Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ТЕОРИЯ АЛГОРИТМОВ

Курс лекций
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2006

СОДЕРЖАНИЕ

1. ОСНОВЫ ТЕОРИИ АЛГОРИТМОВ

1.1. НЕФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ АЛГОРИТМА И НЕОБХОДИМОСТЬ ЕГО УТОЧНЕНИЯ

Уже на самых ранних этапах развития математики в ней стали возникать различные вычислительные процессы чисто механического характера; с их помощью искомые величины ряда задач вычислялись последовательно из данных исходных величин по определённым правилам и инструкциям. Со временем все такие процессы в математике получили название *алгоритмов*. Примерами алгоритмов могут служить процесс нахождения наибольшего общего делителя двух положительных натуральных чисел (*алгоритма Евклида*); процесс сложения десятичных чисел «в столбик»; процесс решения системы линейных уравнений методом последовательного исключения неизвестных, и т. д.

Дадим более формальное определение рассмотренным понятиям.

Определение 1.1.1. (Массовая проблема, алгоритм) *Пусть задано некоторое непустое множество X, элементы которого будем называть исходными объектами, и непустое множество Y, элементы которого будем называть искомыми объектами. Будем говорить, что сформулирована массовая проблема (или массовая задача), если требуется для любого исходного объекта указать определённый искомый объект. Если существует универсальный способ, который позволяет это сделать, то будем говорить, что заданная массовая проблема имеет решение, а указанный способ будем назвать алгоритмом решения массовой проблемы.*

Рассмотрим примеры массовых проблем.

1. Пусть в качестве множества исходных объектов выступают массивы натуральных чисел заданной длины n. В качестве искомых объектов используется то же множество. Сформулируем *массовую проблему сортировки*: по любому массиву требуется указать массив, состоящий из тех же элементов, но упорядоченных по возрастанию. Данная проблема имеет решение: алгоритм сортировки. Заметим, что решение данной проблемы не единственно.

2. Пусть исходные объекты – тройки рациональных чисел (a, b, c), множество искомых объектов – {«да», «нет»}. Массовая проблема формулируется следующим образом: по тройке чисел (a, b, c) требуется выдавать ответ «да», если уравнение $ax^2 + bx + c = 0$ имеет действительные корни, и ответ «нет» в противном случае. Очевидно, что данная массовая проблема также имеет решение.

3. Пусть множество исходных объектов – это множество программ, записанных на языке Pascal, множество искомых объектов – {«да», «нет»}. Массовая проблема: по программе определить, зацикливается она (ответ «да») или нет (ответ «нет»). Существование или отсутствие алгоритма решения этой массовой проблем не очевидно.

Приведённое выше определение алгоритма как способа решения определённой массовой проблемы не является математически строгим. Однако все алгоритмы обладают некими общими чертами, или свойствами, ясно вырисовывающимися из предыдущих примеров и признающихся характерными для понятия алгоритма:

1. Алгоритм работает с конструктивными объектами. Под *конструктивным объектом* понимается такой объект, который может быть описан при помощи конечного слова в некотором алфавите. Примерами конструктивных объектов могут служить рациональное число, массив рациональных чисел, слово. Входные данные алгоритма, промежуточные величины и конечный результат алгоритма всегда являются такими конструктивными объектами.

2. Алгоритм – это дискретный процесс. Процесс построения результатов алгоритма идёт в дискретном времени. В начальный момент задаётся исходная конечная система объектов, а в каждый последующий момент времени (на каждом *шаге алгоритма*) новая система объектов получается по определённому закону из системы объектов, имевшихся на предыдущем шаге.

3. Алгоритм детерминирован. Система объектов, получаемых на любом (не начальном) шаге алгоритма, однозначно определяется объектами, полученными в предшествующие шаги.

4. Шаг алгоритма элементарен. Закон получения последующей системы объектов из предшествующей должен быть простым и локальным.

5. Алгоритм обладает свойством массовости. Начальная система объектов алгоритма может выбираться из некоторого (потенциально бесконечного) множества.

Понятие алгоритма, определяемое этими пятью свойствами, также не является строгим: в формулировках свойств встречаются слова «способ», «объект», «простой», «локальный», точный смысл которых не установлен. В дальнейшем это нестрогое понятие алгоритма будет называться *интуитивным понятием алгоритма*.

Интуитивное понятие алгоритма хоть и не является строгим, но оно настолько ясно, что к началу XX века практически не было серьёзных случаев, когда математики разошлись бы во мнениях относительно того, является ли алгоритмом тот или иной конкретно заданный процесс. Однако в XX веке положение существенно изменилось. На первый план выдвинулись такие массовые проблемы, существование алгоритма решения которых было сомнительным. Действительно, одно дело – доказать существование алгоритма, другое дело – доказать отсутствие алгоритма. Первое можно сделать путём фактического описания процесса, решающего задачу; в этом случае достаточно и интуитивного понятия алгоритма, чтобы удостовериться в том, что описанный процесс есть алгоритм. Доказать отсутствие алгоритма таким путём невозможно. Для этого надо точно знать, что такое алгоритм. В 1920-х годах задача точного определения понятия алгоритма стала одной из центральных математических проблем. Она была решена в 1936 году, когда практически одновременно и независимо друг от друга были опубликованы работы А. Чёрча, С. К. Клини,

А. М. Тьюринга и Э. Л. Поста. Хотя в техническом отношении полученные уточнения понятия алгоритма были различными, вскоре была установлена взаимная моделируемость этих понятий. Произведённое уточнение дало немедленный эффект – вскоре была доказана алгоритмическая неразрешимость многих прикладных математических задач.

1.2. АРИФМЕТИЧЕСКИЕ И ИНТУИТИВНО ВЫЧИСЛИМЫЕ ФУНКЦИИ

Первым шагом на пути к математическому уточнению интуитивного понятия алгоритма является сведение решения массовой проблемы к процессу вычисления значений некоторой функции.

Сделаем следующее замечание. Начиная этого момента под множеством натуральных чисел будем понимать целые неотрицательные числа. Таким образом, $\mathbf{N} = 0, 1, ...$

Определение 1.2.1. (Арифметическая функция) *Функцию будем называть арифметической, если ее аргументы и значение принадлежат множеству натуральных чисел. Таким образом, арифметические функции – это функции вида* $f : \mathbf{N}^n \to \mathbf{N}$.

Пусть имеется некая массовая проблема с множеством исходных объектов X и множеством искомых объектов Y. Для существования решения массовой проблемы необходимо, чтобы элементы множеств X и Y были конструктивными объектами. Следовательно, элементы этих множеств можно занумеровать натуральными числами[1]. Пусть $x \in X$ – некий исходный объект, обозначим через $n(x)$ его номер. Если в массовой проблеме по исходному объекту x требуется получить искомый объект $y \in Y$ с номером $n(y)$, то определим арифметическую функцию $f : \mathbf{N}^n \to \mathbf{N}$, такую что $f(n(x))=n(y)$.

В качестве примера построения арифметических функций по массовым проблемам рассмотрим массовые проблемы из предыдущего параграфа.

1. Если дан массив $[x_1, x_2, \ldots, x_n]$ натуральных чисел, то ему в соответствие можно поставить натуральное число $2^{x_1} \cdot 3^{x_2} \cdot \ldots \cdot p(n)^{x_n}$, где $p(n)$ – n-е простое число. Рассмотрим для примера массив длины 5:

$$[x_1, x_2, x_3, x_4, x_5] \mapsto 2^{x_1}3^{x_2}5^{x_3}7^{x_4}11^{x_5}.$$

Данная нумерация определяет арифметическую функцию f (например, $f(73500) = f(2^2 3^1 5^3 7^2 11^0) = 2^0 3^1 5^2 7^2 11^3 = 4891425$).

2. Любое рациональное число имеет некоторый натуральный номер. Нумерация множества искомых объектов проблемы тривиальна:

$$\{\text{«да»}, \text{«нет»}\} \mapsto \{1, 0\}.$$

Для данной массовой проблемы можно построить арифметическую функцию одного аргумента, воспользовавшись приемом из предыдущего примера, а

[1] Процесс нумерации конструктивных объектов можно также рассматривать в качестве массовой проблемы.

можно рассмотреть функцию трех аргументов (три номера элементов исходной тройки).

3. Нумерация текстов программ может быть осуществлена следующим образом: любую программу можно воспринимать как запись числа в 256-ричной системе счисления (если для записи программы использовались символы таблицы ASCII).

Переход от массовой проблемы к арифметической функции позволяет свести вопрос о существовании решения массовой проблемы к вопросу существования эффективного способа вычислений значений арифметической функции по ее аргументу (аргументам).

Выделим из всех арифметических функций следующее подмножество.

Определение 1.2.2. (Вычислимая функция) *Арифметическую функцию* $f: \mathrm{N}^n \to \mathrm{N}$ *назовем вычислимой, если существует алгоритм, позволяющий для любого набора значений ее аргументов вычислить значение функции (или указать, что функция на данном наборе не определена)*.

Так как в определении вычислимой функции используется интуитивное понятие алгоритма, то часто вместо термина «вычислимая функция» используется термин «интуитивно вычислимая функция».

Таким образом, массовая проблема имеет решение, если арифметическая функция, соответствующая этой проблеме, является интуитивно вычислимой. Вместо уточнения понятия алгоритма можно рассматривать уточнение понятия «вычислимая функция». Обычно при этом действуют по следующей схеме:

1. Вводят точно определенный класс функций.

2. Убеждаются, что все функции из этого класса являются вычислимыми.

3. Принимают гипотезу (тезис) о том, что класс вычислимых функций совпадает с введенным классом функций.

1.3. МАШИНЫ ТЬЮРИНГА

Машины Тьюринга предложены Аланом Тьюрингом[1] в 1936 году для уточнения интуитивного понятия алгоритма.

Машина Тьюринга состоит из следующих компонентов.

1. Внешний алфавит $A = \{a_0, a_1, a_2, \ldots, a_n\}$. Внешний алфавит состоит из конечного множества символов. Одна буква алфавита – a_0 – является особой. Это пустая буква (пробел). В дальнейшем такую пустую букву будем обозначать как λ.

2. Лента – потенциально бесконечная в обе стороны и разделённая на ячейки. Каждая ячейка содержит один из символов алфавита A, но только конечное множество ячеек не заполнены пустой буквой.

3. Управляющая головка. Головка способна воспринимать символ в ячейке ленты и изменять его, а также передвигаться вправо и влево. При этом в конкретный момент времени воспринимается только одна определённая ячейка, и

[1] Тьюринг А. (A. Turing).

за один такт работы машины Тьюринга головка перемещается либо на одну ячейку вправо, либо на одну ячейку влево, либо остаётся на месте.

4. Внутренний алфавит $Q = \{q_0, q_1, q_2, \ldots, q_m\}$. Внутренний алфавит служит для представления всех возможных внутренних состояний машины Тьюринга. В любой конкретный момент времени машина Тьюринга находится в одном из этих состояний. После одного такта работы состояние машины может измениться. Имеются два особых состояния машины Тьюринга: q_0 – заключительное (пассивное) состояние, если машина Тьюринга попадает в это состояние, то она прекращает работу; q_1 – начальное состояние из которого стартует машина.

5. Программа.. Это таблица, которая состоит из $n + 1$ столбцов и m строк. В ячейки таблицы помещается команда вида $a_j t q_l$.

Таблица 1

Запись программы для машины Тьюринга

	a_0	a_1	...	a_i	...	a_n
q_1						
...						
q_k				$a_j t q_l$		
...						
q_m						

Если команда $a_j t q_l$ находится в строке q_k, столбце a_i, то она означает следующее: если машина Тьюринга находится в состоянии q_k, а управляющая головка при этом обозревает символ a_i, то

1. головка записывает в обозреваемую ячейку символ a_j (возможно, $a_j = a_i$);
2. головка должна переместиться в направлении $t \in \{L, S, R\}$, где L означает «влево», R – «вправо», а S – «оставаться на месте»;
3. машина должна перейти в состояние q_l.

Некоторые ячейки таблицы могут быть пустыми, то есть команд в программе не более чем $m(n+1)$.

Заметим, что программа полностью определяет работу машины Тьюринга. Более того, если известна программа, можно восстановить машину Тьюринга, то есть внешний алфавит и множество внутренних состояний.

Точное определение машины Тьюринга выглядит следующим образом.

Определение 1.3.1. (Машина Тьюринга) *Будем называть машиной Тьюринга систему вида $(A, a_0, Q, q_0, q_1, Sh, \tau)$, где*

1. A – конечное множество, внешний алфавит;

2. $a_0 \in A$ – пустая буква алфавита;

3. Q – конечное множество, внутренние состояния;

4. $q_0 \in Q$ – заключительное состояние;

5. $q_1 \in Q$ – начальное состояние;

6. $Sh = \{L, S, R\}$ – множество возможных сдвигов управляющей головки;

7. τ – программа, отображение вида

$$\tau : A \times (Q \setminus \{q_0\}) \ni (a_i, q_k) \mapsto (a_j, t, q_l) \in A \times Sh \times Q.$$

Отметим, что машина Тьюринга представляет собой точно определённый математический объект.

1.4. ВЫЧИСЛИМОСТЬ ПО ТЬЮРИНГУ

Основной целью данного параграфа будет описание класса арифметических функций, вычислимых на машинах Тьюринга.

Для представления натуральных чисел на ленте машины Тьюринга будем использовать «единичную» систему счисления. Натуральные числа в данной системе счисления записываются следующим образом: 0 = |, 1 = ||, 2 = ||| и т. п.

Рассмотрим машины Тьюринга с внешним алфавитом $A = \{\lambda, |\}$.

Опишем процесс вычисления на машине Тьюринга значений некоторой арифметической функции $f(x_1, x_2, \ldots, x_n)$. Пусть дан набор значений аргументов $(a_1, a_2, \ldots, a_n)$ функции f. Запишем значения аргументов в «единичной» системе счисления на ленте машины Тьюринга, разделив их пробелом:

$$\underbrace{||\ldots|}_{a_1}\lambda\underbrace{||\ldots|}_{a_2}\lambda\ldots$$

Поместим управляющую головку под самый левый символ |. Запустим машину. Возможны следующие ситуации:

1. Машина Тьюринга через некоторое время останавливается, при этом на ленте машины Тьюринга записано некое число m в «единичной» системе счисления (и больше ничего).

2. Машина Тьюринга «зацикливается», то есть никогда не останавливается.

3. Машина Тьюринга останавливается, однако на ленте записано более чем одно число в единичной системе счисления.

Определение 1.4.1. (Функция, вычислимая по Тьюрингу) *Будем говорить, что машина Тьюринга вычисляет арифметическую функцию $f(x_1, x_2, \ldots, x_n)$, если для любого набора $(a_1, a_2, \ldots, a_n)$ значений аргументов функции машина либо получает в качестве ответа значение m, причём $m = f(a_1, a_2, \ldots, a_n)$, либо не останавливается, если функция f на наборе $(a_1, a_2, \ldots, a_n)$ не определена. Арифметическую функцию будем называть вычислимой по Тьюрингу, если существует машина Тьюринга, которая ее вычисляет.*

Множество всех вычислимых по Тьюрингу функций обозначим через T.

По определению, для установления того, является ли некая арифметическая функция вычислимой по Тьюрингу, достаточно предъявить машину, которая вычисляет эту функцию. Рассмотрим несколько примеров вычислимых по Тьюрингу функций.

1. $f_1(x) \equiv 0$. Покажем, что $f_1(x) \in T$. Построим машину, которая её вычисляет. Эта машина должна работать следующим образом: стереть на ленте все символы |, затем написать один символ | и остановится. Вот программа, которая это выполняет:

	λ	|
q_1	$\|Sq_0$	λRq_1

2. $f_2(x,y) = x + y$. Программа для вычисления данной функции должна работать следующим образом: стереть два первых символа |, затем переместить головку вправо до пустой ячейки, записать туда символ | и остановится.

	λ	|
q_1	–	λRq_2
q_2	λSq_0	λRq_3
q_3	$\|Sq_0$	$\|Rq_3$

Команда в ячейке λq_2 предназначена для исходного слова вида $|\lambda||$ (когда первый аргумент функции равен 0). Одна ячейка в таблице не используется.

В приведенных примерах можно было бы добавить в программы (таблицы) произвольное количество пустых строк. Значит, если существует одна машина Тьюринга для вычисления некоторой функции, то существует бесконечное множество машин для вычисления данной функции.

В качестве упражнения постройте машины Тьюринга для вычисления следующих функций:

1. $f(x) = x + 3$,

2. $f(x) = 2x$,

3. $f(x) = 3x + 2$,

4. $f(x,y) = \begin{cases} 3x, & \text{если } y \geq 2; \\ x+4, & \text{если } y < 2 \end{cases}$

5. $f(x, y) = \max(x, y)$.

Для усиления вычислительных свойств машины Тьюринга (расширения множества T) было предложено несколько модификаций машины. Рассмотрим некоторые из них.

1. Машины Тьюринга с другим внешним алфавитом. В качестве внешнего алфавита предлагалось рассматривать множества $A_1 = \{\lambda, 0, 1\}$ (для записи чисел в двоичной системе счисления), $A_2 = \{\lambda, 0, 1, ..., 9\}$ (для записи чисел в десятичной системе).

2. k-ленточные машины Тьюринга. Для записи исходных данных и проведения промежуточных расчетов предлагалось использовать несколько лент. Соответственно, в ячейки таблицы-программы помещается не одна, а k команд (каждая команда для каждой ленты).

3. Одноленточная машина Тьюринга с несколькими головками. На ленте оперирует несколько головок, каждая из которых действует независимо от других (при этом нужно определить, что делать в случае, если две разные головки обозревают одну и ту же ячейку, и хотят сделать в ней запись).

Было доказано, что любая из модификаций машины Тьюринга моделируется простейшей машиной, рассмотренной в начале данного параграфа. Таким образом, множество функций T является достаточно широким. В данный момент времени не найдено ни одной интуитивно вычислимой функции, для ко-

торой нельзя было бы составить вычисляющую машину Тьюринга. Это позволяет выдвинуть следующий

Тезис Тьюринга: *любая интуитивно вычислимая функция является вычислимой по Тьюрингу*.

Обратите внимание, что данный тезис нельзя доказать, так как он связывает точное понятие вычислимости по Тьюрингу с неточным понятием интуитивно вычислимой функции.

1.5. МАШИНЫ ШЁНФИЛДА

Рассмотрим ещё один способ уточнения интуитивного понятия алгоритма. Это способ принадлежит Джозефу Шёнфилду[1]. Как и Тьюринг, Шёнфилд предложил рассматривать некие вычислительные машины, называемые машинами Шёнфилда. Отличительной особенностью машин Шёнфилда является то, что программирование на них напоминает написание программы на языке ассемблер. Кроме этого, использование машин Шёнфилда позволяет значительно упростить доказательства классических теорем теории алгоритмов.

Машина Шёнфилда имеет бесконечное число регистров, пронумерованных натуральными числами, начиная с 0. Каждый из этих регистров может содержать любое натуральное число. Для любой программы нам будет необходимо лишь конечное число регистров, которое нетрудно заранее определить по программе. Кроме этого у машины есть специальная память для программы, а также счётчик команд, всегда содержащий некоторое натуральное число.

Программа для машины Шёнфилда состоит из конечного списка команд, последовательно пронумерованных натуральными числами начиная с 0.

Перед запуском в машину вводится программа, в регистры заносятся начальные данные, а в счётчик команд заносится значение 0. После этого шаг за шагом осуществляется работа машины.

Шаг машины состоит в исполнении команды, номер которой указан в счётчике команд. Если такого номера команды в программе нет, то машина останавливается.

Существует два типа команд:

- `INC I` – увеличивает содержимое `I`-го регистра на 1 и увеличивает содержимое счётчика команд на 1. После этого машина переходит к следующему шагу;
- `DEC I, n` – если содержимое `I`-го регистра больше нуля, то уменьшает содержимое `I`-го регистра на 1 и заносит `n` в счётчик команд. Если содержимое `I`-го регистра равно 0, то увеличивает содержимое счётчика команд на 1.

Процесс вычисления некой функции $f(x_1, ..., x_n)$ на машине Шёнфилда будет состоять в следующем: i-е значение аргумента функции заносим в `I`-й регистр и запускаем машину; если машина останавливается, то значение функ-

[1] Joseph R. Shoenfild

ции – это значение в регистре 0; в противном случае считаем, что значение функции не определено на данном наборе аргументов.

Определение 1.5.1. (Функция, вычислимая по Шёнфилду) *Арифметическая функция называется вычислимой по Шёнфилду, если существует машина Шёнфилда, которая её вычисляет.*

Множество всех функций, вычислимых по Шёнфилду, обозначим *Sh*. О связи интуитивно вычислимых функций и множества *Sh* выдвигается следующий

Тезис Шёнфилда: *любая интуитивно вычислимая функция является вычислимой по Шёнфилду.*

На практике для написания программ для машины Шёнфилда удобным оказывается использование макросов. Макрос – это некая совокупность команд машины, имеющая короткое обозначение. Достаточно очевидной является следующая

Теорема 1.5.1. (Об элиминации макросов) *Любая программа с макросами эквивалентна некоторой программе, не содержащей макросов.*

При элиминации макросов особого внимания требуют команды вида `DEC I, n`. Данные команды необходимо изменить так, чтобы переход осуществлялся на строку с соответствующим номером в программе без макросов. При элиминации макросов возможно потребуется переименование некоторых регистров (замена номеров регистров, используемых в макросах, новыми).

Опишем несколько полезных макросов.

Макрос `GOTO n`. Данный макрос осуществляет переход на `n`-ю строку программы. Состоит макрос из следующей пары команд

```
0: INC 0
1: DEC 0, n
```

Макрос `ZERO I`. Это макрос, который обнуляет содержимое `I`-го регистра. Состоит такой макрос всего из одной команды – `DEC I, 0`.

Макрос `[`i`]` $\rightarrow$ `[`j`],(`k`)`, где $i, j \neq k$. Этот макрос копирует содержимое i-го регистра в j-й регистр, используя k-й регистр в качестве вспомогательного. Содержимое i-го регистра при $i \neq j$ не изменяется. Программа для макроса `[`i`]` $\rightarrow$ `[`j`],(`k`)` при $i \neq j$ выглядит так:

```
0: ZERO j
1: ZERO k        обнулили j-й и k-й регистры
2: GOTO 5        переход на 5-ю команду
3: INC j
4: INC k
5: DEC i, 3      если в i-м регистре не 0, увеличим i-й и k-й регистры
6: GOTO 8
7: INC i
8: DEC k, 7      копирование из k-го регистра в i-й
```

Если $i = j$, то можно взять любую программу, которая ничего не меняет, например

```
0: GOTO 1
```

Макрос $F([i_1], ...,[i_k]) \rightarrow [j]$ вычисляет значение функции F, вычислимой по Шёнфилду с помощью программы P, от содержимого регистров $[i_1]$, ..., $[i_k]$, записывает это значение в j-й регистр и при этом не изменяет значения остальных регистров.

Приведём примеры некоторых машин Шенфилда. Следующая машина определяет функцию сложения $\text{sum}(x, y) = x + y$.

```
0: [2] → [0]
1: GOTO 3
2: INC 0
3: DEC 1, 2
```

Для функции умножения $\text{mult}(x, y) = x * y$ можно написать такую машину:

```
0: ZERO 0
1: GOTO 3
2: sum([0], [2])→[0]
3: DEC 1, 2
```

В качестве упражнений реализуйте машины Шенфилда для функций, упоминавшихся в упражнениях предыдущего параграфа.

Процесс программирования машин Шенфилда существенно легче, чем аналогичный процесс для машин Тьюринга. Объясняется это тем, что машины Шенфилда представляют собой *вычислительное устройство с произвольным доступом к данным*. В любой момент времени мы можем работать с любым регистром машины. Машины Тьюринга – это пример *устройства с последовательным доступом к данным*, так как чтобы работать с некоторой ячейкой ленты машины Тьюринга нужно последовательно перейти к этой ячейке.

1.6. ЧАСТИЧНО ВЫЧИСЛИМЫЕ ФУНКЦИИ

Частично вычислимые функции[1] – это формализация понятия вычислимости, предложенная в 1935 году А. Чёрчем и С. Клини. В основе данной формализации понятия алгоритма лежит идея представление любой функции через комбинации неких простейших функций.

Будем называть *простейшими* следующие арифметические функции:

1. $0(x) = 0$ – константа ноль;
2. $s(x) = x + 1$ – функция следования;
3. $I_m(x_1, x_2, \ldots, x_n) = x_m$ $(1 \le m \le n)$ – функции выбора аргумента (их n).

Заметим, что любая простейшая функция является всюду определённой и вычислимой (как в интуитивном смысле, так и по Тьюрингу и Шенфилду).

[1] Исходно такие функции назывались частично рекурсивными функциями, но в 1995 году известный американский математик Роберт Соар предложил изменить терминологию, и большинство математиков мира согласилось с его предложением.

Далее рассмотрим следующие операции над функциями.

1. Операция суперпозиции.

Рассмотрим арифметические функции $f_1(x_1, ..., x_n)$, ..., $f_m(x_1, ..., x_n)$ и $\varphi(x_1, ..., x_m)$. Будем говорить, что функция $\psi(x_1, ..., x_n)$ получена из функций $f_1, ..., f_m$ и φ применением операции суперпозиции, если $\psi(x_1, ..., x_n) = \varphi(f_1(x_1, ..., x_n), ..., f_m(x_1, ..., x_n))$.

Очевидно, что в случае, когда функции $f_1, ..., f_m$ и φ являются вычислимыми, то функция ψ также является вычислимой. Функция ψ является определённой на наборе $(a_1, a_2, ..., a_n)$, если на данном наборе определена каждая функция f_i и функция φ является определённой на наборе $(f_1, f_2, ..., f_m)$.

Приведём несколько примеров функций, полученных при помощи операции суперпозиции.

1. $0(x_1, x_2, ..., x_n) = 0(I_m(x_1, x_2, ..., x_n)) = 0$;
2. $c_1(x) = s(0(x)) = 1$, $c_2(x) = s(c_1(x)) = 2$;
3. $c_m(x_1, x_2, ..., x_n) = m$ $(m \geq 1, m \in \mathrm{N})$;
4. $s_2(x) = s(s(x)) = x + 2$, $s_{m,k}(x_1, x_2, ..., x_n) = x_m + \kappa$.

2. Операция примитивной рекурсии.

Опишем простейший случай использования операции примитивной рекурсии. Пусть дано некое натуральное число a и арифметическая функция $h(x, y)$. Определим арифметическую функцию $f(x)$ по следующей схеме:

$$\begin{cases} f(0) = a, \\ f(x+1) = h(x, f(x)). \end{cases}$$

В этом случае говорят, что функция $f(x)$ строится из константы a и функции $h(x, y)$ при помощи оператора примитивной рекурсии.

Легко видеть, что в случае, когда функция $h(x, y)$ является вычислимой, функция $f(x)$ также является вычислимой. Значения функции $f(x)$ можно получить последовательно: $f(0) = a$, $f(1) = h(0, f(0))$, $f(2) = h(1, f(1))$ и т. д.

Рассмотрим несколько примеров функции, которые могут быть получены с использованием операции примитивной рекурсии.

1. $sgn(x) = \begin{cases} 0, & x = 0, \\ 1, & x \neq 0. \end{cases}$

Опишем данную функцию следующим образом:

$$\begin{cases} \mathrm{sgn}(0) = 0, \\ \mathrm{sgn}(x+1) = c_1(x, \mathrm{sgn}(x)). \end{cases}$$

2. $\overline{\mathrm{sgn}}(x) = \begin{cases} 1, & x = 0, \\ 0, & x \neq 0. \end{cases}$

Данную функцию можно задать так:

$$\begin{cases} \overline{\text{sgn}}(0) = 1, \\ \overline{\text{sgn}}(x+1) = 0(x, \overline{\text{sgn}}(x)). \end{cases}$$

Рассмотрим теперь более общий случай операции примитивной рекурсии. Пусть даны арифметические функции $g(x_1, ..., x_n)$ и $h(x_1, ..., x_n, y, z)$. Определим арифметическую функцию $f(x_1, ..., x_n, y)$ по следующей схеме:

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

В этом случае говорят, что функция $f(x_1, ..., x_n, y)$ строится из функций $g(x_1, ..., x_n)$ и $h(x_1, ..., x_n, y, z)$, при помощи оператора примитивной рекурсии.

Рассмотрим примеры.

1. Пусть дана следующая схема:

$$\begin{cases} f(x, 0) = I_1(x), \\ f(x, y+1) = s_{3,1}(x, y, f(x, y)). \end{cases}$$

Очевидно, что данная схема определяет функцию $f(x, y)$ как $x + y$. Далее для этой функции будем использовать обозначение $\text{sum}(x, y) = x + y$.

2. Рассмотрим схему

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

Данная схема определяет функцию $f(x, y)$ как $x * y$.

Две рассмотренных операции над простейшими функциями позволяют описать достаточно большое подмножество арифметических функций.

Определение 1.6.1. (Примитивно рекурсивная функция) *Арифметическая функция называется примитивно рекурсивной, если она может быть получена из простейших за конечное число применений операций суперпозиции и примитивной рекурсии.*

Множество всех примитивно рекурсивных функций обозначим как *Pr*. Отметим важное свойство примитивно рекурсивных функций – они являются всюду определёнными.

В качестве упражнения покажите, что следующие функции являются примитивно рекурсивными.

1. $power(x, y) = x^y$;
2. $x - 1 = \begin{cases} x - 1, \text{ если } x > 0 \\ 0, \quad \text{если } x = 0; \end{cases}$
3. $x - y = \begin{cases} x - 1, \text{ если } x \geq y \\ 0, \quad \text{если } x = 0; \end{cases}$
4. $|x - y|$.

Класс примитивно рекурсивных функций является достаточно широким. Однако существуют вычислимые функции, которые не принадлежат этому классу. Рассмотрим, например, функцию Аккермана[1]:

$$\begin{cases} A(0,y) = y+1, \\ A(x,0) = A(x-1,1), \\ A(x,y) = A(x-1, A(x,y-1)). \end{cases}$$

Данная функция задана при помощи рекурсии (не примитивной рекурсии!) и является всюду определённой и интуитивно вычислимой (у нас имеется способ нахождения значений этой функции). Функция Аккермана не является примитивно рекурсивной. Доказательство этого факта опирается на следующую идею: примитивно рекурсивные функции не могут быстро расти, а функция Аккермана растёт очень быстро (проверьте!).

Для построения развитой теории частично вычислимых функций рассматривается ещё одна операция над функциями.

3. Операция минимизации.

Будем говорить, что функция $f(x_1, ..., x_n)$ получается из функции $g(x_1, ...,x_n, y)$ при помощи операции минимизации (μ-оператора), и обозначать

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

если выполнено условие: $f(x_1, ..., x_n)$ определена и равна y тогда и только тогда, когда $g(x_1, ...,x_n, 0)$, ..., $g(x_1, ...,x_n, y\text{-}1)$ определены и не равны 0, а $g(x_1, ...,x_n, y) = 0$. Заметим, что процесс вычисления функции $f(x_1, ..., x_n)$ сводится к последовательному перебору значений y.

Рассмотрим пример. Пусть $g(x, y, z) = |(y + z) - x|$ и пусть $f(x, y) = \mu z[g(x,y,z) = 0]$. Легко убедиться, что $f(x, y) = x - y$. Заметим, что эта функция не является всюду определённой.

Определение 1.6.2. (Частично вычислимая функция) *Арифметическая функция называется частично вычислимой, если она может быть получена из простейших за конечное число применений операций суперпозиции, примитивной рекурсии и минимизации.*

Множество всех частично вычислимых функций обозначим Pc.

Определение 1.6.3 (Общерекурсивная функция) *Арифметическая функция называется общерекурсивной, если она является частично вычислимой и всюду определённой.*

Множество всех общерекурсивных функций обозначим $C\nu$.

Непосредственно из приведённых определений следует, что $Pr \subseteq C\nu \subseteq Pc$. Нетрудно установить, что данные включения являются строгими: $Pr \subset C\nu \subset Pc$. Таким образом, существуют общерекурсивные функции, для получения кото-

[1] Ackermann W.

рых *необходимо* использовать операцию минимизации (функция Аккермана, например).

Как и в случае машин Тьюринга и Шенфилда, на данный момент времени не найдено ни одной интуитивно вычислимой функции, которая не принадлежала бы множеству *Pc*. Это позволяет выдвинуть следующий

Тезис Чёрча: *любая интуитивно вычислимая функция является частично вычислимой.*

1.7. КОДИРОВАНИЕ АЛГОРИТМОВ

В теории алгоритмов важное значение имеет свойство алгоритма быть представленным неким натуральным числом. При этом исчезает принципиальная разница между командами и данными, что позволяет рассматривать алгоритмы как данные для других алгоритмов.

Существует несколько способов кодирования алгоритмов. Опишем один из них. Вначале рассмотрим построение кодов для последовательностей натуральных чисел $x_1, x_2, ..., x_n$.

Определение 1.7.1. (Код последовательности) *Пусть дана некая конечная последовательность натуральных чисел $x_1, x_2, ..., x_n$. Кодом этой последовательности, обозначаемым далее как $\langle x_1, x_2, ..., x_n \rangle$, назовём число $p_1^{x_1+1} p_2^{x_2+1} ... p_n^{x_n+1}$, где p_i – i-е простое число. Таким образом*

$$\langle x_1, x_2, ..., x_n \rangle = p_1^{x_1+1} p_2^{x_2+1} ... p_n^{x_n+1}.$$

Отметим следующие особенности данного кодирования последовательностей. Не все натуральные числа являются кодами последовательностей (например, таковым не будет число $2^4 3^0 5^2$). Однако, разным последовательностям натуральных чисел всегда будут соответствовать разные коды. Функции, которые позволяют узнать, является ли число кодом некоторой последовательности, закодировать данную последовательность чисел, узнать по коду составляющие последовательность числа, являются частично вычислимыми.

Пусть теперь дан алгоритм. Этому алгоритму соответствует некая интуитивно вычислимая функция, а ей (согласно тезису Шёнфилда) соответствует некая (в общем случае не единственная) машина Шёнфилда.

Закодируем машины Шёнфилда. Кодом команды `INC I` будем считать число $\langle 0, I \rangle$, кодом команды `DEC I, n` – число $\langle 1, I, n \rangle$. Кодом программы из n команд будем считать следующее число

$$e = \langle \text{ код 1-й команды, код 2-й команды,..., код n-й команды } \rangle.$$

Таким образом, мы получили отображение множества всех машин Шёнфилда во множество натуральных чисел.

В теории алгоритмов важной является теорема, утверждающая эквивалентность математических уточнений понятия алгоритма.

Теорема 1.7.1. *Множества T, Pc и Sh совпадают.*

Используя данную теорему, мы можем при решении теоретических задач рассматривать то из уточнений понятия алгоритма, которое является наиболее выгодным в конкретном случае. Доказательство данной теоремы является достаточно технически сложным. Опишем идею данного доказательства для случая $Pc = Sh$. Чтобы доказать приведённое равенство, необходимо показать, что $Pc \subset Sh$ и $Sh \subset Pc$. Первое включение доказывается сравнительно просто: для этого достаточно описать машины Шёнфилда, вычисляющие простейшие функции, и описать способ реализации на машинах Шёнфилда операций суперпозиции, примитивной рекурсии и минимизации.

Для доказательства включения $Sh \subset Pc$ необходимо построить по каждой машине Шёнфилда некую частично вычислимую функцию. Эту функцию можно построить таким образом, что она оказывается общей для всего класса машин Шёнфилда, вычисляющих функции от n переменных. То есть построенная функция U будет зависеть не только от начальных значений $x_1, x_2, ..., x_n$, но и от кода машины e:

$$U = U(e, x_1, x_2, ..., x_n).$$

К рассмотрению подобных функций, зависящих от (кодов) алгоритмов, мы вернёмся позже.

1.8. АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ЗАДАЧИ

Целью данного параграфа будет доказательство существования алгоритмически неразрешимых проблем, то есть таких массовых задач, для решения которых не существует универсального способа.

Теорема 1.8.1. *Существуют алгоритмически неразрешимые задачи.*

Доказательство. Для доказательства данной теоремы проведём оценку мощности множества арифметических функций и множества вычислимых функций.

Рассмотрим множество характеристических функций, то есть множество функций вида

$$\chi_P(\bar{x}) = \begin{cases} 1, & \text{если} \quad \bar{x} \in P \\ 0, & \text{если} \quad \bar{x} \notin P \end{cases}$$

где $P \subseteq \mathrm{N}^k$. Каждая характеристическая функция является арифметической. Характеристических функций столько, сколько подмножеств у N^k. Мощность множества подмножеств N^k – континуум ($|\, 2^{\mathrm{N}^k} | = c$). Таким образом, мощность множества всех арифметических функций не менее, чем континуум.

Рассмотрим множество интуитивно вычислимых функций. Каждой вычислимой функции соответствует некая машина Шёнфилда. Каждую машину Шёнфилда можно закодировать натуральным числом. Следовательно, множество интуитивных функции не более чем счётно (имеет мощность $\aleph_0$).

Так как $\aleph_0 < c$, то получаем утверждение теоремы. □

Данное доказательство не является конструктивным, так как не содержит построения алгоритмически неразрешимой задачи. Приведём пример такой задачи.

Рассмотрим множество машин Шёнфилда, вычисляющих функции одного аргумента. Назовем машину Шёнфилда *самоопределённой*, если она выдаёт некое значение, будучи запущенной на своём собственном коде, и *несамоопределённой* в противном случае (машина зацикливается на своём коде).

Утверждение 1.8.1. *Задача определения типа машины Шёнфилда по её коду является алгоритмически неразрешимой.*

Доказательство. Допустим противоположное. Пусть данная задача является алгоритмически разрешимой. Тогда существует машина Шёнфилда *S*, которая запускается на коде некой машины и выдаёт в нулевом регистре 0, если это код несамоопределённой машины, и 1, если это код самоопределённой машины.

Построим машину S_1. Для этого дополним машину *S* следующими командами:

```
m+1: DEC 0, m+3
m+2: GOTO m+4
m+3: GOTO m+3
```

Данные команды останавливают машину S_1, когда в регистре 0 содержится 0, и зацикливают машину, если в нулевом регистре содержится 1.

Запустим машину S_1 на собственном коде. Если S_1 – самоопределенная, то она должна зациклиться, то есть она несамоопределенная. Аналогично, машина S_1 не может принадлежать и классу несамоопределенных машин. Таким образом, получили противоречие, которое доказывает алгоритмическую неразрешимость задачи определения класса машины Шёнфилда. □

Естественно, число алгоритмически неразрешимых задач не ограничивается приведённой. К таким задачам относится, например, задача определения класса (противоречивая, выполнимая, логический закон) произвольной формулы логики предикатов, что было доказано А. Чёрчем в 1936 году.

1.9. УНИВЕРСАЛЬНЫЕ ФУНКЦИИ

Предложенные уточнения понятия алгоритма – машины Тьюринга, частично вычислимые функции, машины Шёнфилда – обладают одно особенностью. Одна конкретная машина или функция реализует единственный алгоритм. В то же время, современные компьютеры представляют собой устройства, которые могут реализовывать множество алгоритмов. Оказывается, можно описать машину или частично вычислимую функцию для моделирования совокупности машин или функций. Такая машина при решении некоторой задачи в качестве одного из параметров может получать числовой код алгоритма решения и моделировать алгоритм по его коду. По сути, такая машина является неким *транслятором алгоритмов*.

Дадим строгое математическое определение описанным понятиям.

Определение 1.9.1. (Универсальная функция) *Пусть дано некое множество арифметических функций M от n аргументов. Функция* $U(n, x_1,...,x_n)$ *называется универсальной функцией для множества M, если выполняются два условия:*

1. для любого фиксированного n_0 *функция* $U(n_0, x_1,...,x_n)$ *от n аргументов принадлежит множеству M;*

2. для любой функции $f(x_1,...,x_n) \in M$ *существует число* n_0 *(возможно, не единственное), такое что* $f(x_1,...,x_n) \equiv U(n_0, x_1,...,x_n)$.

Универсальная функция ведёт себя как современная ЭВМ, в качестве программы которой выступает значение первого аргумента функции.

Лемма 1.9.1. *Для множества частично вычислимых функций от n аргументов существует универсальная функция.*

Доказательство. Доказательство данной леммы основано на доказательстве включения $Sh \subset Pc$. Напомним, что для этого строилась одна функция $U(e, x_1,...,x_n)$, которая зависела от кода машины *e*. Функция *U* по построению была частично вычислимой. Для любой частично вычислимой функции $f(x_1,...,x_n)$ можно указать некое n_0, являющееся кодом машины Шёнфилда, которая вычисляет *f*. При этом, по построению функции *U* будет выполняться условие $f(x_1,...,x_n) \equiv U(n_0, x_1,...,x_n)$. □

Покажем теперь, что не для всех множеств арифметических функций можно построить универсальную функцию.

Лемма 1.9.2. *Для множества общерекурсивных функций от n аргументов универсальной функции не существует.*

Доказательство. Рассмотрим множество общерекурсивных функций одного аргумента, которое обозначим как $C\upsilon_1$. Предположим, что $U(n, x)$ – универсальная функция для этого множества. Рассмотрим функцию $g(x) = U(x, x) + 1$. Так как $U(n, x)$ – общерекурсивная функция, то и функция $g(x)$ является общерекурсивной. Следовательно, существует число $n_0 : g(x) = U(n_0, x)$. Однако,

$$U(n_0, n_0) = g(n_0) = (\text{по определению } g) = U(n_0, n_0) + 1.$$

Получили противоречие. Следовательно, для множества $C\upsilon_1$ универсальной функции не существует. Случаи, когда количество аргументов общерекурсивной функции больше единицы, рассматриваются аналогично. □

1.10. НЕКОТОРЫЕ ТЕОРЕМЫ ТЕОРИИ АЛГОРИТМОВ

Опишем в виде теорем некоторые фундаментальные результаты теории алгоритмов.

Многие частично вычислимые функции не являются всюду определёнными. Например, функция $f(x,y)=x-y$ определена, только если $x \geq y$. Отдельные частично вычислимые функции можно доопределить так, чтобы они стали общерекурсивными.

Определение 1.10.1. (Доопределение) *Пусть дана некая функция* $f(x_1,...,x_n) \in Pc$. *Доопределением данной функции назовём общерекурсивную функцию* $\widetilde{f}(x_1,...,x_n) \in C\upsilon$ *такую, что*

$$\widetilde{f}(x_1,...,x_n)=\begin{cases} f(x_1,...,x_n), & \text{если } f(x_1,...,x_n) \text{ определено,} \\ m \in N, & \text{если } f(x_1,...,x_n) \text{ не определено.} \end{cases}$$

Доопределением функции $f(x,y)=x-y$ может служить функция

$$\widetilde{f}(x,y)=\begin{cases} x-y, & \text{если } x \geq y, \\ 0, & \text{если } x < y. \end{cases}$$

Оказывается, справедлива следующая теорема.

Теорема 1.10.1. (О доопределении) *Существует частично вычислимая функция, которую нельзя доопрделить.*

Доказательство. Рассмотрим универсальную функцию $U(n,x)$ для множества частично вычислимых функций одного аргумента. Пусть $d(x)=\overline{\operatorname{sgn}}U(x,x)$. Допустим, существует доопределение функции $d(x)$, которое обозначим как $\widetilde{d}(x)$. Функция $d(x)$ является общерекурсивной, а следовательно и частично вычислимой. Значит, существует номер n_0, такой что $\widetilde{d}(x)=U(n_0,x)$. Значение $\widetilde{d}(n_0)$ определено, значит определено и значение $U(n_0,n_0)$. Следовательно, определено значение $d(n_0)=\overline{\operatorname{sgn}}U(n_0,n_0)$. Можно записать

$$\overline{\operatorname{sgn}}U(n_0,n_0)=d(n_0)=\widetilde{d}(n_0)=U(n_0,n_0)$$

Но равенство $\overline{\operatorname{sgn}}U(n_0,n_0)=U(n_0,n_0)$ не выполняется ни для каких натуральных чисел. Следовательно, предположение о существовании доопределения функции $d(x)$ было ложным. □

Опишем случай, когда доопределение частично вычислимой функции возможно.

Определение 1.10.2. (Рекурсивное множество) *Пусть M – некое подмножество* N^n. *Назовём множество M рекурсивным, если характеристическая функция этого множества*

$$\chi_M(x_1,...,x_n)=\begin{cases}1, & \text{если } (x_1,...,x_n)\in M,\\ 0, & \text{если } (x_1,...,x_n)\notin M,\end{cases}$$

является общерекурсивной.

Тот факт, что множество *M* является рекурсивным, означает наличие алгоритма, который по любому натуральному числу (набору натуральных чисел) может определить, принадлежит ли это число (набор) множеству *M*. Примерами рекурсивных множеств являются: конечное множество, множество чётных чисел. Приведём следующую теорему без доказательства.

Теорема 1.10.2. *Если областью определения частично вычислимой функции является рекурсивное множество, то существует доопределение данной функции.*

Рассмотрим частично вычислимую функцию от двух аргументов $f(x,y)$. Этой функции соответствует некое натуральное n_0, такое что выполняется условие $f(x,y)=U(n_0,x,y)$, где U – универсальная функция для частично вычислимых функций двух аргументов. Напомним, что U можно воспринимать в роли своеобразного «компьютера», выполняющего программы машин Шёнфилда. Опишем следующее преобразование произвольной программы *S*, вычисляющей значение функции от двух аргументов:

1. Скопировать регистр 1 в регистр 2;
2. Обнулить регистр 1, увеличить регистр 1 на единицу *x* раз;
3. Выполнить программу *S*.

Само описываемое преобразование – перенумерация команд исходной машины *S*, добавление в начало программы новых команд – алгоритмично. Значит, ему соответствует некая вычислимая функция τ. Эта функция τ зависит от аргумента *x* и от исходной программы *S* (номера этой программы): $\tau=\tau(n,x)$. Также заметим, что функция τ является общерекурсивной. Результат вычисления функции $U(n_0,x,y)$ на данных *x* и *y* совпадает с результатом вычисления универсальной функции двух аргументов $\tilde{U}(\tau(n_0,x),y)$ при данном *y*, если к программе n_0 применить преобразование τ, зависящее от *x*.

Приведённые рассуждения справедливы для любого n_0. Значит,

$$U(n,x,y)\equiv\tilde{U}(\tau(n_0,x),y) \qquad (10.1)$$

Имеет место более общее утверждение, называемое

Лемма 1.10.1. (Лемма о параметризации) *Для универсальной функции множества частично вычислимых функций n аргументов* $U(n,x_1,...,x_m,x_{m+1},...,x_n)$ *существует общерекурсивная функция* $\tau(n,x_1,...,x_m)$, *такая что*

$$U(n,x_1,...,x_m,x_{m+1},...,x_n)\equiv\tilde{U}(\tau(n,x_1,...,x_m),x_{m+1},...,x_n),$$

где $\widetilde{U}$ – универсальная функция множества частично вычислимых функций n-m аргументов.

Рассмотрим теорему, играющую центральную роль в теории алгоритмов.

Теорема 1.10.3. (Теорема Клини о неподвижной точке) *Пусть $U(n,x)$ – универсальная функция для множества частично вычислимых функций одного аргумента, $f(x)$ – произвольная общерекурсивная функция. Тогда существует число n_0 такое, что*

$$U(n_0,x) \equiv U(f(n_0),x).$$

Доказательство. Рассмотрим следующую вспомогательную функцию:

$$g(x,y) = U(f(\tau(x,x))y),$$

где τ – функция из равенства (10.1). Функция *g* является частично вычислимой функцией от двух аргументов. Следовательно, если $U_2(m,x,y)$– универсальная функция для множества частично вычислимых функций двух аргументов, то существует номер *m*, такой что

$$g(x,y) = U_2(m,x,y).$$

По лемме о параметризации можно записать $U_2(m,x,y) = U(\tau(m,x),y)$. Учитывая определение функции *g*(*x*, *y*), получаем равенство

$$U(\tau(m,x),y) = U(f(\tau(x,x)),y)).$$

Положив в данном равенстве *x* = *m* и обозначив для краткости $\tau(m,m) = n_0$, получим утверждение теоремы. □

Рассмотрим следствие из теоремы Клини.

Следствие 1.10.1. *Пусть $U(n,x)$ – универсальная функция для множества частично вычислимых функций одного аргумента. Тогда существует число p, что для любого x*

$$U(p,x) = p.$$

Доказательство. Рассмотрим множество всех частично вычислимых функций, являющихся константами. Определим следующее отображение:

$\varphi(l)$ = минимальное *n*, при котором $U(n,x)$ задаёт функцию-константу, равную *l*. Функция $\varphi(l)$ является общерекурсивной. Значит, по теореме Клини существует *p*:

$$U(p,x) \equiv U(\varphi(p),x).$$

По смыслу определения функции φ, $U(\varphi(p),x) = p$. Отсюда и получается требуемое утверждение. □

Данное следствие имеет следующую интересную интерпретацию: существует программа, которая при любых входных данных печатает свой текст. В качестве упражнения предлагается написать такие программы.

Рассмотрим теорему, которая позволяет получить доказательство алгоритмической неразрешимости многих проблем.

Теорема 1.10.4. (Теорема Раиса) *Пусть F – некое множество частично вычислимых функций одного аргумента, причём существуют функции, принадлежащие F, и функции, не принадлежащие F. Тогда множество номеров всех функций из F не является рекурсивным.*

Доказательство. Обозначим через N_F множество номеров всех функций из F. Предположим, что N_F является рекурсивным множеством. Это означает, что его характеристическая функция χ_{N_F} является общерекурсивной. Если N_F – рекурсивное множество, то и $\overline{N}_F$ – рекурсивное множество, так как $\chi_{\overline{N}_F}(x) = 1 - \chi_{N_F}(x)$. По определению множества F ни множество N_F, ни множество $\overline{N}_F$ не является пустым.

Определим следующую функцию:

$$g(x) = \begin{cases} a \in \overline{N}_F, & \text{если } x \in N_F, \\ b \in N_F, & \text{если } x \in \overline{N}_F. \end{cases}$$

Функция $g(x)$ является общерекурсивной, так как, фактически, $g(x) = a * \chi_{N_F} + b * \chi_{\overline{N}_F}$.

По теореме о неподвижной точке существует n:

$$U(n,x) = U(g(n),x).$$

Так как $N_F \cup \overline{N}_F$, то $n \in N_F$ или $n \in \overline{N}_F$. Оба случая являются невозможными.

Пусть, к примеру, $n \in N_F$. Тогда $U(n,x)$ определяет функцию, которая принадлежит F. Значит, и $U(g(n),x) \in F$. Но при $n \in N_F$ имеем $g(n) = a$, $a \notin N_F$, а значит и $U(g(n),x) \notin F$. Получили противоречие. Единственное предположение теоремы – это предположение о рекурсивности набора N_F. Оно не верно. □

Теорема Раиса допускает следующую интерпретацию: не существует программы, которая может проверять некие нетривиальные свойства *всех* программ. Например, нельзя написать программу, которая бы по тексту любой программы на Паскале устанавливала, зацикливается ли программа или нет. В качестве упражнения опишите, как использовать теорему Раиса для доказательства отсутствия такой программы.

2. ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ

2.1. СЛОЖНОСТЬ АЛГОРИТМОВ

Предметом классической теории алгоритмов является уточнение понятия алгоритма и установление алгоритмической разрешимости массовых проблем. Однако существование алгоритма для задачи ещё не означает её практическую разрешимость. В качестве примера рассмотрим задачу *коммивояжёра*: имеется n городов, соединённых сетью дорог, требуется посетить все города по маршруту наименьшей протяжённости. Тривиальный алгоритм решения задачи состоит в переборе всех возможных маршрутов. Если граф, представляющий города и соединяющие их дороги, является полным, то необходимо рассмотреть $(n-1)!$ различных вариантов. Уже при малых n данная задача не может быть решена, так как количество вариантов становится астрономически большим. В связи с этим возникает необходимость введения некоторых мер сложности алгоритмов для поиска среди возможных алгоритмов решения массовой проблемы оптимального.

Попытаемся описать сложность какого-либо определённого алгоритма. Вообще говоря, оценка сложности алгоритма зависит от *вычислительной модели*, которая используется для описания алгоритма. В качестве вариантов *меры сложности* алгоритма, возможно рассматривать такие величины, как быстродействие (время работы) алгоритма, количество используемой алгоритмом памяти и другие величины.

Из рассмотренных нами вычислительных моделей – машины Тьюринга, машины Шёнфилда, частично вычислимые функции – для дальнейшего изложения фиксируем в качестве основной модели машины Шёнфилда. Будем анализировать временную сложность вычислений на машине Шёнфилда. Пусть M – машина Шёнфилда (программа), которая соответствует некоторому алгоритму (реализует алгоритм), $x_1x_2...x_n$ – слово (число), которое кодирует входные данные программы. Обозначим через $T(M, x_1x_2... x_n)$ количество шагов машины M, которые требуются для того, чтобы выполнить программу на входных данных $x_1x_2... x_n$.

Функция T является неудобной для изучения, так как в общем случае она устроена достаточно неоднородно. Определим на основе функции T новую функцию $t_M(n)$:

$$t_M(n) = \max T(M,x).$$

Здесь X – множество всевозможных наборов входных данных алгоритма, причём длина наборов (количество элементов) не превосходит n. Фактически, функция $t_M(n)$ – функция от длины входных данных алгоритма. Эта функция характеризует сложность алгоритма в наихудшем случае.

Итак, мы получили возможность вычислять временную сложность алгоритма, реализуемого на машине Шёнфилда. Можно сказать, что мы определили некую функцию, которая по алгоритму получает функцию $t_M(n)$.

Пусть теперь мы пытаемся установить сложность некой массовой проблемы. Очевидный подход заключается в следующем: если для решения массовой проблемы существует несколько алгоритмов, то сложность массовой проблемы – это сложность самого хорошего (быстрого) алгоритма для её решения. Как оказывается, данных подход применим не всегда.

Теорема 2.1.1. (Блюм, об ускорении, 1971) *Существует такая вычислимая функция f, что любую машину M, вычисляющую f, можно ускорить, то есть построить машину* M_1*, также вычисляющую f, и при этом*

$$t_{M_1}(n) \le \log_2 t_M(n)$$

для почти всех n.

Смысл теоремы Блюм: имеются такие массовые проблемы, алгоритмы решения которых допускают бесконечное улучшение.

В теории сложности принят подход, при котором анализ сложности массовой проблемы проводится на основе установления принадлежности данной проблемы к особым классам, называемым классами сложности.

Определение 2.1.1. (Класс сложности) *Пусть C – некоторое выделенное множество функций* $t_M(n)$*. Классом сложности A назовём множество всех массовых проблем, для решения каждой из которых существует такой алгоритм (машина) M, что функция* $t_M(n)$ *принадлежит C.*

Рассмотрим и обсудим следующий пример. Определим множество:

DTIME($t(n)$) = {массовая проблема a | $\exists$ машина M для a и $t_M(n) = O(t(n))$}.

Для произвольной функции натурального аргумента $t(n)$ при помощи DTIME($t(n)$) можно образовать класс сложности, состоящий из массовых проблем, для решения которых существуют алгоритмы со временем работы, сравнимым с $t(n)$.

Замечание. Напомним, что запись $f(x) = O(g(x))$ означает следующее:

$$f(x) = O(g(x)) \Leftrightarrow \lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| = k > 0$$

Обоснованием подхода, при котором функции сложности рассматриваются с точностью до мультипликативной константы, может служить тот факт, что реальная скорость выполнения программы на компьютере определяется не только количеством шагов алгоритма, но и длительностью элементарного шага (временем выполнения машинной команды).

Следующей целью данной главы будет рассмотрения конкретных классов вычислительной сложности и взаимосвязи между ними.

2.2. КЛАССЫ СЛОЖНОСТИ P И EXP

Определим следующий важный класс сложности P:

$$\mathrm{P} = \bigcup_{k \geq 0} \mathrm{DTIME}(n^k).$$

Класс сложности P определяет массовые проблемы, для решения которых существуют алгоритмы, со временем работы, сравнимым с некоторым полиномом.

Класс сложности P обладает одной очень важной особенностью: он замкнут относительно операции суперпозиции. Пусть у нас имеются две массовые проблемы A и B и пусть $A \in \mathrm{P}$. Если удалось построить алгоритм сведения проблемы B к проблеме A, время работы которого ограничено полиномом, то можно утверждать, что $B \in \mathrm{P}$.

Все рассмотренных нами ранее вычислительные модели (машины Тьюринга, машины Шёнфилда, частично вычислимые функции) преобразуются друг в друга при помощи алгоритмов с полиномиальной сложностью. Отсюда следует возможность менее формального изложения материала, при которой в дальнейшем будут рассматриваться реализации алгоритмов не на машинах Шёнфилда, а в неком обобщённом виде, более привычном для программистов.

Рассмотрим пример. Пусть требуется умножить две произвольные матрицы размером $n \times n$. Тривиальный алгоритм решения данной массовой проблемы будет содержать участок кода, подобный этому:

```
for i := 1 to n do
  for j   := 1 to n do
    for k   := 1 to n do
      c[i,j]   := c[i,j] + a[i,k] * b[k,j] ;
```

Легко установить, что внутренний оператор циклов будет выполняться n^3 раз. Следовательно, общее время работы данного алгоритма сравнимо с n^3. Значит, массовая проблема умножения матриц принадлежит классу P.

Опишем ещё один важный класс сложности EXP:

$$\mathrm{EXP} = \bigcup_{k \geq 0} \mathrm{DTIME}(2^{n^k}).$$

Класс сложности EXP определяет массовые проблемы, для решения которых существуют алгоритмы со временем работы, сравнимым с некой экспоненциальной функцией.

Обратите внимание: если задана некоторая массовая проблема A из P, то данная массовая проблема принадлежит и EXP. Иными словами, $\mathrm{P} \subseteq \mathrm{EXP}$. Установление данного факта тривиально. Достаточно взять любой алгоритм для решения массовой проблемы A и дополнить его следующим циклом, который ничего не выполняет:

```
for i := 1 to 2^n do
  begin end;
```

Верно ли обратное включение EXP $\subset$ P и, следовательно, равенство P = EXP? Иными словами, можно ли для решения любой массовой проблемы из EXP предложит полиномиальный алгоритм? Ответ на данный вопрос даёт следующая теорема.

Теорема 2.2.1. (Хартманис, 1965) *Существуют массовые проблемы, для решения которых невозможно предложить алгоритм из класса сложности P.*

Из теоремы Хартманиса следует, что P $\subset$ EXP, то есть классы P и EXP существенно различаются.

2.3. КЛАСС СЛОЖНОСТИ NP

Рассмотрим следующую модификацию машины Шёнфилда. Добавим к машине специальный регистр с номером -1 и особый *угадывающий модуль*. Работа модифицированной машины будет происходить следующим образом. В любой момент времени машина может перейти в *недетерминированное состояние*. В этом состоянии угадывающий модуль машины записывает в регистр -1 некое случайное число из диапазона 0..*k*. Далее машина Шёнфилда переходит в нормальное состояние и продолжает свою работу. Естественно, эта работа может зависеть от содержимого регистра -1.

Недетерминированные машины Шёнфилда позволяют ввести новый класс сложности вычислений.

Определение 2.3.1. (Класс сложности NP) *Пусть для массовой проблемы существует алгоритм, время работы которого на недетерминированной машине Шёнфилда ограничено полиномом. В этом случае, будем говорить, что массовая проблема принадлежит классу сложности NP.*

В качестве примера массовой проблемы из класса сложности NP рассмотрим упоминавшуюся задачу коммивояжёра. Для её решения можно предложить недетерминированную машину Шёнфилда, угадывающий модуль которой записывает в регистр -1 число, кодирующее некий маршрут (последовательность городов). В детерминированном состоянии машина Шёнфилда декодирует по числу маршрут, вычисляет его длину и сравнивает с неким эталонным (кратчайшим) маршрутом. Работа машины в детерминированном состоянии ограничена полиномом. Следовательно, задача коммивояжёра принадлежит классу сложности NP.

Если предложить менее формальное определение класса сложности NP, то можно сказать, что к этому классу относятся проблемы, которые решаются перебором возможных ответов. Машина пытается угадать ответ задачи. Для этого она переходит в недетерминированное состояние, генерирует некое число, а затем за полиномиальное время проверяет, является ли это число ответом задачи или нет.

Из определения классов сложности P и NP следует включение P $\subseteq$ NP. Является ли это включение строгим? На данный момент (2006 год) этот факт не доказан. Данная проблема известна как проблема P=NP?

Теорема 2.3.1. *Любая недетерминированная машина Шёнфилда моделируется некоторой детерминированной машиной.*

Для моделирования недетерминированной машины Шёнфилда на детерминированной машине известны алгоритмы, временная сложность которых экспоненциальная. Следовательно, имеет место утверждение

$$P \subseteq NP \subseteq EXP$$

Так как $P \subset EXP$, то одно из включений в приведенном соотношении является строгим.

2.4. СВОДИМОСТЬ. NP – ПОЛНЫЕ ЗАДАЧИ

Выше было указано, что класс P является замкнутым относительно операции суперпозиции. Формализуем приведённые рассуждения при помощи понятия сводимости.

Определение 2.4.1. (Полиномиальная сводимость) *Пусть A и B – две массовые проблемы. Будем говорить, что проблема A полиноминально сводится к проблеме B и обозначать это как* $A \propto B$, *если существует алгоритм с полиномиальной сложностью, который позволяет свести любую индивидуальную задачу из A к индивидуальной задаче из B с возможностью соответствующей интерпретации ответов.*

Понятие полиномиальной сводимости позволяет ввести понятие NP-полных задач.

Определение 2.4.2. (NP-полная задача) *Массовая проблема A называется NP-полной задачей, если выполняются два условия:*

1. $A \in NP$;

2. для любой проблемы $B \in NP$ *справедливо* $B \propto A$.

Если для какой-либо NP-полной задачи существует полиномиальный разрешающий алгоритм, то и для любой задачи из класса сложности NP существует полиномиальный разрешающий алгоритм. Если какая-то NP-полная задача не лежит в классе P, то и все NP-полные задачи не лежат в классе P.

Рассмотрим следующую массовую проблему. Пусть булева функция $f(x_1, ..., x_n)$ задана своей КНФ. Требуется установить, является ли данная булева функция выполнимой. Исторически NP-полнота данной проблемы была доказана первой.

Теорема 2.4.1. (Стефан А. Кук, 1971) *Проблема проверки выполнимости произвольной КНФ является NP-полной.*

В настоящее время список NP-полных задач состоит из нескольких тысяч задач. Под классификацию NP-полной попадает практически любая задача, где решение ищется перебором возможных вариантов.

2.5. АНАЛИЗ СЛОЖНОСТИ РЕКУРСИВНЫХ АЛГОРИТМОВ

Рекурсия является достаточно общим алгоритмическим приёмом. Данный приём заключается в решении исходной задачи путём разбиения её на несколько сходных подзадач. Использование рекурсии часто позволяет достаточно просто представить и записать алгоритмы. Для многих практически важных задач лучшие оценки сложности дают алгоритмы, использующие рекурсию. Рассмотрим несколько примеров.

1. Сортировка чисел.

Рассмотрим последовательность натуральных чисел $x_1, x_2, ..., x_n$, которую необходимо упорядочить по возрастанию. Очевидный алгоритм решения данной задачи состоит в нахождении минимального члена исходной последовательности, постановки его на первое место в результирующей последовательности и повторении данного шага для оставшейся неотсортированной последовательности. Легко установить, что данный алгоритм требует $O(n^2)$ попарных сравнений в худшем случае.

Предложим для задачи сортировки рекурсивный алгоритм. Пусть $n = 2^k$. При k=1 алгоритм упорядочивает последовательность одним сравнением. Пусть алгоритм определён для некоторого k. Тогда при $k + 1$ алгоритм работает так:

1. Последовательность $x_1, x_2, ..., x_{2^{k+1}}$ разбивается на две подпоследовательности: $x_1, x_2, ..., x_{2^k}$ и $x_{2^k+1}, x_{2^k+2}, ..., x_{2^{k+1}}$.

2. К обеим подпоследовательностям длинны 2^k применяется построенный алгоритм. Получаем две упорядоченные последовательности: $x'_1, x'_2, ..., x'_{2^k}$ и $x'_{2^k+1}, x'_{2^k+2}, ..., x'_{2^k+1}$.

3. Осуществляется слияние двух упорядоченных последовательностей сравнением их левых элементов x'_1 и x'_{2^k+1} и помещением наименьшего в начало результирующей последовательности.

Если длина исходной последовательности n не равна 2^k, последовательность дополняется нулями (или, напротив, большими числами), чтобы её длина стала степенью двойки.

Пусть $T(n)$ – число попарных сравнений, используемых в данном рекурсивном алгоритме для произвольной последовательности длины n. Тогда получаем соотношение

$$\begin{cases} T(n) = 2T(n/2) + n - 1, \\ T(2) = 1. \end{cases} \tag{5.2}$$

Утверждение 2.5.1. *Для рекуррентного соотношения (5.2) справедлива формула*

$$T(2^k) = 2^k k - 2^k + 1.$$

Проверка данного утверждения производится индукцией по k.

Для произвольного n справедливо:

$$T(n) = O(n \log n).$$

Таким образом, можно утверждать, что приведённый рекурсивный алгоритм сортировки лучше по порядку исходного «наивного» алгоритма.

2. Возведение в степень.

Пусть зафиксирован элемент a некоторой алгебраической системы с операцией умножения[1] и натуральное число n. Требуется найти a^n. Тривиальный алгоритм требует для этого $n - 1$ умножение.

Рассмотрим рекурсивный алгоритм умножения. Пусть $u(n) = a^n$. Если $n = 1$, то $u(1) = a$. Допустим, что $n > 1$. Вычислим числа $k = [n/2]$ и $l = n \bmod 2$, где операции $[x]$ и mod обозначают соответственно деление нацело и остаток от деления. Тогда справедливо

$$\begin{cases} u(n) = u(n/2) \cdot u(n/2), & \text{если } l = 0 \\ u(n) = u([n/2]) \cdot u([n/2]) \cdot a, & \text{если } l = 1. \end{cases}$$

Нетрудно убедиться, что в данном алгоритме используется не более $2[\log_2 n]$ умножений.

3. Умножение матриц.

Рассмотрим задачу нахождения произведения двух матриц одинакового размера $n \times n$. Пусть $n = 2$ и пусть

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \; B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \; C = AB = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

Используем стандартный способ нахождения матрицы C:

$$c_{11} = a_{11}b_{11}+a_{12}b_{21}, \quad c_{12} = a_{11}b_{12}+a_{12}b_{22}, \quad c_{21} = a_{21}b_{11}+a_{22}b_{21}, \quad c_{22} = a_{21}b_{12}+a_{22}b_{22}.$$

Данный способ требует 8 умножений и 4 сложения. На первый взгляд кажется невозможным улучшить этот результат. Однако Штрассеном было предложено вычислять элементы матрицы C по следующей схеме. Вначале находим промежуточные значения:

$$p_1 = (a_{11} + a_{22})(b_{11} + b_{22}), \quad p_2 = (a_{21}+ a_{22})b_{11}, \quad p_3 = a_{11}(b_{12} - b_{22}), \quad p_4 = a_{22}(b_{21} - b_{11}),$$

$$p_5 = (a_{11} + a_{12})b_{22}, \quad p_6 = (a_{21} - a_{11})(b_{11} + b_{12}), \quad p_7 = (a_{12} - a_{22})(b_{21} + b_{22}).$$

Затем вычисляем элементы матрицы C:

$$c_{11} = p_1+p_4 - p_5+p_7, \quad c_{12}=p_3+p_5, \quad c_{21}=p_2+p_4, \quad c_{22} = p_1 - p_2 + p_3+p_6.$$

Алгоритм Штрассена требует 7 умножений и 18 сложений.

Пусть теперь $n = 2^{k+1}$. Тогда алгоритм Штрассена работает так: матрицы A и B размера $2^{k+1} \times 2^{k+1}$ представляются как 2×2-матрицы над кольцом матриц

[1] Подобная общая формулировка позволяет применять алгоритм не только к числам, а и, например, к матрицам.

порядка $2^k \times 2^k$ и применяется изложенный выше алгоритм умножения матриц 2×2. Если $n \neq 2^{k+1}$, то находят такое k, что $2^k < n < 2^{k+1}$, и к матрицам добавляется нужное число нулевых строк и столбцов.

Пусть $T(2^k)$ – число арифметических операций, используемых в алгоритме Штрассена на матрицах размера $2^{k+1} \times 2^{k+1}$. Тогда справедливо соотношение

$$\begin{cases} T(2^{k+1}) = 7T(2^k) + 18(2^k)^2, \\ T(2^0) = 1. \end{cases} \tag{5.3}$$

Утверждение 2.5.2. *Для соотношения (5.3) справедлива формула*

$$T(2^k) = 7^{k+1} - 6*2^{2k}.$$

Доказательство данного утверждения проводится по индукции.

Ясно, что выполняется

$$T(n) = O(7^{\log_2 n}) = O((2^{\log_2 7})^{\log_2 n}) = O(n^{\log_2 7}).$$

Так как $\log_2 7 \approx 2{,}807$, то алгоритм Штрассена лучше по порядку обычного алгоритма умножения матриц.

Усилиями ряда математиков (Пан, Виноград, Романи, Копперсмит) экспоненту сложности матричного умножения удалось уменьшить до $\approx 2{,}376$. Однако на данный момент не найдено алгоритма (и не доказано отсутствие такового) для экспоненты сложности, равной 2.

4. Решение рекуррентных соотношений, соответствующих рекурсивным алгоритмам.

При анализе временной сложности рекурсивных алгоритмов возникает потребность в решении рекуррентных соотношений специального вида. Рассмотрим несколько видов таких соотношений и опишем подходы к их решению.

Пусть задано рекуррентное соотношение следующего вида

$$\begin{cases} T(n) = aT(\frac{n}{k}) + f(n), \\ T(1) = c. \end{cases} \tag{5.4}$$

Здесь a, k, c – некоторые константы, $f(n)$ – заданная функция. Запишем несколько значений искомой функции $T(n)$:

$$\begin{aligned} &T(1) = c, \\ &T(k) = aT(1) + f(k) = ac + f(k), \\ &T(k^2) = aT(k) + f(k^2) = a^2c + af(k) + f(k^2), \\ &T(k^3) = aT(k^2) + f(k^3) = a^3c + a^2f(k) + af(k^2) + f(k^3), \\ &\qquad \cdots \\ &T(k^p) = a^pc + \underbrace{a^{p-1}f(k) + a^{p-2}f(k^2) + \ldots + f(k^p)}_{p \text{ слагаемых}}. \end{aligned}$$

Обратите внимание, что рекуррентное соотношение (5.4) позволяет однозначно вычислить значение функции $T(n)$ только при n, являющихся степенями числа k. Вопрос о решении рекуррентного соотношения (5.4) сводится к вопросу о возможности компактной записи выражения для суммы S

$$S = a^{p-1}f(k) + a^{p-2}f(k^2) + ... + f(k^p)$$

Рассмотрим один из случаев, когда это возможно. Пусть $f(n) = bn^t$, где b,t – некоторые константы. В этом случае сумма S имеет следующий вид:

$$S = b(a^{p-1}k^t + a^{p-2}k^{2t} + ... + k^{pt}).$$

Выражение в скобках является геометрической прогрессией. Первым членом этой прогрессии будем считать k^{pt}, тогда последний член – $a^{p-1}k^t$, знаменатель прогрессии – $\frac{a}{k^t}$.

Если $\frac{a}{k^t} = 1$ и, соответственно, а = k^t, то все члены прогрессии одинаковы и равны k^{pt}. Тогда $S = bpk^{pt} = bpa^p$, а $T(k^p) = a^pc + bpa^p = a^p(c + bp)$.

Если $\frac{a}{k^t} \neq 1$, то по формуле для суммы геометрической прогрессии

$$S = b \cdot \frac{k^{pt}\left(\left(\frac{a}{k^t}\right)^p - 1\right)}{\frac{a}{k^t} - 1} = \frac{bk^t(a^p - k^{pt})}{a - k^t}.$$

Значит, в этом случае получаем $T(k^p) = a^pc + \frac{bk^t(a^p - k^{pt})}{a - k^t}$. Таким образом, получаем выражение $T(k^p)$:

$$T(k^p) = \begin{cases} a^p(c + bp), & \text{если } a = k^t \\ a^pc + \frac{bk^t(a^p - k^{pt})}{a - k^t}, & \text{если } a \neq k^t. \end{cases}$$

При необходимости можно получить выражение для $T(n)$, выполнив замену $k^p = n$. Подытожим приведённые выкладки в виде леммы.

Лемма 2.5.1. *Решением рекуррентного соотношения*

$$\begin{cases} T(n) = aT(\frac{n}{k}) + bn^t, \\ T(1) = c. \end{cases}$$

где a, b, c, k, t – некоторые константы, является функция

$$T(n)=\begin{cases} n^{\log_k a}(c+b\log_k n), & \text{если } a=k^t \\ n^{\log_k a}c+\dfrac{bk^t(n^{\log_k a}-n^t)}{a-k^t}, & \text{если } a\neq k^t. \end{cases}$$

Приведём некоторые другие рекуррентные соотношения, возникающие при анализе рекурсивных алгоритмов.

Если рекурсия вызывает аддитивное уменьшение сложности задачи, то используется соотношение

$$\begin{cases} T(n)=aT(n-k)+f(n), \\ T(0)=c. \end{cases}$$

Иногда рекурсия ведёт к «квадратичному» уменьшению сложности. В этом случае возникает соотношение

$$\begin{cases} T(n)=a\sqrt{n}T(\sqrt{n})+f(n), \\ T(2)=c. \end{cases}$$

Для решения приведенных соотношений в некоторых частных случаях применимы соображения, использовавшиеся при решении соотношения (5.4).

2.6. ТОЧНАЯ ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ В НАИЛУЧШЕМ, НАИХУДШЕМ И СРЕДНЕМ СЛУЧАЕ

Полученные нами ранее оценки сложности для алгоритмов являлись достаточно грубыми. Как правило, мы ограничивались установлением функции сложности «с точность до О-большое». В данном и следующих параграфах основной целью будет получение точной оценки времени выполнения для некоторых конкретных алгоритмов.

Рассмотри следующую задачу: дан массив из N чисел, требуется найти минимальное число в массиве. Для решения этой задачи можно предложить следующий алгоритм, записанный в форме, близкой к языку Pascal:

```
min := M[1];                          {1}
for i := 2 to N do                    {2}
  if M[i] < min then                  {3}
                  min := M[i];        {4}
```

Попробуем подсчитать число шагов выполнения этого алгоритма, причём сделать это как можно точнее. Первый оператор выполняется один раз. Оператор проверки условия цикла `{2}` выполнится $N-1$ раз (в действительности, в начале цикла выполняется инициализация, так что можно увеличить общее число операторов на единицу). Проверка условия – оператор `{3}` – выполняется при каждой итерации цикла, то есть $N-1$ раз. Особого внимания заслуживает оператор `{4}`. Количество выполнений этого оператора зависит от данных исходной задачи и не выражается в явной форме. Как правило, в большинстве алгоритмов встречаются операторы, количество выполнений которых зависит от исходных данных. В связи с этим при точной оценке сложности алгоритма

принят подход, согласно которому вместо всех возможных наборов входных данных проводится оценка сложности в трёх случаях – наилучшем, наихудшем и среднем.

Вернёмся к нашему алгоритму. Что такое «наилучший» случай для данного алгоритма? Очевидно, таковым будет случай когда минимальный элемент является первым элементом в исходном массиве. В этом случае количество выполнений оператора `{4}` будет равно нулю. Наихудший случай соответствует ситуации, когда исходный массив отсортирован по убыванию. Тогда оператор `{4}` будет выполнятся при каждой итерации цикла, то есть $N - 1$ раз.

Рассмотрим «средний» случай. Для вывод оценки количества выполнений оператора `{4}` сделаем следующее предположения о природе элементов исходного массива: будем считать, что все элементы массива различны[1]. При первой итерации цикла вероятность выполнения оператора `{4}` равна $\frac{1}{2}$, так как такова вероятность того, что добавляемый к «куче» чисел (из одного элемента) новый элемент окажется в этой «куче» наименьшим. То есть можно сказать, что при первой итерации цикла оператор `{4}` «выполнится» $\frac{1}{2}$ раз. При второй итерации цикла вероятность выполнения оператора `{4}` равна $\frac{1}{3}$: «куча» состоит из двух элементов, мы добавляем третий, вероятность того, что он самый маленький в «куче» – $\frac{1}{3}$. Продолжая рассуждения, получим, что в среднем случае количество о выполнений оператора `{4}` выражается формулой

$$S = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{N}. \qquad (6.7)$$

Рассмотрим следующую сумму, играющую важную роль при анализе алгоритмов:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.$$

Данная сумма представляет собой частичную сумму гармонического ряда $(1, \frac{1}{2}, \frac{1}{3}, ...)$, являющегося расходящимся. Известна следующая оценка для H_n:

$$H_n = \ln n + \gamma + \frac{1}{2n} + o(\frac{1}{n}), \qquad (6.8)$$

где $\gamma = 0{,}5772156649...$ – это *постоянная Эйлера.*

Используя данную оценку, мы можем принять выражение для суммы (6.7):

[1] Без этого предположения вывод оценки становится затруднительным.

$$S = H_N - 1 \approx \ln N - 0{,}42 \approx \ln N .$$

Полученная оценка является весьма точной (с точностью до аддитивной константы). Таким образом, для количества выполнений оператора {4} получены оценки: $B(N) = 0$, $W(N) = N - 1$, $A(N) = \ln N$ (здесь и далее: $B(N)$ – наилучший случай, $W(N)$ – наихудший случай, $A(N)$ – средний случай). Если это необходимо, теперь легко можно подсчитать общее число выполнения операторов в нашем алгоритме. Естественно, требуется рассмотрение трёх случаев. Кроме этого, иногда для отдельных операторов вводят так называемые *стоимости выполнения*, характеризующие время выполнения отдельного оператора. Пусть, например, стоимости выполнения операторов нашего алгоритма равны c_1, c_2, c_3, c_4 соответственно. Тогда алгоритм выполняется за время

- $c_1 + (N-1)c_2 + (N-1)c_3 + 0*c_4 = c_1 + (N-1)(c_2 + c_3)$ в наилучшем случае;
- $c_1 + (N-1)c_2 + (N-1)c_3 + c_4 \ln N = c_1 + (N-1)(c_2 + c_3) + c_4 \ln N$ в среднем случае;
- $c_1 + (N-1)c_2 + (N-1)c_3 + (N-1)c_4 = c_1 + (N-1)(c_2 + c_3 + c_4)$ в наихудшем случае.

Приведённый пример показывает, что общая задача точного анализа алгоритмов является достаточно нетривиальной. При её выполнении активно используется математический аппарат, в частности, теория вероятностей. В дальнейшем рассмотрим задачу анализа на некоторых конкретных примерах алгоритмов.

2.7. АЛГОРИТМЫ СОРТИРОВКИ И ИХ АНАЛИЗ

1. Алгоритм быстрой сортировки: описание.

Задача сортировки – одна из наиболее часто встречающихся при обработке данных. В параграфе, посвященном рекурсивным алгоритмам, был предложен алгоритм сортировки, имеющий сложность $O(N \log_2 N)$ по числу сравнений. Рассмотрим ещё один алгоритм сортировки и проведём более точную оценку его сложности.

Идея алгоритма *быстрой сортировки* предложена Ч. Хоаром и заключается в следующем. Пусть требуется отсортировать по возрастанию массив *M,* содержащий *N* чисел. Выберем элемент массива *M,* называемый *осевым* и переупорядочим массив так, чтобы все элементы, расположенные левее осевого, были меньше его, а элементы, расположенные правее осевого элемента, были больше его. В левой и правой части (относительно осевого элемента) массива элементы не упорядочиваются. Затем алгоритм быстрой сортировки применяется к левой и правой части массива (т. е. вызывается рекурсивно).

Запись алгоритма быстрой сортировки может выглядеть следующим образом:

```
{предположение: M - глобальный массив}
{сортируем от индекса first до last}
procedure QuickSort(first, last)
begin
```

```
  if first < last then begin
    {находим позицию осевого элемента}
    pivot := FindPivot(first, last)
    QuickSort(first, pivot-1);        {сортируем левую часть}
    QuickSort(pivot + 1, last)        {сортируем правую часть}
  end
end;
```

В данной записи использовалась функция `FindPivot` для нахождения позиции осевого элемента. Она допускает несколько вариантов реализации. Рассмотрим один из простейших:

```
function FindPivot(first, last)
begin
  pivot_value := M[first];
  pivot := first;
  for i := first + 1 to last do
    if M[i] < pivot_value then begin      {Comp}
      inc(pivot);
      M[i] <-> M[pivot]                   {Swap}
    end;
  M[first] <-> M[pivot];
  result   := pivot
end;
```

Логика работы данного кода заключается в следующем. За значение осевого элемента принимается значение первого элемента просматриваемой части массива. Затем выполняется проход по массиву. Если обнаружен элемент, меньший осевого, то указатель осевого элемента `pivot` увеличивается на единицу, и найденный элемент переставляется с элемент с новым номером `pivot`. После очередной итерации цикла в массиве можно выделить четыре части. Одна состоит из первого – осевого – элемента массива. Вторую часть составляют элементы с индексами от `first` до `pivot` – это элементы, меньшие осевого. Третья часть массива – элементы с индексами от `pivot+1` до `i` – элементы, большие осевого. Наконец, четвёртая часть – это непросмотренные элементы массива.

2. Алгоритм быстрой сортировки: анализ среднего случая.

После обсуждения алгоритма быстрой сортировки проведём его анализ. Будем анализировать число сравнений элементов сортируемого массива (оператор `{Comp}`).

При вызове функции `FindPivot` для участка массива длины N она выполняет $N - 1$ сравнение. Таким образом, общее число сравнений в алгоритме быстрой сортировки зависит от того, сколько раз вызывается функция `FindPivot` и для участков какой длины она вызывается.

Начнём анализ алгоритма быстрой сортировки со среднего случая. Пусть $A(N)$ – количество сравнений в среднем для массива длины N. Так как быстрая сортировка является рекурсивным алгоритмом, найдём рекуррент-

ное соотношение для $A(N)$. Очевидно, что $A(0)=0$, $A(1)=0$. Пусть при очередном вызове функция `FindPivot` вернула значение *P*. Тогда

$$A(N)=(N-1)+A(P-1)+A(N-P) \text{ для } N\geq 2.$$

Число *P* может с равной вероятностью принять любое значение от 1 до *N*. Таким образом, среднее значение $A(N)$ для $N\geq 2$ выражается как

$$A(N)=(N-1)+\frac{1}{N}\sum_{i=1}^{N}A(i-1)+A(N-i)).$$

В записанной сумме аргумент первого слагаемого пробегает значения от 0 до *N* – 1, а аргумент второго слагаемого – те же значения, но в обратном порядке. Следовательно,

$$A(N)=(N-1)+\frac{2}{N}\sum_{i=0}^{N-1}A(i) \qquad \text{для } N\geq 2, \tag{7.9}$$

$$A(1)=A(0)=0$$

Для упрощения рекуррентного соотношения (7.9) избавимся от дроби $\frac{1}{N}$, умножая соотношение на *N*. Кроме этого, запишем явно выражение для $A(N-1)$:

$$A(N)\cdot N=(N-1)N+2\sum_{i=0}^{N-1}A(i),$$

$$A(N)\cdot N=(N-1)N+2A(N-1)+2\sum_{i=0}^{N-2}A(i), \tag{7.11}$$

$$A(N-1)\cdot(N-1)=(N-2)(N-1)+2\sum_{i=0}^{N-2}A(i). \tag{7.12}$$

Вычитая из равенства (7.11) равенство (7.12), получим

$$A(N)\cdot N-A(N-1)\cdot(N-1)=(N-1)N+2A(N-1)-(N-2)(N-1)=$$
$$=2A(N-1)+2N-2.$$

Это позволяет записать окончательное рекуррентное соотношение, не содержащее суммы:

$$A(N)=\frac{(N+1)A(N-1)+2N-2}{N}. \tag{7.13}$$

Будем решать рекуррентное соотношение (7.13). Разделим обе части соотношения на *N+1:*

$$\frac{A(N)}{N+1}=\frac{(N+1)A(N-1)+2N-2}{N(N+1)}=\frac{A(N-1)}{N}+\frac{2N-2}{N(N+1)}.$$

Сделаем замену $D(N)=\frac{A(N)}{N+1}$ (очевидно, $D(1)=\frac{A(1)}{2}=0$):

$$D(N)=D(N-1)+\frac{2N-2}{N(N+1)}.$$

Для функции $D(N)$ легко получить выражение через сумму:

$$D(N)=\sum_{k=1}^{N}\frac{2k-2}{k(k+1)}=2\left[\sum_{k=1}^{N}\frac{k-1}{k(k+1)}\right]=2\left[\sum_{k=1}^{N}\frac{1}{k+1}-\sum_{k=1}^{N}\frac{1}{k(k+1)}\right]=$$

$$=2\left[\sum_{k=1}^{N}\frac{1}{k+1}-\sum_{k=1}^{N}\frac{1}{k}-\frac{1}{k+1}\right]=2\left[H_{N+1}-1-\left(1-\frac{1}{N+1}\right)\right].$$

Теперь запишем выражение для $A(N)$*:*

$$A(N)=(N+1)D(N)=2(N+1)H_{N+1}-4(N+1)+2\approx 2(N+1)\ln(N+1)-(4-2\gamma)N$$

Сознательно огрубив оценку для $A(N)$, можем записать

$$A(N)\approx 2(N+1)\ln(N+1).$$

Рассмотрение среднего случая для числа сравнений закончено.

3. Асимптотически оптимальный алгоритм сортировки.

Прежде чем выполнять оценку алгоритма быстрой сортировки в наилучшем и наихудшем случае, проанализируем вопрос: насколько оптимальным является данный алгоритм по числу сравнений?

Пусть имеется некий алгоритм, выполняющий сортировку последовательности чисел $x_1, x_2, ..., x_N$ путём сравнения элементов последовательности. Этому алгоритму можно поставить в соответствие *дерево сортировки,* имеющее следующую структуру. Узлы дерева – точки разветвления – содержат сравнение неких элементов последовательности x_i и x_j. Листья дерева – концевые точки – представляют собой перестановку исходной последовательности, соответствующую отсортированной последовательности. Количество сравнений, которое должен выполнить алгоритм в конкретном случае равняется количеству рёбер, ведущих от корня дерева сортировки к некоторому его листу. Лучшему случаю соответствует самый короткий путь от корня к листу, худшему – самый длинный, а для подсчёта среднего случая надо разделить суммарную длину всех путей от корня к листьям на количество листьев.

Количество возможных листьев дерева сортировки равно *N*!, где *N* – число элементов сортируемой последовательности. Пусть *K* – определённый уровень дерева сортировки, считая от корня (корень – это нулевой уровень).

Тогда количество узлов на *K*-ом уровне равно 2^K. В случае, когда дерево сортировки является идеально сбалансированным, выполняется неравенство

$$2^{K+1} \geq N!$$

Логарифмируя данное неравенство и использовав *формулой Стирлинга* $N! \approx \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$, получим следующую оценку для числа *K:*

$$K > (N + \frac{1}{2}) \log_2 N - 1{,}4427N.$$

Таким образом, нижняя оценка числа сравнений для любого алгоритма сортировки, использующего сравнения, удовлетворяет асимптотике $O(N \log_2 N)$. Следовательно, рассмотренный алгоритм быстрой сортировки является асимптотически оптимальным по числу сравнений. Заметим, что ранее, в параграфе посвященным рекурсивным алгоритмам, был рассмотрен алгоритм сортировки слиянием с оценкой сложности $O(N \log_2 N)$. Значит, этот алгоритм также является асимптотически оптимальным.

Используя идею дерева сортировки, легко понять, какие ситуации при быстрой сортировке соответствуют наилучшему и наихудшему случаю. Очевидно, что наименьшее число сравнений будет достигаться, когда в результате очередного вызова `FindPivot` осевой элемент оказывается ровно в середине сортируемого участка. Таким образом,

$$B(N) = (N-1) + B\left(\frac{N}{2}\right) + B\left(\frac{N}{2}\right).$$

Приходим к рекуррентному соотношению

$$\begin{cases} B(N) = (N-1) + 2B\left(\frac{N}{2}\right), \\ B(2) = 1. \end{cases}$$

Его решение было рассмотрено в параграфе, посвященном рекурсивным алгоритмам:

$$B(N) = N \log_2 N - N + 1.$$

Наихудший случай для числа сравнений получается, если каждый вызов `FindPivot` разбивает массив на максимально неравные участки, то есть когда осевой элемент оказывается либо на первой, либо на последней позиции. В этом случае выполняется рекуррентное соотношение

$$\begin{cases} W(N) = (N-1) + W(N-1) + W(1), \\ W(1) = 0. \end{cases}$$

Решение данного соотношения выражается формулой

$$W(N) = \frac{N(N-1)}{2}.$$

Таким образом, в худшем случае алгоритм быстрой сортировки отнюдь не является «быстрым». Он работает с таким же порядком скорости, как, например, алгоритм пузырьковой сортировки.

4. Сортировка подсчетом.

Выше было доказано, что любой алгоритм сортировки, основанный на сравнении элементов сортируемой последовательности, в лучшем случае выполняет $O(N \log_2 n)$ сравнений, а, значит, и работает не менее $O(N \log_2 n)$ шагов.

Рассмотрим алгоритм, называемый *сортировка подсчётом,* который не использует сравнения элементов и в определённых ситуациях позволяет выполнять сортировку массива длинны *N* за время, сравнимое с $O(N)$.

Пусть *M* – сортируемый массив длины *N*. Предположим, что элементами массива *M* являются целые числа, лежащие в диапазоне от 1 до *K*. Заведём специальный массив подсчётов *C* из *K* элементов. Перед началом сортировки обнулим элементы массива *C*. Сделаем первый проход по массиву *M*, подсчитывая количество различных элементов и запоминая результат в массиве *C*. После подсчёта элементов заполним массив *M,* используя информацию из массива C.

Приведённый алгоритм иллюстрируется следующей программой:

```
procedure CountSort
begin
  for i := 1 to K do
    C[i] := 0;              {1}
  for i := 1 to N do
    inc(C[M[i]]);           {2}
       j := 1;
  for i := 1 to K do
    for t := 1 to C[i] do begin
      M[j] := i;            {3}
      inc(j);               {4}
    end;
end;
```

Проведём анализ данной программы. Оператор `{1}` выполняется *K* раз, оператор `{2}` – *N* раз. Легко понять, что и операторы, отмеченные как `{3}` и `{4}`, в сумме выполнятся ровно *N* раз. Таким образом, время работы алгоритма можно оценить как $O(N+K)$. Следовательно, сортировка подсчётом выполняется за *линейное* (относительно числа элементов исходного массива) время. Платой за подобную «резвость» является наличие дополнительного массива и необходимость операций с ним. Если, например, сортируется массив из 100 чисел, каждое из которых имеет тип `integer`, потребуется дополнительный массив из 4294967296 элементов. В этом случае ни о каком

преимуществе по скорости (а тем более по используемой памяти) перед традиционными алгоритмами сортировки говорить не приходиться.

2.8. АЛГОРИТМЫ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ

1. Вычисление значения многочлена в точке.

Работа с многочленами – традиционная задача вычислительной математики. Она интересна сама по себе, а также как составная часть более сложных задач. Напомним некоторые определения.

Определение 2.8.1. *Пусть зафиксировано некое поле F. Выражение*

$$A(x) = a_0 + a_1 x + a_2 x^2 + ...a_{n-1} x^{n-1} = \sum_{j=0}^{n-1} a_j x^j ,$$

где $a_j, x \in F$*, будем называть многочленом от переменной x над полем F степени не выше n – 1 (если* $a_{n-1} \neq 0$ *то говорят, что степень многочлена равна n – 1).*

В дальнейшем будем считать, что многочлены рассматриваются над полем **C** комплексных чисел . Из определения следует, что многочлен однозначно задаётся набором $(a_0, a_1, ..., a_{n-1})$ своих *коэффициентов.*

Рассмотрим следующую задачу: требуется по известному многочлену $A(x)$ и значению переменной x_0 вычислить значение многочлена $y = A(x_0)$.

Приведём «наивный» алгоритм решения данной задачи:

```
y := a[0];
for j := 1 to n-1 do
  y := y + a[j]*x^j;      {x^j означает возведение в степень j}
```

Нетрудно установить, что в данном алгоритме выполняется $A(n) = n - 1$ сложений и $M(n) = 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$ умножения. Также очевидно, что данный алгоритм не является оптимальным. Рассмотрим следующую его модификацию, которая позволяет значительно сократить число умножений:

```
y := a[0];
p := x;
for j := 1 to n-1 do
  begin
    y := y + a[j]*p;
    p := p*x
  end;
```

В приведённой программе на каждой итерации цикла выполняется одно сложение и два умножения. Следовательно, общее количество сложений $A(n) = n - 1$, а количество умножений $M(n) = 2(n-1)$.

Результат предыдущего алгоритма по числу умножений также можно улучшить. Запишем многочлен $A(x)$ в следующем виде

$$A(x) = a_0 + x(a_1 + x(a_2 + ...x(a_{n-2} + xa_{n-1})...).$$

Начнём вычисление значения многочлена с внутренних скобок, постепенно продвигаясь «наружу». На каждом шаге требуется оно умножение и одно сложение:

```
y := 0;
for j := n-1 downto 0 do
  y := y*x + a[j]
```

Вычисление значения многочлена подобным образом называется *вычислением по схеме Горнера.* Такой алгоритм требует $A(n) = n$ сложений и $M(n) = n$ умножений.

Алгоритм вычисления по схеме Горнера также допускает улучшение. Оно достигается путём предварительной обработки исходного многочлена. Данную обработку можно выполнить как вручную, так и реализовать в виде подпрограммы. Рассмотрим алгоритм обработки подробнее.

Предположим, что

1. исходный многочлен является *унимодальным,* то есть коэффициент при старшей степени многочлена равен 1;

2. степень многочлена *n*–1 на единицу меньше некоторой степени двойки ($n-1 = 2^k - 1,\ n = 2^k$) [1]. Пусть многочлен $A(x)$ удовлетворяет данным предположения. Запишем $A(x)$ в следующем виде:

$$A(x) = (x^j + b)B(x) + C(x).$$

Здесь показатель степени *j* равен 2^{k-1}, а число *b* равно $a_{j-1} - 1$. Подобный выбор показателя степени и константы *b* приводит к тому, что многочлены $B(x)$ и $C(x)$ будут унимодальными и иметь степень $2^{k-1} - 1$. Описанное преобразование затем применяется к многочленам $B(x)$ и $C(x)$.

Приведём пример подобного преобразования многочлена:

$$A(x) = x^7 + 4x^6 - 8x^4 + 6x^3 + 9x^2 + 2x - 3,$$

$$A(x) = (x^4 + 5)(x^3 + 4x^2 + 8) + (x^3 - 11x^2 + 2x - 37),$$

$$A(x) = (x^4 + 5)\left[(x^2 - 1)(x + 4) + (x + 12)\right] + \left[(x^2 + 1)(x - 11) + (x - 26)\right].$$

Подсчитаем, сколько действий требуется для вычисления значения преобразованного многочлена $A(x)$. Количество сложений равно 10, а количест-

[1] В некоторых случаях для использования приводимого далее способа вычисления значения многочлена в точке выгодно преобразовать многочлен (добавить нулевые коэффициенты и одночлен старшей степени) для выполнения данных условий.

во умножений – 3. Но, кроме этого, необходимо ещё два умножения: одно для того, чтобы получить x^2 из x, а другое – для получения x^4 из x^2. Следовательно, всего необходимо 5 умножений.

Нетрудно установить (составив соответствующие рекуррентные соотношения), что в общем случае количество умножений выражается числом $M(N) = \frac{N}{2} + \log_2 N$, а количество сложений – числом $A(N) = \frac{3N-1}{2}$, где N – степень многочлена.

Подытожим приведенные способы вычисления значения многочленов в точке x_0 в виде таблицы (N – степень многочлена):

Таблица 2

Способы вычисления значения многочленов

Способ	**Умножения**	**Сложения**
Модифицированный «наивный»	$2N$	N
Схема Горнера	$N+1$	$N+1$
Предварительная обработка	$\frac{N}{2} + \log_2 N$	$\frac{3N-1}{2}$

2. Умножение многочленов.

Рассмотрим задачу умножения двух многочленов. Пусть $A(x) = \sum_{j=0}^{n-1} a_j x^j$ и $B(x) = \sum_{j=0}^{n-1} b_j x^j$ – многочлены степени $n-1$. Их произведением будет многочлен степени $2n-2$. Коэффициенты многочлена $C(x)$ можно вычислить по простой формуле:

$$c_j = \sum_{j=0}^{n-1} a_k b_{j-k} .$$

Использование данной формулы предполагает, что векторы коэффициентов многочленов $A(x)$ и $B(x)$, изначально имеющие длину n, дополнены нулевыми элементами до длины $2n-1$ ($a_i = b_i = 0$ для $i = n,...,2n-1$).

Запишем алгоритм для вычисления произведения многочленов:

```
for j:= 0 to 2*n - 2 do
  begin
    c[j] := 0;
    for к := 0 to j do
      c[j] := c[j] + a[k]*b[j-k]
end;
```

Данный «наивный» алгоритм требует $1+2+...+2n-1 = n(2n-1) = O(n^2)$ умножений. Попытаемся улучшить этот показатель.

Прежде чем описать модифицированный алгоритм умножения многочленов напомним некоторые сведения из алгебры. Пусть $A(x)$ – произвольный многочлен степени n – 1. Рассмотрим следующий набор из n точек:

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}.$$

Если в данном наборе все значения x_i различны, то он однозначно определяет некий многочлен степени не выше n – 1. В свою очередь, многочлен $A(x)$ также однозначно определяет некий набор точек, если положить $y_i = A(x_i)$.

Очевидно, что переход от коэффициентов многочлена $A(x)$ к набору его значений требует времени не более, чем $O(n^2)$: при помощи схемы Горнера значение многочлена считается в одной точке за время $O(n)$, а общее число точек равно n.

Получение коэффициентов многочлена по набору его значений в точках *(интерполяция)* может быть выполнено по *формуле Лагранжа*

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}.$$

Решение этой задачи также требует времени $O(n^2)$.

Как связаны рассмотренные вопросы с задачей умножения многочленов? Дело в том, что умножение двух многочленов можно выполнить следующим образом. Вначале вычисляются значения многочлена $A(x)$ степени n – 1 в $2n$ – 1 точках $x_0, x_1, ..., x_{2n-2}$. Получается набор

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{2n-2}, y_{2n-2})\}.$$

Затем вычисляется значение многочлена $B(x)$ степени n – 1 в тех же точках $x_0, x_1, ..., x_{2n-2}$ и получается набор

$$\{(x_0, z_0), (x_1, z_1), ..., (x_{2n-2}, z_{2n-2})\}.$$

После этого вторые компоненты каждой пары набора перемножаются (это требует $O(n)$ операций):

$$\{(x_0, y_0 z_0), (x_1, y_1 z_1), ..., (x_{2n-2}, y_{2n-2} z_{2n-2})\}$$

Этот набор описывает значения многочлена $C(x) = A(x) \cdot B(x)$ в точках $x_0, x_1, ..., x_{2n-2}$. Для получения коэффициентов многочлена $C(x)$ достаточно по набору точек провести интерполяцию.

Количество операций, необходимое для получения произведения многочленов, можно уменьшить, если научиться эффективно вычислять значения многочленов и производить интерпретацию в определённом наборе точек.

Алгоритм, который будет рассмотрен далее, строит набор значений многочлена в точках, являющихся комплексными корнями из единицы.

Определение 2.8.2. *Комплексное число* $\omega \in C$ *называется корнем n-ой степени из единицы, если выполняется равенство*

$$\omega^n = 1.$$

Значение $\omega_n = e^{\frac{2\pi i}{n}}$ *называется главным корнем n-ой степени из единицы.*

Корни *i*-ой степени из единицы обладают следующими свойствами, приводимыми без доказательств:

1. $\omega_n^k = \omega_{dn}^{dk}$.
2. Если *n* – чётное число, то $\omega_n^{n/2} = \omega_2 = -1$;
3. Если *к* не кратно *n,* то $\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$;
4. Если *n* – чётное число, то, возведя в квадрат все *n* корней *n* -ой степени из единицы, мы получим $\frac{n}{2}$ корней $\frac{n}{2}$-ой степени из единицы, причём каждый корень будет получен два раза.

Определение 2.8.3. *Пусть* $A(x)$ *– многочлен,* $a = (a_0, a_1, ..., a_{n-1})$ *– вектор его коэффициентов. Вектор* $y = (y_0, y_1, ..., y_{n-1})$*, элементы которого вычисляются по формуле*

$$y_k = A(\omega_n^k),$$

где ω_n *– главный корень n-ой степени из единицы, называется дискретным преобразованием Фурье вектора a.*

В дальнейшем для дискретного преобразования Фурье вектора *a* будет использоваться запись

$$y = DFT_n(a).$$

Использовав описанную выше идею вычисления произведения многочленов $A(x)$ и $B(x)$, можно записать, что

$$c = DFT_{2n-1}^{-1}(DFT_{sn-1}(a) \cdot DFT_{sn-1}(b)).$$

В данной формуле *a, b, c* – это векторы коэффициентов многочленов $A(x)$, $B(x)$ и $C(x)$ соответственно, DFT^{-1} – преобразование, обратное дискретному преобразованию Фурье, точка обозначает поэлементное умножение двух векторов. Таким образом, для эффективного умножения многочленов требуется эффективный алгоритм, выполняющий дискретное преобразование Фурье.

Опишем алгоритм, называемый *быстрым преобразованием Фурье.* Данный алгоритм применим, если длина вектора, подвергаемого преобразованию, является степенью двойки[1]. Этот алгоритм относится к классу алгоритмов «разделяй и властвуй». Заметим, что для любого многочлена $A(x)$ справедливо тождество

$$y = A(x) = A^0(x^2) + x \cdot A^1(x^2),$$

где $A(x) = a_0 + a_2 x + a_4 x^2 + ... a_{n-2} x^{n-2}$, $A^1(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{n-1}$.

В алгоритме быстрого преобразования Фурье в исходном массиве *a* выделяются два подмассива с чётными и нечётными элементами, затем алгоритм рекурсивно вызывается для них, после этого результаты двух вызовов комбинируются в окончательный ответ.

```
function RFFT(a);
// входной параметр - вектор a, результата - вектор y=DFT(a)
n := length(a)          // считаем количество элементов в a
if n = 1 then return a // DFT вектора длины 1 - это сам вектор
else begin
  // разделяем массив a
  a0 := [a[0], a[2], ..., a[n-2]];
  a1 := [a[1], a[3], ..., a[n-1]];

  // рекурсивный вызов RFFT
  y0 := RFFT(a0);  {*}
  y1 := RFFT(a1);

  // объединяем в окончательный результат
  w := 1;
  wn := e^((2*Pi*i)/n);    // wn - главный комплексный корень
  for k := 0 to n-1 do
    begin
      y[к] := y0[к] + w * y1[k];
      w := w * wn
    end
end;
```

Дадим необходимые комментарии. Оператор, помеченный {*}, вычисляет вектор y^0, у которого $y_k^0 = A^0(\omega_{\frac{n}{2}}^k)$. Однако, используя свойство комплексных корней из единицы, можно записать:

$$y_k^0 = A^0(\omega_{\frac{n}{2}}^k) = A^0(\omega_n^{2k}).$$

Аналогичные рассуждения применяются к вектору y^1. Количество итераций в цикле построения вектора *y* можно уменьшить в два раза, если

[1] В противном случае вектор дополняется нулевыми элементами.

использовать свойство комплексных корней $\omega_n^k = -\omega_n^{k+\frac{n}{2}}$ (справедливое при чётных n):

```
for k := 0 to (n/2)-1 do
  begin
    y[k] := y0[k] + w * y1[k];
    y[k+(n/2)] := y0[k] - w * y1[k];
    w := w * wn
  end
```

Для алгоритма *RFFT* справедливо следующее рекуррентное соотношение, выражающее общее число операций:

$$\begin{cases} T(n) = 2T(n/2) + bn, \\ T(1) = c. \end{cases}$$

Здесь b и c – некоторые константы. Как было показано в параграфе, посвященном рекурсивным алгоритмам, решение данного соотношения удовлетворяет оценке $T(n) = O(n \log_2 n)$. Таким образом, алгоритм *RFFT* позволяет провести вычисление значений многочлена в n точках, являющихся комплексными корнями *n-й* степени из единицы, за время $T(n) = O(n \log_2 n)$.

Покажем, что задача интерполяции многочлена по вычисленным значениям также имеет сложность $O(n\log_2 n)$. Запишем дискретное преобразование Фурье в матричной форме:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Квадратную матрицу в правой части данного равенства обозначим как V_n. Если считать, что индексы элементов данной матрицы изменяются от 0 до $n - 1$, то элемент υ_{ij} равен ω_n^{ij}.

Обратное преобразование Фурье, которое позволит найти коэффициенты многочлена по коэффициентам Фурье, может быть записано в матричной форме как

$$a = V_n^{-1}(y).$$

Теорема 2.8.1. *Элемент с индексом* (j, i) *матрицы* V_n^{-1} *равен* $\dfrac{\omega_n^{-ij}}{n}$.

Доказательство. Пусть утверждение теоремы выполняется. Рассмотрим матричное произведение $V_n^{-1} \cdot V_n$ и элемент полученной матрицы с индексом (j, j'):

$$[V_n^{-1} \cdot V_n]_{jj'} = \sum_{i=0}^{n-1} \left(\frac{\omega_n^{-ij'}}{n} \right) \omega_n^{ij'} = \sum_{i=0}^{n-1} \frac{\omega_n^{i(j'-j)}}{n}$$

Если $j = j'$, то последняя сумма равна 1. Если $j \neq j'$, то по свойству комплексных корней n-ой степени из единицы данная сумма равна 0. Таким образом, матрица $V_n^{-1} \cdot V_n$ является единичной матрицей. Значит, матрица V_n^{-1}, элемент которой с индексом (j, i) равен $\frac{\omega_n^{-ij}}{n}$, действительно является обратной матрицей к матрице V_n. □

Зная структуру матрицы V_n^{-1}, легко записать формулу для получения элемента вектора a:

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{i(j'-j)}$$

Для вычисления вектора a можно использовать алгоритм *RFFT* с небольшими модификациями: заменить ω_n на ω_n^{-1}, в конце разделить элементы полученного вектора на n. Следовательно, в данном случае задача интерполяции многочлена может быть выполнена за время $O(n \log_2 n)$.

Подводя итог, запишем схему алгоритма, выполняющего умножение двух многочленов.

1. Исходные данные: вектор A длины K коэффициентов первого многочлена, вектор B длины M коэффициентов второго многочлена. Требуемый результат: вектор C длины $K + M - 1$ коэффициентов произведения многочленов.
2. Найти число N, такое что $N = 2^P \geq K+M-1$. Дополнить векторы A и B нулевыми элементами до длины N.
3. Выполнить преобразование $\widetilde{A} = DFT_N(A)$, $\widetilde{B} = DFT_N(B)$.
4. Найти вектор C, выполнив почленное умножение векторов A и B.
5. Выполнить преобразование $C = DFT_N^{-1}(\widetilde{C})$.
6. При необходимости, «урезать» вектор C до длины $K + M - 1$, убрав старшие коэффициенты.

В настоящее время используются алгоритмы, позволяющее получит дискретное преобразование Фурье и не применяющие рекурсию (они имеют лучшую константу в асимптотике $O(n \log_2 n)$, а также алгоритмы для преобразования Фурье в кольце вычетов по модулю некоторого простого числа (их применение оправдано, если коэффициенты исходного многочлена – целые числа).

3. Алгоритм Винограда для умножения матриц.

В параграфе, посвященном рекурсивным алгоритмам, был рассмотрен алгоритм Штрассена умножения матриц, имеющий сложность $O(N^{\log_2 7})$. Однако использование данного алгоритма выдвигает довольно жёсткие требование к исходным матрицам (квадратные, размер является степенью двойки), а также требует довольно сложной программной реализации.

Математик Виноград предложил свой алгоритм умножения матриц. Как будет показано далее, этот алгоритм работает за время $O(N^3)$, но он выполняется быстрее, чем стандартные алгоритмы умножения.

Идею алгоритма Винограда рассмотрим на примере. Пусть требуется умножить матрицу размера 2 на 4 на матрицу размера 4 на 3:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{pmatrix}$$

Запишем выражение для вычисления элемента c_{11} в явной форме:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

Выполним преобразование данного выражения следующим образом:

$$c_{11} = (a_{11} + b_{21})(b_{11} + a_{12}) + (a_{13} + b_{41})(b_{31} + a_{41}) - a_{11}a_{12} - b_{11}b_{21} - a_{13}a_{14} - b_{31}b_{41}.$$

На первый взгляд кажется, что число арифметических операций при таком способе вычисления c_{11} только возросло. Однако новый способ позволяет выполнить предварительно некоторые операции, запомнить их результат, а затем использовать в дальнейшем. Например, выражение для c_{12}

$$c_{12} = (a_{11} + b_{22})(b_{12} + a_{12}) + (a_{13} + b_{42})(b_{31} + a_{14}) - a_{11}a_{12} - b_{11}b_{22} - a_{13}a_{14} - b_{32}b_{42}.$$

содержит слагаемые $-a_{11}a_{12} - a_{13}a_{14}$, которые уже использовались при вычислении c_{11}.

Использовав данную идею, можно предложить следующий алгоритм, оформленный в виде программы:

```
// Умножение A[1..N,1..K] на B[1..K,1..M]
L := K div 2;

// считаем множители для A
for i := 1 to N do
  RF[i] := A[i,1] * A[i,2];
  for j := 2 to L do
    RF[i] := RF[i] + A[i,2j-1] * A[i,2j1];

// считаем множители для B
for i := 1 to M do
```

```
  CF[i] := B[1,i] * B[2,i];
  for j := 2 to L do
    CF[i] := CF[i] + B[2j-1,i] * B[2j,i];

// вычисляем C = A * B
for i := 1 to N do
  for j := 1 to M do
    C[i,j] := - (RF[i] + CF[j]);
    for p := 1 to L do
    C[i,j] := C[i,j]+(A[i,2p-1]+B[2p,j])*(A[i,2p] + B[2p-1,j])

// если K - нечетное, требуется добавить слагаемые
if K mod 2 = 1 then
                  for i := 1 to N do
                    for j := 1 to M do
                      C[i,j] := C[i,j] + A[i,K]*B[K,j])
```

Подсчитаем количество арифметических операций, выполняемых в данном алгоритме в случае чётной общей размерности перемножаемых матриц:

Таблица 3

Количество арифметических операций в алгоритме Винограда

Действие	**Умножения**	**Сложения**
Предварительная обработка *A*	*NL*	N(L-1)
Предварительная обработка *B*	*ML*	M(L-1)
Вычисление *C*	*NML*	*NM(3L + 2)*
Общая сумма	$\frac{(NM+N+M)K}{2}$	$\frac{(N+M)(K-2)+NM(3K+4)}{2}$

Как видно из таблицы 3, в алгоритме Винограда выполняется почти в два раза меньше умножений, чем в стандартном алгоритме умножения матриц. Правда в алгоритме увеличивается количество сложений, и требуется дополнительная память для хранения промежуточных результатов. Однако, как показывает практика, применение алгоритма Винограда оправдано в подавляющем большинстве случаев.

3. ЛОГИКА ВЫСКАЗЫВАНИЙ

3.1. ПОНЯТИЕ ВЫСКАЗЫВАНИЯ. ОПЕРАЦИИ С ВЫСКАЗЫВАНИЯМИ

Одним из основных объектов изучения логики являются высказывания. Будем использовать следующее определение.

Определение 3.1.1. (Высказывание) *Высказывание – это повествовательное предложение, про которое можно сказать, истинно оно или ложно.*

Данное определение, по сути, сводит понятие высказывания к понятию предложения. Более точное определение высказывания не представляется возможным, так как высказывание является первоначальным, базовым понятием логики.

Рассмотрим следующие примеры предложений.

1. Число $\sqrt{2}$ является иррациональным.
2. $2 + 2 = 5$.
3. Минск – столица Беларуси.
4. Если число $\sqrt{2}$ – иррациональное, то число $\sqrt{2}+1$ также является иррациональным.
5. В десятичной записи числа $\pi + e$ встречаются 100 семёрок подряд.
6. Который час?
7. Идите решать задачу к доске.

Первые пять предложений из приведённых примеров являются высказываниями, последние два – нет. Мы можем утверждать, что первое, третье и четвёртое высказывание истинны, второе высказывание ложно. Что же касается пятого предложения, то мы, наверное, никогда не узнаем истинно оно или ложно, однако с точки зрения математической логики оно является высказыванием.

Для математической логики не является важным установление истинности элементарных фактов. Это удел физики, философии и других прикладных наук. Математическая логика позволяет устанавливать и проверять правильность *рассуждений.* Все высказывания математическая логика намеренно сводит к двум возможным классам: истинные высказывания и ложные высказывания. Для обозначения класса истинных высказываний мы будем использовать символ **T**, класс ложных высказываний обозначим как **F**.

Рассмотрим два высказывания: «2+2=4» и «Если отец или мать смотрят телевизор, то его смотрит их сын и не смотрит их дочь». Интуитивно понятно, что второе высказывание имеет более сложную структуру, чем первое. Очевидно, что в русском языке (как впрочем, и в любом другом) есть способы для построения сложных высказываний из простых. Данные способы мы назовём операциями над высказываниями.

Определение 3.1.2. (Операции над высказываниям) *Пусть даны высказывания, обозначаемые далее как A и B. Тогда высказывание*

1) «не A» называется отрицанием высказывания A;

2) «A и B» называется конъюнкцией высказываний A и B;

3) «A или B» называется дизъюнкцией высказываний A и B;

4) «если A, то B» называется импликацией высказываний A и B;

5) «A тогда и только тогда, когда B» называется эквивалентностью (эквиваленцией) высказываний A и B;

Для обозначения операций введём символы: $\neg$ – отрицание, & – конъюнкция, $\vee$ – дизъюнкция, $\rightarrow$ – импликация, $\leftrightarrow$ – эквивалентность.

Зависимость значения истинности результата операции от значений аргументов определяется таблицей 4.

Таблица 4

Операции над высказываниями

A	B	$\neg$A	A&B	A$\vee$B	A$\rightarrow$B	A$\leftrightarrow$B
F	**F**	**T**	**F**	**F**	**T**	**T**
F	**T**	**T**	**F**	**T**	**T**	**F**
T	**F**	**F**	**F**	**T**	**F**	**F**
T	**T**	**F**	**T**	**T**	**T**	**T**

Прокомментируем некоторые операции над высказываниями. В русском языке союз «или» понимается в двух смыслах: разделительном – или то, или другое, но не оба, и соединительном – или то, или другое, или оба. Как видим из таблицы, союз «или» будет пониматься в соединительном смысле.

Рассмотрим операцию импликации. Проанализируем следующий пример. Отец говорит сыну-студенту: «Если ты сдашь сессию на "отлично", то я куплю тебе машину». В каких случаях отец нарушит своё обещание, то есть солжёт? Пусть сын сдал сессию на «отлично», а отец купил ему машину. Обещание отца, очевидно, было правдой. Если сын сдал сессию, а отец не купил ему машину, то самое мягкое, что можно сказать о таком отце, что он лжец. Если сын не сдал сессию, а машина все-таки была куплена, назвать отца лжецом мы не можем (он просто очень щедрый человек). Если сын не сдал сессию, то и отец вправе не покупать ему машину, своё обещание он выполняет. Эти ситуации отражаются значением операции импликации.

В естественном языке предложения вида «если ..., то ...» обычно несут причинно-следственный оттенок. В математической логике высказывания, являющиеся частями импликации, могут быть произвольными. Примером такой импликации может служить высказывание «Если 2 + 2 = 5, то луна сделана из зелёного сыра». По таблице истинности данное высказывание должно считаться истинным.

Если для высказываний определены операции, то можно разделить высказывания на простые и сложные по следующему признаку.

Определение 3.1.3. (Атомарное и сложное высказывание) *Высказывание, выражающее единичный факт, называется атомарным (простым). Высказывание, полученное из других при помощи операций над высказываниями, называется сложным.*

3.2. ФОРМУЛЫ ЛОГИКИ ВЫСКАЗЫВАНИЙ. ИНТЕРПРЕТАЦИЯ

Переход к использованию формул в логике высказываний аналогичен переходу от арифметики к алгебре. Использование формул позволяет указать свойства, присущие операциям с любыми высказываниями.

Дадим чёткое определение формулы логики высказываний. Рассмотрим множество символов, которое назовём *алфавитом*. В этом множестве выделим следующие группы символов:

1) константы – символы **T** и **F**;
2) переменные – символы A, B, ..., A_1, (теоретически, бесконечный набор);
3) символы логических операций – символы $\neg$, $\&$, $\vee$, $\rightarrow$, $\leftrightarrow$;
4) служебные символы – символы (и);

Замечание. Любой бесконечный алфавит можно воспринимать как конечный, если заменить обозначения индексов по следующей схеме: A_1 означает A', $A_2 - A''$, и так далее.

Определение 3.2.1. (Формула логики высказываний) *Последовательность символов назовём формулой логики высказываний (ФЛВ), если она построена по следующим правилам:*

1. Константа и переменная являются ФЛВ.

2. Если F_1 и F_2 – произвольные ФЛВ, то $(\neg F_1)$, $(F_1 \,\&\, F_2)$, $(F_1 \vee F_2)$, $(F_1 \rightarrow F_2)$, $(F_1 \leftrightarrow F_2)$ – ФЛВ.

3. Других ФЛВ нет.

В этом индуктивном определение первый пункт задаёт исходные ФЛВ, а второй пункт даёт правила построения новых ФЛВ из имеющихся.

Запись вида $F_1(A, B, C)$ в дальнейшем будет означать что, в ФЛВ F_1 входят переменные *A, B, C* и только они.

Если следовать определению ФЛВ строго, то при записи формулы надо использовать много скобок. Это неудобно для восприятия. Чтобы уменьшить количество скобок условимся ввести приоритет (силу связывания) для символов логических операций. Будем считать, что символ $\neg$ имеет наивысший приоритет, далее идет $\&$, затем $\vee$, затем $\rightarrow$ и $\leftrightarrow$. Последние две связки имеют одинаковый приоритет. Используя эти соглашения формулу $((A \,\&\, B) \rightarrow C)$ можно записать так: $A \,\&\, B \rightarrow C$. Формулы вида $(((A \,\&\, B) \,\&\, C) \,\&\, D)$ будем записывать как $A \,\&\, B \,\&\, C \,\&\, D$. Иногда так же будем заменять символ $\neg$ на черту над формулой: $(\neg(A \,\&\, B))$ означает $\overline{A \,\&\, B}$.

Подчеркнём, что произвольная формула логики высказываний является всего лишь последовательностью символов. Придание символам формулы и самой формуле определенного значения называется *интерпретацией*.

Определение 3.2.2. (Интерпретация ФЛВ) *Пусть $F(A_1,..., A_n)$ – некая формула логики высказываний, $B_1, ..., B_n$ – набор произвольных высказываний ($B_i \in \{\boldsymbol{F}, \boldsymbol{T}\}$). Заменим в F переменные A_i на высказывания B_i, константы будем понимать как произвольное ложное (константа **F**) или истинное (константа **T**) высказывание, символы логических операций – как соответствующие операции над высказываниями, скобки будут определять порядок выполнения операций над высказываниями. В результате вместо ФЛВ мы получим некое сложное высказывание, которое истинно или ложно. Описанный процесс получения из строки, представляющей ФЛВ, некоего высказывания назовём интерпретацией. Значение полученного высказывания назовём значением ФЛВ при данной интерпретации.*

Интерпретацию можно понимать как некое отображение φ, ставящее в соответствие строке, представляющей ФЛВ, истинное или ложное значение. Таким образом, можно использовать нотацию

$$\varphi : F \rightarrow \{\mathbf{F},\mathbf{T}\},$$

где F – множество всех ФЛВ, а значение формулы F при интерпретации φ обозначать как $\varphi(F)$.

Если ФЛВ $F(A_1, ..., A_n)$ зафиксирована, то значение формулы при разных интерпретациях фактически определяется набором значений ее переменных. В связи с этим можно говорить о значении ФЛВ $F(A_1, ..., A_n)$ на наборе значений $(B_1, ..., B_n)$.

Выделим особый вид интерпретации.

Определение 3.2.3. (Модель ФЛВ) *Пусть $F(A_1, ..., A_n)$ – некая ФЛВ. Интерпретацию φ назовем моделью формулы F, если $\varphi(F) = \boldsymbol{T}$.*

Пусть $F = \{F_1(A_1, ..., A_n), F_2(A_1, ..., A_n), ..., F_k(A_1, ..., A_n)\}$ – некое множество ФЛВ. Интерпретацию φ назовем моделью множества F, если $\varphi(F_i) = \boldsymbol{T}$ для любой $F_i \in F$.

Это определение позволяет выделить специальные классы ФЛВ.

Определение 3.2.4. (Противоречие, выполнимая ФЛВ, тавтология) *Противоречием назовем ФЛВ, у которой нет моделей. Выполнимой назовем ФЛВ, у которой существует хотя бы одна модель. Тавтологией назовем ФЛВ, для которой любая интерпретация является моделью.*

Для любой формулы логики высказываний можно определить, к какому классу из вышеперечисленных классов она относится. Для этого достаточно рассмотреть все различные интерпретации данной формулы. Их 2^n, где n – количество переменных, входящих в формулу.

3.3. РАВНОСИЛЬНЫЕ ФЛВ

Рассматривая множество формул логики высказываний, можно заметить, что некоторые строки задают такие ФЛВ, значения которых совпадают при любых интерпретациях.

Определение 3.3.1. (Равносильные ФЛВ) *Формулы логики высказываний F и G назовем равносильными, если для любой интерпретации φ выполняется равенство $\varphi(F) = \varphi(G)$.*

Тот факт, что ФЛВ F и G равносильны, будем записывать так: $F \equiv G$.

Чтобы установить равносильность двух ФЛВ достаточно перебрать все возможные наборы их переменных и сравнить на одинаковых наборах значения формул. Понятие равносильности тесно связано с понятием выполнимости.

Лемма 3.3.1. *$F \equiv G$ тогда и только тогда, когда формула $(F \leftrightarrow G)$ – тавтология.*

Доказательство данной леммы оставляем в качестве упражнения.

Приведем в виде списка равносильности, наиболее часто используемые на практике (некоторые из них имеют собственные названия):

1. $A \,\&\, A \equiv A$
2. $A \vee A \equiv A$
3. $A \,\&\, B \equiv B \,\&\, A$ (коммутативный закон)
4. $A \vee A \equiv B \vee A$ (коммутативный закон)
5. $(A \,\&\, B) \,\&\, C \equiv A \,\&\, (B \,\&\, C)$ (ассоциативный закон)
6. $(A \vee B) \vee C \equiv A \vee (B \vee C)$ (ассоциативный закон)
7. $A \,\&\, (B \vee C) \equiv (A \,\&\, B) \vee (A \,\&\, C)$ (дистрибутивный закон)
8. $A \vee (B \,\&\, C) \equiv (A \vee B) \,\&\, (A \vee C)$ (дистрибутивный закон)
9. $A \,\&\, \mathbf{F} \equiv \mathbf{F}$ (свойство нуля)
10. $A \,\&\, \mathbf{T} \equiv A$ (свойство единицы)
11. $A \vee \mathbf{F} \equiv A$ (свойство нуля)
12. $A \vee \mathbf{T} = \mathbf{T}$ (свойство единицы)
13. $\neg(\neg A) \equiv A$ (закон «двойного отрицания»)
14. $\neg(A \,\&\, B) \equiv (\neg A) \vee (\neg B)$ (правило де Моргана[1])
15. $\neg(A \vee B) \equiv (\neg A) \,\&\, (\neg B)$ (правило де Моргана)
16. $A \,\&\, \neg A \equiv \mathbf{F}$

[1] Де Морган А. (Augustus De Morgan, 1806-1871)

17. $A \vee \neg A \equiv \mathbf{T}$ (закон исключения третьего)

18. $A \to B \equiv (\neg A) \vee B$

19. $A \to B \equiv (\neg B) \to (\neg A)$

20. $(F \to G) \& (G \to F) \equiv F \leftrightarrow G$

21. $(A \& B) \vee (\neg A \& B) \equiv B$ («склеивание» конъюнкций)

22. $(A \vee B) \& (\neg A \vee B) \equiv B$ («склеивание» дизъюнкций)

Использование равносильных ФЛВ позволяет для любой формулы получить равносильную некоего специального вида.

Определение 3.3.2. (Литерал) *Литералом назовем формулу вида A или* $\neg A$ *где A – переменная.*

Определение 3.3.3. (Элементарная конъюнкция (дизъюнкция)) *Элементарной конъюнкцией (дизъюнкцией) назовем конъюнкцию (дизъюнкцию) литералов.*

Определение 3.3.4. (Конъюнкт, дизъюнкт) *Конъюнктом (дизъюнктом) назовем конъюнкцию (дизъюнкцию) литералов или констант.*

Определение 3.3.5. (Конъюнктивная нормальная форма) *Будем говорить что формула F находится в конъюнктивной нормальной форме (КНФ), если она является конъюнкцией элементарных дизъюнкций или элементарной дизъюнкцией.*

Определение 3.3.6. (Дизъюнктивная нормальная форма) *Будем говорить что формула F находится в дизъюнктивной нормальной форме (ДНФ), если она является дизъюнкцией элементарных конъюнкций или элементарной конъюнкцией.*

Теорема 3.3.1. *Для любой формулы логики высказываний существует равносильная формула в КНФ и равносильная формула в ДНФ.*

Доказательство. В качестве доказательства теоремы приведем алгоритм приведения произвольной формулы к КНФ

Шаг 1. Исключить из исходной формулы эквивалентность и импликацию, используя равносильности 18 и 20.

Шаг 2. Переместить отрицание к литералам с помощью равносильностей 13-15.

Шаг 3. Если формула содержит подформулу вида $A \vee (B \& C)$, то заменить ее на равносильную формулу $(A \vee B) \& (A \vee C)$.

Аналогичный алгоритм можно реализовать и для приведения формулы к ДНФ. □

В качестве примера рассмотрим приведение к КНФ формулы $\neg(A \leftrightarrow B) \& A$.

Шаг1. $\neg(A \leftrightarrow B) \& A \equiv \neg((A \to B) \& (B \to A)) \& A \equiv \neg((\neg A \vee B) \& (\neg B \vee A)) \& A$.

Шаг 2.

$\neg((\neg A \vee B) \& (\neg B \vee A)) \& A \equiv (\neg(\neg A \vee B) \vee \neg(\neg B \vee A)) \& A \equiv$

$\equiv ((A \& \neg B) \vee (B \& \neg A)) \& A.$

Шаг3. $((A \& \neg B) \vee (B \& \neg A)) \& A \equiv ((A \& \neg B) \vee B) \& ((A \& \neg B) \vee \neg A) \& A$.

$((A \& \neg B) \vee B) \& ((A \& \neg B) \vee \neg A) \& A \equiv$

$\equiv (A \vee B) \& (\neg B \vee B) \& (A \vee \neg A) \& (\neg B \vee \neg A) \& A.$

$(A \vee B) \& (\neg B \vee B) \& (A \vee \neg A) \& (\neg B \vee \neg A) \& A \equiv (A \vee B) \& \mathbf{T} \& \mathbf{T} \& (\neg B \vee \neg A) \& A$

$(A \vee B) \& \mathbf{T} \& \mathbf{T} \& (\neg B \vee \neg A) \& A \equiv (A \vee B) \& (\neg B \vee \neg A) \& A$.

Таким образом, $\neg(A \leftrightarrow B) \& A \equiv (A \vee B) \& (\neg B \vee \neg A) \& A$.

В принципе, для построения дизъюнктивной и конъюнктивной нормальных форм возможно применение методов, использовавшихся для построения СДНФ и СКНФ булевой функции по таблице истинности. Для приведенной формулы такое построение даст более короткую ДНФ: $\neg(A \leftrightarrow B) \& A \equiv A \& \neg B$.

3.4. ЛОГИЧЕСКОЕ СЛЕДСТВИЕ

При рассуждении в естественном языке один факт может формулироваться как следствие других фактов. При этом считается, что если все базовые факты (посылки) истинны, то и новый факт (заключение) является истинным.

Для логики высказываний данный вид рассуждений определяется как получение логического следствия.

Определение 3.4.1. (Логическое следствие) *Формула логики высказываний G называется логическим следствием формул F_1, ..., F_n, если любая модель множества формул F_1, ..., F_n является моделью формулы G.*

Логическое следствие обозначается следующим образом: $F_1, ..., F_n \models G$.

Формула G из определения может быть истинна на таком наборе, на котором некоторые из F_i ложны. Но истинность всех F_i *гарантировано* ведет к истинности G.

Лемма 3.4.1. *Если $F \models G$, то $M_F \subseteq M_G$, где M_F обозначает множество всех моделей формулы F, M_G – множество всех моделей формулы G.*

Это утверждение непосредственно следует из определения логического следствия.

Сформулируем теоремы, которые позволяют сводить проверку логического следствия к проверке класса некой формулы.

Теорема 3.4.1. (Основная теорема теории логического вывода)

1) $F \models G$ тогда и только тогда, когда формула $(F \to G)$ – тавтология.

2) $F_1,...,F_n \models G$ тогда и только тогда, когда $(F_1 \& ... \& F_n) \to G$ – тавтология.

Доказательство данной теоремы оставляем в качестве упражнения.

Теорема 3.4.2. *F_1, F_2, ..., F_n ⊨ G тогда и только тогда, когда $F_1 \& F_2 \& \ldots \& F_n \& \neg G$ – противоречие.*

Использование определения логического следствия позволяет дать другое определение равносильных ФЛВ, эквивалентное предыдущему.

Определение 3.4.2. (Равносильные ФЛВ) *Две формулы логики высказываний F и G назовем равносильными, если F ⊨ G и G ⊨ F.*

3.5. МЕТОД РЕЗОЛЮЦИЙ ДЛЯ ЛОГИКИ ВЫСКАЗЫВАНИЙ

Метод резолюций был предложен Дж. Робинсоном[1] в 1965 году. Этот метод применяется для установления того факта, что некоторая ФЛВ является противоречием.

Опишем идею метода резолюций. Пусть дана ФЛВ *F*. Получим с помощью некой специальной процедуры логическое следствие *F* – формулу G_1. Далее применим эту процедуру к формуле G_1 для получения формулы G_2, и так далее. Если на некотором шаге мы получим формулу **F** (то есть «тождественная ложь»), то исходная формула является противоречием.

Действительно, так как $F \models G_1$, то по лемме 4.1 $M_F \subseteq M_{G1}$. Учитывая, что множество моделей формулы «тождественная ложь» есть $\varnothing$, получаем цепочку $M_F \subseteq M_{G1} \subseteq \ldots \subseteq \varnothing$. Значит, $M_F = \varnothing$ и формула *F* является противоречием.

Лемма 3.5.1. *Пусть $F \models G$. Тогда $F \& G \equiv F$.*

Доказательство. Так как $F \models G$, *то* $F \to G \equiv \mathbf{T}$ (теорема 4.1). Получаем $F \equiv F \& \mathbf{T} \equiv F \& (F \to G) \equiv F \& (\neg F \vee G) \equiv (F \& \neg F) \vee (F \& G) \equiv \mathbf{F} \vee (F \& G) \equiv F \& G$.

Определение 3.5.1. (Резольвента) *Пусть даны две формулы $A \vee B$ и $\neg A \vee C$. Резольвентой этих формул назовем формулу $B \vee C$.*

Лемма 3.5.2. *Резольвента – это логическое следствие порождающих ее формул.*

Доказательство. Пусть $A \vee B$ и $\neg A \vee C$ – исходные формулы, $B \vee C$ – их резольвента. Рассмотрим формулу $((A \vee B) \& (\neg A \vee C)) \to (B \vee C)$. Можно показать, что это тавтология. Значит, по теореме 4.1, $A \vee B, \neg A \vee C \models B \vee C$. □

Рассмотрим подробнее алгоритм метода резолюций для формул логики высказываний. Исходная формула преобразуется в равносильную формулу в КНФ. Образуется множество дизъюнктов этой формулы. На каждом шаге алгоритма во множестве дизъюнктов ищется пара формул, которые можно резольвировать, и к множеству дизъюнктов добавляется резольвента пары. Если на некотором шаге получится резольвента **F**, то исходная формула является противоречием.

Приведем пример использования алгоритма.

---

[1] Дж. Робинсон (J. Robinson)

1. $(\neg A \to \neg B) \& (\neg C \to \neg A) \& B \& \neg C$ – исходная формула;
2. $(A \vee \neg B) \& (C \vee \neg A) \& B \& \neg C$ – формула в КНФ;
3. $A \vee \neg B \,\&\, C \vee \neg A, B, \neg C$ – исходное множество дизъюнктов;
4. $A \vee \neg B, C \vee \neg A, B, \neg C, \neg B \vee C$ – резольвировали $A \vee \neg B$ и $C \vee \neg A$;
5. $A \vee \neg B, C \vee \neg A, B, \neg C, \neg B \vee C, C$ – резольвировали $\neg B \vee C$ и $B (B \equiv B \vee \mathbf{F})$;
6. $A \vee \neg B, C \vee \neg A, B, \neg C, \neg B \vee C, C,$ **F** – резольвировали C и $\neg C$. Исходная формула – противоречие.

При помощи метода резолюций можно установить, выполняется ли логическое следствие $F_1, F_2 \ldots, F_n \models G$. Для этого по теореме 3.4.2 достаточно проверить, является ли формула $F_1 \& F_2 \& \ldots \& F_n \& \neg G$ противоречием.

Пусть требуется проверить $A \vee B, A \to C, B \to D \models C \vee D$. Применим метод резолюций. $(A \vee B) \& (A \to C) \& (B \to \mathrm{D}) \& \neg (\mathrm{C} \vee D)$ – формула, построенная по теореме 3.4.2. Эта формула в КНФ:

$$(A \vee \mathrm{B}) \& (\neg \mathrm{A} \vee \mathrm{C}) \& (\neg \mathrm{B} \vee \mathrm{D}) \& \neg \mathrm{C} \& \neg \mathrm{D}.$$

Запишем исходные дизъюнкты и получаемые резольвенты.

1. $A \vee B$;
2. $\neg A \vee C$;
3. $\neg B \vee D$;
4. $\neg C$;
5. $\neg D$;
6. $B \vee C$ – резольвента 1 и 2;
7. $C \vee D$ – резольвента 3 и 6;
8. D – резольвента 4 и 7;
9. **F** – резольвента 5 и 8.

Алгоритм закончен. Формула, построенная по теореме 3.4.2, – противоречие. Значит исходное логическое следствие имеет место.

3.6. ПРИМЕНЕНИЕ ЛОГИКИ ВЫСКАЗЫВАНИЙ ПРИ РЕШЕНИИ ЗАДАЧ

Решение задач при помощи логики высказываний обычно проводят в три этапа. На первом этапе задача формализуется, то есть условие и вопрос задачи записывается в виде набора формул логики высказываний. На втором этапе полученные формулы преобразуются в равносильные и анализируются. Третий этап заключается в интерпретации полученных результатов на естественном языке.

Решение задач при помощи логики высказываний проиллюстрируем на примерах.

Задача 3.6.1. *Семья, состоящая из отца О, матери М, сына С и двух дочерей А и Д купила телевизор. Условились, что в первый вечер будут смотреть телевизор в таком порядке:*

1. Когда отец О смотрит телевизор, то мать М делает то же.

2. Сын С и дочь Д, оба или один из них, смотрят телевизор.

3. Из двух членов семьи – мать М и дочь А – смотрит телевизор одна и только одна.

4. Дочь А и сын С или оба смотрят телевизор, или оба не смотрят.

5. Если дочь Д смотрит передачу, то отец О и сын С делают то же.

Кто из членов семьи в этот вечер смотрит передачу?

Решение. Данная задача может быть отнесена к типу задач, в которых требуется упростить набор правил. Для решения задачи запишем исходные правила при помощи ФЛВ, обозначив буквами *О, М*, *С*, *А*, *Д* высказывания, истинные, если соответствующий член семьи смотрит телевизор:

1. $O \to M = F_1$.

2. $C \vee Д = \mathrm{F}_2$.

3. $(\mathrm{M} \to \neg \mathrm{A}) \vee (A \to \neg M) = F_3$.

4. $A \leftrightarrow C = \mathrm{F}_4$.

5. $Д \to (O \,\&\, C) = \mathrm{F}_5$.

Так как все пять правил должны выполняться, то выражение $F_1 \& F_2 \& F_3 \& F_4 \& F_5$ должно быть тавтологией.

Упростим формулу F_3:

$$(\mathrm{M} \to \neg \mathrm{A}) \vee (A \to \neg M) \equiv (\neg M \vee \neg A) \,\&\, (\neg A \vee \neg M) \equiv \neg M \vee \neg A.$$

Преобразуем формулу F_5:

$$Д \to (O \,\&\, C) \equiv \neg Д \vee (\mathrm{O} \,\&\, \mathrm{C}) \equiv (\neg Д \vee \mathrm{O}) \,\&\, (\neg Д \vee \mathrm{C}).$$

Должно выполняться

$$(O \to M) \,\&\, (C \vee Д) \,\&\, (\neg \mathrm{M} \vee \neg \mathrm{A}) \,\&\, (A \leftrightarrow C) \,\&\, (\neg Д \vee \mathrm{O}) \,\&\, (\neg Д \vee \mathrm{C}) \equiv \mathbf{T}.$$

«Склеим» второй и шестой члены конъюнкции:

$$(C \vee Д) \,\&\, (\neg Д \vee C) \equiv \mathrm{C}.$$

Значит, $C = \mathbf{T}$. Из F_4 следует, что $A = \mathbf{T}$. Подставив значение $A = \mathbf{T}$ в формулу $\neg \mathrm{M} \vee \neg \mathrm{A}$, получим:

$$(\neg \mathrm{M} \vee \neg \mathbf{T}) \equiv (\neg \mathrm{M} \vee \mathbf{F}) \equiv \neg \mathrm{M}.$$

Должно выполняться ¬М = **T**, а значит *М* = **F**. Так как должно выполняться (О → М) = **T**, а *М* = **F**, то *О* = **F**. Так как должно выполняться (¬Д ∨ О) = **T**, а *О* = **F**, то ¬Д = **T**, *Д* = **F**.

Таким образом, (*О*, *М*, *С*, *А*, *Д*) = (**F**, **F**, **T**, **T**, **F**). Интерпретируя ответ, получим, что телевизор смотрят сын и дочь *А*.

Задача 3.6.2. *По подозрению в совершенном преступлении задержали Брауна, Джона и Смита. Один из них был уважаемым в городе стариком, другой был малоизвестным чиновником, третий – известным мошенником. В процессе следствия старик говорил правду, мошенник лгал, а третий задержанный в одном случае говорил правду, а в другом – ложь. Вот, что они утверждали:*

1. Браун: Я совершил это. Джон не виноват.

2. Джон: Браун не виноват. Преступление совершил Смит.

3. Смит: Я не виноват. Виноват Браун.

Требуется определить имена старика, мошенника и чиновника, и кто из них виноват, если известно, что преступник один.

Решение. Введем следующие обозначения для высказываний:

1. B = «Браун – преступник».

2. J = «Джон – преступник».

3. S = «Смит – преступник».

Пусть подозреваемый утверждает два факта *X* и *Y*. Если подозреваемый говорит правду в обоих случаях, образуем конъюнкцию его утверждений *X&Y*. Если подозреваемый в обоих случаях лжет, образуем конъюнкцию отрицаний утверждений $\neg X \& \neg Y$. Тот факт, что подозреваемый один раз солгал, один раз сказал правду, соответствует формуле $(\neg X \& Y) \vee (X \& \neg Y)$ (так как не указано, солгал подозреваемый, сказав первый факт или второй). Преобразуем последнюю формулу в равносильную, содержащую конъюнкцию высказываний: $(\neg X \& Y) \vee (X \& \neg Y) = (X \vee Y) \& \neg (X \& Y)$. В нашей задаче для каждого показания нужно рассмотреть три возможных случая (сказал правду, солгал, сказал «полуправду»). Построим таблицу:

Таблица 5

Формальная запись показаний

	Сказал правду	**Солгал**	**Сказал «полуправду»**
Браун	$B \& \neg J$	$\neg B \& J$	$(B \vee \neg J) \& \neg (B \& \neg J)$
Джон	$\neg B \& S$	$B \& \neg S$	$(\neg B \vee S) \& \neg (\neg B \& S)$
Смит	$\neg S \& B$	$S \& \neg B$	$(\neg S \vee B) \& \neg (\neg S \& B)$

Далее требуется рассмотреть шесть вариантов, соответствующих различным наборам показаний. Например, в случае, когда Браун сказал правду, Джон

солгал, а Смит сказал «полуправду», получаем следующую конъюнкцию утверждений, которая должна быть истинна:

$$(B \,\&\, \neg J) \,\&\, (B \,\&\, \neg S) \,\&\, (\neg S \vee B) \,\&\, \neg(\neg S \,\&\, B) = \mathbf{T}.$$

Конъюнкция упрощается, чтобы найти набор переменных, придающий ей истинное значение, или показать, что такого набора нет. Преобразовав левую часть равенства в КНФ, получим:

$$B \,\&\, \neg J \,\&\, (\neg S \vee B) \,\&\, \neg S \,\&\, \neg B = \mathbf{T}.$$

Мы видим, что нет набора переменных, при котором формула истинна, так как в КНФ входит литерал и его отрицание (B и $\neg B$).

Аналогичным образом рассматриваются еще пять вариантов. Проделайте данную работу самостоятельно. Ответ задачи: преступник – Смит, он солгал и является мошенником, Браун сказал полуправду, он чиновник, а Джон – старик.

Задача 3.6.3. *Проверить правильность логических рассуждений: «Или Петя и Ваня братья или они однокурсники. Если Петя и Ваня братья, то Сергей и Ваня не братья. Если Петя и Ваня однокурсники, то Ваня и Миша тоже однокурсники. Значит, или Ваня и Миша однокурсники или Сергей и Ваня не братья».*

Решение. Выделим следующие элементарные высказывания:

1. A = «Петя и Ваня – братья».
2. B = «Петя и Ваня – однокурсники».
3. C = «Сергей и Ваня – не братья».
4. D = «Ваня и Миша – однокурсники».

В задаче формулируется некое логическое следствие из фактов. С учетом введенных обозначений рассуждения задачи можно формализовать так:

$$A \vee B,\ A \to C,\ B \to D \models C \vee D.$$

В предыдущем параграфе корректность данного логического следствия проверялась при помощи метода резолюций. Рассуждения задачи логически правильны.

Приведем несколько задач для самостоятельного решения.

Задача 3.6.4. *В деле об убийстве имеются двое подозреваемых – Петр и Павел. Допросили четырех свидетелей, которые последовательно дали такие показания: «Петр не виноват», «Павел не виноват», «Из двух первых показаний по меньше мере одно истинно», «Показания третьего ложны». Четвертый свидетель оказался прав. Кто преступник?*

Задача 3.6.5. *На двери деканата злоумышленники нарисовали несколько карикатур на преподавателей. Подозрение пало на известных хулиганов и вольнодумцев Пашу и Сашу. Обнаружились три свидетеля, которые заявили:*

1. Первый: Это они сделали вместе.

2. Второй: Рисовал на двери только Саша, Паша в этом не участвовал.

3. Третий: Если Паша рисовал на двери, то Саша тоже принимал в этом участие.

Какой вывод можно сделать из показаний свидетелей, если выяснилось, что все они врали, т.е. говорили прямо противоположное тому, что было на самом деле?

Задача 3.6.6. *Разбирается дело Брауна, Джонса и Смита. Один из них совершил преступление. В процессе расследования каждый из них сделал по два заявления.*

1. Браун: Я не делал этого. Джонс не делал этого.

2. Джонс: Смит сделал это. Браун не виноват.

3. Смит: Я не виноват. Виноват Браун.

Далее было установлено, что один из них дважды солгал, другой дважды сказал правду, а третий раз солгал, раз сказал правду. Кто совершил преступление? Какой будет ответ при условии, что каждый из них один раз сказал правду, а один раз солгал?

4. ЛОГИКА ПРЕДИКАТОВ

4.1. ПРЕДИКАТ: ОПРЕДЕЛЕНИЕ И ПРИМЕРЫ

Логика высказываний обладает довольно слабыми выразительными возможностями. В ней нельзя выразить даже очень простые с математической точки зрения рассуждения. Рассмотрим, например, следующее умозаключение. «Всякое целое число является рациональным. Число 2 – целое. Следовательно, 2 – рациональное число». Все эти утверждения с точки зрения логики высказываний являются атомарными. Средствами логики высказываний нельзя вскрыть внутреннюю структуру и поэтому нельзя доказать логичность этого рассуждения в рамках логики высказываний. Мы рассмотрим расширение логики высказываний, которое называется логика предикатов.

Рассмотрим высказывание «12 делится на 5». Обобщим его, превратив в *высказывателъную форму*: «x делится на 5». Считаем, что здесь $x \in \mathrm{N}$. Для одних x мы получим истинное высказывание, для других – ложное. Тем самым мы задали некую функцию, определенную на N и принимающую истинные или ложные значения.

Определение 4.1.1. (Предикат) *Пусть задано множество M. n-местным предикатом $P(x_1, \ldots, x_n)$ называется функция $P : M^n \to \{\boldsymbol{F}, \boldsymbol{T}\}$, определенная на наборах длины n элементов множества M и принимающая истинное или ложное значение.*

Следуя определению, будем использовать термины «одноместный предикат», «двухместный предикат», «n-местный предикат» для обозначения количества аргументов функции. Отметим, что предикат может содержать *фиктивные аргументы,* т. е. аргументы, от которых значение предиката не зависит.

Будем считать, что высказывание – это нульместный предикат, то есть функция-константа.

Приведем несколько примеров предикатов.

1) $P_1 : \mathrm{N} \to \{\mathbf{F}, \mathbf{T}\}$, $P_1(x)$ = «x – четное число». Тогда $P_1(3)= \mathbf{F}$, $P_1(100) = \mathbf{T}$.

2) $P_2 : \mathrm{N} \times \mathrm{N} \to \{\mathbf{F}, \mathbf{T}\}$, $P_2(x)$= «x делится на y». Тогда $\mathrm{P}_2(6,2)=\mathbf{T}$, $P_2(5,3)=\mathbf{F}$.

3) Пусть задана произвольная функция от n аргументов $y = f(x_1, \ldots, x_n)$. Любой такой функции можно поставить в соответствие предикат $P_{n+1}(x_1, \ldots, x_n, y)$ = «$y = f(x_1, \ldots, x_n)$», который называется *представлением* функции $f(x_1, \ldots, x_n)$.

Определение 4.1.2. (Множество истинности) *Пусть задан n-местный предикат P на множестве M. Подмножество*

$$I_P \subseteq M^n,\ I_P = \{(x_1,\ldots,x_n) \in M^n \,|\, P(x_1,\ldots,x_n) = \mathbf{T}\}$$

назовем множеством истинности предиката P.

Предикат полностью определяется своим множеством истинности. Это позволяет задавать n-местные предикаты на множестве M, описав некое подмножество M^n.

Выделим некие специальные классы предикатов.

Определение 4.1.3. (Классы предикатов) *Предикат P называется*

1. тождественно истинным, если $I_P = M$;

2. тождественно ложным, если $I_P = \varnothing$;

3. выполнимым, если $I_P \neq \varnothing$, $I_P \subset M$.

Определение 4.1.4. (Следствие и равносильность предикатов) *Пусть на множестве M заданы предикаты P и S. Предикат S называется следствием предиката P, если* $I_P \subset I_S$. *Если* $I_P = I_S$, *то предикаты P и S называются равносильными.*

4.2. ОПЕРАЦИИ НАД ПРЕДИКАТАМИ. КВАНТОРЫ

Значение предиката есть «истинна» или «ложь», поэтому для предикатов возможно применять все операции, определенные для высказываний.

Хотя, в принципе, возможно рассматривать данные операции для предикатов, заданных на разных множествах, мы ограничимся применением их к предикатам, определенным на одном и том же множестве. Множество аргументов предиката-результата будем рассматривать как объединение множеств аргументов предикатов-операндов.

Рассмотрим несколько примеров.

1) Пусть заданы два предиката. $P(x)$ определен на множестве N и $P(x)$ = «x – четное число», $S(y)$ также определен на множестве N и $S(y)$ = «y делится на 5». Тогда в результате операции конъюнкции мы получим двухместный предикат $R(x, y)$, определенный на множестве N × N и $R(x, y)$ = «x – четное число и y делится на 5».

2) Пусть заданы два предиката. $P(x)$ определен на множестве N и $P(x)$ = «x – четное число», $S(x)$ определен на множестве N и $S(x)$ = «x делится на 5». В результате операции конъюнкции мы получим одноместный предикат $R(x)$, определенный на N и $R(x)$ = «x – четное число, которое делится на 5».

Если $P(x)$ и $S(x)$ – два предиката, заданные на одном и том же множестве M, то их множества истинности ведут себя при логических операциях следующим образом:

Таблица 6

Поведение множеств истинности при логических операциях

Операция	**Множество истинности I_R**
$R(x) = P(x) \& S(x)$	$I_P \cap I_S$
$R(x) = P(x) \vee S(x)$	$I_P \cup I_S$
$R(x) = \neg P(x)$	$M \setminus I_P$
$R(x) = P(x) \rightarrow S(x)$	$(M \setminus I_P) \cup I_S$

Кроме операций, которые позволяют из одних предикатов получать другие, для предикатов определены особые операции, позволяющие осуществлять

переход от предикатов к высказываниям. Это так называемые *кванторные операции*.

Определение 4.2.1. (Операция использования квантора общности) *Пусть P(x) – это некий предикат, заданный на множестве M. Предложение «Для любого $x \in M$ выполняется P(x)» представляет собой высказывание, которое истинно, если $I_P = M$ и ложно в противном случае. Данное предложение в символической форме записывается в виде $\forall xP(x)$, где символ $\forall$ называется квантором общности. Переход от предиката $P(x)$ к высказыванию $\forall xP(x)$ называется операцией использования квантора общности.*

Определение 4.2.2. (Операция использования квантора существования) *Пусть $P(x)$ – это некий предикат, заданный на множестве M. Предложение «Существуют $x \in M$ такие, что выполняется $P(x)$» представляет собой высказывание, которое истинно, если $I_P \neq \varnothing$ и ложно в противном случае. Данное предложение в символической форме записывается в виде $\exists xP(x)$, где символ $\exists$ называется квантором существования. Переход от предиката $P(x)$ к высказыванию $\exists xP(x)$ называется операцией использования квантора существования.*

Если предикат $P(x)$ определен на конечном множестве $M = \{a_1, a_2, \ldots, a_n\}$, то кванторные операции можно записать в следующем виде:

$$\forall xP(x) = P(a_1) \,\&\, P(a_2) \,\&\, \ldots \,\&\, P(a_n),$$

$$\exists xP(x) = P(a_1) \vee P(a_2) \vee \ldots \vee P(a_n).$$

Таким образом, кванторные операции можно рассматривать как расширение операций конъюнкции и дизъюнкции на случай произвольных (в том числе и бесконечных) множеств.

Определение 4.2.3. (Свободная и связанная переменные) *Пусть $P(x)$ – это некий предикат, заданный на множестве M. Переменную x в предикате $P(x)$ называют свободной (ей можно придавать любые значения). В высказываниях $\forall xP(x)$ и $\exists xP(x)$ переменную x называют связанной (связанной квантором общности и существования, соответственно).*

В определении кванторных операций использовался одноместный предикат. Кванторные операции можно применять и к n-местным предикатам, отдельно к каждому аргументу. При этом применение кванторной операции уменьшает местность (количество аргументов) предиката на единицу.

Например, пусть $P(x,y)$ – произвольный двуместный предикат. Тогда $\forall xP(x,y)$ – одноместный предикат, который зависит от y, а переменная x является связанной квантором общности. Выражение $\exists y \forall xP(x,y)$ является высказыванием, в котором обе переменные связаны.

Рассмотрим следующий пример. Пусть задан предикат $P : \mathrm{N} \to \{\mathbf{F}, \mathbf{T}\}$, $P(x, y)$ = «x делится на y». Высказывание $\exists y \forall xP(x,y)$ означает «Для любого y существует x такое, что x делится на y» и, очевидно, истинно. Высказывание

$\exists x \forall y P(x,y)$ означает «Существует x такое, что для любого y x делится на y» и является ложным. Данный пример показывает, что порядок применения кванторных операций к аргументам предиката является существенным и менять кванторы местами в символической записи кванторных операций нельзя.

4.3. ФОРМУЛЫ ЛОГИКИ ПРЕДИКАТОВ

По аналогии с логикой высказываний, в логике предикатов возможен переход от конкретных предикатов к общим формулам. Определение формулы логики предикатов (ФЛП) строится по схеме, схожей с определением ФЛВ.

Рассмотрим множество символов (алфавит), в котором выделим следующие группы:

1) константы – символы $a, b, ..., a_1, a_2, ...$;

2) переменные – символы $x, y, ..., x_1, y_1, ...$;

3) символы логических операций – $\neg, \&, \vee, \rightarrow, \leftrightarrow, \forall, \exists$;

4) функциональные буквы –; $f_1^1, f_2^1, f_1^2, f_2^2, ...$;

5) предикатные буквы – $P_1^0, P_1^1, P_2^1, P_1^2, P_2^2, ...$;

6) служебные символы – символы (и) и символ , ; (верхний индекс у функциональных и предикатных букв обозначает число аргументов).

Объединение множеств функциональных и предикатных букв алфавита будем в дальнейшем называть *сигнатурой.*

Определение 4.3.1. (Терм) *Последовательность символов назовем термом, если она построена по следующим правилам:*

1. Константа и переменная – терм;

2. Если f_i^n *– функциональная буква,* $t_1, t_2, ..., t_n$ *– термы, то* $f_i^n(t_1, t_2, ..., t_n)$ *– терм;*

3. Других термов нет.

Определение 4.3.2. (Формула логики предикатов) *Последовательность символов назовем формулой логики предикатов (ФЛП), если она построена по следующим правилам:*

1. Пусть P_i^n *– предикатная буква,* $t_1, t_2, ..., t_n$ *– термы (не обязательно разные). Тогда* $P_i^n(t_1, t_2, ..., t_n)$ *– ФЛП, называемая атомарной. Все переменные такой ФЛП назовем свободными, связанных переменных у этой ФЛП нет.*

2. Если F – некая ФЛП, то $(\neg F)$ *– ФЛП. Множество свободных и связанных переменных этих формул совпадает.*

3. Пусть F и G – ФЛП, причем нет переменных, которые были бы свободными в одной формуле и связанными в другой. Тогда $(F \& G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G)$ *– ФЛП. В них свободные переменные остаются свободными, а связанные – связанными.*

4. Если F – некая ФЛП, содержащая свободную переменную x, то $(\forall xF)$ *и* $(\exists xF)$ *– ФЛП. Переменная x в них становится связанной.*

5. Других ФЛП нет.

Из определения ФЛП можно заключить, что любая формула логики высказываний является формулой логики предикатов. В такой ФЛП используются только нульместные предикатные буквы (переменные в ФЛВ).

Запись вида $F(x, y, z)$ в дальнейшем будет означать что в ФЛП F входят свободные переменные x, y, z и только они. Если ФЛП не содержит свободных переменных, будем называть ее *замкнутой.*

Для упрощения записи ФЛП на практике будем применять приоритет операций, аналогичный действующему для ФЛВ, и дополненный правилом: кванторы $\forall$ и $\exists$ связывают сильнее, чем другие логические операции.

Как и в случае формул логики высказываний, формулы логики предикатов не несут никакого логического значения. Для определения значения ФЛП ее необходимо осмыслить, то есть конкретизировать переменные, константы, функции и предикаты, входящие в формулу.

Определение 4.3.3. (Интерпретация ФЛП) *Интерпретация (M, φ) формулы логики предикатов это система из множества M (называемого областью интерпретации или доменом) и отображения φ. Это отображение ставит в соответствие каждой предикатной букве некий предикат, определенный на M, каждой функциональной букве – функцию, определенную на M и со значениями в M, каждой константе – значение из M. Символы логических операций воспринимаются как обозначаемые ими операции над предикатами.*

После интерпретации ФЛП превращается в предикат, определенный на множестве M. Количество аргументов этого предиката равно количеству свободных переменных в исходной ФЛП.

Рассмотрим несколько примеров ФЛП и их интерпретаций.

1) $F(y) = \forall xP(x,y)$ – исходная ФЛП. Пусть M – множество натуральных чисел, в качестве предиката $P(x, y)$ рассмотрим предикат *«*$x \le y$*»*. Тогда исходная формула превращается в предикат $R : N \to \{\mathbf{F}, \mathbf{T}\}$, $R(y) = \forall x(x \le y)$.

2) $F = \exists xP(x, f(x))$ – исходная ФЛП. Пусть M – множество натуральных чисел, $f : x \to x^2$, в качестве предиката $P(x,y)$ рассмотрим предикат *«*$x \ge y$*»*. Тогда исходная формула превращается в выражение $\exists x(x \ge x^2)$, которое представляет собой высказывание (данное высказывание истинно).

Упражнение. Пусть задано множество $M = \mathrm{N} \cup \{0\}$ и два предиката, определенные на этом множестве: $P_1(x,y,z) = (x + y = z)$ и $P_2(x,y,z) = (x * y = z)$. Запишите формулы логики предикатов с использованием сигнатуры $\{P_1,P_2\}$, содержащие свободную переменную x, которые в данной интерпретации истинны тогда и только тогда, когда

1. $x = 0$;

2. $x = 2$;

3. x – четное число.

(Ответом на первый пункт является ФЛП $F(x) = \forall y P_1(x, y, y)$, где вместо $P_1(x, y, y)$ при интерпретации нужно использовать предикат P_1.)

4.4. ОБЩЕЗНАЧИМОСТЬ И ВЫПОЛНИМОСТЬ ФЛП

Для формул логики высказываний были определены понятия «модель», «тавтология», «противоречие», «выполнимая ФЛВ». Дадим соответствующие определения для формул логики предикатов.

Определение 4.4.1. (Модель ФЛП) *Интерпретация (M, φ) называется моделью формулы логики предикатов F, если в этой интерпретации F представляет собой выполнимый предикат.*

Интерпретация (M, φ) называется моделью множества формул логики предикатов $\{F_1, ..., F_n\}$, *если в этой интерпретации каждая* F_i *представляет собой выполнимый предикат.*

Определение 4.4.2. (Выполнимая ФЛП, выполнимое множество ФЛП) *ФЛП назовем выполнимой, если существует ее модель. Множество формул логики предикатов назовем выполнимым, если существует модель этого множества.*

Определение 4.4.3. (Тождественно истинная и тождественно ложная ФЛП) *Пусть дана некоторая ФЛП и задана ее интерпретация. ФЛП называется тождественно истинной (тождественно ложной) в данной интерпретации, если предикат,* который *она представляет в данной интерпретации, является тождественно истинным (тождественно ложным).*

Определение 4.4.4. (Общезначимая ФЛП, противоречие) *ФЛП называется общезначимой или логическим законом, если она является тождественно истинной в любой интерпретации. Назовем ФЛП F противоречием, если формула* $\neg$F *является общезначимой.*

Рассмотрим следующий пример. Пусть дана ФЛП $F(x) = \exists y P(x,y)$. Рассмотрим две интерпретации данной формулы:

a) $M = \mathrm{N}$, $P(x,y) = (x < y)$. Наша формула принимает вид предиката «Существует натуральное число такое, что x его меньше». В этой интерпретации формула оказывается тождественной истинной.

b) $M = \{1,2,3\}$, $P(x,y) = (x < y)$. Формула принимает вид предиката $R(x)$ = «Существует элемент из M такой, что x его меньше». В этой интерпретации формула оказывается выполнимой, но не тождественно-истинной ($R(1) = \mathbf{T}$, $R(2) = \mathbf{T}$, $R(3) = \mathbf{F}$).

Значит, наша формула $F(x) = \exists y P(x,y)$ не является логическим законом.

Рассмотрим еще один пример. Пусть дана ФЛП $F = \forall x(P(x) \vee \neg P(x))$. Данная формула будет тождественно истинной в любой интерпретации (докажите это!), а значит является логическим законом.

Лемма 4.4.1. *Пусть на множестве M заданы предикаты $P(x)$ и $S(x)$, причем предикат $S(x)$ является следствием предиката $P(x)$. ФЛП $P_1(x) \to P_2(x)$ является тождественно истинной в интерпретации, при которой $P_1(x) = P(x)$, $P_2(x) = S(x)$.*

Доказательство. В указанной интерпретации формула $P_1(x) \to P_2(x)$ принимает вид предиката $P(x) \to S(x)$. Множество истинности этого предиката $(M \setminus I_P) \cup Is$. Так как $S(x)$ – следствие $P(x)$, то $I_P \subset I_S$. Значит, $(M \setminus I_P) \cup I_S = M$.□

Теорема 4.4.1. *Следующие формулы логики предикатов являются общезначимыми:*

1. $\forall x P(x) \to \exists x P(x)$
2. $\exists x \forall y P(x, y) \to \forall y \exists x P(x, y)$
3. $\exists x(P_1(x) \,\&\, P_2(x)) \to (\exists x P_1(x) \,\&\, \exists x P_2(x))$
4. $(\forall x(P_1(x) \vee \forall x P_2(x)) \to \forall x(P_1(x) \vee P_2(x))$
5. $\forall x P(x) \to P(y)$
6. $P(y) \to \exists x P(x)$.

Замечание. Формула $\forall y \exists x P(x, y) \to \exists x \forall y P(x, y)$ не является общезначимой. Рассмотрим интерпретацию: M = R, $P(x,y) = (x < y)$. В данной интерпретации высказывание $\forall y \exists x P(x, y)$ истинно, а высказывание $\exists x \forall y P(x, y)$ – ложно. Исходная формула превращается в высказывание $\mathbf{T} \to \mathbf{F}$, которое ложно.

В отличие от логики высказываний, в логике предикатов проверка класса формулы не может быть выполнена простым перебором, так как множество интерпретаций формулы является бесконечным.

По аналогии с логикой высказываний введем для логики предикатов понятие логического следствия.

Определение 4.4.5. (Логическое следствие (для ФЛП)) *ФЛП G называется логическим следствием формул $F_1, F_2, ..., F_n$, если любая модель множества формул $F_1, F_2, ..., F_n$ является моделью формулы G.*

Рассмотрим пример. Докажем по определению, что $\forall x(P(x) \to S(x) \,\&\, R(x)), P(c) \models S(c)$. Пусть имеется некая интерпретация (M, φ), в которой истинны высказывания $\forall x(P(x) \to S(x) \,\&\, R(x))$ и $P(b)$, где b – это значение константы c при интерпретации. Высказывание $\forall x(P(x) \to S(x) \,\&\, R(x))$ будет истинным, в частности, и для $x = b$. Значит, истинной является импликация $P(b) \to S(b) \,\&\, R(b)$. В этом случае истинным

должно быть выражение $S(b)\,\&\,R(b)$, а значит и $S(b)$. $S(b)$ – это значение $S(c)$ при интерпретации (M, φ).

Понятие логического следствия и выполнимость в логике предикатов связаны точно так же, как и в логике высказываний.

Теорема 4.4.2. *$F_1,\ldots ,F_n \models G$ тогда и только тогда, когда множество $\{F_1, \ldots, F_n, \neg G\}$ невыполнимо.*

4.5. ЭРБРАНОВСКИЕ МОДЕЛИ

Рассмотрим выполнимое множество формул логики предикатов. Опишем специальный вид моделей этого множества.

Пусть F – некое множество формул логики предикатов. Введем следующие обозначения. Через H_0 обозначим множество символов констант, содержащихся в F. Если F не содержит констант, то поместим в H_0 произвольный символ константы, например a. Предположим, что введено множество H_i. Тогда H_{i+1} есть объединение множества H_i и термов вида $f(t_1, \ldots, t_n)$, где $t_1, \ldots, t_n \in H_i$, f – символ n-местной функции, содержащейся хотя бы в одной из формул множества F.

Определение 4.5.1. (Эрбрановский универсум) *Множество $H_\infty = H_0 \cup H_1 \cup \ldots \cup H_n \cup \ldots$ называется эрбрановским[1] универсумом множества формул F.*

Рассмотрим примеры.

1) Пусть $F = \{P(x), \neg P(x) \vee Q(f(y)), \neg Q(f(a))\}$. Тогда $H_0 = \{a\}$, $H_1 = \{a, f(a)\}$ и $H_\infty = \{a, f(a), f(f(a)), \ldots\}$.

2) Пусть $F = \{P(x), Q(x) \vee R(x)\}$. Так как множество F не содержит констант, положим $H_0 = \{a\}$. Так как формулы из F не содержат функциональных символов, то $H_0 = H_1 = \ldots = H_\infty = \{a\}$.

3) Пусть $F = \{P(x), Q(x) \vee R(x)\}$. Тогда $H_\infty = \{a, b, f(a,a), f(a,b), f(b,a), f(b,b), f(f(a,a),a), \ldots\}$

Как легко заметить, эрбрановский универсум полностью определяется набором константных и функциональных символов из множества формул F.

Определение 4.5.2. (Эрбрановский базис) *Множество атомарных формул вида $P(t_1,\ldots ,t_n)$, где $t_i \in H_\infty$, а P – n-местный предикатный символ, входящий в формулы из множества F, называется эрбрановским базисом множества F.*

В качестве примера рассмотрим эрбрановкие базисы для эрбрановских множеств, определенных ранее. В первом случае эрбрановским базисом будет множество атомарных формул вида

[1] Эрбран Ж. (Jacques Herbrand, 1909-1931)

$B = \{P(a), Q(a), P(f(a)), Q(f(a)), P(f(f(a))), ...\}$. Эрбрановским базисом для множества из примера 2 будет $B = \{P(a), Q(a), R(a)\}$. Построение эрбрановского базиса для примера 3 выполните самостоятельно в качестве упражнения.

Для решения вопроса о выполнимости множества формул логики предикатов достаточно рассматривать в качестве области интерпретации только эрбрановский универсум. Оказывается, можно ограничиться интерпретациями еще более специального вида.

Определение 4.5.3. (*H*-интерпретация) *Пусть H_∞ – эрбрановский универсум множества формул F. Назовем интерпретацию* (*M*, φ) *H-интерпретацией, если она удовлетворяет следующим условиям:*

1) *доменом интерпретации является множество H_∞, т. е. M =H_∞;*
2) *произвольной константе a из F ставится в соответствие константа a из H_∞;*
3) *произвольному функциональному символу f из F ставится в соответствие функция, переводящая любой набор элементов $t_1,... ,t_n$, $t_i \in H_\infty$ в элемент $f(t_1,... ,t_n)$;*
4) *n-местному предикатному символу P ставится в соответствие некое подмножество из H^n_∞*

Из определения *H*-интерпретации следует, что для ее задания фактически необходимо определить только некоторые подмножества (их столько, сколько разных предикатных символов в *F*) декартовых степеней множества H_∞.

4.6. РАВНОСИЛЬНОСТЬ ФЛП.

Правила построения ФЛП существенно богаче правил для построения ФЛВ. Это, в частности, означает, что вопрос о равносильности формул логики предикатов решается сложнее соответствующего вопроса для формул логики высказываний.

Определение 4.6.1. (Равносильные ФЛП) *Пусть даны две формулы логики предикатов $F(x_1,x_2, ..., x_n)$ и $G(x_1,x_2, ..., x_n)$, содержащие одинаковое множество свободных переменных. Формулы F и G называются*

1. равносильными в данной интерпретации, если в этой интерпретации они представляют равносильные предикаты;

2. равносильными на множестве M, если формулы равносильны в любой интерпретации с доменом M;

3. равносильными, если они равносильны на произвольном множестве.

Тот факт, что ФЛП *F* и *G* являются равносильными, будем обозначать $F \equiv G$.

Рассмотрим некоторые примеры

1) $F_1(x) = \forall x P(x), F_2(x) = \exists x P(x)$. Данные ФЛП являются равносильными на любом множестве из одного элемента.

2) $F_1(x,y,z) = P(x,y) \,\&\, P(x,z), F_2(x,y,z) = P(x,y) \,\&\, P(y,z)$. Рассмотрим интерпретацию, при которой $M = \{0,1\}$, $P(x,y) = (x \leftrightarrow y)$. В данной интерпретации формулы принимают вид $F_1(x,y,z) = (x \leftrightarrow y) \,\&\, (x \leftrightarrow z)$ и $F_2(x,y,z) = (x \leftrightarrow y) \,\&\, (y \leftrightarrow z)$. Очевидно, что формулы равносильны в данной интерпретации. Рассмотрим другую интерпретацию этих формул, при которой $M = \{0,1\}$, $P(x,y) = \neg x$ (y – фиктивная переменная). В этой интерпретации получаем $F_1(x,y,z) = \neg x \,\&\, \neg x = \neg x$ и $F_2(x,y,z) = \neg x \,\&\, \neg y$. В этой интерпретации формулы не равносильны.

Понятия равносильности и общезначимости для формул логики предикатов тесно связаны.

Лемма 4.6.1. *F и G – равносильные ФЛП тогда и только тогда, когда формула (F ↔ G) является общезначимой.*

Как видим, ответ на вопрос «Равносильны ли две ФЛП?» можно свести к установлению факта общезначимости некой формулы.

Приведем некоторые равносильности формул логики предикатов.

Лемма 4.6.2. *Так как любая формула логики высказываний является формулой логики предикатов, то любая равносильность логики высказываний является равносильностью логики предикатов. Более того, если в равносильных формулах логики высказываний заменить переменные на произвольные ФЛП так, чтобы в результате формулы логики высказываний преобразовались в ФЛП, то полученные ФЛП будут равносильными.*

В качестве примера рассмотрим равносильные ФЛВ $\neg(A \vee B) \equiv \neg A \,\&\, \neg B$. Заменим переменную A в формулах на ФЛП $\forall x P_1(x)$, а переменную B – на ФЛП $P_2(z,y)$. В результате мы получим две ФЛП $F_1(z,y) = \neg(\forall x P_1(x) \vee P_2(z,y))$ и $F_2(z,y) = \neg\forall x P_1(x) \,\&\, \neg P_2(z,y)$, которые равносильны: $F_1(z,y) \equiv F_2(z,y)$.

Лемма 4.6.3. (Переименование связанных переменных) *Если в ФЛП входит некая связанная переменная, то ее можно переименовать (не использовав обозначения свободных переменных), и в результате этого получится равносильная формула.*

Пусть дана формула $\forall z P_1(x,y,z) \,\&\, P_2(x,y)$. Переименуем связанную переменную z (имена x и y использовать нельзя): $z = u$. Получим формулу $\forall u P_1(x,y,u) \,\&\, P_2(x,y)$, равносильную исходной.

Лемма 4.6.4. (Перенос кванторов через отрицание) *Имеют место следующие равносильности:*

$$\neg\forall x F(x) \equiv \exists x(\neg F(x)),$$

$$\neg\exists x F(x) \equiv \forall x(\neg F(x)),$$

где F – произвольная ФЛП.

Из приведенных равенств следует, что

$$\forall x F(x) \equiv \neg \exists x (\neg F(x)),$$

$$\exists x F(x) \equiv \neg \forall x (\neg F(x)).$$

Лемма 4.6.5. *Если $F_1(x)$ – произвольная ФЛП, содержащая переменную x, F_2 – ФЛП, не содержащая x, то*

$$\forall x F_1(x) \,\&\, F_2 \equiv \forall x (F_1(x) \,\&\, F_2),$$

$$\exists x F_1(x) \,\&\, F_2 \equiv \exists x (F_1(x) \,\&\, F_2),$$

$$\forall x F_1(x) \vee F_2 \equiv \forall x (F_1(x) \vee F_2),$$

$$\exists x F_1(x) \vee F_2 \equiv \exists x (F_1(x) \vee F_2).$$

Лемма 4.6.6. (Перестановка одноименных кванторов) *Для произвольной ФЛВ F(x,y)*

$$\forall x \forall y F(x, y) \equiv \forall y \forall x F(x, y),$$

$$\exists x \exists y F(x, y) \equiv \exists y \exists x F(x, y).$$

Замечание. Как было показано на конкретном примере $\exists x \forall y F(x, y) \neq \forall y \exists x F(x, y)$.

Лемма 4.6.7. (Расширение области действия кванторов) *Если $F_1(x)$ и $F_2(x)$ – произвольные ФЛП, зависящие от x, то*

$$\forall x F_1(x) \,\&\, \forall x F_2(x) \equiv \forall x (F_1(x) \,\&\, F_2(x)),$$

$$\exists x F_1(x) \vee \exists x F_2(x) \equiv \exists x (F_1(x) \vee F_2(x)).$$

Замечание. В общем случае $\forall x F_1(x) \vee \forall x F_2(x) \neq \forall x (F_1(x) \vee F_2(x))$ и $\exists x F_1(x) \,\&\, \exists x F_2(x) \neq \exists x (F_1(x) \,\&\, F_2(x))$. В этих формулах первая часть является логическим следствием второй. Например, утверждение «У нас есть студенты, которые являются отличниками и спортсменами» более сильное, чем утверждение «У нас есть студенты, которые являются отличниками и есть студенты, которые являются спортсменами».

4.7. НОРМАЛЬНЫЕ ФОРМЫ

Используя равносильные преобразования, для каждой формулы логики предикатов можно построить некую равносильную формулу специального вида. Понятия конъюнктивной и дизъюнктивной нормальных форм для формул логики предикатов вводятся аналогично логике высказываний. Сохраняются и алгоритмы приведения к КНФ и ДНФ. В логике предикатов вводятся дополнительные нормальные формы.

Определение 4.7.1. (Предваренная нормальная форма) *Формула логики предикатов находится в предваренной нормальной форме (ПНФ), если она имеет вид*

$$Q_1 x_1 Q_2 x_2 ... Q_k x_k F ,$$

где Q_i – квантор общности или квантор существования, а формула F не содержит кванторов и содержит только операции $\neg, \&$ и $\vee$, причем операция отрицания применяется только к атомарным подформулам.

Теорема 4.7.1. *Для любой формулы логики предикатов существует равносильная формула в ПНФ.*

Доказательство. В качестве доказательства теоремы приведем алгоритм приведения произвольной формулы к ПНФ

Шаг 1. Исключить из исходной формулы эквивалентность и импликацию.

Шаг 2. Переместить отрицание к атомарным формулам.

Шаг 3. Переместить кванторы вперед, используя при необходимости переименование связанных переменных. □

Рассмотрим примеры. Пусть

$$F(x) = \exists y P_1(x, y) \vee \neg(\forall y P_2(x, y)) .$$

Данная формула не содержит операций эквивалентности и импликации. Перемесив в формуле отрицание к атомарным формулам, получим формулу:

$$F_1(x) = \exists y P_1(x, y) \vee \exists y \neg P_2(x, y) .$$

В полученной формуле переместим вперед квантор существования и получим формулу в ПНФ:

$$F_2(x) = \exists y (P_1(x, y) \vee \neg P_2(x, y)) .$$

Следующий пример. Пусть

$$F = \exists x \forall y P_1(x, y) \& \exists x \forall y P_2(x, y) .$$

Формула не содержит импликации, эквивалентности или отрицаний. Для того, чтобы применить лемму 4.6.5 и вынести $\exists x$ за скобки, переименуем во второй части конъюнкции переменную x на z:

$$F_1 = \exists x \forall y P_1(x, y) \& \exists z \forall y P_2(z, y) .$$

У полученной формулы вынесем за скобки $\exists x$, затем $\exists z$, затем $\forall y$:

$$F_2 = \exists x (\forall y P_1(x, y) \& \exists z \forall y P_2(z, y)) ,$$

$$F_3 = \exists x \exists z (\forall y P_1(x, y) \& \forall y P_2(z, y)) ,$$

$$F_4 = \exists x \exists z \forall y (P_1(x, y) \& P_2(z, y)) .$$

Последняя формула является формулой в ПНФ.

Определение 4.7.2. (Сколемовская нормальная форма) *Формула логики предикатов находится в сколемовской нормальной форме (СНФ), если она имеет вид*

$$\forall x_1 \forall x_2 ... \forall x_k F ,$$

где формула F не содержит кванторов и имеет конъюнктивную нормальную форму.

Цель приведения формулы к сколемовской нормальной форме (сколемизации, преобразования Сколема[1]) – получить формулу, которая не содержит кванторов существования.

Опишем алгоритм приведения произвольной формулы к СНФ.

Шаг 1. Привести исходную формулу к ПНФ.

Шаг 2. Бескванторную часть полученной формулы привести к КНФ.

Шаг 3. Исключить кванторы существования по следующему правилу: если в предваренной ФЛП есть квантор существования, примененный к переменной x_i которому предшествуют n кванторов общности, то переменная x_i в теле формулы заменяется на произвольную функциональную букву с аргументами, определяемыми предыдущими кванторами общности, а квантор существования и переменная из предваренной формы исключаются. Подобную процедуру проделываем для всех кванторов существования слева направо. Если квантор существования в записи формулы является первым, то соответствующая переменная меняется на незадействованный символ константы.

Рассмотрим примеры приведения формулы к СНФ. Пусть дана формула

$$\forall x_1 \forall x_2 \exists x_3 \forall x_4 F(x_1, x_2, x_3, x_4) .$$

Квантор существования применяется к переменной x_3, ему предшествуют два квантора общности, примененные к переменным x_1 и x_2. Заменим в формуле x_3 на $f(x_1,x_2)$, а выражение $\exists x_3$ исключим из предваренной формы. Получим формулу без кванторов существования:

$$\forall x_1 \forall x_2 \forall x_4 F(x_1, x_2, f(x_1, x_2), x_4) .$$

Второй пример. Рассмотрим формулу

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \forall x_5 \exists x_6 G(x_1, x_2, x_3, x_4, x_5, x_6) .$$

Выражение $\exists x_1$ удаляем из формулы, а x_1 заменяем символом-константой:

$$\forall x_2 \exists x_3 \forall x_4 \forall x_5 \exists x_6 G(a, x_2, x_3, x_4, x_5, x_6) .$$

Выражение $\exists x_3$ удаляем из формулы, а x_3 заменяем на $f(x_2)$:

[1] Сколем Т. (Thoralf Skolem, 1887-1963)

$$\forall x_2 \forall x_4 \forall x_5 \exists x_6 G(a, x_2, f(x_2), x_4, x_5, x_6).$$

И, наконец, удаляем из формулы $\exists x_6$, а x_6 меняем на $g(x_2, x_4, x_5)$:

$$\forall x_2 \forall x_4 \forall x_5 G(a, x_2, f(x_2), x_4, x_5, g(x_2, x_4, x_5)).$$

Имеет место следующая теорема.

Теорема 4.7.2. *Для любой формулы логики предикатов существует формула в СНФ одновременно выполнимая (или невыполнимая) с исходной.*

4.8. МЕТОД РЕЗОЛЮЦИЙ В ЛОГИКЕ ПРЕДИКАТОВ

Ранее был рассмотрен алгоритм метода резолюций применительно к формулам логики высказываний. Этот алгоритм допускает модификацию и применение к логике предикатов. Напомним, что данный алгоритм применяется для установления того факта, что некоторая формула является невыполнимой.

Для применения алгоритма метода резолюций исходная формула представляется в виде множества дизъюнктов. Для этого она предварительно приводится к сколемовской нормальной форме. Затем кванторы общности игнорируются (предполагается, что переменные связаны кванторами, но сами кванторы не пишутся). Для полученного множества дизъюнктов старое (из логики высказываний) правило получения резольвент уже не годится. Действительно, рассмотрим множество дизъюнктов $\{P(x), \neg P(a)\}$, где x – переменная, а a – константа. Мы не можем получить из этого множества резольвенту **F**. Однако содержательно понятно, что в этом случае нужно сделать. Так как переменная x была связана квантором общности, то можно заменить x на a. Сделав такую подстановку, получим множество $\{P(a), \neg P(a)\}$, из которого резольвента **F** выводима тривиальным образом. Дадим более четкие определения.

Определение 4.8.1. (Подстановка) *Назовем подстановкой множество равенств*

$$\sigma = \{x_1 = t_1, x_2 = t_2, ..., x_n = t_n\},$$

где $x_1, x_2, \ldots, x_n$ – различные переменные, $t_1, t_2, \ldots, t_n$ – термы, причем для любого i терм t_i не содержит переменой x_i.

Если $\sigma = \{x_1 = t_1, x_2 = t_2, ..., x_n = t_n\}$ – некая подстановка, а F – формула логики предикатов, то через $\sigma = (F)$ будем обозначать формулу, полученную одновременной заменой x_i на t_i. Например, пусть $\sigma = \{x_1 = f(x_2), x_2 = c, x_3 = g(x_4)\}$, $F = R(x_1, x_2, x_3) \vee \neg P(f(x_2))$, то $\sigma(F) = R(f(x_2), c, g(x_4)) \vee \neg P(f(c))$.

Для удобства введем пустую подстановку ε – подстановку, не содержащую равенств.

Определение 4.8.2. (Произведение подстановок) *Пусть ξ и η – две подстановки, тогда произведение подстановок ξ и η – это новая подстановка*

$\xi \circ \eta$, которая заключается в последовательном применении подстановки ξ, а затем подстановки η.

Пусть $\sigma = \{x_1 = f(x_2)\} \circ \{x_2 = c\} \circ \{x_3 = g(x_4)\}$, $F = R(x_1, x_2, x_3) \vee \neg P(f(x_2))$. Тогда $\sigma(F) = R(f(c), c, g(x_4)) \vee \neg P(f(c))$.

Определение 4.8.3. (Унификатор) *Пусть {F_1, F_2,..., F_n} – множество формул логики предикатов. Подстановка σ называется унификатором этого множества если $\sigma(F_1) = \sigma(F_2) = ... = \sigma(F_n)$. Множество формул унифицируемо, если существует унификатор этого множества.*

Если множество формул унифицируемо, то, как правило, существует несколько его унификаторов. Среди всех унификаторов множества можно выделить наибольший.

Определение 4.8.4. (Наиболее общий унификатор) *Унификатор σ множества формул называется наиболее общим унификатором этого множества, если для любого другого унификатора τ того же множества существует подстановка ξ такая, что $\tau = \sigma \circ \xi$.*

Приведем алгоритм нахождения наиболее общего унификатора множества формул *F*. Этот алгоритм называется *алгоритмом унификации.*

Шаг 1. Положить $k = 0, F_k = F, \sigma_k = \varepsilon$.

Шаг 2. Если множество F_k состоит из одной формулы, то выдать в качестве результата σ_k и завершить работу. В противном случае найти *множество рассогласований* в F_k. Для этого, просматривая все формулы из F_k слева направо, найти первую позицию в которой формулы различаются и выделить переменные и термы начинающиеся с этой позиции.

Шаг 3. Если во множестве рассогласований существует переменная ϑ_k и терм t_k, не содержащий ϑ_k, то перейти к шагу 4, иначе выдать сообщение о том, что множество F_k не унифицируемо и завершить работу.

Шаг 4. Положить $\sigma_{k+1} = \sigma_k \circ \{\vartheta_k = t_k\}$. Получить множество $F_{k+1} = \sigma(F_k)$

Шаг 5. Положить *к = к + 1* и перейти к шагу 2.

Продемонстрируем работу алгоритма на примере. Пусть $F = \{P(x, f(y)), P(a, u)\}$ (*x*, *y* и *u* – переменные). На первом проходе алгоритма будет найдена подстановка $\sigma_1 = \{x = a\}$, так как множество рассогласований – это $\{x, a\}$. Тогда $F_1 = \{P(a, f(y)), P(a, u)\}$. На втором шаге алгоритма получим подстановку $\sigma_2 = \{x = a\} \circ \{u = f(y)\}$ и $F_2 = \{P(a, f(y))\}$. Так как это множество состоит из одной формулы, то алгоритм закончит работу.

Отметим, что так как в алгоритме унификации на шаге 4 из множества формул удаляется одна переменная, то он всегда заканчивает свою работу. Приведем следующую теорему без доказательства.

Теорема 4.8.1. *Если множество формул унифицируемо, то подстановка, выдаваемая алгоритмом унификации, является наиболее общей.*

Подстановка и алгоритм унификации позволяют сформулировать определение резольвенты для формул логики предикатов.

Определение 4.8.5. (Резольвента для формул логики предикатов) *Пусть даны две формулы* $\neg P(t_1,...,t_n) \vee F$ *и* $P(s_1,...,s_n) \vee F$. *Резольвентой этих формул назовем формулу* $\sigma(F) \vee \sigma(G)$, *где* σ *– наиболее общий унификатор множества* $\{P(t_1,...,t_n), P(s_1,...,s_n)\}$.

В логике предикатов подстановка позволяет уменьшать длину дизъюнктов, используя следующее правило.

Определение 4.8.6. (Правило склейки) *Правилом склейки назовем следующее: из дизъюнкта* $\Diamond P(t_1,...,t_n)... \vee \Diamond P(s_1,...,s_n) \vee F$ *выводим дизъюнкт* $\sigma(\Diamond P(t_1,...,t_n)) \vee \sigma(F)$, *где* σ *– наиболее общий унификатор множества* $\{P(t_1,...,t_n),...,P(s_1,...,s_n)\}$, $\Diamond$ *– знак отрицания или его отсутствие (если он стоит перед одной из записанных выше атомарных формул, то он стоит и перед другими).*

Например, правило склейки, примененное к $\neg P(x,y) \vee \neg P(a,a) \vee Q(x,y,z)$, дает $\neg P(a,a) \vee Q(a,a,z)$.

На каждом шаге алгоритма метода резолюций для логики предикатов во множестве дизъюнктов ищется пара формул, которые можно резольвировать по правилу для логики предикатов или формула, к которой можно применить правило склейки. Если на некотором шаге получится резольвента **F**, то исходная формула является противоречием.

Рассмотрим примеры применения метода резолюций. Как и для логики высказываний, метод резолюций для логики предикатов обычно применяется для установления или опровержения факта следования одной ФЛП из других. При этом для получения множества дизъюнктов обычно каждую посылку и заключение отдельно преобразуют к СНФ.

Рассмотрим пример. Требуется установить при помощи метода резолюций, является ли рассуждение корректным:

$$\exists x(P(x) \,\&\, \forall y(D(y) \to L(x,y))), \forall x(P(x) \to \forall y(Z(y) \to \neg L(x,y))) \models \forall x(D(x) \to \neg Z(x)).$$

Применив приведение к СНФ посылок и заключения, образуем следующее исходное множество дизъюнктов (обратите внимание на использование разных констант при переходе к СНФ):

$$\{P(a), \neg D(y) \vee L(a,y), \neg P(x) \vee \neg Z(y) \vee \neg L(x,y), D(b), Z(b)\}.$$

Запишем получающиеся на каждом новом шаге алгоритма резольвенты:

1. $L(a,b)$ – резольвента $\neg D(y) \vee L(a,y)$ и $D(b)$ (подстановка $y = b$).

2. $\neg P(a) \vee \neg Z(b)$ – резольвента $\neg P(x) \vee \neg Z(y) \vee \neg L(x,y)$ и $L(a, b)$ (подстановка $x = a, y = b$).

3. $\neg Z(b)$ – резольвента $P(a)$ и $\neg P(a) \vee \neg Z(b)$.

4. **F** – резольвента $Z(b)$ и $\neg Z(b)$.

Так как в результате алгоритма получилось, что исходное множество дизъюнктов не является выполнимым, исходное рассуждение является корректным.

Следующий пример. Требуется установить корректность $\forall x(P(x) \to S(x)), \neg P(a) \models \neg S(a)$. Образуем исходное множество дизъюнктов:

$$\{\neg P(x) \vee S(x), \neg P(a), S(a)\}.$$

В данном множестве можно сделать тривиальную подстановку $x = a$:

$$\{\neg P(a) \vee S(a), \neg P(a), S(a)\}.$$

Попытки найти в этом множестве резольвируемые дизъюнкты терпят неудачу. Выскажем предположение, что это множество формул является выполнимым и попробуем найти его модель. Пусть доменом модели будет множество натуральных чисел, $P(x)$ = «x – четное», $S(x)$ = «x делится на 3», $a = 9$. Получим набор утверждений «9 – нечетное или делится на 3», «9 – нечетное», «9 делится на 3», каждое из которых истинно. Значит, наша интерпретация является моделью исходного множества и это множество выполнимо.

Более «экзотическую» модель множества $\{\neg P(a) \vee S(a), \neg P(a), S(a)\}$ можно построить, использовав эрбрановский универсум. В данном случае $H_\infty = \{a\}$. Зададим предикаты $P(x)$ и $S(x)$, определив их множества истинности в H_∞. Пусть $I_P = \varnothing$, $I_s = H_\infty = \{a\}$. Символ константы a интерпретируем самим собой. Получим вместо исходного множества формул набор из трех истинных высказываний, что, очевидно, означает выполнимость множества.

Как было доказано А. Черчем[1], для логики предикатов не существует универсального алгоритма, который по произвольной ФЛП устанавливает ее класс. Алгоритм метода резолюций гарантировано завершает свою работу, если исходная формула является противоречием. В противном случае алгоритм может не остановится («зациклится»).

При программной реализации метода резолюций особое внимание уделяется качественному алгоритму унификации и стратегия выбора резольвируемых предложений. При определении стратегии выбора, как правило, стремятся уйти от полного перебора дизъюнктов для поиска резольвент. Перечислим некоторые стратегии.

1. *Стратегия предпочтения одночленов.* Как нетрудно заметить, применение правила резолюции к дизъюнктам длины m и n дает резольвенту длины $m + n - 2$ (длина понимается как количество членов дизъюнкции). Эта длина будет меньше длины одного из исходных дизъюнктов, если другой дизъюнкт имеет

[1] Черч А. (Alonzo Church)

длину 1, т. е. является одночленам. Исходя из этого, стратегия предпочтения одночленов рекомендует первоочередное применение правила резолюции к одночленам.

2. *Стратегия использования подслучаев.* Назовем дизъюнкт *D подслучаем* дизъюнкта *C*, если существует подстановка σ такая, что можно *C* представить в виде $\sigma(D) \vee C_1$. Стратегия заключается в отбрасывании всех возникающих при выводе дизъюнктов *C*, если ранее были получены их подслучаи.

3. *Стратегия применения несущего множества.* Выделим в исходном множестве дизъюнктов *M* некое (по возможности большее) подмножество дизъюнктов M_1, про которое известно, что оно является выполнимым. Множество $M \setminus M_1$, назовем *несущим.* При получении резольвенты правило резолюций применяется только тогда, когда хотя бы один из дизъюнктов принадлежит несущему множеству. При этом итоговая резольвента добавляется к несущему множеству.

4.9. ИСПОЛЬЗОВАНИЕ ЛОГИКИ ПРЕДИКАТОВ ПРИ РЕШЕНИИ ЗАДАЧ

Решение прикладных задач при помощи логики предикатов обычно начинается с формализации исходных условий. Для облегчения представления условия задачи в виде формул рассмотрим таблицу 7.

Таблица 7

Формализации аристотелевских суждений классической логики

Вид суждения	**Запись в классической логике**	**Запись в виде ФЛП**
Общеутвердительное	Все *S* суть *P* (*SaP*)	$\forall x(S(x) \to P(x))$
Частичноутвердительное	Некоторые *S* суть *P* (*SiP*)	$\exists x(S(x) \,\&\, P(x))$
Общеотрицательное	Ни одно *S* не суть *P* (*SeP*)	$\forall x(S(x) \to \neg P(x))$
Частичноотрицательное	Некоторые *S* не суть *P* (*SoP*)	$\exists x(S(x) \,\&\, \neg P(x))$

Рассмотрим несколько примеров формализации.

1. Предложение на естественном языке: «Ни у одной ящерицы нет волос». Пусть $S(x)$ = «x – ящерица», $P(x)$ = «x – имеет волосы». Получаем формализацию $\forall x(S(x) \to \neg P(x))$.

2. «Все волки, кроме бешеных, боятся людей». Пусть $S(x)$ = «x – волк», P(x) = « x – бешеный», $B(x)$ = «x боится людей». Получим формализацию $\forall x((S(x) \,\&\, \neg P(x) \to B(x))$.

3. «Собаки умнее кошек». Пусть $S(x)$ = «x – собака», $P(x)$ = «x – кошка», $C(x,y)$ = «x умнее y». Рассмотрим вначале формулу $K(y) = \forall x(S(x) \to C(x,y))$, которая является отражением высказывательной формы «Все собаки умнее y». Используя данную формулу, получим требуемую формализацию: $\forall y(P(y) \to K(y)) = \forall y(P(y) \to \forall x(S(x) \to C(x,y)))$.

4. «Две змеи при встрече всегда съедают друг друга». Пусть $S(x)$ = «x –

змея», $P(x,y)$ = «x встретился с y», $C(x, y)$ = «x съедает y». Тогда требуемая формализация: $\forall x \forall y(S(x) \,\&\, S(y) \,\&\, P(x, y) \to C(x, y) \,\&\, C(y, x))$.

Основное множество задач по логике предикатов связано с проверкой правильности логических рассуждений. При этом могут использоваться различные методы, включая метод резолюций.

Задача 4.9.1. *Всякая команда, которая может победить ЦСКА, является командой высшей лиги. Ни одна команда высшей лиги не может победить ЦСКА. Значит, ЦСКА непобедима.*

Решение. Пусть $S(x)$ = «x – команда высшей лиги», $P(x)$ = «x – может победить ЦСКА». Тогда получаем, что необходимо проверить следующее логическое следствие:

$$\forall x(P(x) \to S(x)), \forall x(S(x) \to \neg P(x)) \models \neg(\exists x P(x)).$$

Применим метод резолюций. Образуем исходное множество дизъюнктов:

$$\{\neg P(x) \vee S(x), \neg S(x) \vee \neg P(x), P(a)\}.$$

Запишем получающиеся на каждом новом шаге алгоритма резольвенты:

1. $\neg P(x)$ – резольвента первого и второго дизъюнктов.

2. **F** – резольвента $\neg P(x)$ и $P(a)$ (подстановка $x = a$).

Исходное множество дизъюнктов не является выполнимым, рассуждение является корректным.

Задача 4.9.2. *Всякий, кто может решить эту задачу, математик. Иванов не может ее решить. Значит, он не математик.*

Решение. Пусть $S(x)$ = «x – математик», $P(x)$ = «x – может решить задачу». Константа a будет обозначать Иванова. Требуется установить корректность $\forall x(P(x) \to S(x)), \neg P(a) \models \neg S(a)$. Данное утверждение рассматривалось при описании метода резолюций, и было доказано, что оно не является корректным.

Следующие задачи предлагаются для самостоятельного решения.

Задача 4.9.3. *Если бы какая-нибудь команда могла обыграть ЦСКА, то и какая-нибудь команда высшей лиги могла бы обыграть ЦСКА. «Динамо» – команда высшей лиги, но не может обыграть ЦСКА. Значит, ЦСКА непобедима.*

Задача 4.9.4. *Студенты любят покушать. Некоторые студенты худые. Не все те, кто любит покушать, студенты. Значит, некоторые любители покушать не являются худыми студентами.*

Задача 4.9.5 *Розовая вода пахнет приятно. Ни одно лекарство не пахнет приятно. Если человек болен, то он должен принимать лекарство или соблюдать режим. Соблюдать режим и не принимать лекарство нельзя. Значит, если человек употребляет розовую воду, то он не болен.*

Задача 4.9.6. *Доказать методом резолюций, что если Кащей бессмертен, то он не человек.*

При формализации математических утверждений часто используются так называемые *ограниченные кванторы*. Рассмотрим утверждение «для всякого $x > 5$ существует $y > 0$ такое, что $xy = 1$». Запись данного утверждения в виде формулы $\forall x \exists y(xy = 1)$ очевидно, не является верной. Для правильного перевода надо немного изменить исходное предложение по форме (не меняя, разумеется, смысла): «для всякого x справедливо: если $x > 5$, то существует y такой, что $y > 0$ и xy=1». Правильная формализация имеет вид $\forall x((x > 5) \to \exists y((y > 0) \,\&\, (xy = 1)))$.

Если рассматривать более длинные исходные предложения, то соответствующие им формулы будут, вообще говоря, довольно громоздкими. Для того, чтобы частично избавиться от усложнения при переводе на язык логики предикатов, вводятся ограниченные кванторы.

Определение 4.9.1. (Ограниченный квантор) *Пусть $B(x)$ – формула с одной свободной переменной x. Тогда выражение $\forall B(x)$ называется ограниченным квантором общности, а выражение $\exists B(x)$ – ограниченным квантором существования.*

С помощью ограниченных кванторов предложение «для всякого $x > 5$ существует $y > 0$ такое, что $xy = 1$» можно записать довольно просто: $(\forall x > 5)(\exists y > 0)(xy = 1)$.

Более формально, ограниченные кванторы вводятся следующим образом: формула $\forall B(x)F(x)$ есть сокращение формулы $\forall x(B(x) \to F(x))$, формула $\exists B(x)F(x)$ – сокращение формулы $\exists x(B(x) \,\&\, F(x))$.

Ограниченные кванторы часто вводятся неявно. Для переменных, пробегающих множество истинности формулы $B(x)$, вводят специальное обозначение. Например, в геометрии довольно часто применяется следующее соглашение: буквами A, B, C,... обозначаются точки, буквами a, b, c, ... прямые, а буквами $\alpha, \beta, \gamma,$ ... – плоскости, т. е. с нашей точки зрения первые переменные пробегают множество истинности формулы Т(x), вторые – Пр(x), третьи – Пл(x).

Последовательное оформление этой идеи приводит к понятию *многосортной (многоосновной) логики предикатов*. Строгих определений давать не будем, укажем только отличие от обсуждавшейся ранее (одноосновной или односортной) логики предикатов. При построении формулы переменные и константы разбиты по сортам. В примере с геометрией таких сортов три: переменные, принимающие в качестве значений точки, прямые и плоскости. Далее, для каждого функционального символа указано, какой сорт имеет первый аргумент, какой – второй и т. д., какой сорт имеет значение функции. Аналогичная информация имеется и для каждого символа предиката. Для интерпретации берется не одно множество, а столько, сколько сортов переменных (эти множества на-

зываются *основами*). Для геометрии таких основ три: множество точек, множество прямых, множество плоскостей.

4.10. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Наиболее распространенная в наше время технология решения задач на компьютере состоит в том, что вначале программист должен разработать алгоритм решения задачи, а затем записать его на определенном формальном языке. Осознание того, что вычисление – частный случай логического вывода, а алгоритм – формальное задание функции, привело к идее *логического программирования*. Суть этой идеи состоит в том, чтобы компьютеру предлагать не алгоритмы, а описания предметной области задачи и саму задачу в виде некоторой аксиоматической системы, а решение задачи в виде вывода такой системы. От программиста при таком подходе требуется описать с достаточной степенью полноты предметную область и формулировку задачи на языке этой системы, а поиск вывода, приводящего к решению задачи, поручается компьютеру.

Одним из примеров языков, реализующих данную концепцию, является язык логического программирования ПРОЛОГ (Prolog). Данный язык был разработан в конце 60-х годов двадцатого века.

В ПРОЛОГЕ исходное множество фактов предметной области представляется в виде одной из следующих формул (называемых «дизъюнкты Хорна»):

1. *факт* – $p(t_1,\ldots,t_n)$;
2. *правило* – $q(s1, \ldots, s_n) : -q_1(s_1, \ldots, s_n), \ldots, q_k(s_1, \ldots, s_n)$;

Выполнение программы инициализируется следующей формулой

3. *запрос* – $?r_1(u_1, \ldots, u_n), \ldots, r_m(u_1, \ldots, u_n)$

Здесь $p(t_1, \ldots, t_n)$, $q_i(s_1, \ldots, s_n)$, $r_i(u_1, \ldots, u_n)$ – атомарные формулы логики предикатов; t_i, s_i и u_i – термы.

Факты программы представляют собой утверждения, считающиеся истинными и накапливаемые в специальной базе знаний. Переменные в фактах считаются связанными кванторами общности. Правило представляет собой следующее утверждение:

$$q_1(s_1,\ldots,s_n)\,\&\ldots\&\,q_k(s_1,\ldots,s_n)\rightarrow q(s_1,\ldots,s_n).$$

Переменные в правилах также считаются связанными кванторами общности. В запросе переменные считаются связанными кванторами существования. Запрос расценивается как логическое следствие фактов и правил. Используя метод резолюций и алгоритмы унификации, программа пытается установить логическое следствие для запроса. При этом, если запрос содержит переменные, то ищутся все возможные значения переменных, при которых запрос является логическим следствием. Набор возможных значений переменных определяется константами из фактов и правил.

Рассмотрим в качестве примера следующую программу, записанную в нотации, близкой языку ПРОЛОГ:

```
1. птица(X) :- откладывает_яйца(X), имеет_крылья(X).
2. рептилия(X) :- откладывает_яйца(X), имеет_чешую(X).
```

```
3. откладывает_яйца(ворона).
4. откладывает_яйца(питон).
5. имеет_крылья(ворона).
6. имеет_чешую(питон).
7. ? птица(ворона).
```

Первые две строки программы являются правилами, затем следуют четыре факта. Последняя строка – это утверждение-запрос, истинность которого ПРОЛОГ пытается проверить, используя метод резолюций, правила и факты. В качестве ответа данная программа выдаст строку `True`.

Рассмотрим еще один пример программы:

```
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 очевиден. В качестве целей можно задать, например, следующие:

```
5a. ? Grandparent(elizabeth, henry).
5b. ? Grandparent(elizabeth, V).
5c. ? Grandparent(U, V).
```

В первом случае проверяется утверждение для конкретных переменных (ответ программы – `True`), во втором случае программа найдет всех, для кого `Elizabeth` является бабушкой (ответ – `william, henry`), в третьем случае будет выдан список всех пар вида <бабушка (дедушка), внук (внучка)> (ответ – `<elizabeth, henry>, <elizabeth, william>`).

Логическое программирование используется в настоящее время для построения экспертных систем (устранение неполадок, управление предприятием, аналитическая медицина) и различных реализаций систем искусственного интеллекта.

5. ФОРМАЛЬНЫЕ АКСИОМАТИЧЕСКИЕ ТЕОРИИ

5.1. АКСИОМАТИЧЕСКИЕ ТЕОРИИ

Понятие «теория» определяется как набор обобщённых положений, образующих науку или раздел науки. Примерами теорий могут служить теория эволюции Дарвина, теория относительности. В любой теории можно выделить две образующие ее части. Первая часть – это система понятий, характеризующих предметную область теории. Вторая часть – система утверждений о предметной области, построенная с использованием системы понятий. Например, в теории эволюции понятиями могут служить «вид», «естественный отбор», утверждением – закон естественного отбора («Выживает сильнейший»).

Рассмотрим математические теории. Их особенностью является то, что они рассматривают абстрактные понятия. Попытка дать точное определение абстрактному понятию обречена на неудачу: одни понятия должны определяться через какие-то другие, те, в свою очередь, сами должны определяться через какие-то понятия, и так далее.

В данной ситуации выходом является описание теории как *аксиоматической*. При задании произвольной аксиоматической теории используется следующая схема:

1. *Задается некоторое множество понятий (терминов) называемых первичными*. На этом этапе выделяется одно или несколько множеств объектов теории соответствий между ними. Первичным понятиям не даются определения, а все их свойства, которыми можно пользоваться в аксиоматической теории, описываются в аксиомах.

2. *Выделяется некоторое подмножество высказываний о первичных понятиях – аксиом*. Для первичных понятий невозможно доказать все справедливые утверждения об этих понятиях. При доказательстве любого утверждения нужно опираться на какие-то предыдущие утверждения, при их доказательстве, в свою очередь, – на следующие, и так без конца. Поэтому необходимо выделить некоторые утверждения и объявить их справедливыми в данной теории. Вопрос о том, какие утверждения о первичных понятиях выбираются в качестве аксиом, заслуживает специального рассмотрения.

3. *При помощи первичных понятий даются определения всех остальных понятий*. Хотя первичные понятия не определяются, но все другие понятия, которые предполагается использовать в теории, должны быть строго определены через первичные и через понятия, смысл которых был определен раньше.

4. *На основе аксиом и определений логическим путем выводятся новые утверждения о первичных и определяемых понятиях*. Получаемые новые утверждения называются *теоремами* данной аксиоматической теории.

Подчеркнем, что выбор системы аксиом теории условен. Одно и то же утверждение теории может быть аксиомой, если оно так выбрано, а может выступать в качестве теоремы, если выбор аксиом осуществлен по-иному.

При выборе системы аксиом обычно руководствуются тем, что общее количество аксиом должно быть небольшим. В таком случае правильность аксиом будет вызывать меньшее сомнений. Еще одним желательным условием при выборе аксиом является их независимость, то есть никакая из аксиом не должна выводится из остальных. Если система аксиом зависима, это просто означает, что некоторые аксиомы системы являются лишними и могут быть «безболезненно» удалены.

Первая попытка создать аксиоматическую теорию принадлежит Евклиду (III век до нашей эры). Это планиметрия, описанная в его труде «Начала». Приводя описание планиметрии, Евклид рассматривал в качестве первичных понятия «точка», «прямая» и «лежать на». Последнее из понятий характеризовало некоторое соответствие между точками и прямыми. В качестве пяти аксиом планиметрии Евклид выбрал наиболее, *на его взгляд*, очевидные утверждения о точках и прямых.

5.2. ФОРМАЛЬНЫЕ ТЕОРИИ

Строгие аксиоматические теории первоначально разрабатывались для конкретной области математики. Однако многие математические теории оперируют утверждениями сходной структуры. Рассмотрим утверждение планиметрии: «Если одна прямая параллельна другой прямой, а эта другая прямая, в свою очередь, параллельна третьей прямой, то первая и третья прямые – параллельны». Сравним приведенное утверждение со следующим: «Если одно число равно другому, а это другое число равно третьему числу, то первое и третье числа равны». Ясно, что структура рассуждений в обоих утверждениях одинакова, хотя объекты и операции с этими объектами различны.

Для обобщения математической теории можно применить следующий подход: заменить конкретные объекты и операции над объектами на некоторые символические значения. Конкретные утверждения конкретной теории будут получаться в процессе *интерпретации* обобщенного утверждения, то есть придания смысла символам объектов и операций.

Для двух приведенных выше утверждений сформируем следующее обобщенное: «Если $a * b$ и $b * c$, то $a * c$». Приведем несколько интерпретаций данного обобщенного утверждения. Если a, b и c обозначают некие прямые, символ $*$ имеет смысл «параллельны», то обобщенное утверждение становится утверждением о параллельных прямых. Если a, b и c – числа, символ $*$ обозначает равенство, то обобщенное утверждение трансформируется в утверждение о равенстве чисел. Для получения утверждения о свойствах деления придадим символам a, b и c значения неких целых чисел, а символу $*$ – смысл операции деления и получим: «Если одно число делится на другое, а это другое – на третье, то первое число делится на третье».

Математическая теория, которая оперирует не конкретными объектами, а некими их символическими эквивалентами, называется *формальной* теорией. Если формальная теория построена как аксиоматическая, то она называется *формальной аксиоматической теорией*.

При построении формальной аксиоматической теории важно понимать, что для конкретных математических теорий символы формальной теории необходимо будет интерпретировать. Любая формальная теория допускает множество интерпретаций. Достаточно очевидным (хотя и не обязательным) является условие, при котором аксиомы формальной теории трансформируются при интерпретации в истинные утверждения. Если для некоторой интерпретации данное условие выполнено, то такая интерпретация называется *моделью* теории. Как правило, если формальная теория строится по некой реальной математической, то формализм теории выбираются так, чтобы эта реальная математическая теория была ее моделью. В идеале, одна формальная математическая теория может иметь в качестве моделей несколько реальных теорий.

В истории математики переход от аксиоматических теорий к построению и работе с формальными теориями произошел в конце XIX века. Оказалось, что формальная теория, построенная для теории множеств, может описать практически все реальные математические теории (конечно, при соответствующих интерпретациях). Однако в самой теории множеств было обнаружено несколько неразрешимых парадоксов[1]. Для выхода из данной ситуации было предложено не только формализировать объекты и операции математической теории, но также формализировать рассуждения теории (заменить интуитивную логику строгой математической) и формализировать понятия «доказательство» и «теорема». Полученная в результате данного процесса теория будет *полностью формальной*.

Наука о полностью формальных аксиоматических теориях называется *метаматематикой* (тем самым подчеркивается некий ее высший смысл по отношению к обычной математике). В полностью формальных аксиоматических теориях вывод теорем и проверка их корректности может осуществляться при помощи определенных алгоритмических процедур.

Основное требование, выдвигаемое при создании формальной аксиоматической теории, называется *полнотой* теории. Полнота означает, что любому верному утверждению реальной математической теории будет соответствовать некая теорема формальной теории. Если формальная теория оказывается не полной, то она не может описать все теоремы реальной теории, а значит существуют утверждения реальной теории, которые нельзя формально доказать. К сожалению, большинство формальных теорий являются не полными по отношению к реальным математическим теориям. Иногда создать полную формальную теорию для реальной математической просто невозможно. Так, Гедель[2]

---

[1] Один из парадоксов (парадокс Рассела) популярно излагается в виде следующей задачи: «Брадобрей бреет всех жителей деревни, которые не бреются сами. Кто бреет брадобрея?».

[2] Курт Гедель (Kurt Gödel), 1906-1978.

доказал, что для такой распространенной и часто используемой теории, как арифметика натуральных чисел, полной формальной теории создать нельзя.

5.3. ОПРЕДЕЛЕНИЕ И ПРИМЕРЫ ФОРМАЛЬНЫХ АКСИОМАТИЧЕСКИХ ТЕОРИЙ

В данном параграфе будет дано основное определение главы – определение формальной аксиоматической теории. Однако непосредственно перед этим рассмотрим несколько вспомогательных понятий.

Определение 5.3.1. (Алфавит, символ, слово) *Алфавитом будем называть любое конечное или счетное множество* $A = \{a_1,a_2,...\}$. *Элементы этого множества будем называть символами (буквами). Словом длины n в алфавите A назовем набор из n упорядоченных элементов из A.*

Дадим примечания к данному определению. Заметим, что любой счетный алфавит можно свести к конечному алфавиту, определив правило записи индексов символов. Так, бесконечный счетный алфавит $A = \{a_1,a_2,...\}$ сводится к конечному двухэлементному алфавиту $A' = \{a',\}$, при этом символ a_1 записывается как a', a_2 – как a'', и так далее. Если мы рассматриваем слово $W = (b_1, b_2, ..., b_n) \in A^n$, то далее будем записывать его в виде $W = b_1b_2 \ldots b_n$. Определим также особое *пустое слово* λ, которое не содержит символов. Множество всех слов алфавита A (включая пустое слово) будем обозначать как A^*.

Напомним определение понятия «отношение».

Определение 5.3.2. (Отношение) *Пусть M – произвольное множество. Будем говорить, что на множестве M определено n-местное отношение R, если выделено подмножество* $R \subset M^n$. *Если для некоторого упорядоченного набора* $(a_1, a_2, ..., a_n)$ *выполняется* $(a_1, a_2, ..., a_n) \in R$, *то будем говорить, что элементы* $a_1, a_2, ..., a_n$ *находятся в отношении R.*

Рассмотрим строгое определение формальной аксиоматической теории.

Определение 5.3.3. (Формальная аксиоматическая теория) *Будем говорить, что определена формальная аксиоматическая теория (ФАТ) T, если заданы*

1. алфавит A;

2. множество слов F ($F \subset A^*$*), называемых формулами;*

3. множество формул B ($B \subset F$*), называемых аксиомами теории T;*

4. множество $\{R_1, R_2, ..., R_m\}$, *где каждый элемент* R_i *является отношением на множестве формул и называется правилом вывода.*

Замечание 1. Обычно множество F задается при помощи индуктивного определения, т. е. существует способ, позволяющий проверить любое слово на принадлежность к F.

Замечание 2. Множество B может быть конечным или бесконечным. Если B – бесконечно, то оно, как правило, задается при помощи конечного множест-

ва *схем аксиом* и правил порождения конкретных аксиом из схем. Аналогичное замечание справедливо и для множества правил вывода.

Рассмотрим следующий пример. Определим теорию $\mathrm{N\,Z}$ следующим образом. Пусть

1) алфавит $A = \{0,1,2,3,4,5,6,7,8,9\}$;
2) формула – любое слово (возможно пустое);
3) множество аксиом $B = \{1,2,3,4,5,6,7,8,9\}$;
4) отношение R_1 задается условием $(X, X0) \in R_1$ где X – любая формула (таким образом, $(123,1230) \in R_1$, но $(123,1235) \notin R_1$).

Определение 5.3.4. (Непосредственный вывод формулы) *Пусть определена ФАТ T и пусть $F_1, F_2, ..., F_n, G$ – некие формулы этой теории. Если существует правило вывода R_i такое, что $(F_1, F_2, ..., F_n, G) \in R_i$, то говорят, что формула G непосредственно выводима из формул $F_1, F_2, ..., F_n$ по правилу вывода R_i. Формулы F_k называют посылками, а G – заключением.*

Непосредственный вывод обозначают следующим образом:

$$\frac{F_1, F_2, ..., F_n}{G} R_i.$$

Для теории $\mathrm{N\,Z}$ можем записать:

$$\frac{457}{4570} R_1,$$
$$\frac{000}{0000} R_1.$$

Определение 5.3.5. (Вывод формулы) *В формальной теории T выводом формулы G из формул $F_1, F_2, ..., F_n$ называется последовательность формул $E_1, E_2, ..., E_k$ такая, что, $E_k = G$ (E_k совпадает с G), а любая формула E_i $(1 \le i < к)$ является либо аксиомой, либо некой исходной формулой F_j, либо непосредственно выводима из ранее полученных формул $E_{j_1}, ..., E_{j_m}$.*

Если в теории T существует вывод формулы G из множества $F_1, F_2, ..., F_n$, то это записывается так:

$$F_1, F_2, ..., F_n \vdash_T \mathrm{G}.$$

Иногда символ T опускают, если из контекста ясно, о какой теории идет речь. Формулы $F_1, ..., F_n$ называют *гипотезами вывода*.

Рассмотрим описанную ранее теорию $\mathrm{N\,Z}$. Покажем, что $13 \vdash_{\mathrm{NZ}} 1300$. Имеем следующую последовательность формул: $E_1 = 13$ (гипотеза), $E_2 = 130$ (непосредственный вывод из E_1), $\mathrm{E}_3 = 1300$ (непосредственный вывод из E_2).

Введенные понятия позволяют дать четкое математическое определения понятия «теорема».

Определение 5.3.6. (Теорема) *Если* $\vdash_T G$, *то G называют теоремой теории T, то есть теорема – это формула, выводимая только из аксиом.*

Замечание. Теоремы о теории *T* иногда называют *метатеоремами* для избегания путаницы.

Вновь рассмотрим теорию N Z и покажем, что $\vdash_{NZ} 400$.

1. 4 – аксиома;
2. 40 – по правилу вывода;
3. 400 – по правилу вывода.

Сформулируем простейшие свойства, присущие любой формальной теории (Г обозначает произвольное множество формул, *A*, *B*, *C* – некоторые формулы).

1. Г, $A \vdash A$.
2. Если Г, A, $B \vdash C$, то Г, B, $A \vdash C$.
3. Если Г $\vdash A$, то Г, $B \vdash A$.
4. Если Г, $A \vdash B$ и Г $\vdash A$, то Г $\vdash B$. В частности, если Г, $A \vdash B$ и $\vdash A$, то Г $\vdash$ B.
5. Если Г $\vdash A_i$ $(i = \overline{1,m})$ и $A_1, A_2, ..., A_m \vdash B$, то Г $\vdash B$.

Рассмотрим несколько примеров ФАТ, которые будем использовать в дальнейшем. Определим теорию $\mathbf{T}_1$ следующим образом:

1) алфавит A = {А, Б, В, ..., Э, Ю, Я};
2) $F = A^*$ (формулой является любое слово);
3) $B = A \cup \lambda$ (аксиомы – любая буква и пустое слово);
4) правило вывода

$$\frac{X}{\xi X \xi} R,$$

где $X \in F$ – любая формула, $\xi \in A$ – произвольный символ алфавита. Теоремами теории $\mathbf{T}_1$ будут слова, читающиеся одинаково слева направо и наоборот.

Рассмотрим формальную аксиоматическую теорию $\mathbf{T}_2$:

1) алфавит теории – множество $\{x, y, z, +, \times, (,)\}$;
2) формулой является любое слово;
3) аксиомы – формулы *x*, *y*, *z*;
4) правила вывода

$$\frac{\varphi \quad \psi}{(\varphi + \psi)} R_1 \text{ и } \frac{\varphi \quad \psi}{(\varphi \times \psi)} R_2,$$

где φ и ψ – любые формулы. Данная теория позволяет в качестве теорем получать простейшие алгебраические выражения. Покажем, что $\vdash ((x + y) \times (y + z))$:

1.*x*	аксиома
2.y	аксиома
3.(*x*+*y*)	по правилу вывода R_1 из 1 и 2

4.z	аксиома
5.$(y + z)$	R_1 2, 4
6.$((x + y) \times (y + z))$	R_1 3, 5

Теории, подобные $\mathbf{T}_1$ и $\mathbf{T}_2$, называются *формальными грамматиками*, поскольку обычно их назначение – задать правила построения слов некоторого формального языка. Формальные грамматики успешно применяются при описании синтаксиса языков программирования. Например, можно описать формальную грамматику, теоремами которой будут синтаксически правильные программы на языке Pascal.

5.4. ИНТЕРПРЕТАЦИЯ, ПОЛНОТА И НЕПРОТИВОРЕЧИВОСТЬ ФАТ

Как было сказано выше, любая формальная аксиоматическая теория оперирует некими символами, не привязанными к конкретным объектам. Простейший способ осмысления формальной теории – это установление связи между формулами (элементами формальной теории) и элементами некоего множества в «реальном мире».

Определение 5.4.1. (Интерпретация ФАТ) *Пусть дана некая формальная аксиоматическая теория T. Интерпретацией (в широком смысле) формальной аксиоматической теории T назовем связывание формул теории с элементами некоего множества M, называемого областью интерпретации.*

Рассмотрим, например, формальную теорию $\mathbf{T}_1$ и множество *M*, состоящее из всех слов русского языка (для простоты будем считать, что слова вида «ЪЪЪ» или «ЫРЪЖД» тоже являются русскими). Установим связь между формулами $\mathbf{T}_1$ и русскими словами следующим тривиальным образом: будем считать формулу соответствующим русским словом. Таким образом мы определили некую интерпретацию формальной теории $\mathbf{T}_1$.

Рассмотрим другую интерпретацию теории $\mathbf{T}_1$. В качестве области интерпретации выберем натуральные числа. Формуле из $\mathbf{T}_1$ поставим в соответствие натуральное число по следующему правилу: символ А обозначает цифру 1, символ Б – цифру 2, ..., символ З – цифру 9, остальные символы обозначают цифру 0. При такой интерпретации формула ПАПА перейдет в натуральное число 0101 = 101 (кстати, формула ЛАПА тоже перейдет в число 101).

Приведенные примеры позволяют утверждать, что произвольная формальная теория допускает множество самых различных интерпретаций.

Особенностью человеческого мышления является то, что в нем не находятся объекты «реального мира». Мышление оперирует только утверждениями о таких объектах. В связи с этим уместна аналогия с некой границей или экраном, окружающем «реальный мир». Данный экран содержит все мыслимые утверждения (истинные или ложные) обо всех объектах реального мира:

1. Моя машина синего цвета.
2. 2+2=4.
3. …

Зафиксируем в «реальном мире» некоторое множество M, и рассмотри утверждения только об объектах этого множества. Пусть, например, M – множество всех слов русского языка. Тогда утверждениями об M будут:

1. В слове «мама» 4 буквы.
2. Слово «потоп» – палиндром.
3. Слово «папа» оканчивается на букву «к».
4. …

Мы рассматриваем все утверждения, поэтому среди них есть как истинные, так и ложные. Мы можем продолжить данный процесс, выделив среди всех утверждений о множестве M класс утверждений одного вида:

1. Слово «потоп» – палиндром.
2. Слово «арбуз» – палиндром.
3. ...

В данном случае класс утверждений можно описать так: утверждения вида «слово F – палиндром», т. е. утверждения о палиндромах.

Более узкое понятие интерпретации для формальной аксиоматической теории предполагает сопоставление формул и утверждений об элементах некоего множества.

Определение 5.4.2. (Интерпретация ФАТ) *Пусть дана некая формальная аксиоматическая теория T. Интерпретацией формальной аксиоматической теории T назовем совокупность двух элементов* (M, φ)*, где M – некое множество, называемое областью интерпретации, а* φ *– отображение, которое связывает любую формулу из T с утверждением некоторого класса об элементах множества M.*

Вновь рассмотрим формальную теорию $\mathbf{T}_1$. Построим ее интерпретацию следующим образом. В качестве M возьмем множество слов русского языка. В качестве класса утверждений об элементах M рассмотрим утверждения вида «слово F – палиндром». Установим связь между формулами $\mathbf{T}_1$ и утверждениями: $F \ni F \mapsto$ «слово F – палиндром». При данной интерпретации мы можем говорить, что формуле ПОТОП из $\mathbf{T}_1$ соответствует истинное утверждение, формуле ПАПА соответствует ложное утверждение.

Определение 5.4.3. (Выполнимая формула ФАТ) *Пусть выбрана некая интерпретация формальной аксиоматической теории T. Формулу F назовем выполнимой в данной интерпретации, если при данной интерпретации формуле F соответствует истинное утверждение.*

Определение 5.4.4. (Модель множества формул) *Интерпретация называется моделью множества формул* Γ *формальной аксиоматической теории T, если любая формула из* Γ *оказывается выполнимой в данной интерпретации.*

Для любой формальной теории представляют интерес прежде всего такие интерпретации, при которых аксиомы теории оказываются истинными. Введем следующее определение.

Определение 5.4.5. (Модель ФАТ) *Интерпретация называется моделью формальной аксиоматической теории T, если любой аксиоме теории при этой интерпретации соответствует истинное утверждение.*

Как правило, различные модели одной формальной аксиоматической теории принято классифицировать в зависимости от мощности области интерпретации. Так, говорят о *конечных моделях* теории, *счетных моделях* и т. п.

В рассмотренном выше примере интерпретация формальной теории $\mathbf{T}_1$ являлась ее моделью, так как любая буква является палиндромом (будем считать пустое слово палиндромом по определению).

Важно понимать следующее. Серьезные формальные аксиоматические теории начинают строить, отталкиваясь от утверждений о неких объектах (т. е. теория строится *для* каких-то понятий, а не понятия получаются по теории). В этом случае на роль аксиом логично выбрать формализованные *истинные* утверждения. Естественно, желательно, чтобы количество аксиом было малым (или их множество просто описывалось). Правила вывода для теории также строят так, чтобы они сохраняли истинность.

Еще раз вернемся к рассмотренной интерпретации (модели) теории $\mathbf{T}_1$. Данная модель обладает интересным свойством: не только каждой теореме из $\mathbf{T}_1$ соответствует истинное высказывание, но и для любого истинного утверждения из класса утверждений интерпретации (для любого палиндрома русского языка) найдется в теории соответствующая теорема. Данное свойство формальной теории называется *полнотой*.

Определение 5.4.6. (Полнота ФАТ) *Формальная аксиоматическая теория T называется полной в интерпретации* (M, φ)*, если данная интерпретация является моделью теории и любому истинному утверждению (из класса интерпретации) об объектах множества M соответствует некая теорема из T.*

Свойство полноты теории позволяет говорить о том, что теория полностью описывает («накрывает») все истинные утверждения о некоторых объектах. Свойство полноты – «хорошее» свойство теории, отвергающее существование в классе утверждений таких, которые являются истинными, но не доказуемыми.

5.5. ИСЧИСЛЕНИЕ ВЫСКАЗЫВАНИЙ

Как было указано выше, в полной формальной аксиоматической теории формализуются не только объекты, но и рассуждения о них. Изучением структуры рассуждений занимается логика. Если формализовать логику, то есть построить для логики формальную теорию, то мы получим основу для построения на базе этой теории любой другой формальной теории. Для логики обычно рассматривают формализации логики высказываний, затем на этой базе строят формализации для логики предикатов.

Для логики высказываний было предложено несколько формализации. Мы рассмотрим формализацию Алонзо Черча, называемую исчисление высказываний Черча.

Определение 5.5.1. (Исчисление высказываний) *Исчисление высказываний* $\mathcal{L}$ *– это формальная аксиоматическая теория, в которой*

1. алфавит – набор из символов $\neg, \to, (,), a, b, a_1, ...,$ *где первые два символа называют связками, следующие два – служебными символами, остальные – переменными;*

2. формулы строятся по следующим правилам:

a) *переменная является формулой;*

b) *если A и B формулы, то* $\neg A$, $(A \to B)$ *– формулы;*

3. аксиомы задаются при помощи следующих схем (A, B, C обозначают любые формулы):

A1: $A \to (B \to A)$

A2: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

A3: $(\neg A \to \neg B) \to (B \to A)$

4. имеется единственное правило вывода (A и B – любые формулы)

$$\frac{A,\ A \to B}{B}\ MP.$$

Дадим некоторые комментарии. И аксиомы, и правило вывода *MP* задаются при помощи схем, так что фактически $\mathcal{L}$ обладает бесконечным набором и аксиом, и правил вывода. Обозначение правила вывода как *MP* связано с его латинским названием *Modus ponens* – правило отделения. При записи формул из $\mathcal{L}$ будут опускаться скобки для внешней импликации (для более «эстетичного» вида записи).

Может возникнуть естественный вопрос: по какому принципу введены аксиомы и правило вывода теории? Еще раз напомним, что хотя создатель теории вправе выбирать составляющие ее элементы произвольным образом, желательным является условие полноты теории в той (хотя бы в той) интерпретации, для которой теория оформлялась. Как мы увидим в дальнейшем, исчисление высказываний является полной теорией для логики высказываний. Однако в данный момент нас пока не интересует конкретный смысл символов теории, мы оперируем с ними формально.

Рассмотрим некоторые примеры выводимых в $\mathcal{L}$ формул. Покажем, что $\vdash A \to A$.

1. $A \to ((A \to A) \to A)$	A1; $A \to A\|B$
2. $(A \to ((A \to A) \to A)) \to (A \to (A \to A)) \to (A \to A)$	A2; $A \to A\|B, A\|C$
3. $(A \to (A \to A)) \to (A \to A)$	MP 1, 2
4. $A \to (A \to A)$	A1; $A\|B$
5. $A \to A$	MP 4, 3

В данном выводе записи в правой части означают следующее. Запись вида MP 1, 2 означает применение правила вывода к формулам в строках 1 и 2. Обозначение A2; $A \to A \mid B, A \mid C$ расшифровывается как использование второй схемы аксиом, где вместо B подставляется формула $A \to A$, а вместо C – формула A.

Итак, формула $A \to A$ является теоремой теории $\mathcal{L}$. Вывод данной теоремы (доказательство) остается справедливым, если под символом A понимать произвольную формулу. Именно этим оправдано обозначение $A \to A$, а не, допустим, $b \to b$ (b – переменная).

Покажем, что в теории $\mathcal{L}$ выводимо $A \vdash B \to A$.

1. A — гипотеза
2. $A \to (B \to A)$ — A1
3. $B \to A$ — MP 1, 2

Всякую доказанную выводимость можно использовать далее как новое, производное правило вывода (при необходимости доказательство выводимости может быть повторено в произвольном месте вывода). Последняя доказанная выводимость называется *правилом введения импликации*:

$$\frac{A}{B \to A}\ (\to^{+}).$$

5.6. ТЕОРЕМА ДЕДУКЦИИ ИСЧИСЛЕНИЯ ВЫСКАЗЫВАНИЙ

В теории $\mathcal{L}$ имеется метатеорема, которая позволяет уменьшать количество гипотез при рассмотрении вывода некоторой формулы.

Теорема 5.6.1. (Теорема дедукции) *Если* $\Gamma, A \vdash B$, *то* $\Gamma \vdash A \to B$ *и обратно.*

Следствие 5.6.1. *Если* $A \vdash B$, *то* $\vdash A \to B$ *и обратно.*

Следствие 5.6.2. $A \to B, B \to C \vdash A \to C$.

Данное следствие задает новое правило вывода, которое называется *правилом транзитивности*.

Следствие 5.6.3. $A \to (B \to C), B \vdash A \to C$.

Это правило называется *правилом сечения*.

Введем для краткости следующие обозначения:

$$A \vee B := \neg A \to B, \qquad A \,\&\, B := \neg(A \to \neg B).$$

Приведем список из некоторых производных правил вывода теории $\mathcal{L}$:

1) $A \vdash A \vee B$ – правило введения дизъюнкции ($\vee^{+}$).
2) $A, B \vdash A\&B$ – правило введения конъюнкции ($\&^{+}$).
3) $A\&B \vdash A$, $A\&B \vdash B$ – правила исключения конъюнкции ($\&^{-}$).
4) $A \vdash \neg\neg A$ – правило введения двойного отрицания.

Набор подобных правил облегчает вывод конкретных теорем исчисления высказываний.

5.7. ОСНОВНЫЕ СВОЙСТВА ИСЧИСЛЕНИЯ ВЫСКАЗЫВАНИЙ

Как отмечалось ранее, исчисление высказываний создавалось для формализации логики высказываний. Более точно, исчисление высказываний задает формальные правила для порождения тавтологий логики высказываний, то есть универсально верных суждений.

Рассмотрим следующую интерпретацию исчисления высказываний, называемую *естественной интерпретацией*. Сопоставим формуле исчисления высказываний изображаемую ей ФЛВ. Истинными будем считать формулы, изображающие тавтологии.

Опишем некоторые свойства теории L, применительно к ее естественной интерпретации.

Теорема 5.7.1. *Естественная интерпретация формальной теории L является ее моделью.*

Доказательство. Любая аксиома теории L при естественной интерпретации соответствует тавтологии логики высказываний, что проверяется непосредственно. □

Заметим, что правило вывода MP из двух тавтологий получает третью (сохраняет тавтологичность). Следовательно, любая теорема теории L представляет тавтологию в логике высказываний.

Как уже отмечалось ранее, исчисление высказываний позволяет представить любую тавтологию логики высказываний в виде теоремы. Таким образом:

Теорема 5.7.2. *В своей естественной интерпретации формальная теория L является полной.*

В алфавите исчисления высказываний имеется символ ($\neg$), который будучи применен к некой формуле *F* меняет ее смысл при интерпретации на противоположный (например, если *F* при интерпретации оказывалась тавтологией, то $\neg$F будет при интерпретации противоречием).

Теорема 5.7.3. *В своей естественной интерпретации формальная теория L является непротиворечивой, то есть не существует такой формулы F из L, что одновременно $\vdash F$ и $\neg F$.*

Доказательство. Все теоремы L при естественной интерпретации переходят в тавтологии. Отрицание тавтологии не есть тавтология. Следовательно, в L не выводимы одновременно теорема и ее отрицание. □

5.8. ИСЧИСЛЕНИЕ ПРЕДИКАТОВ

Опишем формальную аксиоматическую теорию для логики предикатов. При этом будем использовать лишь необходимый минимум символов, аксиом и правил вывода.

Определение 5.8.1. (Исчисление предикатов) *Исчисление предикатов* $\mathcal{PL}$ *– это формальная аксиоматическая теория, в которой*

1. алфавит – набор из символов

a) $\neg, \rightarrow$ *– связки*,

b) *(,) – служебные символы*,

c) $\forall, \exists$ *– кванторы*,

d) $x, y, ..., x_1, y_1, ...$ *– переменные*,

e) $a, b, ..., a_1, b_1, ...$ *– константы*,

f) $f, g, ...$ *– функторы*,

g) $P, Q, ...$ *– предикаты*;

2. формулы строятся по следующим правилам:

a) *<формула> := <атом>/*$\neg$*<формула>/(<формула>*$\rightarrow$*<формула>)/*

$\forall$ *<переменная> <формула>/*$\exists$ *<переменная> < формула>*

b) <атом> := <предикат>(<список термов>)

c) <список термов> := <терм>/<терм>, <список термов>

d) <терм> := <константа>/<переменная>/<функтор> (<список термов>)

при этом используются понятия свободной и связанной (квантором) переменной

3. аксиомы задаются при помощи следующих схем:

A1: $\mathrm{A} \rightarrow (\mathrm{B} \rightarrow \mathrm{A})$

A2: $(\mathrm{A} \rightarrow (\mathrm{B} \rightarrow \mathrm{C})) \rightarrow ((\mathrm{A} \rightarrow \mathrm{B}) \rightarrow (\mathrm{A} \rightarrow \mathrm{C}))$

A3: $(\neg \mathrm{A} \rightarrow \neg \mathrm{B}) \rightarrow (\mathrm{B} \rightarrow \mathrm{A})$

A4: $\forall x \mathrm{A}(x) \rightarrow \mathrm{A}(t)$

A5: $\mathrm{A}(t) \rightarrow \exists x \mathrm{A}(x)$

где терм t является свободным для переменной x

4. имеется три правила вывода:

$$\frac{A, A \rightarrow B}{B} MP$$

$$\frac{B \rightarrow A(x)}{B \rightarrow \forall x A(x)} \forall^{+}$$

$$\frac{A(x) \rightarrow B}{\exists x A(x) \rightarrow B} \exists^{+}.$$

Обратите внимание на то, что теория $\mathcal{PL}$ содержит теорию $\mathcal{L}$ в качестве подтеории.

Как и для исчисления высказываний, для исчисления предикатов можно привести примеры теорем и выводимых формул. В исчислении предикатов остается верной метатеорема о дедукции.

Естественной интерпретацией формальной теории $\mathcal{PL}$ является такая, при которой формуле из $\mathcal{PL}$ ставится в соответствие изображаемая ей формула логики предикатов. При этом истинными будем считать логические законы, то есть общезначимые формулы логики предикатов. Верны следующие метатеоремы:

Теорема 5.8.1. *Любая теорема теории $\mathcal{PL}$ является общезначимой формулой логики предикатов.*

Теорема 5.8.2. (Теорема Геделя о полноте $\mathcal{PL}$) *Любая общезначимая формула логики предикатов является теоремой формальной теории $\mathcal{PL}$.*

Теорема 5.8.3. *Формальная теория $\mathcal{PL}$ в своей естественной интерпретации является непротиворечивой.*

5.9. ТЕОРИИ ПЕРВОГО ПОРЯДКА. ФОРМАЛЬНАЯ АРИФМЕТИКА

Как отмечалось ранее, любая реальная математическая теория строится на базе формальной теории для логики, например, на базе исчисления предикатов. При этом к аксиомам исчисления предикатов, которые называются *логическими*, добавляются *собственные* аксиомы теории. Аналогичным образом дополняются правила вывода. Алфавит теории конкретизируется, то есть набор предикатных символов, констант и функторов рассматривается в связи с последующей интерпретацией теории. Если в формальной теории разрешено применять кванторы только к переменным, а не к функторам или предикатам, то такая формальная теория называется *формальной теорией первого порядка*. Практика показывает, что теорий первого порядка оказывается достаточно для формализации во всех разумных случаях.

Рассмотрим примеры некоторых теорий первого порядка (при этом логическая часть теории – аксиомы и правила вывода – отдельно не описывается).

I. Теория равенства ε. Это формальная аксиоматическая теория первого порядка, в которой имеются:

1. собственный двухместный предикат = (вместо $=(x, y)$ пишут $(x = y)$);
2. собственные схемы аксиомы:

E1: $\forall x \quad (x = x)$

E2: $(x = y) \to (A(x, x) \to A(x, y))$.

Приведем примеры некоторых теорем теории ε.

1. $\vdash t = t$ для любого терма t. Данная теорема легко выводится из аксиом **E1** и **A4** по правилу *MP*.

2. $\vdash x = y \to y = x$.

3. $\vdash x = y \to (y = z \to x = z)$.

II. Формальная арифметика $\mathcal{AR}$. Опишем формальную теорию для арифметики натуральных (более точно, $\mathrm{N} \cup \{0\}$), чисел.

В алфавите формальной арифметики имеются символы для обозначения переменных, служебные символы и символы логических операций. Отдельно выделены: символ для предметной константы a_1 (при интерпретации эта константа будет изображать число 0), один одноместный функциональный символ f_1^1 (при интерпретации он будет обозначать операцию перехода к следующему числу), два двухместных функциональных символа f_1^2 и f_2^2 (при интерпретации будут соответствовать операциям сложения и умножения), одна предикатная буква P_1^2 (обозначающая отношение равенства). Для наглядности будем далее записывать: 0 вместо a_1, t' вместо $f_1^1(t)$, $t + u$ вместо $f_1^2(t, u)$, $t \times u$ вместо $f_2^2(t, u)$, $t = u$ вместо $P_1^2(t, u)$ и $t \neq u$ вместо $\neg P_1^2(t, u)$.

Множество собственных аксиом формальной арифметики $\mathcal{AR}$ состоит из девяти аксиом:

AR1: $(x_1 = x_2) \to ((x_1 = x_3) \to (x_2 = x_3))$

AR2: $(x_1 = x_2) \to (x_1' = x_2')$

AR3: $0 \neq x_1'$

AR4: $(x_1' = x_2') \to (x_1 = x_2)$

AR5: $x_1 + 0 = x_1$

AR6: $x_1 + x_2' = (x_1 + x_2)'$

AR7: $x_1 \times 0 = 0$

AR8: $x_1 \times x_2' = (x_1 \times x_2) + x_1$

AR9: $A(0) \to (\forall x(A(x) \to A(x')) \to \forall x A(x))$

где первые восемь аксиом заданы явно, а девятая (аксиома индукции) является схемой, в которой $A(x)$ – произвольная формула.

Правила вывода формальной теории $\mathcal{AR}$ – это три правила вывода теории $\mathcal{PL}$ и одно собственное правило, называемое *правило индукции*:

$$\frac{A(0), \forall x(A(x) \to A(x'))}{\forall x A(x)} I.$$

Естественной интерпретацией теории $\mathcal{AR}$ будем считать арифметику натуральных чисел. Функциональные и предикатные символы будем понимать так, как указано выше. При этом аксиомы теории приобретают следующий содержательный смысл:

AR1: Свойство равенства.

AR2: За каждым натуральным числом непосредственно следует точно одно натуральное число.

AR3: Ноль не следует ни за чем.

AR4: Каждое натуральное число непосредственно следует не более чем за одним натуральным числом.

AR5, AR6: Рекурсивное определение сложения.

AR7, AR8: Рекурсивное определение умножения.

AR9: Индукция.

Аксиомы теории $\mathcal{AR}$ специально выбраны так, чтобы все обычные факты, верные в арифметике натуральных чисел и формулируемые на языке $\mathcal{AR}$, были бы теоремами теории $\mathcal{AR}$.

Рассмотрим следующие примеры формул из теории $\mathcal{AR}$:

1. $F_1(x, y) = \exists z(x + z = y)$. Данная формула определяет следующий двухместный предикат $P(x, y) = (x \le y)$.

2. $F_2(x, y) = (x \le y)\&\neg(x = y)$. Данная формула определяет предикат $P(x, y) = (x < y)$.

3. $F_3(x) = \exists y(x = y + y)$. Формула задает предикат $P(x) =$ «x – четное число».

4. $F_4(x, y) = (\neg(x = 0))\&\exists z(x \times z = y)$. Это формальная запись утверждения «число x делит число y».

5. $F_5(x) = (\neg(x = 0))\&(\neg(x = 0'))\&\forall y(F_4(y, x) \to ((y = x) \vee (y = 0')))$. Данная формализация представляет собой запись предиката «число x – простое».

6. $\vdash_{AR} \forall x(F_5(x) \to \exists y((x < y)\&F_5(y)))$. Данная теорема теории $\mathcal{AR}$ представляет собой теорему арифметики «Множество простых чисел бесконечно».

Достаточно сложно привести пример истинного в арифметике утверждения, которое не являлось бы теоремой теории $\mathcal{AR}$. Тем не менее такие утверждения есть! Первый, довольно искусственный и громоздкий пример принадлежит Геделю (1931 год). Следовательно, описанная формальная теория $\mathcal{AR}$ не является полной по отношению к арифметике натуральных чисел.

ЛИТЕРАТУРА

1. Ахо А., Хопкрофт Дж., Ульман Дж., Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
2. Верещагин Н. К., Шень А., Вычислимые функции. М.: МЦНМО, 1999.
3. Грин Д., Кнут Д., Математические методы анализа алгоритмов. М. Мир, 1987.
4. Кнут Д., Искусство программирования для ЭВМ. Т. 1 – 3. М. Мир, 1976-1978.
5. Непейвода Н. Н., Прикладная логика. Ижевск: Изд-во Удм. ун-та, 1997.
6. Eitan M. Gurari, An Introduction to the Theory of Computation. Computer Science Press, 1989.
7. Sedgewick R, Algorithms, Addison-Wesley Publsh., 1984.