 4 переменных

Карты Карно удобны для минимизации и неполностью определенных функций. Например, на рис 1.6, б представлена функция 4 переменных, у которой не определено 5 значений. Неопределенные значения можно заменить любыми — 0 или 1. Следовательно, функцию на рис. 1.6, б можно заменить любой из 32 полностью определенных функций. Покрывая таблицу минимальным числом максимальных квадратов (или прямоугольников) со сторонами, равными степени двойки, так, чтобы они обязательно покрыли все единицы и не покрывали ни одного нуля, получим следующую минимальную ДНФ всех возможных функций, представленных на этой диаграмме: $x \neg z \vee zu$.

Пример 1.5

Таблица 1.5 повторяет изображенную на рис. 1.2 таблицу II, представляя частичное отображение F из множества трехразрядных двоичных векторов в множество

двуухразрядных двоичных векторов примера 1.1. Две двоичные функции f_1 и f_2 представляют соответствующие разряды результата отображения F .

Таблица 1.5

X	Y	z	f1	f2
0	0	0	-	-
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	-	-
1	1	0	1	1
1	1	1	-	-

Из карт Карно (рис. 1.7) легко видеть, что минимальные ДНФ этих функций имеют вид: $f_1 = y \vee \neg x$ и $f_2 = xy$.

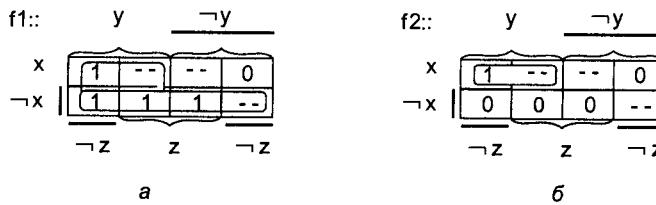


Рис. 1.7. Карта Карно для функций f_1 и f_2 примера 1.5

Пример 1.6

Рассмотрим задачу построения схемы отображения цифр, которые набирает пользователь, нажимая клавиши на клавиатуре микрокалькулятора, в их изображение (рис. 1.8) (подобное устройство используется, например, футбольными судьями для высвечивания номера заменяемого футболиста). Двоичное кодирование цифр осуществляется непосредственно в момент нажатия клавиши в четырехразрядный двоично-десятичный код. Декодирование изображений осуществляется семисегментной структурой светодиодов, специальное расположение которых позволяет высвечивать различные цифры: подача напряжения на соответствующую полоску f_0-f_6 вызывает ее свечение. Непосредственно логическая схема отображения СО имеет четыре двоичных входа и семь двоичных выходов (по числу сегментов светодиодов).

Пусть в схеме, кроме десяти цифр со стандартным двоично-десятичным кодированием, возможно представить знак минус кодом $<1111>$. Семь двоичных функций f_0-f_6 представлены таблицей истинности 1.6; функции эти не полностью определены, поскольку коды, не соответствующие десятичным цифрам и знаку минус, не могут появиться на входе схемы отображения. Мы можем дополнить их произвольными значениями исходя, например, из критерия минимальности фун-

кции. Заполнение табл. 1.6 очевидно: например, для изображения цифры 5 (двоичный код на входе CO <0101>) следует подать напряжение на все полоски свето-диодов, кроме f3 и f6.

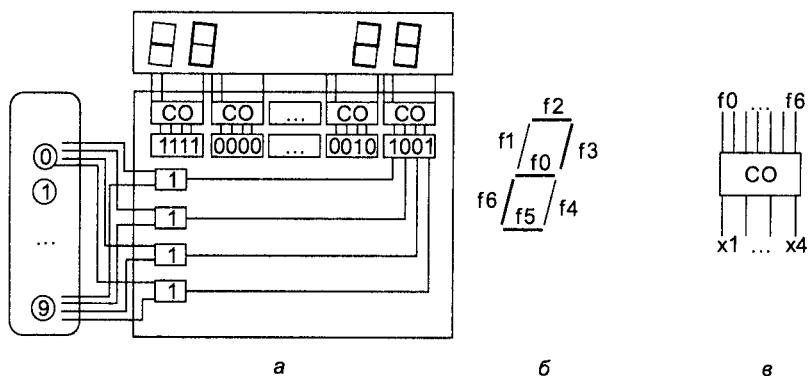


Рис. 1.8. Отображение цифр в микрокалькуляторе (а); семисегментная структура светодиодов для отображения цифры 5 (б); один разряд схемы отображения (в)

Таблица 1.6

x1	x2	x3	x4	f0	f1	f2	f3	f4	f5	f6
0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	0	1	1
0	0	1	1	1	0	1	1	1	1	0
0	1	0	0	1	1	0	1	1	0	0
0	1	0	1	1	1	1	0	1	1	0
0	1	1	0	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	0
1	0	1	0	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-
1	1	1	1	1	0	0	0	0	0	0

Минимизация неполностью определенных функций f0–f6 может быть легко проведена с помощью карт Карно.

На рис. 1.9 приведены две карты Карно функций f0 и f2. После минимизации можем записать:

$$f0 = x1x4 \vee \neg x2x4 \vee x2\neg x3 \vee x2\neg x4 \vee x1\neg x3;$$

$$f2 = \neg x1x2 \vee x1\neg x2 \vee \neg x3\neg x4 \vee \neg x1x3x4.$$

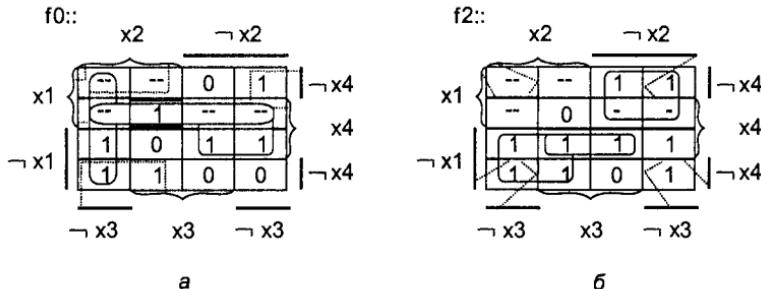


Рис. 1.9. Карты Карно функций f_0 и f_2 схемы отображения микрокалькулятора

В выпускаемых промышленностью схемах отображения присутствует также дополнительный вход для высвечивания десятичной точки и другой вход для подавления изображения незначащих нулей в старших разрядах числа.

Для минимизации функций пяти и более переменных использование карт Карно затруднено, во-первых, большей громоздкостью таблиц и, во-вторых, тем, что при числе переменных большем 2 не удается расположить все склеиваемые термы рядом. Тем не менее и для пяти, и для шести переменных использование этого подхода полезно. Карты Карно используются во многих приложениях. Один из удачных примеров применения этого подхода для решения проблемы анализа и синтеза законов управления в системе «Импульсный усилитель мощности – электродвигатель» приведен в [30]. Приведем еще один пример использования карт Карно.

Пример 1.7

Рассмотрим проблему словесной формулировки условия достижения консенсуса (общего согласия) в вычислительной сети [12]. Проблема ставится следующим образом. Возможность достижения консенсуса в сети зависит от четырех параметров сети:

- тип процессоров; процессоры могут быть асинхронными (A) или синхронными (C);
- сохранение порядка сообщений в канале: {порядок сохраняется (П), либо без сохранения (Б)};
- вид передачи сообщений: {широковещательная передача (Ш), точка-точка (Т)};
- ограничение коммуникации во времени: {коммуникация ограничена (О), не ограничена (Н)}.

Исследования [12] показали, что консенсус достижим только в сетях со следующими наборами параметров: (A, П, Ш, Н), (A, П, Ш, О), (C, Б, Т, О), (C, Б, Ш, О), (C, П, Ш, Н), (C, П, Ш, О), (C, П, Т, Н), (C, П, Т, О). Как коротко сформулировать условия достижимости консенсуса в вычислительных сетях?

На основе теории двоичных функций это можно сделать, не понимая, что такое консенсус, и не имея никакого представления о вычислительных сетях. Построим

карту Карно двоичной функции *Консенсус* четырех двоичных переменных, каждая из которых соответствует одному из указанных параметров.

Простая минимизация дает следующее выражение этой функции:
 $C_P \vee C_O \vee \text{ШП} = C(P \vee O) \vee \text{ШП}$.

Выразим эту формулу на естественном языке: «Консенсус достичим только в вычислительных сетях с синхронным процессором, в которых или сохраняется порядок сообщений, или есть ограничения коммуникации по времени, а также в таких сетях, в которых возможна широковещательная передача сообщений с сохранением их порядка». Эквивалентную формулировку получим из эквивалентной формулы: $(C \vee \text{Ш})P \vee CO$. Это дает: «Консенсус достичим только в вычислительных сетях или с синхронным процессором или широковещательных, в которых сохраняется порядок сообщений, а также в таких сетях с синхронным процессором, в которых существуют ограничения времени на передачу сообщений».

Сообщения

	Широковещательные				Точка-точка
	1	1	1	1	
Синхронный	1	1	1	1	Порядок сохраняется
Процессор	0	1	1	0	Передача сообщений
Асинхронный	0	0	0	0	Без сохранения порядка
	1	1	0	0	Порядок сохраняется
Нет ограничения				Нет ограничения	

Время доставки сообщений

Единицы стоят для значений функции на наборах:
 $(A, P, \text{Ш}, H), (A, P, \text{Ш}, O), (C, B, T, O), (C, B, \text{Ш}, O),$
 $(C, P, \text{Ш}, H), (C, P, \text{Ш}, O), (C, P, T, H), (C, P, T, O)$

Функциональная полнота

В соответствии с определением 1.2, функционально полным набором (или базисом) называется такое множество булевых функций, суперпозицией которых могут быть выражены любые булевые функции. Один из таких базисов — базис Буля — нами определен: это три функции **И**, **ИЛИ**, **НЕ**. Исследование проблем, связанных с базисами, чрезвычайно важно для практики: функции базиса — это тот полный набор строительных блоков, из которых можно строить все другие двоичные функции от любого числа переменных, а следовательно, реализовывать любые конечные функциональные преобразователи.

Важными для практики и интересными с теоретической точки зрения являются вопросы:

- **почему** функции **И**, **ИЛИ**, **НЕ** такие особенные, что с их помощью можно построить любую другую булеву функцию?
- **существуют** ли еще какие-нибудь базисы, кроме базиса Буля?
- **является** ли некоторый заданный базис минимальным (то есть не содержит ли он излишних функций, выражющихся суперпозицией других)?
- **как** проверить, является ли заданный набор функций базисом, и если не является, как дополнить его другими функциями, чтобы получившееся множество составило базис?

Введем некоторые определения и обозначения.

Определение 1.5. Замыканием множества M булевых функций назовем такое множество булевых функций, которые можно получить суперпозицией функций из M . Замыкание множества M обозначим $[M]$.

Пусть B — множество всех двоичных функций. Очевидно, что множество M двоичных функций будет базисом, только если $[M] = B$. Рассмотрим свойства замыканий двоичных функций.

Теорема 1.4. Пусть $M, N \subseteq B$. Тогда:

- а) $M \subseteq [M]$;
- б) $[[M]] = [M]; [B] = B$;
- в) $M \subseteq N \Rightarrow [M] \subseteq [N]$;
- г) $[M] \subseteq B$;
- д) если M — базис и $M \subseteq [N]$, то N — тоже базис.

Доказательство теоремы просто. Утверждения а), б), в) и г) следуют непосредственно из определений. Докажем д). $M \subseteq [N] \Rightarrow [M] \subseteq [[N]]$ на основании в), следовательно, $[M] \subseteq [N]$ на основании б). Но поскольку M — базис, $[M] = B$. Отсюда $B \subseteq [N]$, но поскольку г) $[N] \subseteq B$, то $[N] = B$.

Попробуем найти другие базисы, отличные от базиса Буля. Согласно законам де Моргана, $\neg(p \vee q) = \neg p \wedge \neg q$. Следовательно, $p \vee q = \neg(\neg p \wedge \neg q)$. Таким образом, дизъюнкция выражается через конъюнкцию и отрицание, следовательно, суперпозицией функций **{И, НЕ}** можно построить все функции базиса Буля — то есть **ИЛИ** можно выбросить из этого базиса. Некоторые другие базисы представлены в табл. 1.7. Их обоснование очевидно.

Рассмотрим конъюнктивный базис. Он является **минимальным**, поскольку выбрасывание из множества **{И, НЕ}** любой функции превращает оставшееся одноэлементное множество в не-базис. Действительно, например, с помощью суперпозиции произвольного числа функций **НЕ** можно построить только функцию **НЕ** и тождественную функцию **ИДЕНТ**, то есть $f(x) = x$. Заметим, что суперпозицией унарных функций множества $M = \{\text{ИДЕНТ}, \text{НЕ}\}$ можно построить только функции этого множества. Множество булевых функций, обладающее этим свойством, называется **замкнутым классом** двоичных функций.

Таблица 1.7

Обоснование	Базис
$\neg(p \vee q) = \neg p \neg q$; следовательно, $p \vee q = \neg(\neg p \neg q)$	{И, НЕ} — конъюнктивный базис
$\neg(pq) = \neg p \vee \neg q$; следовательно, $pq = \neg(\neg p \vee \neg q)$	{ИЛИ, НЕ} — дизъюнктивный базис
$\neg p = p \oplus 1$;	{И, \oplus , 1} — базис Жегалкина
$p q = \neg(pq)$; следовательно, $\neg p = p q$; $pq = \neg(p q) = (p q) p q$	{ } — базис Шеффера
$p \downarrow q = \neg(p \vee q)$; следовательно, $\neg p = p \downarrow p$; $p \vee q = \neg(p \downarrow q) = (p \downarrow q) \downarrow (p \downarrow q)$;	{ \downarrow } — базис Пирса

Пример 1.8

Конъюнкции, то есть все функции вида $x_1 \wedge x_2 \wedge \dots \wedge x_m$, тоже составляют замкнутый класс. Очевидно, однако, что, например, функция, которая на наборе $(0, 0, \dots, 0)$ имеет значение 1, нельзя представить суперпозицией таких функций. Таким образом, {И} не является базисом, следовательно, конъюнктивный базис {И, НЕ} является минимальным.

Рассмотрим более подробно базис Жегалкина.

Алгебра Жегалкина и линейные функции

Алгебра Жегалкина — это алгебра над множеством двух бинарных булевых функций (И, \oplus) и нульварной функции 1. Легко проверить следующие соотношения в этой алгебре:

$$\begin{aligned} p \oplus q &= q \oplus p; \\ p(q \oplus r) &= pq \oplus pr; \\ p \oplus p &= 0; \\ p \oplus 0 &= p. \end{aligned}$$

Справедливы в этой алгебре, конечно, и все соотношения табл. 1.4, включающие эти функции. Если в произвольной формуле, включающей только функции базиса Жегалкина, раскрыть скобки, то получим бесскобочную формулу, имеющую вид суммы (по модулю два) произведений, то есть некоторый полином. Он называется полиномом Жегалкина.

Пример 1.9

Преобразуем аналитически функцию $f(p, q, r) = \neg p \vee q \oplus qr(p \vee r)$ примера 1.2 в полином Жегалкина. Поскольку $\neg q = 1 \oplus p$, а

$$p \vee q = \neg(\neg p \neg q) = 1 \oplus (1 \oplus p)(1 \oplus q) = p \oplus q \oplus pq,$$

то после подстановки и раскрытия скобок имеем:

$$\begin{aligned} f(p, q, r) &= \neg p \vee q \oplus qr(p \vee r) = ((1 \oplus p) \oplus q \oplus (1 \oplus p)q) \oplus (qr(p \oplus r \oplus pr)) = \\ &= (1 \oplus p \oplus q \oplus q \oplus pq) \oplus (pqr \oplus qr \oplus pqr) = 1 \oplus p \oplus pq \oplus qr. \end{aligned}$$

Теорема 1.5. Любая булева функция может быть представлена в виде полинома Жегалкина, причем единственным образом.

Доказательство. Существование полинома для любой функции гарантируется тем, что функции $\{\text{И}, \oplus, 1\}$ образуют базис. Далее, легко видеть, что число возможных членов в полиноме с m переменными равно 2^m . Поэтому число различных полиномов Жегалкина от m переменных равно 2 в степени 2^m , то есть числу возможных двоичных функций от m переменных. Поскольку одна и та же формула не может представлять различные функции, то тем самым между множествами двоичных функций и полиномов Жегалкина от m переменных установлено взаимнооднозначное соотношение.

Пример 1.10

Построим для функции $f(p, q, r) = \neg p \vee q \oplus rq (p \vee r)$ примера 1.2 полиномом Жегалкина непосредственно из таблицы истинности (см. табл. 1.3). Эту таблицу повторим здесь.

Таблица 1.8

p	q	r	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Будем искать коэффициенты полинома

$$f(p, q, r) = a_0 \oplus a_p p \oplus a_q q \oplus a_r r \oplus a_{pq} pq \oplus a_{pr} pr \oplus a_{qr} qr \oplus a_{pqr} pqr$$

Всего коэффициентов 8, каждый коэффициент может быть 0 или 1, число возможных вариантов равно $2^8 = 256$ – как раз столько, сколько всех возможных булевых функций от трех переменных. Для нахождения коэффициентов заданной функции используем таблицу ее значений.

Искомые коэффициенты последовательно найдем из следующей системы уравнений:

$$f(0, 0, 0) = 1 = a_0; \text{ отсюда } a_0 = 1;$$

$$f(1, 0, 0) = 0 = a_0 \oplus a_p = 1 \oplus a_p; \text{ отсюда } a_p = 1;$$

$$f(0, 1, 0) = 1 = a_0 \oplus a_q = 1 \oplus a_q; \text{ отсюда } a_q = 0;$$

$$f(0, 0, 1) = 1 = a_0 \oplus a_r = 1 \oplus a_r; \text{ отсюда } a_r = 0;$$

$$f(1, 1, 0) = 1 = a_0 \oplus a_p \oplus a_q \oplus a_{pq} = 1 \oplus 1 \oplus 0 \oplus a_{pq}; \text{ отсюда } a_{pq} = 1;$$

$$f(1, 0, 1) = 0 = a_0 \oplus a_p \oplus a_r \oplus a_{pr} = 1 \oplus 1 \oplus 0 \oplus a_{pr}; \text{ отсюда } a_{pr} = 0;$$

$$f(0, 1, 1) = 0 = a_0 \oplus a_q \oplus a_r \oplus a_{qr} = 1 \oplus 0 \oplus 0 \oplus a_{qr}; \text{ отсюда } a_{qr} = 1;$$

$f(1,1,1) = 0 = a_0 \oplus a_p \oplus a_q \oplus a_r \oplus a_{pq} \oplus a_{pr} \oplus a_{qr} \oplus a_{pqr} = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus a_{pqr}$;
отсюда $a_{pqr} = 0$.

Найденное представление совпадает с представлением, полученным для этой функции ранее аналитически: $f(p, q, r) = 1 \oplus p \oplus pq \oplus qr$.

Булева функция, полином Жегалкина которой имеет вид $a_0 \oplus \sum a_{xi}x_i$, называется линейной.

Замкнутые классы булевых функций

Решение проблемы нахождения любых других базисов булевых функций, отличных от указанных в табл. 1.5, было найдено Эмилем Постом в 1921 году на пути изучения замкнутых классов таких функций. **Множество М булевых функций называется замкнутым классом, если $[M] = M$** . Мы уже встречали замкнутые классы булевых функций: например, $[\text{ИДЕНТ, НЕ}] = \{\text{ИДЕНТ, НЕ}\}$. Другие примеры — (а) множество всех конъюнкций от различного числа аргументов, (б) множество всех дизъюнкций от различного числа аргументов. Существует множество разных замкнутых классов функций. Э. Постом было выделено пять таких классов, на основе которых им была решена проблема базисов булевых функций. Ниже для каждого такого класса вводится его определение, примеры и формулируется теорема о том, что множество функций, обладающих указанным свойством С, составляет замкнутый класс. Доказать эти теоремы просто, доказательство основано на простой индукции по глубине суперпозиции функций. Базу индукции составляет утверждение о том, что тождественная функция обладает указанным свойством С, а шаг индукции — что если произвольная функция $f(x_1, x_2, \dots, x_m)$ обладает свойством С и все функции F_1, F_2, \dots, F_m обладают свойством С, то функция $f(F_1, F_2, \dots, F_m)$ тоже обладает свойством С. Более подробно о замкнутых классах можно прочитать в [23].

Определение 1.6. Функция $f(x_1, x_2, \dots, x_m)$ сохраняет ноль, если $f(0, 0, \dots, 0) = 0$.

Примером функции, сохраняющей ноль, является конъюнкция. Отрицание не сохраняет ноль. Функция $f(p, q, r) = \neg p \vee q \otimes r q(p \vee r)$ примера 1.2 не сохраняет ноль; из табл. 1.3: $f(0, 0, 0) = 1$. Обозначим M_0 множество всех функций, сохраняющих ноль.

Теорема 1.6. $[M_0] = M_0$.

Определение 1.7. Функция $f(x_1, x_2, \dots, x_m)$ сохраняет единицу, если $f(1, 1, \dots, 1) = 1$.

Примером функции, сохраняющей единицу, является конъюнкция. Отрицание не сохраняет единицу. Функция $f(p, q, r) = \neg p \vee q \oplus r q(p \vee r)$ примера 1.2 не сохраняет единицу; из табл. 1.3: $f(1, 1, 1) = 0$. Обозначим M_1 множество всех функций, сохраняющих единицу.

Теорема 1.7. $[M_1] = M_1$.

Введем отношение порядка на множестве двоичных наборов.

$$(x_1, x_2, \dots, x_m) \leq (y_1, y_2, \dots, y_m)$$

тогда и только тогда, когда для любого $1 \leq i \leq m$, выполняется $x_i \leq y_i$.

Определение 1.8. Функция $f(x_1, x_2, \dots, x_m)$ называется **монотонной**, если для любых двух наборов (x_1, x_2, \dots, x_m) и (y_1, y_2, \dots, y_m) , выполнение $(x_1, x_2, \dots, x_m) \leq (y_1, y_2, \dots, y_m)$ влечет $f(x_1, x_2, \dots, x_m) \leq f(y_1, y_2, \dots, y_m)$.

Обозначим $M_{\text{мн}}$ множество всех монотонных функций. Примером монотонной функции является конъюнкция. Отрицание не является монотонным. Функция $f(p, q, r) = \neg p \vee q \oplus rq(p \vee r)$ примера 1.2 не является монотонной; из табл. 1.3: $f(0, 0, 0) = 1$, а $f(0, 1, 1) = 0$.

Теорема 1.8. $[M_{\text{мн}}] = M_{\text{мн}}$.

Определение 1.9. Функция $f(x_1, x_2, \dots, x_m)$ называется **самодвойственной**, если для любого набора (x_1, x_2, \dots, x_m) , $f(\neg x_1, x_2, \dots, \neg x_m) = \neg f(x_1, x_2, \dots, x_m)$.

Примером самодвойственной функции является отрицание. Конъюнкция не является самодвойственной функцией. Функция $f(p, q, r) = \neg p \vee q \oplus rq(p \vee r)$ примера 1.2 не является самодвойственной. Действительно, из табл. 1.3:

$$f(0, 0, 1) = f(1, 1, 0) = 1.$$

Обозначим $M_{\text{сам}}$ множество всех самодвойственных функций.

Теорема 1.9. $[M_{\text{сам}}] = M_{\text{сам}}$.

Определение 1.10. Функция $f(x_1, x_2, \dots, x_m)$ называется **линейной**, если ее полином Жегалкина имеет вид $a_0 \oplus \sum a_{xi} x_i$.

Примером линейной функции является отрицание (представляемое как $\neg x = 1 \oplus x$). Дизьюнкция не является линейной функцией: ранее мы видели, что $p \vee q = p \oplus q \oplus pq$. Функция $f(p, q, r) = \neg p \vee q \oplus rq(p \vee r)$ примера 1.2 не является линейной: ранее мы видели, что $f(p, q, r) = 1 \oplus p \oplus pq \oplus qr$. Обозначим $M_{\text{лин}}$ множество всех линейных функций.

Теорема 1.10. $[M_{\text{лин}}] = M_{\text{лин}}$.

Теорема Поста

Рассмотрим теперь, как определить, является ли некоторое произвольное множество двоичных функций базисом.

Теорема 1.11 (теорема Поста). Для того чтобы множество N двоичных функций было базисом, то есть $[N] = B$, необходимо и достаточно, чтобы:

1. N содержало бы по крайней мере одну функцию, не сохраняющую ноль: $N \not\subset M_0$, и
2. N содержало бы по крайней мере одну функцию, не сохраняющую единицу: $N \not\subset M_1$, и
3. N содержало бы по крайней мере одну **немонотонную** функцию: $N \not\subset M_{\text{мн}}$, и
4. N содержало бы по крайней мере одну **несамодвойственную** функцию: $N \not\subset M_{\text{сам}}$, и
5. N содержало бы по крайней мере одну **нелинейную** функцию: $N \not\subset M_{\text{лин}}$.

Необходимость. Нужно доказать, что если N – базис, то все условия выполняются. Это элементарно доказывается от противного: если хотя бы одно условие не выполняется, то $[N] \neq B$. Действительно, пусть N является подмножеством какого-нибудь замкнутого класса из перечисленных пяти. Тогда, очевидно, замыкание N не может включать все функции из B : ни один из этих классов не полон. Например, пусть нарушается условие 2, то есть N не содержит ни одной функции, не сохраняющей единицу (все входящие в N функции сохраняют 1). Но все функции, сохраняющие 1, составляют замкнутый класс: их суперпозицией нельзя построить ни одной функции, не сохраняющей 1 (теорема 1.7). Поскольку B содержит и функции, не сохраняющие 1, то суперпозицией функций из N нельзя построить все функции из B , то есть $[N] \neq B$. Необходимость доказана.

Достаточность. Нужно доказать, что если все условия выполняются, то $[N] = B$. Это доказательство проведем конструктивно по следующей схеме: покажем непосредственно, как из функций множества N построить функции базиса (конъюнктивного или дизъюнктивного).

Схема доказательства (и, соответственно, построения функций) имеет следующий вид (рис. 1.10).

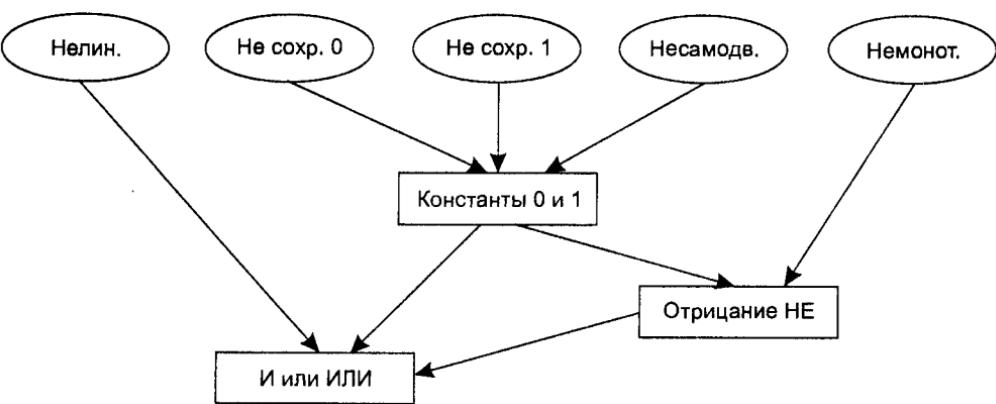


Рис. 1.10. Схема доказательства теоремы Поста

Поясним схему. Пусть в соответствии с условием теоремы Поста множество N имеет функцию, не сохраняющую 0, функцию, не сохраняющую 1, несамодвойственную, немонотонную и нелинейную функции (все эти функции не обязательно различны). В соответствии со схемой, из первых трех функций мы на первом шаге можем построить константы 0 и 1, из констант и немонотонной функции на втором шаге строим функцию НЕ, а на третьем из отрицания, констант и нелинейной функции можем построить конъюнкцию или дизъюнкцию. Переходим к доказательству.

Шаг 1. Пусть F_0 не сохраняет 0 (то есть $F_0(0, \dots, 0) = 1$), F_1 не сохраняет 1 (то есть $F_1(1, \dots, 1) = 0$), а F_n несамодвойственная. Рассмотрим различные варианты.

Пусть F_0 при этом сохраняет единицу (то есть $F_0(1, \dots, 1) = 1$). Тогда при любом x $\phi(x) = F_0(x, \dots, x) = 1$ и $F_1(\phi(x), \dots, \phi(x)) = F_1(1, \dots, 1) = 0$, то есть получили иско-мые функции-константы.

Пусть теперь F_0 не сохраняет единицу, то есть $F_0(1, \dots, 1) = 0$. Но тогда $F_0(x, \dots, x) = \neg x$.

Имея отрицание, с помощью несамодвойственной функции легко получить константу. Действительно, поскольку F_n – функция несамодвойственная, то найдется набор $(\sigma_1, \sigma_2, \dots, \sigma_m)$ такой, что $F_n(\sigma_1, \sigma_2, \dots, \sigma_m) = F_n(\neg\sigma_1, \neg\sigma_2, \dots, \neg\sigma_m) = Const$, где $Const$ либо 0, либо 1. Подставим вместо σ_i в $F_n x$, если $\sigma_i = 1$, и $\neg x$, если $\sigma_i = 0$. Выход этой функции при любом x будет $Const$. Инвертировав $Const$ с помощью отрицания, получим другую константу.

Пример 1.11

Функция f , заданная табл. 1.3, не сохраняет ноль ($f(0,0,0) = 1$) и не сохраняет единицу ($f(1,1,1) = 0$). Следовательно, $f(x,x,x) = \neg x$. Она же является несамодвойственной (существует два противоположных набора значений аргументов, на которых значения f равны: $f(0,0,1) = f(1,1,0) = 1$).

Тогда $f(x,x,\neg x) = 1$, а $\neg f(x,x,\neg x) = 0$ при любом x .

Используя $f(x,x,x) = \neg x$, получаем: $f(x,x,f(x,x,x)) = 1$, а $f(f(x,x,f(x,x,x))) = f(x,x,f(x,x,x)) = 0$ при любом x .

Шаг 2. Пусть F – немонотонная функция. Покажем, как с помощью констант можно построить отрицание. В немонотонной функции всегда существует «единичный интервал» немонотонности, то есть пара соседних наборов $(\sigma_1, \dots, 0, \dots, \sigma_m)$ и $(\sigma_1, \dots, 1, \dots, \sigma_m)$, на которых F меняет значение: $F(\sigma_1, \dots, 0, \dots, \sigma_m) = 1$ и $F(\sigma_1, \dots, 1, \dots, \sigma_m) = 0$. Подставив вместо σ_i – константы, получим из F инвертор.

Пример 1.12

Функция f , заданная табл. 1.3, немонотонна. Один из ее «единичных интервалов» немонотонности – пара наборов $(0, 1, 0)$ и $(0, 1, 1)$, поскольку $f(0,1,0) = 1$ и $f(0,1,1) = 0$. Очевидно поэтому, что $f(0,1,x) = \neg x$. Диаграмма Хассе, представляющая все пары соседних наборов и значения функции f на них, показана на рис. 1.11. Единичные интервалы немонотонности функции f выделены жирными линиями.

Шаг 3. Пусть F – нелинейная функция. Покажем, как с помощью констант и отрицания из нее можно построить конъюнкцию. В нелинейной функции при разложении ее в полином Жегалкина всегда существуют термы, содержащие произведения переменных. Выберем самый короткий такой терм, содержащий не менее двух переменных. Пусть он $K = x_{i1}, x_{i2}, \dots, x_{ik}$. Подставляя в F единицы вместо всех переменных, входящих в K , кроме двух из них, и нули вместо всех переменных F ,

не входящих в терм К, любую нелинейную функцию F можно привести к одной из следующих форм:

xy	$1 \oplus xy$
$x \oplus xy$	$1 \oplus x \oplus xy$
$y \oplus xy$	$1 \oplus y \oplus xy$
$x \oplus y \oplus xy$	$1 \oplus x \oplus y \oplus xy$

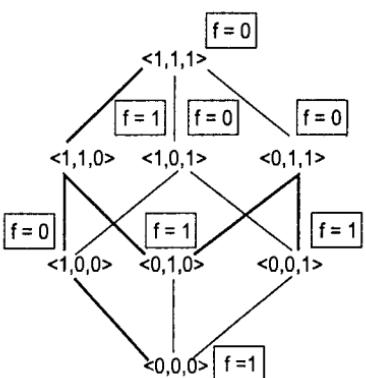


Рис. 1.11. Диаграмма Хассе двоичных наборов

Легко видеть, что с помощью отрицания из всех этих функций можно получить конъюнкцию. Действительно, каждая форма справа является отрицанием соответствующей формы слева, поэтому рассмотрим только четыре левые формы. Первая из них — конъюнкция, последняя — дизъюнкция $x \vee y$. Вторая — конъюнкция $x \wedge \neg y$, третья — конъюнкция $\neg x \wedge y$. Итак, достаточность тоже доказана.

Пример 1.13

Функция f , заданная табл. 1.3, нелинейна: из примера 1.3 $f(p,q,r) = 1 \oplus p \oplus pq \oplus qr$. Выберем один из минимальных термов — пусть это будет pq .

Тогда $f(p,q,0) = 1 \oplus p \oplus pq$. Легко видеть, что это функция $\neg(p \neg q) = \neg p \vee q$. Таким образом, $pq = \neg f(p, \neg q, 0)$, $p \vee q = f(\neg p, q, 0)$. Как результат, только одна эта функция составляет базис, то есть с ее помощью можно построить любую двоичную функцию. Очевидно также, что этот базис минимальный.

Пример 1.14

Таблица 1.8 показывает, принадлежат ли рассмотренным пятью замкнутым классам известные нам функции. Если функция не принадлежит замкнутому классу, в соответствующей позиции ставится знак \checkmark . Для выделения базиса, состоящего из этих функций, нужно найти строки, в совокупности покрывающие все пять столбцов знаком \checkmark . Кроме уже известных нам базисов, из табл. 1.8 можно найти базис Фреге — {импликация, отрицание}, базис Гилберта — {импликация, дизъюнкция, константа 0} и т. д.

Таблица 1.9

	Функции	M_0	M_1	$M_{\text{сам}}$	$M_{\text{мон}}$	$M_{\text{лин}}$
a	0		✓	✓		
b	1	✓		✓		
c	$\neg p$	✓	✓		✓	
d	$p \& q$			✓		✓
e	$p \vee q$			✓		✓
f	$p \Rightarrow q$	✓		✓	✓	✓
g	$p \oplus q$		✓	✓	✓	
h	$p \downarrow q$	✓	✓	✓	✓	✓
i	$p \mid q$	✓	✓	✓	✓	✓
j	$p \equiv q$	✓		✓	✓	

Интересный метод нахождения минимального покрытия, основанный на свойствах логических функций, состоит в том, что строится КНФ логической функции Φ , представляющей все возможные покрытия столбцов, и каждый терм дизъюнктивной нормальной формы этой функции, полученной, например, простыми преобразованиями, дает некоторое покрытие. Для табл. 1.8 функция Φ в форме КНФ имеет вид:

$$\Phi = (b \vee c \vee f \vee h \vee i \vee j) \& (a \vee c \vee g \vee h \vee i) \& (a \vee b \vee d \vee e \vee f \vee g \vee h \vee i \vee j). \\ \& (c \vee f \vee g \vee h \vee i \vee j) \& (d \vee e \vee f \vee h \vee i).$$

Смысл этой функции очевиден. Каждый дизъюнкт определяет, какими строками может быть покрыт один из столбцов. Задача нахождения минимального покрытия и анализ возможных минимальных базисов — то есть раскрытие скобок и получение минимальной ДНФ этой функции (после применения операций неполного склеивания $Kx \vee K\neg x = K \vee Kx \vee K\neg x$ и поглощения $K \vee Kx = K$) остается читателю в качестве упражнения. Каждый терм минимальной ДНФ дает, очевидно, минимальный базис — набор функций, удовлетворяющих всем требованиям теоремы Поста.

Пример 1.15

Рассмотрим, сколько функций может содержать минимальный базис. Очевидно, что самое малое число функций в минимальном базисе — одна, это, например, стрелка Пирса или штрих Шеффера, или функция f примера 1.13. Мы знаем минимальные базисы из двух функций (например, конъюнктивный базис) и из трех функций (базис Жегалкина). А какое максимальное число функций может содержать минимальный базис?

Очевидно, что такой базис содержит не меньше трех и не больше пяти функций. Покажем, что минимальный базис с максимальным числом элементов включает ровно четыре двоичных функции. Действительно, любая функция, не сохраняющая 0, например, не может входить во все остальные четыре замкнутых класса, используемых в теореме Поста, она будет либо немонотонной, либо пессимодвой-

ственной. Пусть это функция 1. Она не сохраняет 0 и несамодвойственна. Другая функция, 0, не сохраняет 1 и также несамодвойственна. Проблема сводится к нахождению пеллинейной, но монотонной функции, а также линейной, но немонотонной функции, причем обе они должны сохранять 0 и 1. Первая — это, например, конъюнкция, вторая — линейная функция $x \oplus y \oplus z$. Таким образом, исключим минимальным базисом является $\{0, 1, x \& y, x \oplus y \oplus z\}$, содержащий четыре двоичных функции, и это число функций максимально возможное в минимальном базисе.

Формы представления булевых функций

К настоящему времени мы знакомы с двумя формами представления булевых функций: таблица истинности и формула (аналитическая запись). Рассмотрим еще две формы представления таких функций.

Семантические деревья

Семантическое дерево — это двоичное дерево, корень которого помечен двоичной функцией от m переменных, из каждого узла идут по два ребра, соответствующих двум значениям очередной переменной, а 2^m листьев помечены соответствующими значениями функции. Удобство этого представления в том, что для многих функций значения у всех листьев некоторых поддеревьев совпадают и построение некоторых ветвей быстро заканчивается, не доходя до самого нижнего уровня.

Бинарные диаграммы решений — БДР

Бинарная диаграмма решений (Binary Decision Diagrams, BDD) [2] — это граф, являющийся модификацией семантического дерева. В БДР узлы с одним и тем же значением функции объединены. Если на каждом уровне БДР все вершины имеют одну и ту же метку (одинаковые переменные), то такая БДР называется упорядоченной (в англоязычной литературе такое представление называется Ordered Binary Decision Diagrams, или сокращенно OBDD). Будем называть такое представление УБДР. Вершины УБДР расположены по уровням, каждому уровню соответствует одна переменная, которая помечает вершины, находящиеся на этом уровне. Из каждой вершины выходят два ребра: одно соответствует нулевому значению соответствующей переменной (будем его изображать нитриховой линией), а другое — единичному значению этой переменной (оно изображается сплошной линией).

На рис. 1.12 показаны все четыре формы представления функции f примера 1.2. Бинарные диаграммы решений используются как компактная форма представления булевой функции. Такое представление полезно во многих случаях, например когда нужно многократно вычислять значения функции при различных наборах значений ее аргументов. Для того чтобы получить значение функции f примера

1.2, например, на языке С, вместо хранения громоздкой таблицы истинности можно вычислить оператор: $f = q?(r?0:1):(p?0:1)$, который построен на основании БДР (см. рис. 1.12). В этом примере использование УБДР позволяет вычислить значение булевой функции, выполнив всего две операции, в то время как при ее вычислении по аналитическому представлению требуется не менее 5 операций.

Формулы	Таблица истинности	Семантическое дерево	Бинарная диаграмма решений																																				
$f(p,q,r) = \neg p \vee q \oplus rq(p \vee r)$ $f(p,q,r) = (\neg p \vee q)(\neg q \vee \neg r)$ $f(p,q,r) = 1 \oplus p \oplus pq \oplus qr$ $f(p,q,r) = \neg p \neg q \vee q \neg r$	<table border="1"> <thead> <tr> <th>p</th><th>q</th><th>r</th><th>f</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	p	q	r	f	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	1	1	1	1	0	$f(p,q,r)$ p=0 $\neg (qr)$ $\neg q$ $\neg r$ $q=0$ $q=1$ $q=0$ $q=1$ $r=0$ $r=1$ $r=0$ $r=1$ 1 0 1 0 1 0 0	
p	q	r	f																																				
0	0	0	1																																				
0	0	1	1																																				
0	1	0	1																																				
0	1	1	0																																				
1	0	0	0																																				
1	0	1	0																																				
1	1	0	1																																				
1	1	1	0																																				

Рис. 1.12. Четыре формы представления двоичной функции

Сложность представления функции с помощью УБДР существенно зависит от порядка переменных. Так, например, УБДР для иного порядка переменных, чем на рис. 1.12, содержит четыре вершины, а не три (рис. 1.13). Интересной проблемой теоретической информатики является нахождение алгоритма, дающего оптимальный порядок переменных булевой функции с точки зрения представления этой функции упорядоченной БДР. В работе [5] представлен один из таких алгоритмов, имеющий, однако, экспоненциальную сложность.

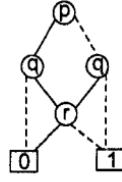


Рис. 1.13. УБДР для функции примера 1.2 с порядком переменных [p,q,r]

Рассмотрим алгоритм, по которому из семантического дерева можно построить УБДР. Алгоритм состоит из трех шагов.

1. Выбрасывание дублирующих значений функции. На рис. 1.14, а в семантическом дереве оставляем только два листа, помеченных соответственно 0 и 1. Результат представлен на рис. 1.14, б.
2. Выбрасывание дублирующих тестовых узлов. Если два различных промежуточных узла в диаграмме являются корнями структурно-идентичных поддиаграмм,

то заменяем их одним эквивалентным. На рис. 1.14, в три промежуточных узла, помеченные переменной z , объединены в один.

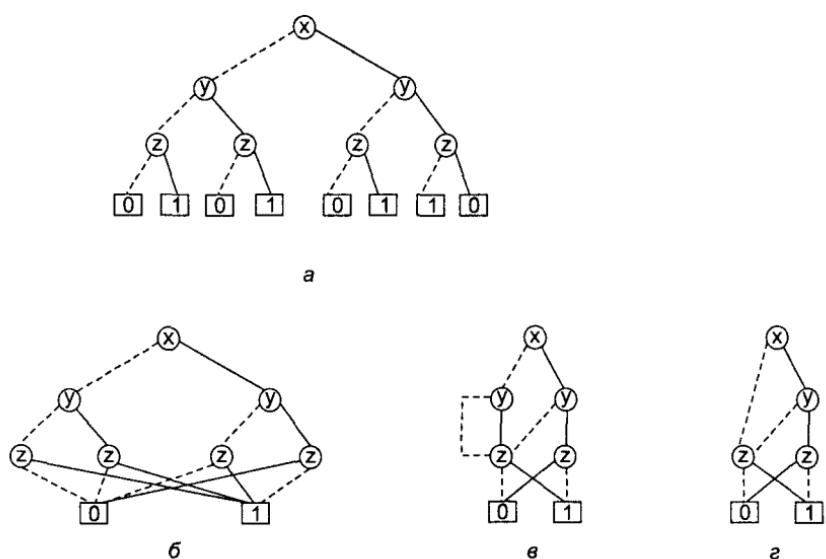


Рис. 1.14. Построение УБДР для булевой функции с порядком переменных $[x,y,z]$ из семантического дерева

3. *Выбрасывание избыточных тестовых узлов.* Если оба исходящих из узла t ребра указывают на один и тот же следующий узел, скажем, u , то узел t можно выбросить, связав все входящие в t ребра с узлом u . На рис. 1.14, г выброшен один из узлов, помеченный y .

Булевы алгебры

Алгеброй называется множество с определенными на нем операциями. Обозначается алгебра обычно парой: (M, Ω) , где M – множество элементов алгебры, а Ω – множество операций алгебры над ее элементами (называющееся сигнатурой). Результаты операций всегда принадлежат M .

Пусть B – алгебра с произвольным множеством M элементов, сигнатура которой содержит две бинарные операции « $+$ » и « \cdot », одну унарную операцию « $'$ » и две нульарные операции (константы) 0 и 1. Такая алгебра B называется *булевой алгеброй* тогда и только тогда, когда для любых $x, y, z \in M$ в ней справедливы следующие законы:

- C1. *Идемпотентность:* $x + x = x$, $x \cdot x = x$;
- C2. *Коммутативность:* $x + y = y + x$, $x \cdot y = y \cdot x$;
- C3. *Ассоциативность:* $x + (y + z) = (x + y) + z$, $x \cdot (y \cdot z) = (x \cdot y) \cdot z$;

- C 4. Поглощение:* $x \cdot (x + y) = x$, $x + (x \cdot y) = x$;
- C 5. Дистрибутивность:* $x + (y \cdot z) = (x + y) \cdot (x + z)$, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$;
- C 6. Универсальные граници:* $x \cdot 0 = 0$; $x + 0 = x$, $x \cdot I = x$, $x + I = I$;
- C 7. Дополнение:* $x \cdot x' = 0$, $x + x' = I$;
- C 8. Законы де Моргана:* $(x \cdot y)' = x' + y'$, $(x + y)' = x' \cdot y'$;
- C 9. Инволютивность:* $(x')' = x$.

Очевидна связь булевых алгебр и двоичных функций. Множество M составляют все двоичные функции, множество операций Ω — дизъюнкция, конъюнкция и отрицание, а нульевые операции — тождественные функции 0 и 1. Но интересно, что и многие другие модели являются булевыми алгебрами и для них можно интерпретировать многие свойства двоичных функций. Например, булеву алгебру представляет собой любое множество всех подмножеств произвольного множества A с операциями объединения, пересечения и дополнения множеств, а нульевые операции — это пустое подмножество и полное множество A .

Множество тождеств $C1—C9$ является избыточным в том смысле, что, например, аксиомы $C1$, $C8$ и $C9$ следуют из остальных шести $C2—C7$. Покажем это для $C1$. Соответствующее утверждение доказано Дедекином.

Теорема 1.12. *Законы идемпотентности $C1$ следуют из законов поглощения $C4$.*

Доказательство. Свойство $C4$: справедливо для любых элементов x и y . Положим в первом тождестве $C4$ $y = x \cdot x$. Тогда это тождество свойства $C4$ перепишется так: $x \cdot [x + (x \cdot x)] = x$. Из второго тождества, в котором положим $y = x$, получим: $x + (x \cdot x) = x$. Левая часть этого тождества как раз совпадает с выражением в квадратных скобках первого тождества. Заменив ее на x , в соответствии с правой частью этого тождества, получим один из законов идемпотентности $x \cdot x = x$. Другой закон идемпотентности доказывается аналогично, если мы заменим в предыдущем рассуждении операцию «+» на «·», и наоборот. Это является частным проявлением так называемого «принципа двойственности» в булевой алгебре.

Теорема 1.13. *Любая общезначимая теорема о булевых алгебрах, в формулировке которой участвуют только операции «+», «·» и «'», остается общезначимой, если в ее формулировке всюду заменить «+» на «·», и наоборот.*

Доказательство этой теоремы весьма просто. Все свойства $C1—C9$ сохраняются при такой замене. Поскольку любые доказательства в булевой алгебре базируются только на этих свойствах, то любое такое доказательство при вышеуказанной замене превращается в доказательство двойственного утверждения.

Интерес к изучению булевых алгебр состоит в том, что как только для некоторой математической структуры — множества и определенных на нем операций — доказано выполнение шести указанных выше законов $C2—C7$, можно быть уверенными, что огромное количество и других свойств булевых алгебр также справедливо для этой структуры. Например, на множестве всех подмножеств произвольного множества справедливы принцип двойственности, законы де Моргана.

Пороговая логика

В 1943 году Уоррен Мак-Каллок и Уолтер Питтс предложили формальную модель нейрона (нервной клетки мозга) как переключающей функции $\{0,1\} \rightarrow \{0,1\}$ в виде логической схемы, имеющей конечное число двоичных входов и один двоичный выход. Каждый вход x_i учитывается в нейроне с некоторым приписанным ему весом w_i . Нейрон возбуждается, если суммарное взвешенное возбуждение его входов не меньше некоторого порога срабатывания θ . Иными словами, выход нейрона равен 1, если $\sum_i w_i * x_i \geq \theta$.

С изменением порога и весов входов логические функции, реализуемые этим нейроном, изменяются. Рассмотрим формальный нейрон с двумя входами $\{x_1, x_2\}$, изображенный на рис. 1.15. Суммарное возбуждение Σ для этой схемы рассчитывается так: $\Sigma = w_1 * x_1 + w_2 * x_2$. Пусть $w_1 = w_2 = 1$; тогда $\Sigma = x_1 + x_2$. Если $\theta = 1$, эта схема реализует дизъюнкцию $x_1 \vee x_2$; при $\theta = 2$ она реализует конъюнкцию $x_1 \wedge x_2$.

Поставим обратную задачу: для заданного нейрона найти такие веса входов и порог его срабатывания, что этот нейрон реализует заданную двоичную функцию, например конъюнкцию $\&$. Очевидно, что для решения задачи для конкретного нейрона рис. 1.14 нужно просто решить систему 4-х неравенств:

для набора $<0, 0>$, $\Sigma = w_1 * x_1 + w_2 * x_2 = 0 < \theta$ (поскольку $0 \& 0 = 0$);

для набора $<0, 1>$, $\Sigma = w_1 * x_1 + w_2 * x_2 = w_2 < \theta$ (поскольку $0 \& 1 = 0$);

для набора $<1, 0>$, $\Sigma = w_1 * x_1 + w_2 * x_2 = w_1 < \theta$ (поскольку $1 \& 0 = 0$);

для набора $<0, 0>$, $\Sigma = w_1 * x_1 + w_2 * x_2 = w_1 + w_2 \geq \theta$ (поскольку $1 \& 1 = 1$).

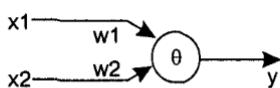


Рис. 1.15. Пример формального нейрона

Очевидно, что этой системе неравенств удовлетворяет решение $\theta = 2$, $w_1 = w_2 = 1$.

Покажем, что в пороговой логике не существует элемента с двумя входами, реализующего функцию сложения по модулю 2. Действительно,

$$0 < \theta \text{ (поскольку } 0 \oplus 0 = 0\text{);}$$

$$w_2 \geq \theta \text{ (поскольку } 0 \oplus 1 = 1\text{);}$$

$$w_1 \geq \theta \text{ (поскольку } 1 \oplus 0 = 1\text{);}$$

$$w_1 + w_2 < \theta \text{ (поскольку } 1 \oplus 1 = 0\text{).}$$

Из второго и третьего неравенства имеем $w_1 + w_2 \geq \theta + \theta$ и, учитывая последнее неравенство, $\theta > w_1 + w_2 \geq \theta + \theta$. Это, однако, противоречит первому неравенству $\theta > 0$.

Контрольные задания

1. Используя следующую лемму о разложении булевой функции:

$$f(x_1, \dots, x_i, \dots, x_m) = (1 \oplus x_i)f(x_1, \dots, 0, \dots, x_m) \oplus x_i f(x_1, \dots, 1, \dots, x_m),$$

доказать, что множество $\{\&, \oplus, 1\}$ – базис. Построить на основе этой леммы представление функции, заданной таблично, в виде полинома Жегалкина.

Указание. Аналогично доказательству леммы 1.1.

2. Проверить свойство ассоциативности:

- а) импликации;
- б) сложения по модулю 2;
- в) стрелки Пирса.

Указание. Использовать таблицы истинности.

3. Проверить свойство дистрибутивности:

- а) импликации и конъюнкции;
- б) импликации и дизъюнкции.

Указание. Использовать таблицы истинности.

4. Выразить функцию $xy \Rightarrow z \vee u$ с помощью стрелки Пирса.

5. Преобразовать равносильно формулы так, чтобы они содержали только указанные операции:

- а) $(x \Rightarrow \neg y) \Rightarrow (y \vee z)$ операции: конъюнкция и отрицание;
- б) $(x \vee y) \& (\neg x \vee \neg y) \Rightarrow (x \Leftrightarrow y)(\neg x \Leftrightarrow \neg y)$ операции: дизъюнкция и отрицание;
- в) $\neg(x \Rightarrow \neg y) \Rightarrow (\neg x \vee z)$ операции: импликация и отрицание.

6. Проверить аналитически равносильность двух формул:

$$(A \vee B)(\neg A \vee C) \text{ и } (A \vee B)(\neg A \vee C)(B \vee C).$$

Указание. В СДНФ, СКНФ и форме полинома Жегалкина булевые функции имеют единственное представление. Поэтому приведение обеих формул к любой из этих форм и последующее их сравнение дает решение задачи.

7. Является ли набор функций $\{\Leftrightarrow, \vee, 0\}$ функционально полным? Реализовать в нем функцию \oplus .

8. Построить логическую схему для вычисления произведения двух двухразрядных двоичных целых чисел со знаком.

9. Построить схему контроля четности в восьмиразрядном двоичном слове (байте). Схема имеет 8 входов и один выход, на котором появляется единичный сигнал в случае нечетного числа единиц во входном слове.

10. Построить функциональную схему двоично-десятичного счетчика по модулю 10.

11. Семисегментная структура светодиодов используется для отображения следующих букв русского алфавита: Б, Г, Е, О, П, Р, С, Ь, Н. Построить схему отображения.

12. Семисегментная структура светодиодов используется для отображения температуры. Кроме десятичных цифр и знака минус, один из четырехразрядных двоичных кодов кодирует символ градуса. Построить схему отображения.
13. Построить формальный нейрон с одним входом, реализующий инвертор.
- Можно ли построить формальный нейрон, реализующий функцию сложения по модулю 2?
 - Можно ли построить с помощью только одного двухходового формального нейрона любые двоичные функции, изменения только значения весов входов и порога?
 - Существует ли «универсальный» двухходовой формальный нейрон с фиксированными весами входов, с помощью которого можно построить любую двоичную функцию?
14. Какие из следующих функций могут быть реализованы одним трехходовым формальным нейроном:
- $f(p, q, r) = \neg p \vee q \oplus qr(p \vee r);$
 - $f(p, q, r) = p \vee \neg q \oplus qr(p \Leftrightarrow \neg r);$
 - $f(p, q, r) = \neg pq \Rightarrow qr(\neg p \vee r).$
- Указание.** Найти (если это возможно) веса входов и порог срабатывания нейрона из очевидных соотношений, которые следуют из того, что трехходовой нейрон реализует заданную функцию.
15. Двоичная логика представляет мир в «черно-белом» свете — элементы ее могут принимать значения только 1 и 0, что соответствует «ДА» и «НЕТ», или «ИСТИНА» и «ЛОЖЬ». Обобщением двоичной логики является многозначная логика, в которой высказывания могут принимать значения, отражающие некоторую долю уверенности, например, в трехзначной логике значения 1, 1/2 и 0 могут соответствовать «ДА», «НАВЕРНОЕ» и «НЕТ», а в четырехзначной значения 1, 2/3, 1/3 и 0 могут соответствовать «ДА», «ВЕРОЯТНО, ЧТО ДА» «ВЕРОЯТНО, ЧТО НЕТ» и «НЕТ». Построить в этих логиках унарную функцию, соответствующую отрицанию, и бинарные функции, соответствующие дизъюнкции и конъюнкции. Выполняются ли в этих логиках законы булевой алгебры?
16. Пусть запись `if p then q else r` представляет двоичную функцию от трех двоичных переменных, которая выдает значение q, если p истинно, и значение r, если p ложно.
- Проверить, эквивалентны ли выражения:
 $\text{if } p \text{ then } (\text{if } q \text{ then } s \text{ else } n) \text{ else } (\text{if } r \text{ then } s \text{ else } n)$
 $\text{и if } (\text{if } p \text{ then } q \text{ else } r) \text{ then } s \text{ else } n.$
 - Составляет ли эта функция базис?
 - Можно ли получить базис, добавив к этой функции `True` и `False`? Выразить в этом базисе отрицание, дизъюнкцию, конъюнкцию, импликацию. Записать эти функции выражениями типа: `if (if p then True else r) then False else q.`

17. Пусть в компьютерной сети по ненадежному каналу требуется передать k нумерованных информационных пакетов длиной d двоичных разрядов каждый. Как можно на приемном конце обеспечить восстановление любого одного потерянного пакета, добавив на передающем конце к k информационным пакетам только один дополнительный $k+1$ -й (избыточный) пакет той же длины d ?

Указание. Используя поразрядную операцию **XOR** над k информационными пакетами на передающем конце, строим $k+1$ -й пакет, также передающийся в канал. Покажите, что любой один пропавший пакет на приемном конце может быть восстановлен с использованием той же поразрядной операции над оставшимися k принятыми правильно пакетами.

18. Следующий фрагмент программы на языке С переписать в эквивалентном виде, используя вместо операции И (`&&`) операцию ИЛИ (`||`):

```
for(i = 0; i < lim &&(c = getchar( ))!= '\n'&&c!= EOF; s[i++] = c);
```

19. Построить схему автоматизации принятия решений судьями в боксерском поединке. Поединок судят трое судей. Судья, засчитывающий очко бойцу А (пропуск удара боксером Б), нажимает на имеющуюся в его распоряжении кнопку А; судья, не засчитывающий результат, кнопку А не нажимаст. Очко засчитывается, если не менее двух судей его засчитали. В этом случае в соответствующем углу должна загореться лампочка.

20. Построить двухходовые пороговые элементы, реализующие конъюнкцию, эквивалентность и импликацию.

ГЛАВА 2 Введение в математическую логику

В главе излагаются начала логики, науки о формализации мышления. Целью главы является демонстрация некоторых связей между продуктивным мышлением человека, порождающим новое знание, и алгоритмическим функционированием компьютеров. В результате изучения материала этой главы читатель должен освоить:

- общее представление о построении и использовании моделей для решения инженерных проблем;
- формально-логические аспекты формулировки теорем и методов их доказательства;
- методы логического вывода для логики высказываний;
- основы логики первого порядка;
- основы логического вывода в логике предикатов первого порядка;
- формальные основы логического программирования.

Формальные модели

Предмет изучения в этой главе — начала математической, или формальной логики. Математическая логика занимается формальными законами построения суждений и доказательств. В логике используется язык, позволяющий описать ситуацию формально, с тем чтобы можно было провести анализ ситуации, то есть строить и формально доказывать утверждения о свойствах ситуации. Логика исследует схемы рассуждений, которые верны в силу одной их формы, независимо от со-

держания. Фактически логика – это множество правил манипулирования формулами, представляющими *формы* рассуждений. Формальная логика игнорирует смысл, содержание предложений естественного языка, для которых формулы логики являются моделями.

Отвлечение от содержания предложений языка в формальной логике есть результат применения операции абстрагирования к рассуждениям естественного языка. Абстрагирование является основным этапом при построении математической модели, оно широко используется в науке для выборочного исследования некоторых аспектов исследуемой проблемы. Современная парадигма научного исследования состоит в том, что формальное изучение любой проблемы начинается с замены реальных объектов их абстрактными представлениями, выбираемыми таким образом, чтобы в этих идеализациях были отражены именно те свойства исходных объектов, которые мы хотим изучать. Цель абстрагирования – выделение тех аспектов, которые существенны для решения проблемы, и игнорирование тех аспектов, которые незначительны, усложняют проблему, делают анализ менее общим или вообще невозможным.

При научном подходе, на уровне рационального исследования, мы имеем дело не с реальной действительностью, а со структурами, абстракциями от материальных объектов. Реальные объекты и ситуации обычно бесконечно сложны, и абстракция применяется для того, чтобы ограничить эту сложность, дать возможность принимать решения. С помощью абстрагирования человек строит формальные модели самых разнообразных по своей природе понятий, процессов и явлений, сущностей реального мира. Такие формальные модели, будучи построенными, далее допускают анализ и преобразование с помощью формальных же средств: абстракции могут быть формально исследованы с точки зрения их свойств (структура, элемент, отношение и т. д.), и при таком анализе исследователь остается в рамках построенной им знаковой системы, абстрагируясь от реальных объектов. Формальные модели позволяют выразить некоторые (существенные для решения конкретной проблемы) свойства объекта в точных терминах математических определений и аксиом так, что затем можно «вывести» характеристики построенной модели, которые объяснят известные и предскажут новые свойства исследуемой реальной сущности. Именно на основе научного подхода к решению инженерных проблем получено бесчетное число впечатляющих результатов в технике, в связи с чем давно укоренилась поговорка «Нет ничего более практического, чем хорошая теория».

Часто при изучении естественных объектов исследователь должен абстрагироваться от несущественных случайных деталей, которые не просто усложняют, но и могут затемнить само явление. Например, при изучении планет удобно думать о них как о материальных точках, имеющих массу и двигающихся по эллиптическим траекториям. Очевидно, что планеты не материальные точки, но при такой абстракции можно понять и предсказать поведение солнечной системы. Конечно, можно построить модели, которые абстрагируются от существенных аспектов реальности. Как результат, выводы, полученные на основе этих моделей, будут неверными.

Процесс конструирования модели не является механическим, он требует интуиции, понимания природы явления и решаемой проблемы.

Получая в результате анализа моделей какие-либо выводы, исследователь пытается применить их к той области реального мира, отображением которой является модель. Поскольку все абстракции неполны и неточны, можно говорить только о приближенном соответствии реальности тех результатов, которые получены исследованием на моделях. Соответствие законов движения, связей и отношений объектов модели элементам реального мира называется *адекватностью*, и степень адекватности определяет, применимы ли такие результаты к конкретной проблеме в реальном мире. Часто адекватность модели определяется рядом условий и ограничений на сущности реального мира, и для того чтобы использовать результаты анализа, полученные на модели, необходимо тщательно проверять эти ограничения и условия (*или обеспечить их выполнение!*).

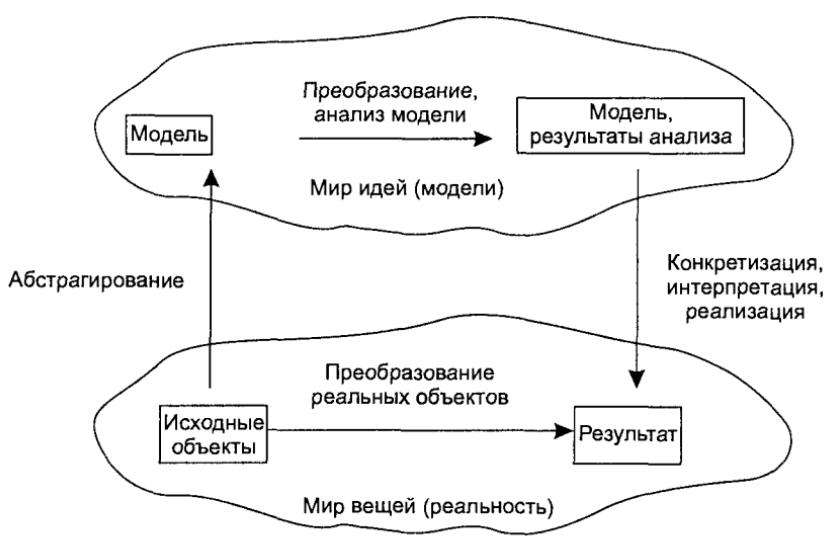


Рис. 2.1. Соотношение моделей и реальных явлений

В предыдущей главе стояла проблема построения конечных функциональных преобразователей в реальном мире. Эта проблема была решена полностью в соответствии с рис. 2.1. Вместо того, чтобы непосредственно начать строить эти преобразователи из реальных объектов (то есть использовать чисто эмпирический подход, «метод тыка»), мы представили эту проблему как задачу построения двоичных функций, реализующих требуемое функциональное отображение (то есть сначала выполнили *абстрагирование*). Мы затем использовали систематический метод решения этой математической задачи (*преобразование модели*). На следующем этапе мы рассмотрели, как произвольные двоичные функции могут быть реализованы из электронных переключателей и транзисторов (фактически мы занимались вопросами *конкретизации, реализации абстрактной модели*). Адекватность ре-

лизации модели логических функций обеспечивается в этой конкретной проблеме поддержанием в определенных границах напряжений и токов в электронных переключателях, определенных временных ограничений на работу переключателей и т. д. Всеми этими средствами достигается коммутативность диаграммы (см. рис. 2.1) в решении этих проблем. (Коммутативность этой диаграммы мы понимаем здесь как возможность получения одного и того же результата из исходных объектов двумя путями: *первый* – это получение искомого результата непосредственным выполнением преобразований объектов в мире вещей, *второй* – это получение результата сначала выполнением операции «абстрагирование» и переходом в мир моделей, затем выполнением операции «преобразование моделей» и, наконец, выполнением операции «конкретизация» с возвращением в мир реальных объектов с реализацией искомого результата на основе анализа и преобразования модели.)

Отметим, что на рис. 2.1 ясно видны различия в задачах фундаментальных и инженерных наук.

- Теоретики занимаются **внутренними** проблемами теорий: разработкой абстрактных моделей, методов их преобразования и анализа. Они интересуются проблемами, лежащими на рис. 2.1 в области «мира идей». Часто они строят формальные структуры впрок, для возможного использования в будущем. В значительно меньшей степени их интересуют вопросы абстрагирования и адекватности моделей, их интерпретация в различных приложениях.
- Перед инженерами стоит задача построения и анализа **реальных** объектов, но вместо решения ее непосредственно, они сначала абстрагируются от всех деталей реальности, которые несущественны для решения поставленной проблемы, выбирают модель, отражающую существенные детали реальности, пользуются разработанными теоретиками методами преобразования и анализа моделей, после чего решают исходную задачу с помощью интерпретации и применения этих теоретических результатов в реальном мире. Для инженеров проблема проверки и обеспечения адекватности используемых моделей является определяющей. Важной является также проблема выбора среди множества формальных моделей такой модели, в рамках которой исходная проблема имеет решение. Чаще всего рамки модели накладывают существенные ограничения на применимость результатов этого подхода в реальном мире. На рис. 2.1 область главных интересов инженеров – это проблемы связи мира реального и мира идей, то есть вопросы адекватного использования существующих моделей для решения конкретных задач практики.

Итак, в современной науке сложилось разумное разделение труда: теоретики обычно не занимаются вопросами приложения своих теоретических конструкций, а инженеры обычно не строят новые формальные модели.

Логика изучает формы мышления и способы их выражения в языке. Формальная математическая логика решает проблемы проверки правильности рассуждений в естественном языке (реальный мир), строя свои модели и правила их преобразования. Для этого логика вводит свои языки – систему формальных обозначений

(формулы) и правила их преобразования. Поэтому логику можно рассматривать как множество правил манипулирования формулами, описывающими утверждения естественного языка. В результате конкретизации (интерпретации) результатов и выводов формальной логики — новых полученных формул, мы получаем новые предложения естественного языка, можем оценить свойства исходных предложений и т. д. Удивительно то, что разработанные еще Аристотелем законы формальной логики для многих приложений адекватны законам мышления человека. Следует ясно понимать, однако, что выводы, полученные с помощью формализма математической логики (так же, как и с помощью любого другого формализма), мы можем использовать в реальной жизни не всегда — только если выполнены ограничения, обеспечивающие адекватность применяемой модели.

Логика высказываний

Из всего разнообразия естественного языка логика высказываний имеет дело только с узким кругом утверждений — повествовательных предложений, которым может быть приписано значение «истина» либо «ложь». Каждому элементарному утверждению в логике высказываний сопоставляется высказывание, обозначающееся буквой. Истинностные значения, которые можно приписать высказываниям, обычно изображаются «И» и «Л», «1» и «0» или *истина* и *ложь*. Кроме простейших высказываний, структура которых не анализируется (они поэтому называются атомами), вводится понятие сложного высказывания или формулы — комбинации более простых высказываний.

Определение 2.1. *Формулы логики высказываний определяются индуктивно над неограниченным множеством атомов (элементарных высказываний) A с помощью двух символов операций (связок) \neg , \Rightarrow и скобок «(» и «)»:*

1. Любой атом из A есть формула.
2. Если P — формула, то $\neg(P)$ тоже формула.
3. Если P, Q — формулы, то $(P \Rightarrow Q)$ — тоже формула.
4. Никаких других формул в логике высказываний нет.

В логике для сокращения формул используются записи:

True для $P \Rightarrow P$, **False** для $\neg\text{True}$ (можно рассматривать **True** и **False** как логические константы).

$(P \vee Q)$ для $((\neg P) \Rightarrow Q)$.

$(P \& Q)$ для $\neg(\neg(P) \vee \neg(Q))$.

$(P \Leftrightarrow Q)$ для $(P \Rightarrow Q) \& (Q \Rightarrow P)$.

Эти соотношения позволяют определить дополнительные логические операции через две базовые операции — импликацию \Rightarrow и отрицание \neg .

Кроме того, обычно вводятся правила приоритетов операций для уменьшения скобок в формулах.

Итак, формулы логики — это просто цепочки (последовательности) символов. Задачей логики является определение смысла, значения формул. В логике каждой формуле можно присвоить одно из двух семантических значений, *истина* или *ложь*, и такое присвоение определяется индуктивно в соответствии с правилами определения 2.1.

Определение 2.2. Каждому атому, входящему в формулу, можно присвоить значение *истина* или *ложь*. Каждая возможная комбинация таких истинностных значений, присвоенных атомам формулы, называется *интерпретацией*. Значения формул для возможных интерпретаций атомов определяются на основе таблиц истинности для основных операций (связок) \neg , \Rightarrow .

Таблица 2.1

P	$\neg P$
ложь	истина
истина	ложь

Таблица 2.2

P	Q	$P \Rightarrow Q$
ложь	ложь	истина
ложь	истина	истина
истина	ложь	ложь
истина	истина	истина

Легко видеть, что логическим константам **True**, **False** присваиваются значения *истина* и *ложь* соответственно.

Очевидно, что логика высказываний и теория булевых функций связаны теснейшим образом: *обе эти модели являются булевыми алгебрами*. Фактически двоичные функции представляют собой функциональную модель логики высказываний, в которой атомные высказывания истолковываются как двоичные переменные. Истинностные значения *истина* и *ложь* в логике соответствуют 1 и 0 в модели двоичных функций, пара основных логических связок — отрицание и импликация — составляют базис (базис Фреже), и все остальные производные логические связки соответствуют известным нам булевым функциям. Для простоты мы дальше будем пользоваться 1 и 0 как истинностными значениями, а также всеми известными нам булевыми функциями в виде логических связок. Естественно пользоваться в логике высказываний результатами и терминологией, известными нам из теории двоичных функций. Логические формулы будем называть логическими функциями (они являются функциями, сопоставляющими каждой интерпретации своих аргументов истинностные значения *истина* или *ложь*).

Определение 2.3. Литерой будем называть атом или его отрицание. Две литеры будем называть противоположными, если одна из них является отрицанием другой.

В формальной логике большую роль играют формулы, принимающие одно и то же значение *истина* на всех интерпретациях. Такие формулы называются *общезначимыми* или *тавтологиями*. Формула, принимающая значение *ложь* на всех интерпретациях своих аргументов, называется *невыполнимой*. Импликации в естественном языке соответствует связки «если ... то ...».

Аппарат логики высказываний весьма похож на формализм теории двоичных функций. Однако при всей похожести теорий задачи, которые они решают, разные. Теория двоичных функций занимается проблемой реализации преобразователей информации. Формальная логика работает с абстракциями, построенными из предложений и рассуждений естественного языка, и интерпретация ее выводов также лежит в области естественного языка.

Логика используется для формализации процессов мышления человека. Многие формальные законы логики кажутся почти тривиальными в естественных рассуждениях. Например, в соответствии с законом двойного отрицания, следующие предложения будут эквивалентными: «Неверно, что вчера не было дождя» и «Вчера был дождь». Однако с помощью логики проясняются также многие задачи, которые, будучи выраженными в естественном языке, представляются очень сложными. Проанализируем несколько примеров.

Пример 2.1

1. Рассмотрим задачу, принадлежащую немецкому логику Э. Шрёдеру: «Один химик высказал предположение, состоящее в том, что соли, которые не окрашены, суть соли, которые не являются органическими соединениями или органические соединения, которые не окрашены. Другой химик оспорил это утверждение. Кто прав?»

Выделим следующие атомные высказывания: «Нечто является солью» (С), «Нечто есть органическое соединение» (О), «Нечто окрашено» (К). Очевидно, что логическая формула, соответствующая высказыванию химика, имеет вид: $C \& \neg K \Rightarrow C \& \neg O \vee O \& \neg K$. Построив таблицу истинности, видим, что эта формула — тавтология, поэтому химик прав, а его оппонент — не прав (по-видимому, он не изучал логику).

2. Проанализируем предложение: «Порядочный человек не может быть вором». Предложение состоит из двух более простых утверждений, которые мы представим в модели атомами: П — «некто есть порядочный человек»; В — «некто является вором».

Логическая формула, представляющая приведенное выше предложение, очевидно, имеет вид: $P \Rightarrow \neg V$. Существует множество формул, равносильных данной, например $\neg P \vee \neg V; \neg(P \& V); V \Rightarrow \neg P$ и т. д. Интерпретируя эти формулы в естественном языке, получим следующие утверждения, которые все эквивалентны между собой:

- $\Pi \Rightarrow \neg B$: «Порядочный человек не может быть вором»;
- $\neg \Pi \vee \neg B$: «Или он не порядочный человек, или же он не вор»;
- $\neg(\Pi \& B)$: «Порядочность и воровство несовместимы»;
- $B \Rightarrow \neg \Pi$: «Если человек вор, то он не является порядочным человеком».

3. Утверждение естественного языка «Если P , то Q , иначе R », широко используемое в программировании, может быть представлено формулой $(P \Rightarrow Q) \& (\neg P \Rightarrow R)$. Это легко проверить по таблице истинности. Таким образом, эквивалентным вышесказанному утверждению является фраза: «Если P , то Q , а если не P , то R ».

4). Утверждение «Симпсон будет признан судом виновным тогда и только тогда, когда все 12 присяжных заседателей признают его виновным» в формульной записи представится так: $B \Leftrightarrow (\Pi_1 \& \Pi_2 \& \dots \& \Pi_{12})$. Равносильная ей формула: $\neg B \Leftrightarrow \neg(\Pi_1 \& \Pi_2 \& \dots \& \Pi_{12})$ по закону де Моргана может быть преобразована в такую: $\neg B \Leftrightarrow (\neg \Pi_1 \vee \neg \Pi_2 \vee \dots \vee \neg \Pi_{12})$. Эту последнюю формулу можно интерпретировать как: «Симпсон **не** будет признан судом виновным тогда и только тогда, когда **хотя бы** один из 12 присяжных заседателей **не признает его вины**», и эта интерпретация, очевидно, эквивалентна исходному утверждению.

5. Рассмотрим задачу Смаллиана.

Друг спросил меня: «Правда ли, что если ты любишь Пэт, то ты также любишь Квинси?». На это я ответил: «Если это правда, то я люблю Пэт, и если я люблю Пэт, то это правда». Кого я люблю в действительности?

Пусть Π и K — атомные высказывания, имеющие смысл: «я люблю Пэт» и «я люблю Квинси». Формула, соответствующая вопросу друга, имеет вид: $\Pi \Rightarrow K$. Надо найти те интерпретации, на которых формула $[(\Pi \Rightarrow K) \Rightarrow \Pi] \& [\Pi \Rightarrow (\Pi \Rightarrow K)]$ принимает истинное значение. По таблице истинности находим только одну такую интерпретацию $\Pi = \text{истина}$, $K = \text{истина}$. Следовательно, автор любит обеих.

С возможностью применения в естественном языке результатов интерпретации формул логики высказывания из всех этих примеров трудно спорить, но на этом пути все же следует быть осторожным. Например, предложения «Он убил, и ему стало страшно» и «Ему стало страшно, и он убил», полностью эквивалентные с точки зрения формальной логики, вовсе не эквивалентны в естественном языке. Другой пример: «Он не мог не закричать» в естественном языке эквивалентно «Он закричал», а с точки зрения логики это неверно. Далее, принятые в формальной логике правило, согласно которому любое высказывание может быть только либо истинным, либо ложным, а третьего не дано, не всегда выполняется в реальной жизни (например, какое истинностное значение приписать высказыванию «Это предложение ложно», которое не может быть ни истинным, ни ложным). Особые трудности вызывает соотношение формальной импликации и причинно-следственных отношений в естественном мире. Например, фраза: «Если прошел дождь, то дорога мокрая» утверждает очевидную связь между фактами A (*прошел дождь*) и B (*дорога мокрая*), что выражается логической формулой $A \Rightarrow B$. Из житейского опыта мы связываем с этим причинно-следственным отношением также и дополнительные факты, например, что дождь может пройти, не оставив следов на земле, и т. д.

нительное знание: «Если дождя не было, то дорога сухая», то есть формула $\neg A \Rightarrow \neg B$ тоже справедлива, ее можно считать следствием исходной формулы. Однако две логические формулы $A \Rightarrow B$ и $\neg A \Rightarrow \neg B$ совершенно различны, из истинности первой формально не следует истинность второй.

Монографии, написанные ведущими специалистами по формальной логике, большое внимание уделяют этой проблеме адекватности — условиям применимости выводов и результатов формальной логики в естественном языке (см., например, [32]). Результаты формальной логики могут быть применены к высказываниям естественного языка, только если выполняются ее постулаты, в частности, высказывания являются только либо истинными, либо ложными («черно-белый взгляд на мир»), они не включают никаких уровней уверенности-неуверенности и возможности, все причинно-следственные отношения между фактами явно выражены, истинностные значения высказываний статичны, не меняются во времени и т. д. Представляемая здесь ветвь логики формализует лишь самую простую часть тех закономерностей, которым подчиняется мышление и язык. Однако даже эта часть является удивительной и впечатляющей по своим результатам. Существуют различные интересные расширения классической логики, например *многозначная логика*, формализующая конкретный набор степеней уверенности в истинности высказываний, *нечеткая логика*, в которой можно оперировать с оценками степени уверенности (вероятностями) в истинности высказываний, *tempоральная логика*, позволяющая формулировать высказываниями о свойствах систем, проходящих изменяющиеся последовательные стадии и т. д. Эти логики, однако, выходят за рамки нашего рассмотрения.

Формулировка и доказательство теорем

Импликация и эквивалентность, или формулы $P \Rightarrow Q$ и $P \Leftrightarrow Q$, имеют важнейшее значение в формулировках и доказательствах теорем. Рассмотрим, как соотносятся с этими формулами различные связки естественного языка, часто встречающиеся при формулировке теорем.

Фраза естественного языка	Формула
Если P , то Q	$P \Rightarrow Q$
Для P необходимо Q	$P \Rightarrow Q$
Для P достаточно Q	$Q \Rightarrow P$
P , если Q	$Q \Rightarrow P$
Необходимым условием P является Q	$P \Rightarrow Q$
Достаточным условием P является Q	$Q \Rightarrow P$
P , если и только если Q	$P \Leftrightarrow Q$
Для P необходимо и достаточно Q	$P \Leftrightarrow Q$
P тогда и только тогда, когда Q	$P \Leftrightarrow Q$

Эту таблицу запомнить трудно, но ее и не надо запоминать, тем более что существует много и других похожих формулировок теорем. Полезно придумать два высказывания с совершенно ясной логической связью, например: (M) «Идет дождь» и (N) «На небе тучи». Ясно, что $M \Rightarrow N$, но не наоборот. Другой подобный пример: (A) «Треугольник равносторонний» и (B) «В треугольнике углы при основании равны». Очевидно, что $A \Rightarrow B$, но не обратно. Пусть теперь требуется доказать теорему: «Р только тогда, когда Q». Попробуем заменить Р и Q высказываниями A и B. Очевидно, что Р может быть ассоциировано только с A, а Q – только с B: «Треугольник равносторонний только тогда, когда треугольник равнобедренный». Следовательно, формулировка «Р только тогда, когда Q» равносильна «Если Р, то Q» или $P \Rightarrow Q$. Очевидно, что в импликации $A \Rightarrow B$ утверждение A сильнее, чем утверждение B, а B слабее, чем A. Здесь понятия *сильнее* и *слабее* относятся к требованиям, налагаемым на элементы универсума – множества реальных объектов, о которых идет речь.

Для доказательства $P \Rightarrow Q$ можно использовать метод доказательства от противного: «Предположим, что Q не выполняется. Докажем, что тогда P не выполняется». Этот метод можно использовать, потому что формулы $P \Rightarrow Q$ и $\neg Q \Rightarrow \neg P$ равносильны. Формула $\neg Q \Rightarrow \neg P$ называется **контрапозицией** формулы $P \Rightarrow Q$. Таким образом, метод доказательства от противного имеет четкое обоснование в логике высказываний, состоящее в том, что формула и ее контрапозиция равносильны.

Доказательства необходимости и достаточности $P \Leftrightarrow Q$ проводятся обычно раздельно: сначала доказывается необходимость («для Р необходимо Q», $P \Rightarrow Q$), а потом – достаточность («для Р достаточно Q», $Q \Rightarrow P$). Такое раздельное доказательство можно использовать, потому что формула $(P \Rightarrow Q) \& (Q \Rightarrow P)$ равносильна $P \Leftrightarrow Q$ (как это проверить?). Аналогично можно использовать равносильность $(P \Rightarrow Q) \& (R \Rightarrow Q)$ формуле $(P \vee R \Rightarrow Q)$. Для доказательства теорем с более сложными схемами формулировок следует искать такие более простые формулы, конъюнкция которых была бы равносильна исходной формуле. Конъюнкция здесь используется потому, что последовательное доказательство истинности нескольких формул равносильно доказательству истинности конъюнкции этих формул.

Пример 2.2

Докажем теорему: «Необходимым условием того, чтобы сепульки были не хроничны или не бифуркальны, является то, что сепульки не могут быть хроничны и бифуркальны одновременно». Эта теорема имеет формальную схему $(\neg X \vee \neg B) \Rightarrow \neg(X \& B)$. Легко видеть, что левая и правая части импликации эквивалентны, следовательно, все это утверждение – тавтология (это можно проверить построением таблицы истинности). Таким образом, для доказательства этой теоремы не нужно ничего знать о сепульках и об их свойствах.

Для доказательства теоремы: «Нильпотентный идеал является модулярным и радикальным» необходимо доказать две более простые теоремы: «Нильпотентный

идеал является модулярным» и: «Нильпотентный идеал является радикальным», потому что формула $N \Rightarrow M \& R$ равносильна конъюнкции $(N \Rightarrow M) \& (N \Rightarrow R)$. Каждую из этих более простых теорем можно доказать и от противного. Обе эти теоремы, конечно, должны доказываться на основе свойств идеалов, в то время как сведение исходной теоремы к двум более простым было определено только на основе понимания ее структуры, без понимания смысла, семантики понятий идеала, нильпотентности и т. п.

Пусть теперь нужно доказать более сильную теорему: «Идеал нильпотент тогда и только тогда, когда он является модулярным и радикальным». Формальное выражение этого утверждения $N \Leftrightarrow M \& R$. Для доказательства всей теоремы достаточно доказать три более простых теоремы (почему?):

- «Если идеал не является модулярным, то он не может быть нильпотентным» ($\neg M \Rightarrow \neg N$).
- «Если идеал не является радикальным, то он не может быть нильпотентным» ($\neg R \Rightarrow \neg N$).
- «Если идеал и модулярный, и радикальный, то он нильпотентный» ($M \& R \Rightarrow N$).

Пример 2.3

Следующий пример – теорема Поста из главы 1. Схема ее формулировки такова: «Для того чтобы P , необходимо и достаточно, чтобы и $\neg Q_0$ и $\neg Q_i$ и $\neg Q_{\text{сам}}$ и $\neg Q_{\text{мон}}$ и $\neg Q_{\text{лин}}$ », где P – утверждение: «множество функций N является базисом», а Q_i – утверждение: «множество функций N принадлежит замкнутому классу M_i ». Иными словами, нужно доказать $P \Leftrightarrow (\neg Q_0 \& \neg Q_i \& \neg Q_{\text{сам}} \& \neg Q_{\text{мон}} \& \neg Q_{\text{лин}})$.

Представляем эту формулу как конъюнкцию:

$$\begin{aligned} \text{необходимость } P &\Rightarrow (\neg Q_0 \& \neg Q_i \& \neg Q_{\text{сам}} \& \neg Q_{\text{мон}} \& \neg Q_{\text{лин}}) \\ \text{и достаточность: } (\neg Q_0 \& \neg Q_i \& \neg Q_{\text{сам}} \& \neg Q_{\text{мон}} \& \neg Q_{\text{лин}}) &\Rightarrow P. \end{aligned}$$

Вместо первой формулы мы доказывали ее контрапозицию, то есть

$$\neg(\neg Q_0 \& \neg Q_i \& \neg Q_{\text{сам}} \& \neg Q_{\text{мон}} \& \neg Q_{\text{лин}}) \Rightarrow \neg P,$$

что равносильно конъюнкции пяти формул:

$$(Q_0 \Rightarrow \neg P) \& (Q_i \Rightarrow \neg P) \& (Q_{\text{сам}} \Rightarrow \neg P) \& (Q_{\text{мон}} \Rightarrow \neg P) \& (Q_{\text{лин}} \Rightarrow \neg P).$$

Таким образом, доказательство теоремы Поста состоит из пяти независимых доказательств того, что если множество N булевых функций принадлежит одному из пяти замкнутых классов (Q_i), то оно не является базисом (OP) (**необходимость**), и одного доказательства: «если N не принадлежит ни одному из пяти замкнутых классов, то оно является базисом» (**достаточность**). Эту последнюю формулу упростить уже нельзя, достаточность теоремы Поста доказывается конструктивно: из функций множества N непосредственно строятся функции конъюнктивного базиса, что нами и проделано в главе 1.

Проверка доказательных рассуждений

Предметом анализа здесь будут формальные законы построения умозаключений естественного языка. Фактически эта область логики формализует умение человека делать выводы из фактов.

Пример 2.4

Рассмотрим рассуждение: «Если бы он не сказал ей, она бы и не узнала. А не спроси она его, он и не сказал бы ей. Но она узнала. Следовательно, она спросила» [11]. Это сложное рассуждение представляет собой конъюнкцию трех утверждений, за которыми следует некоторый вывод. Представим рассуждение структурно в табл. 2.3.

Таблица 2.3

F1:	Если бы он не сказал ей, она бы и не узнала
F2:	А не спроси она его, он и не сказал бы ей
F3:	Но она узнала
∴ R:	Следовательно, она спросила

В предложениях можно выделить *атомы (элементарные высказывания)*: «он сказал ей» (Ск), «она узнала» (У), «она спросила» (Сп) и, кроме того, отдельные утверждения F1, F2, F3 и вывод R. Схема, модель этого рассуждения приведена в табл. 2.4. Кроме логических связок отрицания и импликации в схему входит символ «∴», заменяющий **«следовательно»**. Он показывает, что в рассуждении утверждается, что формула, стоящая за ним, является **следствием** приведенных ранее утверждений.

Таблица 2.4

F1:	$\neg \text{Ск} \Rightarrow \neg \text{У}$
F2:	$\neg \text{Сп} \Rightarrow \neg \text{Ск}$
F3:	У
∴ R:	∴ Сп

Одной из главных проблем логики, сформулированной еще великим Аристотелем, является разработка методов определения того, **является ли заключительное утверждение (R) истинным при условии истинности приведенных фактов единственно в силу формальной структуры рассуждения**, а не исходя из **смысла** утверждения R и фактов F1, F2, F3. Формальная логика как раз позволяет определить это, то есть логика позволяет определить, является ли утверждение R **логическим следствием** фактов F1, F2, F3.

Пример 2.5

Рассмотрим другое рассуждение: «В хоккей играют настоящие мужчины. Трус не играет в хоккей. Я в хоккей не играю. Значит, я трус(?)» Элементарными высказываниями здесь будут: «я играю в хоккей (Х)», «я — настоящий мужчина, не трус (М)». Схема этого рассуждения приведена в таблице 2.5.

Таблица 2.5

F1:	$X \Rightarrow M$
F2:	$\neg M \Rightarrow \neg X$
F3:	$\neg X$
$\therefore R:$	$\therefore \neg M$

Если согласиться с посылками рассуждения, то следует ли из посылок его вывод?

В формальной логике разработано несколько методов проверки правильности рассуждений, то есть проверки того, является ли некоторое утверждение логическим следствием других утверждений. Логическое следствие можно считать **новым знанием, новой информацией, полученной из уже известных фактов**.

Силлогизмы

Силлогизмы — это правильные схемы рассуждений. Впервые силлогизмы исследовались еще Аристотелем и имели фиксированную форму, описываемую предикатами. В более общем смысле силлогизмом называется схема рассуждений из некоторого множества таких схем, в которых заключение всегда верно в силу именно **формы** рассуждения, а не его содержания. Рассмотрим следующие силлогизмы.

$$\frac{\begin{array}{c} P \Rightarrow Q \\ P \end{array}}{\therefore Q}$$

модус поненс (способ спуска)

$$\frac{\begin{array}{c} P \Rightarrow Q \\ \neg Q \end{array}}{\therefore \neg P}$$

*модус толленс
(доказательство от противного)*

$$\frac{\begin{array}{c} P \vee Q \\ \neg P \end{array}}{\therefore Q}$$

дизъюнктивный силлогизм

$$\frac{\begin{array}{c} P \Rightarrow Q \\ Q \Rightarrow R \end{array}}{\therefore P \Rightarrow R}$$

*гипотетический силлогизм
(транзитивность импликации)*

$$\frac{\begin{array}{c} P \oplus Q \\ P \end{array}}{\therefore \neg Q}$$

разделительный силлогизм

$$\frac{\begin{array}{c} P \Rightarrow Q \\ R \Rightarrow Q \\ P \vee R \end{array}}{\therefore Q}$$

простая конструктивная дилемма

$$\begin{array}{l} P \Rightarrow Q \\ R \Rightarrow S \\ P \vee R \\ \hline \therefore Q \vee S \end{array}$$

сложная конструктивная дилемма

$$\begin{array}{l} P \Rightarrow Q \\ P \Rightarrow S \\ \neg Q \vee \neg S \\ \hline \therefore \neg P \end{array}$$

простая деструктивная дилемма

$$\begin{array}{l} P \Rightarrow Q \\ R \Rightarrow S \\ \neg Q \vee \neg S \\ \hline \therefore \neg P \vee \neg R \end{array}$$

сложная деструктивная дилемма

Рассуждения, построенные по схеме силлогизмов, правильны именно в силу своей структуры. Использование в рассуждениях таких стандартных схем гарантирует правильность полученных выводов. Однако, существуют (и их бесконечное число!) рассуждения в естественном языке, которые построены не в соответствии с приведенными выше силлогизмами. Конечно, это не означает, что такие рассуждения неправильны. Например, схема рассуждения «Узнала — спросила» не совпадает ни с одной из схем из приведенного выше списка силлогизмов, но интуитивно мы чувствуем, что оно правильно.

Весь последующий материал данной главы посвящен тому, как проверить правильность произвольного рассуждения. Если рассуждение построено не в соответствии со схемой силлогизма, можно попытаться последовательным применением нескольких силлогизмов доказать справедливость заключения. Попробуем это сделать для рассуждения «Узнала — спросила». Из пары утверждений F2 и F1 этого рассуждения (см. табл. 2.4) в соответствии с «гипотетическим силлогизмом» можем заключить, что утверждение « $\text{Сп} \Rightarrow \neg \text{У}$ » истинно. Назовем его R1:

$$\begin{array}{ll} \hline \text{F2:} & \neg \text{Сп} \Rightarrow \neg \text{Ск} \\ \text{F1:} & \neg \text{Ск} \Rightarrow \neg \text{У} \\ \hline \text{R1:} & \neg \text{Сп} \Rightarrow \neg \text{У} \end{array}$$

Далее, пара утверждений R1 и F3 в соответствии с *модус толленс* гарантирует истинность заключения R этого рассуждения.

Однако, подбор правильных образцов рассуждений — силлогизмов — не является систематическим методом, позволяющим всегда ответить на вопрос, является ли схема данного рассуждения правильной, если она не совпадает ни с одним из указанных силлогизмов. Например, в рассуждении «о хоккее» справедливо ли утверждать, что не играющий в хоккей — трус? Далее мы рассмотрим систематические методы проверки правильности рассуждений, применимые к **любым** схемам рассуждений.

Логическое следствие

Определение 2.4. Формула R называется логическим следствием формулы F (или: R логически следует из F) тогда и только тогда, когда для всякой интерпретации, на которой формула F истинна, R тоже истинна.

Очевидно, что сама формула является логическим следствием самой себя, причем наиболее сильным следствием. Наиболее слабым следствием любой формулы является тавтология — тождественно истинная функция. Например, человек, выводящий из фактов только тривиальную истину, всегда прав. С другой стороны, из ложных фактов можно вывести любое утверждение. Например, знаменитый пример Гильберта «Если $2 \times 2 = 5$, то Луна сделана из зеленого сыра» с точки зрения логики является совершенно правильным умозаключением.

Теорема 2.1. Логическим следствием формулы F является False тогда и только тогда, когда F невыполнима.

Доказательство этой теоремы легко провести на основании определения 2.3.

Очевидно, что если формула F невыполнима, то кроме False из нее можно получить любые следствия.

Определение 2.5. Формула R называется логическим следствием формул F_1, F_2, \dots, F_k (или: R логически следует из F_1, F_2, \dots, F_k) тогда и только тогда, когда для всякой интерпретации, на которой формула $F_1 \& F_2 \& \dots \& F_k$ истинна, R тоже истинна.

Из определения 2.4 видно, что наиболее сильным логическим следствием нескольких фактов является просто конъюнкция этих фактов.

Пример 2.6

Инспектора Крейга из Скотланд-Ярда направили для проверки лечебницы для умалишенных [34]. Каждый из обитателей лечебницы — врач либо пациент — мог быть либо здоров, либо лишен рассудка. Если он был здоров, то говорил правду, если лишен рассудка, то только лгал.

В лечебнице Крейг побеседовал с двумя обитателями, Джонсом и Смитом. Джонс сказал, что Смит — один из врачей больницы, а Смит сказал, что Джонс — пациент. Поразмыслив, Крейг догадался, что в клинике есть или врачи, лишенные рассудка, или пациенты, которые нормальны (очевидно, и то и другое следует пресекать). Как он догадался об этом? В [34] приведено долгое рассуждение, обосновывающее вывод инспектора. Однако вывод этот можно получить чисто механически. Действительно, обозначим:

Ддж — Джонс доктор (следовательно, \neg Ддж — Джонс пациент);

Ндж — Джонс нормален (следовательно, \neg Ндж — Джонс болен);

Дсм — Смит доктор (следовательно, \neg Дсм — Смит пациент);

Нсм — Смит нормален (следовательно, \neg Нсм — Смит болен).

Информация, полученная Крейгом, сводится к четырем фактам:

1. $\neg \text{Ндж} \Rightarrow \text{Дсм}$ (если Джонс нормален, то он говорит правду, и Смит — доктор);
 2. $\neg \text{Ндж} \Rightarrow \neg \text{Дсм}$ (если Джонс болен, то он говорит неправду, и Смит — не доктор);
 3. $\neg \text{Нсм} \Rightarrow \neg \text{Ддж}$ (если Смит нормален, то Джонс — действительно пациент, то есть не доктор);
 4. $\neg \text{Нсм} \Rightarrow \text{Ддж}$ (если Смит не нормален, то он лжет, и Джонс — не пациент, то есть он доктор).

Какой вывод можно сделать из этих фактов? Для получения наиболее сильного следствия построим их конъюнкцию:

$$\begin{aligned}
 & (\text{Ндж} \Rightarrow \text{Дсм})(\neg \text{Ндж} \Rightarrow \neg \text{Дсм})(\text{Нсм} \Rightarrow \neg \text{Ддж})(\neg \text{Нсм} \Rightarrow \text{Ддж}) = \\
 & = (\text{Ндж} \vee \text{Дсм})(\text{Ндж} \vee \neg \text{Дсм})(\neg \text{Нсм} \vee \neg \text{Ддж})(\text{Нсм} \vee \text{Ддж}) = \\
 & = (\neg \text{Ндж} \neg \text{Дсм} \vee \text{Дсм} \text{Ндж})(\neg \text{Нсм} \text{Ддж} \vee \neg \text{Ддж} \text{Нсм}) = \\
 & = \text{Ндж} \neg \text{Дсм} \neg \text{Нсм} \text{Ддж} \vee \neg \text{Ндж} \neg \text{Дсм} \neg \text{Ддж} \text{Нсм} \vee \text{Дсм} \text{Ндж} \neg \text{Нсм} \text{Ддж} \vee \\
 & \vee \text{Дсм} \text{Ндж} \neg \text{Ддж} \text{Нсм}.
 \end{aligned}$$

Первый терм говорит о том, что Джонс доктор, по ненормальный, последний — что он пациент, по нормален. Второй и третий термы говорят подобное о Смите. Их дизьюнкция говорит о том, что по крайней мере одна из этих возможностей реализуется. Каждая из них аномальна. Поэтому у инспектора Крейга есть все основания для расследования.

Пусть теперь мы имеем произвольную схему рассуждений. Определение 2.4 дает нам возможность систематической проверки правильности любой схемы рассуждений с высказываниями, например по таблице истинности. Построим таблицы истинности формул $F_1 \& F_2 \& \dots \& F_k$ и R для рассуждения «Узнала — спросила» примера 2.1 (табл. 2.6).

Таблица 2.6

Формула $F1 \& F2 \& F3$ истинна в табл. 2.6 только на последней интерпретации аргументов, и на этой же интерпретации R тоже истинна. Следовательно, схема для рассуждения «Узнала — спросила» правильна, вывод с необходимостью следует из истинности фактов. Заключительное утверждение здесь является новым знанием, оно не проверялось в реальной жизни специально: проверялись другие, а это является логическим следствием указанных фактов. Для того, чтобы сделать вывод о его истинности, достаточно только убедиться в истинности других высказываний (фактов): $F1, F2, F3$. Конечно, формальная логика не может гарантировать, что эти высказывания — посылки — истинны. Определение истинности посылок выходит за рамки логики, но уж если их истинность установлена, то истинность результата R можно не устанавливать: она будет следовать автоматически.

Проанализируем рассуждение «о хоккее» примера 2.2. Построим таблицу истинности:

Таблица 2.7

X	M	$F1 = X \Rightarrow M$	$F2 = \neg M \Rightarrow \neg X$	$F3 = \neg X$	$F1 \& F2 \& F3$	$R = \neg M$
f	f	t	t	t	t	t
f	t	t	t	t	t	f
t	f	f	f	f	f	t
t	t	t	t	f	f	f

Таблица 2.7 показывает, что в этом рассуждении вывод не является логическим следствием утверждений $F1, F2, F3$. Интерпретация (f, t) является контрпримером: на ней все факты истинны, а утверждение R — ложно. Это означает, что только из фактов $F1, F2$ и $F3$ нельзя установить, является ли говорящий трусом. Решение этой проблемы требует привлечения дополнительных фактов. Интересно, что из этих фактов нельзя также установить и противоположное — является ли говорящий мужчиной.

Хотя две строки этой популярной песни: «В хоккей играют настоящие мужчины» и «Трус не играет в хоккей» звучат по-разному, с логической точки зрения они не различаются: соответствующие логические функции $X \Rightarrow M$ и $\neg M \Rightarrow \neg X$ равносильны, одна является контрапозицией другой.

Используя таблицы истинности, легко проверить, что каждый приведенный выше силлогизм является «правильной» схемой рассуждений. Читатель может на основании этого подхода с построением таблиц истинности легко придумать свои правильные схемы рассуждений.

При всей ясности метода, таблицы истинности неудобны: при большом числе аргументов в рассуждении эти таблицы очень громоздкие. Поэтому в логике были разработаны другие методы анализа рассуждений, основывающиеся на определении понятия логического следствия.

Основная теорема логического вывода

Следующая теорема определяет условия, при которых можно проверить правильность схемы рассуждений без использования таблиц истинности. Именно на этой теореме основаны все методы логического вывода как в логике высказываний, так и в логике предикатов. Поэтому ее можно назвать *основной теоремой теории логического вывода*.

Теорема 2.2. *Формула R является логическим следствием формул F_1, F_2, \dots, F_k тогда и только тогда, когда формула $F_1 F_2 \dots F_k \Rightarrow R$ общезначима.*

Доказательство. (*Необходимость*). Предположим, что R является логическим следствием формулы $F_1 F_2 \dots F_k$, и докажем при этом предположении, что $F_1 F_2 \dots F_k \Rightarrow R$ общезначима. Пусть I есть произвольная интерпретация атомов. Если все F_1, F_2, \dots, F_k истинны на I, то $F_1 F_2 \dots F_k$ истинна на I и по определению логического следствия R истинна на I, и, следовательно, на этой интерпретации $F_1 F_2 \dots F_k \Rightarrow R$ истинна. Если же хотя бы одна F_i ложна на I, то на этой интерпретации $F_1 F_2 \dots F_k$ ложна, но $F_1 F_2 \dots F_k \Rightarrow R$ также истинна независимо от R в силу определения импликации. Следовательно, на I формула $F_1 F_2 \dots F_k \Rightarrow R$ истинна. В силу произвольности интерпретации I этот вывод справедлив для всякой интерпретации, поэтому формула $F_1 F_2 \dots F_k \Rightarrow R$ общезначима.

(*Достаточность*). Предположим, что формула $F_1 F_2 \dots F_k \Rightarrow R$ общезначима. Тогда для всякой интерпретации, на которой все F_i истинны, формула $F_1 F_2 \dots F_k$ тоже истинна, и поскольку $F_1 F_2 \dots F_k \Rightarrow R$ общезначима, на этой интерпретации R тоже истинна в силу определения импликации.

Теорема 2.3. Формула R является логическим следствием формул F_1, F_2, \dots, F_k тогда и только тогда, когда формула $F_1 F_2 \dots F_k \neg R$ невыполнима.

Доказательство. По теореме 2.2 формула R является логическим следствием формул F_1, F_2, \dots, F_k тогда и только тогда, когда формула $F_1 F_2 \dots F_k \Rightarrow R$ общезначима или, что то же, отрицание формулы $F_1 F_2 \dots F_k \Rightarrow R$ невыполнимо. Но по закону де-Моргана $\neg(F_1 F_2 \dots F_k \Rightarrow R)$ равносильно $F_1 F_2 \dots F_k \neg R$. Теорема доказана.

На основании теорем 2.2 и 2.3 вопрос о правильности схемы рассуждения сводится к проверке общезначимости либо невыполнимости некоторой формулы. Эта проверка может быть выполнена множеством различных способов.

Приведение к нормальным формам

Как, анализируя формулу в нормальной форме, можно сделать вывод о ее общезначимости или невыполнимости? Возьмем ДНФ, то есть представление формулы в виде дизъюнкции конъюнкций $K_1 \vee K_2 \vee \dots \vee K_n$. Для того чтобы сделать вывод о ее **общезначимости**, нужно анализировать всю формулу целиком: каждая конъюнкция общезначимой формулы может быть истинной только на нескольких интерпретациях. В то же время вывод о **невыполнимости** дизъюнкции конъюнкций можно сделать легко: **каждая** конъюнкция ДНФ невыполнимой функции должна быть невыполнима, а это очень легко проверить: конъюнкция литер невы-

полнима тогда и только тогда, когда она содержит хотя бы одну пару противоположных литер. Полностью аналогично можно использовать представление в виде КНФ, но для определения общезначимости функции. Очевидными следствиями предыдущих теорем являются:

Теорема 2.4. Для того, чтобы формула R была логическим следствием формул F_1, F_2, \dots, F_k , необходимо и достаточно, чтобы каждый конъюнкт дизъюнктивной нормальной формы представления формулы $F_1 F_2 \dots F_k \Rightarrow R$ содержал пару противоположных литер.

Теорема 2.5. Для того, чтобы формула R была логическим следствием формул F_1, F_2, \dots, F_k , необходимо и достаточно, чтобы каждый дизъюнкт конъюнктивной нормальной формы представления формулы $F_1 F_2 \dots F_k \Rightarrow R$ содержал пару противоположных литер.

Пример 2.7

Докажем правильность схемы рассуждения «Узнала — спросила» примера 2.4 приведением к ДНФ:

$$\begin{aligned} & (\neg C_k \Rightarrow \neg Y)(\neg C_p \Rightarrow \neg C_k) Y \neg C_p = \\ & = (C_k \vee \neg Y)(C_p \vee \neg C_k) Y \neg C_p = (C_k C_p Y \neg C_p) \vee (C_k \neg C_k Y \neg C_p) \vee \\ & \vee (\neg Y C_p Y \neg C_p) \vee (\neg Y \neg C_k Y \neg C_p). \end{aligned}$$

Поскольку каждый конъюнкт содержит пару противоположных литер, эта формула невыполнима и, следовательно, схема рассуждения «Узнала — спросила» праильна. Заметим, что это доказательство много проще построения и анализа таблиц истинности (см. табл. 2.6).

Метод резолюции

Этот метод использует тот факт, что если некоторая формула является невыполнимой, то наиболее сильное следствие этой формулы — константа **False**. Предложенный в 1965 году Дж. Робинсоном [11] метод резолюции позволяет получить максимально сильное следствие множества формул с помощью систематической процедуры последовательного построения логических следствий формулы, называемых **резольвентами**. Иными словами, метод резолюции обладает свойством полноты: для формулы Φ следствие **False** обязательно будет получено, если Φ — невыполнима.

Теорема 2.6. Пусть B — логическое следствие формулы A . Тогда $A \& B \equiv A$.

Доказательство теоремы легко проводится на основании определения понятия логического следствия. Действительно, пусть условие теоремы выполнено. Тогда в соответствии с теоремой 2.1 $A \Rightarrow B \equiv \text{True}$.

Отсюда

$$A \equiv A \& \text{True} \equiv A \& (A \Rightarrow B) \equiv A(\neg A \vee B) \equiv A \& \neg A \vee A \& B \equiv \text{False} \vee A \& B \equiv A \& B.$$

Определение 2.6. Резольвентой двух дизъюнктов $D_1 \vee L$ и $D_2 \vee \neg L$ называется дизъюнкт $D_1 \vee D_2$.

Теорема 2.7. Резольвента является логическим следствием порождающих ее дизъюнктов.

Доказательство. Пусть $D1 \vee L$ и $D2 \vee \neg L$ — два дизъюнкта. Тогда $D1 \vee D2$ — их резольвента. Легко показать, что формула $(D1 \vee L) \& (D2 \vee \neg L) \Rightarrow (D1 \vee D2)$ общеизначима. По теореме 2.1 отсюда следует заключение теоремы.

Метод резолюций доказательства невыполнимости формулы Φ состоит в том, что формула Φ представляется в КНФ и к ней конъюнктивно присоединяются все возможные резольвенты ее дизъюнктов и получаемых в процессе доказательства резольвент. Полнота метода резолюций состоит в том, что он *гарантирует* получение для формулы Φ следствия **False**, если Φ невыполнима. Если же, перебрав все возможные резольвенты формулы Φ , мы не нашли пустую резольвенту, то Φ не является невыполнимой.

Метод резолюций есть фактически правило присоединения к рассуждению, в состав которого входят два утверждения $A \Rightarrow B$ и $\neg A \Rightarrow C$ их следствия — утверждения $B \vee C$, что интуитивно совершенно очевидно. Действительно, первое утверждение гарантирует, что если A истинно, то B тоже истинно, а второе — что если A ложно, то истинно C . Очевидно, что в этом случае хотя бы одно из двух, B или C , всегда истинно. Добавление этого нового утверждения к исходному рассуждению не меняет его смысла.

Пример 2.8

Доказательство правильности рассуждения «Узнала — спросила» сводится к проверке невыполнимости формулы $\Phi = F1 \& F2 \& F3 \& \neg R$, где

$$F1 = \neg Ck \Rightarrow \neg Y, F2 = \neg Cp \Rightarrow \neg Ck, F3 = Y \text{ и } R = Cp.$$

Перечислим все дизъюнкты конъюнктивной нормальной формы Φ и построим возможные резольвенты.

N	Дизъюнкт	Откуда он получен	Интерпретация
(1)	$Ck \vee \neg Y$	Первое утверждение	Известно, что: «Если бы он не сказал ей, она бы не узнала»;
(2)	$Cp \vee \neg Ck$	Второе утверждение	и что «Если бы она его не спросила, он бы не сказал ей»;
(3)	Y	Третье утверждение	и что «Она узнала»
(4)	$\neg Cp$	Отрицание следствия	Докажем, что «Она спросила». Предположим противное, то есть что «Она не спросила»
(5)	$\neg Ck$	Резольвента 2 и 4	Отсюда следует (поскольку по (2): «Если бы она его не спросила, он бы не сказал ей»), что «Он не сказал ей»
(6)	$\neg Y$	Резольвента 1 и 5	Отсюда следует (поскольку по (1): «Если бы он не сказал ей, она бы не узнала»), что «Она не узнала»
(7)	\square	Резольвента 3 и 6, пустой дизъюнкт	Мы пришли к противоречию: ведь известно (3), что «Она узнала». Следовательно, наше предположение «Она не спросила» неверно (оно противоречит фактам), поэтому утверждение «Она спросила» верно

Доказательство методом резолюции сделаем более наглядным, представив его графически.

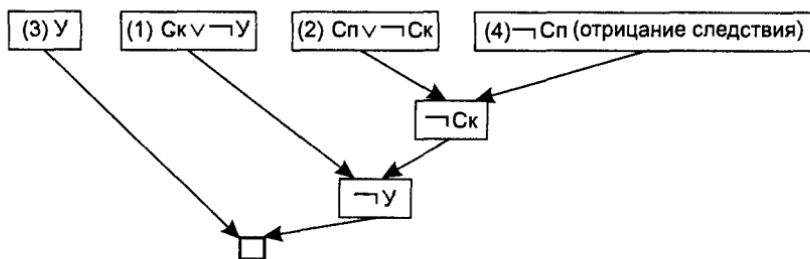


Рис. 2.2. Графическое представление доказательства методом резолюции

Доказательство методом резолюции очень близко обычному словесному доказательству от противного. Это демонстрирует интерпретация последовательных формул вышеприведенной таблицы. Проинтерпретируем еще раз это доказательство в естественном языке по графу (рис. 2.2).

Рассуждение. «Если бы он не сказал ей, она бы и не узнала (1). А не спроси она его, он и не сказал бы ей (2). Но она узнала (3). Следовательно, она спросила (4)».

Доказательство (см. рис. 2.2). «**Предположим противное**, то есть что она не спросила. Тогда из второго утверждения следует, что он ей не сказал. Отсюда и из первого утверждения следует, что она не узнала. Но в третьем утверждении говорится, что она узнала. Таким образом, предположив, что следствие неверно, мы пришли к противоречию. **Поэтому следствие верно**».

Для рассуждения примера 2.2. «В хоккей играют настоящие мужчины. Трус не играет в хоккей. Я в хоккей не играю. Значит, я трус(?)!» метод резолюции дает (см. табл. 2.5):

- (1) $\neg X \vee M$ — первое утверждение: «В хоккей играют настоящие мужчины»;
- (2) $M \vee \neg X$ — второе утверждение: «Трус не играет в хоккей»;
- (3) $\neg X$ — третье утверждение: «Я в хоккей не играю»;
- (4) M — отрицание следствия: «**Предположим**, что неверно, что „Я трус“».

Резольвент нет

Мы не можем построить здесь ни одной резольвенты, не говоря уж о пустой. Отсюда можно заключить, что вывод в этом рассуждении **не является логическим следствием** высказанных утверждений и не следует формально из их истинности. Для доказательства (как и для опровержения) этого вывода необходимы дополнительные факты.

Определение 2.7. Пусть S — множество дизъюнктов. Резолютивный вывод C из S есть такая конечная последовательность C_1, \dots, C_k дизъюнктов, что $C_k = C$ и каждый дизъюнкт C_i или принадлежит S , или является резольвентой дизъюнктов, предшествующих C_i . Вывод пустого дизъюнкта из S называется опровержением S .

Следующая теорема является важнейшей теоремой теории логического вывода в логике высказываний: она утверждает, что используя единственное правило вывода — резолюцию, мы можем проверить правильность любого заключения из фактов, последовательно получая резольвенты и присоединяя их к исходной формулировке.

Теорема 2.8. (*О полноте метода резолюции.*) *Множество дизъюнктов S невыполнимо тогда и только тогда, когда существует резолютивный вывод пустого дизъюнкта из S .*

Необходимость. Доказательство того, что при невыполнимости S всегда найдется резолютивный вывод пустого дизъюнкта из S за конечное число шагов, здесь опускается.

Достаточность. Пусть из S существует резолютивный вывод пустого дизъюнкта. Докажем, что S невыполнимо. Поскольку резольвента R есть логическое следствие порождающих ее дизъюнктов D_1 и D_k , то конъюнктивное присоединение резольвенты R к формуле S не меняет ее значения, поскольку $S = AD_1D_k$, а $D_1D_k \equiv D_1D_kR$ согласно теоремам 2.5 и 2.6. Если среди резольвент S существует пустая резольвента, соответствующая ее логическому следству **False**, то S невыполнима, поскольку равносильна формуле с конъюнктивным членом **False**.

Перебор всех возможных резольвент можно организовать по-разному, и разные стратегии построения резольвент будут (в среднем) иметь различную эффективность (число шагов вывода до получения положительного или отрицательного ответа). Одной из наиболее эффективных стратегий является стратегия движения «от цели» — получение на каждом шаге резольвент из двух дизъюнктов, один из которых — это цель доказательства (инверсия предполагаемого следствия) или резольвента — потомок этой цели. Эта стратегия особенно эффективна в языках логического программирования, где только небольшое число фактов общей базы данных действительно имеют отношение к заданному вопросу. Мы рассмотрим использование метода резолюции в языках логического программирования ниже.

Другие методы

Использование семантических деревьев для проверки общезначимости или невыполнимости формулы было предложено Куайном. Идея основывается на том, что часто нет необходимости строить все дерево до листьев: в некотором узле значение функции может быть определено сразу для всех листьев, которые могут быть построены из этого узла. Метод Куайна был усовершенствован Девисом и Патнемом, предложившими для проверки невыполнимости формулы предварительно представить эту формулу в КНФ. Функция представляется в виде совокупности дизъюнктов, каждый из которых — просто множество литер, и алгоритм сводится к вычеркиванию литер из этих множеств в узлах семантического дерева.

Адекватность логики высказываний

Вопрос об адекватности математической модели при решении конкретных проблем в реальной жизни, как уже говорилось выше, лежит вне самого математичес-

кого аппарата, это проблема, которую должен решать человек, использующий этот аппарат для решения задач практики. В области формализации рассуждений естественного языка всегда надо быть осторожным, применяя выводы логики высказываний к реальной жизни. Хорошую иллюстрацию этому дает так называемая «задача Кислера», приведенная в [22].

Браун, Джонс и Смит обвиняются в преступлении. На допросе они под присягой дали показания:

Браун: *Джонс виновен, а Смит невиновен* ($D \& \neg C$).

Джонс: *Браун без помощи Смита это не смог бы сделать* ($B \Rightarrow C$).

Смит: *Я невиновен, но кто-то из них виновен* ($\neg C \& (B \vee D)$).

Из этих утверждений следует, что виноват Джонс, а двое других подозреваемых невиновны (докажите!). Но задумаемся, **могем ли мы считать эти посылки истинными?** Что, если виновный лжет, а невиновный говорит правду? В этом случае нахождение максимально сильного следствия *шести* посылок:

$$F1 : \neg B \Rightarrow (D \& \neg C); \quad F3 : \neg D \Rightarrow (B \Rightarrow C); \quad F5 : \neg C \Rightarrow \neg C \& (B \vee D);$$

$$F2 : B \Rightarrow \neg(D \& \neg C); \quad F4 : D \Rightarrow \neg(B \Rightarrow C); \quad F6 : C \Rightarrow \neg(\neg C \& (B \vee D))$$

приводит к полностью противоположному выводу. Если же мы будем полагаться на информацию только тех, кто невиновен (то есть учитывать *только* утверждения $F1, F3, F5$), то мы получим третий результат: установить в точности, кто виновен, без дополнительной информации нельзя. Таким образом, результаты анализа одной и той же ситуации с привлечением одного и того же аппарата — логики высказываний — существенно меняются в зависимости от того, *как мы формализуем* задачу проверки правильности рассуждений и *какие* факты мы будем считать установленными.

Другим примером, иллюстрирующим это положение, является доказательство древнегреческого философа Зенона, что движения нет: «Если тело движется, то движение может происходить либо там, где тело есть, либо там, где тела нет. Но движение не может происходить там, где тело есть, потому что тогда бы оно стояло на месте. Движение также не может происходить там, где тела нет: если там тела нет, то как оно может там двигаться? Следовательно, тело двигаться не может». Логическая структура этого доказательства совершенно правильна, но формальная логика не может отвечать за то, что **содержание** умозаключения неверно. Движение все же есть!

Основы логики предикатов и логического вывода

Предикаты

Логика высказываний оперирует с атомами. Атомы являются абстракциями простейших повествовательных предложений, которые могут быть истинными или ложными. Атом рассматривается как неделимое целое, его структура не анализи-

руется. Таким образом, аппарат, подходы и результаты логики высказываний могут быть применены только к очень узкому классу ситуаций — самых простых расуждений на естественном языке.

В естественном языке встречаются более сложные повествовательные предложения, истинность которых может меняться при изменении объектов, о которых идет речь, хотя структура предложений не меняется. Например, предложение «Джон любит Мэри» может быть истинным, а предложение «Джон любит Мэгги» ложным. В логике такие предложения, истинность которых зависит от параметров, абстрагируют с помощью предикатов. Например, предикат $\text{Любит}(x, y)$ на паре $\langle \text{Джон}, \text{Мэри} \rangle$ может принимать значение истина, а на паре $\langle \text{Джон}, \text{Мэгги} \rangle$ — ложь. На латыни слово «предикат» (*praedicatum*) означает «сказанное», то есть то, что в элементарном суждении утверждается о субъекте этого суждения, свойства этого субъекта. Например, высказывание «Собака имеет хвост» истинно, «Лошадь имеет хвост» истинно, а «Человек имеет хвост» ложно. Поэтому заменив субъект в суждении переменной x , получим некую форму « x имеет хвост», которая является функцией от x и принимает значения **истинно** для одних субъектов-аргументов этой функции и **ложно** для других. Формализация подобной высказывательной формы и называется предикатом, то есть функцией, принимающей истинностные значения (*истина либо ложь*) и определенной на произвольной области изменения своей переменной; n -местный предикат является естественным обобщением функции одной переменной. Иными словами, предикат — это переменное высказывание, истинностное значение которого зависит от параметров (переменных).

Определение 2.7. n -местным предикатом $P(x_1, \dots, x_n)$ называется функция $P: M^n \rightarrow \{\text{истина, ложь}\}$, определенная на наборах длины n элементов некоторого множества M и принимающая значения в области истинностных значений (рис. 2.3, а).

Множество M называется предметной областью предиката, а x_1, \dots, x_n — предметными переменными. Пусть M — множество натуральных чисел и $n = 2$. Двуместный предикат $P(x, y): x \geq y + 3$ истинен на парах $\langle 25, 1 \rangle$ и $\langle 15, 12 \rangle$ и ложен на паре $\langle 1, 1 \rangle$. Поскольку предикат каждому элементу множества M^n ставит в соответствие *истина либо ложь*, то можно считать, что предикат выделяет на M^n некоторое подмножество, на котором он истинен (рис. 2.3, б). Таким образом, n -местный предикат P на M может характеризовать (задавать) некоторое подмножество элементов M^n , а именно те n -мерные векторы на M , на которых P истинен.

Предикаты могут быть связаны логическими связками, например, $P(x, y) \Rightarrow Q(x, y)$. Эта формула принимает истинное значение на тех парах аргументов, на которых предикат P ложен или же предикат Q истинен. Очевидно также, что если на **каждом** элементе множества M^n , на котором предикат P истинен, предикат Q тоже истинен, формула $P \Rightarrow Q$ общезначима. Таким образом, формула $P \Rightarrow Q$ общезначима тогда и только тогда, когда множество истинности предиката P является подмножеством множества истинности предиката Q , или, что то же, предикат P **сильнее** предиката Q , как это показано на рис. 2.3, в.

В качестве аргументов предикатов можно использовать и функции, принимающие значения из предметной области. Например, функция $\text{Отец}(x)$ пусть абстра-

гирует предложение естественного языка «Отец человека по имени x ». Тогда Любить(Отец(Отец(x)), y) — тоже предикат, он будет принимать истинное значение на паре <Мэри, Джон>, если дедушка Мэри действительно любит Джона.



Рис. 2.3. Графическое представление предикатов

Итак, в отличие от логики высказываний, в которой структура простейших утверждений не анализируется (они называются атомами), в логике предикатов **атомный предикат имеет структуру**.

Определение 2.8. Термом будем называть переменную или константу предметной области M или функцию, принимающую значения в предметной области. Пусть t_1, \dots, t_n — термы. Атомный предикат — это n -местная функция $F(t_1, \dots, t_n)$ принимающая значение на множестве {истина, ложь}.

Введем понятие формулы — комбинации простейших предикатов.

Определение 2.9. Правильно построенные формулы (ППФ) логики предикатов — это комбинации атомных предикатов и констант с логическими связками. Они определяются рекурсивно над множеством атомных предикатов с помощью символов операций (связок) $\neg, \Rightarrow, \&$, скобок и одной дополнительной связки \forall , которая читается «для всех». Рекурсивно ППФ определяются так:

1. Атомный предикат есть формула.
2. Если P — формула, то $\neg(P)$ тоже формула.
3. Если P, Q — формулы, то $(P \Rightarrow Q)$ — тоже формула.
4. Если P — формула, то $(\forall x) P$ тоже формула.
5. Никаких других формул в логике предикатов нет.

Фактически, добавлением в логике предикатов по сравнению с логикой высказываний является то, что предикаты обычно не имеют фиксированного истинностного значения, их истинность различна для разных интерпретаций, в частности, различных значениях их аргументов. В качестве единственной новой логической связки в логике предикатов используется связка \forall .

В логике предикатов для сокращения формул используются записи:

True для $P \Rightarrow P$, **False** для $\neg\text{True}$ (можно рассматривать **True** и **False** как логические константы),

$P \vee Q$ для $(\neg P) \Rightarrow Q$,

$P \& Q$ для $\neg(\neg P \vee \neg Q)$,

$P \Leftrightarrow Q$ для $(P \Rightarrow Q) \& (Q \Rightarrow P)$,

$(\exists x)P$ для $\neg((\forall x)\neg P)$.

Новые логические связки \forall — «для всех» и \exists — «существует» называются кванторами: \forall — «квантор всеобщности» и \exists — «квантор существования». Формула $(\forall x)P(x)$ читается так: «Для всех x выполняется $P(x)$ ». Если P зависит только от x , то $(\forall x)P(x)$ уже не зависит от x , это константа **True** либо **False**, в зависимости от того, действительно ли для каждого элемента x предметной области высказывание $P(x)$ истинно. Аналогично, логической константой является $(\exists x)P(x)$, что читается так: «Существует такое x , что для него выполняется $P(x)$ ». Для конечной предметной области значения этих связок очевидны. Пусть на $M = \{x_1, x_2, \dots, x_n\}$ определен произвольный предикат $P(x)$. Тогда очевидно:

$$(\forall x)P(x) = P(x_1) \& P(x_2) \& \dots \& P(x_n) = \&_{x \in M} P(x);$$

$$(\exists x)P(x) = P(x_1) \vee P(x_2) \vee \dots \vee P(x_n) = \vee_{x \in M} P(x).$$

Пример 2.9

Пусть $P(x)$ — предикат, определенный на множестве всех людей и означающий « x является студентом». Тогда $(\forall x)P(x)$ ложно, а $(\exists x)P(x)$ истинно. Далее, формула $(\forall x)\text{Любит}(x, \text{Отец}(\text{Отец}(Джон)))$ истинна, если все любят дедушку Джона, а формула $(\exists x)[P(x) \Rightarrow \neg\text{Любит}(\text{Отец}(Джон), x)]$ истинна, если отец Джона не любит хотя бы одного студента. Далее, если $R(x, y)$ — предикат, определенный на множестве всех семейных людей и означающий « x и y — супруги», то

$$(\forall x)(\exists y)R(x, y) \neq (\exists x)(\forall y)R(x, y).$$

Первая формула *истинна*, вторая — *ложна*.

Свободные и связанные переменные

Область действия переменной, указанной в кванторе, если она не очевидна, определяется скобками, например:

$$G(x, y) = (\forall t)[P(t, y) \vee (\forall u)Q(t, y, u)] \Rightarrow (\forall r)R(x, r).$$

Переменная связана, если она находится в области действия квантора. Например, в предикате $(\forall t)P(t, z)$ переменная t связана точно так же, как связана переменная t в интеграле

$$\int_a^b f(t, z) dt.$$

Связанные переменные можно заменять, если это не приводит к изменению смысла формулы. Предыдущий предикат равносителен такому:

$$G(x, z) = (\forall x)[P(x, z) \vee (\forall z)(\exists y)Q(x, y, z)] \Rightarrow (\forall z)R(x, z).$$

Интерпретации

Формула логики предикатов является только схемой высказывания. Формула имеет определенный смысл, то есть обозначает некоторое высказывание естественного языка, если существует какая-либо ее интерпретация. Интерпретировать фор-

мулу — это значит связать с ней непустое множество M (конкретизировать предметную область), а также сопоставить:

- каждой предметной константе конкретный элемент из M ;
- каждой n -местной функциональной букве в формуле конкретную n -местную функцию на M ;
- каждой n -местной предикатной букве в формуле конкретное отношение между n элементами области интерпретации M . На каждой интерпретации формула логики предикатов, в которой все переменные связаны, должна принять конкретное истинностное значение, *истина либо ложь*.

Рассмотрим, например, элементарную формулу $G(f(a, b), g(a, b))$ и следующую ее интерпретацию:

- M — множество действительных чисел;
- a, b — числа 2 и 3 соответственно;
- f — функция сложения аргументов;
- g — функция умножения аргументов;
- G — отношение «не меньше».

При такой интерпретации приведенная формула обозначает высказывание: «сумма $2+3$ не меньше произведения 2×3 », что неверно. Следовательно, $G(f(a, b), g(a, b))$ на этой интерпретации равна *ложь*. На другой интерпретации, когда в качестве a и b мы выберем 2 и 1, $G(f(a, b), g(a, b)) = \text{истина}$.

Две формулы логики предикатов равносильны, если они принимают однаковые истинностные значения на *всех* возможных интерпретациях. В отличие от логики высказываний, значения формул логики предикатов уже нельзя определить на основе таблиц истинности на *всех* возможных интерпретациях: предметная область может быть бесконечной. Единственный способ определить равносильность формул — использовать *аналитические преобразования* на основе установленных равносильностей. Это и составляет основную трудность и качественное отличие этой модели от логики высказываний.

Равносильности логики предикатов

(a) *Комбинация кванторов:*

1. $(\forall x)(\forall y)P(x, y) = (\forall y)(\forall x)P(x, y);$
2. $(\exists x)(\exists y)P(x, y) = (\exists y)(\exists x)P(x, y);$
3. $(\forall x)(\exists y)P(x, y) \neq (\exists y)(\forall x)P(x, y).$

Свойства (a) 1 и 2 аналогичны свойствам коммутативности дизъюнкции и конъюнкции. Свойство 3 доказано в примере 2.9. Оно говорит о том, что перестановка местами кванторов существования и общности *меняет* смысл высказывания, а в общем случае, конечно, и значение его истинности.

(б) Комбинация кванторов и отрицаний:

1. $\neg(\forall x)P(x) = (\exists x)\neg P(x);$
2. $\neg(\exists x)P(x) = (\forall x)\neg P(x).$

Свойства (б) непосредственно следуют из определения квантора Э. Это аналог теорем де Моргана.

Пример 2.10

Афоризм Козьмы Пруткова «Нет столь великой вещи, которую не превзошла бы величиною еще большая» ([32]) формально в виде предиката может быть записан так: $\neg(\exists x \neg \exists y P(y, x))$, где x и y принимают значения в множестве всех вещей, а утверждение $P(a, b)$ означает «Вещь a превосходит величиной вещь b ». Этот предикат равносителен такому: $(\forall x \exists y P(y, x))$, что дает эквивалентную формулировку этого афоризма, менее замысловатую: «Для любой (большой) вещи всегда найдется (еще) большая».

(в) Расширение области действия кванторов (S не зависит от x):

1. $(\forall x)P(x) \vee S = (\forall x)[P(x) \vee S];$
2. $(\exists x)P(x) \vee S = (\exists x)[P(x) \vee S];$
3. $(\forall x)P(x) \& S = (\forall x)[P(x) \& S];$
4. $(\exists x)P(x) \& S = (\exists x)[P(x) \& S].$

Эти свойства являются следствием свойств коммутативности и дистрибутивности дизъюнкций и конъюнкций.

(г) Расширение области действия кванторов (S зависит от x):

1. $(\exists x)P(x) \vee (\exists x)S(x) = (\exists x)[P(x) \vee S(x)];$
2. $(\forall x)P(x) \& (\forall x)S(x) = (\forall x)[P(x) \& S(x)];$
3. $(\exists x)[P(x) \& S(x)] \Rightarrow (\exists x)P(x) \& (\exists x)S(x);$
4. $(\forall x)P(x) \vee (\forall x)S(x) \Rightarrow (\forall x)[P(x) \vee S(x)].$

Свойства (г) 2 и 4 являются следствием свойств 1 и 3 и выводятся с применением теоремы де Моргана. Свойство 1 иллюстрируется следующим высказыванием, обе части которого эквивалентны: «Среди нас есть английский шпион, или же среди нас — японский шпион. Да, в наши ряды затесался английский или японский шпион». Свойство 2 иллюстрирует следующее высказывание, обе части которого, очевидно, также эквивалентны: «Все — летчики, и все — герои. Каждый — и летчик, и герой». Свойство 3. Высказывание: «Наша Света — комсомолка, спортсменка, отличница» более сильное, чем высказывание: «Есть у нас комсомолки, есть спортсменки, есть и отличницы», потому что второе не обязательно означает, что перечисленными качествами: «быть комсомолкой, спортсменкой и отличницей» обладает одно лицо.

Переход от естественного языка к формулам логики предикатов не всегда однозначен, что зачастую отражает двусмысленность естественного языка. В качестве примера рассмотрим формализации, которые возможны для следующего высказывания ([21], с. 59): «На кафедре трудятся преподаватели, которые свободно владеют английским, французским и немецким языками». Возможны следующие четыре формальные представления этого высказывания:

1. $(\forall x)[A(x) \& \Phi(x) \& H(x)]$: «Все преподаватели кафедры свободно владеют всеми тремя языками»;
2. $(\forall x)A(x) \& (\forall x)\Phi(x) \& (\forall x)H(x)$: «Каждый преподаватель кафедры владеет каждым иностранным языком» (это, очевидно, эквивалентно 1);
3. $(\exists x)[A(x) \& \Phi(x) \& H(x)]$: «На кафедре есть преподаватели, которые владеют и английским, и французским, и немецким языками»;
4. $(\exists x)A(x) \& (\exists x)\Phi(x) \& (\exists x)H(x)$: «На кафедре трудятся преподаватели, которые свободно владеют английским, есть те, которые свободно владеют французским, а также и те, которые знают немецкий».

Ограниченные предикаты

В математике часто используются предикаты, различные переменные которых определены на разных множествах значений. Кванторы существования и всеобщности для таких переменных могут сопровождаться указанием того множества значений, на котором переменная определена.

Определение 2.10. *Ограниченым предикатом называется предикат, определенный не на всей предметной области, а на множестве объектов, удовлетворяющих дополнительному условию.*

Простейшие примеры ограниченных предикатов — $(\forall x \in X)P(x)$ и $(\exists x \in X)P(x)$, что имеет смысл: «для всех x , принадлежащих множеству X , справедливо $P(x)$ » и «существует такое x , принадлежащее множеству X , для которого справедливо $P(x)$ ». Очевидно, что предикат $(\forall x \in X)P(x)$ равносителен $(\forall x)(x \in X) \Rightarrow P(x)$ и $(\exists x \in X)P(x)$ равносителен $(\exists x)(x \in X) \& P(x)$.

В более общем случае иногда удобно записывать предикаты относительно объектов, удовлетворяющих дополнительному условию, например: $(\forall x : R(x))P(x)$ и $(\exists x : R(x))P(x)$. Первое из них читается: «для любого x , удовлетворяющего условию $R(x)$, справедливо $P(x)$ », а второе: «существует x , удовлетворяющее условию $R(x)$, для которого справедливо $P(x)$ ». Эти *ограниченные* предикаты равносильно можно представить так:

$$(\forall x : R(x))P(x) \equiv (\forall x)R(x) \Rightarrow P(x) \text{ и } (\exists x : R(x))P(x) \equiv (\exists x)R(x) \& P(x).$$

Например, пусть предикат $P(x)$ означает « x — рисковый человек»; $Ш(x)$ — « x пьет шампанское». Тогда высказывание «Кто не рискует, тот не пьет шампанского» можно формализовать так: $(\forall x)(\neg P(x) \Rightarrow \neg Ш(x))$ «для любого человека верно, что если он не рискует, то он не пьет шампанского», или эквивалентно:

$(\forall x : \neg P(x)) \rightarrow \neg \exists x P(x)$ «любой из тех, которые не рисуют, не пьет шампанского».
Другой пример. Пусть $P(x)$: « x поехал»; $X(x)$ « x хотел поехать». Тогда высказывание «Все те, кто поехал, хотели поехать» формализуется так: $(\forall x : P(x))X(x)$, а высказывание «Все те, которые хотели поехать, поехали» — так: $(\forall x : X(x))P(x)$.

Теорема 2.9. Для ограниченных предикатов выполняются законы де Моргана:

$$\begin{aligned}\neg(\forall x \in X)P(x) &= (\exists x \in X)\neg P(x); \\ \neg(\exists x \in X)P(x) &= (\forall x \in X)\neg P(x); \\ \neg(\forall x : R(x))P(x) &= (\exists x : R(x))\neg P(x); \\ \neg(\exists x : R(x))P(x) &= (\forall x : R(x))\neg P(x).\end{aligned}$$

Доказательство этой теоремы остается в качестве упражнения (задача 2.23).

Пример 2.11

Формальным выражением того факта, что массив x_1, x_2, \dots, x_n упорядочен по возрастанию, является предикат $(\forall i \in \{1, \dots, n-1\})x_i \leq x_{i+1}$, что словесно звучит так: «Массив не упорядочен по возрастанию, если любой его элемент не больше, чем следующий за ним элемент». Выразим формально отрицание этого факта, то есть утверждение, что этот массив не упорядочен по возрастанию:

$$\neg(\forall i \in \{1, \dots, n-1\})x_i \leq x_{i+1} \equiv (\exists i \in \{1, \dots, n-1\})\neg(x_i \leq x_{i+1}) \equiv (\exists i \in \{1, \dots, n-1\})x_i > x_{i+1}.$$

Таким образом, противоположное утверждение звучит так: «Массив не упорядочен по возрастанию, если существует такой его элемент, который больше следующего».

Пример 2.12

Определение достижимого состояния конечного автомата записывается так: «Состояние s конечного автомата является достижимым тогда и только тогда, когда существует такая входная цепочка, что под ее воздействием автомат переходит в это состояние из начального состояния s_0 ». В виде предиката это записывается короче: $\text{Достижимо}(s) \Leftrightarrow (\exists a \in X^*)\delta^*(s_0, a) = s$. Исходя из этого определения, определим понятие недостижимого состояния конечного автомата. В виде предиката это записывается так: $\text{Недостижимо}(s) \Leftrightarrow \neg[(\exists a \in X^*)\delta^*(s_0, a) = s]$. Отсюда: $\text{Недостижимо}(s) \Leftrightarrow (\forall a \in X^*)\delta^*(s_0, a) \neq s$. В словесной формулировке: «Состояние s конечного автомата недостижимо тогда и только тогда, когда под воздействием любой входной цепочки автомат не переходит в это состояние из начального состояния s_0 ».

Пример 2.13

В теории конечных автоматов существует теорема Т:

$$p \approx_{k+1} q \Leftrightarrow (\forall x \in X)[\delta(p, x) \approx_k \delta(q, x) \& \lambda(p, x) = \lambda(q, x)]$$

(здесь $p \approx_k q$ обозначает утверждение «состояние p k -эквивалентно состоянию q », а $\neg p \approx_k q$ обозначает утверждение «состоиния p и q являются k -различимыми»).

Мы можем рассмотреть, на какие более простые части распадается доказательство этой теоремы, даже не понимая, о чём идет речь. Доказательство необходимости состоит здесь в том, что нужно показать

$$p \approx_{k+1} q \Rightarrow (\forall x \in X) [\delta(p, x) \approx_k \delta(q, x) \& \lambda(p, x) = \lambda(q, x)].$$

Доказательство этого более простого утверждения можно провести от противного, то есть доказать его контрапозицию:

$$(\exists x \in X) [\neg \delta(p, x) \approx_k \delta(q, x) \vee \lambda(p, x) \neq \lambda(q, x)] \Rightarrow \neg p \approx_{k+1} q.$$

Из свойств предикатов ясно, что эта формула равносильна конъюнкции двух утверждений:

$$(\exists x \in X) [\neg \delta(p, x) \approx_k \delta(q, x)] \Rightarrow \neg p \approx_{k+1} q$$

и

$$(\exists x \in X) [\lambda(p, x) \neq \lambda(q, x)] \Rightarrow \neg p \approx_{k+1} q.$$

Словами это можно передать следующим образом. **T1**: «Если существует такой элемент x из X , что состояния $\delta(p, x)$ и $\delta(q, x)$ k -различны, то p и q являются $k+1$ -различимыми» и **T2**: «Если существует такой элемент x из X , что $\lambda(p, x)$ не равно $\lambda(q, x)$, то состояния p и q являются $k+1$ -различимыми». Очевидно, что доказательство двух теорем **T1** и **T2**, а также теоремы **T3**:

$$(\forall x \in X) [\delta(p, x) \approx_k \delta(q, x) \& \lambda(p, x) = \lambda(q, x)] \Rightarrow p \approx_{k+1} q$$

в совокупности заменяет доказательство намного более сложной исходной теоремы **T**.

Пример 2.14

Лемма о накачке в теории конечных автоматов имеет следующую формулировку:

Лемма 1. «Пусть L – автоматный язык над алфавитом Σ . Тогда:

$$(\exists n \in N) (\forall x \in L : |x| \geq n) (\exists u, v, w \in \Sigma^*) : [x = uvw \& |uv| \leq n \& |v| \geq 1 \& (\forall i \in N) (uv^i w \in L)].$$

Другая форма этой леммы, которую иногда удобнее применять, записывается так:

Лемма 2. «Пусть L – некоторый язык над алфавитом Σ . Если:

$$(\forall n \in N) (\exists x \in L : |x| \geq n) (\forall u, v, w \in \Sigma^*) : [x = uvw \& |uv| \leq n \& |v| \geq 1 \& (\exists i \in N) (u v^i w \notin L)],$$

то L – не автоматный».

Какую лемму следует доказать, чтобы убедиться в истинности обеих? Или же каждая из лемм требует собственного доказательства? Перепишем формулировку лемм в более компактном виде, представив предикат $x = uvw \& |uv| \leq n \& |v| \geq 1$ как $M(x, u, v, w)$, а $uv^i w \in L$ как $N(i, u, v, w)$. Пусть $A(L)$ означает утверждение: L – автоматный язык. После очевидных преобразований первая лемма может быть представлена как $Q[A(L) \Rightarrow M \& N]$, а вторая – как $Q[A(L) \Rightarrow \neg(M \& N)]$, где Q – одинаковые кванторы. Дальнейший анализ состоит в проверке того, какая из формул, стоящих под знаком кванторов, является следствием другой. Этот анализ предоставим читателю.

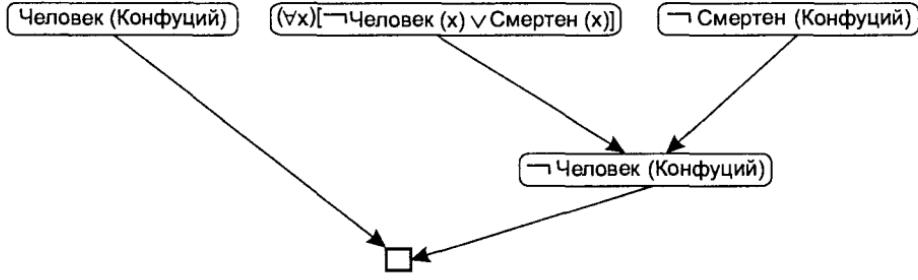
Логический вывод в логике предикатов

Пример 2.15

Рассмотрим доказательное рассуждение «Каждый человек смертен. Конфуций – человек. Следовательно, Конфуций смертен». Представим схему рассуждения.

-
- | | |
|-------|--|
| (F1) | $(\forall x) [\text{Человек}(x) \Rightarrow \text{Смертен}(x)];$ |
| (F2) | Человек (Конфуций); |
| <hr/> | |
| (R) | $\therefore \text{Смертен} (\text{Конфуций}).$ |

Доказательство методом резолюции.



Это доказательное рассуждение, исторически одно из первых, которое было изучено с точки зрения именно формы (а не смысла), приписывается Г. Оккаму (1349 год). В силлогистике, создателем которой был Аристотель, подобные силлогизмы стали изучаться в их общей форме с использованием замещающих переменных, которые сохраняют форму исходного силлогизма. Так, моделью приведенного рассуждения является форма: «Х суть У, и если С есть Х, то С есть У». Заметим, что соответствие структуры доказательства методом резолюций структуре доказательства, применяемого человеком в естественном языке, проявляется здесь очень наглядно. Например, переложенное на естественный язык обоснование рассуждения: «Каждый человек смертен. Конфуций – человек. Следовательно, Конфуций смертен» в соответствии с вышеприведенной схемой метода резолюции выглядит так: «Предположим противное, то есть что Конфуций не смертен. Отсюда, учитывая первое утверждение: Каждый человек смертен, мож но заключить, что Конфуций – не человек. Но второе утверждение говорит, что Конфуций – человек. Следовательно, мы пришли к противоречию, что показывает, что наше предположение неверно».

Скулемовская стандартная форма

В логическом выводе используют предикаты в *предваренной* форме, когда все кванторы вынесены в формуле влево, то есть предваряют формулу. Эту формулу, которая уже не содержит кванторов, можно свести к конъюнктивной нормальной форме. Преобразование в предваренную формулу легко выполнить примене-

нием свойств равносильности и заменой переменных. Например, приведем формулу $(\forall x)P(x) \Rightarrow (\exists x)R(x)$ в предваренную форму:

$$\begin{aligned}(\forall x)P(x) \Rightarrow (\exists x)R(x) &\equiv \neg(\forall x)P(x) \vee (\exists x)R(x) \equiv \\&\equiv (\exists x)\neg P(x) \vee (\exists x)R(x) \equiv (\exists x)[\neg P(x) \vee R(x)] \equiv (\exists x)[P(x) \Rightarrow R(x)].\end{aligned}$$

Логический вывод легко применяется в случае, если в предваренной форме представления фактов присутствует **только** квантор всеобщности. В логике предикатов разработан метод, позволяющий исключить квантор существования из формулы в предваренной форме. Этот метод связан с введением так называемых «скулемовских функций». Рассмотрим этот метод, опуская обоснования, которые можно найти в [37].

Пусть предикат F находится в предваренной форме $(Q_1x_1) \dots (Q_nx_n)M$, где M есть предикат в **конъюнктивной нормальной форме**, а $(Q_1x_1) \dots (Q_nx_n)$ — префикс кванторов. Пусть Q_r есть самый левый квантор *существования*, $1 \leq r \leq n$. Если никакой квантор *всеобщности* не стоит левее Q_r , то выберем новую константу c , отличную от других констант, входящих в M , заменим все вхождения x_r , встречающиеся в M , на c и вычеркнем $(Q_r x_r)$ из префикса. Если же Q_{s1}, \dots, Q_{sm} — все кванторы *всеобщности*, встречающиеся левее Q_r , $1 \leq s_1 < s_2 < \dots < s_m < r \leq n$, выбираем новый m -местный функциональный символ f , отличный от всех других функциональных символов, заменим все, заменим все вхождения x_r , встречающиеся в M , на $f(x_{s1}, x_{s2}, \dots, x_{sm})$ и вычеркнем $(Q_r x_r)$ из префикса. Эту операцию применим для всех кванторов существования в префиксе слева направо; последняя из полученных формул есть *скулемовская стандартная форма*, к которой и применяется алгоритм логического вывода. Константы и функции, использованные для замены переменных квантора существования, называются *скулемовскими константами и функциями*.

Пример 2.16

Построим скулемовскую стандартную форму предиката

$$(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)P(x, y, z, u, v, w).$$

Левее $(\exists x)$ в формуле нет никаких кванторов всеобщности, поэтому заменим переменную x на константу a ; левее $(\exists u)$ стоят кванторы всеобщности $(\forall y)$ и $(\forall z)$, поэтому заменим переменную u на двухместную функцию $f(y, z)$; левее $(\exists w)$ стоят кванторы всеобщности $(\forall y)$, $(\forall z)$ и $(\forall v)$, поэтому заменим переменную w на трехместную функцию $g(y, z, v)$. Скулемовская стандартная форма представлена выше формулой — $(\forall y)(\forall z)(\forall v)P(a, y, z, f(y, z), v, g(y, z, v))$.

Пример 2.17

Докажем теорему о коммутативной группе: «Если произведение любого элемента группы G на себя есть единица этой группы, то группа G коммутативна».

Для доказательства запишем эту теорему формально. Пусть предикат $P(x, y, z)$ означает утверждение: «Если $x \in G$ и $y \in G$, то $z \in G$ и $xy = z$ ». Тогда теорема формально будет выглядеть так (единицу группы обозначим e):

$$R \equiv \{(\forall x)P(x, x, e) \Rightarrow (\forall u)(\forall v)(\forall w)[P(u, v, w) \Rightarrow P(v, u, w)]\}.$$

Докажем теорему R на основании только следующих двух аксиом, справедливых для любой группы G:

A1: «В группе существует единичный элемент», то есть для любого $x \in G$ верно, что: $ex = x$ и $xe = x$;

A2: «Любая группа G удовлетворяет свойству ассоциативности», то есть для любых $x, y, z \in G$ верно, что $(xy)z = x(yz)$.

Формальное выражение этих аксиом:

A1: $(\forall x)[P(e, x, x) \& P(x, e, x)]$;

A2: $(\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)[P(x, y, u) \& P(y, z, v) \Rightarrow [P(u, z, w) \Leftrightarrow P(x, v, w)]]$.

ПРИМЕЧАНИЕ

Формальную запись аксиомы A2 можно словами выразить так: «Если u является произведением x и y , а v является произведением y и z , то uz будет равно некоторому w тогда и только тогда, когда xv будет равно этому же w ».

Для доказательства теоремы покажем, что R является логическим следствием аксиом A1 и A2. Отрицание этой теоремы:

$$\begin{aligned} \neg R &\equiv \neg \{ \neg (\forall x)P(x, x, e) \vee (\forall u)(\forall v)(\forall w)[P(u, v, w) \Rightarrow P(v, u, w)] \} \equiv \\ &(\forall x)P(x, x, e) \& \neg (\forall u)(\forall v)(\forall w)[P(u, v, w) \Rightarrow P(v, u, w)] \equiv \\ &\neg (\forall u)(\forall v)(\forall w)[\neg P(u, v, w) \vee P(v, u, w)] \& (\forall x)P(x, x, e) \equiv \\ &(\exists u)(\exists v)(\exists w)[P(u, v, w) \& \neg P(v, u, w)] \& (\forall x)P(x, x, e) \equiv \\ &(\exists u)(\exists v)(\exists w)(\forall x)[P(u, v, w) \& \neg P(v, u, w) \& P(x, x, e)]. \end{aligned}$$

Скулемовские формы для аксиом и отрицания следствия:

A1: $(\forall x)[P(e, x, x) \& P(x, e, x)]$;

A2: $(\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)[\neg P(x, y, u) \vee \neg P(y, z, v) \vee [P(u, z, w) \Leftrightarrow P(x, v, w)]]$;

$\neg R (\forall x)[P(a, b, c) \& \neg P(b, a, c) \& P(x, x, e)]$.

Список дизъюнктов для метода резолюции:

1. $P(e, x, x)$ из A1;
2. $P(x, e, x)$ из A1;
3. $P(x, x, e)$ из $\neg R$;
4. $\neg P(x, y, u) \vee \neg P(y, z, v) \vee \neg P(u, z, w) \vee P(x, v, w)$ из A2;
5. $\neg P(x, y, u) \vee \neg P(y, z, v) \vee P(u, z, w) \vee \neg P(x, v, w)$ из A2;
6. $P(a, b, c)$ из $\neg R$;
7. $\neg P(a, b, c)$ из $\neg R$.

Алгоритм унификации

В процедуре доказательства методом резолюций для отождествления противоположных пар атомных предикатов (именно атомные предикаты выступают здесь в

роли литер, если говорить в терминах метода резолюции для высказываний) используется так называемая «процедура унификации» — приведение аргументов дизъюнктов, для которых ищется резольвента, к унифицированному (одинаковому) виду. В примере 2.14 при поиске резольвенты двух дизъюнктов $\neg\text{Человек}(x) \vee \text{Смертен}(x)$ и Человек (*Конфуций*) вместо переменной x можно подставить конкретное значение «Конфуций» именно потому, что квантор всеобщности предваряет первый дизъюнкт: поскольку это высказывание истинно для любого элемента предметной области (мижества людей), то оно справедливо и для конкретного человека по имени Конфуций.

Рассмотрим алгоритм унификации конечного множества выражений [37]. Алгоритм унификации пытается найти такие наименее общие замены переменных в выражениях, чтобы эти выражения стали тождественны. Например, чтобы два выражения $P(x)$ и $P(a)$ стали тождественны, необходимо просто переменную x заменить на константу a . Обобщим этот примитивный случай: построим алгоритм, который находит минимальную подстановку переменных всегда, когда она есть. Формализуем простейший случай. Чтобы отождествить $P(a)$ и $P(x)$, сначала найдем рассогласование выражений, которое равно $\{a, x\}$, а потом попытаемся его исключить конкретизацией переменных. В общем случае рассогласование множества выражений W строится выявлением первой слева позиции, на которой не для всех выражений из W стоит один и тот же символ. Далее с этой позиции из каждого выражения в W выделяются подвыражения, совокупность которых и составляет множество рассогласований. Например, если W есть $\{P(x, f(y, z)), P(x, a), P(x, g(h(x)))\}$, то первая позиция рассогласования есть пятая, поскольку у всех выражений первые первые символы « $P(x,$ » совпадают. Начинающиеся с пятой позиции подвыражения подчеркнуты; они и составляют множество рассогласований: $\{f(y, z), a, g(h(x))\}$.

Алгоритм унификации

Шаг 1. Полагаем $k = 0$, $W_k = W$, $\delta_k = \varepsilon$.

Шаг 2. Если W_k — содержит только один дизъюнкт, то останов, W — унифицируемо и δ_k — наименее общий унификатор для W . Если нет, найдем D_k — множество рассогласований для W_k .

Шаг 3. Если в D_k существует пара выражений v_k и t_k , такие, что v_k — переменная, а t_k — терм (возможно, другая переменная), не содержащий эту переменную, то положить $\delta_{k+1} = \delta_k \bullet \{t_k/v_k\}$, $W_{k+1} = W_k \{t_k/v_k\}$, $k = k+1$ и перейти к шагу 2.

В противном случае останов: множество W не унифицируемо.

В этом алгоритме используются обозначения $W_k \{t_k/v_k\}$, что означает подстановку терма t_k вместо всех вхождений переменной v_k , и $\delta_k \bullet \{t_k/v_k\}$, что означает произведение подстановок с очевидным смыслом. Рассмотрим работу алгоритма на примерах.

Пример 2.18 (а)

Наименее общий унификатор для $W = \{P(x, f(x)), P(z, f(x))\}$, очевидно, $\{x/z\}$ (или $\{z/x\}$). После замены переменных $W_1 = \{P(x, f(x)), P(x, f(x))\} = \{P(x, f(x))\}$ содержит только один член — единичный дизъюнкт. Таким образом, замена переменных при резолюции является частным случаем унификации.

Пример 2.18(б)

Найдем наиболее общий унификатор для $W = \{P(a, x, f(g(y))), P(f(z), f(u))\}$.

1. Шаг 1. $k = 0$, $W_0 = W$, $\delta_0 = \varepsilon$. Переходим к шагу 2.
2. Шаг 2. W_0 не является единичным дизъюнктом. $D_0 = \{a, z\}$. Переходим к шагу 3.
3. Шаг 3. Множество рассогласований D_0 содержит пару (a, z) , в которой z — переменная, а терм « a » эту переменную не содержит. Полагаем
 $\delta_1 = \delta_0 \bullet \{a/z\} = \varepsilon \bullet \{a/z\} = \{a/z\}$.
 $W_1 = W_0 \{a/z\} = \{P(a, x, f(g(y))), P(a, f(a), f(u))\}, k = 1$.
4. Шаг 2. W_1 не является единичным дизъюнктом. $D_1 = \{x, f(a)\}$. Переходим к шагу 3.
5. Шаг 3. Множество рассогласований D_1 содержит пару $(x, f(a))$, в которой x — переменная, а терм « $f(a)$ » эту переменную не содержит. Полагаем
 $\delta_2 = \delta_1 \bullet \{f(a)/x\} = \{a/z\} \bullet \{f(a)/x\} = \{a/z, f(a)/x\}$.
 $W_2 = W_1 \{f(a)/x\} = \{P(a, f(a), f(g(y))), P(a, f(a), f(u))\}, k = 2$.
6. Шаг 2. W_2 не является единичным дизъюнктом. $D_2 = \{g(y), u\}$. Переходим к шагу 3.
7. Шаг 3. Множество рассогласований D_2 содержит пару $(g(y), u)$, в которой u — переменная, а терм « $g(y)$ » эту переменную не содержит. Полагаем
 $\delta_3 = \delta_2 \{g(y)/u\} = \{a/z, f(a)/x\} \{g(y)/u\} = \{a/z, f(a)/x, g(y)/u\}$,
 $W_3 = W_2 \{g(y)/u\} = \{P(a, f(a), f(g(y))), P(a, f(a), f(g(y)))\} =$
 $= \{P(a, f(a), f(g(y)))\}, k = 3$.
8. Шаг 2. W_3 содержит только единичный дизъюнкт, Алгоритм завершился. Наиболее общий унификатор для $W = \{P(a, x, f(g(y))), P(z, f(z), f(u))\}$ — это подстановка $\delta_3 = \{a/z, f(a)/x, g(y)/u\}$, то есть нужно везде подставить вместо z константу a , вместо x — функцию $f(a)$, а вместо u — функцию $g(y)$.

Пример 2.19

Докажем справедливость следующего рассуждения [37].

Посылки:

F1:: Некоторые пациенты любят докторов.

F2:: Ни один пациент не любит захарей.

Заключение:

R:: Никакой доктор не является захарем.

Выделим элементарные высказывания:

P(x):: « x — пациент»;

D(x):: « x — доктор»;

$Z(x)$: « x — зондажары»;

$L(x, y)$: « x любит y ».

Предикатная запись рассуждения:

F1::($\exists x:P(x)(\forall y:D(y))L(x, y)$), *Некоторые пациенты любят (всех) докторов.*

F2::($\forall x:P(x)(\forall y:Z(y))\neg L(x, y)$), *Ни один пациент не любит (всех) зондажарей.*

Это же можно записать и так: F2::—($\exists x:P(x)(\exists y:Z(y))L(x, y)$), *Неверно, что существуют пациенты, которые любят зондажарей.*

R::($\forall x:D(x)\neg Z(x)$), *Никакой доктор не является зондажарем.*

Это же можно записать и так: R::—($\exists x:D(x))Z(x)$.

Раскрытие ограниченных предикатов:

F1::($\exists x[P(x) \& (\forall y)(D(y) \Rightarrow L(x, y))]$),

F2::($\forall x[P(x) \Rightarrow (\forall y)(Z(y) \Rightarrow \neg L(x, y))]$),

R::($\forall x(D(x) \Rightarrow \neg Z(x))$).

Отрицание заключения:

—R::($\exists x)\neg(\neg D(x) \vee \neg Z(x))$.

Представление посылок и отрицания заключения в предваренной конъюнктивной нормальной форме:

F1::($\exists x)(\forall x)[P(x) \& (\neg D(y) \vee L(x, y))]$),

F2::($\forall x)(\forall y)(\neg P(x) \vee \neg Z(y) \vee \neg L(x, y))$),

—R::($\exists x:D(x) \& Z(x)$).

Представление посылок и отрицания заключения в скулемовской нормальной форме:

F1::($\forall x)P(a) \& (\neg D(y) \vee L(a, y))$),

F2::($\forall x)(\forall y)(\neg P(x) \vee \neg Z(y) \vee \neg L(x, y))$),

—R::D(b) & Z(b).

Замена переменных:

F1::($\forall y)(P(a) \& (\neg D(y) \vee L(a, y)))$),

F2::($\forall x)(\forall z)(\neg P(x) \vee \neg Z(z) \vee \neg L(x, z))$),

—R::D(b) & Z(b).

Здесь произведена замена переменных, связанных квантором всеобщности: в F2 переменная y заменена на z с тем, чтобы не было коллизий с утверждением F1. Алгоритм унификации позволит автоматически произвести необходимую замену переменных при поиске резольвенты. Если этого не сделать, то могут возникнуть коллизии: как, например, унифицировать два предиката $P(x, f(z))$ и $P(z, f(x))$ из различных дизъюнкций?

Дизъюнкты для доказательства методом резолюции:

1. $P(a)$,
2. $\neg D(y) \vee L(a, y)$,
3. $\neg P(x) \vee \neg Z(z) \vee \neg L(x, z)$,
4. $D(b)$,
5. $Z(b)$.

Резольвенты:

6. $L(a, b)$ резольвента (4) и (2)
7. $\neg P(a) \vee \neg Z(b)$ резольвента (6) и (3)
8. $\neg Z(b)$ резольвента (7) и (1)
9. \square резольвента (8) и (5) – пустая

Доказательство методом резолюции позволяет построить словесную форму убедительного доказательства, каждый шаг которого обоснован.

Посылки:

F1:: *Некоторые пациенты любят врачей.*

F2:: *Ни один пациент не любит сахара.*

Заключение:

R:: Следовательно, *никакой врач не является сахаром.*

Доказательство. Предположим противное, то есть что *существует некий врач, одновременно являющийся и сахаром*. Пусть этот человек b (Отрицание следствия, $D(b) \wedge Z(b)$). Тогда из утверждения (F1:: *Некоторые пациенты любят (всех) врачей*) следует, что *существует пациент (назовем его a), который любит всех врачей и, конечно, любит конкретного врача b* (первое заключение – $L(a, b)$). Отсюда и из второго утверждения (F2:: *Ни один пациент не любит сахара*) следует, что *или a не пациент, или b не сахар* (заключение (7): $\neg P(a) \vee \neg Z(b)$). Но из утверждения F1 мы знаем, что a – пациент (дизъюнкт (1): $P(a)$), а из отрицания заключения ($\neg R$) по предположению, b – сахар (дизъюнкт (5): $Z(b)$). Таким образом, мы пришли к противоречию, и, следовательно, заключение нашего рассуждения верно.

Пример 2.17 (продолжение)

Докажем теперь теорему о коммутативной группе формально. Для этого нужно просто показать существование пустой резольвенты для множества дизъюнктов Д1–Д7 примера 2.17.

Отметим, что при нахождении резольвенты двух дизъюнктов, включающих переменные, связанные кванторами всеобщности, эти переменные «не видны» извне, и в разных дизъюнктах они разные, даже если обозначаются одинаково. При унификации следует это учитывать. Повторим список дизъюнктов для данного примера:

Д1: $P(e, x, x)$;

Д2: $P(x, e, x)$;

Д3: $P(x, x, e)$;

Д4: $\neg P(x, y, u) \vee \neg P(y, z, v) \vee \neg P(u, z, w) \vee P(x, v, w)$;

Д5: $\neg P(x, y, u) \vee \neg P(y, z, v) \vee P(u, z, w) \vee \neg P(x, v, w)$;

Д6: $P(a, b, c)$;

Д7: $\neg P(b, a, c)$.

Резольвенты и порождающие их дизъюнкты, построенные при доказательстве этой теоремы, представлены таблицей.

Родители	Унифицируемая пара предикатов	Унификатор	Полученная резольвента
Д3 и Д4.1	$P(x, x, e), P(x, y, u)$	{x/y, e/u}	Д8: $\neg P(x, z, v) \vee \neg P(e, z, w) \vee P(x, v, w)$;
Д6 и Д8.1	$P(a, b, c), P(x, z, v)$	{a/x, b/z, c/v}	Д9: $\neg P(e, b, w) \vee P(a, c, w)$;
Д9.1 и Д1	$P(e, b, w), P(e, x, x)$	{b/x, b/w}	Д10: $P(a, c, b)$;
Д7 и Д4.4	$P(b, a, c), P(x, v, w)$	{b/x, a/v, c/w}	Д11: $\neg P(b, y, u) \vee \neg P(y, z, a) \vee \neg P(u, z, c)$;
Д3 и Д11.1	$P(x, x, e), P(b, y, u)$	{b/x, b/y, e/u}	Д12: $\neg P(b, z, a) \vee \neg P(e, z, c)$;
Д12.2 и Д1	$P(e, z, c), P(e, x, x)$	{z/x, c/z}	Д13: $\neg P(b, c, a)$;
Д13 и Д5.3	$P(b, c, a), P(u, z, w)$	{b/u, c/z, a/w}	Д14: $\neg P(x, y, b) \vee \neg P(y, c, v) \vee \neg P(x, v, a)$;
Д10 и Д14.1	$P(a, c, b), P(x, y, b)$	{a/x, c/y}	Д15: $\neg P(c, c, v) \vee \neg P(a, v, a)$;
Д3 и Д15.1	$P(x, x, e), P(c, c, v)$	{c/x, e/v}	Д16: $\neg P(a, e, a)$;
Д16 и Д2	$P(a, e, a), P(x, e, x)$	{a/x}	Д17: \square — Пустая резольвента

Логическое программирование

В первом приближении логическое программирование — это использование deductивных процедур (процедур логического вывода) как механизма вычислений. Языки логического программирования являются декларативными в отличие от обычных «процедурных языков». В них в виде логических аксиом формулируются сведения о задаче и предположения, достаточные для ее решения. Сама задача формулируется как целевое утверждение, подлежащее доказательству. Таким образом, программа представляет собой множество аксиом, а вычисление — это конструктивный вывод целевого утверждения из программы. В логическом программировании (мы будем говорить о языке ПРОЛОГ) используется только одно правило вывода — резолюция.

Резолюция обладает важными свойствами — корректностью и полнотой. Резолюция корректна в том смысле, что если с ее помощью из множества формул $S = \{F_1, \dots, F_k\}$ и отрицания формулы R выводится пустой дизъюнкт, то справед-

ливо $F_1 F_2 \dots F_k \Rightarrow R$. Резолюция является полной в том смысле, что если R является логическим следствием множества формул S , то пустой дизьюнкт обязательно выводится из $F_1 F_2 \dots F_k \neg R$. Однако в 1936 году Черчем и Тьюрингом было доказано, что для логики предикатов не существует разрешающего алгоритма для проверки общезначимости (и, следовательно, невыполнимости) формул. Поэтому, запустив процесс логического вывода, в общем случае мы не можем быть уверены в том, что он завершится. Если R не является логическим следствием формул S , то алгоритм доказательства может никогда не завершиться при существовании бесконечного числа возможных резольвент, последовательно получаемых в процессе выполнения алгоритма доказательства. Однако для предикатов с конечной областью определения, и тем более в логике высказываний, не только положительный, но и отрицательный ответ на вопрос о невыполнимости логической функции всегда будет получен методом резолюции в конечное время (за конечное число шагов).

Логический вывод в ПРОЛОГЕ

Логика предикатов и понятие логического вывода были разработаны в первой половине прошлого века, но только в конце 60-х стали поняты огромные возможности логического вывода для построения непроцедурных алгоритмов; тогда же и были разработаны методы резолюции, алгоритм унификации и в конце концов язык логического программирования ПРОЛОГ. Основной вклад в логическое программирование был сделан Аланом Робинсоном (Alan Robinson), Алайном Колмерауером (Alain Colmerauer) и Робертом Ковальски (Robert Kowalski), причем он был сделан сравнительно недавно. В этом языке исходное множество формул, для которого ищется пустая резольвента, представляется в виде так называемых «дизьюнктов Хорна». Хорновские дизьюнкты — это формулы одного из трех типов:

отрицание: $\neg(B_1, \dots, B_n)$

факт: A

импликация (правило): $A \Leftarrow (B_1, \dots, B_m)$,

где A, B_1, \dots — литеры — атомные высказывания или предикаты с отрицаниями или без них в нормальной предваренной форме только с (подразумеваемыми) кванторами всеобщности для всех переменных. Очевидно, что любую логическую формулу можно привести к конъюнкции дизьюнктов Хорна. Действительно, факты можно рассматривать как импликации, не имеющие посылок (антecedентов). Отрицания — как импликации, не имеющие следствий (консеквентов). Поэтому все дизьюнкты Хорна — это формулы вида $A \Leftarrow (B_1, \dots, B_m)$, которые просто являются другой записью импликации $B_1 \& \dots \& B_m \Rightarrow A$, и знак \Leftarrow может читаться как «при условии, что». Итак, все эти формулы представляются в виде дизьюнктов: $\neg B_1 \vee \dots \vee \neg B_m \vee A$ или, что то же, в конъюнктивной нормальной форме. Любой дизьюнкт можно записать как множество литер $\{A, \neg B_1, \dots, \neg B_m\}$, где

знак дизъюнкции подразумевается. Именно к этим дизъюнктам и применяются последовательные шаги метода резолюции.

Таким образом, все задачи логического вывода можно формулировать, пользуясь только дизъюнктами Хорна, и все те задачи, которые являются в принципе разрешимыми, можно решить с помощью метода резолюции.

Рассмотрим пример из [6]. Пусть в нотации, близкой языку ПРОЛОГ, записана программа:

Программа_1::

- 1: птица(Х) \Leftarrow откладывает_яйца(Х). имеет_крылья(Х)
- 2: рептилия(Х) \Leftarrow откладывает_яйца(Х). имеет_чешую(Х)
- 3: откладывает_яйца(ворона)
- 4: откладывает_яйца(питон)
- 5: имеет_чешую(питон)
- 6: имеет_крылья(ворона)
- 7: ?птица(ворона)

В первой строке стоит правило, которое можно понять так: **любое животное является птицей при условии, что оно откладывает яйца и имеет крылья**. Очевидно, что это просто предикат в предваренной нормальной форме с опущенным квантором всеобщности (потому здесь и следует читать: **любое животное**). Этот предикат задан одним дизъюнктом (`птица(Х), -откладывает_яйца(Х), -имеет_крылья(Х)`), где атомные предикаты дизъюнкта просто перечисляются через запятую вместо того, чтобы перечисляться через знак дизъюнкции. Вторая строка — это правило, аналогично определяющее класс рептилий. Третья строка — «`откладывает_яйца(ворона)`» — это факт, который мы считаем истинным. Часто подобные факты присоединяются к программе из базы данных. Последняя строка — это утверждение-вопрос, истинность которого процессор языка ПРОЛОГ пытается проверить с помощью метода резолюции, пользуясь фактами и правилами.

Стратегии вывода

В любом не «игрушечном» логическом выводе мы сталкиваемся с «комбинаторным взрывом» числа возможных резолюций, поскольку общая база данных может содержать огромное множество фактов, не обязательно относящихся к данному конкретному вопросу, и построение **всех возможных резольвент** в этой базе данных бессмысленно. Поэтому при автоматизации логического вывода важнейшей является следующая проблема: какую пару дизъюнктов выбрать в качестве родительской для построения очередной резольвенты. Этот выбор должен выполняться так, чтобы пустая резольвента (если она есть) находилась скорейшим образом. Разработано несколько стратегий такого выбора, хотя общего метода, дающего оптимальное число шагов вывода для любой задачи, не существует. Например, в качестве одной из стратегий может применяться метод выбора на каждом шаге в

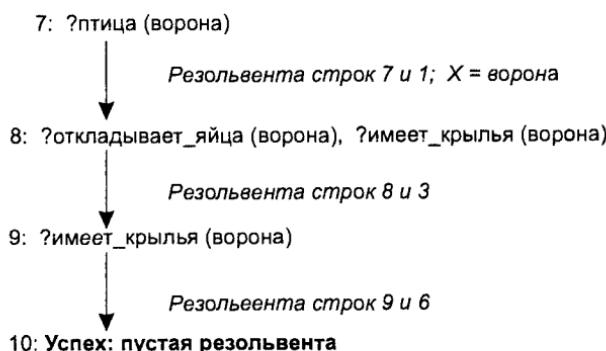
качестве одного из родительских дизъюнктоў такого, который содержит только один дизъюнктивный член.

Выполнение программы на ПРОЛОГЕ следует другой стратегии: *отрицание* вопроса программы принимается за цель. Далее вычисляются резольвенты, порождаемые **целью** (или каким-либо ее потомком) и **каким-либо правилом или фактом**, которые просматриваются последовательно сверху вниз. Если резольвента существует при наилучшей унификации, она вычисляется. Если пустая резольвента с помощью такой стратегии не найдена, то ответ на вопрос отрицателен. В нашем примере резольвентой утверждений:

`-птица(ворона) и
птица(X)←откладывает_яйца(X), имеет_крылья(X)`

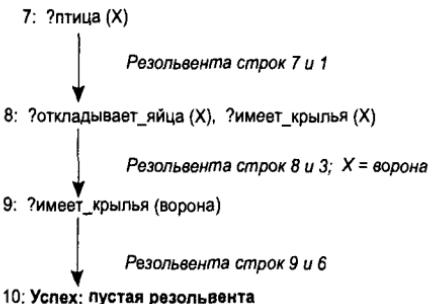
является дизъюнкт, включающий отрицания двух утверждений: `откладывает_яйца(X)` и `имеет_крылья(X)`. Эта пара становится новой целью, для которой снова ищется резольвента. Очевидно, что если в процессе вычислений найдена пустая резольвента, ответ на заданный вопрос утвердительный. Результатом программы на ПРОЛОГЕ являются также и значения переменных, конкретизированные алгоритмом унификации в процессе вычислений, — то есть те значения параметров, при которых справедливо заключение. В примерах далее будем использовать прописные буквы для обозначения переменных, а строчные буквы — для имен конкретных объектов универсума.

Рассмотрим вычисление Программы 1:



Получение пустой резольвенты означает успех вычисления. Фактически стоящий перед атомным предикатом знак вопроса означает вхождение этого предиката в дизъюнкт со знаком отрицания, и каждый такой дизъюнкт представляет собой очередную цель, которую нужно достичь (как бы проверить ее истинность).

Рассмотрим вычисление этой же программы с другой целью: `7: ?птица(X)`, что означает «Существует ли (произвольная) птица?» Пользователя обычно интересует не только сам факт успешного вычисления программы, но и конкретное значение переменной `X`, при котором это возможно:



Заметим, что здесь цель вычисления была достигнута при нахождении единственного примера, который ему удовлетворяет. В общем случае может быть необходимо найти все такие конкретизации, удовлетворяющие цели, то есть процессор ПРОЛОГа должен пытаться найти все возможные резольвенты-продолжения. Рассмотрим следующий пример.

Программа_2::

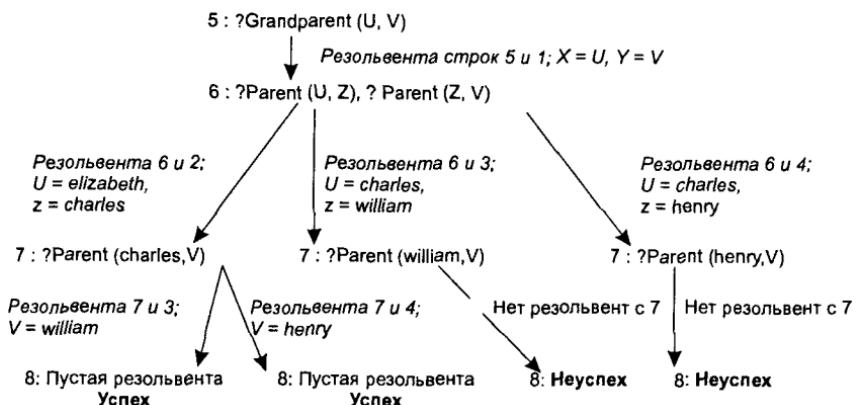
```

1: Grandparent(X,Y) ← Parent(X,Z). Parent(Z,Y)
2: Parent(elizabeth, charles)
3: Parent(charles, william)
4: Parent(charles, henry)
  
```

Содержательный смысл утверждений здесь очевиден. В качестве цели может быть выбрана любая из следующих:

- 5-a: ?Grandparent(elizabeth, henry):
- 5-b: ?Grandparent(elizabeth, V):
- 5-c: ?Grandparent(U, henry):
- 5-d: ?Grandparent(U, V).

Рассмотрим **все** возможные вычисления программы 2 при цели 5-d (то есть определим все возможные пары <бабушка (дедушка), внук (внучка)>, которые можно установить из информации, содержащейся в фактах 2, 3, 4):



В результате работы программы мы получаем значения пар: $\langle U = elizabeth, V = william \rangle$ по одному успешному пути вычислений и $\langle U = elizabeth, V = henry \rangle$ по другому. Итак, эта программа выдает следующий результат: «Элизабет является бабушкой как Уильяма, так и Генри». Дополнительная информация, полученная в процессе вычислений: «Чарльз является их отцом».

В качестве последнего примера рассмотрим логическую программу сортировки списка. Будем считать, что если у списка L выделены начальный элемент (голова) H и весь остальной список T (хвост), то список L будем записывать как $L = H : T$.

Первый оператор логической программы сортировки списка определим как утверждение, которое определяет сортировку списка как результат подходящей вставки головы списка в отсортированный хвост списка:

1. $\text{Sort}(H:T, S) \Leftarrow \text{Sort}(T, L), \text{Insert}(H, L, S)$

Смысл этого утверждения следующий: «список S является результатом сортировки списка $H:T$, если L является результатом сортировки списка T и S есть результат вставки элемента H в подходящее место списка L ». Это определение справедливо для всех списков, кроме пустых; этот частный случай можно учесть указанием конкретного факта: пустой список уже отсортирован:

2. $\text{Sort}([], [])$

Теперь мы должны определить, что означает операция $\text{Insert}(X, L, S)$ вставки элемента X в отсортированный список L с получением отсортированного списка S . В случае если этот элемент X предшествует первому элементу списка L , то список S строится добавлением X в качестве новой головы L :

3. $\text{Insert}(X, H:T, X:H:T) \Leftarrow \text{Precedes}(X, H)$

Это утверждение имеет следующий смысл: «Если элемент X предшествует по порядку элементу H , то результатом вставки X в отсортированный список $H:T$ является отсортированный список $X:H:T$ ».

Если элемент H предшествует элементу X в списке $H:T$, то этот случай можно описать так:

4. $\text{Insert}(X, H:T, H:T1) \Leftarrow \text{Precedes}(H, X), \text{Insert}(X, T, T1)$

что можно интерпретировать так: «Результатом вставки X в отсортированный список $H:T$ является список $H:T1$, если H предшествует X и $T1$ является отсортированным списком, получаемым после вставки X в отсортированный список T ».

Кроме того, следует определить вставку элемента в пустой список:

5. $\text{Insert}(X, [], [X])$.

Таким образом, полная программа сортировки списка имеет пять операторов — утверждений:

Программа_3::

1. $\text{Sort}(H:T,S) \Leftarrow \text{Sort}(T,L), \text{Insert}(H,L,S)$

2. $\text{Sort}([], [])$
3. $\text{Insert}(X, H:T, X:H:T) \Leftarrow \text{Precedes}(X, H)$
4. $\text{Insert}(X, H:T, H:T1) \Leftarrow \text{Precedes}(H, X), \text{Insert}(X, T, T1)$
5. $\text{Insert}(X, [], [X])$.

Эту программу мы можем использовать многими различными способами, задавая различные цели. Например, "?Sort([dog cat pig], [cat dog pig])". При этой цели вычисление проверит, действительно ли список [cat dog pig] является отсортированной версией списка [dog cat pig]. При цели: "?Sort([dog cat pig], S)" вычислитель ПРОЛОГа поместит в переменную S отсортированный список [dog cat pig]. При цели: "?Sort(S, [cat dog pig])" в переменную типа список S будут подставляться все перестановки элементов отсортированного списка [dog cat pig].

Применение логического вывода для анализа схем

Язык ПРОЛОГ имеет множество применений. Рассмотрим в качестве простого примера его применения анализ логических схем [4]. На рис. 2.4 представлена логическая схема полусумматора — основного блока двоичных сумматоров.

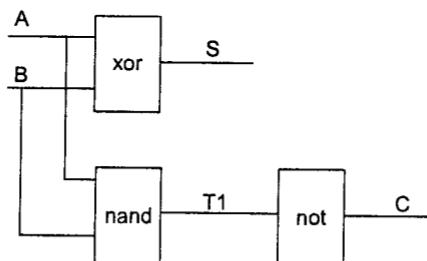


Рис. 2.4. Схема полусумматора

Описание этой схемы на ПРОЛОГе может быть представлено следующим образом:

`half_add(A, B, S, C) ← xor(A, B, S), nand(A, B, T1), not(T1, C).`

Необходима спецификация каждой элементарной функции (базовой подсхемы), включенной в это описание. Такая спецификация прямо повторяет соответствующие таблицы истинности:

`xor(X, X, 0).`

`xor(0, 1, 1).`

`xor(1, 0, 1).`

`nand(0, 0, 1).`

`nand(0, 1, 1).`

`nand(1, 0, 1).`

```
nand(1, 1, 0).
not(0, 1).
not(1, 0).
```

Целями ПРОЛОГа для этой программы могут быть, например, следующие:

```
? half_add(0, 0, S, C).
```

(Каковы выходы S и C этой схемы при входах 0,0? Ответ — $S = 0, C = 0.$)

```
? half_add(A, B, 0, 1).
```

(При каких входах выход S будет нулевым, а выход C — единичным? Ответы — $A = 0, B = 1; A = 1, B = 0.$)

```
? half_add(A, B, X, X).
```

(При каких входах схемы выходы S и C будут совпадать? Ответ — $A = 0, B = 0, X = 0.$)

```
? half_add(X, X, X, 1).
```

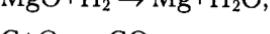
(Может ли быть у схемы выход C равным 1 при одинаковых значениях ее входов и выхода S ? Ответ — *но.*)

Приведенный пример показывает огромное разнообразие возможностей анализа логических схем с помощью ПРОЛОГа.

Экспертные системы

Возможности логического программирования выполнять логический вывод с получением нового знания на основе статического описания ситуации (состояния некоторой прикладной области реального мира) широко используется в разнообразных «экспертных системах» — системах поддержки принятия решений.

В качестве первого примера рассмотрим простейшую экспертную систему в области химического синтеза. Предположим, что мы можем выполнить следующие химические реакции:



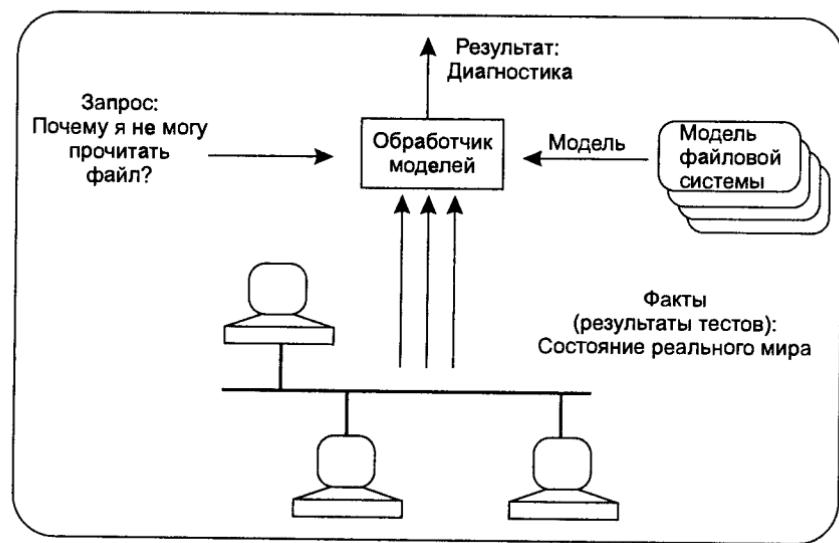
Можно ли получить (и в какой последовательности) H_2CO_3 , если мы имеем MgO , C , H_2 и O_2 ?

Решение этой проблемы можно найти с использованием логического вывода. Действительно, осуществимость первой реакции говорит о том, что «Если есть некоторое количество вещества CO_2 и некоторое количество вещества H_2O , то мы можем получить некоторое количество вещества H_2CO_3 ». Будем факт наличия исходного вещества CO_2 рассматривать как истинность утверждения CO_2 . Таким образом, любая возможная химическая реакция в этой интерпретации предстает как импликация, например, для описываемой реакции: $\text{CO}_2 \& \text{H}_2\text{O} \Rightarrow \text{H}_2\text{CO}_3$.

Очевидно, что для ответа на поставленный выше вопрос можно проверить выводимость утверждения H_2CO_3 при указанных посылках.

Пусть в базе данных находятся данные о всех возможных (в данной области) химических реакциях. Простую экспертную систему химического синтеза можно построить, объединяя эту базу данных с алгоритмом логического вывода.

Рассмотрим другой, более сложный пример экспертной системы — упрощенное описание одной из реально используемых на практике систем управления в компьютерных сетях и вычислительных системах Flipper, разработанной в Европейской исследовательской лаборатории фирмы Хьюлетт–Паккард [10]. Flipper позволяет описать сложную вычислительную систему таким образом, чтобы обеспечить принятие решения по управлению вычислительной системой в реальном времени в процессе ее функционирования.



Flipper выполняет свои функции на основе представления модели управляемой системы в виде логических правил, подобных записи правил в языке ПРОЛОГ. Эти правила фактически задают правила функционирования вычислительной системы. Например, правило, определяющее права доступа пользователя к файлу, имеет вид:

```

IF [User user] has [String mode] accessTo [File file]
[User user] isSuperUser;
OR [User user] isOwnerOf [File file],
[File file] ownerMode [mode];
  
```

OR

- `[user] isOwnerOf [file],`
`[file] isInGroupOf [user],`
`[file] groupMode [mode];`

OR

- `[user] isOwnerOf [file],`
- `[file] isInGroupOf [user],`
- `[file] worldMode [mode]`

Очевиден смысл этих правил: «Любой объект *user* класса User имеет право доступа *mode* класса String к объекту *file* класса File, ЕСЛИ этот объект *user* является привилегированным (SuperUser) ИЛИ если этот объект *user* является владельцем этого объекта *file* И тип доступа владельца к этому файлу совпадает с типом *mode*, ...». Весь фрагмент — это описание правила принадлежности троек объектов <Пользователь, Файл, Тип_доступа> отношению «Пользователь_ИмеетКФайлу_ПравоДоступа».

Проверка запроса о том, почему некоторый конкретный пользователь не имеет доступа к конкретному файлу с конкретным правом доступа, состоит в логическом выводе правильности запроса

? [Петров] has [update] accessTo [compiler.exe]

после того, как экспертная система затребует из реального мира актуальные на данный момент сведения по подлежащим проверке отношениям «Быть привилегированным пользователем», «Быть владельцем файла» и т. д. В данном примере факты, относящиеся к системе, могут со временем меняться (например, список привилегированных пользователей). Поэтому экспертная система должна принимать решение с учетом актуальности такой информации и при этом обеспечить приемлемую скорость работы. Например, можно не каждый раз обращаться за всеми фактами к реальным файлам, а установить срок старения фактов, при истечении которого база фактов должна актуализироваться.

Третий пример — это «система построения экспертных систем» EMYCIN, которая была использована в медицине, геологии, технике и других областях. Правила в EMYCIN также имеют форму ЕСЛИ <набор утверждений> ТО <заключение>. В этой системе можно с каждым утверждением связать коэффициенты уверенности, варьирующиеся от -1 (ложь) до +1 (истина). Например, правило EMYCIN, выраженное на естественном языке, может быть таким:

ЕСЛИ

материал, из которого изготовлена конструкция, входит в список Металлов;

И

Напряжение конструкции больше, чем 0,9

ТО

с уверенностью 0,5 есть основание полагать, что усталость является одним из факторов поведения конструкции под нагрузкой.

Введение степени уверенности, конечно, расширяет возможности экспертной системы. Однако не существует удовлетворительной простой модели, позволяющей проводить логический вывод с утверждениями, имеющими произвольную степень уверенности. В частности, как вычислить степень уверенности логической формулы при заданных степенях уверенности отдельных компонентов? При введении подобных расширений приходится такие правила изобретать самому (EMYCIN использует для этого специальные формулы), однако адекватность полученных результатов в таких системах требует дополнительного обоснования. Например, в EMYCIN часть ЕСЛИ-правила рассматривается как *истина*, когда соответствующий ей коэффициент уверенности больше некоторого порога (скажем, 0,2), и как *ложь*, когда коэффициент меньше другого порога (скажем, -0,2). Очевидно, что эти правила весьма субъективны и при использовании такой системы нужно проверять возможность применения ее результатов к реальной жизни. В то же время существуют впечатляющие примеры очень удачных применений экспертных систем, расширенных введением коэффициентов уверенности.

Контрольные задания

1. Какие перечисленные ниже логические формулы являются следствиями других формул?
 - $(a) xy$, (б) $x \Rightarrow \neg y$, (в) $\neg xy$, (г) $x \oplus \neg y$, (д) $\neg x \Rightarrow y$,
 - $(e) x \downarrow y$, (ж) $\neg(xy)$, (з) $x \vee y$, (и) True, (к) False.
2. Существует теорема геометрии о трех перпендикулярах: «Прямая, проведенная на плоскости перпендикулярно к проекции наклонной, перпендикулярна к самой наклонной». Эквивалентны ли этой теореме следующие утверждения (У1): «Прямая, проведенная на плоскости не перпендикулярно к наклонной, не перпендикулярна к ее проекции» и (У2): «Прямая, проведенная на плоскости перпендикулярно к наклонной, перпендикулярна к ее проекции»?

Указание. Выделить в каждом высказывании более простые высказывания, а именно: пусть высказывание (А) будет: «Прямая на плоскости перпендикулярна наклонной», высказывание (В) будет: «Прямая перпендикулярна проекции наклонной». Тогда структуру сложных высказываний — теоремы и утверждений У1 и У2 — можно представить логическими формулами: $A \Rightarrow B$, $\neg B \Rightarrow \neg A$ и $B \Rightarrow A$. Для решения задачи теперь достаточно проверить эквивалентность соответствующих формул.

3. Пусть установлено, что если сеть Петри инвариантна, то она ограничена. Что можно сказать о неинвариантной сети Петри? Что можно сказать о неограниченной сети Петри?

4. Известно, что если число делится на 6, оно делится на 3 и на 2. Сформулировать контрапозицию.

5. Можно ли выразить любую формулу логики высказываний только через высказывание «Если-то-иначе» и константы True, False?

Указание. Логическая функция «Если a то b иначе c » — это функция трех логических переменных, a , b и c . Она принимает значение b , если a истинно, и c в противном случае. Для решения задачи достаточно проверить, составляет ли набор перечисленных функций функциональный базис.

6. Пусть справедлива теорема: «Нильпотентный идеал является модулярным и радикальным». Нуждаются ли в доказательствах следующие утверждения: «Если идеал не нильпотентный, то он не радикальный» и «Если идеал не модулярен, то он не нильпотентный»?

7. Пусть справедлива теорема: «Для того чтобы сеть Петри была ограничена и жива, достаточно, чтобы она была согласована и инвариантна». Известно, что некоторая сеть Петри не ограничена. Что еще можно сказать о ней? Что можно сказать о неинвариантной сети Петри?

8. Предположим, что справедлива теорема: «Сеть Петри согласована и инвариантна, только если она жива и ограничена». Как доказывать эту теорему? Определить, какие из следующих утверждений являются следствиями этой теоремы:

- Сеть Петри ограничена только в случае, если она инвариантна;
- Несогласованность сети Петри является достаточным условием отсутствия у нее живости и ограниченности;
- Если сеть Петри не согласована, то она не может быть неживой, но ограниченной.

9. На какие этапы должно быть разбито доказательство утверждения: «Для того чтобы МП-автомат был детерминированным и ограниченным, достаточно, чтобы распознаваемый им язык относился к классу LR(k) или к классу LL(k)».

10. [29] Справедливы ли утверждения:

- Если верно, что дифференцируемая функция непрерывна, то невозможно, чтобы функция была дифференцируема и разрывна;
- Если верно, что невырожденная матрица имеет обратную, то также справедливо, что матрица либо вырождена, либо имеет обратную.

11. Пусть известно, что хроничные сепульки всегда латентны или бифуркальны. Какие из следующих утверждений в этом случае истинны:

- сепульки не хроничны только в случае отсутствия у них свойства латентности;
- латентность сепулек не является необходимым условием их хроничности или бифуркальности;
- сепульки бифуркальны только в случае их хроничности либо латентности;

- г) хроничность сепулек является достаточным условием их латентности или бифуркальности;
- д) для того чтобы сепульки были бифуркальны, достаточно только, чтобы они были хроничны;
- е) для нехроничности сепулек необходимо отсутствие у них как бифуркальности, так и латентности.

12. [34] Мистер МакГрегор, владелец лавки из Лондона, сообщил в Скотланд-Ярд, что его ограбили. По обвинению владельца лавки были арестованы три подозрительные личности А, В и С. На основании показаний Мак-Грегора, данных им под присягой, было установлено, что:

- а) Каждый из подозреваемых А, В и С в день ограбления был в лавке и никто туда больше не заходил.

Следующие факты были неопровергнуто установлены следствием:

- б) Если А виновен, то у него был ровно один сообщник;
- в) Если В невиновен, то С тоже невиновен;
- г) Если виновны ровно двое подозреваемых, то А – один из них;
- д) Если С не виновен, то В тоже не виновен.

Против кого Скотланд-Ярд выдвинул обвинение?

13. (Дело о рецидивистах.) Троє рецидивістів, А, В і С, підозрюються в преступлений. Неопровергнуто установлены следующие факты:

- (1) Если А виновен, а В невиновен, то в деле участвовал С;
- (2) С никогда не действует в одиночку;
- (3) А никогда не ходит на дело вместе с С;
- (4) Никто, кроме А, В и С в преступлении не замешан, но по крайней мере один из этой тройки виновен.

Можно ли на основании этих фактов выдвинуть обвинение против В? Против В или С? Против А?

14. (Дело о врунах.) Три школьника, А, В и С, вызваны к директору. В беседе с директором А утверждает, что В врет, В утверждает, что С врет, а С утверждает, что оба, А и В, врут. Что может заключить директор?

15. [34] В примере 2.6 инспектор Крейг поговорил с третьим обитателем больницы, Адамсом. На вопрос: «*Кто вы?*» Адамс ответил нечто такое, что дало Крейгу основание считать, что Адамс – сошедший с ума доктор. Что сказал Адамс?

16. [39] По обвинению в ограблении перед судом предстали А, В, С и D. Установлено следующее:

- (1) Если А и В оба виновны, то С был их соучастником;
- (2) Если А виновен, то по крайней мере один из двух, В и С, был его соучастником;
- (3) С всегда «ходит на дело» вместе с D;

(5) Если A не участвовал в ограблении, то там был D .

Какие выводы можно сделать отсюда? Можно ли отсюда заключить, что B виновен? Можно ли заключить, что виновны A либо D ?

17. Задача об опоздавшем автобусе ([16]). Даны следующие посылки:

- Если Билл поедет на автобусе, то он потеряет свое место, если автобус опаздывает.*
- Билл не сможет вернуться домой, если он потеряет свое место и будет чувствовать себя подавленным.*
- Если Билл не получит работу, то он будет чувствовать себя подавленным и не сможет вернуться домой.*

Какие из следующих предложений верны, если верны указанные выше посылки? Дайте доказательства истинных предложений и контрпримеры к остальным.

- Если Билл поедет на автобусе, то Билл потеряет работу, если автобус опаздывает.*
- Билл получит работу, если он потеряет свое место и сможет вернуться домой.*
- Если Билл не потеряет своего места, то он не сможет вернуться домой и не получит работы.*
- Билл будет чувствовать себя подавленным, если автобус опаздывает или он потеряет свое место.*

18. Записать в форме предиката утверждения:

«Если два объекта из M обладают свойством P , то они совпадают»;

«По крайней мере один студент решил все задачи»;

«Каждую задачу решил по крайней мере один студент».

19. [32] Один из афоризмов Козьмы Пруткова звучит так: «Нет столь великой вещи, которую не превзошла бы величиной еще большая; нет вещи столь малой, в которую не вместилась бы еще меньшая». Записать в форме предиката афоризм, используя атомный предикат $P(x, y)$: « x больше y ». Являются ли обе части афоризма тождественными с точки зрения передаваемой информации?

20. Рассмотрим определение.

Функция $f(x)$, определенная на множестве E , непрерывна в точке x_0 , если:

$$(\forall \varepsilon > 0)(\exists \delta > 0)(\forall x \in E)[(|x - x_0| < \delta) \Rightarrow |f(x) - f(x_0)| < \varepsilon].$$

Построить определение функции, которая не является непрерывной в точке x_0 .

21. Выразить с помощью предикатов:

- утверждение «Один и только один объект обладает свойством P » и его отрицание;

- 6) утверждение «По меньшей мере два объекта обладают свойством Р» и его отрицание.
22. Выразить с помощью предикатов следующие утверждения:
- Все элементы массива $b[j : k]$ нулевые.
 - Ни один элемент массива $b[j : k]$ не нулевой.
 - Некоторые элементы массива $b[j : k]$ нулевые.
 - Все нулевые элементы массива $b[0 : n-1]$ находятся в $b[j : k]$.
 - Значения массива $b[0 : n-1]$ расположены в возрастающем порядке.
23. [28] Определение предела числовой последовательности в словесной формулировке имеет вид: «Число A является пределом последовательности (a_n) , если и только если для всякого положительного числа ϵ существует такое число N , что для всякого n , большего N , $|a_n - A| < \epsilon$ ». В виде предиката это звучит так: $A = \lim a_n \Leftrightarrow (\forall \epsilon \in \mathbb{R}_+)(\exists N)(\forall n)[n > N \Rightarrow |a_n - A| < \epsilon]$. Получить необходимое и достаточное условие истинности утверждения «Число A не является пределом последовательности (a_n) », построив его в форме предиката, а потом сформулировать словесно.
24. Доказать методом резолюции, что если Капей бессмертен, то он не человек.
25. Доказать справедливость рассуждения: «Иван и Петр – братья. Братья имеют одну фамилию. Петр имеет фамилию Сидоров. Следовательно, Иван тоже имеет фамилию Сидоров».
- Указание.** Пусть предикаты: $B(x, y)$: « x и y – братья», $\Phi(x, f)$: « x имеет фамилию f ». Факты:
- $$F1::B(I, \Pi), F2::(\forall x)(\forall y)[B(x, y) \Rightarrow (\forall f)\Phi(x, f) \Leftrightarrow \Phi(y, f)], F3::\Phi(\Pi, C).$$
- Следствие, подлежащее доказательству: $R::\Phi(I, C)$.
26. Можно ли из следующей совокупности фактов:
- Марк был римлянином;
 - Цезарь был диктатором;
 - Те римляне, которые ненавидели диктатора, пытались убить его;
 - Римляне либо были преданы диктатору, либо ненавидели его;
 - Марк не был предан Цезарю
- вывести доказательство того, что Марк пытался убить Цезаря?
27. Проверить правильность рассуждения: «Никакой торговец сепульками сам их не покупает. Некоторые люди, покупающие сепульки, глупы. Следовательно, некоторые глупые люди не торгуют сепульками».
- Указание.** Пусть: $T(x)$: « x торгует сепульками», $\Pi(x)$: « x покупает сепульки», $\Gamma(x)$: « x – глупый человек». Схема рассуждения:
- $$F1::\neg(\exists x:T(x))\Pi(x), F2::(\exists x:\Pi(x))\Gamma(x), R::(\exists x:\Gamma(x))\neg T(x).$$

28. Выполнить вручную программу на языке ПРОЛОГ:

```
дедушка(Х, У) :- отец(Х, В), отец(В, У).
отец(Чарльз, Вильям)
отец(Джереми, Пол)
отец(Вильям, Генри)
отец(Чарльз, Смит)
отец(Смит, Майк)
отец(Смит, Джереми)
?дедушка(Чарльз, Z)
```

которая должна найти всех внуков Чарльза. Ответом является каждая конкретизация переменной Z, полученная алгоритмом унификации на путях построения пустой резольвенты.

29. Вычисление резольвент для программы на ПРОЛОГЕ следует определенной стратегии, при которой вычисляются резольвенты, порождаемые **целью или ее потомком и каким-либо правилом или фактом**, которые просматриваются последовательно сверху вниз. Можно ли утверждать, что ответ на вопрос программы отрицателен, если не найдена пустая резольвента, хотя очевидно, что при этой стратегии строятся не все возможные резольвенты.

30. Построить программу на языке ПРОЛОГ и провести ручную прокрутку ее для решения следующей проблемы: «*Известно, что каждый предок любит своего потомка. A – дедушка B, а B и C – братья. Доказать, что A любит C.*

31. Проанализируем объявление в газете «Экстра-Балт» № 31 от 15 августа 1996 года: «*Выношу благодарность доктору Сан-Ал-Мину за возвращенное мне счастье, дай Бог ему здоровья*». Если Сан-Ал-Мин действительно хороший доктор, то он может вылечить всех. Однако, судя по объявлению, сам себя доктор Сан-Ал-Мин вылечить не может: для его собственного здоровья ему необходима внешняя помощь. Формализуйте это рассуждение и докажите, что Сан-Ал-Мин – плохой доктор.

32. Выполните вручную программу З сортировки списка для цели ?Sort([23 5 7], S).

33. Выполните вручную программу ПРОЛОГА анализа схемы полусумматора для целей ?half_add(A,B,0,1), ?half_add(A, B, X, X), ?half_add(X, X, X, 1).

34. [26] Являются ли правильными следующие рассуждения:

«*Если Джонс не встречал этой ночью Смита, то либо Джонс был убийцей, либо Джонс лжет. Если Смит не был убийцей, то Джонс не встречал Смита этой ночью, и убийство имело место после полуночи. Если же убийство имело место после полуночи, то либо Смит был убийцей, либо Джонс лжет. Следовательно, Смит был убийцей.*

«*Если капиталовложения останутся постоянными, то вырастут правительственные расходы или возникнет безработица. Если расходы правительства не возрастут, то налоги будут снижены. Если налоги будут снижены и капита-*

ловложения останутся постоянными, то безработица не вырастет. Следовательно, расходы правительства возрастут».

35. [20] Проверьте, являются ли правильными следующие рассуждения (возможно, что в них отсутствуют некоторые необходимые посылки):

- «Если философ — дуалист, то он не материалист. Если он не материалист, то он диалектик или метафизик. Гегель не метафизик. Следовательно, Гегель — диалектик или дуалист».
- «Некоторые водные животные не являются рыбами, поскольку эти животные — теплокровные».
- «Все металлы — кристаллические вещества, поскольку ни одно кристаллическое вещество не является пластичным и ни один металл не пластичен».
- «Все дельфины — киты. Ни одна рыба не является дельфином, потому что ни одна рыба не является китом».

36. Доказать, что кисть руки является частью тела человека исходя из того, что рука является частью тела человека, а кисть руки — часть руки, формализовав транзитивность отношения «быть частью».

Указание. Пусть $\mathcal{C}(x, y)$ предикат, истинный тогда, когда x является частью y . Свойство транзитивности записывается так:

$$(\forall x)(\forall y)(\forall z)[\mathcal{C}(x, y) \& \mathcal{C}(y, z) \Rightarrow \mathcal{C}(x, z)].$$

ГЛАВА 3 Конечные автоматы

Из выступлений на семинаре «Software 2000: a View of the Future»
10 апреля 1994 года.

Brian Randell. Я помню Дуга Росса из компании SoftTech, много лет назад говорившего, что 80 или даже 90 % информатики (Computer Science) будет в будущем основываться на теории конечных автоматов.

Herve Gallaire. Я знаю людей из «Боинга», занимающихся системами стабилизации самолетов с использованием чистой теории автоматов. Даже трудно себе представить, что им удалось сделать с помощью этой теории.

Brian Randell. Большая часть теории автоматов была успешно использована в системных программах и текстовых фильтрах в OS UNIX. Это позволяет множеству людей работать на высоком уровне и разрабатывать очень эффективные программы.

Цель главы — представление концепции конечного автомата, который является простейшей моделью вычислительного устройства. Хотя теория конечных автоматов изучает очень простые модели, она является фундаментом большого числа разнообразных приложений. Эти приложения — от языковых процессоров до систем управления реального времени и протоколов связи — покрывают значительную долю систем, разработкой, реализацией и анализом которых занимается информатика.

В результате изучения материала этой главы читатель должен усвоить:

- ❑ общее представление о конечноавтоматных преобразователях информации и их свойствах, методах их задания, особенностях двух основных типов конечноавтоматных преобразователей: автоматах Мили и Мура;
- ❑ проблемы минимизации и проверки эквивалентности конечных автоматов;

- методы реализации конечноавтоматных преобразователей;
- примеры применения КА в различных областях и методы графического формализма конечноавтоматного представления систем обработки информации;
- автоматные модели параллельных процессов.

Автоматное преобразование информации

Не все окружающие нас преобразователи информации выполняют функциональное отображение информации. Результат преобразования $\text{вход} \Rightarrow \text{выход}$ зачастую зависит не только от того, какая информация в данный момент появилась на входе, но и от того, что происходило раньше, от предыстории преобразования. Множество примеров тому дают биологические системы. Например, один и тот же вход — извинение соседа после того, как он наступил вам на ногу в переполненном трамвае — вызовет у вас одну реакцию в первый раз и совсем другую — в пятый раз. Понятно, что ваша реакция будет различной, она будет зависеть от предыдущих событий, произошедших в трамвае, от входной истории.

Таким образом, существуют более сложные, *не функциональные* преобразователи информации; их реакция зависит не только от входа в данный момент, но и от того, что было на входе раньше, от входной истории. Такие преобразователи называются *автоматами*.

Число возможных входных историй бесконечно (счетно), даже если различных элементов входной информации у автомата конечное число (как в конечном функциональном преобразователе). Если автомат по-разному будет себя вести для каждой возможной предыстории, то такой «бесконечный» автомат должен иметь бесконечный ресурс — память, чтобы все эти предыстории как-то запоминать. Введем на множестве предысторий отношение эквивалентности. Две предыстории будем считать эквивалентными, если они одинаковым образом влияют на дальнейшее поведение автомата. Очевидно, что для своего функционирования автомат должен не обязательно запоминать входные истории, достаточно, чтобы автомат запоминал класс эквивалентности, к которому принадлежит данная история.

Случай, когда количество классов эквивалентных входных историй конечно, является простейшим, и именно он вызвал значительный интерес и имеет очень широкие применения. Соответствующая формальная модель называется конечным автоматным преобразователем информации, или просто конечным автоматом.

Определение 3.1. (*Внутренним*) состоянием автомата назовем класс эквивалентности его входных историй.

В своих состояниях автомат запоминает свое «концентрированное прошлое». Неформально состояние системы — это ее характеристика, однозначно определяющая ее дальнейшее поведение, все последующие реакции системы на внешние события. На один и тот же входной сигнал конечный автомат может реагировать по-разному, в зависимости от того, в каком состоянии он находится в данный момент.

Поскольку состояние представляет собой класс эквивалентных предысторий входов, состояние может измениться только при приходе очередного входного сигнала. При получении входного сигнала конечный автомат не только выдает информацию на выход как функцию этого входного сигнала и текущего состояния, но и меняет свое состояние, поскольку входной сигнал изменяет предысторию. Функционирование автомата удобно представлять графически. На рис. 3.1 блок памяти автомата хранит информацию о текущем состоянии S , которое вместе с входным сигналом X определяет выходную реакцию автомата Y и следующее состояние S' .

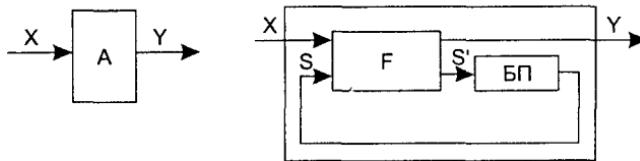


Рис. 3.1. Автоматный преобразователь и его реализация

Пример 3.1

Опишем поведение родителя, отправившего сына в школу. Сын приносит двойки и пятерки. Отец не хочет хвататься за ремень каждый раз, как только сын получит очередную двойку, и выбирает более тонкую тактику воспитания. Задавать автомат удобно графом, в котором вершины соответствуют состояниям, а ребро из состояния s в состояние q , помеченное x/y , проводится тогда, когда автомат из состояния s под воздействием входного сигнала x переходит в состояние q с выходной реакцией y . Граф автомата, моделирующего умное поведение родителя, представлен на рис. 3.2.

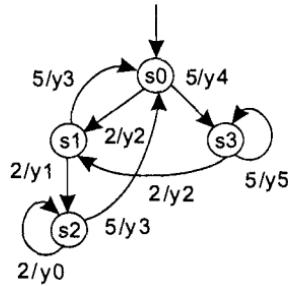


Рис. 3.2. Автомат, описывающий поведение «умного» отца

Этот автомат имеет четыре состояния $\{s0, s1, s2, s3\}$ и два входных сигнала — оценки, полученные сыном в школе: {2, 5}. Начиная с начального состояния $s0$ (оно помечено входной стрелкой), автомат под воздействием входных сигналов переходит из одного состояния в другое и выдает выходные сигналы — реакции на входы. Выходы автомата $\{y0, \dots, y5\}$ будем интерпретировать как действия родителя так:

у0: — брать ремень;

у1: — ругать сына;

у2: — успокаивать сына;

у3: — надеяться;

у4: — радоваться;

у5: — ликовать.

Сына, получившего одну и ту же оценку — двойку, ожидает дома совершенно различная реакция отца в зависимости от предыстории его учебы. Отец помнит, как его сын учился раньше, и строит свое воспитание с учетом его предыдущих успехов и неудач. Например, после третьей двойки в истории 2, 2, 2 сына встретят ремнем, а в истории 2, 2, 5, 2 — будут успокаивать. Каждая предыстория определяет текущее состояние автомата, при этом некоторые входные предыстории эквивалентны (именно те, которые приводят автомат в одно и то же состояние): история 2, 2, 5 эквивалентна пустой истории, которой соответствует начальное состояние.

Текущее состояние автомата представляет все то, что автомат знает о прошлом с точки зрения его будущего поведения — реакций на последующие входы. Эта история в концентрированном виде определена текущим состоянием, и все будущее поведение автомата, как реакция его на последующие входные сигналы, определено *именно текущим состоянием, но не тем, как автомат пришел в него*.

Итак, конечный автомат — это устройство, работающее в дискретные моменты времени (такты). На вход конечного автомата в каждом такте поступает один из возможных входных сигналов, а на его выходе появляется выходной сигнал, являющийся функцией его текущего состояния и поступившего входного сигнала. Внутреннее состояние автомата также меняется. Моменты срабатывания (такты) определяются либо принудительно тактирующими синхросигналами, либо асинхронно, наступлением внешнего события — прихода сигнала.

Определим конечный автомат формально.

Определение 3.2. Конечным автоматом Мили называется шестерка объектов:
 $A = \langle S, X, Y, s_0, \delta, \lambda \rangle$, где:

S — конечное непустое множество (состояний);

X — конечное непустое множество входных сигналов (входной алфавит);

Y — конечное непустое множество выходных сигналов (выходной алфавит);

$s_0 \in S$ — начальное состояние;

$\delta : S \times X \rightarrow S$ — функция переходов;

$\lambda : S \times X \rightarrow Y$ — функция выходов.

Кроме графического представления для автомата можно использовать и табличное, задавая функции переходов и выходов в виде таблиц. Автомат примера 3.1 будет представлен следующими таблицами.

Таблица 3.1

а)

δ	2	5
s0	s1	s3
s1	s2	s0
s2	s2	s0
s3	s1	s3

б)

λ	2	5
s0	y2	y4
s1	y1	y3
s2	y0	y3
s3	y2	y5

Таблица 3.1, а определяет функцию переходов δ так: $\delta(s0, 2) = s1; \delta(s2, 5) = s0\dots$, а табл. 3.1, б определяет функцию выходов λ : $\lambda(s0, 2) = y2; \lambda(s2, 5) = y3; \dots$.

Реализация КА

Рассмотрим два вида реализации КА: программную и аппаратную.

Программную реализацию можно выполнить на любом языке высокого уровня разными способами. На рис. 3.3 представлена блок-схема программы, реализующей поведение автомата примера 3.1. Нетрудно увидеть, что топология блок-схемы программы повторяет топологию графа переходов конечного автомата. С каждым состоянием связана операция NEXT, выполняющая функцию ожидания очередного события прихода нового входного сигнала и чтение его в некоторый стандартный буфер x , а также последующий анализ того, какой это сигнал. В зависимости от того, какой сигнал пришел на вход, выполняется та или иная функция $y0-y5$ и происходит переход к новому состоянию. Построив эту программу и добавив активные устройства, реализующие отдельные выходные операции (бит-ремнем, произносить ругательную или успокаивающую речь и т. д.), можно воспитание своего сына поручить компьютеру.

Аппаратная реализация требует построения устройств памяти для запоминания текущего состояния автомата. Обычно на практике используют двоичные элементы памяти (триггеры), запоминающие значение только одного двоичного разряда. Функциональный блок автомата реализуется как конечный функциональный преобразователь. Таким образом, общий подход к аппаратной реализации конечного автомата таков:

- входные и выходные сигналы и внутренние состояния автомата кодируются двоичными кодами;

- по таблицам переходов и выходов составляются кодированные таблицы переходов и выходов — фактически табличное задание отображения F (см. рис. 3.2);
- по кодированным таблицам переходов и выходов проводится минимизация двоичных функций, и они реализуются в заданном базисе;
- решаются схемотехнические вопросы синхронизации — привязки моментов выдачи выходного сигнала и изменения состояния внутренней памяти к моментам поступления входных сигналов на вход автомата.

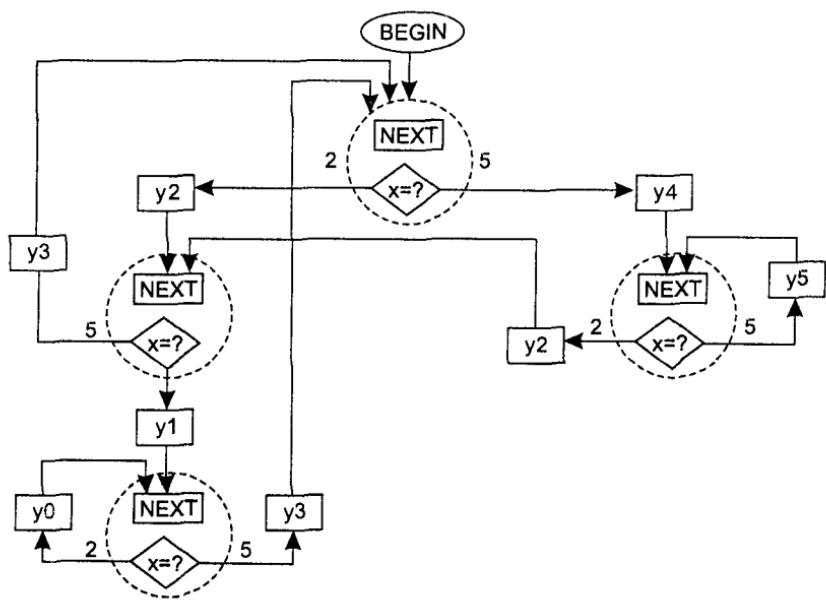


Рис. 3.3. Схема программы, реализующей поведение автомата примера 3.1

Рассмотрим реализацию автомата примера 3.1. Входных сигналов два; мы их закодируем так: «2» \leftrightarrow $<0>$, «5» \leftrightarrow $<1>$. Выходных сигналов (операций) шесть. Закодируем их « y_0 » \leftrightarrow $<000>$; « y_1 » \leftrightarrow $<001>$; ..., « y_5 » \leftrightarrow $<101>$. Внутренних состояний у автомата четыре. Закодируем их « s_0 » \leftrightarrow $<00>$; « s_1 » \leftrightarrow $<01>$; « s_2 » \leftrightarrow $<10>$; « s_3 » \leftrightarrow $<11>$. Структурная схема этого автомата после двоичного кодирования имеет следующий вид:

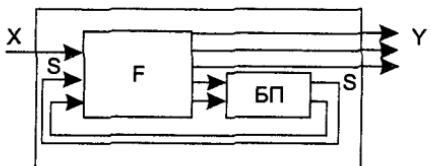


Таблица 3.2 — это кодированная таблица переходов и выходов автомата. Один двоичный разряд x кодирует два входных сигнала, пары двоичных разрядов q_1, q_2 ;

Q_1, Q_2 кодируют соответственно текущее и следующее состояния, разряды z_1, z_2, z_3 кодируют выходной сигнал.

Таблица 3.2

x	q1	q2	Q1	Q2	z1	z2	z3
0	0	0	0	1	0	1	0
0	0	1	1	0	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	1	0
1	0	0	1	1	1	0	0
1	0	1	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	0	1

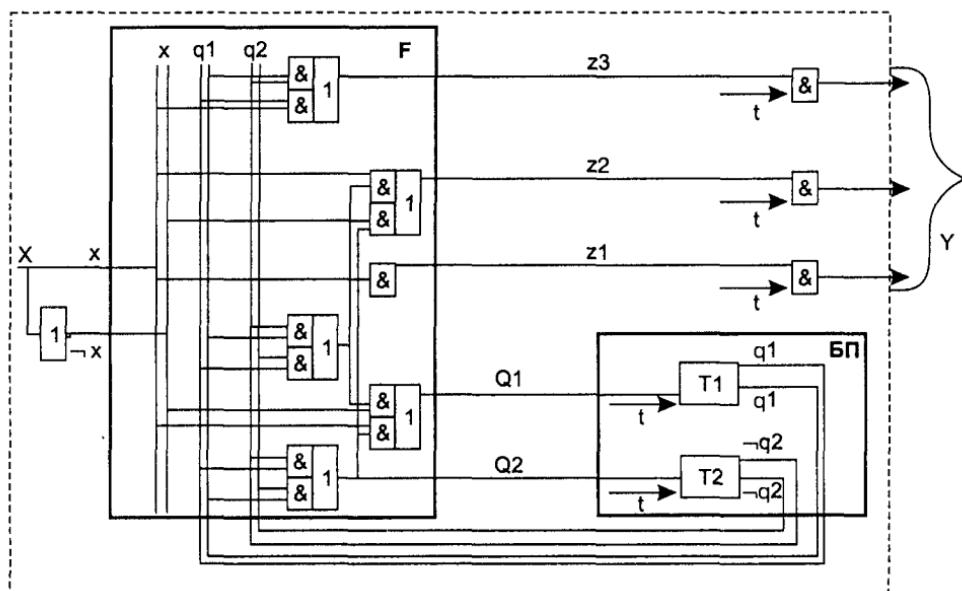


Рис. 3.4. Функциональная схема, реализующая автомат примера 3.1.
Выделены функциональный блок и блок памяти автомата.
Семантика тактового импульса t: «наступило событие
на входе автомата»

После минимизации в классе ДНФ получим аналитические выражения для всех двоичных функций, реализация которых показана на рис. 3.4. Блоки T1 и T2 – триггеры, которые запоминают двоичный сигнал до прихода следующего. Вход t в триггере – синхронизация вход, разрешающий переключение триггера.

Сигнал на этом входе должен появляться в момент наступления события получения автоматом очередного входного сигнала от окружения. Этот же синхро-сигнал обеспечивает получение на выходе импульсного значения выходного сигнала Y как реакцию на поступление сигнала на вход автомата.

Рисунок 3.4 дает схемную реализацию устройства, реализующего функционирование автомата примера 3.1. Если вход X автомата соединить с устройством считываия очередной оценки (два варианта этой оценки кодируются 0 и 1), а выход Y после дешифрирования соединить с исполнительными устройствами, выполняющими, например, награждение сына, его наказание, включающими проигрывание похвалы или укоризны, то на эту схему можно возложить воспитание сына в соответствии с приведенным «умным» алгоритмом, принимающим во внимание историю его учебы.

Часто как входные, так и выходные сигналы автомата кодируются не произвольным образом: их кодировка обычно предопределена конкретным применением автомата. В то же время кодирование *внутренних состояний* автомата на логике его функционирования никак не сказывается (при любом кодировании состояний автомат будет реализовывать то же отображение входных последовательностей на выходные). Однако различное кодирование может влиять на надежность устройства, скорость его переключения, простоту реализации логического блока и т. д. До сих пор ведутся исследования по проблеме оптимального кодирования состояний конечного автомата при различных критериях оптимальности. Например, в докладе [9], представленном на конференции в Женеве, обсуждалась проблема такого кодирования состояний автомата, которое дает реализацию с минимальной рассеиваемой мощностью в элементах памяти. Для этого на определенном типе входных последовательностей учитывается частота переключения между парами всех состояний и наиболее часто переключаемые пары состояний кодируются «ближкими» по Хеммингу кодами. Это в среднем уменьшает число переключаемых триггеров при работе автомата.

Эквивалентность КА: теорема Мура

Две булевые функции Φ и F эквивалентны, если на всех возможных интерпретациях они принимают одинаковые значения. Поскольку число интерпретаций у булевых функций конечно, то, перебрав их все, можно проверить, эквивалентны ли Φ и F . Если число интерпретаций велико, можно привести Φ и F к нормальной форме и сравнить их представления. Для проверки эквивалентности Φ и F можно также проверить аналитически, будет ли общезначима функция $\Phi \equiv F$.

Иная ситуация с конечными автоматами. Два конечных автомата эквивалентны, если реализуемые ими отображения вход-выход эквивалентны. Конечный автомат реализует отображение бесконечного множества входных последовательностей сигналов в бесконечное множество выходных последовательностей сигналов. Поэтому автоматные отображения нельзя сравнить простым перечислением их значений на всех возможных аргументах. Дадим несколько определений.

Расширим функции перехода и выхода автомата так, чтобы они были определены на множествах последовательностей (цепочках) сигналов входного алфавита. Пусть Σ — алфавит (конечное множество символов) и Σ^* — множество цепочек из элементов Σ . Будем обозначать ϵ пустую цепочку, вовсе не содержащую символов, а $^\wedge$ — операцию конкатенации (склеивания) цепочек. Так, $aab^\wedge ba = aabba$. Знак операции конкатенации часто опускают. Цепочки будем обозначать малыми греческими буквами $\alpha, \beta, \gamma, \dots$ Очевидно, ϵ является как левой, так и правой единицей для операции конкатенации: $\alpha\epsilon = \epsilon\alpha = \alpha$.

Определение 3.3. Пусть $A = \langle S, X, Y, s_0, \delta, \lambda \rangle$ — конечный автомат. Расширенными функциями перехода и выхода автомата A называются функции $\delta^*: S \times X^* \rightarrow S$ и $\lambda^*: S \times X^* \rightarrow Y^*$, определенные так:

$$\delta^*(s, \epsilon) = s; \delta^*(s, a\alpha) = \delta^*(\delta(s, a), \alpha);$$

$$\lambda^*(s, \epsilon) = \epsilon; \lambda^*(s, a\alpha) = \lambda(s, a)^\wedge \lambda^*(\delta(s, a), \alpha).$$

Расширенные функции переходов и выходов определены на множестве входных последовательностей (входных цепочках) в отличие от обычных функций переходов и выходов, которые определены на множестве входных сигналов.

Пусть в некоторое состояние автомата не существует пути из начального состояния. Иными словами, в эти состояния автомат не может попасть. Такие состояния автомата называются недостижимыми, остальные — достижимыми. Очевидно, что недостижимые состояния и переходы из них можно отбросить: они не влияют на поведение конечного автомата. Дадим формальное определение.

Определение 3.4. Пусть $A = \langle S, X, Y, s_0, \delta, \lambda \rangle$ — конечный автомат. Состояние $s \in S$ называется **достижимым** тогда и только тогда, когда $(\exists a \in X^*) \delta^*(s_0, a) = s$ (то есть под воздействием какой-либо цепочки входных сигналов автомат попадает в это состояние). Состояние конечного автомата является **недостижимым** тогда и только тогда, когда под воздействием любой входной цепочки автомат не переходит в это состояние: Недостижимо (s) $\Leftrightarrow (\forall a \in X^*) \delta^*(s_0, a) \neq s$.

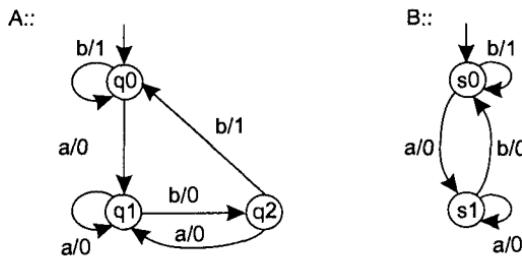


Рис. 3.5. Два конечных автомата

Достижимое множество состояний конечного автомата $A = \langle S, X, Y, s_0, \delta, \lambda \rangle$ строится с помощью алгоритма, основанного на индукции. Алгоритм разобьем на последовательные шаги. На i -м шаге будем строить множество Q_i состояний, достижимых из начального состояния автомата некоторой входной цепочкой дли-

ны не более чем i . Очевидно, что $Q_0 = \{s_0\}$. Для любого i очевидно определение $Q_{i+1} = Q_i \cup (\bigcup_{s \in Q_i} \bigcup_{x \in X} \delta(s, x))$. Очевидно также, что не более чем за $|S|$ шагов $Q_{k+1} = Q_k$. Это множество состояний Q_k будет включать в себя все достижимые состояния автомата $A = \langle S, X, Y, s_0, \delta, \lambda \rangle$.

Вернемся теперь к проблеме эквивалентности конечных автоматов.

Определение 3.5. Конечные автоматы $A = \langle S_A, X_A, Y_A, s_{0A}, \delta_A, \lambda_A \rangle$ и $B = \langle S_B, X_B, Y_B, s_{0B}, \delta_B, \lambda_B \rangle$ называются эквивалентными, если выполнены два условия:

- их входные алфавиты совпадают: $X_A = X_B = X$;
- реализуемые ими отображения совпадают: $(\forall \alpha \in X^*) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha)$.

Пример 3.2

Два конечных автомата (см. рис. 3.5) имеют разное число состояний и эти состояния по-разному называются. Но внешнее поведение у них похожее: реакция первого автомата на входную цепочку $aabbabb$ равна $\lambda_A^*(q_0, aabbabb) = \lambda_B^*(s_0, aabbabb) = 0001001$.

Однако ответить на вопрос, эквивалентны ли эти автоматы, перебором их реакций на всех входные цепочки невозможно: входных цепочек бесконечно много. Поэтому проблема эквивалентности конечных автоматов является нетривиальной.

Оказывается, существует простой метод решения этой проблемы. Этот метод основан на понятии прямого произведения конечных автоматов.

Определение 3.6. Прямыми произведениями конечных автоматов $A = \langle S_A, X, Y_A, s_{0A}, \delta_A, \lambda_A \rangle$ и $B = \langle S_B, X, Y_B, s_{0B}, \delta_B, \lambda_B \rangle$ с одинаковым входным алфавитом X (обозначается $A \times B$) называется автомат $A \times B = \langle S_A \times S_B, X, Y_A \times Y_B, (s_{0A}, s_{0B}), \delta_{A \times B}, \lambda_{A \times B} \rangle$, где:

- $(\forall s_A \in S_A)(\forall s_B \in S_B)(\forall x \in X) \delta_{A \times B}((s_A, s_B), x) = (\delta_A(s_A, x), \delta_B(s_B, x))$;
- $(\forall s_A \in S_A)(\forall s_B \in S_B)(\forall x \in X) \lambda_{A \times B}((s_A, s_B), x) = (\lambda_A(s_A, x), \lambda_B(s_B, x))$.

Иными словами, конечный автомат, являющийся прямым произведением двух конечных автоматов, в качестве своих состояний имеет пары состояний исходных автоматов, его начальное состояние есть пара их начальных состояний, выходной алфавит — множество пар выходных символов автоматов-множителей, а функции переходов и выходов определены покомпонентно. Таким образом, прямое произведение конечных автоматов — это просто два стоящих рядом невзаимодействующих конечных автомата, синхронно работающих на одном общем входе (рис. 3.6).

Теорема 3.1. (Теорема Мура.) Два конечных автомата $A = \langle S_A, X, Y_A, s_{0A}, \delta_A, \lambda_A \rangle$ и $B = \langle S_B, X, Y_B, s_{0B}, \delta_B, \lambda_B \rangle$ с одинаковым входным алфавитом являются эквивалентными тогда и только тогда, когда для любого достижимого состояния (s_A, s_B) в их прямом произведении $A \times B$ справедливо: $(\forall x \in X) \lambda_A(s_A, x) = \lambda_B(s_B, x)$.

Доказательство. (Необходимость.) Пусть A и B эквивалентны, то есть

$$(\forall \alpha \in X^*) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha).$$

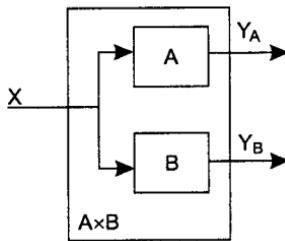


Рис. 3.6. Прямое произведение конечных автоматов А и В

Докажем при этом предположении:

$$(\forall (s_A, s_B) \in S_A \times S_B) \text{ Достижимо } (s_A, s_B) \Rightarrow (\forall x \in X) \lambda_A(s_A, x) = \lambda_B(s_B, x).$$

В соответствии с определением 3.4, свойство достижимости состояния (s_A, s_B) эквивалентно условию $(\exists \alpha \in X^*) \delta^*((s_{0A}, s_{0B}) \alpha) = (s_A, s_B)$. Таким образом, надо доказать, что если

$$(\forall \alpha \in X^*) \lambda_A^*(s_{0B}, \alpha) = \lambda_B^*(s_{0B}, \alpha),$$

то

$$(\forall (s_A, s_B) \in S_A \times S_B) [(\exists \beta \in X^*) \delta^*((s_{0A}, s_{0B}), \beta) = (s_A, s_B) \Rightarrow (\forall x \in X) \lambda_A(s_A, x) = \lambda_B(s_B, x)].$$

Предположим противное, то есть что

$$\neg(\forall (s_A, s_B) \in S_A \times S_B) [(\exists \beta \in X^*) \delta^*((s_{0A}, s_{0B}), \beta) = (s_A, s_B) \Rightarrow (\forall x \in X) \lambda_A(s_A, x) = \lambda_B(s_B, x)],$$

и покажем, что тогда

$$\neg(\forall \alpha \in X^*) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha).$$

Отрицание следствия можно преобразовать в

$$(\exists (s_A, s_B) \in S_A \times S_B) [(\exists \beta \in X^*) \delta^*((s_{0A}, s_{0B}), \beta) = (s_A, s_B) \& (\exists x \in X) \lambda_A(s_A, x) \neq \lambda_B(s_B, x)].$$

Положим $\alpha = \beta x$. Очевидно, что:

$$\begin{aligned} \lambda_A^*(s_{0A}, \alpha) &= \lambda_A^*(s_{0A}, \beta x) = \lambda_A^*(s_{0A}, \beta) \wedge \lambda_A(\delta^*(s_{0A}, \beta), x) = \lambda_A^*(s_{0A}, \beta) \wedge \lambda_A(s_A, x) \neq \\ &\neq \lambda_B^*(s_{0B}, \beta) \wedge \lambda_B(s_B, x) = \lambda_B^*(s_{0B}, \beta) \wedge \lambda_B(\delta^*(s_{0B}, \beta), x) = \lambda_B^*(s_{0B}, \beta x) = \lambda_B^*(s_{0B}, \alpha). \end{aligned}$$

Отсюда:

$$(\exists \alpha \in X^*) \lambda_A^*(s_{0A}, \alpha) \neq \lambda_B^*(s_{0B}, \alpha) \text{ или } \neg(\forall \alpha \in X^*) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha),$$

что и требовалось доказать.

(Достаточность.) Докажем обратное, то есть в предположении

$$(\forall (s_A, s_B) \in S_A \times S_B) \text{ Достижимо } (s_A, s_B) \Rightarrow (\forall x \in X) \lambda_A(s_A, x) = \lambda_B(s_B, x)$$

докажем, что А и В эквивалентны, то есть $(\forall \alpha \in X^*) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha)$.

Это доказывается индукцией по длине входных цепочек. Иными словами, будем доказывать

$$(\forall k \in \mathbb{N})(\forall \alpha \in X^k) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha) \text{ для } k = 0, 1, \dots .$$

Базой индукции является доказательство этого утверждения при $k = 0$. Оно доказывается непосредственно подстановкой $k = 0$, поскольку $\lambda_A^*(s_{0A}, \epsilon) = \lambda_B^*(s_{0B}, \epsilon) = \epsilon$ при входной цепочке ϵ длиной 0.

Шаг индукции. Пусть для некоторого i выполняется

$$(\forall \alpha \in X^i) \lambda_A^*(s_{0A}, \alpha) = \lambda_B^*(s_{0B}, \alpha).$$

Докажем, что это выполнится и для $i+1$. Пусть α — произвольная цепочка длиной i . Поскольку состояние $\delta_A^*(s_{0A}, \alpha)$ в автомате А и состояние $\delta_B^*(s_{0B}, \alpha)$ в автомате В являются достижимыми, из предположения

$$(\forall (s_A, s_B) \in S_A \times S_B) \text{ Достижимо } (s_A, s_B) \Rightarrow (\forall x \in X) \lambda_A(s_A, x) = \lambda_B(s_B, x)$$

следует: $(\forall x \in X) \lambda_A(\delta_A^*(s_{0A}, \alpha)x) = \lambda_B(\delta_B^*(s_{0B}, \alpha), x)$.

$$\text{Отсюда: } (\forall x \in X) (\forall \alpha \in X^i) \lambda_A^*(s_{0A}, \alpha x) = \lambda_B^*(s_{0B}, \alpha x).$$

Но при произвольном $\alpha \in X^i$ и произвольном $x \in X$ очевидно, что αx — произвольная цепочка из X^{i+1} . Теорема доказана.

Пример 3.2 (продолжение)

Прямое произведение конечных автоматов А и В, изображенных на рис. 3.5, представлено на рис. 3.7, а. На рис. 3.7, б показан этот же автомат с выброшенными недостижимыми состояниями. По графу переходов видно, что из всех достижимых состояний под воздействием входных сигналов автомат $A \times B$ выдает пары одинаковых выходных сигналов. Следовательно, автоматы А и В эквивалентны.

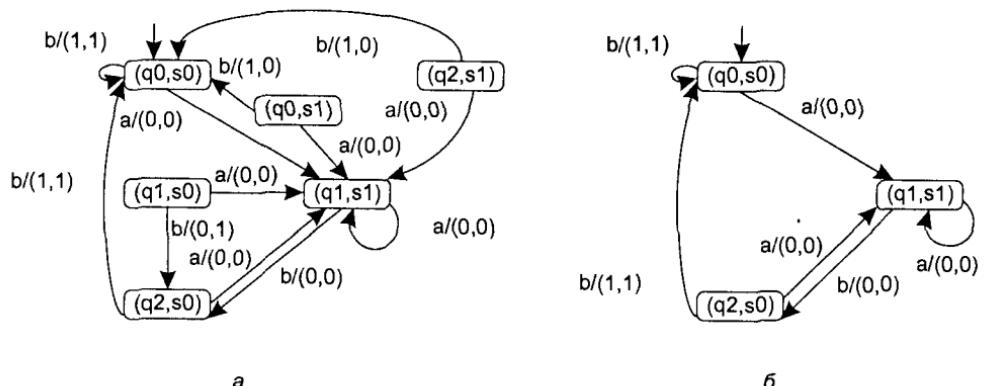


Рис. 3.7. Прямое произведение конечных автоматов примера 3.2

Минимизация КА

Из примера 3.2 видно, что разные автоматы могут функционировать одинаково, даже если у них разное число состояний. Важной задачей является нахождение минимального автомата, который реализует заданное автоматное отображение.

Эквивалентными естественно назвать два состояния автомата, которые нельзя различить никакими входными экспериментами.

Определение 3.7. Два состояния p и q конечного автомата $A = \langle S, X, Y, s_0, \delta, \lambda \rangle$ называются эквивалентными (обозначается $p \approx q$), если $(\forall \alpha \in X^*) \lambda^*(p, \alpha) = \lambda^*(q, \alpha)$.

Для автомата (рис. 3.8, а) состояния q_0 и q_2 эквивалентны: любая входная цепочка, поданная на автомат, находящийся в состоянии q_0 , даст такую же реакцию, как и в случае, когда автомат вначале находился в состоянии q_2 . Эквивалентные состояния можно объединить в один класс и построить новый автомат, состояниями которого являются классы эквивалентных состояний (см. рис. 3.8, б). Эквивалентные состояния объединены в классы, эти классы и являются состояниями нового автомата (рис. 3.8, б). Если мы можем определить на множестве состояний автомата «максимально возможное» разбиение на классы эквивалентности, то, выбирая его классы эквивалентности как новые состояния, получим минимальный автомат, эквивалентный исходному.

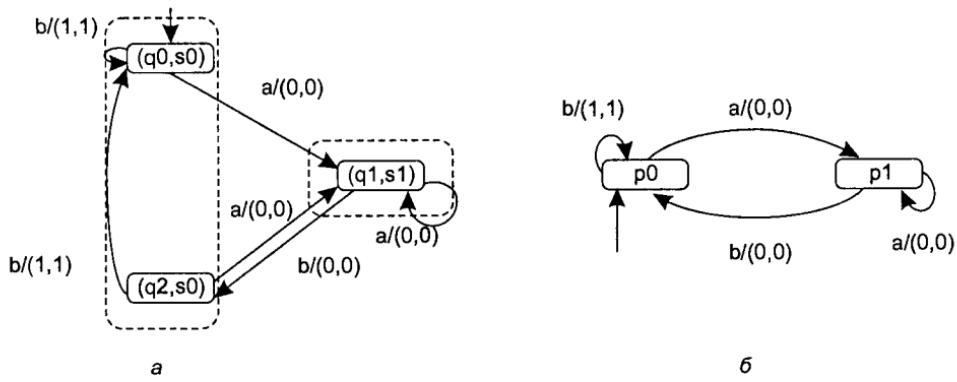


Рис. 3.8. Классы эквивалентных состояний
произведения конечных автоматов примера 3.2

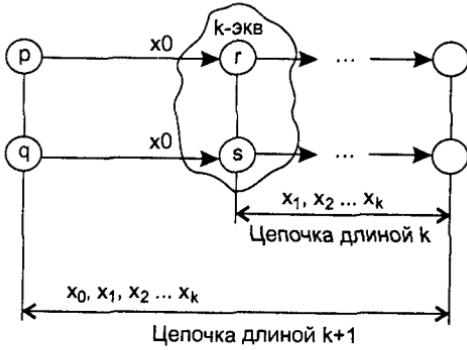
Определение 3.7 не является конструктивным: оно не дает нам процедуры выяснения того, являются ли два состояния эквивалентными, поскольку мы не можем перебрать все входные цепочки. Однако существует алгоритм определения максимального отношения эквивалентности на множестве состояний конечного автомата, который мы сейчас и рассмотрим.

Алгоритм состоит в последовательном построении на множестве состояний автомата A разбиений $\pi_0, \pi_1, \dots, \pi_\infty$, таких, что в один класс разбиения π_k попадают k -эквивалентные состояния, то есть те, которые неразличимы входными цепочками длиной k . Такие состояния будем считать находящимися в отношении эквивалентности \approx_k . Если $\neg(p \approx_k q)$, то p и q назовем k -различимыми. Из определения 3.7: $p \approx_k q \Leftrightarrow (\forall \alpha \in X^*, |\alpha| \leq k) \lambda^*(p, \alpha) = \lambda^*(q, \alpha)$. Очевидно, что в любом автомате все состояния 0-эквивалентны, поскольку при подаче пустой цепочки на вход автомата (цепочки длины 0) выходом является также пустая цепочка независимо от состоя-

ния, в котором автомат находится. Следующее разбиение π_1 также легко построить. Действительно, по определению в один блок π_1 попадают все состояния, в которых автомат одинаково реагирует на входные сигналы: $p \approx_1 q \Leftrightarrow (\forall x \in X) \lambda(p, x) = \lambda(q, x)$. Разбиения π_0 , содержащее один единственный блок, в который входят все состояния автомата, и π_1 , в каждом блоке которого собраны состояния, не различимые входными сигналами, являются исходными при построении цепочки разбиений $\pi_0, \pi_1, \dots, \pi_\infty$. Если мы сможем определить, как строить следующее разбиение из предыдущего, то начиная с π_1 , мы сможем последовательно построить и всю цепочку.

Теорема 3.2. Пусть $p \approx_k q, k > 1$. Для того чтобы $p \approx_{k+1} q$, необходимо и достаточно, чтобы $(\forall x \in X) \delta(p, x) \approx_k \delta(q, x)$. Иными словами, для того, чтобы два k -эквивалентных состояния конечного автомата были бы $k+1$ -эквивалентными, необходимо и достаточно, чтобы под воздействием любого входного сигнала автомат переходил из этих состояний в пару состояний, которые сами были бы k -эквивалентными.

Легко понять справедливость этой теоремы. Действительно, для того чтобы входная цепочка длины $k+1$, например цепочка $x_0x_1x_2\dots x_k$, не различала пару состояний p и q , нужно всего лишь, чтобы автомат из этих состояний переходил под воздействием x_0 в такие состояния, которые не различимы цепочкой $x_1x_2\dots x_k$, то есть чтобы $\delta(p, x_0)$ и $\delta(q, x_0)$ были бы k -неразличимы (см. рисунок).



Докажем теорему формально.

(Необходимость.) Нужно доказать $p \approx_{k+1} q \Rightarrow (\forall x \in X) \delta(p, x) \approx_k \delta(q, x)$. Докажем контрапозицию: $(\exists x \in X) [\neg \delta(p, x) \approx_k \delta(q, x)] \Rightarrow \neg p \approx_{k+1} q$. Если состояния $\delta(p, x)$ и $\delta(q, x)$, в которые попадает автомат из p и q под воздействием x , k -различимы, то пусть $x_1x_2\dots x_k$ — цепочка входных сигналов, их различающая. Тогда, очевидно, цепочка $x_0x_1x_2\dots x_k$ длиной $k+1$ отличает p и q .

(Достаточность.) Нужно доказать $(\forall x \in X) \delta(p, x) \approx_k \delta(q, x) \Rightarrow p \approx_{k+1} q$, если $p \approx_k q$ при $k > 1$. Из определения расширенной функции переходов $\lambda^*(p, x\alpha) = \lambda(p, x)\lambda^*(\delta(p, x)\alpha)$ и $\lambda^*(q, x\alpha) = \lambda(q, x)\lambda^*(\delta(q, x)\alpha)$.

Поскольку $k > 1$ и $p \approx_k q$, $\lambda(p, x) = \lambda(q, x)$. Из того, что $\delta(p, x)$ и $\delta(q, x)$ k -эквивалентны, следует, что $(\forall \alpha \in X^k) \lambda^*(\delta(p, x)\alpha) = \lambda^*(\delta(q, x)\alpha)$. Следовательно, для любых цепочек β длины $k+1$, $\lambda^*(p, \beta) = \lambda^*(q, \beta)$, то есть $p \approx_{k+1} q$, что и требовалось доказать.

Очевидно, что если p и q $k+1$ -эквивалентны, то они k -эквивалентны. Иными словами, блоки разбиения π_{k+1} являются подблоками разбиения π_k . Поскольку число состояний конечно, может быть только конечное число последовательно уменьшающихся разбиений π_k , начиная с максимального разбиения π_0 , содержащего один единственный блок. Более того, очевидно, что их не больше, чем число состояний автомата. Однако последовательное построение уменьшающихся разбиений π_i можно не продолжать дальше, как только два последовательных разбиения совпадают.

Теорема 3.3. Пусть $\pi_{k+1} = \pi_k$. Тогда для любого $i > k$, $\pi_i = \pi_k$.

Доказательство. Из доказанного выше следует, что:

$$(\forall p, q \in S) p \approx_{k+1} q \Leftrightarrow p \approx_k q \& (\forall x \in X) \delta(p, x) \approx_k \delta(q, x).$$

Обозначим $R(p, q, k) \equiv (\forall x \in X) \delta(p, x) \approx_k \delta(q, x)$. Тогда доказанное утверждение теоремы 3.2 запишется:

$$(\forall p, q \in S) p \approx_{k+1} q \Leftrightarrow p \approx_k q \& R(p, q, k).$$

Пусть теперь $p \approx_{r+1} q = p \approx_r q$. Тогда $(\forall p, q \in S) p \approx_r q \Leftrightarrow p \approx_r q \& R(p, q, r)$. Но это значит, что $(\forall p, q \in S) p \approx_r q \Rightarrow R(p, q, r)$.

Предположим теперь, что для некоторого $i > r$ $\approx_i = \approx_r$, но $\approx_{i+1} \neq \approx_i$. Это значит, что $\neg[(\forall p, q \in S) p \approx_i q \Rightarrow R(p, q, i)]$. Однако, поскольку $\approx_i = \approx_r$, а R зависит только от \approx_i , это противоречит утверждению $(\forall p, q \in S) p \approx_r q \Rightarrow R(p, q, r)$ и, следовательно, наше предположение неверно, что и требовалось доказать.

Пример 3.3

Минимизация конечного автомата, заданного таблицей переходов табл. 3.3, проводится последовательным построением разбиений π_0, π_1, \dots .

Таблица 3.3

					π_1		π_2		π_3	
	δ		λ		δ		δ		δ	
	a	b	a	b	a	b	a	b	a	b
1	3	6	1	0	A ₁	A ₁	C ₂	A ₂	D ₃	A ₃
2	4	8	0	1	B ₁	A ₁	B ₂	D ₂	C ₃	E ₃
3	1	4	1	0	A ₁	B ₁	A ₂	B ₂	A ₃	C ₃
4	7	9	0	1	B ₁	A ₁	B ₂	C ₂	C ₃	D ₃
5	9	1	1	0	A ₁	A ₁	C ₂	A ₂	D ₃	A ₃
6	3	5	1	0	A ₁	A ₁	C ₂	A ₂	D ₃	A ₃
7	4	3	0	1	B ₁	A ₁	B ₂	C ₂	C ₃	D ₃
8	4	2	1	0	B ₁	B ₁	B ₂	B ₂	C ₃	B ₃
9	5	7	1	0	A ₁	B ₁	A ₂	B ₂	A ₃	C ₃

Начальное разбиение представляет собой один блок, включающий все состояния, поскольку входные цепочки длиной 0 (пустая цепочка ϵ) не различают состояний: независимо от того, в каком состоянии автомат находился при подаче входа ϵ , выходом тоже будет ϵ . Поэтому

$$\pi_0 = \{A_0 = <1, 2, 3, 4, 5, 6, 7, 8, 9>\}.$$

Разбиение π_1 в один блок объединяет те состояния, которые нельзя различить при подаче цепочек длиной 1. Функция выходов λ при подаче a и b не может различить 1, 3, 5, 6, 8 и 9, поскольку для каждого из этих состояний при подаче на вход автомата a он выдает 1, а при подаче на вход b он выдает 0. Состояния 2, 4 и 7 попадают в другой блок, но между собой входной цепочкой длины 1 их различить нельзя. Поэтому:

$$\pi_1 = \{A_1 = <1, 3, 5, 6, 8, 9>; B_1 = <2, 4, 7>\}.$$

Следующее разбиение π_2 объединяет в один блок те состояния, которые нельзя различить при подаче цепочек длиной 2. Перебирать все такие цепочки долго, поэтому воспользуемся теоремой 3.2. В соответствии с ней, в один блок разбиения π_{k+1} попадут те состояния p и q , для которых справедливо $(\forall x \in X) \delta(p, x) \approx_k \delta(q, x)$. Как было установлено выше, эти состояния должны быть из одного блока предыдущего разбиения. Обратимся к построению π_2 . Построим таблицу переходов, но вместо значения состояния $\delta(p, x)$ будем писать номер блока разбиения π_1 , в которое попадает $\delta(p, x)$. Так, $\delta(3, a) = 1$, а это состояние находится в блоке A_1 . Аналогично, $\delta(3, b) = 4$, а это состояние находится в блоке B_1 .

После такого построения видно, что состояния 1, 3, 5, 6, 8 и 9 нужно разбить на три блока: $<1, 5, 6>$, $<3, 9>$ и $<8>$. Состояния 2, 4 и 7 попадают в одни и те же блоки предыдущего разбиения π_1 , поэтому они попадут в один и тот же блок разбиения π_2 . Итак:

$$\pi_2 = \{A_2 = <1, 5, 6>; B_2 = <2, 4, 7>; C_2 = <3, 9>; D_2 = <8>\}.$$

Аналогично строится π_3 . При его построении не нужно проверять, в какой блок π_2 будет переходить автомат из состояния 8, поскольку оно единственное в блоке разбиения π_2 , и поэтому далее дробиться этот блок не будет. Таким образом,

$$\pi_3 = \{A_3 = <1, 5, 6>; B_3 = <2>; C_3 = <4, 7>; D_3 = <3, 9>; E_3 = <8>\}.$$

Разбиение π_4 совпадает с разбиением π_3 . На основании теоремы 3.3 искомое разбиение π_∞ совпадает с π_3 . Итак, минимальный автомат с эквивалентным поведением имеет 5 состояний, представляющих блоки разбиения π_3 , а его функции переходов и выходов определяются так: $\delta(A_3, a) = D_3$, $\delta(A_3, b) = A_3$, $\lambda(A_3, a) = 1$ и т. д.

Автоматы Мили и Мура

Рассмотренная выше модель называется автоматом Мили. Автоматы Мура образуют другой класс моделей, с точки зрения вычислительной мощности полностью эквивалентный классу автоматов Мили. В автомате Мура $A = <S, X, Y, s_0, \delta, \lambda>$ выходная функция λ определяется не на паре $<\text{состояние}, \text{входной сигнал}>$,

а только на состоянии: $\lambda: S \rightarrow Y$. Пример конечного автомата Мура представлен на рис. 3.9, а. Здесь выход автомата определяется однозначно тем состоянием, в которое автомат переходит после приема входного сигнала. Например, в состояние s_1 можно прийти по трем переходам: из состояния s_0 под воздействием b , из состояния s_2 под воздействием b , из состояния s_1 под воздействием a . Во всех трех случаях выходная реакция автомата одна и та же: выходной сигнал y_2 . Очевидно, что по любому автомату Мура легко построить эквивалентный ему автомат Мили; для автомата Мура (рис. 3.9, а) эквивалентный ему автомат Мили изображен на рис. 3.9, б.

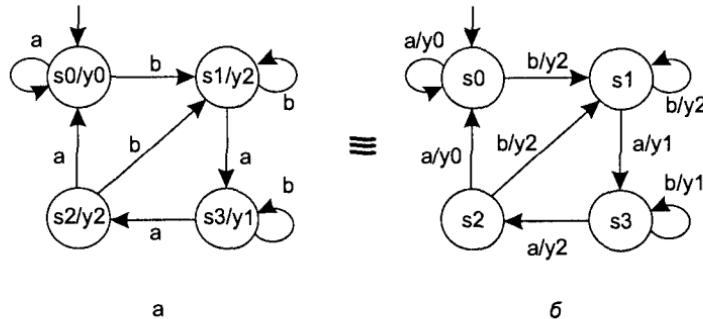


Рис. 3.9. Автомат Мура (а) и эквивалентный ему автомат Мили (б)

Не столь очевидно, что справедливо и обратное: для любого автомата Мили существует эквивалентный ему автомат Мура. Справедливость этого утверждения легко доказывается конструктивно. Рассмотрим рис. 3.10. Каждое состояние s автомата Мили (см. рис. 3.10, а) расщепляется на несколько эквивалентных состояний, с каждым из которых связывается один выходной символ. Для нашего примера это состояния $p_0, p_1, q_0, q_1, r_0, r_1$. Построение переходов эквивалентного автомата Мура ясно из рисунка.

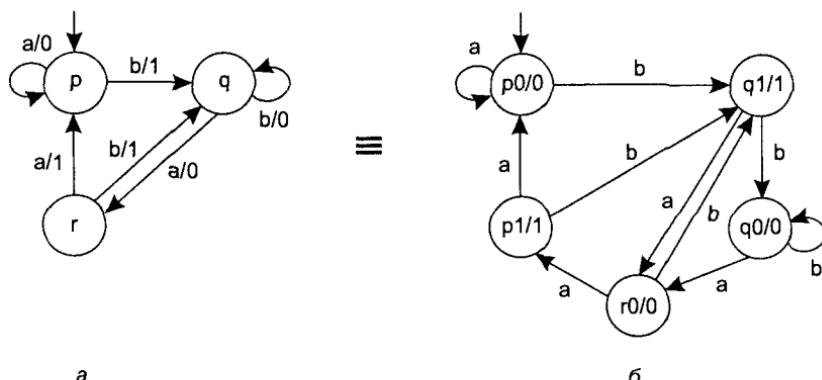
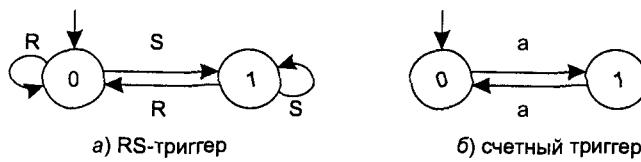


Рис. 3.10. Автомат Мили (а) и эквивалентный ему автомат Мура (б)

Примеры КА

Триггеры

Триггер является простейшим автоматом. Рассмотрим два типа триггеров: RS-триггер и счетный триггер. Состояние этих автоматов является их выходом, то есть это автоматы Мура. В RS-триггере два входа: *Reset* и *Set*. Вход *Reset* сбрасывает, а *Set* устанавливает единичное состояние автомата. В счетном триггере единственный счетный вход переключает автомат из нулевого состояния в единичное и обратно.



Электронные часы

Электронные часы самых разнообразных функциональных возможностей являются одним из наиболее широко применяемых в быту электронных приборов, управление которыми построено на основе конечноавтоматной модели. Электронные часы обычно показывают время, дату, дают возможность установки времени и даты, а также выполняют множество других функций (например, их можно превратить в секундомер со сбросом и остановкой его показаний, в будильник и т. д.). Управление всеми этими возможностями производится встроенным конечноавтоматным преобразователем, входами которого являются события нажатия внешних управляемых кнопок. Структурная схема электронных часов показана на рис. 3.11. Управляющие кнопки обозначены здесь «*а*» и «*б*». Кроме устройства отображения, высвечивающего цифры, и схемы отображения, преобразующей двоично-десятичные коды цифр в семиразрядный код управления светодиодами, на схеме показаны четыре регистра отображения, хранящие двоично-десятичные коды четырех цифр, которые в настоящий момент высвечиваются на циферблате с помощью схемы и устройства отображения, комбинационные схемы «ИЛИ», пропускающие любой из разрешенных кодов на регистры отображения, шина «Управление», разрешающая в каждой ситуации выдачу на регистры отображения сигналов только либо секундомера, либо часов, либо даты. На схеме также присутствуют регистры секундомера и генератор тиков, который выдает сигнал с частотой 1 Гц. На рисунке зафиксирован момент «19 июня, 15 часов, 04 минуты, 43 секунды».

Устройство управления, организующее работу всех элементов электронных часов, построено на основе модели конечного автомата. Граф переходов этого автомата изображен на рис. 3.12. В начальном состоянии отображается время. Это значит, что двоичный код этого состояния (после дешифрирования) открывает выходы

четырех двоично-десятичных регистров, хранящих единицы и десятки минут и единицы и десятки часов на входы четырех комбинационных схем «ИЛИ».

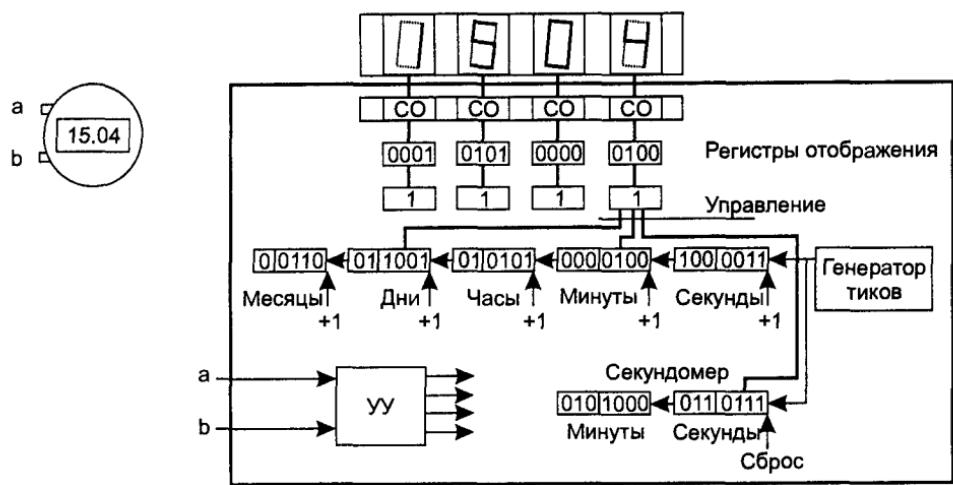


Рис. 3.11. Структурная схема электронных часов

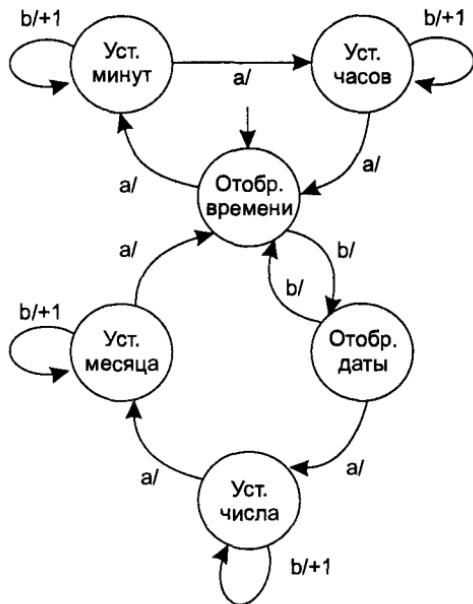


Рис. 3.12. Автомат устройства управления электронными часами

Конечный автомат реагирует на событие нажатия кнопки «*a*» на корпусе часов переходом в состояние «Установка минут», в котором событие нажатия кнопки

«b» вызовет увеличение числа, хранящегося в регистрах, отведенных для минут. При этом переносы из регистра секунд в регистр, отведенный под хранение числа, блокируются. Событие нажатия кнопки «b» в состоянии «Установка месяца» вызовет увеличение числа, хранящегося в регистрах, отведенных для месяца. На рис. 3.12 не показана возможность и алгоритм работы с секундомером.

Промышленность выпускает много типов электронных часов с различными функциональными возможностями. Схемы управления таких часов можно построить, имея навык реализации конечных функциональных преобразователей и построения конечноавтоматных моделей дискретных систем управления.

Схема управления микрокалькулятором

В качестве еще одного примера системы, управляемой событиями (event driven system), рассмотрим карманный микрокалькулятор. Структура его показана на рис. 3.13.

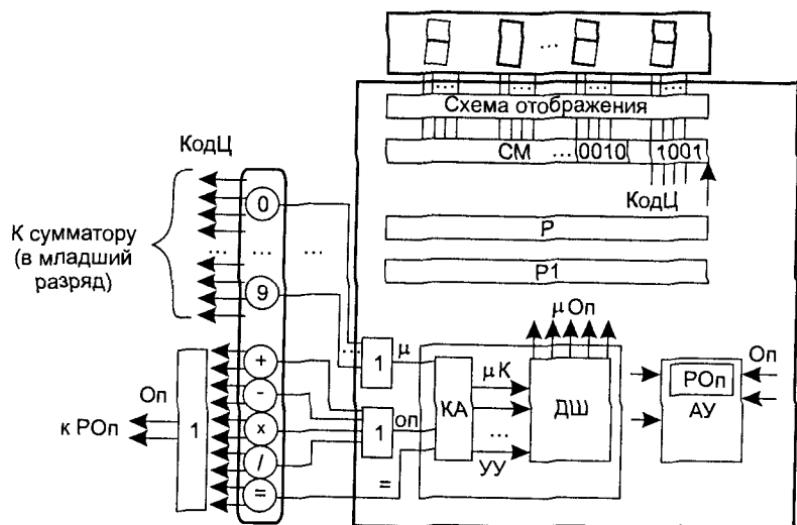


Рис. 3.13. Структурная схема микрокалькулятора

Мы будем рассматривать упрощенный микрокалькулятор с четырьмя арифметическими операциями. В структуре микрокалькулятора можно выделить наборное поле с кнопками для цифр, знаков операций и знака «=», а также устройство отображения результата. В корпусе микрокалькулятора расположены электронные блоки, обеспечивающие «разумную» реакцию микрокалькулятора на поток внешних дискретных событий. Такими событиями являются, конечно, нажатия тех или иных кнопок наборного поля. Состав этих блоков следующий:

- ❑ Схема отображения, преобразующая двоично-десятичные коды цифр в семиразрядные двоичные коды, связанные с подачей напряжения на сегменты светодиодов. Этую схему мы подробно рассматривали в главе 1.

- CM , или сумматор, или регистр результата. Двоично-десятичный код числа, помещенный сюда, высвечивается устройством отображения. В младший разряд сумматора поступают поочередно цифры каждого набираемого на клавиатуре числа, сюда же попадает результат выполнения операции;
- P и $P1$ – рабочие регистры для хранения промежуточных значений (например, первого слагаемого, когда набирается второе);
- AU – арифметическое устройство. По получении сигналов (одной из двух микроопераций) на вход выполняет арифметическую операцию, код которой занесен в регистр операций POp , над числами, помещенными в CM и регистре P . Первый operand операции может находиться в CM , а второй – в регистре P (одна микрооперация) или наоборот (вторая микрооперация) (см. далее).
- UU – устройство управления. Оно организует всю работу микрокалькулятора, выдавая в подходящие моменты элементарные микрокооперации (μOp) типа: «разрешить прием КодЦ (кода цифры) в младший разряд CM », или «сдвинуть содержимое CM на 4 разряда вправо», или «передать код числа из CM в $P1$ », или «очистить CM » и т. д. Микрокооперации появляются как результат дешифрирования дешифратором (ДШ) выходных сигналов управляющего автомата (КА). Выходные сигналы автомата соответствуют микрокомандам, они являются реакцией КА на очередное внешнее событие – нажатие некоторой кнопки на клавиатуре.

КА является дискретным преобразователем, на входе которого – три возможных события («ц», «оп», «==»), а на выходе – микрокоманды (группы микроопераций). Нашей задачей является построение функциональной схемы УУ. Очевидно, что одно и то же событие на входе вызывает разную реакцию КА в зависимости от предыстории. Например, при нажатии цифры либо добавление цифры в конец набираемого числа происходит без очистки сумматора, либо сумматор предварительно очищается. Поэтому с уверенностью можно сказать, что это устройство реализует автоматное преобразование вход-выход.

Конечный автомат в составе устройства управления, получая на вход дискретные события «ц», «оп» и «==», должен организовать согласованную работу всех блоков, то есть выполнение элементарных операций управления блоками в соответствии с наступившим событием и предысторией. Элементарные операции (или микрооперации, μOp) в микрокалькуляторе следующие.

Название	Обозначение	Комментарий
$\mu 0$	$0 \rightarrow CM$	Очистка сумматора
$\mu 1$	$0 \rightarrow P$	Очистка регистра
$\mu 2$	$CM < POp > P \rightarrow CM$	Выполнение операции, помещенной в РОп, первый operand – в сумматоре
$\mu 3$	$P < POp > CM \rightarrow CM$	Выполнение операции, помещенной в РОп, первый operand – в регистре Р

(продолжение)

Название	Обозначение	Комментарий
μ_4	$\text{«оп} \rightarrow \text{Роп}$	Прием операции с клавиатуры в РОп
μ_5	$\text{«ц} \rightarrow \text{СМ}_{1-4}$	Прием кода цифры с клавиатуры в 1—4 разряды СМ
μ_6	$\text{СМ}_{\leftarrow 4}$	Сдвиг сумматора на 4 разряда влево
μ_7	$\text{СМ} \rightarrow \text{P1}$	Межрегистровая пересылка информации
μ_8	$\text{P1} \rightarrow \text{P}$	Межрегистровая пересылка информации
μ_9	$\text{СМ} \rightarrow \text{P}$	Межрегистровая пересылка информации

При построении алгоритма управления микрокалькулятором нужно гарантировать, что реакция на такие стандартные последовательности, как «5 – 3 =» или даже «5+3*4 =» должна быть стандартна: в первом случае мы должны получить 2, а во втором 32 (выполняем операции последовательно, слева направо, без учета приоритетов). Однако существует множество возможностей реализации реакции калькулятора на не совсем стандартные входные цепочки событий, например «5 * = = = =» или «+1= = = = =». Первую цепочку можно считать ошибкой, а можно понимать, как возведение 5 в степень 4, а вторую — последовательное накопление единиц (например, при динамическом подсчете). Это, конечно, дает большую функциональность калькулятору и, следовательно, дополнительные удобства в его применении. Например, при подсчете единиц можно нажимать каждый раз только на одну клавишу «+=» вместо нажатия двух клавиш «+» и «1». Однако подобное увеличение функциональных возможностей требует некоторого усложнения алгоритма работы устройства управления. Выделим три состояния управляющего автомата — три класса эквивалентных предысторий (рис. 3.14).

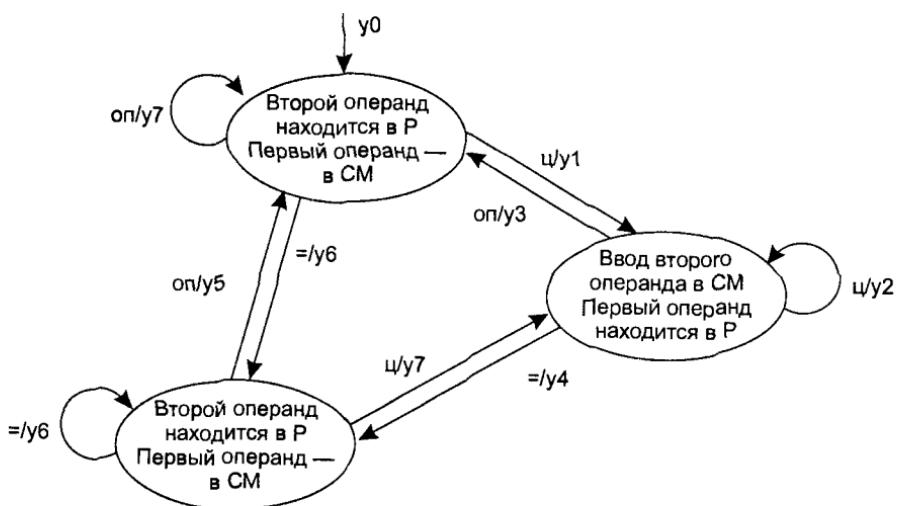


Рис. 3.14. Автомат устройства управления микрокалькулятором

В каждом состоянии автомат должен как-то реагировать на все три события — нажатия кнопок «оп» — операция, «ц» — любой цифры и «==» — знака равенства, поскольку пользователь калькулятора имеет право нажимать любые кнопки в любом порядке. Важно только, чтобы реакция автомата всегда была разумной. Действия на переходах автомата назовем микрокомандами (μK); каждая из микрокоманд может содержать несколько микроопераций (μOp):

y0: < $\mu 0, \mu 1>$ = < $0 \rightarrow CM, 0 \rightarrow P>$
y1: < $\mu 0, \mu 5>$ = < $0 \rightarrow CM, "ц" \rightarrow CM_{1-4}>$
y2: < $\mu 6, \mu 5>$ = < $CM_{1-4}, "ц" \rightarrow CM_{1-4}>$
y3: < $\mu 7, \mu 3, \mu 4, \mu 8>$ = < $CM \rightarrow P1, P < POp > CM \rightarrow CM, "оп" \rightarrow POp, P1 \rightarrow P>$
y4: < $\mu 7, \mu 3, \mu 8>$ = < $CM \rightarrow P1, P < POp > CM \rightarrow CM, P1 \rightarrow P>$
y5: < $\mu 4, \mu 9>$ = < $"оп" \rightarrow POp, CM \rightarrow P>$
y6: < $\mu 2>$ = < $CM < POp > P \rightarrow CM>$
y7: < $\mu 2, \mu 4>$ = < $CM < POp > P \rightarrow CM, "оп" \rightarrow POp>$

Понять работу автомата проще на примерах. После набора $<2, 4, /, 6>$ в регистре Р должно находиться число 24, в сумматоре — 6. Иными словами, первый операнд деления находится в Р. Если следующей будет нажата клавиша «==», то в сумматоре должен появиться результат деления (24/6), а находившееся в сумматоре число должно сохраниться в регистре Р. Иначе говоря, должна выполниться операция, запомненная в регистре операций, причем **первый ее операнд берется из вспомогательного регистра Р, а второй — из сумматора**. Если далее мы опять нажмем на «==», то это можно интерпретировать как повторение предыдущей операции над результатом, находящимся сейчас в сумматоре. Иными словами, будет выполняться арифметическая операция, запомненная в регистре операций, причем **первый ее операнд берется из сумматора, а второй — из вспомогательного регистра Р**. Подобным образом можно понять реакцию устройства управления микрокалькулятора на любую входную цепочку.

На рис. 3.15 представлена структура устройства управления микрокалькулятором с явным представлением дешифратора; импульсные выходы конечного автомата управляет через вентили включением микроопераций μOp (в случае необходимости, когда следующая микрооперация должна начаться после завершения предыдущей, используются элементы задержки τ).

Отметим, что приведенная в этом примере схема управления реализована в одном конкретном типе отечественных микрокалькуляторов «Электроника Д-38». Другие микрокалькуляторы могут по-иному реагировать на непримитивные цепочки входных событий. Например, при поступлении повторного знака операции предыдущая операция может не выполняться, а новая операция записываться в РОп, затирая предыдущую. Аналогично, несколько знаков «==» эквивалентны одному такому знаку. В таком микрокалькуляторе цепочка нажатий входных клавиш $<2, 4, -, 2, -, /, +, 2, =, =, =>$ даст в результате 24, в то время как в нашем микрокалькуляторе она даст в результате 17.

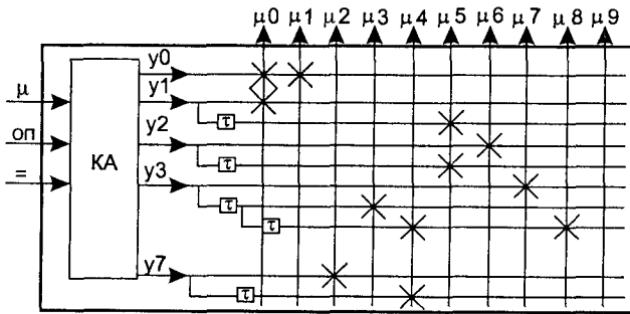


Рис. 3.15. Структура устройства управления микрокалькулятором

Команды операционной системы UNIX

Основные команды оболочки (shell) операционной системы UNIX оперируют на входах, которые рассматриваются унифицированно как последовательность (поток) символов, и результатом их работы также является поток символов, то есть выход имеет ту же стандартную структуру, что и вход. Такой подход позволяет проектировать программы, реализующие эти команды как конечные автоматы. Более того, это позволяет сделать все команды оболочки UNIX принципиально сочленяемыми: выход одной команды может непосредственно без промежуточного преобразования подаваться на вход другой. Например, $A|B$ в UNIX представляет композицию команд, и результат команды A является аргументом команды B. Каждая команда, обрабатывающая последовательности символов, в ОС UNIX реализуется как конечный автомат. Таким образом, несколько выполняемых подряд команд UNIX реализуются как последовательная композиция конечных автоматов.

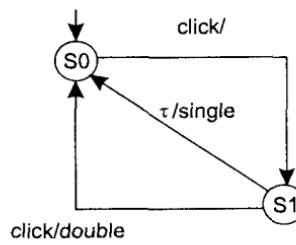
Реактивные системы

В последнее время для инженеров и исследователей, работающих в области информатики, стала ясна необходимость выделения особого класса систем, который был назван «реактивными», или «реагирующими» системами (reactive systems). Это системы, взаимодействующие с окружением и реагирующие на поток внешних событий. Как правило, реактивные системы состоят из нескольких подсистем, и взаимодействие подсистем является частью взаимодействия системы с ее окружением. Микрокалькулятор, электронные часы, система управления проходом в метро — все это реактивные системы. Конечноавтоматная модель является удобным средством описания таких систем: она обладает ясностью семантики (трактовки элементов модели), наглядностью и выразительностью, и в то же время достаточно строга и формальна. Однако классическое графическое представление конечных автоматов страдает рядом недостатков. Главным недостатком является отсутствие параметров, в том числе и с бесконечной областью определения, в частности отсутствие понятия времени. Другие недостатки — отсутствие иерархии состояний, обобщения переходов, средств выражения прерываний и продолжения

нормальной работы после их обработки. Кроме того, в классической модели не определена семантика взаимодействия конечных автоматов.

Простым расширением модели классического конечного автомата является введение понятия «внешнее событие», наступление которого можно считать условием перехода автомата из состояния в состояние. Такими событиями могут быть получение на вход автомата символа или же целого сообщения, прерывание, событие срабатывания таймера. С этим последним типом событий естественно связывается понятие времени в автомате. Действительно, введение понятия времени проще всего связать с ограничением времени пребывания автомата в конкретном состоянии и такое ограничение задать таймером. Событие срабатывания таймера вызовет переход автомата в другое состояние.

Рассмотрим в качестве примера задачу спецификации процесса, определяющего одинарный либо двойной щелчок мыши. Двойным щелчком считается пара нажатий на клавишу мыши, разделяемая менее чем $\tau = 250$ мс. На рисунке представлен график переходов автомата, решающего эту задачу. По первому щелчку мыши (*click*) автомат переходит в состояние S1, и если до истечения следующих $\tau = 250$ мс на вход автомата поступит еще один сигнал-событие *click*, то на выход будет выдан сигнал «*double*», в противном случае — сигнал «*single*».



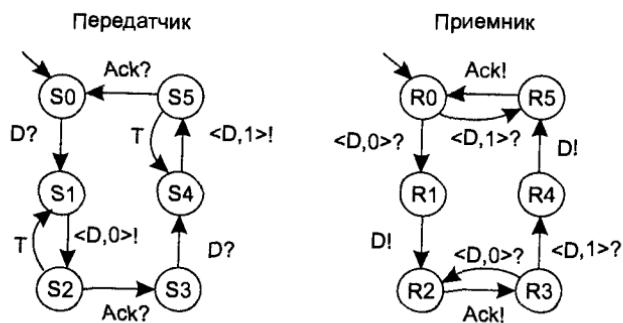
Протокол PAR передачи сообщений в сетях

В качестве более сложного примера конечноавтоматной спецификации реального процесса рассмотрим спецификацию протокола передачи информации в компьютерных сетях.

Термин «протокол» в информатике ведет свое происхождение не от милицейского, а от дипломатического протокола. Дипломаты так характеризуют слово **протокол**: «Общественные отношения обязывают уважать определенные правила, без соблюдения которых существование общества было бы невозможным и их отсутствие привело бы к хаосу и анархии... Протокол дает возможность каждому беспрепятственно выполнять свои обязанности. Кроме организации церемоний протокол определяет методы, рамки, поведение и этикет и устанавливает правила относительно официальной и личной переписки» [14]. Дипломатический протокол — это совокупность правил, регламентирующих взаимодействие независимых (субверенных) государств. Подобные правила должны быть выработаны в любом сообществе людей (и активностей любой природы), общающихся между собой. Даже простой телефонный разговор возможен только при соблюдении некоторых пра-

вил: например, если собеседники будут говорить одновременно, то они друг друга не услышат. Протокол вычислительной сети — это множество правил, регулирующих взаимодействие параллельно функционирующих объектов в сети с целью обеспечения их согласованного поведения при решении некоторой общей задачи. Представление таких правил с помощью модели конечных автоматов широко применяется для наглядной и строгой графической спецификации протоколов.

Рассмотрим классический протокол передачи сообщений «точка-точка», имеющий название PAR (Positive Acknowledge with Retransmission). Два процесса, передатчик и приемник, взаимодействуют через ненадежный канал, который может исказять и терять сообщения. Использование корректирующего кода позволяет все искажения сообщений либо исправлять, либо обнаруживать, и такие сообщения выбрасываются. Как передатчик, так и приемник специфицируются ниже в виде автомата (системы переходов).



На этом рисунке автомат-передатчик в начальном состоянии S0 ждет от пользователя порцию данных D и, получив ее, переходит в состояние S1. В этом состоянии передатчик передает данные в канал, нумеруя сообщение номером 0, и включает таймер T. В состоянии S2, если до срабатывания таймера придет подтверждение Ack, то передатчик перейдет в состояние S3, в противном случае при исчерпании тайм-аута он возвращается в состояние S1 и повторно передает сообщение с данными D, занумерованное 0. В состоянии S3 передатчик снова ожидает события ввода новых данных от пользователя и, получив их, переходит в состояние S4, передает в канал сообщение, занумерованное номером 1 с включением таймера. Далее, в S5 передатчик ожидает подтверждения от приемника о получении этого сообщения. Если подтверждение получено, то цикл передачи повторяется из состояния S0. Если до срабатывания таймера T, установленного при входе автомата в состояние S5, приемник не получит подтверждения о приеме предыдущего сообщения, это сообщение посыпается вновь.

Читателю предлагается самостоятельно проверить различные варианты (сценарии) работы двух взаимодействующих процессов, передатчика и приемника, при различных ситуациях потери как передаваемых сообщений, так и подтверждений, а также при различных задержках сообщений и подтверждений в канале и обнаружить ошибку — некорректность этого протокола.

Протокол выбора лидера в распределенной системе

Рассмотрим конечноавтоматное представление протокола выбора лидера в распределенной системе, приведенное впервые на 6-й Международной конференции по распределенным вычислительным системам в 1986 году [7].

Проблема выбора лидера, то есть одного-единственного «центрального» или «управляющего» процессора в распределенной системе процессоров возникает при построении отказоустойчивых применений вычислительных сетей, например, когда необходимо автоматически заменить отказавший центральный координатор в распределенной базе данных или координатор часов в различных процессорах. Здесь мы рассмотрим одно из возможных решений, при котором все процессоры выполняют один и тот же алгоритм выбора. Этот алгоритм был применен для построения распределенной системы ТЕМПО синхронизации часов в различных процессорах в операционной системе UNIX 4.3 в университете города Беркли. Распределенный синхронизатор основан на схеме *Хозян-Раб* (*Master-Slave*). Процессор-Хозян периодически запрашивает от всех процессоров их текущее время, вычисляет среднее значение текущего времени и посылает каждому Рабу то значение, на которое тот должен подвести свои часы.

Очевидно, что центральный процесс-координатор должен быть в системе единственным. Описываемый здесь протокол выбора лидера гарантирует, что один новый синхронизирующий процессор-Хозян будет выбран автоматически в случае отказа процессора, на котором работал прежний Хозян. Этот протокол также автоматически разрешает коллизии: ситуации, при которых по какой-либо причине в распределенной системе возникают два или больше хозяина. Процессоры связаны недежными каналами связи, которые могут терять,искажать и даже дублировать сообщения. Предполагается, что коммуникационная система позволяет обмениваться сообщениями любым двум процессорам, а также посыпать широковещательные сообщения (*broadcast*) от каждого процессора всем остальным. Любой процессор может отказать в любое время. Отказав, процессор ничего не делает (*fail-stop processor*), а после восстановления выполняет алгоритм выбора лидера с его начального состояния.

Когда Хозян активен, он, рассыпая синхронизирующие сообщения, периодически сбрасывает «таймер объявления выборов» (*ElectionTimeout*) у каждого Раба. Когда Хозян отказывает, тот Раб, чей таймер срабатывает первым, становится кандидатом на место Хозяина.

Алгоритм выбора лидера представлен в [7] в виде систем переходов — некоторого расширения конечноавтоматной модели. Переходы между состояниями здесь происходят либо по прибытии определенного типа сообщения, либо по срабатыванию таймера. Описание алгоритма, выполняемого каждым процессором, содержит восемь состояний (рис. 3.16).

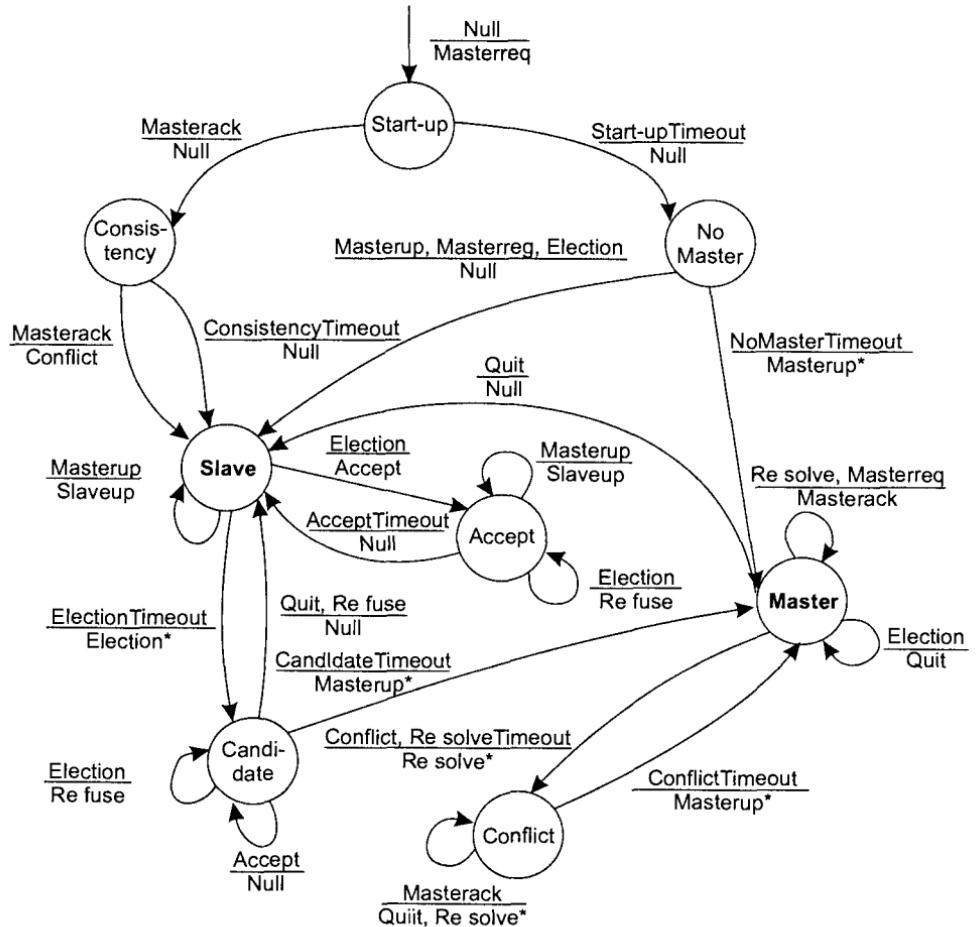


Рис. 3.16. Граф переходов процесса выбора лидера

На рисунке каждое широковещательное сообщение (то есть сообщение, рассылаемое всем процессам группы) помечено звездочкой (*). Дадим краткое описание одного процесса в распределенном алгоритме выбора лидера. Описание представлено действиями процесса в каждом состоянии.

- **Start-up.** Это начальное состояние. Сюда процесс переходит из состояния отказа после восстановления или при начальном присоединении к группе. При входе в это состояние процесс посылает широковещательное сообщение *Masterreq*, информируя **Хозяина** (если он есть) о появлении нового члена.
- **NoMaster.** В это состояние процесс переходит, когда он в состоянии **Start-up** не получил ответа от **Хозяина** в течение длительного времени и его таймер *Start-UpTimeout* срабатывает. На этой стадии своего выполнения процесс полагает, что **Хозяина** в системе не существует, поэтому он сам готовится стать **Хозяином**.

ном. В трех случаях, однако, он становится рабом, а именно: если он, находясь в этом состоянии до исчерпания *NoMasterTimeout* единиц времени, получает следующие сообщения: *Masterreq* от вновь стартующего процесса (что, по-видимому, является ошибкой, но так специфицировано в оригинальной статье), *Election* от раба, только что объявившего выборы, и сообщение *Masterup* от процесса-кандидата в *Хозяева*. Если ни одного из этих событий не происходит пока процесс находится в данном состоянии, то по истечении тайм-аута *NoMasterTimeout* процесс становится *Хозяином* и сам посыпает широковещательное сообщение *Masterup*, переходя в состояние **Master**.

- **Master.** Это состояние процесс достигает либо если он не обнаруживает *Хозяина* после своего старта (восстановления), либо если процесс выиграл выборы у других рабов после отказа *Хозяина*. В этом состоянии процесс периодически рассыпает синхронизирующее сообщение всем другим процессам, тем самым извещая их о том, что *Хозяин* жив (рассылка этих синхронизирующих сообщений здесь не показана).
- **Slave.** В этом состоянии *Раба* процесс при получении каждого синхронизирующего сообщения от *Хозяина* устанавливает таймер *ElectionTimeout* (это на графике не показано). Срабатывание этого таймера происходит, если долго не было синхронизирующего сообщения от *Хозяина*, что может свидетельствовать о том, что процесс *Хозяин* отказал (и, следовательно, надо выбирать нового лидера). В каждом процессе таймеры *ElectionTimeout* у различных *Рабов* устанавливаются случайно после каждого такта синхронизации, чтобы только один из них сработал первым при отсутствии синхронизирующего сообщения (то есть при смерти *Хозяина*).
- **Candidate.** В это состояние процесс переходит, когда срабатывает таймер *ElectionTimeout* у *Раба*. При переходе в это состояние он «объявляет выборы», предлагая себя в качестве кандидата. Для этого он широковещательно рассыпает сообщение *Election*. Процесс остается в состоянии Кандидата некоторое время, получая сообщения *Accept* от принявших его кандидатуру рабов. Каждое такое сообщение переустанавливает *CandidateTimeout* таймер. Если процесс при этом получает сообщение *Refuse*, он возвращается в состояние *Раба*. При срабатывании таймера *CandidateTimeout* процесс становится *Хозяином*, рассыпая всем сообщение *Masterup*, имеющее смысл: «Я объявляю себя Лидером».
- **Accept.** Если процесс, будучи *Рабом*, получает сообщение *Election*, он сразу соглашается на кандидатуру будущего нового хозяина, посыпая в ответ пославшему процессу сообщение *Accept*. В ответ на все последующие сообщения *Election* процесс отвечает сообщением *Refuse*, тем самым храня верность своему первому выбору. По исчерпании тайм-аута *AcceptTimeout* процесс возвращается снова в состояние *Раба*, имея нового выбранного *Хозяина*.
- **Conflict.** Процесс-*Хозяин* (то есть находящийся в состоянии **Master**) переходит в это состояние, если он получил сообщение *Conflict* или же его таймер *ResolveTimeout* исчерпался. В данном состоянии процесс борется с другими *Хозяевами*, которые могут возникнуть в системе, широковещательно рассыпая

сообщения *Resolve*. Любой конкурирующий с данным *Хозяин* будет убит (переведен в состояние *Раба*) широковещательно рассылаемым сообщением *Quit*. После срабатывания таймера *ConflictTimeout* процесс возвращается в состояние **Master**, считая конфликт разрешенным.

- **Consistency.** В это состояние вновь стартующий процесс попадает, получив сообщение *Masterack* от какого-то процесса, находящегося в состоянии **Master** и признающего данный процесс за своего *Хозяина*. Некоторое время (пока не истечет тайм-аут *ConsistencyTimeout*) процесс здесь поджидает, прислушиваясь, нет ли конфликта в системе. О конфликтной ситуации говорит второе сообщение *Masterack* от другого *Хозяина*, который пошлет его в ответ на широковещательное сообщение *Masterrecc*, посланное данным процессом при его старте. В случае обнаружения конфликта процесс хранит верность своему первому *Хозяину*: он посыпает тому предупреждение о наличии конфликта и переходит в состояние **Slave**.

Описанный протокол — весьма типичный представитель распределенных алгоритмов. Совместное поведение нескольких процессов в распределенном алгоритме весьма трудно понять даже в том случае, когда ясно поведение каждого процесса. Анализ поведения таких сложных систем проводится обычно с помощью моделирования. На 4-й Международной конференции по параллельным компьютерным технологиям в 1997 году [1] этот протокол был проанализирован с помощью моделирующей системы и в нем были найдены некоторые некорректности (например, было обнаружено, что в состоянии **NoMaster** специфицированная в протоколе реакция на сообщение *Masterreq* существенно ухудшает качество этого протокола).

Визуальный формализм представления моделей реактивных систем: Statecharts

Во многих приложениях обычная модель конечного автомата расширяется путем добавления переменных. Этот подход дает возможность объединить наглядность представления состояний и переходов между ними с возможностями языков программирования для описания условий и действий, связанных с переходами. Для одной из переменных с конечной областью определения может использоваться специальное название *state* — «состояние». Возможные состояния такого автомата характеризуются наборами значений всех переменных соответствующей программы. Так достигается соответствие между программным представлением поведения реактивной системы и графическим наглядным представлением состояний и переходов. Подобное расширение используется, например, в языке формального описания протоколов и сервисов [17].

Еще одним широко применяемым расширением классической модели конечного автомата являются диаграммы состояний (*Statecharts*), введенные Д. Харелом в [8].

Мы рассмотрим две возможности таких «карт состояний». Первая — это гиперсостояние, объединяющее несколько состояний, имеющих идентичную реакцию на одно и то же событие. Вторая возможность — использование «исторического псевдосостояния». Семантика такого состояния (оно обозначается H) состоит в том, что управление при возврате в гиперсостояние передается тому состоянию, в котором система находилась последний раз прежде, чем она покинула данное гиперсостояние. Переходы между состояниями в такой модели вызываются либо условиями (наступлением истинности предиката над внутренними переменными автомата, например, условие исчерпания буфера), либо событиями. Событиями в диаграммах состояний являются внешние события автомата. Обычно это прием управляющих или информационных сообщений из окружающей среды. Исчерпание тайм-аута также является возможным событием в этом расширении конечно-гого автомата.

На рис. 3.17 показано, как с помощью этих простых расширений удается наглядно представить сложное поведение. На рис. 3.17, а специфицируется прерывание системы в результате ошибки из любого состояния, принадлежащего гиперсостоянию Working, в состояние Crashed. После восстановления (*Recovery*) система продолжает работу из того состояния, в котором ее прервал сигнал *Crash*.

Протокол IEEE 802.12

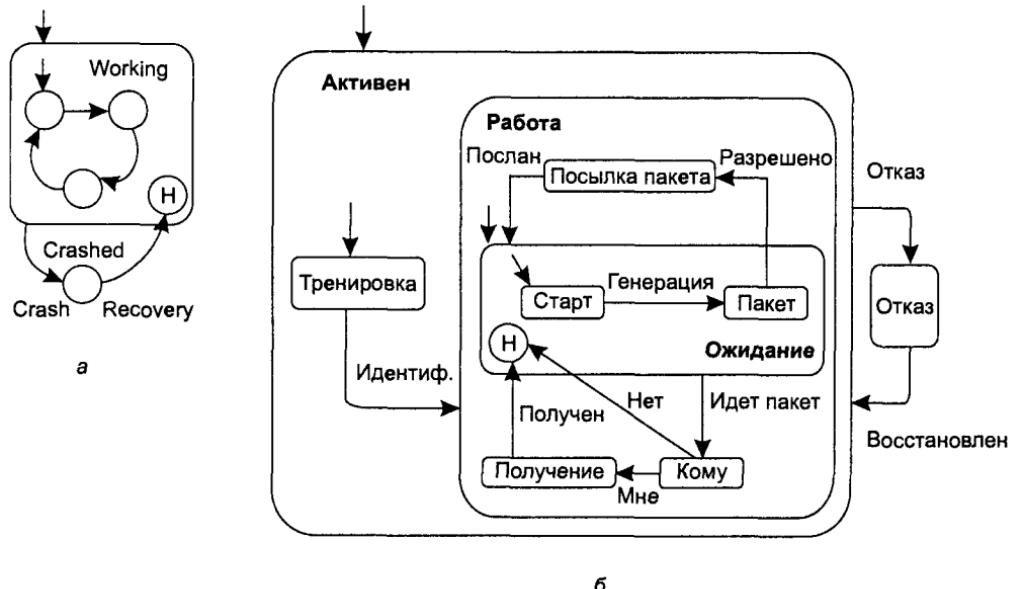


Рис. 3.17. Диаграммы состояний, представляющие прерывание при отказе (а) и спецификацию процесса доступа к среде протокола IEEE 802.12 (б)

Рисунок 3.17, б представляет спецификацию процесса доступа к среде протокола IEEE 802.12 для обмена сообщениями в высокоскоростной локальной сети. Процесс начинает свою работу в состоянии «Тренировка» гиперсостояния «Активен», в котором выполняются операции по идентификации данной рабочей станции, проверке канала и верхнего уровня. Если идентификация выполнилась без ошибок, процесс переходит в состояние «Старт», в котором ожидает очередной пакет, сгенерированный пользователем. Когда пакет для передачи сгенерирован, процесс в состоянии «Пакет» ждет разрешения послать его в канал, и когда разрешение получено, процесс переходит в состояние «Посылка пакета». По завершении пересылки процесс возвращается в состояние «Старт». В любом из двух состояний («Старт» и «Пакет») гиперсостояния «Ожидание» процесс может быть прерван приходом сигнала «Идет пакет» от верхнего уровня. Этот сигнал извещает процесс о том, что в сети начал передаваться пакет (верхний уровень еще не знает, кому), и всем станциям необходимо подготовиться к его приему. Как только получен заголовок пакета, верхний уровень определяет его адресата и информирует процесс о том, ему ли этот пакет. Если пакет предназначен данному процессу, то пакет направляется ему, если нет, то данному процессу посыпается сигнал отката. После приема пакета (или после сигнала, что пакет чужой) процесс возвращается в то состояние гиперсостояния «Ожидание», из которого он был прерван сигналом «Идет пакет». В любом из обсуждавшихся выше состояний процесс может быть прерван сигналом «Отказ», принятие которого заставляет процесс перейти в состояние «Отказ». После восстановления процесс входит в нормальную работу через состояние «Тренировка».

Подобная «квази-автоматная» форма описания динамики функционирования сложных систем в настоящее время широко распространена в системах поддержки проектирования программного обеспечения и именно она лежит в основе стандарта UML – универсального моделирующего языка.

Автомат, регулирующий пешеходный переход

Рассмотрим в качестве еще одного примера использования формализма *Statecharts* простейший автомат, регулирующий пешеходный переход по запросу пешеходов. Внешние события автомата – это события нажатия пешеходами кнопки-запроса на тротуаре и исчерпание тайм-аута. Автомат естественно строить как автомат Мура, в котором выход – регулирование светофора и разрешающий сигнал на переход – это потенциальные сигналы, являющиеся функциями состояния (рис. 3.18).

Выход автомата в каждом состоянии представляется парой: <*Светофор транспорта; светофор пешеходов*>. Например, в состоянии S1 управляющий автомат устанавливает <З; К>, то есть включенными зеленый свет транспорту и красный – пешеходам; в состоянии S6 установлен <Ж, К; К>, то есть желтый и красный свет транспорту (приготовиться) и красный – пешеходам. В начальном состоянии S0 разрешен проезд транспорту, а пешеходам движение запрещено. В состояниях S4, S5 при запрещающем сигнале транспорту зеленый сигнал пешеходам мигает каждые t0 секунд в течение t2 секунд. Запрос на переход принимается только в состо-

янии S_0 , в остальных состояниях он игнорируется. Задержки (тайм-ауты t_0-t_3) устанавливаются в момент перехода автомата в данное состояние, по исчерпании тайм-аута автомат переходит в следующее состояние. В гиперсостоянии Q , объединяющем пару состояний S_4 и S_5 , автомат находится ровно t_2 секунд: внутренние переходы не срывают тайм-аута. Именно для этого здесь удобно использовать гиперсостояние Q .

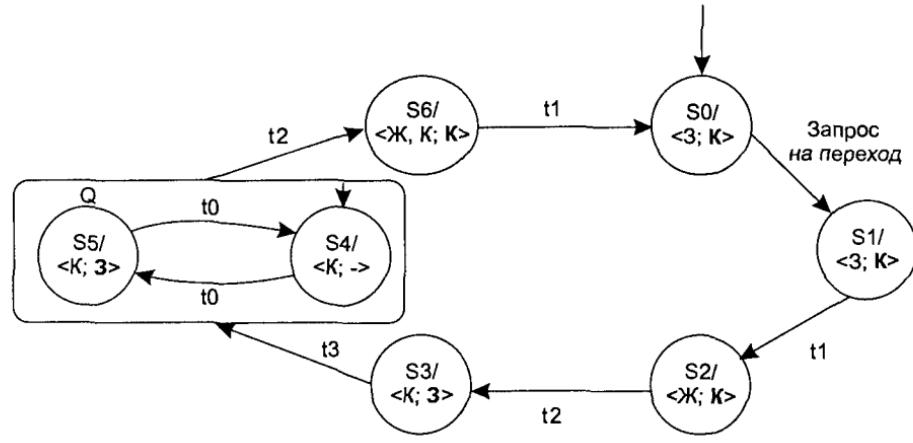


Рис. 3.18. Автомат, регулирующий пешеходный переход.

Графы переходов при спецификации и анализе параллельных программ

Состояние конечного автомата характеризуется тем, что оно однозначно определяет его будущее поведение, поскольку представляет собой класс его эквивалентных предысторий. Если строить подобный график переходов для программы обработки данных, то под состоянием здесь следует понимать вектор, включающий значения всех переменных программы и значение счетчика команд: именно этот вектор определяет будущее поведение программы. Если переменные программы принимают значения из бесконечных множеств, то и число состояний у такой модели бесконечное число. Однако модель с конечным числом состояний может быть часто использована для отражения структуры потока управления в программе. Так, пример предыдущего параграфа показывает, что хотя область значений данных в посылаемых и принимаемых данных протокола IEEE 802.12 бесконечна, эти данные не обрабатываются, и сложное поведение протокола не зависит от данных, а зависит лишь от внешних событий. Поэтому оно адекватно может быть специфицировано моделью с конечным числом состояний. Такое положение характерно именно для реактивных систем и, в частности, для параллельных процессов, в которых основную сложность представляют проблемы согласованного, коорди-

нированного поведения нескольких активностей, управляемых внешними событиями, а не проблемы преобразования данных. Для таких программ конечноавтоматная модель может быть адекватна и полезна. В настоящее время модель состояний и переходов интенсивно используется для представления поведения параллельных взаимодействующих процессов.

Параллельные процессы характерны тем, что они функционируют независимо, но при этом должны согласовывать свои действия в определенные моменты для достижения некоторой общей цели. Такие процессы называются кооперирующими, взаимодействующими.

Рассмотрим, например, проблему взаимного исключения — классическую проблему параллельного программирования. Эта проблема была впервые явно сформулирована Э. Дейкстрой в 1968 году [19]. Два параллельных процесса, P1 и P2, выполняют свои программы и время от времени должны использовать общий ресурс, захватывая его в исключительное использование (блоки памяти, принтер, структуру данных для изменения и т. д.). Если в то время, когда ресурсом пользуется один процесс, его попытается использовать другой, это может привести к ошибочной ситуации (например, porque печатного документа, porque общего файла и т. д.). Поэтому прежде, чем использовать ресурс, процесс должен как-то согласовать это с другим процессом — например, убедиться, что ресурс не занят. Мы рассмотрим решение этой задачи для вычислительных систем с разделяемой памятью, например, многозадачных систем на одном процессоре или многопроцессорных систем с общей памятью. В этих системах занятость ресурса процесс может отметить установкой определенного значения разделяемой специальной переменной (флага), которая может проверяться другим процессом. Одно из очевидных решений этой проблемы можно представить в следующем виде.

P1::	P2::
k0: noncritical1	m0: noncritical2
k1: x1 := 1	m1: x2 := 1
k2: await x2 = 0	m2: await x1 = 0
k3: critical1	m3: critical2
k4: x1 := 0	m4: x2 := 0
k5: goto k0	m5: goto m0

Процессы P1 и P2 находятся сначала в своей некритической секции (метки k0, m0), выполняя внутренние операции. Работа с разделяемым ресурсом происходит в критических секциях (метки k3, m3). Операторы с метками k1, k2 и k4, а также m1, m2 и m4 обеспечивают корректный доступ процессов в их критическую секцию. Для этого процесс Pi имеет свой флаг xi. Флаг сначала установлен в 0, это говорит другому процессу о том, что данный процесс не работает в своей критической секции. Прежде чем войти в критическую секцию, процесс устанавливает свой флаг в 1, предупреждая другой процесс о своем желании. Установив свой флаг, процесс проверяет, установлен ли чужой флаг. Если да, то это говорит о том, что

другой процесс работает в своей критической секции и данному процессу следует подождать. Дождавшись, когда другой процесс выйдет из критической секции, ожидающий процесс входит в свою критическую секцию и после работы с общим ресурсом сбрасывает свой флаг.

Эта простейшая задача синхронизации двух параллельных процессов будет решена корректно, если при выполнении процессами их локальных алгоритмов соблюсти глобальное свойство: процессы никогда не окажутся одновременно в своих критических секциях. Оказывается, что разработка параллельных процессов — непростая проблема, в частности приведенное выше решение некорректно. Для анализа этого решения воспользуемся автоматной моделью процесса (рис. 3.19).

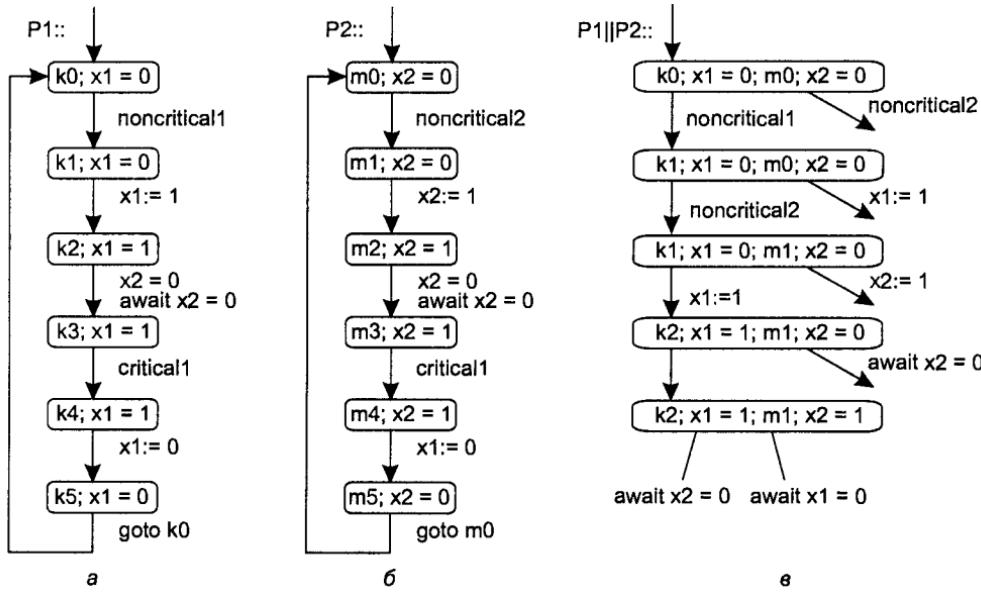


Рис. 3.19. Диаграммы состояний параллельно выполняемых процессов (а) и (б) и фрагмент их параллельной композиции (в)

Состояния процессов — это векторы значений своих переменных и указателя места в программе (счетчика команд). Последовательные состояния связаны переходами, помеченными выполняемыми операторами, которые меняют значения локальных переменных и, следовательно, состояние процессов. Некоторые операторы (например, «await $x2 = 0$ » в процессе P1) могут выполниться только при определенном условии, в данном случае — если $x2$ действительно равно 0. На рис. 3.19, в представлен фрагмент параллельной композиции этих процессов. Параллельная композиция — это процесс, имеющий в качестве состояний векторы значений переменных и указателей обоих процессов. Переходы между состояниями в модели отражают операции в процессах. Поскольку процессы параллельны и их операции выполняются независимо, то если хотя бы один процесс в данном глобальном состоянии мо-

жет выполнить действие, оно выполняется (и в модели присутствует соответствующий переход с изменением глобального состояния). Удобство представления параллельной композиции такой моделью состоит в том, что в ней явно отражаются операции согласования (синхронизации) компонентных процессов.

Фрагмент диаграммы состояний параллельной композиции $P1 \parallel P2$ на рис. 3.19, в демонстрирует некорректность приведенного выше решения проблемы взаимного исключения: он показывает возможность такого развития вычислений процессов, при котором оба они попадают в состояния, в которых будут бесконечно ждать друг друга. Такая ситуация называется блокировкой или дедлоком. Таким образом, с помощью модели систем переходов нам удалось *доказать* некорректность одного из предлагавшихся ранее решений проблемы взаимного исключения параллельных процессов.

Корректное решение этой проблемы было предложено Э. Дейкстрой [19] на основе понятия семафоров. Семафор — это специальный тип данных, принимающих значения из множества {0,1}, на котором определены две операции. Они традиционно обозначаются P и V . Если s — семафор, то $P(s)$ имеет результатом $s-1$, но выполняется только если $s > 0$. $V(s)$ имеет результатом $s+1$. При этом важно, что обе операции *атомны, неделимы* с точки зрения других процессов системы. Это означает, что любые процессы, желающие получить доступ к переменной типа семафор для ее проверки или обновления, получают этот доступ только последовательно друг за другом после полного завершения очередной операции над этой переменной. Решение Э. Дейкстры приведено ниже.

P1::	P2::
k0: noncritical1	m0: noncritical2
k1: P(s)	m1: P(s)
k2: critical1	m2: critical2
k3: V(s)	m3: V(s)
k4: goto k0	m4: goto m0

Это решение удивительно просто и полностью симметрично. Переменная s (типа *семафор*) является общей разделяемой переменной, она равна вначале 1. Для проверки корректности решения построим граф переходов параллельной композиции этих процессов (рис. 3.20).

Этот граф содержит 21 глобальное состояние; среди них нет состояний блокировки и некорректных состояний, содержащих как $k2$, так и $m2$. Таким образом, построив все возможные состояния системы параллельных процессов, мы *доказали*, что решение Э. Дейкстры проблемы взаимного исключения корректно: оно свободно от блокировок и не позволяет войти в критический интервал более чем одному процессу в каждый данный момент.

Отметим, однако, что проверка корректности параллельных процессов построением графа переходов системы процессов с последующим его анализом может быть

применена только для небольших систем, поскольку число глобальных состояний системы растет экспоненциально с ростом числа процессов. Например, граф глобальных состояний системы из N простейших процессов для решения задачи взаимного исключения содержит порядка 6^N состояний, и уже для 5–6 процессов такой подход не применим. Этот эффект называется «взрывом» числа состояний параллельной системы. Кроме того, как уже говорилось выше, невозможно построение адекватной конечноавтоматной модели для программ, содержащих переменные с бесконечной областью определения (конечно, если они реально влияют на исследуемые свойства).

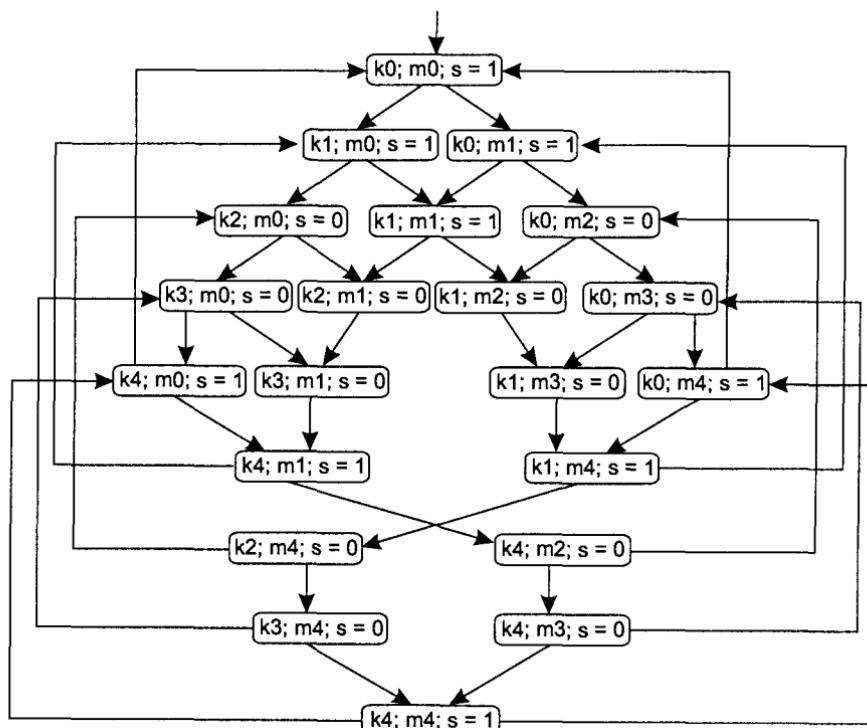


Рис. 3.20. Диаграмма состояний параллельной композиции процессов, решающих проблему взаимного исключения

Проблема умножения: алгоритм, который не может выполнить КА

Хотя конечный автомат как преобразователь информации является мощной моделью, существуют алгоритмические проблемы, которые не могут быть решены с помощью конечного автомата. Одной из таких проблем является проблема умножения двоичных чисел.

Теорема 3.4. Умножение двоичных чисел не может быть выполнено с помощью конечного автомата.

Доказательство теоремы проведем от противного. Предположим, что такой конечный автомат существует. Пусть он имеет n состояний. Подадим на вход автомата два одинаковых двоичных числа 2^{n+1} . Каждое из этих входных чисел записывается $n + 2$ двоичными разрядами. Автомат, выполняяший умножение, получает на вход $n + 1$ пар нулей, за которыми следует пара единиц — последовательное представление пары входных аргументов. Автомат, если он существует, должен выдать в качестве результата число 2^{2n+2} , то есть он должен выдать $2n + 2$ нуля, за которыми следует единица. Иными словами, после того как автомат получит $n + 2$ входных сигнала, он, перейдя в автономный режим, должен выдать еще n нулей, за которыми он должен выдать единицу (рис. 3.21). Однако очевидно, что никакой конечный автомат с n состояниями, работая в автономном режиме (под действием только входных сигналов синхронизации), не может выдать на выход n нулей, за которыми следует единица, поскольку максимальный цикл в автомате с n состояниями не превышает n .

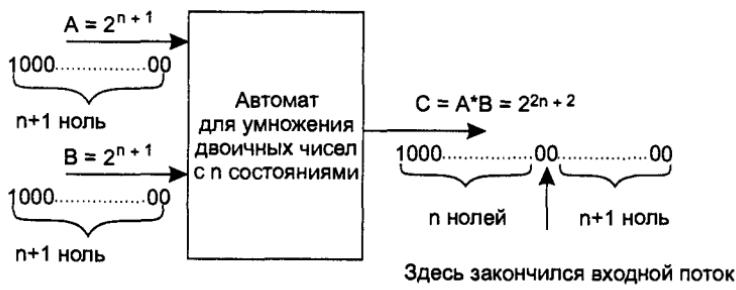


Рис. 3.21. Конечный автомат, выполняющий умножение двоичных чисел

Невозможность построения конечного автомата для решения этой проблемы является отражением того, что объем информации, которую должен был бы «помнить» такой автомат, неограниченно растет с ростом значений исходных чисел. В то же время для выполнения задачи сложения любых двух двоичных чисел автомату достаточно иметь только два состояния, одно из которых помнит наличие, а другое — отсутствие переноса в следующий разряд.

Алgebraическая структурная теория конечных автоматов

Проблема упрощения реализации логического блока конечного автомата связана с формальными свойствами разбиений множества состояний автомата, а эти свойства можно исследовать методами алгебры. Впервые связи между универсальной алгеброй и реализацией конечных автоматов были исследованы в монографии

Хартманиса и Стирнза «Алгебраическая структурная теория последовательных автоматов»¹. Изложенная там *алгебраическая структурная теория конечных автоматов* описывает возможность построения автоматов из отдельных модулей, что в итоге приводит к упрощению их реализации. Эта теория, как пишут авторы, «обладает абстрактной красотой и имеет удивительные приложения». В следующих разделах мы рассмотрим эту теорию.

Кодирование внутренних состояний конечного автомата

Кодирование *внутренних состояний* автомата никак не сказывается на логике его функционирования: при любом кодировании состояний автомат будет реализовывать одно и то же преобразование последовательностей входных сигналов в выходные. Кодирование состояний, однако, может влиять на надежность устройства, скорость его переключения, простоту реализации логического блока и т. д. До сих пор ведутся исследования проблемы оптимального кодирования состояний конечного автомата при различных критериях оптимальности, начатые еще в 60-х годах прошлого века. Например, в докладе [9] решается задача поиска такого кодирования состояний автомата, которое дает реализацию с минимальной рассеиваемой мощностью в элементах памяти. Для этого на определенном типе входных последовательностей учитывается частота переключения между парами всех состояний, и наиболее часто переключаемые пары состояний кодируются близкими по Хеммингу кодами. Это в среднем уменьшает число переключаемых триггеров при работе автомата.

Другим критерием оптимальности кодирования состояний автомата является простота реализации логического блока автомата. Рассмотрим автомат, в котором определена только функция переходов. Начальное состояние автомата не будем выделять, в рассматриваемом нами случае оно не имеет значения. Пусть автомат $A1 = \langle S, X, \delta \rangle$ имеет 8 состояний $S = \{0, \dots, 7\}$ и два входных сигнала, a и b . Функция переходов автомата δ представлена следующей таблицей.

Таблица 3.4. Таблица переходов $A1$

	a	b
0	3	4
1	4	7
2	3	0
3	2	6
4	1	5
5	1	4
6	2	3
7	4	3

¹ Hartmanis J., Stearns R. Algebraic Structure Theory of Sequential Machines // Prentice Hall Inc., N.Y., 1966.

Рассмотрим логические функции, реализующие функцию переходов автомата A1 при трех различных кодированиях внутренних состояний A1. Первый вариант кодирования использует двоичные коды номеров состояний. По кодированной таблице переходов с использованием карты Карно легко построить минимальные ДНФ каждой из двоичных функций, определяющих переключение двоичных разрядов (триггеров) блока памяти при кодировании входного сигнала $x = 0$ для a и $x = 1$ для b .

Кодирование состояний A1				Кодированная таблица переходов			Двоичные функции для трех разрядов
	q1	q2	q3		a	b	
0	0	0	0	000	011	100	$Q_1 = x \neg q_2 \vee x \neg q_1 q_3 \vee \neg q_1 \neg q_2 q_3 \vee \neg x q_1 q_2 q_3$
1	0	0	1	001	100	111	$Q_2 = \neg x \neg q_1 q_2 \vee \neg x \neg q_1 \neg q_3 \vee q_1 q_2 \neg q_3 \vee x q_2 q_3 \vee x \neg q_1 q_3$
2	0	1	0	010	011	000	$Q_3 = x \neg q_1 \neg q_3 \vee q_1 \neg q_2 \neg q_3 \vee \neg x q_1 \neg q_2 \vee \neg x \neg q_1 \neg q_2 q_3 \vee x q_1 q_2$
3	0	1	1	011	010	110	
4	1	0	0	100	001	101	
5	1	0	1	101	001	100	
6	1	1	0	110	010	011	
7	1	1	1	111	100	011	

При этом варианте кодирования каждая функция Q1, Q2 и Q3, определяющая двоичные разряды кода следующего состояния, зависит от всех трех значений q1, q2 и q3 кода текущего состояния автомата. Рассмотрим другой вариант кодирования внутренних состояний автомата A1.

Кодирование состояний A1				Кодированная таблица переходов			Двоичные функции для трех разрядов
	q1	q2	q3		a	b	
0	0	0	0	000	010	011	$Q_1 = \neg x q_2 \vee \neg q_1 q_2$
1	1	0	0	001	011	010	$Q_2 = x q_2 \vee \neg x \neg q_2 \vee x \neg q_1$
2	1	0	1	010	100	111	$Q_3 = \neg x q_1 \neg q_3 \vee \neg x q_2 \neg q_3 \vee x \neg q_1 \neg q_3 \vee \neg x \neg q_1 \neg q_2 q_3$
3	0	1	0	011	101	110	
4	0	1	1	100	010	000	
5	1	1	1	101	011	001	
6	1	1	0	110	101	011	
7	0	0	1	111	100	010	

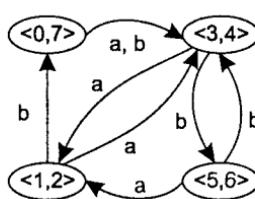
Функции Q1 и Q2 имеют при таком кодировании значительно более простой вид. Принципиальное отличие данного варианта от предыдущего состоит в том, что функции Q1 и Q2 совершенно не зависят от q3. Поэтому автомат A1 при таком

кодировании можно реализовать так. Построим схему **B**, на каждом шаге своего функционирования определяющую первые два разряда кода следующего состояния **A1** (Q_1 и Q_2) по этим же разрядам кода текущего состояния **A1** (q_1 и q_2) и входному сигналу x . Автомат **A1** можно построить как композицию этой схемы **B** с другой схемой **C**, которая по разрядам q_1 и q_2 , полученным от схемы **B**, и разряду q_3 текущего состояния автомата **A1** определяет разряд Q_3 его следующего состояния. Фактически схема **B** представляет собой автомат. Такая реализация называется последовательной декомпозицией конечного автомата **A1**. Очевидно, что подобное кодирование состояний автомата обладает очень удобными свойствами. Можно ли его найти без перебора множества всех возможных кодирований, которых для автомата **A1** более 40 000?

Рассмотрим этот вариант кодирования с другой точки зрения. Каждый разряд двоичного кодирования множества состояний S и любая группа разрядов определяют разбиение этого множества на классы эквивалентности. В один класс попадут те элементы из S , которые имеют одно и то же значение выбранных разрядов. При выбранном нами кодировании разряды q_1 и q_2 одинаковы у состояний 0 и 7 (кодируются 00), 1 и 2 (кодируются 10) и т. д. Это соответствует разбиению множества состояний автомата **A1** на классы эквивалентности, причем эквивалентными здесь естественно считать те состояния, при кодировании которых пары первых двоичных разрядов одинаковы. Это дает разбиение $\pi_1 = \{<0,7>; <1,2>; <3,4>; <5,6>\}$. Поскольку Q_1 и Q_2 – это двоичные разряды следующего состояния автомата, а q_1 и q_2 – те же самые разряды предыдущего состояния, то можно считать, что при таком разбиении в конечном автомате **A1** выделен упомянутый выше «подавтомат» **B**, состояниями которого полагаются блоки разбиения π_1 . Его таблица и граф переходов определяются так.

Таблица 3.5. Таблица переходов **B**:

	δ_a	δ_b
$<0,7>$	$<3,4>$	$<3,4>$
$<1,2>$	$<3,4>$	$<0,7>$
$<3,4>$	$<1,2>$	$<5,6>$
$<5,6>$	$<1,2>$	$<3,4>$



Итак, упрощение реализации автомата **A1** определено связано с тем, что разбиение π_1 на множестве состояний **A1** согласовано с его функцией переходов, а именно, по номеру блока разбиения π_1 , в котором находится текущее состояние автомата **A1**, и входному сигналу однозначно определяется номер блока этого разбиения, в котором будет находиться следующее состояние **A1**. При поиске наиболее простой реализации автомата перебор кодирований его состояний можно заменить задачей поиска таких разбиений его множества состояний, которые обладают свойством «согласованности» с функцией переходов автомата («замкнутостью» относительно этой функции).

Рассмотрим третий вариант кодирования состояний автомата **A1**. Он отличается от предыдущего только значениями кода в третьем разряде:

Кодирование состояний A1			Кодированная таблица переходов			Двоичные функции для трех разрядов	
	q1	q2	q3	a	b		
0	0	0	0	000	011	010	$Q1 = \neg xq2 \vee \neg q1q2$
1	1	0	1	001	010	011	$Q2 = xq2 \vee \neg x \neg q2 \vee x \neg q1$
2	1	0	0	010	101	110	$Q3 = \neg x \neg q3 \vee xq3$
3	0	1	1	011	100	111	
4	0	1	0	100	011	000	
5	1	1	0	101	010	001	
6	1	1	1	110	101	010	
7	0	0	1	111	100	011	

При этом кодировании функции переходов приобретают совсем простой вид. В частности, функция $Q3$ зависит только от $q3$. Это дает возможность независимой реализации групп функций $\{Q1, Q2\}$ и $\{Q3\}$, что приводит к реализации автомата **A1** как параллельной композиции двух независимо функционирующих автоматов.

Новый вариант кодирования состояний **A1**, который определяет такую простую его реализацию, отличается от предыдущего варианта только значениями разряда $q3$. Соответствующее разбиение множества состояний **A1** — это $\pi_2 = \{<0,2,4,5>; <1,3,6,7>\}$. Разбиение π_2 обладает тем же свойством, что и разбиение π_1 , оно тоже согласовано с функцией переходов автомата **A1**.

Таким образом, проблема упрощения реализации логического блока конечного автомата связана с формальными свойствами разбиений множества состояний автомата.

Разбиения и частично упорядоченные множества

Несмотря на то что понятие разбиений уже использовалось ранее, мы введем его здесь формально вместе со сведениями о свойствах разбиений.

Определение 3.8. Разбиением множества M называется множество непересекающихся непустых подмножеств, объединение которых совпадает с M .

Разбиение конечного множества задают перечислением его элементов (блоков). Например, $\pi_1 = \{<0,7>; <1,2>; <3,4>; <5,6>\}$ – это разбиение множества из восьми элементов $\{0, 1, \dots, 7\}$ на четыре блока. Другие возможные разбиения этого же множества: $\pi_2 = \{<0,2,4,5>; <1,3,6,7>\}$, $\pi_3 = \{<0>; <1>; <2>; <3,6>; <4>; <5>; <7>\}$, $\pi_4 = \{<0,7>; <1,2>; <3,4,5,6>\}$.

Множество всех разбиений множества M будем обозначать $\Pi(M)$. Разбиения множества M тесно связаны с отношениями эквивалентности на M . Справедлива следующая теорема.

Теорема 3.5. Всякое отношение эквивалентности на множестве M определяет разбиение M (на классы эквивалентности) и обратно, каждое разбиение M определяет отношение эквивалентности на M .

Пусть ρ – некоторое отношение эквивалентности на M . Подмножество элементов множества M , эквивалентных $a \in M$, называется классом эквивалентности для a : $[a]_\rho = \{x | x \in M \text{ & } x \rho a\}$. Легко видеть, что классы эквивалентности – это блоки разбиения, соответствующего данной эквивалентности. Блок разбиения π , содержащий элемент $a \in M$, обозначается $[a]_\pi$ – точно так же, как и класс соответствующей эквивалентности для a .

На множестве $\Pi(M)$ можно ввести операции сложения и умножения. Результат умножения разбиений определяется очень просто: это разбиение, блоками которого являются пересечения блоков исходных разбиений. Например, $\pi_2 * \pi_4 = \{<0>; <1>; <2>; <3,6>; <4>; <5>; <7>\}$. Формально $[a]_{\pi_2 * \pi_4} = [a]_{\pi_2} \cap [a]_{\pi_4}$. Знак операции умножения разбиений часто опускается. Сумма двух разбиений определяется как разбиение, блоками которого являются объединения пересекающихся блоков исходных разбиений. Например, $\pi_2 + \pi_4 = \{<0,1,2,3,4,5,6,7>\}$. Определим этот алгоритм формально.

Пусть для любого $a \in M$, $[a]^! = [a]_\pi \cup [a]_\xi$.

Для $i > 1$ пусть $[a]^{i+1} = [a]^i \cup \{B | B – \text{блок } \pi \text{ или } \rho, \text{ пересекающийся с } [a]^i\}$.

Тогда $[a]_{\pi+\xi} = [a]^i$, для такого i , что $[a]^i = [a]^{i+1}$.

Этот алгоритм соответствует операции транзитивного замыкания объединения двух отношений эквивалентности. Операции сложения и умножения разбиений обладают, кроме прочих, свойствами идемпотентности (поглощения), коммутативности и ассоциативности:

$$\pi^* \pi = \pi; \pi + \pi = \pi;$$

$$\pi^* \xi = \xi \pi; \pi + \xi = \xi + \pi;$$

$$(\pi^* \xi)^* \tau = \pi^* (\xi^* \tau); (\pi + \xi) + \tau = \pi + (\xi + \tau).$$

Среди всех элементов $\Pi(M)$ существует два особых разбиения – одно, состоящее из единственного блока, включающего все элементы из M , и другое, каждый блок

которого состоит из одного-единственного элемента. Первое называют наибольшим, а второе – наименьшими разбиениями, обозначая их **I** и **0** соответственно. Для множества из 8 элементов эти разбиения имеют вид: $I = \{<0,1,2,3,4,5,6,7>\}$ и $0 = \{<0>; <1>; <2>; <3>; <4>; <5>; <6>; <7>\}$. Очевидно, что для любого разбиения π справедливы соотношения: $I + \pi = I$, $I^* \pi = \pi$, $0 + \pi = \pi$, $0^* \pi = 0$. Разбиение **I** играет роль единицы, а **0** – роль нуля в алгебре разбиений.

Разбиения из $\Pi(M)$ образуют частичный порядок. Два разбиения π и ξ находятся в отношении « \leq », если блоки разбиения π являются подблоками разбиения ξ . Например, $\pi_1 = \{<0,7>; <1,2>; <3,4>; <5,6>\}$ находится в этом отношении с $\pi_4 = \{<0,7>; <1,2>; <3,4,5,6>\}$, и оба они находятся в этом отношении с **I**. Вообще разбиение **I** является наибольшим, а разбиение **0** – наименьшим элементами этого частичного порядка. Диаграмма Хассе перечисленных выше разбиений множества из восьми элементов приведена на рис. 3.22.

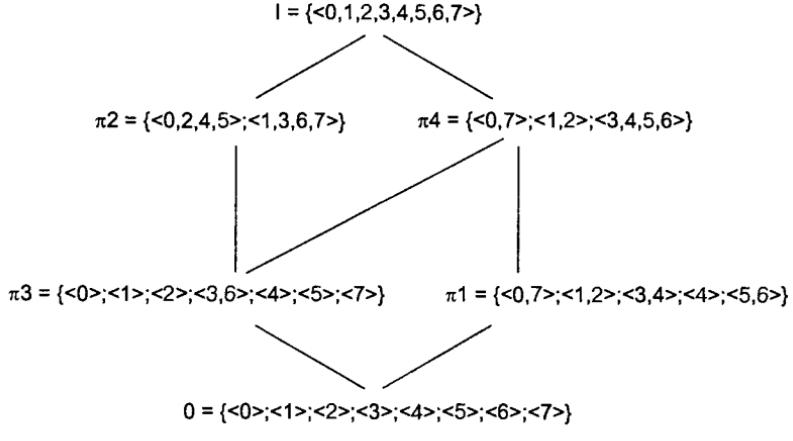


Рис. 3.22. Диаграмма Хассе множества разбиений $\{0, \pi_1, \pi_2, \pi_3, \pi_4, I\}$

Пусть W – частично упорядоченное множество и $U \subseteq W$. Множество U имеет в W точную нижнюю грань w , когда $(\forall u \in U)w \leq u$, причем $(\forall w' \in W)(\forall u \in U)w' \leq u \Rightarrow w' \leq w$. Множество U имеет в W точную верхнюю грань тогда и только тогда, когда существует элемент $w \in W$, такой что $u \leq w$ для любого элемента u из U , причем $(\forall w' \in W)(\forall u \in U)u \leq w' \Rightarrow w \leq w'$. Частично упорядоченное множество W называется решеткой (lattice), если для любой пары элементов из W в множестве W существует как точная нижняя, так и точная верхняя грань этой пары. Справедлива следующая теорема.

Теорема 3.6. *Множество всех возможных разбиений произвольного множества является решеткой.*

Действительно, легко показать, что для любой пары разбиений $\{\pi, \xi\}$ их точная нижняя грань – это их произведение $\pi^* \xi$, а точная верхняя грань – это их сумма

$\pi + \xi$. На множестве всех возможных разбиений множества могут существовать подмножества, также являющиеся решетками. Например, решетку составляет пара $\{\mathbf{0}, \mathbf{I}\}$ разбиений. Множество рассмотренных выше разбиений $\{\mathbf{0}, \pi_1, \pi_2, \pi_3, \pi_4, \mathbf{I}\}$ множества $\{0, \dots, 7\}$ также является решеткой — здесь для каждой пары разбиений существует как нижняя, так и верхняя грани. Рисунок 3.22 представляет диаграмму Хассе этой решетки.

Универсальные алгебры и конгруэнции

Понятие алгебры было введено в главе 2, когда рассматривались свойства булевых алгебр. Как оказывается, конечный автомат тоже можно рассматривать как алгебру, поскольку алгебра — это просто множество с несколькими определенными на нем операциями. Введем формально понятие универсальной алгебры.

Определение 3.9. *Множество M вместе с набором операций $\Sigma = \{\varphi_1, \dots, \varphi_s\}$, $\varphi_i: M^{n_i} \rightarrow M$, где n_i — арность операции φ_i , называется универсальной алгеброй.*

Множество M называется носителем алгебры, Σ называется сигнатурой, а вектор арностей операций $\langle n_1, \dots, n_s \rangle$ называется типом алгебры. Записывается алгебра так: $\langle M; \varphi_1, \dots, \varphi_s \rangle$ или $\langle M; \Sigma \rangle$. Примеры алгебр:

- $\langle \mathbb{R}; +, * \rangle$ — алгебра действительных чисел, ее тип $\langle 2, 2 \rangle$;
- $\langle 2^M; \cup, \cap, \sim \rangle$ — алгебра подмножеств над множеством M , ее тип $\langle 2, 2, 1 \rangle$;
- $\langle \mathbb{N}; + \rangle$ — алгебра целых положительных чисел с одной операцией сложения, ее тип $\langle 2 \rangle$;
- $\langle \Pi(M); +, * \rangle$ — алгебра разбиений множества M , ее тип $\langle 2, 2 \rangle$.

Применение результатов теории универсальных алгебр к теории автоматов возможно потому, что конечный автомат специального вида, у которого определена только функция переходов, тоже можно рассматривать как универсальную алгебру. Действительно, носителем такой алгебры можно считать множество состояний автомата, а сигнатурой — множество функций переходов, по одной для каждого входного сигнала автомата. Например, автомат с множеством состояний S , множеством входных сигналов X и функцией переходов $\delta: S \times X \rightarrow S$ представляется универсальной алгеброй $\langle S; \{\delta_x | x \in X\} \rangle$, где каждая функция δ_x — унарная операция $\delta_x: S \rightarrow S$ для соответствующего x . Конечный автомат $A1 = \langle S, X = \{a, b\}, \delta \rangle$ можно представить универсальной алгеброй $\langle S; \delta_a, \delta_b \rangle$ с носителем S и двумя унарными функциями переходов δ_a и δ_b . Ее тип $\langle 1, 1 \rangle$.

Определение 3.10. *Разбиение π на носителе M алгебры $\langle M; \varphi_1, \dots, \varphi_s \rangle$ такое, что $(\forall i \in 1 \dots s)(\forall a_j, b_j \in M) [a_j]_\pi = [b_j]_\pi \Rightarrow [\varphi_i(a_1, \dots, a_r)]_\pi = [\varphi_i(b_1, \dots, b_r)]_\pi$.*
называется конгруэнцией.

Итак, конгруэнция алгебры — это разбиение, согласованное с операциями алгебры именно в том смысле, который мы рассматривали при кодировании состояний

конечного автомата. А именно, если соответствующие аргументы операции алгебры брать из одного и того же блока конгруэнции, то результат операции всегда будет попадать в один и тот же блок этой конгруэнции.

Например, пусть $\square_N = \langle N; + \rangle$ — алгебра целых положительных чисел с одной операцией сложения. Разбиение π множества целых чисел N на блоки $\pi = \{B_0, B_1, B_2\}$ такие, что в i -й блок попадают все числа, равные i по модулю 3, является конгруэнцией. Действительно, алгебра \square_N имеет только одну бинарную операцию сложения. Для любых $a_1, a_2, b_1, b_2 \in N$, если a_1 находится в одном и том же блоке разбиения π , что и b_1 (то есть они равны по модулю 3), а a_2 находится в одном и том же блоке π , что и b_2 (они также равны по модулю 3), то результаты двух операций $a_1 + a_2$ и $b_1 + b_2$ находятся в одном и том же блоке π (то есть результаты сложений этих чисел также равны по модулю 3). Например, если a_1 и b_1 взять из блока B_1 , а a_2 и b_2 — из блока B_2 , то результаты будут в блоке B_0 в обоих случаях.

Вследствие свойства согласованности конгруэнций относительно операций алгебры, по любой конгруэнции π алгебры можно определить алгебру с той же сигнатурой, но заданную на блоках π . Такая новая алгебра называется фактор-алгеброй исходной алгебры по конгруэнции π . Например, для алгебры \square_N по конгруэнции π можно определить фактор-алгебру $\square_{N/\pi} = \langle \{B_0, B_1, B_2\}; \oplus \rangle$, где операция сложения \oplus определена на блоках B_i очевидным образом: $(\forall a, b \in N) [a]_\pi \oplus [b]_\pi = [a+b]_\pi$. Любая алгебра всегда имеет две тривиальные конгруэнции — нулевое разбиение $\mathbf{0}$ и единичное разбиение \mathbf{I} носителя алгебры. Фактор-алгебра $\square/\mathbf{0}$ изоморфна исходной алгебре \square , фактор-алгебра \square/\mathbf{I} определяет наиболее абстрактное представление алгебры — простейшую алгебру той же сигнатуры с одноэлементным носителем. Любая нетривиальная конгруэнция π алгебры \square , если она существует, определяет фактор-алгебру \square/π , имеющую менее детальную структуру, чем исходная алгебра.

Рассмотрим, как понятия конгруэнции и фактор-алгебры можно использовать для конечных автоматов. Разбиение $\pi_1 = \{<0,7>; <1,2>; <3,4>; <5,6>\}$ множества состояний конечного автомата A_1 , очевидно, является конгруэнцией соответствующей этому автомату алгебры $\langle S; \delta_a, \delta_b \rangle$. Действительно, легко видеть, что функции переходов автомата A_1 «согласованы» с разбиением π_1 . Например, $\delta_a(1) = 4, \delta_a(2) = 3$, то есть можно считать, что δ_a отображает блок $<1,2>$ в блок $<3,4>$, или, формально, $[\delta_a(1)]_{\pi_1} = [\delta_a(2)]_{\pi_1} = <3,4>$. Это же верно и для δ_b : $[\delta_b(1)]_{\pi_1} = [\delta_b(2)]_{\pi_1} = <0,7>$. Непосредственной проверкой можно убедиться, что для π_1 выполняется соотношение:

$$(\forall x \in X)(\forall s, r \in S) [s]_{\pi_1} = [r]_{\pi_1} \Rightarrow [\delta_x(s)]_{\pi_1} = [\delta_x(r)]_{\pi_1}.$$

Последняя формула является определением конгруэнции конечного автомата. А именно,

Определение 3.11. Разбиение π на множестве состояний конечного автомата A является конгруэнцией, если под воздействием любого входного сигнала автомат из двух любых состояний, принадлежащих одному и тому же блоку разбиения π , пере-

ходит в состояния, также принадлежащие одному и тому же (произвольному) блоку этого разбиения.

Очевидно, что разбиение **0** множества состояний на одноэлементные блоки всегда является конгруэнцией, так же как и разбиение **I**, которое содержит один-единственный блок, включающий все состояния автомата. Кроме этих тривиальных конгруэнций автомат может иметь и другие, нетривиальные конгруэнции. Например, для автомата **A1** все разбиения рис. 3.22 являются конгруэнциями.

Конгруэнция π конечного автомата $A = (S, X, s_0, \delta)$ определяет фактор-автомат $A/\pi = (\pi, X, [s_0]_\pi, \delta_\pi)$, состояниями которого являются блоки разбиения π , а функция переходов δ_π для каждого входного сигнала x из X и каждого блока разбиения π индуцирована конгруэнцией π :

$$(\forall x \in X)(\forall s \in S) \delta_\pi([s]_\pi, x) = [\delta(s, x)]_\pi.$$

Для автомата **A1** фактор-автоматы $A1/\pi_1$ по конгруэнциям

$$\pi_1 = \{<0,7>; <1,2>; <3,4,5,6>\} \text{ и } \pi_2 = \{<0,2,4,5>; <1,3,6,7>\},$$

а также по максимальной конгруэнции **I** задаются следующими таблицами и графиками переходов.

Таблица 3.6. Таблица переходов $A1/\pi_1$:

	δ_a	δ_b
$<0,7>$	$<3,4,5,6>$	$<3,4,5,6>$
$<1,2>$	$<3,4,5,6>$	$<0,7>$
$<3,4>$	$<1,2>$	$<3,4,5,6>$

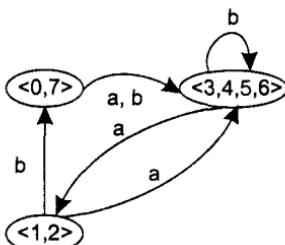


Таблица 3.7. Таблица переходов $A1/\pi_2$:

	δ_a	δ_b
$<0,5,6,7>$	$<1,2,3,4>$	$<1,2,3,4>$
$<1,2,3,4>$	$<1,2,3,4>$	$<0,5,6,7>$

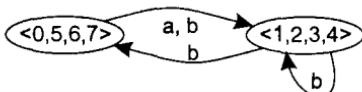
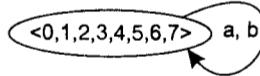


Таблица 3.8. Таблица переходов A1/I::

	δ_a	δ_b
$\langle 0,1,2,3,4,5,6,7 \rangle$	$\langle 0,1,2,3,4,5,6,7 \rangle$	$\langle 0,1,2,3,4,5,6,7 \rangle$



Таким образом, любой автомат А всегда имеет две тривиальных конгруэнции — нулевое разбиение **0** и единичное разбиение **I** множества состояний. Фактор-автомат $A/0$ изоморфен исходному автомата А, фактор-автомат A/I определяет наиболее абстрактное представление автомата — простейший автомат с одним состоянием, у которого каждый входной сигнал оставляет автомат в этом же состоянии. Если автомат А имеет нетривиальную конгруэнцию π , то она определяет фактор-автомат A/π , имеющий менее детальную структуру, чем исходный автомат А.

Основным вопросом структурной теории конечных автоматов является вопрос нахождения и использования нетривиальных конгруэнций и соответствующих фактор-автоматов.

Последовательная декомпозиция конечных автоматов

Пусть конечный автомат А имеет нетривиальную конгруэнцию π . Очевидно, что фактор-автомат A/π представляет поведение исходного автомата А с меньшей детальностью. Если говорить точнее, фактор-автомат A/π задает поведение А с точностью до блоков разбиения π .

Поставим вопрос: чем фактор-автомат A/π может помочь при реализации автомата А? Ведь фактор-автомат предоставляет только часть информации о поведении исходного автомата. Очевидно, что если мы реализуем фактор-автомат A/π , то будем знать номер блока разбиения π , которому принадлежит текущее состояние автомата. Выбрав любое разбиение ρ , ортогональное конгруэнции (то есть такое, что $\pi^*\rho = 0$), и проведя двоичное кодирование блоков разбиения ρ , мы можем построить последовательную декомпозицию исходного автомата в точности так, как это было сделано для автомата **A1** по конгруэнции $\pi_1 = \{<0,7>;<1,2>;<3,4>;<5,6>\}$ и ортогональному ему разбиению $\mu = \{<0,1,3,6>;<2,4,5,7>\}$ при втором варианте кодирования состояний **A1**. В общем случае справедлива следующая теорема.

Теорема 3.7. *Если конечный автомат имеет нетривиальную конгруэнцию, то для него существует последовательная декомпозиция.*

Доказательство. Пусть $\pi = \{B_1, B_2, \dots\}$ — конгруэнция конечного автомата $A = \{S, X, s_0, \delta\}$ и $\rho = \{T_1, T_2, \dots\}$ — произвольное разбиение S такое, что $\pi^*\rho = 0$. Пусть $K(B_i)$ — двоичный код блока B_i разбиения π , а $K(T_j)$ — двоичный код блока

Тj разбиения ρ . Каждое состояние $s \in S$ закодируем кодом $K([s]_\pi)K([s]_\rho)$. Очевидно, что поскольку π — конгруэнция, то соответствующие разряды двоичного кода следующего состояния конечного автомата A могут быть определены как функция только этих же разрядов текущего состояния, что и определяет последовательную декомпозицию автомата.

Параллельная декомпозиция конечных автоматов

Параллельная декомпозиция конечных автоматов также строится просто: если на множестве состояний автомата A существует две ортогональные конгруэнции π и ρ , то пара независимо работающих фактор-автоматов A/π и A/ρ реализуют исходный автомат A. Доказательство этого положения очевидно. В приведенном выше примере реализации автомата A1 при третьем варианте кодирования состояний дополнительно к конгруэнции $\pi_1 = \{<0,7>; <1,2>; <3,4>; <5,6>\}$ была выбрана ортогональная ей конгруэнция $\pi_2 = \{<0,2,4,5>; <1,3,6,7>\}$.

Рисунок 3.23 представляет структуру последовательной и параллельной декомпозиции конечного автомата A.

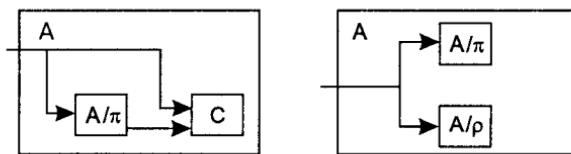


Рис. 3.23. Последовательная и параллельная декомпозиция конечного автомата

Алгоритм поиска конгруэнций конечного автомата

Этот алгоритм основывается на следующей теореме, которую мы приводим без доказательства.

Теорема 3.8. *Множество конгруэнций универсальной алгебры является решеткой.*

Определим замыкание $[[\pi]]_A$ разбиения π на множестве состояний автомата $A = (S, X, \delta)$ как такую конгруэнцию, что $\pi \leq [[\pi]]_A$, причем $[[\pi]]_A$ — это минимальная конгруэнция A, обладающая этим свойством. Замыкание $[[\pi]]_A$ можно построить, используя следующий алгоритм.

- Положим $\pi^0 = \pi$. Это начальное разбиение для последующей итерации.
- Для $i \geq 0$ определим $\pi^{i+1} = \pi^i + \sum_{[r]\pi_i=[s]\pi_i} \sum_{x \in X} \rho(\delta(r, x), \delta(s, x))$.

□ Иными словами, следующее разбиение в каждой итерации строится как сумма предыдущего разбиения и разбиений $\rho(\delta(g, x), \delta(s, x))$ для каждой пары тех состояний, в которые А переходит из каждой пары состояний g и s , принадлежащих одному блоку разбиения предыдущей итерации для каждого входного сигнала.

□ Замыкание $[[\pi]]_A$ определяется как такое разбиение π^i , что $\pi^{i+1} = \pi^i$.

Здесь $\rho(g, s)$ обозначает элементарное разбиение, которое объединяет в один блок только g и s , а все остальные его блоки — одноэлементные. Если $g = s$, то $\rho(g, s) = 0$. Все множество конгруэнций автомата А может быть построено в два этапа. На первом этапе для каждой пары (g, s) различных состояний А строится замыкание $[[\rho(g, s)]]_A$. Все построенные таким образом конгруэнции являются образующими решетки конгруэнций автомата А. На втором этапе для построения полной решетки конгруэнций автомата А следует перебрать все возможные суммы конгруэнций, построенных на первом этапе.

Для автомата **A1**, содержащего 8 состояний, на первом этапе строится

$$7+6+5+\dots+1=28$$

таких замыканий разбиений $\rho(0,1), \rho(0,2), \dots, \rho(6,7)$.

Рассмотрим, например, вычисление $[[\rho(0,1)]]_{A1}$. В соответствии с алгоритмом, $\pi^0 = \rho(0, 1) = \{<0,1>; <2>; <3>; <4>; <5>; <6>; <7>\}$. Для получения π^1 это разбиение надо объединить с разбиениями $\rho(\delta(0, a), \delta(1, a)) = \rho(3, 4)$ и $\rho(\delta(0, b), \delta(1, b)) = \rho(4, 7)$. Поэтому $\pi^1 = \pi^0 + \rho(3, 4) + \rho(4, 7) = \{<0,1>; <2>; <3,4,7>; <5>; <6>\}$. Разбиение π^2 строится как объединение разбиения π^1 с разбиениями:

$$\rho(\delta(3, a), \delta(4, a)) = \rho(1, 2),$$

$$\rho(\delta(3, b), \delta(4, b)) = \rho(5, 6),$$

$$\rho(\delta(3, a), \delta(7, a)) = \rho(2, 4),$$

$$\rho(\delta(3, b), \delta(7, b)) = \rho(3, 6),$$

$$\rho(\delta(4, a), \delta(7, a)) = \rho(1, 4),$$

$$\rho(\delta(4, b), \delta(7, b)) = \rho(3, 5).$$

Таким образом, $\pi^2 = \{<0,1,2,3,4,5,6,7>\} = I$. На этом вычисление $[[\rho(0,1)]]_{A1}$ заканчивается, поскольку получено максимальное разбиение.

Построим теперь замыкание разбиения $[[\rho(0,2)]]_{A1}$. Вначале $\pi^0 = \rho(0, 2)$. Разбиение $\pi^1 = \pi^0 + \rho(\delta(0, a), \delta(2, a)) + \rho(\delta(0, b), \delta(2, b)) = \pi^0 + \rho(3, 3) + \rho(0, 4) = \{<0,2,4>; <1>; <3>; <5>; <6>; <7>\}$.

Далее

$$\begin{aligned} \pi^2 &= \pi^1 + \rho(\delta(0, a), \delta(4, a)) + \rho(\delta(0, b), \delta(4, b)) + \rho(\delta(2, a), \delta(4, a)) + \\ &+ \rho(\delta(2, b), \delta(4, b)) = \pi^1 + \rho(1, 3) + \rho(4, 5) + \rho(0, 5) + \rho(4, 5) = \\ &= \{<0,2,4,5>; <1,3>; <6>; <7>\}. \end{aligned}$$

Окончательно $[[\rho(0, 2)]]_{A1} = \{<0, 2, 4, 5>; <1, 3>; <6, 7>\}$.

Для автомата A1 эта процедура дает 9 нетривиальных конгруэнций $\pi_1 - \pi_9$.

$$\pi_1 = \{<0, 7>; <1, 2>; <3, 4>; <5, 6>\}$$

$$\pi_2 = \{<0, 2, 4, 5>; <1, 3, 6, 7>\}$$

$$\pi_3 = \{<0>; <1>; <2>; <3, 6>; <4>; <5>; <7>\}$$

$$\pi_4 = \{<0, 7>; <1, 2>; <3, 4, 5, 6>\}$$

$$\pi_5 = \{<0, 5>; <1, 3>; <2, 4>; <6, 7>\}$$

$$\pi_6 = \{<0>; <1>; <2>; <3>; <4, 5>; <6>; <7>\}$$

$$\pi_7 = \{<0, 5>; <1, 3, 6, 7>; <2, 4>\}$$

$$\pi_8 = \{<0, 2, 4, 5>; <1, 3>; <6, 7>\}$$

$$\pi_9 = \{<0, 5, 6, 7>; <1, 2, 3, 4>\}$$

На втором этапе к этому множеству должна быть добавлена конгруэнция

$$\pi_{10} = \{<0>; <1>; <2>; <3, 6>; <4, 5>; <7>\},$$

полученная сложением конгруэнций π_3 и π_6 . Суммы пар всех остальных конгруэнций, входящих в множество $\{0, \pi_1, \dots, \pi_{10}, I\}$, принадлежат этому же множеству. Следовательно, построение множества конгруэнций автомата A1 закончено.

Диаграмма Хассе решетки конгруэнций автомата A1 представлена на рис. 3.24.

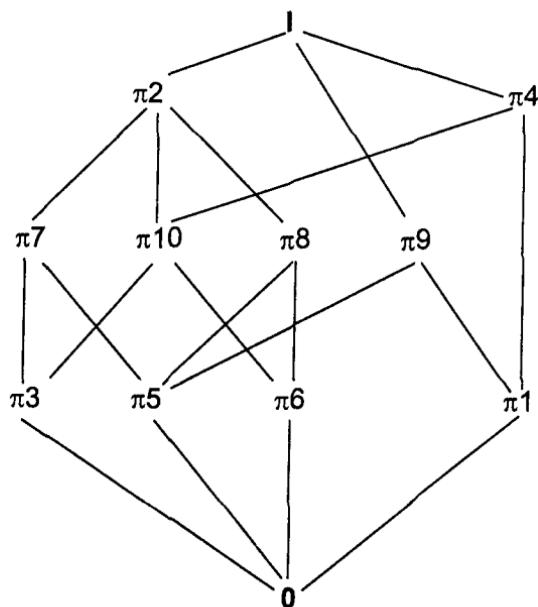


Рис. 3.24. Решётка конгруэнций автомата A1

Контрольные задания

1. (*Задача о синхронизации цепи стрелков* [27].) При попытках описать автоматными моделями поведение биологических объектов (в частности, синхронизации моментов деления клеток) встал вопрос о возможных механизмах, обеспечивающих одновременное включение единого поведения у разных частей сложной системы. Американский ученый Дж. Майхилл сформулировал эту задачу в виде следующей метафоры. Цепь стрелков, которые могут взаимодействовать только со своими непосредственными соседями, должна выстрелить одновременно. Два крайних стрелка — генерал и сержант — имеют только по одному соседу. Проблема состоит в построении автоматной модели, которая описывает их поведение. Функционирование всей цепочки происходит синхронно, по тактам. Выходом каждого автомата является его состояние — то есть эти автоматы являются автоматами Мура. В каждом следующем такте состояние каждого автомата является функцией его состояния в предыдущем такте и состояний его соседей в предыдущем такте. Все автоматы первоначально находятся в начальном состоянии, и после подачи сигнала генералом своему единственному соседу (спонтанный переход автомата-генерала в состояние «приготовиться») должны в одном и том же такте в будущем через конечное (но произвольное) число шагов все одновременно перейти в одно состояние «пли». Сложность проблемы состоит в том, что солдаты «не умеют считать» — иными словами, в соответствующей им автоматной модели число состояний не зависит от длины цепочки стрелков. Несмотря на это, при любом количестве солдат все солдаты цепочки (вместе с офицерами) должны перейти в свои заключительные состояния «пли» в одном и том же такте, номер которого, конечно, зависит от числа солдат в цепочке.
2. Построить конечный автомат, продающий пиво и выдающий сдачу. Автомат может принимать монеты достоинством 5 и 10 пенсов, а кружка пива стоит 15 пенсов. Кроме отверстий для приема монет и выдачи сдачи у автомата есть кнопки «Наливай» и «Сброс».
3. [27]. Построить автономный конечный автомат, выдающий выходную цепочку вида: $101001000100001\dots 10^n1\dots$.
4. Построить конечный автомат, выбрасывающий все комментарии из программы на языке C с учетом возможных строковых констант, которые, конечно, изменять нельзя. Комментарий в языке C — это любая строка, ограниченная скобками: `/* ... */` либо парой слешей `//` и концом строки. Внутри строковой константы могут встретиться константные символы. Они вводятся с двойными кавычками.
5. Построить конечный автомат, выдающий остаток от деления вводимого десятичного числа на 3.
6. Построить конечный автомат, выдающий результат деления вводимого пятиричного числа на 3. Результат выдается в форме: <частное> (<остаток>). Чис-

ло вводится со старших разрядов и заканчивается маркером конца «#». Результат представляется также в пятиричной системе счисления.

7. Построить конечный автомат, выдающий остаток от деления вводимого троичного числа на 4. Построить два варианта автомата: для случая, когда число вводится со старших разрядов, и для случая, когда число вводится с младших разрядов.
8. Программа `wc` в UNIX считает во входном потоке слова, строки и символы. Поток завершается кодом `eof`. Слова разделены пробелами, символами табуляции «\t» и символами перевода строки «\n». Строки разделены символами перевода строки «\n». Построить конечный автомат и на его основе программу `wc`.
9. Построить конечный автомат, выбрасывающий лишние пробелы в тексте.
10. Построить конечный автомат, добавляющий бит нечетности к цепочке из «0» и «1».
11. При разбиении на информационные кадры при передаче потока информации по каналам связи в компьютерных сетях используется метод вставки начального и заключительного флагов (обрамления кадра). Каждый кадр может содержать произвольное число информационных битов, он начинается и заканчивается специальной последовательностью 01111110, называемой флагом. Границы между кадрами на приемном конце могут быть однозначно распознаны, если в потоке битов внутри кадра не встретится 6 подряд идущих единиц. Для того чтобы произвольный поток двоичной информации можно было передавать таким методом, используется прием, называемый «вставка бита» (bit stuffing).
 - а) Построить конечный автомат — кодировщик, подготавливающий информационный кадр для передачи этим протоколом: вставляющий дополнительный нуль в информационный поток после каждого пяти подряд идущих единиц (например, поток битов 01111100101111110 автомат преобразует в 011111010010111110110; здесь выделены автоматически вставленные биты).
 - б) Построить конечный автомат — декодировщик, выполняющий обратную операцию: убирающий лишние нули после получения и распознавания информационного кадра на приемном конце (например, преобразующий поток битов 011111010010111110110 в 01111100101111110) и распознающий начальный и заключительный флаги.
12. Для конечного автомата, заданного графом переходов, построить его реализацию:
 - аппаратную;
 - программную в виде схемы программы;
 - программную в виде программы на языке высокого уровня;
 - программную в виде программы, оперирующей с таблицей переходов и выходов.

13. Для автомата, реализующего блок управления микрокалькулятором, определить содержимое его сумматора после нажатия следующих клавиш: 2, 2, /, 2, −, 2+, /, =, −, 2, =, =, 2, /, 2, =, =.
14. Решение проблемы взаимного исключения параллельных процессов с локальными флагами, рассматриваемое в примере 3.8, оказалось некорректным, потому что каждый процесс сначала устанавливал свой флаг в 1, предупреждая о желании войти в свой критический интервал, а потом ожидал освобождения критического интервала, анализируя значение флага другого процесса. Проверить корректность другого решения этой проблемы, в котором эти две операции выполняются в обратном порядке.
15. Построить автомат Мура, управляющий светофором автоматического регулирования транспорта на обычном перекрестке. Движение регулируется так, что в одном направлении разрешено движение τ_1 секунд, а в другом — τ_2 секунд. (Указание. Единственным входным событием автомата является событие завершения тайм-аута. Поэтому это — автономный автомат, который выдает циклическую последовательность выходов.)
16. (Автоматический светофор [3].) Рассмотрим проблему регулирования транспорта на Т-образном перекрестке (рис. 3.25), на котором основной поток транспорта идет с запада на восток, а с юга подходит второстепенная дорога.

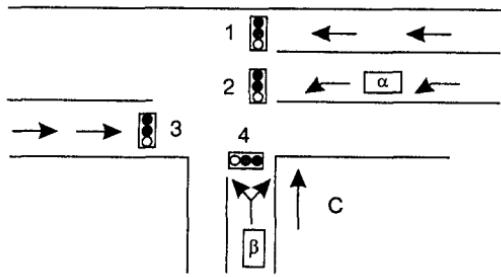


Рис. 3.25. Т-образный перекресток

Четыре светофора управляют движением транспорта, каждый может показывать один из трех сигналов: К, Ж, З. Очевидно, что они должны управляться согласованно, причем так, чтобы не создавалась аварийная ситуация (например, не было бы сигнала «Зеленый» на всех четырех светофорах). Устройство управления выдает четверки сигналов, например, <К; З; Ж; З> — на первом светофоре красный, на втором и четвертом — зеленый, а на третьем — желтый.

Считая, что транспорт в направлении второстепенной дороги движется редко, построить конечный автомат Мура, управляющий перекрестком по запросам транспорта. Запросы транспорта автоматически передаются с помощью сенсоров α и β . На этих сенсорах устанавливается значение 1, если в соответствующем месте появляется машина. Входами в автомат управления служат пары

$<0, 0>$; $<0, 1>$; $<1, 0>$; $<1, 1>$ состояний сенсоров α и β , а выходами – четверки сигналов светофоров.

17. Для протокола PAR (Positive Acknowledge with Retransmission) построить программную спецификацию и найти некорректный сценарий работы протокола – незамеченные приемником дублирование принятого сообщения.
18. Построить автоматическую модель алгоритма функционирования программного калькулятора, включенного в стандарт DOS.
19. Алгоритм Хаффмена построения оптимального двоичного префиксного кода переменной длины, в котором никакая кодовая последовательность не является началом другой кодовой последовательности, исходит из знания вероятности встречаемости символов в представляемом тексте. Алгоритм этот рекурсивный. Если в алфавите два символа, то кодируем их 0 и 1. Если больше, то объединяем два наиболее редких символа (пусть это x и y) в один новый символ (назовем его z), решаем получившуюся задачу, после чего разделяем этот новый символ z , добавляя 0 и 1 к кодовой последовательности, присвоенной этому новому символу для кодирования x и y . Декодирование кода Хаффмена осуществляется с помощью конечного автомата.

Для примера, пусть алфавит состоит из пяти символов: $V = \{a, d, c, d, e\}$ с вероятностями их встречаемости: 0,3(a), 0,25(b), 0,2(c), 0,15(d), 0,1(e). Объединяем d и e в один символ de , получаем новую задачу: 0,3(a), 0,25(de), 0,25(b), 0,2(c). На следующем шаге 0,45(bc), 0,3(a), 0,25(de). Далее, 0,55(ade), 0,45(bc).

Теперь у нас только два символа, ade и bc . Сопоставляем группе ade код 0, группе bc код 1. Далее, при «обратном ходе» расщепляем группу ade на a и de , сопоставляя символу a код 00, а группе de код 01; расщепляем группу bc на два символа, сопоставляя символу b код 10, а символу c – код 11 и т. д. Окончательно, получим следующий код для символов алфавита V : 00(a), 10(b), 11(c), 010(d), 011(e). Этот код оптимален (в среднем имеет наименьшую длину) для цепочек кодов символов с указанными выше вероятностями их встречаемости.

Построить конечный автомат, перерабатывающий входную двоичную цепочку, представляющую код символов данного примера, в последовательность символов алфавита V . Проверить работу автомата на входной цепочке 0011010011011010.

20. Разработать процедуру минимизации неполнотью определенного конечного автомата, то есть такого, у которого некоторые переходы и выходы не определены (считая, что значения функций переходов и выходов на соответствующих парах аргументов $<\text{состояние}, \text{входной сигнал}>$ являются безразличными). (См. [13], с. 98.)
21. Построить полную решетку разбиений множества из четырех элементов; двух элементов; одного элемента.
22. Построить фактор-алгебры $A_N/\mathbf{0}$, A_N/μ и A_N/\mathbf{I} для алгебры $A_N = <\mathbf{N}; +>$ – алгебры целых положительных чисел с одной операцией сложения. Разбиение μ объединяет в один класс все числа, равные по модулю 5. Какие конгруэнции имеет алгебра A_N ?

23. Построить последовательную декомпозицию автомата A_1 по конгруэнции $\pi_9 = \{\langle 0, 5, 6, 7 \rangle; \langle 1, 2, 3, 4 \rangle\}.$
24. Построить параллельную декомпозицию автомата A_1 по двум ортогональным конгруэнциям $\pi_4 = \{\langle 0, 7 \rangle; \langle 1, 2 \rangle; \langle 3, 4, 5, 6 \rangle\}$ и $\pi_5 = \{\langle 0, 5 \rangle; \langle 1, 3 \rangle; \langle 2, 4 \rangle; \langle 6, 7 \rangle\}.$

ГЛАВА 4 Автоматные языки

В данной главе конечные автоматы рассматриваются не как преобразователи информации, реагирующие на отдельные входные сигналы, а как распознаватели последовательностей входных сигналов. Мы рассматриваем, как конечные автоматы могут такие последовательности обрабатывать. Последовательности символов часто называют предложениями, а их множества — языками. Таким образом, конечный автомат можно рассматривать как устройство, выполняющее алгоритмические операции над языками. Простейшей (но нетривиальной) алгоритмической операцией, связанной с языком, является распознавание — выделение автомата всех входных предложений, принадлежащих языку. Фактически такое автоматное задание языка есть метод строгого конечного определения бесконечного множества предложений. Другая операция, которую может выполнять автомат наряду с распознаванием, — преобразование предложений языка в некоторый выход (например, выходную программу), что называется трансляцией. Те языки, которые могут быть распознаны и отранслированы конечными автоматами, называются автоматными языками. Оказывается, что автоматные языки составляют узкий, самый простой класс языков. Обычные языки программирования являются более сложными, они не могут быть описаны этой моделью. Однако значение автоматных языков очень велико: кроме того, что многие языковые процессоры построены на основе модели конечного автомата, все трансляторы языков высокого уровня содержат препроцессор, выделяющий и обрабатывающий лексемы — синтаксические единицы, задаваемые автоматными языками.

В результате изучения материала этой главы читатель должен усвоить:

- общее представление о языках, их распознавании и трансляции;
- методы распознавания автоматных языков;
- проблемы минимизации и проверки эквивалентности конечноавтоматных распознавателей;

- методы реализации распознавателей и трансляторов автоматных языков;
- примеры построения трансляторов автоматных языков;
- регулярные множества и регулярные выражения, их связь с автоматными языками.

ЯЗЫКИ

Определение 4.1. Конечное множество элементов будем называть словарем, элементы словаря — символами, а последовательности символов словаря — цепочками или предложениями. Множество предложений назовем языком. Язык над словарем V будем обозначать L_V , или просто L , если V очевидно.

Пусть V — словарь. Обозначим V^* — множество всех возможных цепочек, составленных из символов словаря V . Если $V = \{a, b, c\}$, то $V^* = \{\epsilon, a, b, c, aa, ab, cba, ccaba, \dots\}$, где ϵ — пустая цепочка, то есть цепочка, вовсе не содержащая символов. Очевидно, что хотя V конечно, V^* — бесконечное счетное множество. Язык — это просто некоторое подмножество V^* : $L \subseteq V^*$. Всего языков над словарем V (как подмножеств счетного множества V^*) бесконечное число; это множество мощности континуум.

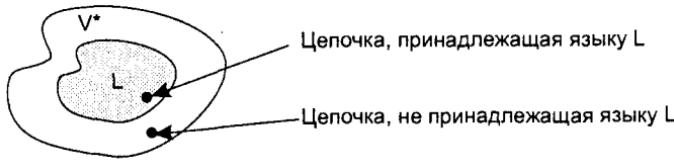


Рис. 4.1. Язык как подмножество цепочек над словарем V

Пример 4.1

Рассмотрим несколько примеров языков. Мы будем писать символы в кавычках только в том случае, если их нужно явно выделить.

1. $V_1 = \{a, b, c\}; L_1 = \{abc, cc\}$. Очевидно, $cc \in L_1$, $cba \notin L_1$.
2. $V_2 = \{a, b, c\}; L_2 = \emptyset$. Очевидно, $cc \notin L_2$.
3. $V_3 = \{a, b, c\}; L_3 = V^*$. Очевидно, $acc \in L_3$, так же как и любая цепочка из этих символов.
4. $V_4 = \{a, b, c\}; L_4 = \{a^nbc^n | n \geq 0\}$. Очевидно, $aabcc \in L_4$, $cbaa \notin L_4$.
5. $V_5 = \{a, b, c\}; L_5 = \{a^nbc^m | n, m \geq 0\}$. Очевидно, $aaaabcc \in L_5$, $cbaa \notin L_5$.
6. $V_6 = \{a, b, c\}; L_6 = \{x | \text{в цепочке } x \text{ количества вхождений } a, b \text{ и } c \text{ равны}\}$. Очевидно, $ccbaba \in L_6$, $ccba \notin L_6$.
7. $V_7 = \{0, 1\}; L_7 = \text{множество четных двоичных чисел}$. Очевидно, $100 \in L_7$, $011 \notin L_7$.

8. $V_8 = \{+, -, 0, \dots, 9\}$; L_8 = множество целых констант. Очевидно, $-67 \in L_8$, $+234 - 3 \notin L_8$, $+2+3\dots 4 \notin L_8$.
9. $V_9 = \{(.,)\}$; L_9 = множество правильных скобочных выражений. Очевидно, $((()) \in L_9$, $)((()) \notin L_9$.
10. $V_{10} = \{+, -, 0, \dots, 9, \cdot\}$; L_{10} = множество вещественных констант.
11. $V_{11} = \{a\}$; $L_{11} = \{\alpha \in V^* \mid \text{длина } \alpha \text{ равна } 2k, \text{ где } k = 0, 1, \dots\}$. Очевидно, $aaaa \in L_{11}$, $a \notin L_{11}$.
12. $V_{12} = \{a\}$; $L_{12} = \{\alpha \in V^* \mid \text{длина } \alpha \text{ равна } 2^k, \text{ где } k = 0, 1, \dots\}$. Очевидно, $aaaa \in L_{12}$, $a \in L_{12}$, $aaa \notin L_{12}$.
13. $V_{13} = \{\text{большой}, \text{ мощный}, \text{ тяжелый}, \dots, \text{кран}, \text{локомотив}, \text{контейнер}, \dots, \text{ставит}, \text{ведет}, \text{кладет}, \text{везет}, \dots\}$; L_{13} = русский язык.

Грамматики. Автоматные грамматики и языки

Сложность работы с языками состоит в том, что язык — это в общем случае бесконечное множество, а бесконечные объекты даже задавать трудно: их невозможно задать простым перечислением элементов. Механизмы задания языков имеют особую важность в теории формальных языков. Один из распространенных методов задания языка использует так называемый «определитель множества» вида $\{w \mid \text{утверждение о цепочке } w\}$, где w — это обычно выражение с параметрами, а *утверждение* определяет некоторые условия, накладываемые на параметры. Языки L_1 , L_5 , L_{11} и некоторые другие определены именно так. Но во многих случаях использовать этот метод трудно или просто невозможно. Например, как задать L_9 — множество правильных скобочных выражений, как определить множество всех идентификаторов языка программирования, как задать сам язык программирования, наконец, естественный (например, русский) язык?

Определение 4.2. Любой конечный механизм задания языка называется грамматикой.

Существует два типа грамматик (рис. 4.2): порождающие и распознавающие. Под порождающей грамматикой языка L понимается конечный набор правил, позволяющий строить все «правильные» предложения языка L и ни одного «неправильного». Распознавающая грамматика задает критерий принадлежности произвольной цепочки данному языку. Это фактически некоторый алгоритм, принимающий в качестве входа символ за символом произвольную цепочку над словарем V и дающий на выходе один из двух возможных ответов: «данная цепочка принадлежит языку L » либо «данная цепочка не принадлежит языку L ». В действительности этот алгоритм должен разделить все возможные входные цепочки на два класса: одни — принадлежащие языку L , а другие — не принадлежащие языку L (см. рис. 4.1).



Рис. 4.2. Порождающая и распознавающая грамматики

Роль распознавающей грамматики может выполнить конечный автомат без выхода. Свяжем с некоторыми состояниями автомата метку «Да», а с остальными — метку «Нет». Тогда все множество входных цепочек автомата разбьется на два класса: одни — приводящие автомат в одно из состояний, помеченных «Да», все другие — приводящие автомат в одно из состояний, помеченных «Нет».

Определение 4.3. Конечным автоматом-распознавателем называется пятерка объектов: $A = \langle S, X, s_0, \delta, F \rangle$, где:

S — конечное непустое множество (состояний);

X — конечное непустое множество входных сигналов (входной алфавит);

$s_0 \in S$ — начальное состояние;

$\delta: S \times X \rightarrow S$ — функция переходов;

$F \subseteq S$ — множество заключительных (финальных) состояний.

Определим обобщенную функцию переходов автомата $\delta^*: S \times X^* \rightarrow S$ точно так же, как в определении 3.3.

Определение 4.4. Конечный автомат-распознаватель $A = \langle S, X, s_0, \delta, F \rangle$ допускает входную цепочку $a \in X^*$, если a переводит его из начального в одно из заключительных состояний, то есть если $\delta^*(s_0, a) \in F$. Множество всех цепочек, допускаемых автоматом A , образует язык, допускаемый A .

Обозначим L_A язык, допускаемый автоматом $A = \langle S, X, s_0, \delta, F \rangle$.

Очевидно, $L_A = \{a \in X^* \mid \delta^*(s_0, a) \in F\}$.

Определение 4.5. Язык, для которого существует распознавающий его конечный автомат, называется автоматным языком.

Оказывается, не для всех языков существуют распознавающие их конечные автоматы. Иными словами, существуют и неавтоматные языки.

Графически вершины, соответствующие заключительным состояниям, изображаются кружком, выделенным жирной или двойной линией, а в написании будем помечать их символом $+$: s^+ . Для представления автомата-распознавателя будем использовать как таблицу переходов, так и граф переходов.

Пример 4.2

Построим распознаватели для всех языков, рассмотренных в примере 4.1.

1. $V_1 = \{a, b, c\}; L_1 = \{abc, cc\}$.

Полностью определенный граф переходов автомата, распознающего L_1 , приведен на рис. 4.3, а. Входные цепочки «abc» и «cc» (и только они) переводят автомат из начального в одно из заключительных состояний s_5 или s_3 . Очевидно, что любой конечный язык может быть задан конечным автоматом и поэтому *все конечные языки — автоматные*.

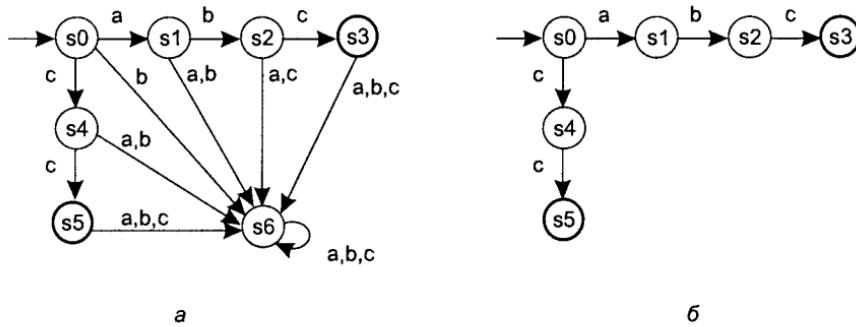


Рис. 4.3. Автоматы, распознающие L_1

Граф переходов этого же распознающего автомата, у которого, однако, график переходов неполностью определен, приведен на рис. 4.3, б. Здесь и далее будем считать, что если переход под воздействием некоторого входного сигнала a не определен в состоянии s конечного автомата-распознавателя, то это по умолчанию означает, что существует **незаключительное** состояние, в которое этот автомат переходит из s при подаче a , причем под воздействием всех входных сигналов автомат **не выходит** из этого незаключительного состояния. Это, конечно, и означает, что сигнал a в состоянии s не допускается (запрещен). В частности, в языке L_1 нет цепочек, начинающихся с b , поэтому сигнал b в состоянии s_0 запрещен; далее, если автомат находится в состоянии s_1 (после подачи сигнала a), не могут появиться сигналы b и c и т. д.

Продолжим построение автоматов, допускающих языки примера 4.1.

$$2. V_2 = \{a, b, c\}; L_2 = \emptyset.$$

Любой автомат с пустым множеством заключительных состояний допускает L_2 .

$$3. V_3 = \{a, b, c\}; L_3 = V_3^*.$$

Автомат с единственным состоянием, которое является заключительным, имеющий три перехода из этого состояния в него же, помеченные символами из V_3 , допускает L_3 .

$$4. V_4 = \{a, b, c\}; L_4 = \{a^nbc^n \mid n \geq 0\}.$$

Этот язык не является автоматным. На рис. 4.4 представлен один из вариантов попытки построить такой автомат, из которого видно, что автомат фактически

должен «подсчитывать» число символов a в первой части цепочки, чтобы число символов c было в точности ему равно. Но конечный автомат имеет лишь конечную память, поэтому он не может распознавать *все* цепочки L_4 . Это рассуждение, конечно, не доказательство. Дадим строгое доказательство неавтоматности L_4 .

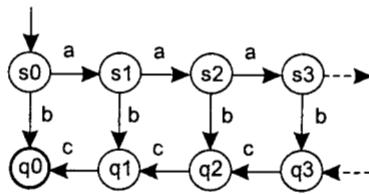
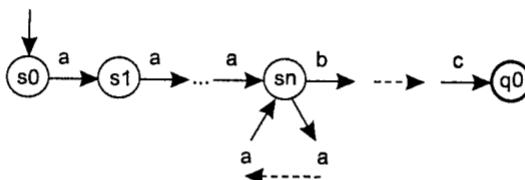


Рис. 4.4. Попытка построения автомата, распознающего L_4

Теорема 4.1. Язык $L_4 = \{a^nbc^n \mid n \geq 0\}$ неавтоматный.

Доказательство. Предположим противное, то есть что существует автомат A , который распознает все цепочки языка $L_4 = \{a^nbc^n \mid n \geq 0\}$ и не распознает ни одной цепочки, не принадлежащие ему. Пусть число состояний автомата A равно k . Рассмотрим цепочку a^kbc^k . Под воздействием первой части цепочки — k последовательных символов « a » — автомат A не может не вернуться в какое-либо из состояний, в котором он уже находился. Пусть это будет состоянием s_n , и длина цикла — цепочки « $aaaa \dots a$ », под воздействием которой автомат переходит из s_n в s_n , равна t , причем $t > 0$.



Тогда цепочка a^{k-r} так же, как и цепочка $a^{k-r}a^r = a^k$, переводит автомат A из начального состояния в состояние s_n . Но поскольку цепочка $a^kbc^k = a^{k-r}a^rbc^k$ переводит A из начального в заключительное состояние, то и цепочка $a^{k-r}bc^k$ переводит A из начального в заключительное состояние. Но поскольку $r > 0$, то это значит, что A распознает цепочку $a^{k-r}bc^k$, которая не принадлежит языку $L_4 = \{a^nbc^n \mid n \geq 0\}$. Итак, мы пришли к противоречию, предположив существование конечного автомата, распознающего L_4 .

Более общий критерий, позволяющий сделать заключение, является ли автоматным произвольный язык, дает так называемая *Лемма о накачке*.

5. $V_5 = \{a, b, c\}; L_5 = \{a^nbc^m \mid n, m \geq 0\}$.

В отличие от предыдущего языка L_5 — автоматный. Автомат (рис. 4.5) только с двумя состояниями распознает L_5 . Простота этого распознавателя объясняется тем, что он не должен отслеживать количества символов а и с во входных цепочках.

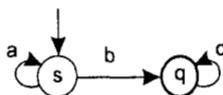


Рис. 4.5. Автомат, распознающий L_5

6. $V_6 = \{a, b, c\}; L_6 = \{x \mid \text{в цепочке } x \text{ количества вхождений } a, b \text{ и } c \text{ равны}\}$. L_6 не автоматный.
7. $V_7 = \{0, 1\}; L_7 = \text{множество четных двоичных чисел}$. Автомат (рис. 4.6) с двумя состояниями распознает L_7 .

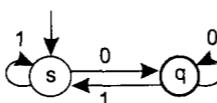


Рис. 4.6. Автомат, распознающий L_7

8. $V_8 = \{+, -, 0, \dots, 9\}; L_8 = \text{множество целых констант}$. Автомат (рис. 4.7) с тремя состояниями распознает L_8 . Вместо конечного числа цифр на переходах здесь поставлена просто буква ц, обозначающая любую цифру.

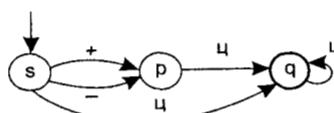
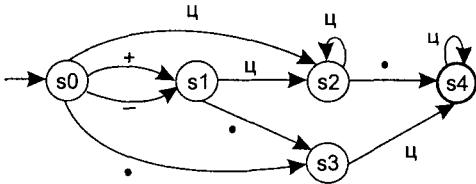
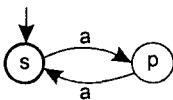


Рис. 4.7. Автомат, распознающий L_8

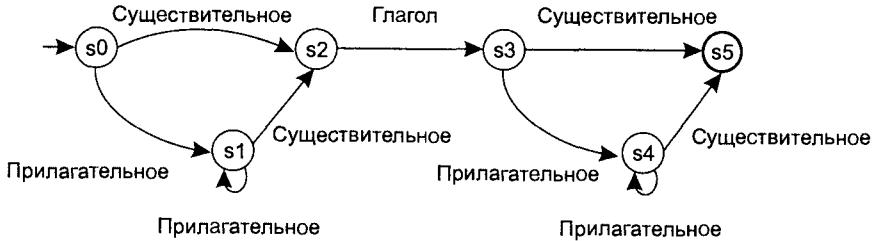
9. $V_9 = \{(,)\}; L_9 = \text{множество правильных скобочных выражений}$. L_9 не автоматный. Распознавание вложенных скобок требует запоминания произвольных объемов информации: глубина вложенности может быть любой.
10. $V_{10} = \{+, -, 0, \dots, 9, «.»\}; L_{10} = \text{множество вещественных констант}$. Автомат (рис. 4.8) распознает L_{10} . Заметьте, что автомат правильно распознает константы, в которых можно не писать знак и либо целая, либо дробная часть может быть опущена, но не обе сразу. Десятичная точка как признак вещественного числа обязательна.

Рис. 4.8. Автомат, распознающий L_{10}

11. $V_{11} = \{a\}; L_{11} = \{\alpha \in V^* \mid \text{длина } \alpha \text{ равна } 2k, \text{ где } k = 0, 1, \dots\}$. Автомат (рис. 4.9) распознает L_{11} .

Рис. 4.9. Автомат, распознающий L_{11}

12. $V_{12} = \{a\}; L_{12} = \{\alpha \in V^* \mid \text{длина } \alpha \text{ равна } 2^k, \text{ где } k = 0, 1, \dots\}$. L_{12} не автоматный.
13. $V_{13} = \{\text{большой}, \text{мощный}, \text{тяжелый}, \dots, \text{кран}, \text{локомотив}, \text{контейнер}, \dots, \text{ставит}, \text{ведет}, \text{кладет}, \text{везет}, \dots\}$; $L_{13} = \text{русский язык}$. Оказывается, русский язык, как и другие естественные языки, не автоматный. Естественные языки описываются намного более сложными моделями, чем конечные автоматы. Однако очень узкое подмножество русского языка может быть задано конечным автоматом. Например, граф переходов.



задает структуру большого числа предложений вида: «Большой мощный кран грузит контейнер» или «Маленький зеленый паровозик везет длинный груженый тяжелый состав». Если понимать «существительное», «прилагательное» и «глагол» на переходах графа как сокращенное обозначение конечного числа соответствующих элементов словаря V_{13} , то этот график действительно является графиком конечного автомата-распознавателя подмножества русского языка.

Лемма о накачке

В предыдущем разделе было показано, что не все языки представимы конечными языками. Лемма о накачке является важным теоретическим результатом, по-

зволяющим во многих случаях проверить, является ли данный язык автоматным. Поскольку все конечные языки являются автоматными, эту проверку имеет смысл делать только для бесконечных языков.

Итак, пусть L содержит бесконечное число цепочек. Предположим, что L распознается конечным автоматом A с n состояниями. Для проверки автоматности языка L выберем произвольную цепочку этого языка α , которая имеет длину n . Если автомат распознает L , то цепочка α допускается этим автоматом, то есть в автомате A существует путь длины n из начального в одно из заключительных состояний, помеченный символами цепочки α . Путь этот не может быть простым, он должен проходить ровно через $n+1$ состояние, в то время как автомат A имеет n состояний. Это значит, что этот путь проходит по меньшей мере два раза через одно и то же состояние автомата A , то есть на этом пути есть цикл с повторяющимся состоянием. Пусть это повторяющееся состояние q_k . Разделим цепочку α на три части, так что $\alpha = uvw$, где u — подцепочка, переводящая A из начального состояния в q_k , v — подцепочка, переводящая A из состояния q_k опять в состояние q_k , и w — подцепочка, переводящая A из состояния q_k в заключительное состояние. Заметим, что как u , так и w могут быть пустыми, но подцепочка v не может быть пустой. Но тогда очевидно, что автомат A должен допускать также и цепочку $uvvw$, поскольку повторяющаяся подцепочка v снова проходит по циклическому пути из q_k в q_k , а также и цепочку $uvvvw$, и любую цепочку вида $uv^i w$ для любого $i \geq 0$. Это рассуждение и составляет доказательство Леммы о накачке. Термин «накачка» в названии леммы отражает возможность многократного повторения некоторой подстроки в любой строке подходящей длины любого бесконечного автоматного языка. Более формально лемма о накачке формулируется так.

Лемма о накачке (I). Пусть L — автоматный язык над алфавитом V . Тогда:

$(\exists n \in \mathbb{N})(\forall \alpha \in L : |\alpha| \geq n)(\exists u, v, w \in V^*) :$

$$\left[\alpha = uvw \& |uv| \leq n \& |v| \geq 1 \& (\forall i \in \mathbb{N})(uv^i w \in L) \right]$$

Другая форма этой леммы, которую иногда удобнее применять, чтобы показать неавтоматность некоторого языка, записывается так.

Лемма о накачке (II). Пусть L — некоторый язык над алфавитом V . Если:

$(\forall n \in \mathbb{N})(\exists \alpha \in L : |\alpha| \geq n)(\forall u, v, w \in V^*) :$

$$\left[\alpha = uvw \& |uv| \leq n \& |v| \geq 1 \& (\exists i \in \mathbb{N})(uv^i w \notin L) \right],$$

то L — не автоматный.

Лемма — это теорема, использующаяся для доказательства других теорем. Рассмотрим пример доказательства теоремы, устанавливающей неавтоматность конкретного языка, с помощью леммы о накачке.

Теорема 4.2. Язык $L = \{\beta\bar{\beta}^R \mid w \in \{0,1\}^*\}$ — неавтоматный (здесь β^R — это цепочка, обратная к β).

Доказательство. Предположим противное, а именно, что язык L — автоматный и он распознается некоторым конечным автоматом A с n состояниями. Рассмотрим цепочку α этого языка, такую что $|\alpha| \geq n$. Эта цепочка по предположению допуска-

ется автоматом А, и поскольку путь, помеченный этой цепочкой на графе переходов А, содержит по крайней мере один цикл, то цепочку α можно представить как конкатенацию трех подцепочек, $\alpha = uvw$, причем цепочка v не пуста. Язык L будет автоматным, если А вместе с $\alpha = uvw$ допускает также и $uvvw$, и $uvvvw$ и т. д. В какую из подцепочек u, v или w может входить символ c? Если c входит в подцепочки u или w, то «накачка» v нарушает симметрию допускаемой цепочки, если c входит в v, то «накачка» v приведет к удвоению, утроению и т. п. символа c, что также недопустимо. Таким образом, в допускаемой автоматом А достаточно длинной цепочке α не существует непустой подцепочки, повторение которой произвольное число раз сохраняет структуру $\beta\gamma^k$, то есть также дает цепочку, допускаемую этим автоматом. Следовательно, язык L — неавтоматный.

Эквивалентность и минимизация конечноавтоматных распознавателей

Очевидно, что два распознавателя следует считать эквивалентными, если они распознают один и тот же язык. Проблема эквивалентности двух распознавателей А и В состоит в нахождении ответа на вопрос: совпадают ли языки, распознаваемые А и В. Проблема минимизации распознавателя А состоит в нахождении такого распознавателя А', который, распознавая тот же язык, что и А, имеет минимально возможное число состояний. Оказывается, что проблемы эквивалентности и минимизации конечных автоматов-распознавателей только в некоторых чертах отличаются от таковых для конечных автоматов-преобразователей. Поэтому мы дадим основные утверждения без доказательств: они полностью аналогичны таковым для автоматов преобразователей.

Определение 4.6. Пусть $A = \langle S_A, X, s_{0A}, \delta_A, F_A \rangle$ и $B = \langle S_B, X, s_{0B}, \delta_B, F_B \rangle$ — два конечных автомата-распознавателя с одним и тем же входным алфавитом. Прямым произведением их называется автомат $A \times B = \langle S_A \times S_B, X, (s_{0A}, s_{0B}), \delta_{A \times B}, F_A \times F_B \rangle$, где $(\forall s_A \in S_A)(\forall s_B \in S_B)(\forall x \in X)\delta_{A \times B}((s_A, s_B), x) = (\delta_A(s_A, x), \delta_B(s_B, x))\infty$.

Как и для автоматов-преобразователей входных цепочек, прямое произведение автоматов-распознавателей — это просто два таких автомата с одними и теми же входами, работающими синхронно.

Теорема 4.3. (Теорема Мура.) Два конечных автомата-распознавателя $A = \langle S_A, X, s_{0A}, \delta_A, F_A \rangle$ и $B = \langle S_B, X, s_{0B}, \delta_B, F_B \rangle$ с одинаковым входным алфавитом являются эквивалентными тогда и только тогда, когда в их прямом произведении $A \times B$ для любого достижимого состояния (s_A, s_B) справедливо: $s_A \in F_A \equiv s_B \in F_B$.

Иными словами, два автомата-распознавателя эквивалентны тогда и только тогда, когда при их синхронном функционировании под воздействием одних и тех же входных символов они на каждом шаге всегда попадают либо оба в заключитель-

ные, либо оба в незаключительные состояния. Справедливость этой теоремы очевидна.

Определение 4.7. Два состояния p и q конечного автомата-распознавателя $A = \langle S, X, s_0, \delta, F \rangle$ называются эквивалентными, если $(\forall \alpha \in X^*) \delta^*(p, \alpha) \in F \equiv \delta^*(q, \alpha) \in F$.

Для автомата рис. 4.3, б состояния s_3 и s_5 эквивалентны: любая входная цепочка, поданная на автомат, находящийся в состоянии s_3 , будет недопустима, точно так же, как и в случае, когда автомат находится в состоянии q_2 . Эквивалентные состояния можно объединить в один класс и построить новый автомат, состояниями которого являются классы эквивалентных состояний. Если мы можем определить на множестве состояний автомата максимально возможное разбиение на классы эквивалентности, то, выбирая его классы эквивалентности как новые состояния, получим минимальный автомат, эквивалентный исходному.

Алгоритм минимизации конечного автомата-распознавателя состоит в построении на множестве его состояний таких разбиений $\pi_0, \pi_1, \dots, \pi_\infty$, что в один класс разбиения π_k попадают k -эквивалентные состояния, то есть находящиеся в отношении эквивалентности \approx_k . Если $\neg(p \approx_k q)$, то p и q назовем k -различимыми. Из определения 4.5: $p \approx_k q \Leftrightarrow (\forall \alpha \in X^{k^*}) \delta^*(p, \alpha) \in F \equiv \delta^*(q, \alpha) \in F$. При подаче пустой цепочки на вход автомата (цепочки длиной 0) автомат остается в том же состоянии, в котором он находился. Поэтому разбиение π_0 , состоит из двух блоков: в один блок попадают все заключительные состояния, а в другой — все незаключительные состояния. Оно является исходным при построении цепочки разбиений $\pi_0, \pi_1, \dots, \pi_\infty$. Определив, как строить следующее разбиение из предыдущего, начиная с π_0 , мы сможем последовательно построить всю цепочку.

Теорема 4.4. Пусть в конечном автомате-распознавателе $p \approx_k q$, $k \geq 0$. Для того чтобы $p \approx_{k+1} q$, необходимо и достаточно, чтобы $(\forall x \in X) \delta(p, x) \approx_k \delta(q, x)$.

Доказательство этой теоремы полностью аналогично доказательству теоремы 3.2, и мы его здесь опускаем.

Очевидно, что если p и q $k+1$ -эквивалентны, то они k -эквивалентны. Иными словами, блоки разбиения π_{k+1} являются подблоками разбиения π_k . Поскольку число состояний конечно, может быть только конечное число последовательно уменьшающихся разбиений π_k , начиная с максимального разбиения π_0 , содержащего два блока. Более того, очевидно, что их не больше, чем N — число состояний автомата. Однако последовательное построение уменьшающихся разбиений π_r можно не продолжать дальше, как только два последовательных разбиения совпадают. Доказательство следующей теоремы также аналогично соответствующей ей теореме для конечных автоматов-преобразователей. Мы даем ее без доказательства.

Теорема 4.5. Пусть $\pi_{k+1} = \pi_k$. Тогда для любого $i > k$ $\pi_i = \pi_k$.

Пример 4.3

Минимизация конечного автомата, заданного таблицей переходов 4.1, проводится последовательным построением разбиений π_0, π_1, \dots . Начальное состояние здесь 0, заключительные — 1, 3, 7.

Таблица 4.1

			π_0		π_1	
	δ		δ		δ	
	a	b	a	b	a	b
0	3	2	B_0	A_0	D_1	B_1
1+	6	5	A_0	A_0		
2	4	7	A_0	B_0	B_1	D_1
3+	1	7	B_0	B_0	C_1	D_1
4	6	3	A_0	B_0	B_1	D_1
5	3	9	B_0	A_0	D_1	B_1
6	9	3	A_0	B_0	B_1	D_1
7+	1	3	B_0	B_0	C_1	D_1
8	7	6	B_0	A_0	D_1	B_1
9	4	7	A_0	B_0	B_1	D_1

Очевидно, что начальное разбиение представляет собой два блока, включающие один заключительные, а другой — незаключительные состояния. Поэтому $\pi_0 = \{A_0 = <0, 2, 4, 5, 6, 8, 9>; B_0 = <1, 3, 7>\}$.

Разбиение π_1 в один блок объединяет те состояния, которые нельзя различить при подаче цепочек длиной 1, то есть те, которые под воздействием одного и того же входного сигнала переходят в один и тот же блок разбиения π_0 . Поэтому

$$\pi_1 = \{A_1 = <0, 5, 8>; B_1 = <2, 4, 6, 9>; C_1 = <1>; D_1 = <3, 7>\}.$$

Следующее разбиение π_2 в один блок объединяет те состояния, которые нельзя различить при подаче цепочек длиной 2, то есть те, которые под воздействием одного и того же входного сигнала переходят в один и тот же блок предыдущего разбиения π_1 . Итак

$$\pi_2 = \{A_2 = <0, 5, 8>; B_2 = <2, 4, 6, 9>; C_2 = <1>; D_2 = <3, 7>\}.$$

Таблица 4.2

	δ	
	a	b
$<0, 5, 8>$	$<3, 7>$	$<2, 4, 6, 9>$
$<1>^+$	$<2, 4, 6, 9>$	$<0, 5, 8>$
$<2, 4, 6, 9>$	$<2, 4, 6, 9>$	$<3, 7>$
$<3, 7>^+$	$<1>$	$<3, 7>$

Разбиение π_2 совпадает с разбиением π_1 . На основании теоремы 4.6 искомое разбиение π_∞ совпадает с π_1 . Таким образом, минимальный автомат с эквивалентным

поведением имеет 4 состояния, представляющих блоки разбиения π_1 . Таблица переходов этого минимизированного автомата представлена в табл. 4.2. Начальным состоянием здесь является состояние $\langle 0, 5, 8 \rangle$, заключительными — два состояния $\langle 1 \rangle$ и $\langle 3, 7 \rangle$.

Недетерминированные конечноавтоматные распознаватели

Важную роль в теории языков играют недетерминированные распознаватели, которые являются обобщением детерминированных. Недетерминированные конечные автоматы-распознаватели могут быть двух типов: либо существует переход, помеченный пустой цепочкой ϵ , либо из одного состояния выходят несколько переходов, помеченных одним и тем же символом. На рис. 4.10, а представлены оба случая. Переход из состояния s_0 , помеченный ϵ , говорит о том, что в любой момент, если автомат находится в состоянии s_0 , он спонтанно, без подачи входного сигнала, может оказаться в состоянии s_1 (а может и остаться в состоянии s_0). Два перехода из состояния s_1 , помеченные a , разрешают автомatu переход в любое из состояний s_2 или s_3 под воздействием a . Причем один из этих переходов в этом примере автомата приводит в заключительное состояние, а другой — нет.

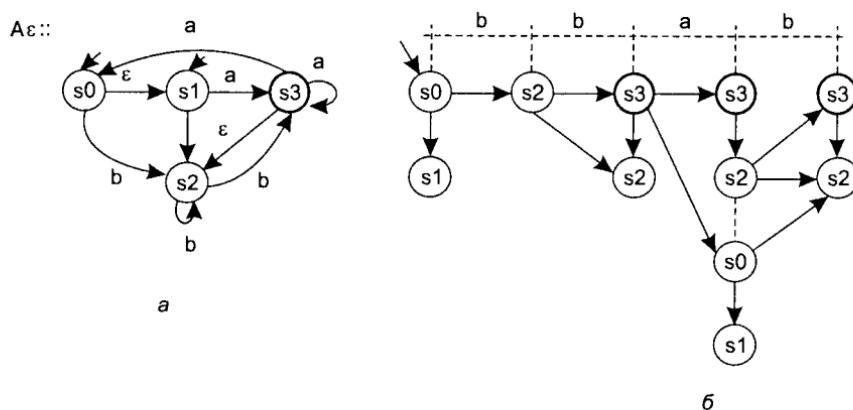


Рис. 4.10. Недетерминированный конечный автомат-распознаватель с ϵ -переходами (а); возможные поведения этого автомата при подаче входной цепочки bbab (б)

Определение 4.8. Недетерминированным конечным автоматом-распознавателем называется пятерка $A = \langle S, X, S_0, \delta, F \rangle$, где

S — конечное непустое множество (состояний);

X — конечное непустое множество входных сигналов (входной алфавит);

$S_0 \subseteq S$ — множество начальных состояний;

$\delta: S \times (X \cup \epsilon) \rightarrow 2^S$ — функция переходов. Здесь 2^X обозначен булсан X , то есть множество всех подмножеств множества X ;

$F \subseteq S$ — множество заключительных (финальных) состояний.

Как понимать такой недетерминированный распознаватель? Как определить его поведение?

Нельзя считать, что такой автомат ведет себя абсолютно неопределенно. Автомат на рис. 4.10, а, находясь в начальном состоянии s_0 , может без подачи входного сигнала перейти в состояние s_1 . Если на его вход подать сигнал b , то из состояния s_0 автомат переходит в состояние s_2 , а из состояния s_1 нет возможных путей. При подаче на вход следующего сигнала b автомат из состояния s_2 может перейти в любое из двух состояний, s_2 и s_3 , причем из состояния s_3 он может спонтанно перейти также в состояние s_2 . На рис. 4.10, б показано возможное поведение этого автомата при подаче входной цепочки $bbab$. На этой диаграмме хорошо видно, что автомат спонтанно может выполнить переход, помеченный пустой цепочкой ϵ (такие переходы определены на диаграмме вертикальными ребрами), а каждый переход, помеченный входным символом, может быть выполнен только при подаче этого входа.

Как определить основное понятие «распознавание входной цепочки» для недетерминированных автоматов? Вопрос этот нетривиальный, на него может быть несколько ответов в зависимости от приложения этой модели. Подход, который принят в теории формальных языков, состоит в следующем.

Определение 4.9. Недетерминированный конечный автомат распознает входную цепочку σ , если существует путь, помеченный символами цепочки σ , из начального в одно из заключительных состояний автомата, возможно, с учетом спонтанных переходов. Недетерминированный конечный автомат распознает язык L , если он распознает все цепочки этого языка, и только их.

Распознает ли автомат рис. 4.10, а входную цепочку bb ? Да, поскольку под воздействием этой цепочки он может перейти из s_0 в заключительное состояние s_3 . Допустима и цепочка $bbab$, поскольку существует путь (см. рис. 4.10, б).

$s_0 \xrightarrow{b} s_2 \xrightarrow{b} s_3 \xrightarrow{a} s_3 \xrightarrow{\epsilon} s_2 \xrightarrow{b} s_3$

С другой стороны, ни пустой цепочки ϵ , ни цепочки b этот автомат не допускает. Рисунок 4.11 представляет детерминированный автомат, эквивалентный изображенному на рис. 4.10 недетерминированному автоматау.

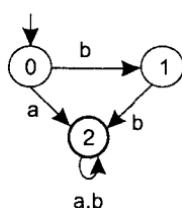


Рис. 4.11. Детерминированный автомат, эквивалентный изображенному на рис. 4.10 недетерминированному автоматау

Таким образом, в нашем примере для недетерминированного автомата существует эквивалентный ему детерминированный конечный автомат, распознающий тот же самый язык. Оказывается, что это — общее правило: недетерминированные конечноавтоматные распознаватели распознают то же множество языков, что и детерминированные.

Теорема 4.6. Для любого недетерминированного конечного автомата-распознавателя существует эквивалентный ему детерминированный конечный автомат-распознаватель.

Доказательство этой теоремы проведем конструктивно, то есть предложим алгоритм построения по заданному недетерминированному конечному автомату-распознавателю эквивалентного ему детерминированного конечного автомата-распознавателя. Сначала покажем, как недетерминированный автомат привести к автомату без ϵ -переходов, а потом — как для недетерминированного автомата без ϵ -переходов построить эквивалентный ему детерминированный автомат, допускающий тот же язык.

Пусть $A_\epsilon = \langle S, X, S_0, \delta_\epsilon, F \rangle$ — недетерминированный конечный автомат с ϵ -переходами, причем $\delta_\epsilon: S \times (X \cup \{\epsilon\}) \rightarrow 2^S$. Покажем, как привести A_ϵ к недетерминированному автомату $A = \langle Q, X, Q_0, \delta, F \rangle$ без ϵ -переходов.

Определение 4.10. ϵ -замыканием состояния $s \in S$, обозначаемым $\Xi(s)$, назовем множество всех состояний, которые достижимы из s без подачи входного сигнала.

Множеству $\Xi(s)$ принадлежит само состояние s и все те состояния, которые достижимы из s по ϵ -переходам. Для автомата, изображенного на рис. 4.10:

$$\Xi(s0) = \{s0, s1\}, \Xi(s1) = \{s1\}, \Xi(s2) = \{s2\}, \Xi(s3) = \{s3, s2\}.$$

Множеством состояний A являются ϵ -замыкания состояний A_ϵ . Для автомата на рис. 4.10 множество состояний $\{\Xi(s0), \Xi(s1), \Xi(s2), \Xi(s3)\}$.

Множеством начальных состояний A является $Q_0 = \bigcup_{q \in \Xi(s0)} \Xi(q)$: все состояния, достижимые из любого начального без подачи входного сигнала, также, очевидно, являются начальными. В автоматах, изображенном на рис. 4.10, начальными состояниями являются $\Xi(s0)$ и $\Xi(s1)$.

Множеством заключительных состояний A являются такие ϵ -замыкания состояний A_ϵ , в которые входят заключительные состояния A_F : из каждого такого состояния без подачи входного сигнала можно оказаться в заключительном состоянии. Для автомата на рис. 4.10 множество заключительных состояний $\{\Xi(s3)\}$.

Функция переходов δ автомата A определяется так:

$$(\forall \Xi(s)) (\forall a \in X) \delta(\Xi(s), a) = \bigcup_{q \in \delta_\epsilon(\Sigma(s), a)} \Xi(q).$$

Иными словами, при воздействии входного сигнала a автомат A переходит из ϵ -замыкания состояния s в ϵ -замыкания всех тех состояний q , в которые A_ϵ переходит под воздействием a из всех состояний, достижимых из s по воздействию ϵ . Для автомата на рис. 4.10 таблица переходов эквивалентного недетерминированного автомата без ϵ -переходов имеет вид (см. табл. 4.3).

Таблица 4.3

	a	b
$q_0 = \Xi(s_0) = \{s_0, s_1\}$	$\{\Xi(s_2), \Xi(s_3)\}$	$\{\Xi(s_2)\}$
$q_1 = \Xi(s_1) = \{s_1\}$	$\{\Xi(s_2), \Xi(s_3)\}$	\emptyset
$q_2 = \Xi(s_2) = \{s_2\}$	\emptyset	$\{\Xi(s_2), \Xi(s_3)\}$
$q_3^+ = \Xi(s_3) = \{s_3, s_2\}$	$\{\Xi(s_0), \Xi(s_3)\}$	$\{\Xi(s_2), \Xi(s_3)\}$

Граф переходов этого недетерминированного автомата представлен на рис. 4.12. По сравнению с автоматом рис. 4.10 здесь добавилось новое начальное состояние, а также по паре переходов из состояний q_0 и q_3 .

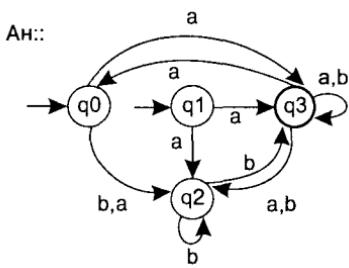


Рис. 4.12. Недетерминированный конечный автомат-распознаватель без ϵ -переходов

Рассмотрим теперь алгоритм построения по заданному недетерминированному автомату $A_n = <S, X, Q_0, \delta_n, F_n>$ без ϵ -переходов эквивалентного ему детерминированного автомата. Заметим, что в автомате A_n множество начальных состояний может содержать несколько состояний (на рис. 4.12 — два начальных состояния: $\{q_0, q_1\}$). В качестве состояний искомого детерминированного автомата естественно выбрать множество состояний исходного недетерминированного автомата (см. рис. 4.10, б), в качестве начального состояния — множество начальных состояний исходного недетерминированного автомата $A_d = <2^S, X, Q_0, \delta_d, F_d>$, функция переходов δ_d которой определяется очевидным образом:

$$(\forall Q \subseteq S)(\forall a \in X): \delta_d(Q, a) = \cup_{s \in Q} \delta_n(s, a).$$

Таблица 4.4

	a	b
$p_0 = \{q_0, q_1\}$	$\{q_2, q_3\}$	$\{q_2\}$
$p_1 = \{q_2\}$	\emptyset	$\{q_2, q_3\}$
$p_2^+ = \{q_2, q_3\}$	$\{q_0, q_3\}$	$\{q_2, q_3\}$
$p_3^+ = \{q_0, q_3\}$	$\{q_0, q_3\}$	$\{q_2, q_3\}$

Множество F_d заключительных состояний искомого автомата состоит из всех таких множеств состояний, которые включают хотя бы одно заключительное состо-

яние исходного автомата: $F_n = \cup_{Q \in \Sigma} Q \cap F_n \neq \emptyset$. Таблица 4.4 представляет таблицу переходов искомого автомата только для состояний, достижимых из начального. Здесь оба заключительных состояния p_2 и p_3 эквивалентны. После минимизации этого автомата получим автомат, график которого представлен на рис. 4.11.

Синтаксические диаграммы. Связь синтаксических диаграмм и автоматных языков

Синтаксические диаграммы — это направленные графы с одним входным ребром и одним выходным ребром и помеченными вершинами. Синтаксическая диаграмма задает язык: цепочку пометок при вершинах на любом пути от входного ребра к выходному ребру будем считать цепочкой языка, задаваемого синтаксической диаграммой.

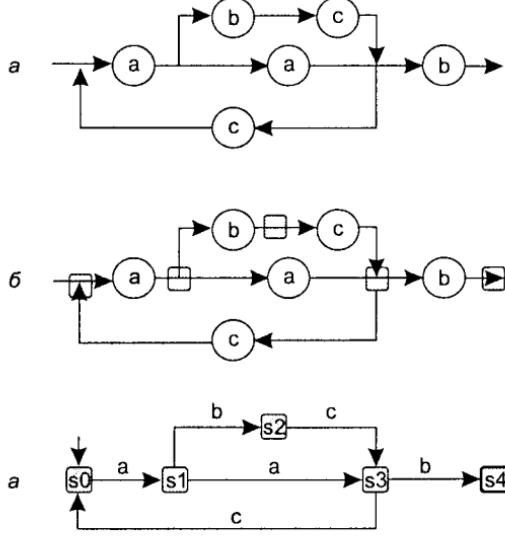


Рис. 4.13. Синтаксическая диаграмма (a) и соответствующий ей конечный автомат (б, в)

Диаграмма, представленная на рис. 4.13, а задает язык, включающий цепочки aab , $aacabc$ и т. д. Поэтому можно считать, что **синтаксическая диаграмма** — это одна из форм порождающей грамматики автоматных языков. Синтаксические диаграммы и конечные автоматы имеют тесную связь: любой автоматный язык задается синтаксической диаграммой и обратно, по любой синтаксической диаграмме можно построить конечный автомат (в общем случае недетерминированный), распознавающий тот же язык, который задает синтаксическая диаграмма (см. рис. 4.13).

Построив по синтаксической диаграмме соответствующий распознающий конечный автомат, можно затем реализовать этот автомат либо аппаратно, либо программно. Таким образом, синтаксические диаграммы могут служить не только для порождения, но и для распознавания автоматных языков.

Трансляторы автоматных языков

Нашей целью является обработка языков программирования. Основной операцией, выполняемой над языками программирования, является их трансляция — со-поставление каждой цепочке языка некоторого выхода. Этот выход может быть эквивалентной программой на языке ассемблера (в этом случае транслятор называется компилятором), или же выход программы представляет непосредственно тот результат, который предполагается получить при выполнении исходной программы (и тогда транслятор называется интерпретатором).

Построение транслятора автоматного языка можно связать с распознаванием входной цепочки: с каждым входным сигналом при переходе распознающего автомата из одного состояния в другое, как и в обычном автомате-преобразователе, можно связать выполнение некоторого действия, которое назовем *семантическим действием*. Определим понятия синтаксиса и семантики.

Определение 4.11. Синтаксисом языка называются правила построения предложений языка. Семантикой языка называются правила интерпретации предложений языка, то есть правила сопоставления им значений из некоторого (произвольного) множества значений.

Рассмотрим, как строить транслятор для одного из самых простых языков — языка L_8 целых констант. Конечный автомат-транслятор этого языка представлен на рис. 4.14, а. Это просто распознающий автомат (см. рис. 4.7), дополненный семантическими действиями, выполняемыми при анализе очередного входного символа. Кроме того, сюда добавлено дополнительное состояние γ , в которое автомат переходит, как только прекратится входная строка (или же, чаще всего, очередным символом цепочки будет не цифра). Это показывает служебный символ EOS.

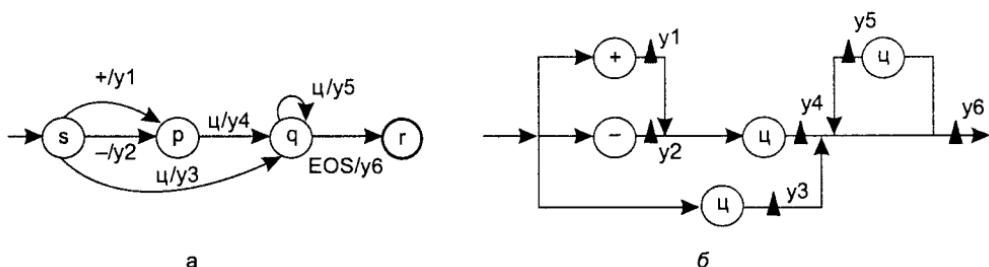


Рис. 4.14 Конечный автомат, транслирующий язык L_8 , (а) и синтаксическая диаграмма с семантическими операциями для трансляции L_8 (б)

Предложениями языка L₈ являются цепочки символов — знаков и цифр. Результатом трансляции этих предложений должно быть число во внутреннем машинном представлении. Указанные на рис. 4.14 семантические функции имеют вид:

y1: z:= 1; // положительный знак константы запоминаем как значение переменной z;

y2: z:= -1; // аналогично с отрицательным знаком константы;

y4: r:= φ(ц); // переменная r содержит число, изображаемое цифрой ц во входной цепочке;

y3: z:= 1; r:= φ(ц); // константа без знака есть положительная константа;

y5: g:= r*10 + φ(ц); // очередная цифра числа;

y6: A:= z* r. // когда число закончилось, результат во внутреннем представлении присваивается некоторой выходной переменной.

Преобразование φ(ц) из «изображения» в «значение» может быть выполнено, например, с помощью операции *ord* языка Pascal, выдающего порядковый номер (код) символа. Если коды всех цифр идут подряд, то φ(ц) = *ord*(ц) – *ord*(«0»).

Удобнее представлять транслятор с помощью синтаксической диаграммы, нагруженной семантическими функциями. Рисунок 4.14, б показывает такой транслятор для целых констант. Синтаксической диаграмме легко сопоставить детерминированный конечный автомат-преобразователь, реализация которого как аппаратным, так и программным способом проста.

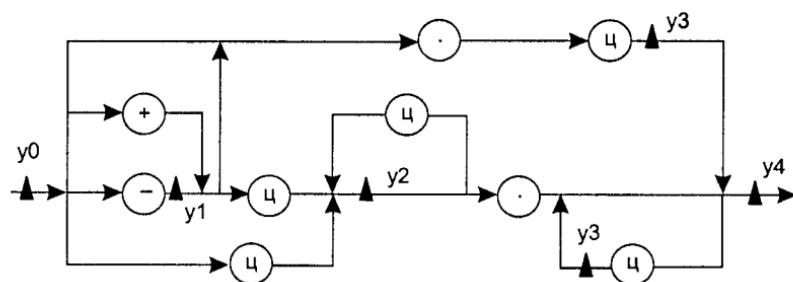


Рис. 4.15. Синтаксическая диаграмма с семантическими операциями для трансляции языка вещественных констант

На рис. 4.15 приведен транслятор языка вещественных констант. В константах могут быть опущены знак и целая либо дробная часть, но не обе сразу. Семантические операции этого транслятора определяются так:

y0: z:= 1; n:= 0; k:= 0,1; d:= 0;

y1: z:= -1;

y2: n:= n*10 + φ(ц);

y3: d:= d + φ(ц)/k; k:= k/10;

y4: A:= z*(n + d).

Транслятор языка ЛИНУР

В качестве примера применения рассмотренной выше техники построим транслятор языка ЛИНУР – языка линейных алгебраических уравнений. Пусть пользователь на экране дисплея записывает следующую последовательность символов.

«Решить систему 4 уравнений:

$$43,7 X[1] - 18,91 X[3] + 6,7 X[4] = 2,64;$$

$$2,0 X[2] - 16,2 X[4] = -21,4;$$

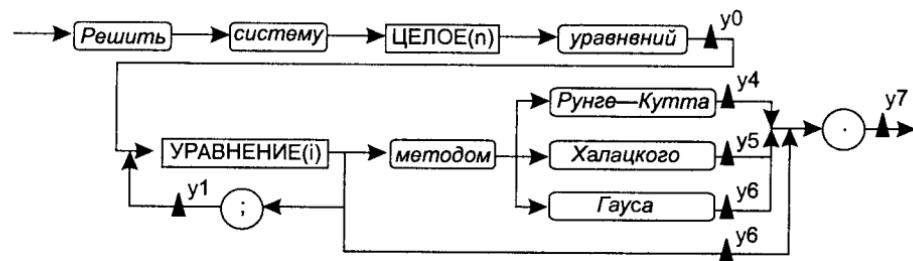
$$4,7 X[1] - 8,1 X[4] = 0,63;$$

$$6,9 X[2] - 191,5 X[3] = -15,6$$

методом Гаусса.»

В результате работы транслятора, принимающего на вход эту цепочку, пользователь ожидает получить решение (вектор неизвестных X) или информацию о причине невозможности получения решения. Построим синтаксические диаграммы языка, выделив отдельные конструкции – подъязыки СИСТЕМА – наиболее общую конструкцию, УРАВНЕНИЕ, ЧИСЛО и ЦЕЛОЕ.

СИСТЕМА::



УРАВНЕНИЕ(i)::



ЧИСЛО(a)::

...

ЦЕЛОЕ(·)::

...

Синтаксические диаграммы для конструкций ЧИСЛО и ЦЕЛОЕ определены выше. Семантические операции транслятора определяются так:

y0: Зарезервировать матрицу A[nxn] и векторы B[n], X[n]; A[] := 0.0; B[] := 0.0;
 i:= 1;
 y1: i:= i + 1;
 y2: A[i, j]:= A[i, j] + a;
 y3: B[i]:= B[i] + b;
 y4: Runge(n, A, B, X, E);
 y5: Haletskij(n, A, B, X, E);
 y6: Gauss(n, A, B, X, E);
 y7: if E = 0 then print(X, n)
else print («Ошибка типа», E).

Здесь Runge(n, A, B, X, E), Haletskij(n, A, B, X, E) и Gauss(n, A, B, X, E) — подпрограммы, дающие решение линейной системы из n уравнений, заданной матрицей A и вектором правых частей B. Вектор результатов помещается в массив X. Параметр E — это индикатор ошибки. Если E = 0, то ошибок при решении нет, если E = 1, то, например, матрица коэффициентов несовместна и т. д.

Синтаксическая диаграмма, описывающая язык ЛИНУР, разбита на отдельные синтаксические диаграммы для отдельных конструкций языка, которые сами являются языками, и вместо соответствующей части диаграммы вставляется блок (прямоугольник) с именем этого подъязыка. Каждой такой конструкции соответствует распознавающая процедура; эти процедуры вызываются по мере необходимости, как только блок с соответствующим именем встретится на дуге синтаксической диаграммы. При построении транслятора эти процедуры могут обмениваться параметрами. Так, в описании процедуры ЧИСЛО параметр *a* является формальным параметром, в который помещается оттранслированная величина, соответствующая численному значению числа. Транслятор запускается вызовом самой общей процедуры: СИСТЕМА. Очевидно, что построенный нами транслятор является интерпретатором.

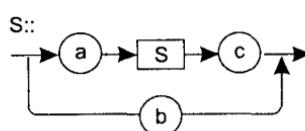


Рис. 4.16. Синтаксическая диаграмма языка $L_4 = \{a^nbc^n \mid n \geq 0\}$

Заметим, что синтаксические диаграммы с вершинами, соответствующими отдельным подъязыкам, могут задавать более сложные языки — так называемые *контекстно-свободные языки*. Например, на рис. 4.16 представлена одна синтаксическая диаграмма языка $L_4 = \{a^nbc^n \mid n \geq 0\}$, который не является автоматным. Узнать автоматный язык в представлении синтаксическими диаграммами просто: в совокупности зависимостей диаграмм не должно быть рекурсии.

Транслятор языка EXPR

В качестве другого примера построим транслятор языка EXPR, в котором все операции — это присваивания переменным значений арифметических выражений и операции вывода. Язык арифметических выражений во всей его сложности — это не автоматный язык, поскольку, как мы знаем, скобочные выражения автоматными грамматиками не могут быть заданы. Мы упростим задачу в том смысле, что не разрешим писать скобки в арифметических выражениях. Для простоты также будем использовать только целые числа. Пусть пользователь на экране дисплея записывает следующую последовательность символов:

```
x:= 15;
y:= 28 - x;
a1:= 387 - x/27;
b:= a1+48*x - y;
вывод(a1 - y).
```

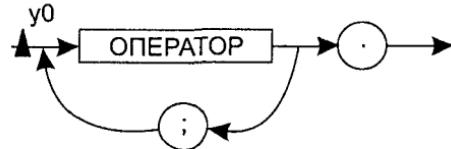
В результате работы транслятора, принимающего на вход эту цепочку, пользователь получает решение (значение выражения $a1 - y$). Построим синтаксические диаграммы языка, выделив отдельные конструкции — подъязыки ПРОГРАММА — наиболее общую конструкцию, ОПЕРАТОР, ВЫРАЖЕНИЕ, ТЕРМ и ОПЕРАНД.

ПРОГРАММА здесь — это последовательность операторов, разделенных точкой с запятой; существует два типа операторов — присваивания и вывода. Различить их можно по первому имени: если это имя «вывод», то распознаваемый оператор — оператор вывода. Точка после оператора говорит о конце программы. Для учета приоритета операторов введена конструкция ТЕРМ: *оператор* — это последовательность *термов*, соединенных знаками операций низкого приоритета, в то время как сама конструкция ТЕРМ — это последовательность *операндов*, разделенных знаками операций высокого приоритета. Очевидно, для выполнения программы необходимо хранить значения идентификаторов; для этого семантические программы должны сформировать и использовать таблицу имен. Если идентификатор встречается в выражении — правой части оператора присваивания, то этой переменной где-то раньше в программе должно было быть присвоено значение, которое должно уже храниться в таблице имен. Это может быть проверено при трансляции, и в случае использования неприсвоенной переменной в правой части оператора присваивания транслятор может указать на ошибку.

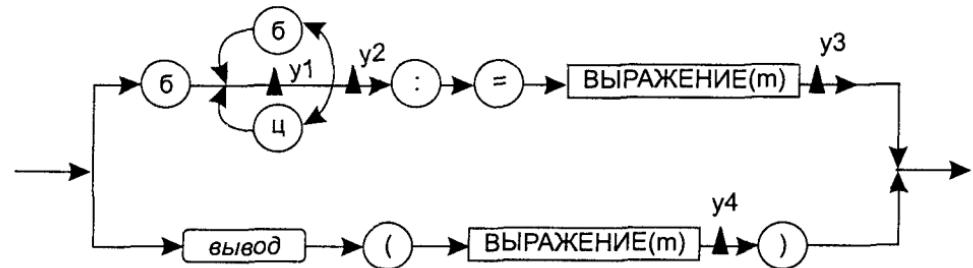
Заметьте, что построенный транслятор без какой-либо переделки может обрабатывать **любые** программы на этом языке, содержащие тысячи операторов и включающие арифметические выражения любой сложности (но без скобок).

Определение семантических операций транслятора остается в качестве упражнения.

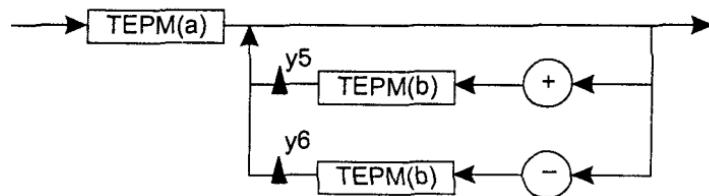
ПРОГРАММА:::



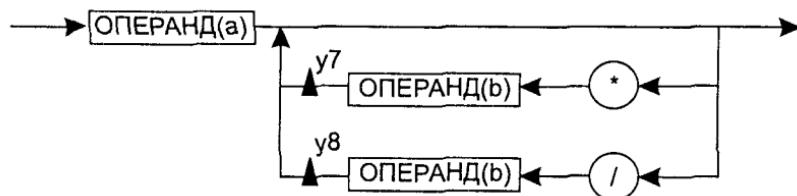
ОПЕРАТОР:::



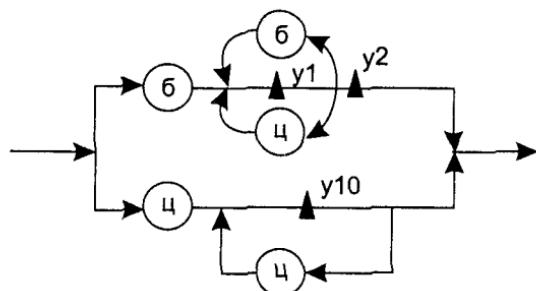
ВЫРАЖЕНИЕ(a):::



TERM(a):::



ОПЕРАНД(a):::



Регулярные множества и регулярные выражения

Регулярные множества

В данном разделе мы рассмотрим класс множеств цепочек над конечным словарем, которые очень легко описать формулами некоторого вида. Эти множества называются регулярными.

Определение 4.12. Пусть V_1 и V_2 — множества цепочек. Определим три операции на этих множествах.

1. Объединение: $V_1 \cup V_2 = \{a | a \in V_1 \text{ или } a \in V_2\}$.

2. Конкатенация (произведение, склеивание): $V_1 \bullet V_2 = \{a\beta | a \in V_1, \beta \in V_2\}$. Знак операции конкатенации обычно опускается.

Пример: $V_1 = \{abc, ba\}, V_2 = \{b, cb\}$. $V_1V_2 = \{abcb, abcba, bab, bacb\}$.

Обозначим V^n произведение n множеств V : (здесь ϵ -пустая цепочка), $V^{n+1} = V^nV$.

Пример: $V_1 = \{abc, ba\}, V_1^2 = \{abca, abcb, baba, baabc\}$.

3. Итерация: $V^* = V^0 \cup V^1 \cup V^2 \cup \dots = \bigcup_{n=0}^{\infty} V^n$.

Пример: $V = \{a, bc\}, V^* = \{\epsilon, a, bc, aa, abc, bcbc, bca, aaa, aabc, \dots\}$.

Определение 4.13. Класс регулярных множеств над конечным словарем V определяется так:

1. \emptyset — регулярное множество;

2. $\{\epsilon\}$ — регулярное множество;

3. $(\forall a \in V)\{a\}$ — регулярное множество;

4. Если S и T — регулярные множества, то регуляры:

объединение $S \cup T$;

конкатенация ST ;

итерация S^* и T^* .

5. Если множество не может быть построено конечным числом применения правил 1–4, то оно нерегулярио.

Примеры регулярных множеств: $\{ab, ba\}^* \setminus \{aa\}$; $\{b\}(\{c\} \cup \{d, ab\})^*$. Примеры нерегулярных множеств: $\{a^n b^n | n > 0\}$; $\{\alpha | \text{в цепочке } \alpha \text{ количества вхождений символов } a \text{ и } b \text{ совпадают}\}$.

Регулярные выражения

Регулярные множества хороши тем, что их можно очень просто описать формулами, которые мы назовем регулярными выражениями.

Определение 4.14. Класс регулярных выражений над конечным словарем V определяется так:

1. \emptyset и ϵ – регулярные выражения;
2. $(\forall a \in V)a$ – регулярное выражение;
3. Если R_1 и R_2 – регулярные выражения, то регулярными выражениями являются:
 - их сумма $R_1 + R_2$;
 - их произведение $R_1 R_2$;
 - их итерация R_1^* и R_2^* .
4. Если выражение не построено конечным числом применения правил 1–3, то оно не является регулярным.

Знак произведения можно опускать. Для уменьшения числа скобок, как и в любой алгебре, используются приоритеты операций: итерация самая приоритетная; менее приоритетно произведение; самый низкий приоритет у сложения.

Примеры регулярных выражений: $ab + ba^*$; $(aa)^*b + (c + dab)^*$.

Очевидно, что регулярные множества и регулярные выражения весьма близки. Но они представляют собой разные сущности: регулярное множество – это множество цепочек (в общем случае бесконечное), а *регулярное выражение* – это формула, схематично показывающая, как было построено соответствующее ей регулярное множество с помощью перечисленных выше операций (и эта формула конечна).

Пусть R^\wedge – регулярное множество, соответствующее регулярному выражению R . Тогда:

1. Если $R = \emptyset$, то $R^\wedge = \emptyset$.
2. Если $R = \epsilon$, то $R^\wedge = \{\epsilon\}$.
3. Если $R = a$, то $R^\wedge = \{a\}$.
4. Если $R = R_1 + R_2$, то $R^\wedge = R_1^\wedge \cup R_2^\wedge$.
5. Если $R = R_1 \bullet R_2$, то $R^\wedge = R_1^\wedge \bullet R_2^\wedge$.
6. Если $R = R_1^*$, то $R^\wedge = R_1^{\wedge*}$.

Таким образом, регулярное выражение – это конечная формула, задающая бесконечное множество цепочек, то есть язык.

Рассмотрим примеры регулярных выражений и соответствующих им языков.

Регулярное выражение	Соответствующий язык
ba^*	Все цепочки, начинающиеся с b , за которым следует произвольное число символов a
$a^*ba^*ba^*$	Все цепочки из a и b , содержащие ровно два вхождения b

Продолжение

Регулярное выражение	Соответствующий язык
$(a+bb)^*$	Все цепочки из a и b , в которые символы b входят только парами
$(a+b)^*(aa+bb)(a+b)^*$	Все цепочки из a и b , содержащие хотя бы одну пару рядом стоящих a или b
$(0+1)^*11001(0+1)^*$	Все цепочки из 0 и 1, содержащие подцепочку 11001
$a(a+b)^*b$	Все цепочки из a и b , начинающиеся символом a и заканчивающиеся символом b

Очевидно, что множество цепочек регулярно тогда и только тогда, когда оно может быть представлено регулярным выражением. Однако одно и то же множество цепочек может быть представлено различными регулярными выражениями, например, множество цепочек, состоящее из символов a и содержащее не менее двух a может быть представлено выражениями: aa^*a ; a^*aaa^* ; aaa^* ; $a^*aa^*aa^*$ и т. д.

Определение 4.15. Два регулярных выражения R_1 и R_2 называются эквивалентными (обозначается $R_1 = R_2$) тогда и только тогда, когда $R_1^\wedge = R_2^\wedge$.

Таким образом, $aa^*a = a^*aaa^* = aaa^* = a^*aa^*aa^*$. Возникает вопрос, как определить эквивалентность двух регулярных выражений.

Теорема 4.7. Для любых регулярных выражений R , S и T справедливо:

1. $R+S = S+R$; $R+R=R$; $(R+S)+T = R+(S+T)$; $\emptyset+R = R$;
2. $R\epsilon = \epsilon R = R$; $(RS)T = R(ST)$; $\emptyset R = R\emptyset = \emptyset$; но в общем случае $RS \neq SR$;
3. $R(S+T) = RS+RT$; $(S+T)R = SR+TR$;
4. $R^* = \epsilon + R + R^2 + R^3 + \dots + R^k R^*$; $RR^* = R^*R$; $R(SR)^* = (RS)^*R$;
5. $R^+ = R + R^2 + R^3 + \dots$; $R^*R^+ = RR^*$; $R^++\epsilon = R^*$.

Эти соотношения можно доказать, проверяя равенство соответствующих множеств цепочек. Их можно использовать для упрощения регулярных выражений. Например: $b(b+aa^*b) = b(\epsilon b + aa^*b) = b(\epsilon + aa^*)b = ba^*b$. Отсюда $b(b+aa^*b) = ba^*b$, что неочевидно.

Теорема Клини

Регулярные выражения — это конечные формулы, задающие регулярные языки. Но подобным же свойством обладают и конечные автоматы — они тоже задают языки. Возникает вопрос: как соотносятся между собой классы языков, задаваемые конечными автоматами и регулярными выражениями? Обозначим Λ_A множество автоматных языков, Λ_R — множество регулярных языков. Стефан Клини, американский математик, доказал следующую теорему.

Теорема 4.8. (Теорема Клини.) Классы регулярных множеств и автоматных языков совпадают, то есть $\Lambda_A = \Lambda_R$.

Иными словами, каждый автоматный язык может быть задан формулой (регулярным выражением) и каждое регулярное множество может быть распознано конечным автоматом. Мы докажем эту теорему конструктивно, в два шага. На первом шаге докажем, что любой автоматный язык является регулярным множеством (или, что то же, для любого конечного автомата можно построить регулярное выражение, задающее распознаваемый этим автоматом язык). На втором шаге докажем, что любое регулярное множество является автоматным языком (или, что то же, по любому регулярному выражению можно построить конечный автомат, допускающий в точности цепочки соответствующего регулярного множества).

Введем в рассмотрение модель графа переходов как обобщение модели конечного автомата. В графе переходов одна начальная и произвольное количество заключительных вершин, а направленные ребра помечены в отличие от конечного автомата не символами, а регулярными выражениями. Граф переходов допускает цепочку α , если α принадлежит множеству цепочек, описываемому произведением регулярных выражений $R_1 R_2 \dots R_n$, которые помечают путь из начальной вершины в одну из заключительных вершин. Множество цепочек, допускаемых графиком переходов, образует допускаемый им язык.

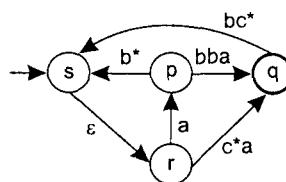


Рис. 4.17. Граф переходов

На рис. 4.17 изображен график переходов, который допускает, например, цепочку $abbca$, поскольку путь $s \rightarrow r \rightarrow p \rightarrow s \rightarrow r \rightarrow q$, который ведет в заключительное состояние q , помечен цепочкой регулярных выражений $\epsilon \cdot a \cdot b^* \cdot \epsilon \cdot c \cdot a$. Конечный автомат является частным случаем графа переходов и поэтому все языки, которые допускаются автоматами, допускаются и графиками переходов.

Теорема 4.9. Каждый автоматный язык является регулярным множеством, $\Lambda_A \subseteq \Lambda_R$.

Доказательство. Граф переходов с одной начальной и одной заключительной вершинами, у которого единственное ребро из начальную в заключительную вершину помечено регулярным выражением R , допускает язык R^\wedge (рис. 4.18).

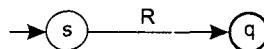


Рис. 4.18. Граф переходов, допускающий регулярный язык R^\wedge

Докажем теперь, что каждый автоматный язык является регулярным множеством приведением любого графа переходов без изменения допускаемого им языка к эквивалентному виду (рис. 4.18).

Любой конечный автомат и любой граф переходов всегда можно представить в нормализованной форме, в которой только одна начальная вершина только с исходящими ребрами и только одна заключительная вершина только с входящими ребрами (рис. 4.19).

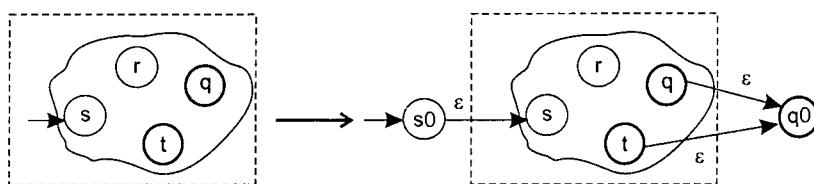


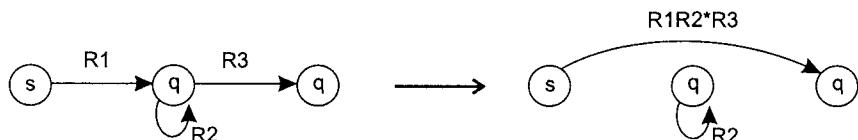
Рис. 4.19. Граф переходов с одной начальной и одной заключительной вершинами

С графом переходов, представленным в нормализованной форме, могут быть выполнены две операции редукции — редукция ребра и редукция вершины — с сохранением допускаемого этим графом переходов языка:

а) редукция ребра:



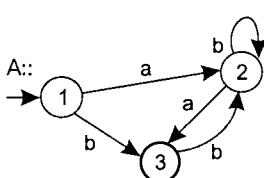
б) редукция вершины (замена выполняется для каждого пути, проходящего через вершину p , с последующим ее выбрасыванием как недостижимого состояния):



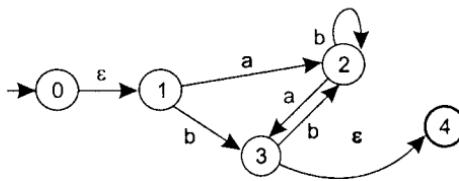
Очевидно, что каждая операция редукции не меняет языка, распознаваемого графом переходов, но уменьшает либо число ребер, либо число вершин, и в конце концов редукции приведут граф переходов к виду, приведенному на рис. 4.18. Терема доказана: каждый автоматный язык является регулярным множеством.

Пример 4.4

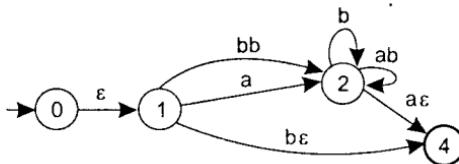
Пусть задан конечный автомат А:



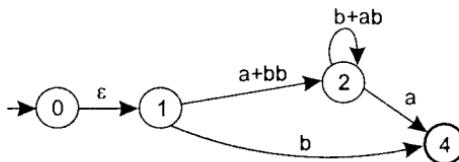
Строим эквивалентный граф переходов в нормализованном виде.



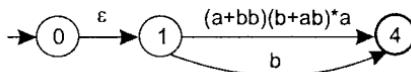
Редукция вершины 3:



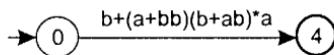
Редукция дуг и применение правила $R\varepsilon = R$:



Редукция вершины 2:



Редукция дуги и вершины 1:



Таким образом язык, распознаваемый автоматом A , задается регулярным выражением: $R_A = b + (a+bb)(b+ab)^*a$.

Докажем теорему Клини в другую сторону.

Теорема 4.10. Каждое регулярное множество является автоматным языком: $\Lambda_R \subseteq \Lambda_A$.

Доказательство. Покажем, что для каждого регулярного выражения R может быть построен конечный автомат A_R (возможно, недетерминированный), распознавающий язык, задаваемый R . Определение таких автоматов дадим рекурсивно.

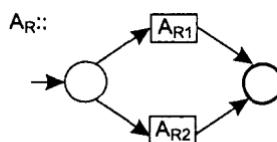
- Если $R = \emptyset$, то $A_R = \rightarrow \bigcirc \bigcirc$ (\bigcirc (два несвязанных состояния)).

2. Если $R = \epsilon$, то $A_R = \xrightarrow{\epsilon} \text{старт} \xrightarrow{\epsilon} \text{конец}$.

3. Если $R = a$, то $A_R = \xrightarrow{a} \text{старт} \xrightarrow{a} \text{конец}$.

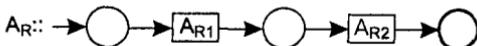
Пусть теперь автомат $A_R \xrightarrow{\epsilon} \text{старт} \xrightarrow{\epsilon} A_R \xrightarrow{\epsilon} \text{конец}$ с одним начальным и одним заключительным состоянием допускает язык, задаваемый R . Тогда

4. Если $R = R_1 + R_2$, то



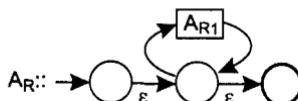
(начальные и заключительные состояния автоматов A_{R1} и A_{R2} совмещаются).

5. Если $R = R_1 R_2$, то:



(начальное состояние A_{R1} и заключительное состояние A_{R2} совмещаются).

6. Если $R = R_1^*$, то:

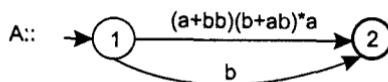


(начальное и заключительное состояния A_{R1} совмещаются).

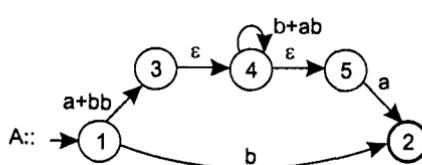
Пример 4.4 (продолжение)

Пусть задано регулярное выражение $R_A = b + (a + bb)(b + ab)^*a$. Построим автомат A , допускающий язык R_A^* .

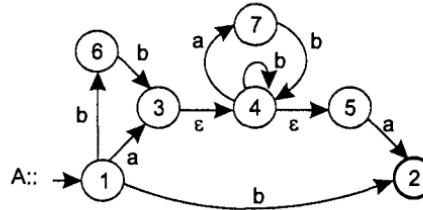
По правилу 4,



По правилам 4, 5 и 6,



По правилам 5 и 6,



Построенный автомат недетерминированный, с мгновенными переходами. После построения эквивалентного детерминированного автомата и проведения минимизации получим автомат, изоморфный исходному автомата в примере 4.4.

Пример применения регулярных выражений: построение лексических анализаторов алгоритмических языков

Лексемой называется минимальная структурная единица языка, имеющая смысл. Символ не является лексемой. Например, символ «`:`» не имеет смысла в языке Pascal, он может встречаться только совместно с символом «`=`», и эта пара имеет смысл «операция присваивания». Отдельная цифра не имеет своего смысла, это либо часть константы, либо часть идентификатора. Операция присваивания, константа, идентификатор — это все примеры лексем.

Структура любого транслятора состоит из двух частей. В первом, входном блоке, из входного потока выделяются лексемы и передаются далее для последующего синтаксического анализа. Этот блок называется лексическим анализатором. Лексический анализатор выполняет и другую работу: он выбрасывает из программы пробелы и комментарии, переводит числовые константы в машинное представление и т. д. Теоретически лексический анализатор интересен тем, что все распознаваемые им конструкции описываются автоматной грамматикой и, следовательно, регулярными выражениями.

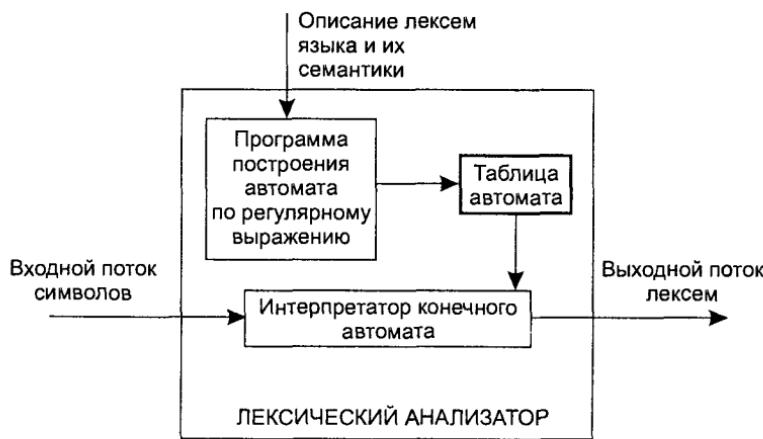


Рис. 4.20. Структура настраиваемого лексического анализатора

Наиболее интересно построение **настраиваемых** лексических анализаторов, которые применяются в так называемых «компиляторах компиляторов», то есть компиляторах, настраиваемых на входной язык вводимым описанием грамматики (рис. 4.20). Обычно описание лексем задается с помощью регулярных выражений.

Пример применения регулярных выражений: поиск в текстах по образцам в UNIX

Поиск по образцам имеют в качестве сервиса практически все современные системы работы с текстами и файлами – Norton Commander, среда программирования Borland, редакторы текстов в OS UNIX. Регулярные выражения являются основным средством задания короткой формулой множества образцов для поиска, то есть того множества цепочек, которые необходимо найти в тексте. В UNIX такой поиск осуществляется командой GREP (Global Regular Expression Print). Синтаксис регулярных выражений, которыми задаются образцы в различных системах, несколько отличается от алгебраической записи, применявшейся в этой главе, и друг от друга, но суть сохраняется.

Рассмотрим программу GREP поиска текстов по образцам операционной системы UNIX. Программа ищет в символьном файле подстроки, характеризуемые заданным образом – регулярным выражением. Все возможные искомые подстроки рассматриваются GREP как регулярный язык, задаваемый этим регулярным выражением. Для задания множества символов здесь применяются квадратные скобки, используются также средства для указания дополнения множества символов и множества, состоящего из нескольких подряд идущих символов, например,

[abd] – множество, состоящее из символов a, b, d;

[^abq] – множество, состоящее из всех символов, кроме символов a, b, q;

[a-z0-7] – множество, состоящее из всех букв (от a до z) и восьмеричных цифр (от 0 до 7);

\s – непосредственное значение символа s (а не его специальное управляющее значение);

. – любой символ кроме EOF;

* – повторение 0 или более раз;

+ – повторение 1 или более раз;

\$ – конец строки;

| – символ альтернативы (или)

^ – начало строки.

Примеры выражений в синтаксисе UNIX (примеры взяты из руководства Borland 4.5):

Регулярное выражение	Объяснение	Тексты, в которых образцы будут найдены	Тексты, в которых образцы не будут найдены
[^a-z]main\ *()	Подстрока, включающая слово <i>main</i> , перед которым стоит символ не буква; за ним может следовать произвольное число пробелов (возможно, их нет), за которыми идет открывающая скобка	8main(i, j:integer) if (main ()) halt; if (main(gr,a)) halt;	mymain()
[,:?]\$	Подстрока, состоящая из символа запятая, двоеточие или вопрос, стоящая в конце строки (знак \$ означает конец строки)	the following: Where are you?	main()
[a-c]:\\data\\.fil	Поскольку слэш (\) и точка(.) имеют специальное значение, то чтобы указать, что именно они должны встретиться, используется символ «\»	aa: \\data.filips &b: \\data.fil\ 88rc: \\data.filon	f: \\data.fil a: data.fil a: \\data\\.fil
[0-9]+ [0-9]+\. [0-9]* . [0-9]+	Десятичное число без знака, с дробной частью или без нее	A+870.6 (.381MK = H) R:= 345612. + a	A+B = C A./OB/ Ht...?

При задании лексем в системах построения компиляторов обычно также используется этот синтаксис регулярных выражений. Например, идентификатор, то есть любая последовательность букв и цифр, начинающаяся с буквы, в этом синтаксисе задается так: [a-z][a-z0-9]*. Если прописные и строчные буквы различаются, то идентификатор определится так: [a-zA-Z][a-zA-Z0-9]*.

Контрольные задания

- Построить конечный автомат с входным алфавитом $V = \{a, b\}$, распознающий:
 - пустой язык $L = \emptyset$;
 - язык, содержащий только пустую цепочку;
 - язык $V^* - L$, где L – некоторый заданный автоматный язык;
 - язык L^* , где L – некоторый заданный автоматный язык;
 - язык $L = \{ \alpha || \alpha|_a \text{ четно, } a|\alpha|_b \text{ нечетно} \}$;
 - все цепочки, заканчивающиеся кодом $aabb$;
 - все цепочки, включающие строку $aabb$;
 - все цепочки, в которых за каждым a непосредственно следует b .

2. Построить конечный автомат с входным алфавитом $V = \{a, b, c\}$, распознающий:
 - а) все цепочки, в которых за каждым а когда-нибудь в будущем следует b;
 - б) все цепочки, в которых две последние буквы не совпадают;
 - в) все цепочки, начинающиеся и заканчивающиеся различными символами;
 - г) все цепочки, включающие по крайней мере один символ a и один символ b.
3. Константы с плавающей точкой в языке С имеют следующий формат представления:

[<цифры>].[<цифры>]<э>{–,+}[<цифры>]

[<цифры>] — одна или более десятичных цифр (от 0 до 9);

<э> — признак экспоненты, задаваемый символом Е или е. Либо целая, либо дробная часть константы может быть опущена, но не обе сразу. Либо десятичная точка с дробной частью, либо экспонента могут быть опущены, но не обе сразу. Знак «+» или «–» может присутствовать или отсутствовать. Построить детерминированный конечный автомат, распознающий константы с плавающей точкой в языке С.

4. Построить схему кодового замка, открывающегося только при наборе кода 10010 или цепочки, заканчивающейся этим кодом, двумя способами. Сначала построить и реализовать детерминированный автомат, допускающий все эти цепочки а затем — соответствующий недетерминированный автомат наиболее простого вида и свести его к минимальному детерминированному.
5. Разработать процедуру преобразования произвольного конечного автомата, допускающего L , в конечный автомат, допускающий $L - \{\epsilon\}$.
6. Разработать процедуру преобразования произвольного конечного автомата, допускающего L , в конечный автомат, допускающий $L \cup \{\epsilon\}$.
7. Построить конечный автомат, допускающий все числа в десятичной записи, которые делятся на 7.
8. Построить конечный автомат, допускающий все числа в десятичной записи, сумма цифр которых делится на 7.
9. Минимизировать конечные автоматы рис. 4.3, а, б.
10. Построить распознаватель операторов FORMAT языка FORTRAN.
11. Построить схему алгоритма трансляции вещественных констант по синтаксической диаграмме (см. рис. 4.15).
12. Целые константы без знака в языке С определяются так:
 - а) *десятичные*: <последовательность цифр, начинающаяся не с нуля>
 - б) *восьмеричные*: 0 <последовательность цифр>
 - в) *шестнадцатеричные*: 0 {x, X} <последовательность цифр и букв a–f или A–F>

Построить синтаксическую диаграмму распознавателя и транслятор этого языка.

13. Построить блок-схемы программ, реализующих транслятор с языка ЛИНУР.
14. В транслятор с языка ЛИНУР ввести печать информации о синтаксических ошибках. В каком блоке он определит ошибку в уравнении
$$5.2X[2] + 1.6 - X[3] = 0?$$
15. В транслятор с языка ЛИНУР ввести возможность управления необходимой точностью решения.
16. В транслятор с языка ЛИНУР ввести проверку соответствия
 - а) числа уравнений и размерности системы;
 - б) индексов при X и размерности системы.
17. Построить транслятор операторов печати языка Pascal.
18. Решить задачу обработки потока телеграмм — построить транслятор, принимающий на вход поток телеграмм и выдающий результаты обработки. Поток состоит из последовательности телеграмм, разделенных специальным разделителем #, весь поток заканчивается символом eof. Каждая телеграмма состоит из последовательности слов, разделенных пробелами, слово — это произвольная последовательность букв. Требуемый выход состоит из печати для каждой телеграммы ее номера, числа слов, числа «длинных» слов (за них берется повышенная плата) и стоимости телеграммы исходя из заданной стоимости одного обычного и одного длинного слова. Аналогичную суммарную информацию требуется выдать по всему потоку телеграмм.
19. Построить автоматы, распознающие языки, задаваемые регулярными выражениями:
$$a^*b^*; a^*a^*; a^*+b^*; (a+b)^*; (a^*b^*)^*; (a^*+b^*)^*, a^*b+b^*a.$$
20. Определить, эквивалентны ли два регулярных выражения: $a(ba)^*b^*$ и $(ab)^*a(b^*)^*$ всеми возможными способами.
21. Построить регулярное выражение, задающее множество всех таких слов над словарем $\{a, b, c\}$, в которых за символом b а) обязательно стоит символ c б) не может стоять символ c . Построить конечные автоматы, распознающие соответствующие языки.
22. Опишите регулярными выражениями следующие лексемы языка Pascal: комментарии, служебные слова, идентификаторы, константы.
23. Является ли алгебра регулярных выражений булевой алгеброй?
24. Построить регулярные выражения, задающие языки, определенные в примерах 4.1 и 4.2.
25. Пусть Π_A — множество всех разбиений конечного множества A . Пусть сигнатура Ω операций на Π_A совпадает с сигнатурой операций булевой алгебры. Определите бинарные операции объединения и пересечения, унарную операцию получения обратного разбиения и две пульварные операции максимального разбиения I (состоящего только из одного блока) и минимального разбиения O

(состоящего только из одноэлементных блоков). Является ли алгебра разбиений (Π_A, Ω) булевой алгеброй? Какие свойства булевой алгебры не выполняются в алгебре разбиений?

26. В редакторе MS Word существует опция печати страниц редактируемого документа с указанием номеров страниц или интервалов таких номеров. Страницы и интервалы разделяются запятыми, например: 1, 3, 5–12. В интервалах страницы могут не только возрастать, но и убывать. Построить программу, выдающую в качестве выхода последовательность номеров страниц, выводимых на печать, и информирующую об ошибках, когда синтаксис нарушен.
27. Доказать, что регулярные выражения \emptyset^* и ϵ эквивалентны.
28. Доказать, что регулярные выражения $(ba+a^*ab)^*a^*$ и $(a+ab+ba)^*$ эквивалентны.
29. Доказать, что регулярные выражения $(ab^*c)^*$ и $\emptyset^*+a(b+ca)^*c$ эквивалентны.

ГЛАВА 5 Машины Тьюринга

Конечные автоматы — как преобразователи входных последовательностей сигналов, так и распознаватели множеств цепочек — выполняют преобразование входной информации в соответствии с некоторыми правилами, то есть реализуют некоторый алгоритм переработки информации. Иными словами, автоматы — это устройства, механически выполняющие алгоритмы. Можно строить различные модели устройств, автоматически выполняющих алгоритмы, и исследовать классы алгоритмов, которые могут быть реализованы на этих моделях.

Алгоритм является фундаментальной концепцией информатики, и вопросы построения автоматических устройств, реализующих алгоритмы, — это один из главных вопросов вычислительной науки. Какой класс алгоритмов может быть представлен конечными автоматами, независимо от числа состояний автоматов, их функций переходов и выходов? Поставим вопрос по-другому. Можно ли для любого алгоритма переработки информации найти конечный автомат, его выполняющий? Ведь число различных конечных автоматов бесконечно.

Очевидно, что ответом здесь будет «нет». Конечные автоматы могут решать только узкий класс алгоритмических проблем. Конечный автомат как автоматическое устройство, перерабатывающее информацию, ограничен в своих возможностях. Например, мы уже знаем, что никакой КА не может решить проблему умножения двух чисел, с помощью КА невозможно распознать язык $\{a^n b c^n \mid n > 0\}$. Иными словами, не все алгоритмы обработки информации могут быть реализованы конечными автоматами. Класс алгоритмов, которые могут быть реализованы конечными автоматами, весьма ограничен.

В данной главе мы рассмотрим более мощные автоматические устройства по сравнению с конечными автоматами — машины Тьюринга. Как оказывается, с помощью машин Тьюринга можно реализовать любой алгоритм. Понимание этого утверждения требует рассмотрения понятия алгоритма и подходов к формализации алгоритма. Именно это и составляет предмет изучения данной главы.

В результате изучения материала главы читатель должен:

- получить общее представление об алгоритмах и методах их формального представления;
- научиться представлять простейшие алгоритмы в виде программы для машины Тьюринга;
- понять смысл и значение тезиса Черча–Тьюринга, получить представление об алгоритмически неразрешимых проблемах.

Формальные модели алгоритмов

Понятие алгоритма

Почти во всех сферах жизни человек сталкивается с рецептами решения задач, предписаниями, инструкциями, правилами и т. п. Они однозначно определяют порядок выполнения действий для получения желаемого результата. Предписание, удовлетворяющее некоторым дополнительным требованиям, называется алгоритмом.

Определение 5.1. Алгоритм – это определенное на некотором языке конечное предписание (способ, рецепт), задающее дискретную последовательность исполнимых элементарных операций для решения проблемы. Процесс выполнения предписания состоит из отдельных шагов, на каждом из которых выполняется одна очередная операция.

Это определение, понятное в интуитивном смысле, не является, однако, формальным. Действительно, что означает «элементарная операция»? Или «предписание»? С какими объектами работает алгоритм: числами, матрицами, словами?... Все это требует уточнения, если мы хотим говорить об алгоритмах строго. Алгоритмы в интуитивном смысле не являются математическими объектами, к ним не применимы формальные методы исследования и доказательства. Поэтому в XX веке были предприняты усилия в попытках формализации понятия алгоритма. Формализация понятия алгоритма необходима по разным причинам. Например, сравнение двух алгоритмов по эффективности, проверка их эквивалентности и т. д. возможны только на основе их формального представления.

Формализация понятия алгоритма

Впервые необходимость формального понятия алгоритма возникла в связи с проблемой **алгоритмической неразрешимости** некоторых задач. Долгое время математики верили в возможность того, что все строго поставленные математические задачи могут быть алгоритмически решены, нужно только найти алгоритм их решения. Вера в универсальность алгоритмических методов была подорвана работой Курта Геделя (1931 год), в которой было показано, что некоторые математические проблемы не могут быть решены с помощью алгоритмов из некоторого класса.

са. Этот класс алгоритмов определяется некоторой формальной конкретизацией понятием алгоритма. Встал вопрос: являются ли алгоритмически неразрешимыми эти проблемы только в рамках использованной Геделем модели алгоритма или же для решения этих проблем вообще нельзя придумать никакого алгоритма ни в каком смысле? Общность результата Геделя зависит от того, совпадает ли использованный им класс алгоритмов с классом всех алгоритмов в интуитивном смысле. Поэтому поиск и анализ различных уточнений и формализаций алгоритма и соотношение этих формализаций с интуитивным понятием алгоритма является практически важным.

К настоящему времени предложен ряд формальных моделей алгоритма. Курт Гедель определил алгоритм как последовательность правил построения сложных математических функций из более простых, Алонзо Черч использовал формализм, называемый λ -исчислением, Алан Тьюринг предложил гипотетическое автоматическое устройство, которое сейчас называется машиной Тьюринга, и определил алгоритм как программу для этой машины, А. А. Марков определил алгоритм как конечный набор правил подстановок цепочек символов и т. д.

Удивительным научным результатом является доказательство эквивалентности всех этих и нескольких других формальных определений алгоритма. Эквивалентность двух абстрактных моделей алгоритма состоит в том, что любой класс проблем, которые можно решить с помощью моделей одного типа, можно решить и на моделях другого типа (фактически в рамках одной модели можно выразить другие). Оказалось, что все алгоритмы в точном смысле для этих формальных моделей являются алгоритмами в интуитивном смысле, и все известные алгоритмы могут быть представлены алгоритмами в точном смысле (в рамках этих формализмов).

На основании этих результатов в информатике получило признание следующее положение: «Любое разумное определение алгоритма, которое может быть предложено в будущем, окажется эквивалентным уже известным определениям», что означает, по сути, предположение об *адекватности понятий алгоритма в интуитивном смысле и алгоритма в точном смысле в одном из перечисленных эквивалентных формализмов*. Это положение в настоящее время широко используется в качестве *гипотезы*, обоснованной в силу того, что не удалось найти противоречащих ей примеров. Эту гипотезу, однако, *невозможно доказать строго*, поскольку понятие алгоритма в интуитивном смысле является неформальным.

Исторически Алонзо Черч первый предложил отождествить интуитивное понятие алгоритма с одним из эквивалентных между собой точных определений. Алан Тьюринг независимо высказал предположение, что любой алгоритм в интуитивном смысле может быть представлен машиной Тьюринга (а значит, и в любой другой эквивалентной форме). Это предположение известно как *тезис Черча–Тьюринга*. Тезис Черча–Тьюринга просто отражает нашу уверенность в том, что разработанные формальные модели алгоритма достаточно полно представляют наше интуитивное его понимание.

Тезис Черча–Тьюринга имеет важное практическое значение. Например, поскольку каждый компьютер (с потенциально бесконечной памятью) может моделировать машину Тьюринга (см. пример 5.3) и, следовательно, алгоритмы в любом другом формализме, то из этого тезиса следует, что все компьютеры, как маленькие персональные компьютеры, так и большие суперкомпьютеры, эквивалентны с точки зрения *принципиальной* возможности решения алгоритмических проблем.

Определение машины Тьюринга среди других эквивалентных определений кажется наиболее удобным для формального определения понятия алгоритма.

Машина Тьюринга

По сути своей, алгоритм есть механический процесс обработки информации. Впервые Алан Тьюринг определил понятие алгоритма исходя из понятия автоматически работающей машины; более того, он предложил формальную модель такого устройства, которое интуитивно моделирует действия человека, решающего задачу, руководствуясь некоторым алгоритмом. Это устройство было названо машиной Тьюринга. Как оказывается, машина Тьюринга является весьма простым расширением модели конечного автомата.

При выполнении алгоритма в интуитивном смысле мы можем пользоваться потенциально неограниченной памятью, запоминая в процессе выполнения алгоритма по мере необходимости нужную информацию, например, на листочке бумаги. В то же время основным ограничением конечного автомата является конечность числа его состояний, а значит, его памяти. Можно предположить, что именно поэтому конечный автомат не может быть использован как модель устройства, выполняющего произвольные алгоритмы.

Если к модели КА добавить способность запоминания произвольно больших объемов информации, то его возможности по выполнению алгоритмов расширятся и мы получим автомат с более широкими возможностями по преобразованию информации, иными словами, с более широким классом алгоритмов, которые могут быть выполнены автоматами этого нового типа. Алан Тьюринг в 1936 году предложил формальную модель вычислителя, которая является результатом простого добавления потенциально бесконечной памяти к конечному автомatu.

Рассмотрим, чем отличается машина Тьюринга от простой модели конечного автомата. Конечный автомат можно представить себе как устройство с конечным числом внутренних состояний, работающее с двумя лентами: входной и выходной (рис. 5.1). Конечный автомат работает по тактам. На каждом такте он читает с помощью некоторой входной головки символ из обозреваемой ячейки входной ленты, изменяет свое состояние и печатает некоторый символ выходного алфавита в обозреваемую ячейку выходной ленты, после чего две его головки чтения и записи перемещаются на одну позицию вправо. Описание функционирования конечного

автомата можно считать его программой: в ней просто перечислено конечное число четверок (команд) $\langle s, a, p, y \rangle$, где s — текущее состояние, a — очередной входной сигнал, p — следующее состояние и y — очередной выходной сигнал. Программа КА — это просто перечисление аргументов и соответствующих результатов частично-определенной функции переходов и выходов автомата δ : $S \times X \rightarrow S \times Y$.

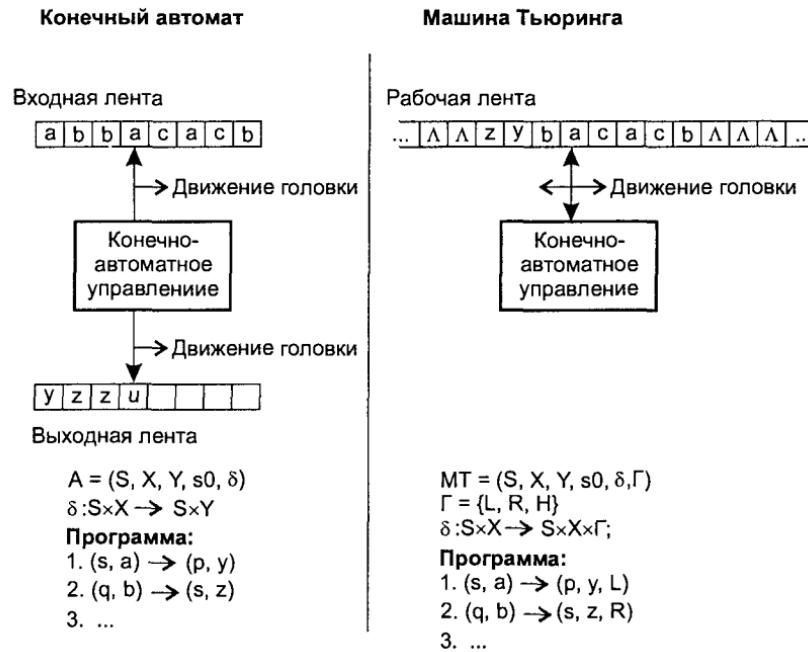


Рис. 5.1. Сравнение определений машины Тьюринга и конечного автомата

В своем вычислительном устройстве Тьюринг смоделировал доведенный до самых элементарных операций процесс выполнения произвольного алгоритма человеком. Человек имеет конечную память, и в этом смысле его можно представить системой с конечным числом состояний. Исходная информация к алгоритму обычно представляется в виде цепочки символов. Можно себе представить, что эта информация представлена в виде слова (конечной последовательности символов) конечного словаря. Выполняя алгоритм, человек-вычислитель использует дополнительную память (которая может быть потенциально бесконечной, например листы бумаги) для записи информации, причем эта запись производится им последовательно, символ за символом. При вычислениях человек может возвращаться к ранее записанной информации, стирать некоторую информацию и т. д. Таким образом, *элементарными операциями при выполнении алгоритма можно считать запись и стирание символа, а также перенесение внимания с одного участка записи на другой*. Предложенная Тьюрингом формальная модель отличается от конечно-

го автомата только в этих двух аспектах: она имеет одну бесконечную рабочую ленту, с которой читает и куда пишет символы, и одну головку чтения-записи, которая может двигаться по рабочей ленте в любую сторону (см. рис. 5.1). Такая свобода движения головки чтения-записи, по сути, означает возможность создавать и впоследствии анализировать промежуточную информацию. Как оказывается, такое простое расширение возможностей радикально увеличивает вычислительную мощность машин Тьюринга по сравнению с обычными конечными автоматами.

Машина Тьюринга работает по тактам. На каждом такте она читает символ из обозреваемой ячейки рабочей ленты, изменяет свое состояние в зависимости от своего внутреннего состояния и прочитанного символа и печатает символ в обозреваемую ячейку рабочей ленты, после чего ее головка чтения-записи может переместиться на одну позицию влево, вправо или остается на месте. Описание функционирования МТ можно считать ее программой, которая представлена конечным набором пятерок (команд) $\langle s, a, p, y, D \rangle$, где s, a, p и y имеют тот же смысл, что и в конечном автомате, а D — направление перемещения головки по рабочей ленте, которое может быть одним из трех значений: L — влево, R — вправо и H — оставаться на месте. Иными словами, программа MT — это просто конечный список пятерок, представляющих собой аргументы и соответствующие им результаты частично-определенной функции переходов и выходов δ : $S \times X \rightarrow S \times X \times \Gamma$.

Машина Тьюринга имеет один конечный рабочий алфавит X , в котором входные и выходные символы не различаются: выходной символ, напечатанный на ленте, машина может прочитать в последующих тактах. Для удобства обычно считают, что X содержит пустой символ Λ , находящийся во всех ячейках рабочей ленты слева и справа от конечной цепочки «значащих» символов в начале работы.

Рассмотрим, как работает машина Тьюринга. Конфигурацией машины Тьюринга называется ее текущее состояние, текущее состояние рабочей ленты и место расположения головки. При работе МТ в каждом такте происходит смена конфигураций. Пусть МТ находится в состоянии s и в обозреваемой ячейке ленты находится символ a . Если в программе МТ нет команды для пары $\langle s, a \rangle$, то МТ останавливается. Если в программе МТ несколько команд для данной пары $\langle s, a \rangle$, то это — недетерминированная машина Тьюринга, в ней выполняется одна из нескольких возможных команд с левой частью $\langle s, a \rangle$. Очевидно, что в любой момент работы на ленте МТ находится только конечная цепочка «значащих» символов. После останова машины Тьюринга эта цепочка является результатом переработки входной цепочки. Таким образом, МТ является *автоматом-преобразователем* символьных цепочек.

Машина Тьюринга также может быть *распознавателем* множеств цепочек. В такой МТ выделяются специальные заключительные состояния **стоп** или **«!»**, и если МТ, работающая как распознаватель, останавливается в одном из этих состояний при пустой входной ленте, то она распознает входную цепочку. Если в заключительном состоянии останавливается машина-преобразователь Тьюринга,

га, то информация на рабочей ленте является результатом переработки входной информации.

Примеры машин Тьюринга

Пример 5.1

Построим МТ, распознающую язык $\{a^nbc^n\}$. Представлять МТ будем подобно конечному автомату с помощью графа переходов, в котором каждой команде $\langle q, x, p, y, D \rangle$ соответствует ребро, направленное из вершины, помеченной состоянием q , в вершину, помеченную состоянием p . Само ребро помечено входным символом x , выходным символом y и направлением движения головки D (рис. 5.2).

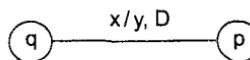


Рис. 5.2. Графическое представление команды машины Тьюринга $\langle q, x, p, y, D \rangle$

Только эта последняя деталь (указание направления движения головки) отличает граф переходов МТ от графа переходов конечного автомата.

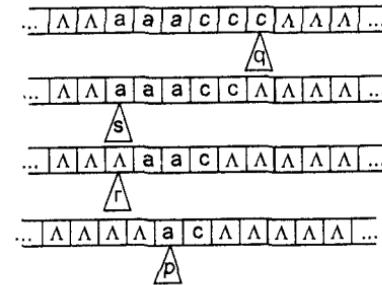
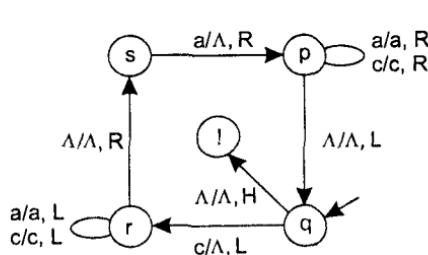
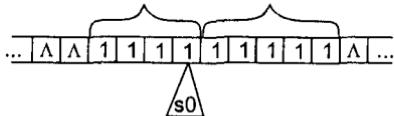


Рис. 5.3. Граф переходов машины Тьюринга, распознающей язык $\{a^nbc^n\}$, и несколько ее конфигураций при обработке входной цепочки «aaaccc»

Начальная конфигурация этой МТ (когда она находится в начальном состоянии q , а головка расположена против крайнего правого символа входной цепочки) и несколько промежуточных конфигураций, возникающих при ее работе (рис. 5.3), поясняют алгоритм распознавания.

Пример 5.2

Рассмотрим машину Тьюринга, находящую наибольший общий делитель двух положительных целых чисел, заданных в упарной системе счисления, в которой числу N соответствует N последовательных символов «1». Начальная конфигурация машины Тьюринга пусть будет такова.

Число $m = 4$ Число $n = 5$ 

Программу этой машины Тьюринга зададим с помощью графа переходов.

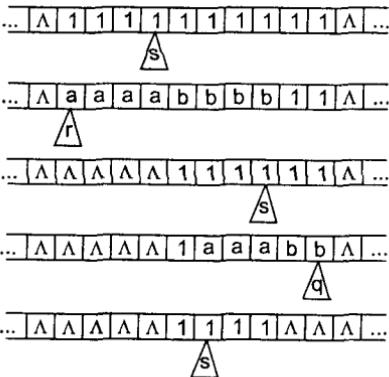
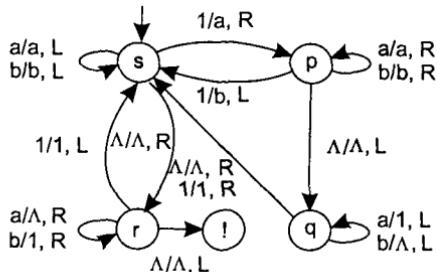


Рис. 5.4. Граф переходов машины Тьюринга, вычисляющей НОД двух чисел, и несколько ее конфигураций при обработке пары чисел $<4, 6>$

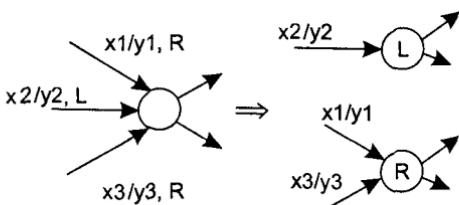
Стратегия вычисления НОД двух чисел здесь — это повторяющиеся действия по нахождению меньшего из двух чисел и вычитанию его из большего до тех пор, пока не получатся равные числа. Алгоритм работы этой МТ поясняется последовательно возникающими ее конфигурациями при обработке входа $(4, 6)$ на рис. 5.4.

Свойства машины Тьюринга

Рассмотрим несколько свойств машин Тьюринга.

Теорема 5.1. Для любой машины Тьюринга существует эквивалентная машина Тьюринга, такая, что все ребра, входящие в каждую вершину, помечены одним и тем же направлением движения головки.

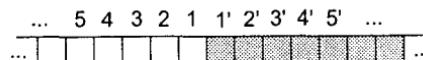
Доказательство легко следует из правила эквивалентного преобразования исходной машины Тьюринга.



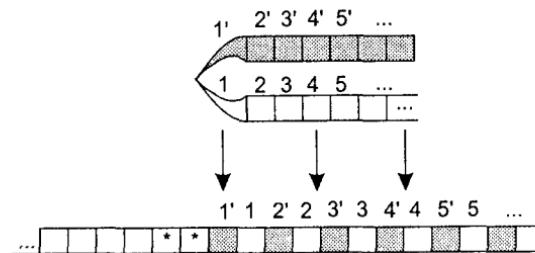
Каждое состояние исходной МТ расщепляется на несколько эквивалентных состояний, в каждое из которых приходят ребра только с одинаковой пометкой о направленности движения головки.

Теорема 5.2. Для любой машины Тьюринга существует эквивалентная машина Тьюринга, работающая на полубесконечной ленте.

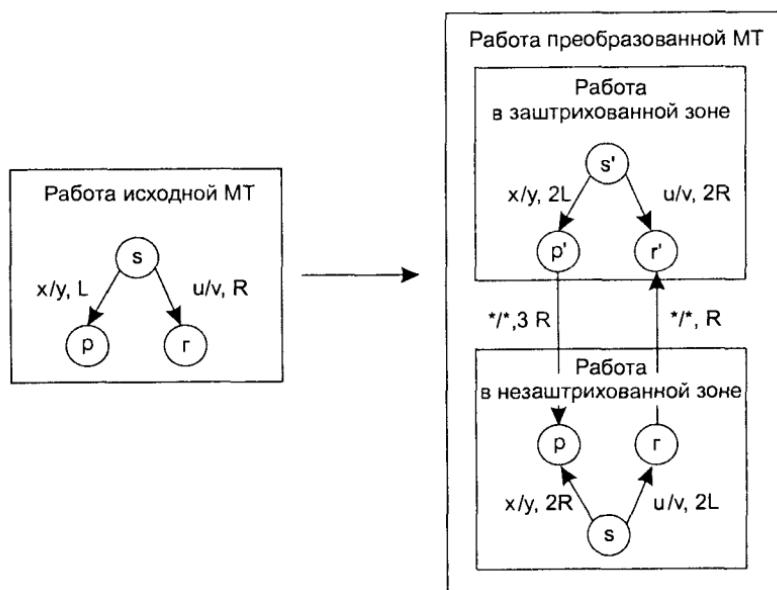
Доказательство этой теоремы конструктивное, то есть мы дадим алгоритм, по которому для любой машины Тьюринга может быть построена эквивалентная машина Тьюринга с объявленным свойством. Во-первых, занумеруем ячейки рабочей ленты МТ, то есть определим новое расположение информации на ленте:



Затем перенумеруем ячейки, причем будем считать, что символ «*» не содержится в словаре МТ:



Наконец, изменим машину Тьюринга, удвоив число ее состояний, и изменим сдвиг головки считывания-записи так, чтобы в одной группе состояний работа машины была бы эквивалентна ее работе в заштрихованной зоне, а в другой группе состояний машина работала бы так, как исходная машина работает в незаштрихованной зоне. Если при работе МТ встретится символ «*», значит головка считывания-записи достигла границы зоны.



Начальное состояние новой машины Тьюринга устанавливается в одной или другой зоне в зависимости от того, в какой части исходной ленты располагалась головка считывания-записи в исходной конфигурации. Очевидно, что слева от ограничивающих маркеров «*» лента в эквивалентной машине Тьюринга не используется.

Реализация машины Тьюринга

Рассмотрим вопрос о реализации машины Тьюринга. Очевидно, что *параллельную* память в современных компьютерах можно считать аналогом бесконечной в одну сторону рабочей ленты машины Тьюринга. Какого количества и каких команд достаточно, чтобы реализовать операции машины Тьюринга? Очевидно, что для этого достаточно только пяти типов команд.

Команда	Пояснение
сдвиг {влево, вправо}	Сдвиг головки на одну позицию на ленте влево или вправо
переход, р	Безусловный переход к команде с номером р
если, s, р	Условный переход к команде с номером р, если в обозреваемой ячейке находится символ s
печать, s	Печать символа s в обозреваемую ячейку
стоп	Останов

Например, начало программы машины Тьюринга, которая находит наибольший общий делитель (см. рис. 5.4), имеет вид:

```

00: если, a, 02; /*если прочитан символ a, то на команду 02*/
01: переход, 05; /*если нет, то на проверку другого символа*/
02: печать, a;
03: сдвиг, влево;
04: переход, 00; /*возврат в начальное состояние*/
05: если, b, 07; /*если прочитан символ b, то на команду 07 */
06: переход, 10; /* если нет, то на проверку другого символа*/
07: печать, b;
08: сдвиг, влево;
09: ...

```

Из этого примера можно сделать следующий вывод: для того чтобы автоматическое вычислительное устройство могло реализовать любой алгоритм, достаточно, чтобы оно могло работать с линейной памятью — аналогом полубесконечной входной ленты машины Тьюринга — и могло выполнять по крайней мере пять **перечисленных выше операций**. Известно, что все современные компьютеры включают подобные команды. Это означает, что все они являются универсальными вы-

числителями. Известно также, что кроме этих команд компьютеры включают также множество и других видов команд. Очевидно, эти дополнительные команды не **расширяют принципиально** возможностей компьютеров, но существенно повышают удобство и, главное, эффективность вычислителей.

Другой интересный вывод связан с возможностью реализации машины Тьюринга на языке высокого уровня. Поставим вопрос: какие управляющие конструкции достаточны для реализации любой машины Тьюринга? Очевидно, что с помощью последовательности операторов, выбора (**if-then-else**), цикла (**while**) и перехода (**goto**) любую программу машины Тьюринга можно реализовать. Однако справедливо и более сильное утверждение: оператор перехода в этом наборе лишний. Действительно, пусть *state* — переменная типа *состояние*, *symbol* — переменная типа *символ*. Пусть элементарными операциями языка являются *Читать(c)* — чтение очередного символа из обозреваемой ячейки входной ленты (или моделирующего ее массива) и помещение его в переменную *c*, *Писать(c)* — печатать символ *c* в обозреваемую ячейку входной ленты и *Сдвиг(D)* — имитация сдвига головки чтения-записи по рабочей ленте МТ (фактически изменение индекса читаемого элемента одномерного массива). Очевидно, что любая программа машины Тьюринга может быть построена так:

```

begin
j
state = s0;           // начальное состояние
while ( state != стоп ) do // повторяем цикл, пока не придем в заключительное состояние
{
    Читать(symbol);
    ...
    if (state == s & & symbol == x) // реализация
    {
        // команды машины Тьюринга
        state = p; // <(s, x) →(p, y, D)>
        Писать(y);
        Сдвиг (D)
    }
    else
        if (state == j
        ...
end

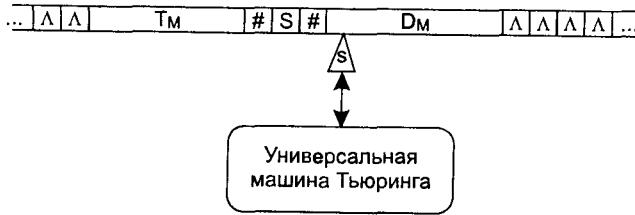
```

Отсюда следует важный вывод: любую машину Тьюринга (и, следовательно, **любой алгоритм**) можно реализовать на языке высокого уровня с использованием только трех управляющих конструкций: **последовательность, выбор и цикл**. Тем самым мы доказали известную основную теорему структурного программирования, доказанную впервые совсем из других соображений Бомом и Якопини. Смысл этой важнейшей теоремы и более широко, структурного программирования как подхода в целом, состоит в том, что для спецификации алгоритмов достаточно очень небольшого числа программных структур — последовательности, выбора и цикла, каждая с одной точкой входа и одной точкой выхода. Использование только этих

простых конструкций проясняет соотношение между статическим текстом программы и возможными динамическими путями всех выполнений программы.

Универсальная машина Тьюринга

Программа машины Тьюринга представляет собой конечный набор пятерок — команд вида $(s, x) \rightarrow (p, y, D)$. Если составить список таких пятерок и указать начальное состояние, то любой человек может сымитировать работу данной машины Тьюринга. Эта работа будет выполняться по шагам. На каждом шаге очередное состояние и очередной входной символ (тот, который находится в обозреваемой ячейке ленты) составляют пару, которую надо найти в левой части одной из команд. Если такой пары нет, машина останавливается. Если такая пара найдена, то остальные три символа пятерки представляют собой следующее состояние, символ, который надо напечатать в обозреваемой ячейке, и направление движения вдоль ленты. Очевидно, что этот алгоритм может быть выполнен и автоматически, например, подходящей машиной Тьюринга. МТ, выполняющая такой алгоритм, называется «Универсальной машиной Тьюринга». В качестве исходной информации она может иметь описание D_M имитируемой машины M , ее начальное состояние S и входную ленту T_M этой машины.



Если машина Тьюринга M работает на полубесконечной ленте, то универсальная машина Тьюринга может сымитировать ее работу, тратя несколько шагов своей работы на имитацию каждого шага машины M . Интересен вопрос: что будет делать универсальная машина Тьюринга, если в качестве имитируемой машины ей предложить описание ее самой?

Реализация универсальной машины Тьюринга остается читателю в качестве упражнения.

Алгоритмически неразрешимые проблемы

Идея построения алгоритмов — рецептов решения проблем — тысячи лет. Люди придумали множество алгоритмов для решения разнообразных проблем. Существуют ли какие-нибудь проблемы, для которых алгоритмов найти нельзя?

Выше мы упоминали о существовании алгоритмически неразрешимых проблем, то есть проблем, для которых не существует алгоритмов их решения. Утвержде-

ище это весьма сильное. Оно означает не только то, что мы не знаем **сейчас** соответствующего алгоритма, оно говорит о том, что *такого алгоритма мы никогда найти не сможем*. Долгое время математики считали, что для любой четко сформулированной проблемы алгоритм найти можно. Например, Д. Гильберт считал, что в математике не может быть неразрешимых проблем: «*In mathematics there is nothing which cannot be known*». Его целью в начале XX века было представить математику в виде такой формальной системы, что в ней все проблемы могли бы быть сформулированы в виде утверждений, которые либо истинны, либо ложны, а также найти алгоритм, который по заданному утверждению в этой системе мог бы определить его истинность. Гильберт рассматривал эту проблему как фундаментальную открытую проблему математики.

Впервые Курт Гедель в 1931 году представил строго сформулированную математическую проблему, для которой не существует работающего ее алгоритма. Сам по себе этот факт является удивительным, однако он имеет также большое практическое значение. Для проблем, алгоритмическая неразрешимость которых доказана, бессмысленно искать методы их решения точно так же, как бессмыслены поиски вечного двигателя. Оказалось, что таких алгоритмически неразрешимых проблем много, например, не существует алгоритма проверки общезначимости логической формулы в логике предикатов. Мы сейчас сформулируем одну из таких проблем и докажем ее алгоритмическую неразрешимость. Эта проблема известна как «проблема останова».

Не для каждой программы очевидно, что она завершится. Например, очень долго было неизвестно, справедлива ли следующая теорема Ферма: «Не существует таких целых a, b, c , что $a^n + b^n = c^n$ ». Построим алгоритм:

```
bool Fermat (unsigned n)
    for (unsigned a = 1; true: a++)
        for (unsigned b = 1; true: b++)
            for (unsigned c = 1; true: c++)
                if (a**n + b**n == c**n) stop;
```

Останавливается ли этот алгоритм для $n = 724$?

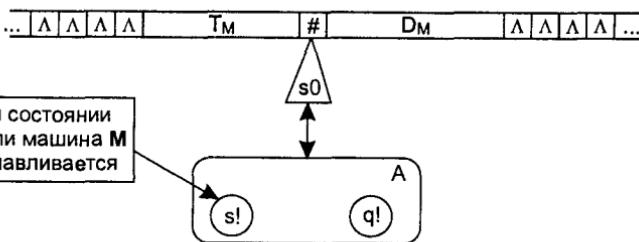
Удобно иметь универсальный алгоритм А, который после анализа записи произвольного алгоритма Х и его исходных данных даст ответ, останавливается ли алгоритм Х или нет. Очевидно, что такой универсальный алгоритм А был бы очень полезен: он мог бы дать решение не только теоремы Ферма, но и многих других подобных проблем. Однако справедлива следующая теорема.

Теорема 5.3. *Не существует алгоритма, позволяющего по описанию произвольного алгоритма и его исходных данных определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.*

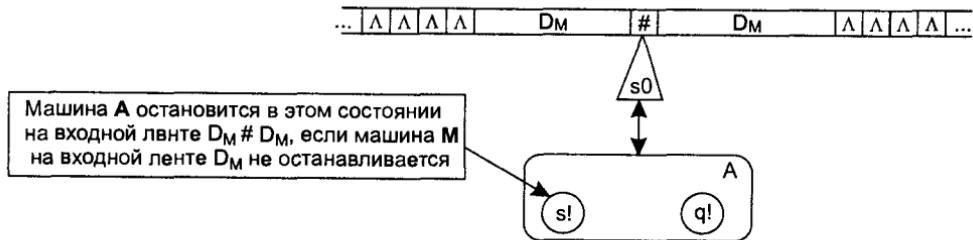
Доказательство. Будем доказывать эту теорему для представления алгоритмов как машин Тьюринга, а их исходных данных — как цепочки символов на рабочей ленте. Если мы докажем, что не существует машины Тьюринга, работающей эту проблему, то на основании тезиса Черча–Тьюринга не может существовать никакого алгоритма ее решения.

Итак, в эквивалентной формулировке теорема звучит так: не существует машины Тьюринга, которая по описанию произвольной машины Тьюринга и ее входной ленты определяет, останавливается ли эта машина Тьюринга на этой входной ленте или работает бесконечно.

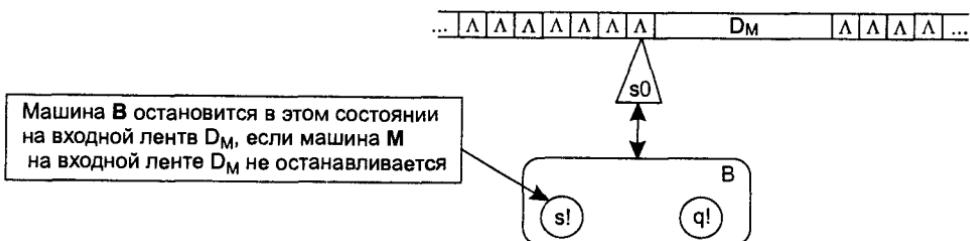
Предположим противное, то есть что такая машина Тьюринга A существует. Машина Тьюринга A должна иметь два заключительных состояния, $s!$ и $q!$, такие, что по описанию произвольной машины Тьюринга D_M и ее входной ленты T_M машина A придет в состояние $s!$ и остановится, если M на входной ленте T_M не останавливается, и машина A придет в состояние $q!$ и остановится, если M на входной ленте T_M останавливается:



Частным случаем входной ленты T_M является такой вход машины M, который является описанием ее самой.

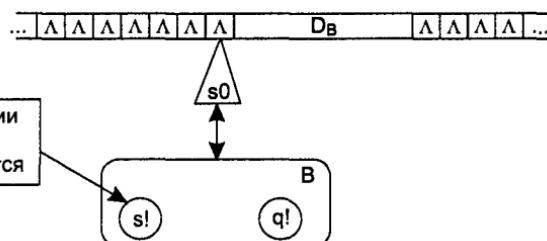


Таким образом, если A существует, то по описанию любой машины Тьюринга M она перейдет в состояние $s!$ и остановится, если M не останавливается, имея в качестве входной ленты описание самой себя. Но если существует A, то существует и машина B, которая сначала печатает маркер «#», копирует содержимое входной ленты слева от маркера, а потом работает, как A.



Таким образом, если A существует, то имея на входе описание D_M любой машины Тьюринга M, машина Тьюринга B перейдет в состояние $s!$ и остановится, если M не останавливается, имея в качестве входной ленты описание D_M .

Рассмотрим, как будет себя вести машина B, если в качестве входа ей предъявить описание ее самой!



Из предыдущего следует, что если наше предположение верно, то существует машина Тьюринга B, которая, имея на входе описание D_B , перейдет в состояние $s!$ и остановится, если B не останавливается, имея в качестве входной ленты описание D_B .

Таким образом, мы пришли к противоречию, и исходное предположение о существовании машины Тьюринга A, решающей проблему останова, неверно. Теорема доказана.

Проблема останова не единственная, для которой не существует алгоритмов решения. Многие другие проблемы, часто важные для практики, оказались алгоритмически неразрешимыми. Перечислим некоторые из них.

- Проблема эквивалентности алгоритмов:** по двум произвольным заданным алгоритмам (представленным, например, в виде программ для машины Тьюринга или на языке программирования Pascal) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.
- Проблема эквивалентности грамматик:** по двум произвольным грамматикам, представленным в виде наборов синтаксических диаграмм, определить, совпадают ли определяемые ими языки.
- Проблема тотальности:** по произвольному заданному алгоритму определить, будет ли он завершаться (останавливаться) на всех возможных входных наборах.
- Десятая проблема Гильберта:** «Найти алгоритм, который по заданному многочлену $P(x_1, \dots, x_n)$ с целыми коэффициентами определяет, имеет ли уравнение $P = 0$ решение в целых числах». Ю. В. Матиясевич [25] показал, что такого алгоритма не существует.
- Проблема соответствия (Э. Пост):** «Пусть $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_n, \beta_n)$ — упорядоченные пары непустых цепочек. Существует ли такое число $p > 0$, для которого цепочки $\alpha_{i1} \dots \alpha_{ip}$ и $\beta_{i1} \dots \beta_{ip}$ совпадают?»

Множество интересных проблем в теории формальных грамматик также неразрешимы. Например, задают ли две контекто-свободные грамматики один и тот же язык? Не существует также алгоритма, проверяющего непустоту пересечения КС-языков.

Частично разрешимые проблемы

Существует множество алгоритмически неразрешимых проблем и некоторые из них представлены выше. Однако некоторые проблемы «менее» неразрешимы, чем другие. Для простоты ограничимся только такими проблемами, которые в качестве результата выдают только «ДА» или «НЕТ».

Рассмотрим проблему останова. Она состоит в том, что по заданной программе M и ее входным данным D необходимо определить, останавливается M на D или нет. Это как раз проблема с возможными ответами «ДА» и «НЕТ». Очевидно, что мы можем **частично** решить эту проблему, вовсе не составляя специального алгоритма, анализирующего M и D , а просто запустить M на D . Если M остановится на D , то ответ будет «ДА». Трудности возникают только, если M не останавливается на D . В каждый момент, пока алгоритм не остановился, мы не знаем, остановится ли он впоследствии или нет. Иными словами, не существует алгоритма, который смог бы определить, что M не останавливается на D . Проблема останова является **«частично разрешимой»** проблемой.

К этому же типу частично разрешимых проблем относится и проблема невыполнимости логической формулы в исчислении предикатов (или сводящейся к ней проблемы логического вывода с предикатами). Если формула невыполнима, метод резолюции обязательно остановится и выдаст ответ «ДА». Однако если формула не является невыполнимой, в общем случае алгоритм резолютивного вывода должен перебрать все возможные резолюции, а их может быть бесконечное количество при бесконечной области определения переменных. Оказывается, не существует и никакого другого алгоритма, позволяющего проверить невыполнимость произвольной формулы логики предикатов и гарантирующего всегда ответы «ДА» или «НЕТ». Поэтому, как указано в главе 2, метод резолюции — это самый лучший из возможных методов (с точки зрения не эффективности, конечно, а с точки зрения принципиальной возможности получения результата).

Проблема эквивалентности и проблема тотальности не являются частично разрешимыми. Иными словами, не существует алгоритма, который для любой пары алгоритмов давал бы ответ «ДА», даже если эти алгоритмы эквивалентны.

Контрольные задания

- Построить МТ, распознающую язык Дикка — скобочные скелеты арифметических выражений. Примеры цепочек этого языка: «(())»; «((()) (((())))».
- Построить МТ, распознающую язык $\{a^{n^2}\}$. Примеры цепочек этого языка: «а»; «аааа».

Указание. Воспользоваться соотношением: $(k + 1)^2 = k^2 + 2k + 1$ или тем, что квадрат натурального числа представим как сумму последовательных нечетных чисел.

3. Построить МТ, распознающую язык, цепочки которого содержат одинаковые количества вхождений символов a , b и c . Пример цепочки этого языка: « $baacbc\bar{c}ba$ ».
4. Построить МТ, а) увеличивающую на один двоичные числа; б) уменьшающую на один двоичные числа.
5. Построить МТ, переводящую число из пятиричной системы счисления в семиричную.
6. Построить МТ, переводящую двоичное число в унарную форму, и наоборот, из унарной формы в двоичное представление.
7. Построить МТ, подсчитывающую произведение двух заданных чисел в унарной записи.

Литература

1. *Borshchev A. V., Karpov Yu. G., Roudakov V. V., Filippov A., Sintotskij A., Fedorenko S.* Analysis of a Distributed Election Algorithm Using COVERS 3.0. A Case Study // Proceedings of the 4th Int. Conference. PaCT-97. Lecture Notes in Computer Science, № 1277, 1997, pp. 409–423.
2. *Bryant R. E.* Graph-Based Algorithms for Boolean Function Manipulation // IEEE Transaction on Computers, vol. C-35, № 8, August 1986, pp. 677–691.
3. *Carroll J., Long D.* Theory of Finite Automata with Introduction to Formal languages // Prentice-Hall International, 1989, 438 p. [CL89].
4. *Clocksin W. F.* Logic programming and digital circuit analysis // J. Logic Programming, 1987, 4, 59–82.
5. *Friedman S., Supowit K.* Finding the optimal Variable Ordering for Binary Decision Diagrams // IEEE Transactions on Computers, vol. C-39, № 5, May 1990, pp. 711–13.
6. *Goldschlager L., Lister A.* Computer Science. A modern introduction. // Prentice Hall, 1988, 330 p.
7. *Riccardo Gusella, Zatti Stefano.* An Election Algorithm for a Distributed Clock Synchronization Program // 6th International Conference on Distributed Computing Systems, 1986, pp. 364–371.
8. *Harel D.* Statecharts: a Visual Formalism for complex systems // Science of Computer Programming. Vol. 8, 1987, pp. 231–274.
9. *Koegst M., Franke G., Feske K.* State Assignment for FSM Low Power Design // Proceedings of European Design Automation Conf. «EURO-DAC'96», Geneva, Switzerland, 1996, pp. 28–33.
10. *Pell A., Goh C., Mellor P.* Data + Understanding = Management // Hewlett-Packard Laboratories Technical Report, HPL-93 – 24, March, 1993.
11. *Robinson J. A.* A Machine-Oriented Logic Based on the Resolution Principle // Journal of the ACM. Vol. 12, 1965, № 1.

12. Turek J., Sasha D. The Many Faces of Consensus in Distributed Systems // IEEE Computer, July, 1992.
13. Биркгоф Г., Барти Т. Современная прикладная алгебра. М.: Мир, 1976.
14. Вуд Дж., Серре Ж. Дипломатический церемониал и протокол. М., 1976.
15. Гаазе-Рапопорт М. А., Поспелов Д. А. От амебы до робота: модели поведения. М.: Наука, 1987.
16. Грис Д. Наука программирования. М.: Мир, 1984.
17. Гузовская А. Н. и др. Язык формального описания протоколов и сервисов (ФОПС). АН Латв. ССР. Институт Электроники и вычислительной техники, Рига, 1986.
18. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
19. Дейкстра Э. Взаимодействие последовательных процессов//Языки программирования (под ред. Ф. Женюи). М.: Мир, 1972. С. 9–86.
20. Ивлев Ю. В. Логика/Учебник для вузов. М.: 1998.
21. Институт международных образовательных программ. СПбГТУ, 1999.
22. Клини С. Математическая логика. М.: Мир, 1978.
23. Кузнецов О. П., Адельсон-Вельский Г. М. Дискретная математика для инженера. М.: Энергоатомиздат, 1988.
24. Кузнецов О. П., Адельсон-Вельский Г. М. Дискретная математика для инженера. М.: Энергия, 1980.
25. Матиясевич Ю. В. Диофантовы перечислимые множества//Доклады АН СССР, 1970. Т. 191. С. 279–282.
26. Мендельсон Э. Введение в математическую логику. М., 1971.
27. Минский М. Вычисления и автоматы. М.: Мир, 1978.
28. Никольская И. Л. Математическая логика. М.: Высшая школа, 1981.
29. Пензов Ю. Е. Элементы математической логики и теории множеств. Изд-во Саратовского университета, 1968.
30. Попов Б. Н. Анализ и синтез законов управления системой «Импульсный усилитель мощности – электродвигатель»//Известия АН РАН. Теория и системы управления. 1996, № 3. С. 94–102.
31. Поспелов Д. А. Логические методы анализа и синтеза схем. М.: Энергия, 1974.
32. Поспелов Д. А. Фантазия или наука. На пути к искусственному интеллекту. М.: Наука, 1982.
33. Рейнорд-Смит В. Дж. Теория формальных языков. Вводный курс. М.: Радио и связь, 1988.
34. Смалиан Р. Принцесса или тигр. М.: Мир, 1985.
35. Таненбаум А. Компьютерные сети. СПб.: Питер, 2002.

36. Фудзисава Т., Касами Т. Математика для радиоинженеров. Теория дискретных структур. М.: Радио и связь, 1984.
37. Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983.
38. Черч А. Введение в математическую логику. М.: Издательство иностранной литературы, 1960.
39. Шапиро С. И. Решение логических и игровых задач. М.: Радио и связь, 1984.