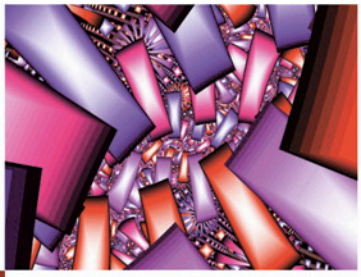
С. Лавров

bhv
www.bhv.ru
www.bhv.kiev.ua

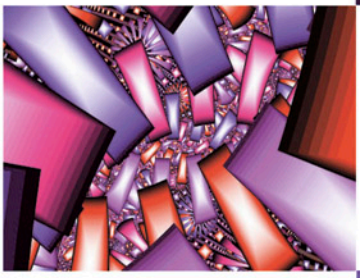
ПРОГРАММИРОВАНИЕ
МАТЕМАТИЧЕСКИЕ ОСНОВЫ, СРЕДСТВА, ТЕОРИЯ

МАСТЕР

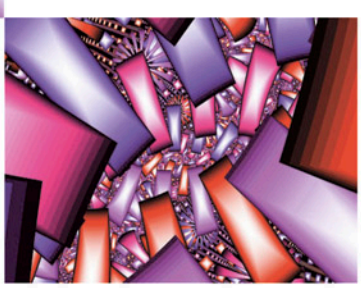
УЧЕБНОЕ ПОСОБИЕ



Математическая логика
Теория множеств
Теория вычислимости



Основные понятия и конструкции языков программирования



Анализ свойств программ

Святослав Лавров

Программирование.
Математические основы, средства, теория

Санкт-Петербург
«БХВ-Петербург»
2002

УДК 681.3.06

Современное программирование излагается как искусство заставить компьютер решить задачу, возникшую перед человеком. Даны единые основания математики и программирования, краткие сведения из области графов, теории вероятностей и информации (в ее математическом толковании). Приведены основные понятия и конструкции современных языков программирования. Рассмотрен ряд вопросов теории программирования с упором на математическую семантику языковых конструкций.

Для студентов и преподавателей вузов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Ангелины Лужиной*
Зав. производством	*Николай Тверских*
Фото на обложке	*Адама Бартоса*

Оригинал-макет подготовлен С. С. Лавровым

Лавров С. С.
Программирование. Математические основы, средства, теория. — СПб.: БХВ-Петербург, 2002. — 320 с.: ил.

ISBN 5-94157-069-4

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 26.02.02.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 24,51.
Доп. тираж 3000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9 линия, 12.

Содержание

Введение

Современное программирование, понимаемое как искусство описать задачу, возникшую перед человеком, и заставить машину (компьютер) ее решить, остро нуждается как в средствах описания способа решения задачи, так и в средствах формализации понятий и связей между ними в различных областях. Как и многие другие виды человеческой деятельности, оно может быть уподоблено сооружению, нуждающемуся в солидном фундаменте, надежных стенах и, хотя бы легкой, крыше. Каждому из этих элементов в книге выделено по главе, их краткие заголовки вынесены в ее название.

Под фундаментом программирования понимаются его математические основы — глава 1. На самом деле математика и программирование выстроены на общем фундаменте, складывающемся из таких дисциплин, как математическая логика, теория множеств и теория вычислений. Их дополняют краткие сведения из области графов, вероятностей и информации (в ее математическом смысле).

Фундамент оставляет, как ему и положено, впечатление чего-то солидного и незыблемого. В какой-то степени так оно и есть. Но покоится этот фундамент не на скальном основании, а на зыбком грунте естественного языка, общечеловеческих представлений и здравого смысла. Грунтовые воды делают свое дело, заставляя все возведенное на нем пошатываться, а временами — разрушаться. Об особенностях и последствиях этого процесса также рассказывается в книге. Однако у автора (как, по-видимому, у всех пишущих на эти темы) нет ответа на некоторые возникающие по ходу изложения вопросы. Иногда это оговаривается явно, иногда сомнительные высказывания, оценки или эпитеты берутся в кавычки, иногда автор обходит такие темы молча.

А в какой мере математика вообще способна послужить основой для чего бы то ни было? Р. Фейнман в своем знаменитом курсе физики, говоря о связи физики с другими науками, выразился так: «. . . на замечательной связи, объединяющей физику с математикой, мы не задержимся. (Математика, с нашей точки зрения, не наука в том смысле, что она не относится к *естественным* наукам. Ведь мерило ее справедливости отнюдь не опыт.) Кстати, не все то, что не наука, уж обязательно плохо. Любовь, например, тоже не наука.»

Стены программирования со встроенным в них оборудованием — это сложившиеся в нем понятия и конструкции — глава 2. Вместе взятые, они для множества людей создают условия для выпуска весьма разнообразной продукции. По бокам постоянно вырастают новые корпуса. А в старом центральном здании трудятся люди, которые хранят проверенные жизнью традиции и в меру сил передают молодежи свой опыт и знания. Но хорошо хранить верность традициям, лишь пока они не окостенели, не начали мешать творчеству, не стали бедствием сродни оголтелому новаторству.

Крыша над зданием призвана защитить людей, в нем находящихся и работающих, хотя бы от непогоды. При стихийных бедствиях ни крыша, ни стены не помогают. Ураганы срывают крыши над корпусами, при наводнениях вода заливает низко расположенные помещения, землетрясения разрушают здания

целиком, особенно если они возведены недобросовестными строителями, к тому же — наспех.

Люди ищут защиту в науке (по Фейнману — не естественной), в теории — глава 3, придающей их деятельности солидность или хотя бы видимость солидности. Но с наукой временами случаются казусы, при которых ее служителям приходится объяснять не то, чему была посвящена их деятельность, а причины ее неудач. Крыши, увы, имеют обыкновение протекать.

В этих заметках местами ведется довольно длинный разговор вокруг хорошо известных, классических, формул. При всем почтении к формулам следует придавать гораздо большее значение понятийному контексту, в котором те или иные формулы получены и могут быть применены. Сами формулы легко найти в справочнике. Контекст же в работах, написанных в «академическом» стиле, часто лишь скупо проговаривается, а то и подразумевается (принимаем такую-то систему аксиом или такие-то допущения — и все). А его надо знать и понимать. Использование формул вне породившего их контекста приводит, как говорится, к непредсказуемым последствиям.

Если после всего сказанного читатель не утратил желание поближе познакомиться с книгой, то — Бог в помощь.

Книга написана по материалам курсов лекций, читавшихся в 70-х – начале 80-х годов на математико-механическом факультете Ленинградского (в то время) государственного университета. Раздел (глава) 2 представляет собой существенную переработку работы [27]. Глава 3 — менее существенную переработку вышедшей небольшим тиражом книги [28].

Книга делится на разделы трех уровней. Разделы первого уровня часто называются главами, как это уже было сделано выше. Ссылки на формулы внутри разделов даются по их номерам. Ссылка вида (1.2.3–4) относится к формуле (4) из раздела 1.2.3.

Пунктуация во фрагментах программ и некоторых других формальных текстах следует правилам используемого языка, а не русской грамматики. Конец доказательства теоремы или леммы отмечен знаком $\lhd$.

Считаю своим приятным долгом поблагодарить В. Тумасониса, чьи замечания помогли значительно улучшить изложение главы 2, Р. И. Подловченко за многочисленные полезные замечания по главе 3, членов кафедры МО ЭВМ мат.-мех ф-та СПбГУ, содействовавших своим интересом написанию этой книги, среди них особо — М. В. Дмитриеву за отчаянные попытки сломить сопротивление чиновников из издательства СПбУ к изданию главы 3 в виде отдельной книги, а также В. П. Котлярова и его сотрудников (СПбГТУ) за внимание к работе и за содействие к появлению в конце концов издания [28].

Но, может быть, более всего я должен благодарить терпеливых слушателей моих лекций. А у бывших студентов-первокурсников матмеха ЛГУ я, вероятно, должен просить прощения за все несовершенства курса математической логики, который я им читал из года в год.

1. Математические основы

Здесь охвачены лишь начальные сведения из некоторых математических дисциплин с некоторым уклоном в сторону математической логики (причины объяснены во Введении, в остальном учитывались потребности последующих глав).

1.1. Формальные языки

1.1.1. Неформальный взгляд на формализацию

Математика состоит из ряда математических дисциплин или *теорий*. Некоторые из них так и называются: «теория чисел», «теория вероятностей» и т. п. Примерами, более знакомыми читателю, могут служить элементарная арифметика и элементарная геометрия, изучаемые в средней школе. Их мы и будем использовать ниже в качестве источников иллюстративных примеров.

Математическую теорию можно строить с разной степенью формализации. Первоначально математические понятия возникают как абстракции явлений реального мира. Если у Иры, Оли и Кати в руках по яблоку, то всего яблок столько же, сколько девочек. Совокупность всех упомянутых здесь яблок имеет некоторое общее свойство с совокупностью девочек. Примерно так люди пришли к понятию числа «три» и вообще к понятию натурального (положительного) числа, а заодно и к (может быть, неявному) понятию совокупности.

Сходным образом можно пояснить понятие сложения натуральных чисел (переход от чисел, характеризующих две совокупности каких-либо реальных объектов, к числу, характеризующему объединенную совокупность) и другие арифметические понятия. В геометрии понятие точки возникло как идеализация колышка, забитого в землю, или следа ножки циркуля на листе бумаги. К понятию прямой можно прийти, отвлекаясь почти от всех реальных свойств натянутой веревки или луча света, попадающего от далекой звезды в глаз наблюдателя.

Между прочим, проверка того, насколько близка форма этого луча к идеальной прямой, отличается ли сумма углов реального треугольника космических размеров от 180°, короче — соблюдаются ли аксиомы геометрии в реальном мире, требует постановки сложнейших физических экспериментов. Не слишком ли суров был Фейнман в своих оценках? Любовь прекрасна — спора нет. Но, может быть, математика еще прекраснее? А программирование? . .

Построение и истолкование математической теории, когда каждое понятие более или менее явно подразумевает некоторые явления окружающей нас действительности, называется *содержательным*.

При таком построении за любым математическим утверждением видится совокупность разнообразных, но в чем-то сходных ситуаций реального мира. Например, формуле «$3+2=5$» может соответствовать вопрос: «У Коли есть три книги, а у Саши — две, сколько книг у них вместе?» и ответ: «Пять книг». Ссылки на подобные примеры не только допустимы, но и весьма желательны.

В школьной математике мы знакомимся и с чисто формальными методами, например, с формальными приемами преобразования выражений на уроках алгебры.

Так, в цепочке равенств

$$(a+b)(a-b) = a(a-b) + b(a-b) = a^2 - ab + ba - b^2 = a^2 - b^2 \tag{1}$$

использован, во-первых, целый ряд равенств, выражающих свойства чисел и операций над ними, как-то:

$$\begin{cases} (x+y)z = xz + yz, & (x+y) + z = x + (y+z), \\ x - y = x + (-1)\cdot y, & x + y = y + x, \\ x = 1 \cdot x, & n + (-n) = 0, \\ xy = yx, & x \cdot 0 = 0, \\ xx = x^2, & x + 0 = x, \end{cases} \tag{2}$$

где вместо x, y и z могут подразумеваться любые выражения, а вместо n — любое число. Во-вторых, применен общий принцип подстановки: если выражение A содержит в себе некоторое другое выражение C (в частности, A может совпадать с C), а выражение B получается из A заменой C на равное ему выражение D, то выражения A и B равны. В роли выражений C и D могут выступать левая и правая (или наоборот) части любого из равенств (2), а в роли A — любое правильно построенное выражение. В-третьих, цепочка равенств

$$K = L = M = N$$

представляет собой сокращенную запись совокупности равенств

$$K = L,\ L = M,\ M = N.$$

В-четвертых, неявно использован еще один принцип: если $A = B$ и $B = C$, то $A = C$. Его двукратное применение позволяет прийти к равенству

$$(a+b)(a-b) = a^2 - b^2. \tag{3}$$

Упражнение 1.1.1. Опираясь на равенства (2), обосновать равенства:

а. $x(y+z) = xy + xz$,
б. $(x-y)z = xz - yz$,
в. $x - x = 0$.

Упражнение 1.1.2. Выписать опущенные шаги цепочки (1), связанные с применением равенств $(x+y) + z = x + (y+z)$ и $x + 0 = x$.

Мы не зря так подробно остановились на этом простом примере — при построении и исследовании формальных теорий, чему посвящена бóльшая часть главы 1, очень важно помнить о тех элементарных шагах преобразований формул, которые обычно подразумеваются, и уметь их восстанавливать. Кроме того, необходима именно такая степень детальности, хотя бы на некоторых шагах.

Формализация математической теории связана, прежде всего, с отказом от употребления естественного, русского или иного, языка, таящего в своих глубинах немало рифов. Например, слова «в руках по яблоку» можно понять и так: «в каждой руке по яблоку». Выбор между этими вариантами читатель должен

был сделать на основе последовавшего за этим утверждения автора — для логики вещь чудовищная.

Вместо слов «от перемены мест слагаемых сумма не меняется» пишется $x + y = y + x$. Утверждение «две прямые, порознь параллельные третьей, параллельны между собой» записывается в виде $p\|r \wedge q\|r \Rightarrow p\|q$. Но и формализованный язык должен обладать своей структурой — системой и иерархией обозначений. Так, буквами (x, y, p, ...), возможно с индексами и штрихами (например, χ_1'), обычно обозначаются объекты (числа, прямые и пр.) теории. Некоторые знаки ('+', '$\cdot$', ...) обозначают операции над этими объектами, другие ('=', '$\|$' и т. п.) — отношения между ними, третьи ('$\wedge$', '$\Rightarrow$' и др.) помогают из простых утверждений образовывать более сложные. Выражения — последовательности символов — могут обозначать объекты, возникающие в результате применения операций к другим объектам ($x + y$ — сумма чисел x и y, $\rho(A, B)$ — расстояние между точками A и B, AB — прямая, проведенная через точки A и B и т. д.), утверждения относительно этих объектов ($x = y$, $p \| q$, $n = 0$, ...), а также комбинации или последовательности таких утверждений. Пример цепочки равенств (1) показывает, что довольно длинная цепь рассуждений может быть представлена одним формальным выражением. Правило подстановки — это иллюстрация того, как некоторый способ рассуждения

> «Воспользуемся равенством $(x + y)z = xz + yz$ для случая, когда x обозначает объект a, y — объект b, а z — объект $(a - b)$. Получим $(a + b)(a - b) = a(a - b) + b(a - b)$.»

заменяется правилом формального преобразования выражений.

Далее, на время проведения подобных преобразований можно забыть, что буквы x, y, p, q обозначают числа или прямые, и тем более — что в понятии числа или прямой отражаются какие-то свойства окружающего мира. Однако важно, что за каждой формальной теорией стоит ее содержательный прообраз. Его факты и закономерности, разумеется, должны отражаться как в выражениях, принимаемых за исходные, так и в правилах их преобразования. Да и сам этот прообраз должен обладать хоть какой-то научной ценностью — ведь совсем нетрудно, например, строго формализовать правила игры в домино.

Занимаясь формальным преобразованием выражений, мы вправе и даже обязаны не задумываться ни над смыслом исходных, промежуточных и окончательных выражений, ни над обоснованностью правил, по которым одни выражения получаются из других. Как только мы обращаемся к смыслу, мы рискуем внести в наши формальные преобразования что-то, не предусмотренное правилами формальной теории.

Так в многовековой истории попыток вывести аксиому параллельности из других аксиом геометрии Евклида известен ряд случаев, когда на основе «здравого смысла» в доказательствах использовались утверждения, эквивалентные этой аксиоме. Разумеется, такое «доказательство» — уже не формальное доказательство (нельзя даже сказать, что оно ошибочно, так как при формальном подходе нельзя говорить о правильности или ошибочности доказательства, можно лишь отличать доказательства от недоказательств).

В чистом виде ни содержательный, ни формальный подход в математике почти не встречаются. Чисто содержательны, пожалуй, лишь математические таблицы, вроде таблиц логарифмов или простых чисел (при всей их внешней бессодержательности и строго формальном виде). Если понятие логарифма уже освоено и воспринято, то информация «$\log 2 = 0.30103$» становится просто фактом реального мира. Чисто формальны некоторые кусочки этой книги, если извлечь их из контекста. В остальном же при построении математических теорий оба подхода сопутствуют друг другу.

Естественно, что математики тяготеют к формальному построению теории, ибо на замене содержательных понятий абстрактными и обозначении их символами основан весь путь развития математики и все достигнутые ею успехи. С другой стороны, любая фраза, начинающаяся словами «Легко видеть» или «Отсюда следует», апеллирует к пониманию читателя, к его навыкам во владении формальным аппаратом, а это уже не укладывается в рамки строгой формализации. Работы, формализованные до конца, были бы чрезвычайно раздуты и совершенно не читаемы. Наконец, математик обязан соотносить свои теории с действительностью (пусть эта действительность ограничивается кругом деятельности его коллег), если он не хочет уподобить занятие наукой раскладыванию пасьянсов.

Чем же все-таки полезна формализация, почему она плодотворна в математике? То, что проделано по формальным правилам, легко поддается проверке. Если все этапы последовательных формальных преобразований выдержали проверку, то достоверность связи между исходными и окончательными выражениями сомнений не вызывает. Споры между математиками не должны возникать, если только автор доказательства может сослаться на правила, по которым выполнены все шаги доказательства. Нельзя спорить и по поводу самих правил или исходных выражений теории, ибо формальный подход исключает необходимость ссылок на смысл или истинность этих правил и выражений. Обсуждать можно только применимость той или иной математической теории к конкретной ситуации реального мира (например, применимость планиметрии к вычислению площадей протяженных участков земной поверхности; см. также пример, приводимый несколько ниже). Но некоторые математики считают, что и это не должно их интересовать, и упрекнуть их в чем-нибудь трудно.

Ну и, наконец, — это уже аргументация второй половины XX века — без строжайшей формализации и регламентации было бы невозможно прибегнуть к помощи вычислительных машин ни в точных науках, ни во многих других областях человеческой деятельности.

Одной и той же формальной теории могут быть даны различные содержательные истолкования. Совсем тривиальный пример нам дает арифметика, в которой считать можно все, что угодно: яблоки, книги, звезды, ворон и т. д. Равенство (3) оказывается справедливым при любой содержательной интерпретации обозначений a и b и операций сложения и умножения, если только в этой интерпретации истинны все равенства (2). В частности, оно справедливо в области как целых, так и рациональных, действительных или комплексных чисел.

С некоторыми оговорками оно справедливо и при некоторых других интерпретациях.

Пример. В качестве объектов рассматриваются векторы и действительные числа, обозначаемые буквами. Под сложением и вычитанием понимаются сложение и вычитание векторов либо чисел (не допуская, конечно, сложения вектора с числом), а под умножением — либо умножение чисел, либо умножение числа на вектор (или наоборот), либо скалярное (но не векторное) умножение векторов (однако, в равенстве $(xy)z = x(yz)$ по крайней мере одна из букв x, y или z должна обозначать число). Знаки операций воспринимаются в зависимости от контекста, как и символ '0', обозначающий либо число, либо нулевой вектор. Можно убедиться, что любое из равенств (2) при этих соглашениях справедливо.

Упражнение 1.1.3. Доказать, что в параллелограмме с заданными диагоналями произведение сторон на косинус угла между ними не зависит от угла между диагоналями. (Указание: обозначить векторы-диагонали через $2a$ и $2b$).

Известно много примеров, когда разные физические явления описываются одинаковыми математическими уравнениями. Это означает, что одна и та же математическая теория (исследующая свойства решений этих уравнений) допускает разные физические интерпретации.

Но, может быть, еще чаще встречаются случаи интерпретации одной математической теории внутри другой. Так, вся аналитическая геометрия — это, по сути дела, геометрическая интерпретация теории алгебраических уравнений первого и второго порядка. Достойно внимания то, что сама геометрия, включая теорию конических сечений, возникла за много веков до ее аналитической сестры, где понятию конического сечения соответствует термин «кривая второго порядка». Не только в этом случае наглядные представления раньше становятся объектом научного исследования, чем эквивалентные им формальные средства, в конечном счете быстрее приводящие к цели. Методы математического анализа применяются в геометрии, теории чисел и других математических дисциплинах.

Некоторые дисциплины (например, функциональный анализ, а также теория множеств и математическая логика, с элементами которых мы будем знакомиться в этом разделе) первоначально и возникли ради возможности интерпретации их результатов в других разделах математики. Короче говоря, обнаружилось, что математические теории, допускающие лишь одну интерпретацию, составляют скорее исключение, чем правило (даже формализованная арифметика имеет нестандартные интерпретации, существенно отличающиеся от привычной арифметики натурального ряда). Все это стимулировало развитие математических дисциплин в XX веке именно как формальных теорий, не связанных с какой-либо содержательной интерпретацией.

Наконец, заменив математические утверждения и рассуждения выражениями — последовательностями символов, мы получаем возможность сделать сами эти рассуждения (доказательства) предметом математической теории и, как уже было сказано, предметом компьютерной обработки (автоматическое доказательство теорем и т. п.). Математическая логика — это и есть такая теория. Она

исследует общие свойства формальных теорий, их возможности и ограничения, а также связи между формальными теориями и их содержательными интерпретациями, трактуя, впрочем, последние тоже достаточно абстрактно. Начиная со следующего раздела, наше изложение будет становиться все более формальным.

1.1.2. Алфавиты, слова, языки

Формальные теории, не пользуясь естественным языком, нуждаются в собственном формальном языке, на котором записываются встречающиеся в них выражения. Опишем простые средства, используемые для этой цели.

Алфавит — это конечное множество $A = \{a_1, \ldots, a_n\}$, элементы которого называются *буквами* или *символами*. Любая конечная последовательность букв $b_1 \ldots b_k$, где $b_i \in A$ для $i = 1, \ldots, k$ называется *словом* в алфавите A. Слово может состоять всего лишь из одной буквы, а может и вообще не содержать букв, тогда оно называется *пустым* и обозначается λ. Число букв в слове φ называется его *длиной* и обозначается $|\varphi|$. В частности, $|\lambda| = 0$.

Над словами определена операция *сцепления*. Результатом сцепления слов $\varphi = b_1 \ldots b_k$ и $\psi = c_1 \ldots c_m$ является слово $b_1 \ldots b_k\, c_1 \ldots c_m$. Его мы будем обозначать $\varphi\psi$ (другими словами, для операции сцепления не будем применять никакого специального знака, подобно тому, как в школьной алгебре произведение x на y обычно обозначается просто xy, а не $x \cdot y$ и не $x \times y$). Очевидно, что $|\varphi\psi| = |\varphi| + |\psi|$.

Слова $\varphi = b_1 \ldots b_k$ и $\psi = c_1 \ldots c_m$ будем называть *равными*, если $k = m$ и $b_i = c_i$ для $i = 1, \ldots k$, т. е. когда всюду на соответствующих местах в словах φ и ψ стоят одинаковые буквы. Равенство слов φ и ψ будем обозначать $\varphi = \psi$ (предполагая, что алфавит A не содержит буквы $=$, если это не так, то вместо нее здесь надо воспользоваться иной буквой). Слова $\varphi = b_1 \ldots b_k$ и $\psi = c_1 \ldots c_m$ не равны (обозначение $\varphi \neq \psi$), если условие их равенства не выполнено, т. е. если $k \neq m$, или $k = m$, но при некотором i $(1 \leq i \leq k)$ $b_i \neq c_i$.

Очевидно, что для любого слова φ справедливы равенства $\varphi\lambda = \varphi$ и $\lambda\varphi = \varphi$, и для любых трех слов φ, ψ и χ — равенство $(\varphi\psi)\chi = \varphi(\psi\chi)$, где скобки обозначают последовательность выполнения операций сцепления. Последнее свойство (ассоциативность операции сцепления) позволяет записывать результат двух и более сцеплений без скобок, например, $\varphi\psi\chi$ или $\varphi_1\varphi_2 \ldots \varphi_s$.

Множество всех слов в алфавите A (включая пустое слово) обозначается A^*. Любое подмножество множества A^* называется *языком* в алфавите A, а совокупность правил, позволяющих установить принадлежность слова языку, — *грамматикой* этого языка. Иногда такую грамматику называют *распознающей*, в отличие от *порождающей* грамматики, позволяющей выписывать произвольные слова данного языка и только их. Обычно требуется, чтобы как порождение, так и распознавание (задача более трудная) осуществлялись *эффективно*, т. е. за конечное (и желательно — не слишком большое, соразмерное длине слова) число шагов.

Если слово ψ представлено в виде $\psi = \varphi\chi$, то слово φ называется *началом*, а слово χ — *концом* слова ψ при данном представлении. Если слово ψ представлено в виде $\psi = \chi_1\varphi\chi_2$, где χ_1 и χ_2 — произвольные (возможно, пустые) слова, то говорят, что слово φ *входит* в слово ψ, а само представление слова ψ в указанном виде называется *вхождением* слова φ в слово ψ. Вхождения $\psi = \chi_1\varphi\chi_2$ и $\psi = \chi_1'\varphi\chi_2'$ *различны*, если $\chi_1 \neq \chi_1'$.

Упражнение 1.1.4. Доказать, что если вхождения $\psi = \chi_1\varphi\chi_2$ и $\psi = \chi_1'\varphi\chi_2'$ различны, то $|\chi_1| \neq |\chi_1'|$.

Упражнение 1.1.5. Если $\psi = \chi_1\varphi\chi_2$ и $\psi = \chi_1'\varphi\chi_2'$, то либо слово χ_1 — начало слова χ_1', либо слово χ_1' — начало слова χ_1 (либо то и другое вместе).

Упражнение 1.1.6. Сколько существует различных вхождений пустого слова в слово длины n?

Из утверждения упражнения 1.1.4 следует, что для задания вхождения слова φ в слово ψ (если оно существует) достаточно указать место вхождения, например, номер первой буквы слова φ в слове ψ или длину предшествующего этой букве начала слова ψ.

Вхождение $\psi = \chi_1\varphi\chi_2$ называется *более левым*, чем вхождение $\psi = \chi_1'\varphi\chi_2'$, если $|\chi_1| < |\chi_1'|$. Вхождение φ в ψ называется *самым левым*, если не существует другого, более левого, вхождения φ в ψ. Говорят, что вхождения $\psi = \chi_1\varphi\chi_2$ и $\psi = \chi_1'\varphi\chi_2'$ слова φ в слово ψ *пересекаются*, если $|\chi_1| \leq |\chi_1'| < |\chi_1| + |\varphi|$ или $|\chi_1'| \leq |\chi_1| < |\chi_1'| + |\varphi|$ (иначе говоря, если $\big||\chi_1'| - |\chi_1|\big| < |\varphi|$). Например, слово РОКОКО содержит два пересекающихся вхождения слова ОКО — одно с $\chi_1 =$Р, $\chi_2 =$КО, второе — с $\chi_1' =$РОК, $\chi_2' = \lambda$.

Вхождением буквы a в слово ψ называется вхождение слова φ, состоящего из одной буквы a, в ψ.

Упражнение 1.1.7. Доказать, что никакие два различные вхождения буквы a в слово ψ не пересекаются.

1.1.3. Структура формальных теорий

Формальная теория определяется заданием четырех ее элементов: *алфавита*, множества *формул*, множества *аксиом* и совокупности *правил вывода*. Мы начали описание формальной теории фразой, написанной на естественном языке. Нет ли здесь противоречия? Разумеется, нет. Теория — это аппарат, предназначенный, в частности, для использования человеком. Правила пользования этим аппаратом должны быть изложены на языке, хорошо понятном человеку. Такой язык называется *метаязыком* или *языком метатеории* по отношению к собственно теории. Метаязык и сам может быть формализован — люди, знакомые с описанием некоторых языков программирования, хорошо это знают. Но пока ограничимся простым русским языком, стараясь не прибегать к смысловым и лингвистическим изыскам и вывертам. О языке логических теорий речь пойдет в разд. 1.2, а сейчас изложим общие принципы действия аппарата.

Множество $\mathbf{F}$ формул формальной теории — это язык в алфавите $\mathbf{A}$ этой теории. Порождающая грамматика этого языка должна быть достаточно простой и эффективной, настолько простой, чтобы и распознавание слов, являющихся формулами, трудностей не представляло.

Фигурные буквы: $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, ... используются в метаязыке для обозначения формул. Их можно снабжать индексами или иными пометками. Разумеется, эти буквы, как и любые обозначения метаязыка, не должны принадлежать алфавиту формальной теории, и когда говорится «формула $\mathcal{A}$», то имеется в виду не формула, состоящая из одной буквы $\mathcal{A}$, а формула теории, обозначенная этой буквой. В элементарной математике слова «число x», «точка A» понимаются аналогично.

Множество $\mathbf{G}$ аксиом формальной теории — это некоторое подмножество множества ее формул. Аксиомы, если их число конечно, задаются списком, если нет — то какими-либо схемами или правилами, позволяющими эффективно распознавать аксиомы среди прочих формул.

Правило вывода должно давать возможность по данному набору формул $\mathcal{A}_1, \ldots, \mathcal{A}_m$ и формуле $\mathcal{B}$ установить, находятся ли эти $m+1$ формула в определенной связи (в определенном отношении) между собой. Если да, то говорят, что формула $\mathcal{B}$ является *непосредственным следствием* формул $\mathcal{A}_1, \ldots, \mathcal{A}_m$ (непосредственно *выводима* из них) по этому правилу. Число m этих формул, называемых *посылками* правила, обычно фиксировано для каждого правила, но иногда допускаются правила с произвольным числом посылок. Формула $\mathcal{B}$ называется *заключением* правила. Совокупность $\mathbf{R}$ правил вывода формальной теории должна быть конечной.

Правила вывода записывают в виде

$$\mathcal{A}_1, \ldots, \mathcal{A}_m \vdash \mathcal{B}$$

или же в виде

$$\frac{\mathcal{A}_1, \ldots, \mathcal{A}_m}{\mathcal{B}}.$$

При желании можно рассматривать аксиомы как частные случаи правил вывода — правила, у которых $m = 0$, и говорить, что аксиома — это непосредственно выводимая формула (из пустого набора посылок).

При содержательной интерпретации формальной теории формулы соответствуют утверждениям, аксиомы — исходным утверждениям, которые могут быть признаны содержательно истинными, правила вывода — схемам рассуждений, позволяющих из одних истинных утверждений получать другие, также истинные. Теперь нам надо определить понятие, которое в рамках формальной теории соответствует содержательному понятию доказательства — последовательности утверждений, приводящих от исходных утверждений, принимаемых за истинные, к их логическим следствиям. На каждом шаге доказательства используется либо одна из аксиом, либо один из приемов рассуждений, формально представленных в виде правил вывода. Утверждение, полученное в конце этого процесса, и является итогом доказательства — теоремой. Так мы приходим к следующему определению.

Выводом формулы $\mathcal{B}$ в формальной теории $\mathbf{T}$ называется конечная последовательность формул $\mathcal{B}_1, \ldots, \mathcal{B}_n$, где $\mathcal{B}_n = \mathcal{B}$, а каждая из формул $\mathcal{B}_i$ для $i = 1, \ldots n$ — это либо аксиома формальной теории, либо непосредственное следствие каких-либо предыдущих формул последовательности (т. е. формул, содержащихся среди $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$) по одному из правил вывода.

Если рассматривать аксиомы как правила вывода с пустым набором посылок, то определение вывода можно сформулировать короче, требуя лишь, чтобы каждая из формул вывода была непосредственно выводима из каких-либо предшествующих ей в этом выводе формул.

Формула $\mathcal{B}$ формальной теории $\mathbf{T}$ называется *выводимой* (в этой теории), если существует ее вывод в теории $\mathbf{T}$. Выводимые формулы называются также *теоремами* рассматриваемой формальной теории.

Утверждение, что формула $\mathcal{B}$ выводима в теории $\mathbf{T}$, т. е. является ее теоремой, кратко записывается в виде

$$\vdash_{\mathbf{T}} \mathcal{B}$$

или, если из контекста ясно, о какой теории идет речь, — в виде

$$\vdash \mathcal{B}.$$

Чтобы не возникали сомнения, идет ли речь о теоремах формальной теории или о теоремах метауровня, будем эти *метатеоремы* нумеровать, всегда использовать эти номера, ссылаясь на них, а заглавное слово «теорема» (а также «лемма» и «следствие») набирать вразрядку.

Пример. Рассмотрим формальную теорию $\mathbf{L}_1$ с алфавитом $\mathbf{A}=\{a, b\}$, формулами которой являются любые слова в этом алфавите, аксиомами — формулы $\mathcal{A}_1 = a$, $\mathcal{A}_2 = b$, $\mathcal{A}_3 = aa$ и $\mathcal{A}_4 = bb$, и в которой действуют два правила вывода:

$$\mathcal{A} \vdash a\mathcal{A}a \tag{4}$$

и

$$\mathcal{A} \vdash b\mathcal{A}b. \tag{5}$$

Легко убедиться, что теоремами этой теории являются любые непустые слова — палиндромы (перевертыши) в алфавите $\mathbf{A}$, одинаково читающиеся слева направо и справа налево, и только они. Например, формула, т. е. слово, $abbabba$ имеет вывод, состоящий из четырех формул (*шагов* вывода):

$\mathcal{B}_1 = a$ (аксиома $\mathcal{A}_1$),
$\mathcal{B}_2 = bab$ (правило (5) при $\mathcal{A} = \mathcal{B}_1$),
$\mathcal{B}_3 = bbabb$ (правило (5) при $\mathcal{A} = \mathcal{B}_2$),
$\mathcal{B}_4 = abbabba$ (правило (4) при $\mathcal{A} = \mathcal{B}_3$).

В дальнейшем подобные выводы будем записывать в сокращенном виде, не прибегая к обозначениям вида $\mathcal{B}_i$, заменяя их в пояснениях к шагам номером i шага, на котором формула получена, и обходясь без слов «аксиома», «правило» и «при». Для данного примера:

1) a $(\mathcal{A}_1)$
2) bab $(1; (5))$

3) $bbabb$ (2; (5))
4) $abbabba$ (3; (4))

(список посылок правила предшествует номеру правила).

Этот пример может вызвать некоторые недоуменные вопросы. Какие утверждения изображают формулы теории $\mathbf{L}_1$? Почему утверждения, представленные аксиомами, следует считать истинными? Каким схемам рассуждений соответствуют правила этой теории?

Естественный ответ на первый вопрос таков — формула $\mathcal{A}$ изображает утверждение: «$\mathcal{A}$ — палиндром». При этом аксиомы, действительно, соответствуют истинным утверждениям: a — палиндром, ..., bb — палиндром и содержат полный список палиндромов длиной не свыше 2. В правилах вывода представлены простые умозаключения: если $\mathcal{A}$ — палиндром, то как $a\mathcal{A}a$, так и $b\mathcal{A}b$ — палиндромы.

Среди утверждений вида «$\mathcal{A}$ — палиндром», где $\mathcal{A}$ — слово в алфавите a, b, встречаются как истинные (например $aaaaa$ — палиндром), так и ложные ($aabb$ — палиндром). Если утверждение «$\mathcal{A}$ — палиндром» истинно, то, как уже отмечалось, формула $\mathcal{A}$ выводима в теории $\mathbf{L}_1$, и наоборот — если $\vdash_{\mathbf{L}_1} \mathcal{A}$, то это утверждение истинно.

Это важное свойство формальной теории — выводимость всех формул, изображающих содержательно истинные утверждения, и только таких формул — называется ее *полнотой*.

Итак, теория $\mathbf{L}_1$ из нашего примера полна (по отношению к интерпретации, которая была ей дана).

Упражнение 1.1.8. Построить полную формальную теорию $\mathbf{L}_2$ с алфавитом $\mathbf{A}_2 = \{x, y, z, \text{`}+\text{'}, \text{`}\times\text{'}, \text{`}(\text{'}, \text{`})\text{'}\}$, любая теорема $\mathcal{A}$ которой допускает интерпретацию: «$\mathcal{A}$ — правильно построенное арифметическое выражение, в котором числа обозначаются буквами x, y, z, в качестве операций используются лишь сложение и умножение, порядок исполнения операций определяется только скобками». Построить вывод формулы $((x + y) \times (y + z))$ в теории $\mathbf{L}_2$.

Теории, подобные $\mathbf{L}_1$ или $\mathbf{L}_2$, называются *формальными грамматиками*, поскольку их назначение — задать правила построения слов некоторого формального языка (языка палиндромов или языка термов). Роль формальных грамматик как в логике, так и в программировании очень существенна. Фрагменты грамматики некоего модельного языка программирования рассыпаны по главам 2 и 3, где они то дополняют, то сменяют друг друга.

Наряду с утверждениями, доказываемыми (в содержательном изложении теории) безусловно, часто проводятся условные доказательства, имеющие силу лишь в предположении истинности некоторых дополнительных утверждений (условий доказываемой теоремы). Формальным аналогом таких условных доказательств являются *выводы из гипотез*.

Списком (или *совокупностью*) *гипотез* называется некоторая конечная последовательность формул $\mathcal{C}_1, \ldots, \mathcal{C}_k$. Такие списки обычно обозначаются прописными греческими буквами Γ, Δ, Ξ, ... В частности, список может быть пустым, т. е. не содержать ни одной формулы.

Выводом формулы $\mathcal{B}$ *из списка гипотез* Γ называется конечная последовательность формул $\mathcal{B}_1, \ldots, \mathcal{B}_n$, где $\mathcal{B}_n = \mathcal{B}$ и каждая из формул $\mathcal{B}_i$ для $i = 1, \ldots n$ является либо аксиомой, либо одной из гипотез (т. е. содержится в списке Γ), либо непосредственным следствием каких-либо предыдущих формул (из числа $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$) по одному из правил вывода. Разумеется, всюду речь идет о формулах, аксиомах и правилах вывода одной и той же формальной теории.

Формула $\mathcal{B}$ называется *выводимой из списка гипотез* $\Gamma = \mathcal{C}_1, \ldots, \mathcal{C}_k$, если существует ее вывод из этого списка в рассматриваемой формальной теории $\mathbf{T}$. Обозначается это так

$$\Gamma \vdash_{\mathbf{T}} \mathcal{B},$$

или подробнее

$$\mathcal{C}_1, \ldots, \mathcal{C}_k \vdash_{\mathbf{T}} \mathcal{B}.$$

И здесь индекс $_{\mathbf{T}}$ у знака '$\vdash$' можно опускать, если ясно, какая теория имеется в виду.

Из определения видно, что вывод из пустого списка гипотез — это просто вывод в данной формальной теории, так что обозначение

$$\vdash_{\mathbf{T}} \mathcal{B}$$

не является двусмысленным.

Теорема 1.1.1. *Если* $\Gamma \vdash \mathcal{D}_1, \ldots, \Gamma \vdash \mathcal{D}_s$ *и* $\mathcal{D}_1, \ldots, \mathcal{D}_s \vdash \mathcal{B}$, *то* $\Gamma \vdash \mathcal{B}$.

Доказательство. Пусть $\mathcal{B}_{i1}, \ldots, \mathcal{B}_{in_i}$ — вывод $\mathcal{D}_i$ из Γ, а $\mathcal{B}_1, \ldots, \mathcal{B}_n$ — вывод $\mathcal{B}$ из $\mathcal{D}_1, \ldots \mathcal{D}_s$. Выпишем последовательность формул

$$\mathcal{B}_{11}, \ldots, \mathcal{B}_{1n_1}, \ldots, \mathcal{B}_{s1}, \ldots, \mathcal{B}_{sn_s}, \mathcal{B}_1, \ldots, \mathcal{B}_n. \tag{6}$$

Покажем, что она представляет собой вывод формулы $\mathcal{B}$ из списка гипотез Γ. Действительно, $\mathcal{B}_n = \mathcal{B}$, так как $\mathcal{B}_n$ — это последняя формула вывода $\mathcal{B}$ из $\mathcal{D}_1, \ldots \mathcal{D}_s$. Каждая из формул $\mathcal{B}_{ik}$, где $1 \le i \le s, 1 \le k \le n_i$, принадлежит выводу формулы $\mathcal{D}_i$ из Γ и, следовательно, является либо аксиомой, либо гипотезой из списка Γ, либо непосредственным следствием некоторых формул, предшествующих ей в этом выводе. Но те же формулы предшествуют ей и в последовательности (6). Таким образом, любая из этих формул может быть включена в последовательность (6), рассматриваемую как вывод из списка Γ. В частности, это относится и к формулам $\mathcal{B}_{in_i}$ для $i = 1, \ldots, s$.

Каждая из формул $\mathcal{B}_j$ для $j = 1, \ldots, n$ является либо аксиомой, либо непосредственным следствием формул, предшествующих ей в последовательности $\mathcal{B}_1, \ldots, \mathcal{B}_n$, а следовательно, и в последовательности (6), либо, наконец, одной из гипотез $\mathcal{D}_1, \ldots \mathcal{D}_s$, например, $\mathcal{D}_i$. Но в последнем случае она совпадает с формулой $\mathcal{B}_{in_i}$, стоящей раньше ее в последовательности (6). Тогда она может быть включена в вывод на том же основании, на каком в нем содержится формула $\mathcal{B}_{in_i}$.

Итак, последовательность (6) удовлетворяет всем требованиям определения вывода формулы $\mathcal{B}$ из списка Γ, так что $\mathcal{B}$ выводима из этого списка. $\lhd$

Упражнение 1.1.9. Пусть каждая из формул списка Γ содержится в списке Δ. Доказать, что если $\Gamma \vdash \mathcal{B}$, то $\Delta \vdash \mathcal{B}$.

Упражнение 1.1.10. Пусть $\mathcal{C}_1, \ldots, \mathcal{C}_k \vdash_{\mathbf{T}} \mathcal{A}$. Построим формальную теорию $\mathbf{T}_1$ с теми же алфавитом, множеством формул и множеством аксиом, что у теории $\mathbf{T}$, и отличающуюся от нее лишь тем, что к совокупности правил вывода теории $\mathbf{T}$ добавлено правило $\mathcal{C}_1, \ldots, \mathcal{C}_k \vdash \mathcal{A}$. Доказать, что если $\vdash_{\mathbf{T}_1} \mathcal{B}$, то и $\vdash_{\mathbf{T}} \mathcal{B}$.

Рассмотренный выше пример и упражнение 1.1.8 показывают, что в рамках нашего определения могут быть построены формальные теории, допускающие содержательную интерпретацию. И все же у читателя, безусловно знакомого с логическими знаками и их использованием в логических формулах, может остаться законное чувство неудовлетворенности. Например, схема рассуждений: если $\mathcal{A}$ — палиндром, то и $a\mathcal{A}a$ — палиндром, несомненно правильна, но это схема рассуждений о палиндромах. В математике же (как и в любой науке) применяются гораздо более общие и универсальные схемы утверждений, например: «если верно утверждение $\mathcal{A}$ и верно, что из $\mathcal{A}$ следует $\mathcal{B}$, то и $\mathcal{B}$ должно быть верно», или «если из $\mathcal{A}$ следует $\mathcal{B}$, но $\mathcal{B}$ ложно, то и $\mathcal{A}$ должно быть ложно». Желательно, чтобы рассуждения (или утверждения) такого и даже более общего типа, а вернее — их формальные эквиваленты, входили органической частью в любую формальную теорию, которую мы будем рассматривать. Для этого язык формальной теории должен включать подходящие изобразительные средства, и сами схемы рассуждений должны быть представлены в ней либо правилами вывода, либо в виде аксиом или, в крайнем случае, — в виде легко выводимых теорем, а в самом крайнем случае — в виде легко доказываемых метатеорем.

Мы уже говорили в разд. 1.1.1, что содержательные математические теории имеют дело с такими понятиями, как объекты, их свойства (типы), отношения между ними, операции, которые над ними можно выполнять. Хотелось бы, чтобы и эти понятия имели прямые эквиваленты в любой формальной теории. По этим причинам данное выше определение формальной теории следует конкретизировать, придав ему черты, отображающие понятийную и логическую структуру математических теорий.

1.2. Логические формальные теории

1.2.1. Язык

Собственно логика начинается лишь тогда, когда относительно рассматриваемых объектов делаются некоторые высказывания (утверждения). Понятие высказывания примерно соответствует понятию повествовательного предложения в обычном языке.

Но в повседневной жизни почти каждое высказывание несет на себе отпечаток желаний и чувств говорящего, его сомнений, ожиданий и надежд. Цель логики как науки — отвлечься от всего этого и сосредоточиться лишь на одном свойстве высказываний — их истинности, причем истинности непреходящей, что позволяет избавиться от категории времени. Например, высказывания: «1=1», «$2 < 5$», «$x^2 \geqslant 0$», «*все прямые углы равны друг другу*» — истинны, причем высказывание «$x^2 \geqslant 0$» истинно, какое бы значение ни принимала переменная x. Высказывания: «$0 \neq 0$», «$2 \geqslant 5$», «*любые две прямые, перпендикулярные третьей, перпендикулярны друг другу*», «$x = x + 1$» — ложны, причем последнее высказывание ложно при любом значении x. Высказывания: «$x > 1$», «*точка A лежит на прямой p*» могут быть как истинны, так и ложны, в зависимости от того, что обозначают переменные x, A и p.

На этих содержательных примерах мы познакомились с тремя типами высказываний: тождественно истинными, тождественно ложными и имеющими переменное значение истинности. Все тождественно истинные высказывания в грубом приближении эквивалентны, поскольку логика интересуется прежде всего истинностью или ложностью высказывания (хотя иногда отнести высказывание к одному из трех названных типов невозможно без анализа структуры высказывания и его связей с какими-либо иными высказываниями).

Введем символику (язык), к которому будем прибегать при построении формальных логических теорий. Тождественно истинное высказывание обозначим буквой T (от английского слова true — истина). Для тождественно ложного высказывания примем обозначение F (от false — ложь). Символы T и F называются *пропозициональными* (от латинского propositio — предложение, высказывание) или *логическими константами*. Для обозначения высказываний, способных принимать различные значения, будем использовать символы B_1, B_2, ..., называемые *пропозициональными переменными* или *пропозициональными буквами*.

Любые обозначения высказываний в логике называются *формулами*. Самые простые формулы — это введенные нами пропозициональные константы T и F и пропозициональные буквы (переменные) B_1, B_2, ... Эти формулы называются *элементарными*, *атомарными* или *первичными*. Впоследствии (с. 26) мы пополним этот класс еще одной разновидностью обозначений для высказываний.

Из уже имеющихся формул можно образовывать более сложные, пользуясь *логическими связками*: '$\neg$', '$\wedge$', '$\vee$', '$\supset$' и '$\equiv$', а также скобками '(' и ')'. Если $\mathcal{A}$ и $\mathcal{B}$ — формулы (первичные или уже построенные), то записи (слова)

$$\neg\mathcal{A},\ (\mathcal{A} \wedge \mathcal{B}),\ (\mathcal{A} \vee \mathcal{B}),\ (\mathcal{A} \supset \mathcal{B}),\ (\mathcal{A} \equiv \mathcal{B}), \tag{1}$$

также являются формулами. Напомним, что здесь выписаны не сами формулы, а их *схемы* в метаязыке. Исходя из элементарных формул и многократно применяя эти схемы, можно построить обширный класс формул теории.

Можно считать, что логические связки — это знаки операций с логическими операндами и значением. Эти операции называются (в указанном порядке): *отрицание, конъюнкция, дизъюнкция, импликация* и *эквивалентность* (реже *эквиваленция*). Иногда так же называются и соответствующие формулы из списка (1). В каждом случае формулы $\mathcal{A}$ и (кроме первого случая) $\mathcal{B}$ называются *частями* (или *членами*) формул (1). В формуле $(\mathcal{A} \supset \mathcal{B})$ формула $\mathcal{A}$ называется *посылкой*, а $\mathcal{B}$ — *заключением* импликации.

Каждая из частей может, в свою очередь, содержать другие связки, но связка, явно выписанная в формулах (1), называется *главной* в своей формуле. Например, в формуле

$$((B_1 \vee \neg B_2) \supset \neg(B_2 \equiv (B_3 \wedge B_1))) \tag{2}$$

главной связкой является '$\supset$', поскольку эта формула построена по схеме $(\mathcal{A} \supset \mathcal{B})$, где $\mathcal{A}$ — это $(B_1 \vee \neg B_2)$, а $\mathcal{B}$ — $\neg(B_2 \equiv (B_3 \wedge B_1))$.

Подформулой данной формулы называется она сама, а также подформулы ее частей. Отсюда следует, что подформулами некоторой формулы, кроме ее самой, являются ее части (если они есть), части ее частей (с той же оговоркой), части этих частей и т. д. Части и подформулы формулы — это не просто формулы, которые в ней можно увидеть, а определенные вхождения этих формул.

Элементарные подформулы любой формулы не имеют частей и поэтому не содержат никаких подформул, кроме самих себя. По этой причине их называют также *минимальными* подформулами. *Собственными подформулами* формулы называются все ее подформулы, кроме нее самой.

Для примера построим так называемое дерево разбора (синтаксического) формулы (2) — рис. 1.

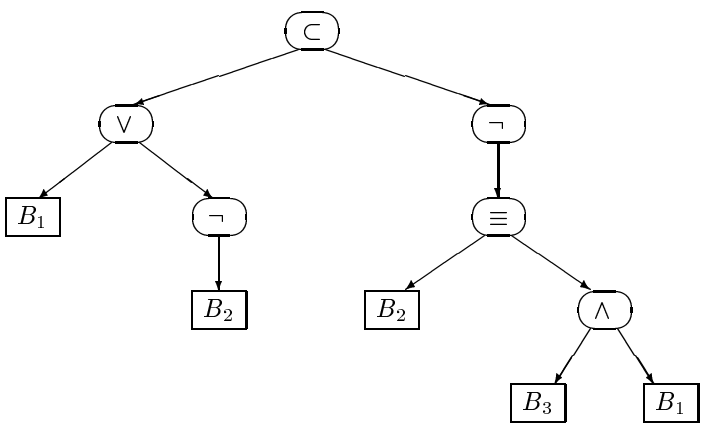


Рис. 1

Все это дерево, как и каждое его собственное поддерево, представляет одну из подформул данной формулы. Элементарным подформулам соответствуют

5 терминальных вершин дерева, неэлементарным — 6 поддеревьев с корнями в нетерминальных вершинах, каждая из них помечена главной связкой такой подформулы.

Назвав логические связки операциями, мы взяли на себя обязанность описать правила исполнения этих операций, вычисления их результатов. Эту сторону языка логических формальных теорий называют иногда *алгеброй высказываний*.

При этом термин «формула» заменяют, как это принято в алгебре, термином «(пропозициональная) форма». Эти правила сведены в следующую таблицу.

$V(\mathcal{A})$	$V(\neg\mathcal{A})$	$V(\mathcal{B})$	$V(\mathcal{A}\wedge\mathcal{B})$	$V(\mathcal{A}\vee\mathcal{B})$	$V(\mathcal{A}\supset\mathcal{B})$	$V(\mathcal{A}\equiv\mathcal{B})$
T	F	T	T	T	T	T
T		F	F	T	F	F
F	T	T	F	T	T	F
F		F	F	F	T	T

В этой таблице, как и всюду в дальнейшем, $V(\mathcal{A})$ обозначает значение формулы $\mathcal{A}$ (а $V(\neg\mathcal{A})$, естественно, — значение формулы $\neg\mathcal{A}$ и т. д.).

Формула $\neg\mathcal{A}$ читается «не $\mathcal{A}$», «неверно, что $\mathcal{A}$» или «отрицание $\mathcal{A}$» и обозначает высказывание, противоположное по значению высказыванию $\mathcal{A}$.

Формула $(\mathcal{A}\wedge\mathcal{B})$ читается «$\mathcal{A}$ и $\mathcal{B}$» и понимается как высказывание, истинное тогда и только тогда, когда высказывания $\mathcal{A}$ и $\mathcal{B}$ оба истинны. Из различных смыслов, которые союз «и» имеет в русском языке, здесь выбран только один. Нельзя, например, обозначать через $a\wedge b$ совокупность, состоящую из элементов a и b.

Формула $(\mathcal{A}\vee\mathcal{B})$ читается «$\mathcal{A}$ или $\mathcal{B}$». Она обозначает высказывание, истинное, когда истинно хотя бы одно из высказываний $\mathcal{A}$ и $\mathcal{B}$, и ложное, когда они оба ложны. И здесь выбран только один из различных оттенков смысла союза «или» в русском языке.

Формула $(\mathcal{A}\supset\mathcal{B})$ может читаться как «$\mathcal{A}$ влечет $\mathcal{B}$», «если $\mathcal{A}$, то $\mathcal{B}$», «из $\mathcal{A}$ следует $\mathcal{B}$». Все эти обороты речи в русском языке (и их аналоги — в любом другом) имеют особенно много смысловых ассоциаций, обычно подразумевающих причинную связь между $\mathcal{A}$ и $\mathcal{B}$. Читателю надо отрешиться от подобных ассоциаций и запомнить, что формула $(\mathcal{A}\supset\mathcal{B})$ означает высказывание, истинное всегда, за исключением случая, когда $\mathcal{A}$ означает истинное, а $\mathcal{B}$ — ложное высказывание.

Значения в первых двух строчках столбца $V(\mathcal{A}\supset\mathcal{B})$ таблицы не должны вызывать сомнений. Странным может показаться то, что формуле $(\mathcal{A}\supset\mathcal{B})$ приписывается какое-то значение (более того — значение логической истины) и тогда, когда высказывание $\mathcal{A}$ ложно. Рассмотрим, однако, несомненно истинное высказывание «*если* n *делится на* 10, *то* n *делится на* 5». Но если истинно это общее высказывание, то истинны должны быть и его частные случаи. При $n=25$ оно превращается в «*если* 25 *делится на* 10, *то* 25 *делится на* 5». Здесь посылка «25 *делится на* 10» ложна, а заключение «25 *делится на* 5» истинно. Этот случай предусмотрен третьей строкой таблицы. Примем $n=12$: «*если* 12 *делится*

на 10, *то* 12 *делится на* 5». Здесь и посылка и заключение ложны, что соответствует четвертой строке таблицы. Даже если кто-то и испытывает ощущение, что общее утверждение как бы не относится к подобным случаям, их все же нельзя воспринимать как опровержение этого общего утверждения.

Ну а как быть с ложным утверждением «*если* n *делится на* 5, *то* n *делится на* 10»? Не дают ли его частные случаи для $n = 12$ (посылка и заключение ложны) и $n = 20$ (посылка и заключение истинны) оснований для пересмотра таблицы (ее четвертой и первой строк)? Нет, из ложности общего утверждения логически не вытекает ложность его частных случаев. Скорее, наоборот, — это можно считать еще одним примером осмысленности третьей строки таблицы: истинные заключения могут быть получены и из ложной посылки.

В соответствии с таблицей истинности для формулы $(\mathcal{A} \supset \mathcal{B})$ должны считаться истинными и такие утверждения: «*если* $2 \times 2 = 5$, *то существуют такие целые числа* $n > 2$, $a > 0$, $b > 0$, c, *что* $a^n + b^n = c^n$» и «*если* $1 > 0$, *то* $x + y = y + x$» — отсутствие видимой смысловой связи между посылкой и заключением подобного утверждения не должно мешать приписать ему некоторое значение истинности (в обоих случаях T, даже если бы большая теорема Ферма еще не была доказана). Предмет логики — не смысл высказываний, а возможная связь между их логическими значениями.

Есть еще два способа прочтения формулы $(\mathcal{A} \supset \mathcal{B})$: «$\mathcal{A}$ — *достаточное условие для* $\mathcal{B}$» и «$\mathcal{B}$ — *необходимое условие для* $\mathcal{A}$». В математике слова «достаточно» и «необходимо» всегда понимаются именно в таком смысле.

Наконец, формула $(\mathcal{A} \equiv \mathcal{B})$ имеет такие возможные прочтения: «$\mathcal{A}$ эквивалентно $\mathcal{B}$», «$\mathcal{A}$ тогда и только тогда, когда $\mathcal{B}$», «$\mathcal{A}$ — необходимое и достаточное условие для $\mathcal{B}$». Она обозначает высказывание, истинное, когда высказывания $\mathcal{A}$ и $\mathcal{B}$ оба истинны или оба ложны, и ложное, когда значения истинности этих высказываний различны.

Ряд интересных результатов, относящихся к области математической логики, можно получить, пользуясь для обозначения высказываний только пропозициональными константами T и F, переменными B_i, а также метаобозначениями $\mathcal{A}$, $\mathcal{B}$, ... Многие из этих результатов будут рассмотрены в разд. 1.3. Но стоит еще раз взглянуть на примеры, приведенные на с. 11 и далее, и мы увидим, что глубинную структуру высказываний эти средства не раскрывают. Они не дают возможности связать значения высказываний со свойствами объектов, к которым эти высказывания относятся.

В расширенной символике для обозначения конкретных объектов будем применять символы a_1, a_2, ..., называемые *предметными константами*. Подразумевается, что предметная константа всегда обозначает один и тот же объект. В арифметике такой константой является, например, 0. В элементарной геометрии иногда используются константы: d — для обозначения величины прямого угла, π — для обозначения отношения длины окружности к ее диаметру. Для собственно геометрических объектов (точек, прямых и пр.) констант нет.

В логике мы стремимся сделать обозначения более стандартными, единообразными. Поэтому в качестве констант можно ограничиться лишь указанными

символами a_1, a_2, ..., а их соответствие традиционным обозначениям задавать при описании интерпретации. Заметим, что обозначение a_i, в котором индексом у буквы a служит буква i, а не натуральное число, не является символом рассматриваемого языка. Довольно часто мы все же будем прибегать к обозначению a_i, но уже в качестве метаобозначения для какой-нибудь (произвольной) предметной константы, а не того объекта, который та обозначает.

Символы x_1, x_2, ..., называемые *предметными переменными*, также обозначают некоторые объекты, но, в отличие от предметных констант, каждая из предметных переменных может обозначать любой, совершенно произвольный, объект.

Для обозначения операций над объектами, порождающих новые объекты, служат так называемые *функциональные буквы*: f_1^1, f_2^1, ..., f_1^2, f_2^2, ... Верхний индекс k у буквы f_i^k указывает, сколько операндов (аргументов) должно быть у соответствующей операции. Нижний индекс i — это порядковый номер функциональной буквы в ряду обозначений f_1^k, f_2^k, ...

Требуемое число операндов у операции часто называют числом мест (для операндов) или ее вместимостью (хуже — местностью, еще хуже — арностью). Поэтому функциональные буквы f_1^1, f_2^1,... и соответствующие им операции называют *одноместными* (в латинизированной терминологии — *унарными*), буквы f_1^2, f_2^2, ... — *двухместными* (*бинарными*), буквы f_1^3, f_2^3, ... — *трехместными* (*тернарными*) и т. д.

В математической практике роль функциональных букв выполняют, во-первых, знаки операций: '$+$', '$-$', '$\times$', '$\sqrt{\ }$', '$|\ |$' (в обозначении $|\,x\,|$ для абсолютной величины числа x) и т. п., во-вторых, символы элементарных функций (обычно одноместных): exp, log, sin, ..., в-третьих, обозначения функций f, g, F, φ и т. д. Символы первых двух категорий связаны со вполне определенными функциональными объектами, их можно назвать функциональными константами. Примеры таких свойств мы видели в перечне равенств (1.1.1–2). Символы третьей категории, напротив, совершенно нейтральны и могут обозначать произвольные функции (операции). Для них напрашивается название функциональных переменных. Но в тех формальных теориях, которые мы будем изучать, функциональных переменных и обозначений для них не будет. В разд. 1.5 будет показано, что в рамках теории множеств эту роль могут исполнить обычные предметные переменные. У нас они могут иногда появляться вместо f_i^k в роли метапеременных для функциональных букв.

Обозначениями наиболее общего вида для объектов теории являются *термы*. Они строятся из предметных констант и переменных и функциональных букв с помощью *синтаксических символов*: '(', ')' и ',' по следующим правилам:

1) любая предметная константа — это терм,

2) любая предметная переменная — это терм,

3) если t_1, ..., t_k — уже построенные термы, а f_i^k — k-местная функциональная буква, то выражение вида $f_i^k(t_1, \ldots, t_k)$ — также терм.

Некоторые авторы к этим трем правилам добавляют четвертое:

4) никаких других термов не существует.

Но проще руководствоваться принципом: нет других правил — значит, нет и других термов.

Примеры термов:

$$a_1,\ x_{12},\ f^1_1(x_1),\ f^3_5(a_2, f^2_{10}(x_3, f^1_4(x_1))), f^1_2(f^1_7(f^2_4(a_1, x_2)))).$$

Заметим, что синтаксические символы в записи терма избыточны. Опустив их, мы получим в последнем случае запись

$$f^3_5 a_2 f^2_{10} x_3 f^1_4 x_1 f^1_2 f^1_7 f^2_4 a_1 x_2,$$

которая (благодаря указанию вместимости каждой функциональной буквы) может быть расчленена на входящие в нее внутренние термы так же однозначно, как и с помощью этих символов. Включаем мы скобки и запятые в запись термов только ради простоты их прочтения.

Упражнение 1.2.1. Описать распознающую грамматику термов — способ проверки, является ли данное слово термом в бесскобочной записи.

Объект, соответствующий терму $f^k_i(t_1, \ldots, t_k)$, получается в результате применения операции, обозначенной буквой f^k_i, к объектам, представленными термами $t_1, \ldots, t_k$.

Напомним, что хотя мы и говорим «терм t_1», «терм $f^k_i(t_1, \ldots, t_k)$», сами записи «t_1», «$f^k_i(t_1, \ldots, t_k)$» и т. п. термами не являются — это метаобозначения.

Упражнение 1.2.2. В развитие упражнения 1.1.2 построить порождающую грамматику термов — формальную теорию, теоремами которой были бы любые термы и только они.

В решении этого упражнения можно увидеть, как разрешается противоречие между требованием конечности алфавита формальной теории и использованием в ней бесконечного множества символов. В дальнейшем мы свободно будем пользоваться бесконечными алфавитами, не оговаривая, что нетрудно свести их к конечным. Впрочем, по существу бесконечный алфавит нужен лишь для предметных переменных и то лишь для того, чтобы иметь возможность писать сколь угодно сложные термы, а впоследствии и формулы. Любой терм или формула содержат лишь конечное число переменных.

Расширим, теперь уже окончательно, понятие элементарной формулы.

Предикатными буквами называются символы $P^1_1, P^1_2, \ldots, P^2_1, P^2_2, \ldots$ Как в случае функциональных букв, верхний индекс k у буквы P^k_i обозначает количество мест для аргументов, а нижний — i — служит порядковым номером.

Если P^k_i — k-местная предикатная буква, а $t_1, \ldots, t_k$ — термы, то запись вида $P^k_i(t_1, \ldots, t_k)$ также представляет собой элементарную формулу. Элементарность понимается здесь в том смысле, что такая формула не содержит внутри себя других формул, хотя может содержать сколь угодно сложные термы — обозначения не высказываний, а объектов этих высказываний.

При содержательной интерпретации каждой предикатной букве P_i^k ставится в соответствие некоторое свойство, которым могут обладать или не обладать упорядоченные наборы из k объектов. Обычно такие свойства называются k-местными отношениями (иногда — *предикатами*), а термин «свойство» сохраняется за одноместными отношениями. Примеры отношений: «*быть положительным числом*», «*быть точкой*», «*быть прямой*» — одноместные отношения (свойства); «*быть меньше чем*», «*быть равным*», «*лежать на*» (отношение между точкой и прямой), «*проходить через*» (отношение между прямой и точкой, отличающееся от предыдущего лишь порядком аргументов) — двухместные отношения; «*давать в сумме*», «*лежать между*» (отношение между точками на прямой) — трехместные отношения.

Элементарная формула $P_i^k(t_1, \ldots, t_k)$ содержательно понимается как запись утверждения, что объекты, являющиеся значениями термов t_1, $\ldots$, t_k, находятся между собой в некотором определенном отношении. Когда мы хотим словами объяснить содержательный смысл отношения, обозначенного предикатной буквой P_i^k, то проще и точнее это можно сделать на примере формулы $P_i^k(x_1, \ldots, x_k)$, особенно при большом k. Например, лучше сказать, что формула $P_1^3(x_1, x_2, x_3)$ обозначает утверждение «*сумма чисел x_1 и x_2 равна x_3*», чем определить букву P_1^3 как обозначение для отношения «*давать в сумме*». В метаязыке никакие конкретные свойства предикатных букв P_i^k (как и функциональных букв f_i^k) не фиксируются — это должно быть сделано при построении каждой отдельной формальной теории с помощью ее аксиом.

Элементарные формулы вида $P_i^k(t_1, \ldots, t_k)$ и любые формулы, которые могут быть из них получены с помощью только логических связок $\neg$, $\wedge$, $\vee$, $\supset$ и $\equiv$, обозначают высказывания, истинность которых, вообще говоря, зависит от значений входящих в эти формулы переменных. В математике, да и не только в ней, часто возникает потребность сформулировать утверждение, истинность которого не зависит от конкретного выбора объекта или объектов, упоминаемых в нем. Примерами подобных утверждений могут быть тождества (1.1.1–2), или утверждение «*через две различные точки можно провести прямую и притом лишь одну*». В последнем случае, хотя речь и идет о единственной прямой, проходящей через две некоторые точки, фактически описывается свойство не какой-либо конкретной прямой, а всей совокупности точек и прямых, рассматриваемых в геометрии.

В нашей формальной символике должны найтись средства для записи и таких утверждений. Расширим понятие формулы, добавив к уже имеющимся еще одно правило построения формул: если x_i — предметная переменная, а $\mathcal{A}$ — формула, то $\forall x_i \mathcal{A}$ и $\exists x_i \mathcal{A}$ — тоже формулы. Выражения $\forall x_i$ и $\exists x_i$ называются *кванторными связками* или *кванторами* (по переменной x_i). Квантор $\forall x_i$ называется *квантором всеобщности*, а $\exists x_i$ — *квантором существования*. В формуле вида $\forall x_i \mathcal{A}$ (или $\exists x_i \mathcal{A}$) формула $\mathcal{A}$ называется ее *частью*, а связка $\forall x_i$ (соответственно $\exists x_i$) — *главной связкой*. Иногда говорят, что в этих формулах квантор «навешен» на формулу $\mathcal{A}$. Определение подформулы остается прежним, но область его применения расширяется на формулы вновь введенного вида. Фор-

мула $\mathcal{A}$ называется также *областью действия* квантора $\forall x_i$ в формуле $\forall x_i \mathcal{A}$ (квантора $\exists x_i$ — в формуле $\exists x_i \mathcal{A}$).

Формуле $\forall x_i \mathcal{A}$ приписывается следующий содержательный смысл: «утверждение $\mathcal{A}$ истинно, каково бы ни было значение переменной x_i (каков бы ни был объект, обозначенный этим символом)». Отсюда способ прочтения этой формулы: «для всех x_i $\mathcal{A}$» или «для любого x_i $\mathcal{A}$». Формула $\exists x_i \mathcal{A}$ читается: «существует x_i, такое что $\mathcal{A}$», подробнее — обозначает утверждение: «среди всех возможных значений переменной x_i найдется такое, при котором утверждение $\mathcal{A}$ истинно». В разд. 1.2.2 будут сформулированы более точные правила, по которым (в конкретных условиях) формуле $\forall x_i \mathcal{A}$ или $\exists x_i \mathcal{A}$ приписывается (если это вообще возможно) значение T или F (истина или ложь). А пока соберем воедино определение формулы и введем еще некоторые термины.

Элементарными формулами формальной теории $\mathbf{T}$ называются:

1) пропозициональные константы T и F,

2) пропозициональные буквы B_i, имеющиеся в теории $\mathbf{T}$,

3) слова вида $P_i^k(t_1, \ldots, t_k)$, где P_i^k — предикатная буква теории $\mathbf{T}$, а $t_1, \ldots, t_k$ — термы этой теории.

Формулы формальной теории $\mathbf{T}$ — это элементарные формулы данной теории, а также:

4) слова вида $\neg\mathcal{A}$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, $(\mathcal{A} \supset \mathcal{B})$, $(\mathcal{A} \equiv \mathcal{B})$, где $\mathcal{A}$ и $\mathcal{B}$ — ранее построенные формулы теории $\mathbf{T}$,

5) слова вида $\forall x_i \mathcal{A}$ и $\exists x_i \mathcal{A}$, где x_i — предметная переменная теории $\mathbf{T}$, а $\mathcal{A}$ — уже построенная формула этой теории.

Пункты 1) – 5) образуют полный набор правил построения формул. Пункты 1) – 3) выделены из этого перечня только для того, чтобы ввести термин «элементарная формула» для формул, не содержащих ни логических, ни кванторных связок.

Выражениями формальной теории будем называть ее термы и формулы.

Любое вхождение переменной x_i в формулу $\forall x_i \mathcal{A}$ или $\exists x_i \mathcal{A}$ называется *связанным*. Это относится и к случаю, когда формула $\forall x_i \mathcal{A}$ (или $\exists x_i \mathcal{A}$) является собственной подформулой некоторой формулы $\mathcal{B}$. Если вхождение переменной x_i в формулу $\mathcal{B}$ не связанно (т. е. не содержится ни в какой ее подформуле вида $\forall x_i \mathcal{A}$ или $\exists x_i \mathcal{A}$), то оно называется *свободным*. Подчеркнем, что эти определения относятся не к переменным, а к вхождениям переменных в формулу, т. е. к каждому отдельному появлению символа x_i в том или ином месте формулы. Говорят, что переменная *связана* в формуле, если последняя содержит хотя бы одно связанное вхождение этой переменной, и *свободна* в формуле, которая содержит свободное вхождение этой переменной. При этом оказывается, что переменная может быть как связанной, так свободной в одной и той же формуле.

Пример. В формуле

$$(P_1^2(x_1, x_2) \supset \forall x_1 (P_3^2(x_2, x_1) \vee \exists x_3 (P_2^3(x_1, x_3, x_2) \wedge P_5^1(x_3))))$$

содержатся 4 вхождения переменной x_1 и по 3 вхождения переменных x_2 и x_3. Все вхождения переменной x_3 связанны, так как все они содержатся в подформуле $\exists x_3 (P_2^3(x_1, x_3, x_2) \wedge P_5^1(x_3))$ данной формулы. Все вхождения переменной

x_2 свободны, так как формула не содержит кванторов по этой переменной, а следовательно, и подформул вида $\forall x_2\,\mathcal{A}$ или $\exists x_2\,\mathcal{A}$. Первое вхождение переменной x_1 свободно, а остальные три — связанны, так как они содержатся либо в кванторе $\forall x_1$ (второе вхождение), либо в области действия этого квантора (третье и четвертое вхождения). Таким образом, можно сказать, что переменная x_2 свободна в данной формуле, переменная x_3 — связана, переменная x_1 — одновременно свободна и связана, а переменная x_4 — ни связана, ни свободна, так как данная формула вообще не содержит вхождений этой переменной.

Формула, не содержащая свободных вхождений переменных, называется *замкнутой*.

1.2.2. Интерпретации

Уже не раз выше было сказано, что формальные теории интересны не столько сами по себе, сколько из-за возможности сопоставить символам и выражениям некоторые содержательные понятия. В общих чертах говорилось и о том, как это делается. Теперь в нашем изложении металогики настало время дать более точные определения.

Будем говорить, что задана *интерпретация* I некоторой формальной теории $\mathbf{T}$, если:

1) указано некоторое множество D (конечное или бесконечное, но не пустое), называемое *предметной областью* интерпретации I;

2) каждой предметной константе a_i теории $\mathbf{T}$ (если они есть) сопоставлен некоторый элемент a_i^* предметной области D;

3) каждой функциональной букве f_i^k теории $\mathbf{T}$ (с той же оговоркой) сопоставлена k-местная операция f_i^{k*} над элементами области D, т. е. функция, которая любой совокупности $u_1, \ldots, u_k$ объектов этой области ставит в соответствие один определенный элемент этой же области;

4) каждой пропозициональной букве B_i теории $\mathbf{T}$ сопоставлено одно из двух логических значений: «истина» или «ложь» (обозначаемых, как всегда, буквами T и F); значение, сопоставленное B_i, обозначим B_i^*;

5) каждой предикатной букве P_i^k теории $\mathbf{T}$ сопоставлено k-местное отношение P_i^{k*} между объектами из области D.

В этом определении ничего не говорится, что может быть сопоставлено предметным переменным x_i. Подразумевается, что каждая предметная переменная может обозначать (т. е. ей может быть сопоставлен) любой элемент предметной области D. Ввиду открывающегося здесь произвола бессмысленно говорить о значении переменной (а также о значении выражения, т. е. терма или формулы) в данной интерпретации. Понятие значения выражения приобретает смысл, когда, кроме интерпретации, заданы значения переменных, входящих в это выражение.

Назовем *набором* значений переменных $y_1, \ldots, y_m$ множество s пар: $s = \{(y_1, v_1), \ldots, (y_m, v_m)\}$, где $v_1, \ldots, v_m$ — элементы предметной области D рассматриваемой интерпретации. Все переменные $y_1, \ldots, y_m$, содержащиеся в наборе, должны быть попарно различны, а их значения $v_1, \ldots, v_m$ могут частично

(или полностью) совпадать. Набор задает условия, в которых желательно знать значение отдельной переменной, а также терма или формулы.

Значение терма t или формулы $\mathcal{F}$ в интерпретации I на наборе s обозначим $V_{\mathrm{I},s}(t)$ или, соответственно, $V_{\mathrm{I},s}(\mathcal{F})$. Так как в большинстве случаев бывает ясно, о какой интерпретации идет речь, то индекс I можно не писать и пользоваться обозначениями $V_s(t)$ и $V_s(\mathcal{F})$. Если же из контекста ясно, какой набор имеется в виду, или это безразлично, то можно опускать и индекс s и писать $V(t)$ или $V(\mathcal{F})$, что мы фактически и делали в предыдущем разделе, когда определяли способ вычисления значений формул с различными главными связками с помощью соответствующих таблиц истинности (где предметные переменные явно не присутствовали в формулах $\mathcal{A}$ и $\mathcal{B}$).

Значения термов определяются по следующим правилам.

1) $V_s(a_i) = a_i^*$ для любого набора s, т. е. значение предметной константы зависит только от интерпретации, но не от набора.

2) Если переменная x_i совпадает с одной из переменных $y_1, \ldots, y_m$ набора s, например, с переменной y_j, то $V_s(x_i) = v_j$, в противном случае $V_s(x_i)$ не определено. Другими словами, если $s = \{(y_1, v_1), \ldots, (y_m, v_m)\}$, то $V_s(y_j) = v_j$, $j = 1, \ldots, m$, а значения остальных переменных на наборе s не определены.

3) $V_s(f_i^k(t_1, \ldots, t_k)) = f_i^{k*}(V_s(t_1), \ldots, V_s(t_k))$, т. е. значение терма $f_i^k(t_1, \ldots, t_k)$ получается применением операции f_i^{k*} к значениям термов $t_1, \ldots, t_k$. Если хотя бы одно из значений $V_s(t_1), \ldots, V_s(t_k)$ не определено, то и значение терма $f_i^k(t_1, \ldots, t_k)$ на наборе s также не определено.

Теорема 1.2.1. *Если все переменные, входящие в терм t, содержатся в наборе s, то значение терма t на этом наборе определено и принадлежит предметной области D рассматриваемой интерпретации. В противном случае (когда в терм t входит хотя бы одна переменная, не содержащаяся в наборе s) значение $V_s(T)$ не определено.*

Справедливость теоремы вытекает из того, что: 1) каждому правилу построения термов соответствует правило вычисления их значений, 2) по определению интерпретации каждая предметная константа a_i^* — элемент предметной области D, 3) k-местная операция f_i^{k*} всюду определена над этой же областью, 4) по определению набора значений переменных все эти значения также принадлежат области D.

В дальнейшем будем считать подобные разъяснения достаточно полными доказательствами (для теорем сходного уровня сложности), а сейчас приведем более полное доказательство. Оно проводится индукцией по числу n вхождений функциональных букв в терм t.

Доказательство. База индукции — это случай $n = 0$. При этом терм не содержит функциональных букв, т. е. строится по одному из правил 1) или 2) разд. 1.2.1. Если терм образован по правилу 1), то он представляет собой предметную константу a_i, его значение a_i^* определено и принадлежит области D. Если терм образован по правилу 2), то он представляет собой предметную переменную x_i, которая либо содержится среди переменных набора s, тогда ее значение

известно и принадлежит D, либо не входит в их число, тогда значение терма не определено. В обоих случаях утверждение теоремы справедливо.

Индукционный шаг. Пусть $n > 0$ и для всех термов, содержащих менее n вхождений функциональных букв, утверждение теоремы верно. При $n > 0$ терм t может быть образован только по правилу 3) и, следовательно, он имеет вид $f_i^k(t_1, \ldots, t_k)$. Все термы $t_1, \ldots, t_k$, вместе взятые, содержат $n-1$ вхождение функциональных букв, следовательно, и каждый из них содержит менее n вхождений этих букв. Возможны два случая. Если терм t содержит вхождения переменных только из набора s, то и в каждый из термов $t_1, \ldots, t_k$ входят только такие переменные. По предположению индукции значения $V_s(t_1), \ldots, V_s(t_k)$ этих термов должны быть определены и принадлежать области D. К этой совокупности значений применима операция f_i^{k*}, результат ее применения также принадлежит области D и представляет собой значение терма t. Если же в терм t входит переменная не из набора s, то она должна входить хотя бы в один из термов $t_1, \ldots, t_k$. Значение этого терма, опять по предположению индукции, должно быть не определено, но при этом, в соответствии с правилом 3, не определено и значение терма t.

По индукции теорема верна при любом n. $\lhd$

Сформулируем правила вычисления значений формул на некотором произвольном наборе s (в фиксированной интерпретации I). Если переменная x_i входит в число переменных $y_1, \ldots, y_m$ набора s, например, совпадает с переменной y_j, то обозначим через s' набор, получающийся из s заменой значения v_j переменной y_j любым значением, принадлежащим области D (в частности, оно может совпасть с v_j). Если же x_i не содержится в наборе s, то через s' обозначим набор, получающийся из набора s добавлением к нему пары (x_i, v), где v — произвольный элемент области D. В обоих случаях значения всех переменных, отличных от x_i, должны быть одинаковыми в обоих наборах. Набор s' будем называть *x_i-вариацией* набора s.

1) Значение пропозициональной константы T или F — это она сама, независимо ни от набора, ни даже от интерпретации.

2) Значение пропозициональной буквы B_i равно B_i^* на любом наборе s: $V_s(B_i) = B_i^*$, оно зависит лишь от интерпретации.

3) Значение элементарной формулы вида $P_i^k(t_1, \ldots, t_k)$ определено лишь в том случае, когда определены значения $V_s(t_1), \ldots, V_s(t_k)$, оно равно T, если эта упорядоченная совокупность значений термов принадлежит отношению P_i^{k*}, и F — в противном случае. Короче: $V_s(P_i^k(t_1, \ldots, t_k)) = P_i^{k*}(V_s(t_1), \ldots, V_s(t_k))$, где P_i^{k*} обозначает функцию, которая каждой упорядоченной совокупности k значений из области D ставит в соответствие логическое значение T или F.

4) Если определено значение $V_s(\mathcal{A})$ формулы $\mathcal{A}$, то значение формулы $\neg\mathcal{A}$ вычисляется с помощью таблицы на с. 23, а если, кроме того, определено и значение $V_s(\mathcal{B})$, то и значения формул $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, $(\mathcal{A} \supset \mathcal{B})$ и $(\mathcal{A} \equiv \mathcal{B})$ находятся с помощью той же таблицы. Если же значение какой-либо части формулы одного из перечисленных здесь видов не определено, то не определено и значение самой формулы.

5) Значения формул $\forall x_i \mathcal{A}$ и $\exists x_i \mathcal{A}$ на наборе s определены лишь в том случае, если значения формулы $\mathcal{A}$ определены на всех *x_i-вариациях* s' набора s. При этом $V_s(\forall x_i \mathcal{A}) = T$, если $V_{s'}(\mathcal{A}) = T$ для всех наборов s' и $V_s(\forall x_i \mathcal{A}) = F$, если $V_{s'}(\mathcal{A}) = F$ хотя бы на одном таком наборе; $V_s(\exists x_i \mathcal{A}) = F$, если $V_{s'}(\mathcal{A}) = F$ на всех наборах s', и $V_s(\exists x_i \mathcal{A}) = T$, если $V_{s'}(\mathcal{A}) = T$ хотя бы на одной x_i-вариации набора s.

1.2.3. Две точки зрения на математику

По существу данные выше определения ничем не отличаются от определений разд. 1.2.1, они лишь несколько более формализованы. Когда мы говорим, что квантор $\forall x_i$ читается как «для любого значения переменной x_i», или что значение формулы $\forall x_i \mathcal{A}$ на наборе s равно T тогда и только тогда, когда формула $\mathcal{A}$ принимает значение T на любой x_i-вариации набора s, то фактически в обоих случаях мы утверждаем одно и то же. Вторая формулировка подробнее и содержит в себе явное упоминание о значениях переменных, входящих в формулу $\mathcal{A}$, и о значении самой этой формулы. Это помогает более конкретно проводить рассуждения о свойствах формул, содержащих кванторы. Но сами рассуждения ведутся в рамках той же логики, правила которой определяются и обосновываются.

При этом в обоих случаях мы рассчитываем охватить в своем сознании всю совокупность возможных значений переменной x_i (т. е. всю предметную область) и говорить о значениях формулы $\mathcal{A}$ при всех этих значениях переменной x_i. Это — далеко идущее предположение. Если предметная область конечна, то его можно принять с оговорками лишь по поводу затрат на те или иные проверки. Если же она бесконечна, то ситуация становится гораздо сложнее. Пусть, например, предметная область — это множество положительных целых чисел, формула $\mathcal{B}$ — это запись утверждения, что при некотором n уравнение $x^n + y^n = z^n$ не имеет решений, а формула $\mathcal{A}$ имеет вид $(n > 2 \supset \mathcal{B})$.

До недавнего времени математика для многих значений n еще не знала ответа на вопрос, справедливо ли утверждение, выраженное формулой $\mathcal{A}$, но и не могла указать ни одного значения n, для которого оно было бы ложно. Что можно было в этих условиях сказать о значении формулы $\forall n \mathcal{A}$ в интерпретации с указанной предметной областью при обычном понимании операций сложения и возведения в степень и предиката равенства? Во всяком случае, нельзя было утверждать ни что она истинна, ни что она ложна. Многие математики склонны считать, что формула $\forall n \mathcal{A}$ обладает в этой интерпретации некоторым, хотя и неизвестным в ту пору, значением истинности. Другие придерживаются более осторожной точки зрения, полагая, что пока большая теорема Ферма не доказана, говорить об истинности формулы $\forall n \mathcal{A}$ вообще не имеет смысла.

Сторонники первой, так называемой классической, точки зрения считают, что дизъюнкция $(\forall n \mathcal{A} \vee \neg \forall n \mathcal{A})$ истинна в этой (и любой другой) интерпретации, так как любое из предположений $V(\forall n \mathcal{A}) = T$ или $V(\forall n \mathcal{A}) = F$ приводит к $V(\forall n \mathcal{A} \vee \neg \forall n \mathcal{A}) = T$. Их оппоненты — конструктивисты (и интуиционисты) —

предпочитают не приписывать никакого истинностного значения формуле, значения частей которой неизвестны.

То, что было только что сказано о различии двух точек зрения — классической и конструктивной, нельзя назвать даже примитивным их изложением, настолько оно кратко, неполно и страдает отсутствием мотивировок. Но уже из сказанного ясно, что различие начинается тогда, когда рассматриваются бесконечные предметные области (даже само понятие предметной области конструктивисты едва ли готовы принять в изложенной выше трактовке). Между тем, ни один человек, ни даже все человечество не имело дела с бесконечными физическими совокупностями чего бы то ни было. Все результаты, которые дает людям наука (в частности, математика), применяются только к конечным совокупностям. Может быть, можно вообще изгнать понятие бесконечности из математики? Простейшие примеры показывают, что это приводит, по меньшей мере, к чрезвычайному усложнению большинства определений, утверждений и доказательств.

Попытаемся, например, вместо всего натурального ряда чисел ограничиться лишь его отрезком от 0 до некоторого N. Не очень трудно определить «сложение» (будем обозначать эту операцию знаком '$\oplus$') так, чтобы равенство $m \oplus n = m+n$ сохранялось всегда, когда обычная сумма $m + n$ не превосходит N. Если же $m + n > N$, то можно объявить, что результат операции $m \oplus n$ не определен. В этом случае закон коммутативности «сложения» следует сформулировать так: если «сумма» $m \oplus n$ определена, то и $n \oplus m$ определена, и $m \oplus n = n \oplus m$. Сравнительно просто вводится операция «вычитания» и отрицательные числа (не превосходящие N по абсолютной величине). Но заметно более громоздкие оговорки требуются в законе ассоциативности «сложения» — из того, что «суммы» $m \oplus n$ и $(m \oplus n) \oplus p$ определены, не следует, что определены $n \oplus p$ и $m \oplus (n \oplus p)$, так как при $m < 0, n > 0, p > 0$ может случиться, что $m+n+p \leqslant N$, но $n+p > N$.

Сделаем еще одну попытку: при $m + n > N$ примем $m \oplus n = N$ (поскольку чисел, больших N, для нас сейчас не существует, а считать, что $m \oplus n < N$ было бы бессмысленно). Тогда уравнение $m \oplus x = n$ не имеет единственного решения (при $n = N$) и не ясно, что назвать «разностью» целых чисел. Как бы ни поступить, ассоциативный закон $(m \oplus n) \oplus p = m \oplus (n \oplus p)$ может нарушаться (если некоторые из чисел m, n, p отрицательны).

Итак, теория, в которой не рассматриваются бесконечные совокупности, едва ли может быть простой и изящной. Но она была бы не лишена практического значения — с похожими трудностями сталкиваются конструкторы (или «архитекторы») вычислительных машин, в которых невозможно иметь дело с бесконечным разнообразием чисел. Поэтому так называемые арифметические операции вычислительных машин оказываются не всегда выполнимыми, а если они и выполнимы, то не всегда подчиняются тем же законам, что соответствующие операции в математике.

Разумеется, наши примеры не доказывают, что нельзя построить никакой интересной и богатой практически ценными результатами математической теории, в которой не рассматривались бы бесконечные совокупности, подобные нату-

ральному ряду чисел. Многовековой опыт математики и ее приложений говорит, что ограничиться только такими теориями невозможно. Однако известен подход — так называемые нестандартные арифметика и математический анализ, при котором постулируется существование числа (натурального), превосходящего любые другие числа, но значение этого числа не фиксируется и может меняться по мере появления в ходе рассуждений новых конкретных чисел.

Следующий вопрос — это насколько широко должна быть открыта дверь для бесконечных совокупностей. Классическая математика не признает здесь почти никаких ограничений. Попытки обойтись без всяких ограничений привели к появлению противоречивых понятий (подробнее — см. ниже, в разд. 1.5.7). Поэтому классическая математика ставит некоторые ограничения на природу рассматриваемых в ней бесконечных совокупностей, но лишь самые минимальные — такие, которые, как можно надеяться, предотвращают появление противоречий, но в остальном оставляют руки математиков свободными.

Конструктивная математика, напротив, стремится достаточно жестко ограничить использование понятие бесконечности. Признается лишь один ее вид — потенциальная возможность многократно применять некоторые достаточно простые операции к немногочисленным первичным объектам или к объектам, уже построенным за конечное число шагов. Всякое поползновение иметь дело с *актуальной бесконечностью* — результатом как бы завершенного бесконечного процесса — решительно отвергается. Так, беря в качестве первичных объектов буквы некоторого (разумеется, конечного) алфавита и пустое слово, а в качестве единственной операции — приписывание буквы к слову, получаем возможность строить слова в этом алфавите — сколь угодно длинные, но конечные. Эти слова (вместе с некоторым их истолкованием) являются единственными рассматриваемыми объектами, все математические понятия, о которых допускается говорить, могут (и должны) быть представлены такими словами. Заметим, что это не так уж мало, — любую математическую публикацию можно рассматривать как одно слово в конечном алфавите, иногда довольно длинное (и редко допускающее конструктивное истолкование). Однако уже такое понятие, как язык — бесконечное множество слов в данном алфавите (см. разд. 1.1.2), — остается за рамками конструктивной математики, так как построение этого множества никогда не может быть завершено.

Пусть надо доказать справедливость утверждения вида $\exists x \mathcal{A}(x)$. Математик-классик часто рассуждает так. Предположим, что истинно обратное утверждение: $\forall x \neg \mathcal{A}(x)$. Если это предположение удается логическим путем привести к противоречию, то должно найтись значение x, для которого $\mathcal{A}(x)$ истинно, т. е. исходное утверждение верно (от противного). Наш математик признает, что этим способом доказана лишь «чистая теорема существования», не дающая возможности предъявить значение x с требуемым свойством. Конструктивист не находит ничего «чистого» в таком доказательстве. Ему подавай способ (алгоритм) нахождения этого значения. Если же это может быть сделано лишь приближенно, то для любой допустимой погрешности требуется еще указать, сколько шагов приближения должно быть проделано, чтобы уложиться в ее пределы. Математика-

прикладника такой результат вполне бы устроил, но поскольку достичь его весьма непросто, он готов занять половинчатую позицию — пусть будет предложен какой-то алгоритм, а о точности результата можно судить и по каким-либо косвенным признакам.

Сказанное далеко не исчерпывает особенностей ни классического, ни конструктивного подхода. Но уже отсюда видно, что способ вычисления значения формулы вида $\forall x_i \mathcal{A}$, который мы описали выше, имеет неконструктивный характер. Таким образом, мы заняли классическую позицию и будем на ней стоять. Это не означает, что автор является приверженцем этой позиции, скорее наоборот. Но если сравнивать математические результаты, полученные классическими и конструктивными средствами, то обнаруживается нечто похожее на то, что проявилось при сравнении обычной арифметики с арифметикой, в которой существует «максимальное» натуральное число N. Конструктивные определения математических понятий формулируются более громоздко, получаемые (с бóльшим трудом) результаты сопровождаются бóльшим количеством условий и оговорок и т. д. Вероятно, при этом мы больше приближаемся к реальной действительности, абстракции же классической математики дальше стоят от жизни, но человеку, имеющему тягу к абстракции, с ними проще работать, они обозримее (при прочих равных условиях), а это много значит, особенно при первом знакомстве с предметом. Замечено, кроме того, что чем больше практических приложений находит та или иная математическая дисциплина, тем более сходны, как правило, выводы, получаемые в ней классическими и конструктивными методами (т. е. тем конструктивнее она по своему содержанию). Все это и побудило нас остановиться в данном изложении элементов математической логики и теории множеств на классическом подходе. Но читатель предупрежден.

1.2.4. Значения формул

Вернемся к правилам определения значения формулы на данном наборе.

Теорема 1.2.2. *Если все переменные, свободно входящие в формулу $\mathcal{A}$, содержатся в наборе s, то значение формулы $\mathcal{A}$ на этом наборе определено и равно T или F. В противном случае значение формулы $\mathcal{A}$ не определено.*

Доказательство (индукция по числу n логических и кванторных связок в формуле $\mathcal{A}$). Базу индукции составляют правила 1) – 3) вычисления значения формулы на данном наборе. Для первых двух правил утверждение теоремы является прямым следствием соответствующих определений. Справедливость теоремы при использовании правила 3), когда формула имеет вид $P_i^k(t_1, \ldots, t_k)$, устанавливается тем же способом, который был использован на индукционном шаге доказательства теоремы 1.2.1.

Шаг индукции связан с анализом правил 4) и 5). Для формулы $\mathcal{A}$, главная связка которой — логическая, достаточно заметить, что в любой из ее частей число связок меньше, чем в самой $\mathcal{A}$, после чего переход от справедливости теоремы для частей к ее справедливости для $\mathcal{A}$ затруднений не вызывает.

Пусть, наконец, формула $\mathcal{A}$ имеет вид $\forall x_i \mathcal{B}$ или $\exists x_i \mathcal{B}$. Формула $\mathcal{B}$ содержит $n-1$ связку и для нее, по предположению индукции, теорема верна. Каждое свободное вхождение переменной в формулу $\mathcal{A}$ — это свободное вхождение той же переменной в формулу $\mathcal{B}$. Кроме них формула $\mathcal{B}$ может содержать свободные вхождения только переменной x_i. Для определения значения формулы $\mathcal{A}$ на наборе s следует рассмотреть значения формулы $\mathcal{B}$ на всех x_i-вариациях s' набора s. По определению, переменная x_i обязательно содержится в наборе s', даже если она не содержалась в наборе s. Но тогда, если все свободные переменные формулы $\mathcal{A}$ содержатся в наборе s, то все свободные переменные $\mathcal{B}$ содержатся в наборе s' и, по предположению индукции, значение формулы $\mathcal{B}$ определено на любом таком наборе. При этом, по правилу 5) определено и значение формулы $\mathcal{A}$ на наборе s, и этим значением может быть только T или F. Если же одна из свободных переменных формулы $\mathcal{A}$ не содержится в наборе s, то эта же переменная не содержится ни в каком из наборов s', значение $V_{s'}(\mathcal{B})$ не определено, а вместе с ним не определено и $V_s(\mathcal{A})$. $\lhd$

Следующая теорема устанавливает важное свойство логических формул, которое следует учитывать при всякой попытке записать некоторое математическое утверждение в виде такой формулы.

Теорема 1.2.3. *Значение логической формулы зависит только от значений переменных, входящих в эту формулу свободно.*

Доказательство. Как и для предыдущей теоремы, оно требует индукции по числу n логических и кванторных связок в формуле. Остановимся лишь на наиболее существенных его моментах.

Пусть $\mathcal{A}$ — рассматриваемая формула, s_1 и s_2 — такие наборы значений переменных, входящих в формулу $\mathcal{A}$, что всем переменным, входящим в нее свободно, в этих наборах сопоставлены одни и те же значения. Теорема утверждает, что $V_{s_1}(\mathcal{A}) = V_{s_2}(\mathcal{A})$. Достаточно рассмотреть случай, когда набор s_2 — это x_i-вариация набора s_1 для переменной x_i, не свободной в формуле $\mathcal{A}$. Общий случай можно свести к этому, рассматривая цепочку наборов, первый из которых совпадает с s_1, последний — с s_2 и каждый следующий является вариацией предыдущего.

Для всех подформул $\mathcal{B}$ формулы $\mathcal{A}$ надо установить, что

$$V_{s_1}(\mathcal{B}) = V_{s_2}(\mathcal{B}). \tag{3}$$

В доказательстве наиболее труден случай, когда $\mathcal{B}$ имеет вид $\forall x_j \mathcal{C}$ или $\exists x_j \mathcal{C}$. В формуле $\mathcal{C}$ переменная x_i не свободна, как и в формуле $\mathcal{B}$. Требуется рассмотреть значение формулы $\mathcal{C}$ на всех x_j-вариациях s_1' набора s_1, а также на всех x_j-вариациях s_2' набора s_2. Эти вариации можно объединить в пары так, чтобы наборы s_1' и s_2' отличались друг от друга лишь значением переменной x_i. При этом s_2' оказывается x_i-вариацией s_1'. Равенство $V_{s_1'}(\mathcal{C}) = V_{s_2'}(\mathcal{C})$ выражает то же самое, что и утверждение теоремы, но для подформулы $\mathcal{C}$, содержащей на одну связку меньше, чем формула $\mathcal{B}$, и заведомо меньше связок, чем $\mathcal{A}$. По предположению индукции оно обязано выполняться. Для определения значений частей равенства (3) мы получаем одну и ту же совокупность значений формулы

$\mathcal{C}$, что и доказывает его справедливость. Так проводится индукционный шаг для подформул указанного вида. $\lhd$

1.2.5. Примеры

Покажем, как используется язык логических формул для записи различных содержательных утверждений. Начнем не с математических утверждений, а с отношений родства. Подразумеваемая интерпретация имеет своей предметной областью множество ныне живущих и ранее живших людей. Вместо стандартных обозначений x_1, x_2, ... будем использовать буквы x, y, z, u, v в качестве предметных переменных, обозначающих отдельных людей. Нам понадобятся следующие предикаты: $M(x)$ означает, что x — мужчина, $P(x, y)$ — что x — один из родителей y, $S(x, y)$ — что x и y — супруги (при этом x может быть мужем, а y — женой, или наоборот), $E(x, y)$ — что x и y — один и тот же человек. Таким образом, M — это одноместная предикатная буква, P, S и E — двухместные.

Как записать, что x и y — брат и сестра (понимая под этим, что у них есть хотя бы один общий родитель)? В этом словесном утверждении содержится указание на пол как x, так и y, так что составными частями формулы должны быть: $M(x)$ и $\neg M(y)$. Ясно также, что в формуле должно присутствовать обозначение, скажем — z, для общего родителя x и y. Из теоремы 1.2.3 видно, что переменная z, в отличие от x и y, не должна быть свободной в искомой формуле, так как содержание утверждения «x и y — брат и сестра» зависит от того, какие люди обозначены буквами x и y, и лишь от возможности сослаться на третьего человека, но не от того, был ли он заранее обозначен буквой z. Утверждение, что некто z является родителем как x, так и y, записывается в виде $(P(z, x) \wedge P(z, y))$, а утверждение, что у x и y есть общий родитель, — в виде $\exists z\,(P(z, x) \wedge P(z, y))$. Вся требуемая формула выглядит так: $((M(x) \wedge \neg M(y)) \wedge \exists z\,(P(z, x) \wedge P(z, y)))$.

В дальнейшем будем опускать самые внешние скобки и скобки, выделяющие пары соседних членов в последовательности членов, соединенных одним и тем же знаком ассоциативной операции — конъюнкции или дизъюнкции. Немного изменим условие и попытаемся записать в виде логической формулы утверждение «x — брат y». Поскольку пол y теперь уже неизвестен (и произволен), нужно исключить случай, когда y обозначает того же человека, что x:

$$M(x) \wedge \neg E(x, y) \wedge \exists z\,(P(z, x) \wedge P(z, y)). \tag{4}$$

А теперь определим предикат B с тем же смыслом. Поскольку требуется описать лишь свойства новой предикатной буквы, ничего не утверждая относительно конкретных лиц, в формуле (на основании той же теоремы 1.2.3) не должно быть свободных переменных (она должна быть замкнутой):

$$\forall x\,\forall y\,(B(x, y) \equiv (M(x) \wedge \neg E(x, y) \wedge \exists z\,(P(z, x) \wedge P(z, y)))). \tag{5}$$

Запишем теперь утверждение, что у каждого человека есть только одна мать. Оно также не относится к какому-либо определенному человеку, поэтому и здесь потребуется замкнутая формула:

$$\forall x\,\exists y\,(P(y, x) \wedge \neg M(y) \wedge \neg\exists z\,(P(z, x) \wedge \neg M(z) \wedge \neg E(z, y))). \tag{6}$$

Читается эта формула так: у каждого человека x есть родитель y женского пола, и не существует другого родителя z женского пола, не совпадающего с y. Можно написать и другую формулу с тем же самым смыслом, содержащую меньше отрицаний:

$$\forall x\,\exists y\,(P(y,\,x)\wedge\neg M(y)\wedge\forall z\,((P(z,\,x)\wedge\neg M(z))\supset E(z,\,y))). \tag{7}$$

Рассмотрим теперь несколько примеров из арифметики. Предметной областью теперь будет натуральный ряд. Цифра 1 будет выступать в роли предметной константы, функциональная буква m будет обозначать операцию умножения (так что $m(x,\,y)$ — это произведение чисел x и y). Буквы E и L будут обозначать отношения «равно» и «меньше».

Новый предикат D, выражающий отношение делимости (x на y), может быть введен формулой

$$D(x,\,y)\equiv\exists z\,E(x,\,m(y,\,z)), \tag{8}$$

а одноместный предикат P со смыслом «быть простым числом» вводится формулой

$$P(x)\equiv\forall y\,(D(x,\,y)\supset(E(y,\,1)\vee E(y,\,x))). \tag{9}$$

Формулы (8) и (9) в отличие от (5) и вопреки тому, что утверждалось в связи с этой формулой, не замкнуты. Но это уже дань традиции — подразумевать самые внешние кванторы всеобщности. Традиция опирается на свойство предметных переменных обозначать произвольные значения, без каких бы то ни было ограничений, если последние не оговорены явно.

После этого теорема Евклида о бесконечности множества простых чисел, т. е. о том, что не существует наибольшего простого числа, записывается в виде формулы

$$\forall x\,(P(x)\supset\exists y\,(L(x,\,y)\wedge P(y))). \tag{10}$$

Перейдем к интерпретации, предметную область которой образует множество всех точек и прямых на евклидовой плоскости. Будут использоваться предикаты: $P(x)$ — «x — это точка», $L(x)$ — «x — прямая», $I(x,\,y)$ — «x и y инцидентны», т. е. «точка x лежит на прямой y» или «прямая x проходит через точку y», $E(x,\,y)$ — «x и y совпадают», $R(x,\,y)$ — «прямые x и y параллельны», $Q(x,\,y)$ — «прямые x и y перпендикулярны», $S(x,\,y,\,z)$ — «точка y лежит на отрезке с концами x и z», $K(x,\,y,\,u,\,v)$ — «отрезок с концами x и y конгруэнтен отрезку с концами u и v».

Утверждение «точки x, y, z лежат на одной прямой» записывается в виде: $P(x)\wedge P(y)\wedge P(z)\wedge\exists u\,(L(u)\wedge I(x,\,u)\wedge I(y,\,u)\wedge I(z,\,u))$, или, учитывая, что из трех точек на прямой одна обязательно принадлежит отрезку с концами в двух других точках: $S(x,\,y,\,z)\vee S(y,\,z,\,x)\vee S(z,\,x,\,y)$.

Утверждение «x — проекция точки y на прямую u» можно записать так: $P(x)\wedge P(y)\wedge L(u)\wedge I(x,\,u)\wedge\exists v\,(L(v)\wedge I(x,\,v)\wedge I(y,\,v)\wedge Q(u,\,v))$.

Более сложно записывается утверждение: «прямая u является биссектрисой угла xyz». В нашем распоряжении нет предиката, выражающего конгруэнтность углов. Поэтому написать формулу, в точности соответствующую данному утверждению, нельзя. Но можно воспользоваться известными сведениями из геометрии и записать в виде формулы какое-нибудь утверждение, эквивалентное

данному. Например, можно опереться на то, что, выбрав на сторонах угла произвольные точки p и q, находящиеся на одинаковом расстоянии от вершины, и взяв произвольную точку r на биссектрисе угла, мы получим конгруэнтные отрезки pr и qr. Сначала опишем вспомогательный предикат R со смыслом «точка p лежит на луче, выходящем из точки x в сторону точки y»:

$$R(x, y, p) \equiv (\neg E(y, x) \wedge P(p) \wedge \neg E(p, x) \wedge (S(x, p, y) \vee S(x, y, p))).$$

А теперь и требуемое утверждение:

$$\begin{aligned} &P(x) \wedge P(y) \wedge P(z) \wedge L(u) \wedge \forall v\,(L(v) \wedge I(x, v) \wedge I(z, v) \supset \neg I(y, v)) \\ &\quad \wedge \forall p \forall q\,(R(p, y, x) \wedge R(q, y, z) \wedge K(y, p, y, q) \\ &\qquad \supset \forall r\,(P(r) \wedge I(r, u) \wedge \neg E(r, y) \supset K(p, r, q, r))). \end{aligned}$$

В первой строчке этой формулы описаны типы объектов x, y, z и u и указано, что точка y не должна лежать на (какой-либо) прямой, проходящей через точки x и z. Во второй — введены две вспомогательные точки p и q лучей yx и yz на равных расстояниях от точки y. В третьей — про произвольную точку r прямой u, отличную от y, сказано, что при этих условиях отрезки pr и qr должны быть конгруэнтны.

Рассматривая эти примеры, можно заметить, что одному и тому же содержательному утверждению могут соответствовать разные логические формулы. В примере с биссектрисой, кроме содержательной интерпретации предикатов P, L, I, E, S и K, совершенно явно были использованы некоторые известные свойства геометрических объектов. На основе других свойств можно было бы прийти к существенно иной формуле. Содержательные свойства рассматриваемых интерпретаций были использованы не столь явно и в других примерах. Но свобода в записи формул остается и там, где подобные свойства не используются. Так, в последнем примере квантор $\forall r$ можно было бы поместить не там, где он стоит, а сразу за квантором $\forall q$. После этого возможно было бы связку '$\supset$' в первом (вслед за $\forall r$ на его новом месте) вхождении заменить на '$\wedge$'. Одна из задач математической логики в том и заключается, чтобы дать возможность накапливать ранее полученные результаты (утверждения, истинность которых уже доказана) для получения новых, вводить новые обозначения (предметные константы, функциональные и предикатные буквы) для сокращения записи формул, осуществлять преобразования формул в эквивалентные им при любой интерпретации для придания им большей наглядности или для удобства их использования в новых логических выводах. Как логика со всем этим справляется, будет описано позже, в разд. 1.4.

Среди возможных формул, соответствующих данному содержательному утверждению, выбирались наиболее «естественные». Впрочем, какое свойство формулы называть естественностью, определить довольно трудно. Но одну особенность «естественных» формул все же стоит отметить. Назовем элементарные формулы, а также формулы, главная связка которых — отрицание или квантор, одночленными. «Естественная формула», начинающаяся с квантора, обычно имеет вид: $\forall x \mathcal{B}$, $\exists x \mathcal{A}$ или $\forall x(\mathcal{A} \supset \mathcal{B})$, где $\mathcal{A}$ — одночленная формула или конъюнкция одночленных формул, $\mathcal{B}$ — одночленная формула или дизъюнкция одночленных

формул. Чтобы понять, в чем заключается естественность именно таких формул, рассмотрим кванторные формулы, не имеющие указанного вида. Формула $\forall x\,(\mathcal{B}_1 \wedge \mathcal{B}_2)$ по сути содержит два независимых утверждения: $\forall x\mathcal{B}_1$ и $\forall x\mathcal{B}_2$. Формула $\exists x\,(\mathcal{A}_1 \vee \mathcal{A}_2)$ оставляет впечатление незавершенности — хотелось бы знать, в каком случае (при каких дополнительных условиях) можно утверждать, что существует значение x, удовлетворяющее формуле $\mathcal{A}_1$, и в каком случае — формуле $\mathcal{A}_2$. Сходные замечания можно сделать и по поводу формулы $\exists x(\mathcal{A}_1 \supset \mathcal{A}_2)$. Более завершенной была бы формула $\mathcal{A}'_1 \supset \exists x\mathcal{A}_2$, где посылка $\mathcal{A}'_1$ не содержит вхождений переменной x, относительно которой утверждается существование значения со свойством $\mathcal{A}_2$, и поэтому может быть вынесена из-под квантора.

1.2.6. Выполнимость и общезначимость

Формула $\mathcal{A}$ называется *истинной* в интерпретации I, если ее значение в этой интерпретации на любом наборе s равно T: $V_{\mathrm{I},s}(\mathcal{A}) = T$. Формула $\mathcal{A}$ называется *ложной* в интерпретации I, если она в этой интерпретации на любом наборе s принимает значение F: $V_{\mathrm{I},s}(\mathcal{A}) = F$.

Формула не обязана быть истинной или ложной в данной интерпретации. Так, формула (1.2.5–4) выражала утверждение, относящееся к произвольным лицам, обозначенным буквами x и y. В силу произвола при выборе этих лиц эта формула могла оказаться истинной при одном выборе и ложной — при другом. Такая неопределенность нежелательна, она идет вразрез с традицией, отмеченной в связи с формулами (1.2.5–8) и (1.2.5–9). Формула (1.2.5–5) описывала правило вычисления значения вновь вводимого предиката и поэтому обязана была оставаться истинной в любых условиях. Точно так же формула (1.2.5–7) должна быть истинной в любых условиях по своему содержательному смыслу. Однако она может оказаться и ложной, если интерпретация, в которой мы ее пытаемся использовать, сама чересчур конкретна, например, относится к еще неотлаженной программе или к неполной базе данных (а база данных, описывающая отношения родства, и не может быть полной, так как невозможно проследить эти отношения до прародительницы Евы).

Истинными в той сугубо абстрактной интерпретации, которая рассматривалась в разд. 1.2.5, должны быть и формулы (1.2.5–8), (1.2.5–9) и (1.2.5–10). Формула $\forall x_1\,\exists x_2 P_1^2(x_1, x_2)$ истинна в этой интерпретации, если $P_1^2(x_1, x_2)$ интерпретируется как $x_1 = x_2$, и ложна, если $P_1^2(x_1, x_2)$ обозначает $x_1 > x_2$ (так что смысл высказывания $\forall x_1\,\exists x_2 P_1^2(x_1, x_2)$ — «для каждого натурального x_1 найдется меньшее его натуральное число x_2»). Но это уже другая интерпретация.

Из теоремы 1.2.3 следует, что всякая замкнутая формула на любом наборе принимает одно и то же значение, так как она не содержит свободных переменных. Поэтому в каждой отдельной интерпретации замкнутая формула либо истинна, либо ложна, но, как только что было показано, она может быть истинна в одной и ложна в другой интерпретации. Напротив, наудачу взятая незамкнутая формула обычно принимает на разных наборах разные значения и поэтому не истинна и не ложна в некоторой данной интерпретации.

Особый интерес в логике представляют формулы, значения которых не зависят ни от набора, ни от интерпретации. Они выражают утверждения истинные (или ложные) независимо от предмета разговора, в силу только своей собственной структуры. Такие тривиально истинные (ложные) утверждения могут служить опорными пунктами в любой схеме рассуждений, в любом доказательстве.

Формула $\mathcal{F}$ называется *выполнимой*, если она истинна хотя бы в одной интерпретации, и *невыполнимой* (*противоречивой*), если она ложна в любой интерпретации. Формула называется *общезначимой* (*тавтологичной* или просто *тавтологией*), если она истинна в любой интерпретации. Ясно, что общезначимая формула выполнима и что, если $\mathcal{F}$ общезначима, то формула $\neg\mathcal{F}$ невыполнима. Если формула $\mathcal{F}$ составлена из формул $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$,..., соединенных только логическими связками, то ее общезначимость или невыполнимость (если одна из них имеет место) может быть установлена приемом, который представлен в следующей таблице на примере формулы $\mathcal{F}$ вида $(\mathcal{A}\supset\mathcal{B})\equiv(\neg\mathcal{B}\supset\neg\mathcal{A})$:

$V(\mathcal{A})$	$V(\mathcal{B})$	$V(\mathcal{A}\supset\mathcal{B})$	$V(\neg\mathcal{A})$	$V(\neg\mathcal{B})$	$V(\neg\mathcal{B}\supset\neg\mathcal{A})$	$V(\mathcal{F})$
T	T	T	F	F	T	T
T	F	F	F	T	F	T
F	T	T	T	F	T	T
F	F	T	T	T	T	T

Предполагается, что каждая из формул $\mathcal{A}$ и $\mathcal{B}$ истинна или ложна в произвольной интерпретации I. Для каждой из четырех возможных интерпретаций в таблице вычислено значение формулы $\mathcal{F}$, продвигаясь изнутри наружу. Во всех случаях значение $V(\mathcal{F})$ равно T. Так как никакие свойства набора s не были востребованы в вычислениях, формула $\mathcal{F}$ истинна в любой интерпретации I, т. е. общезначима.

Еще пример: установим общезначимость формулы

$$(\mathcal{A}\supset(\mathcal{B}\supset\mathcal{C}))\supset((\mathcal{A}\supset\mathcal{B})\supset(\mathcal{A}\supset\mathcal{C}))$$

при тех же предположениях относительно входящих в нее подформул, обозначенных $\mathcal{A}$, $\mathcal{B}$ и $\mathcal{C}$. В следующей таблице для всех восьми возможных комбинаций значений этих подформул вычислены значения всех остальных подформул.

$(\mathcal{A}$	$\supset$	$(\mathcal{B}$	$\supset$	$\mathcal{C}))$	$\supset$	$((\mathcal{A}\supset\mathcal{B})$	$\supset$	$(\mathcal{A}\supset\mathcal{C}))$
1	3	1	2	1	7	4	6	5
T	T	T	T	T	T	T	T	T
T	F	T	F	F	T	T	F	F
T	T	F	T	T	T	F	T	T
T	T	F	T	F	T	F	T	F
F	T	T	T	T	T	T	T	T
F	T	T	F	F	T	T	T	T
F	T	F	T	T	T	T	T	T
F	T	F	T	F	T	T	T	T

В первой строке таблицы выписана только сама данная формула, исходные значения $\mathcal{A}$, $\mathcal{B}$ и $\mathcal{C}$ размещены только под их первыми вхождениями, а вычислен-

ные значения остальных подформул — под их главными связками. Номера во второй строке таблицы указывают на порядок вычисления значений. В столбце под номером 7 вычислены значения всей формулы. Поскольку эти значения оказались равными T для всех восьми комбинаций значений подформул $\mathcal{A}$, $\mathcal{B}$ и $\mathcal{C}$, формула общезначима.

Если формула $\mathcal{F}$ содержит n подформул $\mathcal{A}_1, \ldots, \mathcal{A}_n$, соединенных логическими связками, причем вид этих подформул не расшифрован, то для проверки формулы $\mathcal{F}$ на общезначимость надо рассмотреть 2^n комбинаций возможных значений формул $\mathcal{A}_1, \ldots, \mathcal{A}_n$ и для каждой комбинации вычислить значение формулы $\mathcal{F}$. Объем этой работы велик, поэтому такие вычисления желательно заменить рассуждениями, отдельные части которых охватывают по нескольку случаев сразу.

Докажем, например, что формула $\mathcal{F}$, имеющая вид

$$(\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{A} \supset \mathcal{C}) \supset (\mathcal{A} \supset (\mathcal{B} \wedge \mathcal{C}))),$$

общезначима. Допустим, что это не так. Тогда должна найтись интерпретация I, в которой $\mathcal{F}$ не истинна. Будем рассматривать только такую интерпретацию. По-прежнему считаем, что значения подформул $\mathcal{A}$, $\mathcal{B}$ и $\mathcal{C}$, а значит — и всех прочих, не зависят от набора. Поскольку формула $\mathcal{F}$ — это импликация и, по допущению $V(\mathcal{F}) = F$, то значения ее частей должны быть: $V(\mathcal{A} \supset \mathcal{B}) = T$, $V((\mathcal{A} \supset \mathcal{C}) \supset (\mathcal{A} \supset (\mathcal{B} \wedge \mathcal{C}))) = F$ (все значения берутся в интерпретации I). Точно так же заключаем, что $V(\mathcal{A} \supset \mathcal{C}) = T$, $V(\mathcal{A} \supset (\mathcal{B} \wedge \mathcal{C})) = F$ и $V(\mathcal{A}) = T$, $V(\mathcal{B} \wedge \mathcal{C}) = F$. Далее, из $V(\mathcal{A}) = T$ и $V(\mathcal{A} \supset \mathcal{B}) = T$ следует, что $V(\mathcal{B}) = T$, а $V(\mathcal{A}) = T$ и $V(\mathcal{A} \supset \mathcal{C}) = T$ дают $V(\mathcal{C}) = T$. Из $V(\mathcal{B}) = T$ и $V(\mathcal{C}) = T$ получаем $V(\mathcal{B} \wedge \mathcal{C}) = T$, что расходится с ранее найденным равенством $V(\mathcal{B} \wedge \mathcal{C}) = F$. Полученное противоречие опровергает сделанное допущение, доказывая тем самым общезначимость формулы $\mathcal{F}$.

Этот же прием можно использовать и для доказательства общезначимости некоторых формул, содержащих произвольное число подформул. Возьмем для примера формулу $\mathcal{F}$ вида $\mathcal{G} \equiv \mathcal{H}$, где $\mathcal{G}$ — это $\mathcal{A}_1 \supset (\mathcal{A}_2 \supset \ldots \supset (\mathcal{A}_n \supset \mathcal{B}) \ldots)$, а $\mathcal{H}$ — $(\mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n) \supset \mathcal{B}$. Если она не общезначима, то в некоторой интерпретации I значения $V_1 = V(\mathcal{G})$ и $V_2 = V(\mathcal{H})$ должны быть различны. Рассмотрим по очереди два возможных случая.

а. $V_1 = T, V_2 = F$. Из второго равенства получаем: $V(\mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n) = T$, откуда $V(\mathcal{A}_1) = T, \ldots, V(\mathcal{A}_n) = T$, и $V(\mathcal{B}) = F$. Отсюда и из первого равенства последовательно находим: $V(\mathcal{A}_2 \supset \ldots \supset (\mathcal{A}_n \supset \mathcal{B}) \ldots) = T, \ldots, V(\mathcal{A}_n \supset \mathcal{B}) = T$, $V(\mathcal{B}) = T$.

б. $V_1 = F$, $V_2 = T$. Из $V_1 = F$ поочередно находим значения частей всех импликаций: $V(\mathcal{A}_1) = T$, $V(\mathcal{A}_2 \supset \ldots \supset (\mathcal{A}_n \supset \mathcal{B}) \ldots) = F, \ldots, V(\mathcal{A}_{n-1}) = T$, $V(\mathcal{A}_n \supset \mathcal{B}) = F, V(\mathcal{A}_n) = T, V(\mathcal{B}) = F$. Из этого следует $V(\mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n) = T$, так что $V_2 = F$.

Полученное в обоих случаях противоречие приводит к выводу, что интерпретаций с различными значениями $\mathcal{G}$ и $\mathcal{H}$ быть не может, следовательно, формула $\mathcal{F}$ общезначима.

Сходные рассуждения иногда позволяют доказывать общезначимость формул, содержащих кванторы. В качестве примера рассмотрим формулу $\mathcal{F}$ вида $\forall x\,(\mathcal{A} \supset \mathcal{B}) \supset (\exists x\mathcal{A} \supset \mathcal{B})$, в которой подформула $\mathcal{A}$ может зависеть, а подформула $\mathcal{B}$ не зависит от переменной x (т. е. не содержит свободных вхождений этой переменной).

Если $\mathcal{F}$ не общезначима, то найдутся такие интерпретация I и набор s, что $V_s(\forall x\,(\mathcal{A} \supset \mathcal{B})) = T$, $V_s(\exists x\mathcal{A} \supset \mathcal{B}) = F$, $V_s(\exists x\mathcal{A}) = T$, $V_s(\mathcal{B}) = F$. Индекс I у буквы V опущен, а индекс s сохранен, так как сейчас нам предстоит рассмотреть для интерпретации I совокупность x-вариаций набора s. Из равенства $V_s(\exists x\mathcal{A}) = T$ следует, что в этой совокупности найдется такой набор s', что $V_{s'}(\mathcal{A}) = T$. Формула $\mathcal{B}$ не содержит свободных вхождений переменной x, так что $V_{s'}(\mathcal{B}) = V_s(\mathcal{B}) = F$ (по теореме 1.2.3). Из $V_{s'}(\mathcal{A}) = T$ и $V_{s'}(\mathcal{B}) = F$ следует $V_{s'}(\mathcal{A} \supset \mathcal{B}) = F$, а отсюда — $V_s(\forall x\,(\mathcal{A} \supset \mathcal{B})) = F$. Это противоречит одному из ранее установленных равенств, поэтому формула $\mathcal{F}$ общезначима.

1.2.7. Некоторые общезначимые формулы

В этом разделе будут даны различные общезначимые формулы, знание которых столь же полезно и даже необходимо, как знание формул (1.1.1–2) и им подобных в элементарной алгебре. Все они будут содержать подформулы, обозначенные буквами $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ..., возможно, снабженными индексами. Эти подформулы, за исключением некоторых специально оговоренных случаев, могут быть совершенно произвольными. Поэтому формулы данного раздела выражают свойства не этих подформул, а тех логических операций (связок), с помощью которых они построены из этих подформул, подобно тому, как формула $x + y = y + x$ выражает свойство операции сложения, а не чисел, обозначенных буквами x и y. Свойства логических операций часто называют законами логики, некоторые законы имеют свои названия, которые будут указаны в скобках.

Общезначимость практически всех приведенных ниже формул может быть установлена методами предыдущего раздела. На этом мы почти не будем останавливаться.

Свойства дизъюнкции аналогичны свойствам конъюнкции. Это видно хотя бы из столбцов $V(\mathcal{A} \wedge \mathcal{B})$ и $V(\mathcal{A} \vee \mathcal{B})$ таблицы на с. 23, переходящих один в другой при перемене местами знаков '$\wedge$' и '$\vee$' и при одновременной замене значения T на F и наоборот. Это свойство, называемое *двойственностью* операций конъюнкции и дизъюнкции, позволяет многим общезначимым формулам, содержащим связки '$\wedge$' и '$\vee$' (и константы T и F), сопоставить аналогичные формулы, где эти связки (и константы) меняются местами. Такие формулы будут группироваться попарно.

Итак, начинаем наш список общезначимых формул.

$$(\mathcal{A} \wedge \mathcal{A}) \equiv \mathcal{A}, \qquad (\mathcal{A} \vee \mathcal{A}) \equiv \mathcal{A}$$

— свойство идемпотентности конъюнкции и дизъюнкции или просто идемпотентность (в переводе: тождество степеней) этих операций.

$$(\mathcal{A} \wedge T) \equiv \mathcal{A}, \qquad (\mathcal{A} \vee F) \equiv \mathcal{A}$$
$$(\mathcal{A} \wedge F) \equiv F, \qquad (\mathcal{A} \vee T) \equiv T$$

— частные случаи правила вычисления значений конъюнкции и дизъюнкции, специального названия не имеют.

$$(\mathcal{A} \wedge \mathcal{B}) \equiv (\mathcal{B} \wedge \mathcal{A}), \qquad (\mathcal{A} \vee \mathcal{B}) \equiv (\mathcal{B} \vee \mathcal{A})$$

— коммутативность конъюнкции и дизъюнкции.

$$(\mathcal{A} \wedge (\mathcal{B} \wedge \mathcal{C})) \equiv ((\mathcal{A} \wedge \mathcal{B}) \wedge \mathcal{C}), \qquad (\mathcal{A} \vee (\mathcal{B} \vee \mathcal{C})) \equiv ((\mathcal{A} \vee \mathcal{B}) \vee \mathcal{C})$$

— ассоциативность конъюнкции и дизъюнкции. Благодаря ассоциативности многочастные конъюнкции и дизъюнкции (впрочем, здесь два — это уже много) можно писать вообще без скобок: $\mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n$ и $\mathcal{A}_1 \vee \mathcal{A}_2 \vee \ldots \vee \mathcal{A}_n$, что мы и делали в предыдущем разделе без должного обоснования. Любой порядок выполнения операций приводит к одному и тому же результату, а именно: многочастная конъюнкция $\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n$ принимает значение T тогда и только тогда, когда все ее части имеют значение T, дизъюнкция $\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n$ имеет значение F в том и только в том случае, когда значения всех ее частей равны F.

Упражнение 1.2.3. Пусть имеется список из n формул: $\mathcal{A}_1\ \mathcal{A}_2\ \ldots\ \mathcal{A}_n$. Вставим между любыми двумя соседними формулами $\mathcal{A}_i$ и $\mathcal{A}_{i+1}$ знак '$\wedge$' и заключим эту пару формул в скобки: $(\mathcal{A}_i \wedge \mathcal{A}_{i+1})$, уменьшив тем самым длину списка на единицу. Повторим это еще $n-2$ раза, чтобы получить одну формулу. Доказать, что значение полученной формулы (на некотором наборе) зависит только от значений формул $\mathcal{A}_1, \ldots, \mathcal{A}_n$ и не зависит от порядка, в котором образовывались конъюнкции соседних формул.

Из ассоциативности и коммутативности вытекает общезначимость формул:

$$\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n \equiv \mathcal{A}_{i_1} \wedge \ldots \wedge \mathcal{A}_{i_n}, \qquad \mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n \equiv \mathcal{A}_{i_1} \vee \ldots \vee \mathcal{A}_{i_n},$$

где $(i_1, \ldots, i_n)$ — произвольная перестановка чисел $1, \ldots, n$.

Чтобы еще более сократить число явных скобок при записи формул, принимают следующий порядок (по убыванию) старшинства логических связок:

1) отрицание и кванторные связки (это было предопределено уже сформулированными правилами записи формул),
2) конъюнкция,
3) дизъюнкция,
4) импликация,
5) эквивалентность.

Операция, встречающаяся в этом списке раньше другой (старшая из двух), и исполняется раньше нее, даже если это не предписано скобками.

Продолжаем список общезначимых формул:

$$(\mathcal{A} \wedge (\mathcal{B} \vee \mathcal{C})) \equiv ((\mathcal{A} \wedge \mathcal{B}) \vee (\mathcal{A} \wedge \mathcal{C})), \qquad (\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C})) \equiv ((\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C}))$$

или, в соответствии с введенными правилами старшинства связок:

$$\mathcal{A}\wedge(\mathcal{B}\vee\mathcal{C})\equiv\mathcal{A}\wedge\mathcal{B}\vee\mathcal{A}\wedge\mathcal{C},\qquad\mathcal{A}\vee\mathcal{B}\wedge\mathcal{C}\equiv(\mathcal{A}\vee\mathcal{B})\wedge(\mathcal{A}\vee\mathcal{C})$$

— дистрибутивность конъюнкции относительно дизъюнкции и дизъюнкции по отношению к конъюнкции. Эти формулы также допускают естественные обобщения, например:

$$(\mathcal{A}_1\vee\ldots\vee\mathcal{A}_m)\wedge(\mathcal{B}_1\vee\ldots\vee\mathcal{B}_n)\\ \equiv\mathcal{A}_1\wedge\mathcal{B}_1\vee\ldots\vee\mathcal{A}_1\wedge\mathcal{B}_n\vee\ldots\vee\mathcal{A}_m\wedge\mathcal{B}_1\vee\ldots\vee\mathcal{A}_m\wedge\mathcal{B}_n.$$

Далее можно привести группу формул, в самом общем виде описывающих, что (и при каких условиях) следует из конъюнкции или дизъюнкции и из чего следует конъюнкция или дизъюнкция:

$$\mathcal{A}\wedge\mathcal{B}\supset\mathcal{A},\qquad\mathcal{A}\supset\mathcal{A}\vee\mathcal{B},$$
$$\mathcal{A}\wedge\mathcal{B}\supset\mathcal{B},\qquad\mathcal{B}\supset\mathcal{A}\vee\mathcal{B},$$
$$(\mathcal{C}\supset\mathcal{A})\supset((\mathcal{C}\supset\mathcal{B})\supset(\mathcal{C}\supset\mathcal{A}\wedge\mathcal{B})),\qquad(\mathcal{A}\supset\mathcal{C})\supset((\mathcal{B}\supset\mathcal{C})\supset(\mathcal{A}\vee\mathcal{B}\supset\mathcal{C})).$$

В прошлом разделе была доказана общезначимость формулы

$$\mathcal{A}_1\supset(\mathcal{A}_2\supset\ldots\supset(\mathcal{A}_n\supset\mathcal{B})\ldots)\equiv\mathcal{A}_1\wedge\mathcal{A}_2\wedge\ldots\wedge\mathcal{A}_n\supset\mathcal{B}.$$

Пользуясь этой схемой, последнюю пару формул в данной группе можно переписать в виде

$$(\mathcal{C}\supset\mathcal{A})\wedge(\mathcal{C}\supset\mathcal{B})\supset(\mathcal{C}\supset\mathcal{A}\wedge\mathcal{B}),\qquad(\mathcal{A}\supset\mathcal{C})\wedge(\mathcal{B}\supset\mathcal{C})\supset(\mathcal{A}\vee\mathcal{B}\supset\mathcal{C}).$$

Формулы этой группы следующим образом обобщаются на многочастный случай:

$$\mathcal{A}_1\wedge\ldots\wedge\mathcal{A}_n\supset\mathcal{A}_i,\qquad\mathcal{A}_i\supset\mathcal{A}_1\vee\ldots\vee\mathcal{A}_n$$

$$(\mathcal{B}\supset\mathcal{A}_1)\wedge\ldots\wedge(\mathcal{B}\supset\mathcal{A}_n)\supset(\mathcal{B}\supset\mathcal{A}_1\wedge\ldots\wedge\mathcal{A}_n),\\ (\mathcal{A}_1\supset\mathcal{B})\wedge\ldots\wedge(\mathcal{A}_n\supset\mathcal{B})\supset(\mathcal{A}_1\vee\ldots\vee\mathcal{A}_n\supset\mathcal{B}).$$

Если общезначима формула $\mathcal{A}\supset\mathcal{B}$, то формула $\mathcal{A}$ называется *более сильной*, а $\mathcal{B}$ — *более слабой* в этой паре. Следующие формулы

$$\mathcal{A}\supset\mathcal{B}\equiv(\mathcal{A}\wedge\mathcal{B}\equiv\mathcal{A}),\qquad\mathcal{A}\supset\mathcal{B}\equiv(\mathcal{A}\vee\mathcal{B}\equiv\mathcal{B})$$

носят название законов поглощения (в конъюнкции более сильный член поглощает более слабый, а в дизъюнкции наоборот). Так как конъюнкция сильнее, а дизъюнкция слабее любой из своих частей, то часто законы поглощения записывают в форме:

$$\mathcal{A}\wedge(\mathcal{A}\vee\mathcal{B})\equiv\mathcal{A},\qquad\mathcal{A}\vee\mathcal{A}\wedge\mathcal{B}\equiv\mathcal{A}.$$

Очень употребительны формулы:

$$\neg(\mathcal{A}\wedge\mathcal{B})\equiv\neg\mathcal{A}\vee\neg\mathcal{B},\qquad\neg(\mathcal{A}\vee\mathcal{B})\equiv\neg\mathcal{A}\wedge\neg\mathcal{B}$$

— законы де Моргана, проливающие дополнительный свет на двойственность операций конъюнкции и дизъюнкции.

Отрицание фигурирует также в формулах:

$\neg\neg\mathcal{A} \equiv \mathcal{A}$ — закон двойного отрицания,
$\mathcal{A} \vee \neg\mathcal{A}$ — закон исключенного третьего,
$\neg(\mathcal{A} \wedge \neg\mathcal{A})$ — закон противоречия.

Следующая группа формул описывает основные свойства импликации.

$\mathcal{A} \supset (\mathcal{B} \supset \mathcal{A})$ — закон утверждения заключения,
$\neg\mathcal{A} \supset (\mathcal{A} \supset \mathcal{B})$ — закон отрицания посылки,
$(\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})) \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C}))$ — левая самодистрибутивность,
$(\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{B} \supset \mathcal{C}) \supset (\mathcal{A} \supset \mathcal{C}))$ — транзитивность,
$\mathcal{A} \supset \mathcal{B} \equiv \neg\mathcal{B} \supset \neg\mathcal{A}$ — закон контрапозиции,
$(\neg\mathcal{B} \supset \neg\mathcal{A}) \supset ((\neg\mathcal{B} \supset \mathcal{A}) \supset \mathcal{B})$ — приведение к противоречию,
$(\mathcal{A} \supset \mathcal{B}) \supset ((\neg\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B})$ — разбор случаев.

Операции конъюнкции, дизъюнкции и импликации выражаются друг через друга и отрицание, как показывают следующие формулы:

$\mathcal{A} \wedge \mathcal{B} \equiv \neg(\neg\mathcal{A} \vee \neg\mathcal{B})$, $\mathcal{A} \vee \mathcal{B} \equiv \neg(\neg\mathcal{A} \wedge \neg\mathcal{B})$,
$\mathcal{A} \wedge \mathcal{B} \equiv \neg(\mathcal{A} \supset \neg\mathcal{B})$, $\mathcal{A} \vee \mathcal{B} \equiv \neg\mathcal{A} \supset \mathcal{B}$,
$\mathcal{A} \supset \mathcal{B} \equiv \neg(\mathcal{A} \wedge \neg\mathcal{B})$, $\mathcal{A} \supset \mathcal{B} \equiv \neg\mathcal{A} \vee \mathcal{B}$,

а также

$\neg(\mathcal{A} \supset \mathcal{B}) \equiv \mathcal{A} \wedge \neg\mathcal{B}$, $\neg(\mathcal{A} \wedge \mathcal{B}) \equiv \mathcal{A} \supset \neg\mathcal{B}$.

Основные свойства эквивалентности даются формулами:

$(\mathcal{A} \equiv \mathcal{B}) \equiv (\mathcal{A} \supset \mathcal{B}) \wedge (\mathcal{B} \supset \mathcal{A})$ — определяющее соотношение,
$(\mathcal{A} \equiv \mathcal{B}) \equiv \mathcal{A} \wedge \mathcal{B} \vee \neg\mathcal{A} \wedge \neg\mathcal{B}$,
$(\mathcal{A} \equiv \mathcal{B}) \supset ((\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{A} \equiv \mathcal{C}))$, — транзитивность,
$\neg(\mathcal{A} \equiv \mathcal{B}) \equiv (\neg\mathcal{A} \equiv \mathcal{B})$, $\neg(\mathcal{A} \equiv \mathcal{B}) \equiv (\mathcal{A} \equiv \neg\mathcal{B})$.

В вышеприведенных формулах кванторы могли содержаться неявно — внутри формул $\mathcal{A}, \mathcal{B}, \ldots$ Примеры общезначимых формул, явно содержащих кванторы, предварим замечанием относительно интерпретаций с конечной предметной областью $D = \{v_1, \ldots, v_n\}$. Пусть x — некоторая переменная, $\mathcal{A}(x)$ — формула, содержащая свободные вхождения переменной x, s — набор значений над предметной областью D. Существует лишь n различных x-вариаций s' набора s. Поэтому значение формулы $\forall x\mathcal{A}(x)$ или $\exists x\mathcal{A}(x)$ на наборе s может быть определено в результате рассмотрения лишь конечного числа значений формулы $\mathcal{A}(x)$ на наборах s'.

В этих условиях нетрудно написать формулу, не содержащую кванторов $\forall x$ и $\exists x$, значение которой на наборе s совпадает со значением формулы $\forall x\mathcal{A}(x)$ или $\exists x\mathcal{A}(x)$. Для этого введем новые предметные константы $b_1, \ldots, b_n$ и сопоставим каждому элементу v_i области D свою предметную константу b_i. Нетрудно убедиться, что значение формулы $\mathcal{A}(x)$ на x-вариации s' набора s, в которой переменной x придано значение v_i, совпадает со значением формулы $\mathcal{A}(b_i)$ на наборе s. Как всегда, формула $\mathcal{A}(b_i)$ — это результат подстановки константы b_i вместо каждого свободного вхождения переменной x в формулу $\mathcal{A}(x)$. Будем писать $\mathcal{A}_i$ вместо $\mathcal{A}(b_i)$.

Из сказанного следует, что

$$V_s(\forall x\mathcal{A}(x)) = V_s(\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n), \qquad V_s(\exists x\mathcal{A}(x)) = V_s(\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n).$$

Итак, в рассматриваемой конечной интерпретации формула $\forall x\mathcal{A}(x)$ эквивалентна формуле $\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n$, а формула $\exists x\mathcal{A}(x)$ — формуле $\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n$. Поэтому любой формуле, содержащей подобную многочастную конъюнкцию или дизъюнкцию (или и то и другое), можно сопоставить аналогичную формулу с кванторами. Если исходная (бескванторная) формула общезначима, то преобразованная формула (с кванторами) будет истинна в любой конечной интерпретации. Довольно часто она оказывается истинной в произвольной интерпретации, т. е. общезначимой. В частности, формуле

$$(\mathcal{A}_1 \supset \mathcal{B}) \wedge \ldots \wedge (\mathcal{A}_n \supset \mathcal{B}) \supset (\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n \supset \mathcal{B})$$

можно сопоставить такую кванторную формулу

$$\forall x(\mathcal{A}(x) \supset \mathcal{B}) \supset (\exists x\mathcal{A}(x) \supset \mathcal{B})$$

при условии, что формула $\mathcal{B}$ не содержит свободных вхождений переменной x. В предыдущем разделе было показано, что эта формула общезначима.

Аналогично, формуле

$$(\mathcal{B} \supset \mathcal{A}_1) \wedge \ldots \wedge (\mathcal{B} \supset \mathcal{A}_n) \supset (\mathcal{B} \supset \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n)$$

сопоставляется при том же условии кванторная формула

$$\forall x(\mathcal{B} \supset \mathcal{A}(x)) \supset (\mathcal{B} \supset \forall x\mathcal{A}(x))$$

Нетрудно проверить, что и эта формула общезначима.

Обобщая формулу $\mathcal{A}_i \wedge \ldots \wedge \mathcal{A}_n \supset \mathcal{A}_i$, можно получить формулу

$$\forall x\mathcal{A}(x) \supset \mathcal{A}(b),$$

где b — произвольная предметная константа. Эта формула общезначима. Не будет ли общезначимой более общая формула

$$\forall x\mathcal{A}(x) \supset \mathcal{A}(t),$$

если t — произвольный терм? Простой пример опровергает это предположение. Пусть $\mathcal{A}(x)$ — формула вида $\exists yP(x, y)$, а терм t имеет вид $f(y)$. Рассмотрим интерпретацию, предметная область которой — множество всех целых чисел, предикату P соответствует отношение $<$, а функциональной букве f — операция, заключающаяся в увеличении значения аргумента на 1. В обычных обозначениях формула $\forall x\mathcal{A}(x) \supset \mathcal{A}(t)$ примет вид $\forall x\,\exists y\,(x < y) \supset \exists y\,(y + 1 < y)$. Ясно, что эта формула ложна в рассматриваемой интерпретации (ее посылка истинна, а заключение — ложно). Понятна и причина, по которой формула $\exists y\,(x < y)$, истинная при любом значении x, превратилась в ложную формулу при подстановке терма $y + 1$ вместо x — этот терм содержит y и поэтому не свободен для подстановки вместо x в формулу $\exists y\,(x < y)$, так как свободное вхождение x расположено в области действия квантора $\exists y$.

Если же потребовать, чтобы терм t был свободен для переменной x в формуле $\mathcal{A}(x)$, то формула

$$\forall x\,\mathcal{A}(x) \supset \mathcal{A}(t) \tag{11}$$

оказывается общезначимой. Однако доказательство этого утверждения не тривиально, и ему будет уделено место ниже, в разд. 1.2.8.

Кванторным аналогом формулы $\mathcal{A}_i \supset \mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n$ является формула

$$\mathcal{A}(t) \supset \exists x \mathcal{A}(x),$$

общезначимая при том же условии — терм t свободен для переменной x в формуле $\mathcal{A}(x)$.

Законы де Моргана допускают обобщение как на многочастный случай:

$$\neg(\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n) \equiv \neg\mathcal{A}_1 \vee \ldots \vee \neg\mathcal{A}_n, \qquad \neg(\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n) \equiv \neg\mathcal{A}_1 \wedge \ldots \wedge \neg\mathcal{A}_n,$$

так и на кванторные формулы:

$$\neg\forall x \mathcal{A} \equiv \exists x \neg\mathcal{A}, \qquad \neg\exists x \mathcal{A} \equiv \forall x \neg\mathcal{A}.$$

Эти две эквивалентности так часто используются в математике (особенно в ходе доказательств от противного), что стоит уделить им еще немного внимания. Пусть Q — один из кванторных символов '$\forall$' или '$\exists$'. Обозначим через $\bar{\mathrm{Q}}$ двойственный символ ('$\exists$' или '$\forall$' соответственно). Рассмотрим отрицание формулы, начинающейся с нескольких кванторов (вместе образующих так называемую кванторную приставку): $\neg \mathrm{Q}_1 x_1 \ldots \mathrm{Q}_m x_m \mathcal{A}$. Применяя m раз обобщенные законы де Моргана, убедимся, что справедлива эквивалентность:

$$\neg \mathrm{Q}_1 x_1 \ldots \mathrm{Q}_m x_m \mathcal{A} \equiv \bar{\mathrm{Q}}_1 x_1 \ldots \bar{\mathrm{Q}}_m x_m \neg\mathcal{A}.$$

Вспомним также замечание из разд. 1.2.5 относительно «естественных» формул, где под знаком квантора всеобщности часто стоит формула вида $\mathcal{A} \supset \mathcal{B}$, причем формула $\mathcal{A}$ выделяет некоторый класс интересующих нам объектов среди всех возможных, а $\mathcal{B}$ выражает некоторое утверждение об общем свойстве всех объектов этого класса. Так же часто встречаются формулы вида $\exists x(\mathcal{A} \wedge \mathcal{B})$, утверждающие существование объекта со свойством $\mathcal{B}$ среди объектов класса $\mathcal{A}$.

Рассмотрим, как выглядят отрицания подобных утверждений. По общим формулам

$$\neg\forall x(\mathcal{A} \supset \mathcal{B}) \equiv \exists x \neg(\mathcal{A} \supset \mathcal{B}), \qquad \neg\exists x(\mathcal{A} \wedge \mathcal{B}) \equiv \forall x \neg(\mathcal{A} \wedge \mathcal{B}).$$

Но у нас были формулы

$$\neg(\mathcal{A} \supset \mathcal{B}) \equiv \mathcal{A} \wedge \neg\mathcal{B}, \qquad \neg(\mathcal{A} \wedge \mathcal{B}) \equiv \mathcal{A} \supset \neg\mathcal{B}.$$

С их помощью получаем эквивалентности:

$$\neg\forall x(\mathcal{A} \supset \mathcal{B}) \equiv \exists x(\mathcal{A} \wedge \neg\mathcal{B}), \qquad \neg\exists x(\mathcal{A} \wedge \mathcal{B}) \equiv \forall x(\mathcal{A} \supset \neg\mathcal{B}).$$

позволяющие заменять отрицания «естественных» формул формулами также «естественного» вида. Эти два приема преобразования формул часто комбинируются в разных сочетаниях, например:

$$\neg\forall x(\mathcal{A} \supset \exists y(\mathcal{B} \wedge \forall z(\mathcal{C} \supset (\mathcal{G} \supset \mathcal{H})))) \equiv \exists x(\mathcal{A} \wedge \forall y(\mathcal{B} \supset \exists z(\mathcal{C} \wedge \mathcal{G} \wedge \neg\mathcal{H}))).$$

1.2.8. Теоремы об истинности и общезначимости

Теорема 1.2.4. *Если формулы $\mathcal{A}$ и $\mathcal{A} \supset \mathcal{B}$ принимают значение T на некотором наборе s значений переменных, то и формула $\mathcal{B}$ принимает на наборе s значение T.*

Доказательство. По условию $V_s(\mathcal{A}) = T$. На наборе s значение формулы $\mathcal{B}$ определено, иначе на нем не было бы определено значение формулы $\mathcal{A} \supset \mathcal{B}$. Из равенства $V_s(\mathcal{B}) = F$ следовало бы $V_s(\mathcal{A} \supset \mathcal{B}) = F$, что противоречит условию. Следовательно, $V_s(\mathcal{B}) = T$. $\lhd$

Эта теорема имеет очевидное

Следствие 1.2.5. *Если формулы $\mathcal{A}$ и $\mathcal{A} \supset \mathcal{B}$ истинны в некоторой интерпретации, то и формула $\mathcal{B}$ истинна в этой интерпретации. Если формулы $\mathcal{A}$ и $\mathcal{A} \supset \mathcal{B}$ общезначимы, то и формула $\mathcal{B}$ общезначима.*

Теорема 1.2.6. *Если формула $\mathcal{A}$ истинна в некоторой интерпретации* I, *то и формула $\forall x \mathcal{A}$ истинна в этой интерпретации (x — произвольная предметная переменная).*

Доказательство. Предположим, что формула $\forall x \mathcal{A}$ ложна в интерпретации I, т. е. существует такой набор s, что $V_s(\forall x \mathcal{A}) = F$. Но тогда должна найтись такая x-вариация s' набора s, что $V_{s'}(\mathcal{A}) = F$. А это означает, что формула $\mathcal{A}$ не истинна в интерпретации I, чего по условию быть не может. Следовательно, формула $\forall x \mathcal{A}$ истинна в интерпретации I. $\lhd$

Отсюда получаем

Следствие 1.2.7. *Если формула $\mathcal{A}$ общезначима, то и формула $\forall x \mathcal{A}$ общезначима.*

Теорема 1.2.8 (об эквивалентности). *Пусть формула $\mathcal{A}_2$ получается из формулы $\mathcal{A}_1$ в результате замены одного или нескольких вхождений подформулы $\mathcal{B}$ формулы $\mathcal{A}_1$ на формулу $\mathcal{C}$. Пусть $y_1, \ldots, y_m$ — все свободные переменные формул $\mathcal{B}$ и $\mathcal{C}$, которые связаны в формуле $\mathcal{A}_1$. Тогда формула*

$$\forall y_1 \ldots y_m (\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{A}_1 \equiv \mathcal{A}_2) \tag{12}$$

общезначима.

Доказательство. Предположим, что в некоторой интерпретации формула (12) не истинна, т. е. существует такой набор s, что

$$V_s(\forall y_1 \ldots \forall y_m (\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{A}_1 \equiv \mathcal{A}_2)) = F.$$

Тогда

$$V_s(\forall y_1 \ldots \forall y_m (\mathcal{B} \equiv \mathcal{C})) = T, \tag{13}$$

$$V_s(\mathcal{A}_1 \equiv \mathcal{A}_2) = F. \tag{14}$$

Из равенства (13) следует, что $V_{s_1}(\mathcal{B} \equiv \mathcal{C}) = T$, т. е. что

$$V_{s_1}(\mathcal{B}) = V_{s_1}(\mathcal{C}), \tag{15}$$

где s_1 — любая $(y_1, \ldots, y_m)$-вариация набора s, т. е. набор, возможно, отличающийся от набора s лишь значениями переменных $y_1, \ldots, y_m$.

Пусть $\mathcal{D}_1$ — какое-либо вхождение подформулы в формулу $\mathcal{A}_1$, содержащее хотя бы одно из вхождений формулы $\mathcal{B}$ в формулу $\mathcal{A}_1$, заменяемых на $\mathcal{C}$ при преобразовании $\mathcal{A}_1$ в $\mathcal{A}_2$. Обозначим через $\mathcal{D}_2$ соответствующее вхождение подформулы в формулу $\mathcal{A}_2$. Таким образом, $\mathcal{D}_2$ получается из $\mathcal{D}_1$ заменой на $\mathcal{C}$

всех тех вхождений $\mathcal{B}$ в $\mathcal{D}_1$, которые заменяются на $\mathcal{C}$ при переходе от $\mathcal{A}_1$ к $\mathcal{A}_2$. Покажем, что

$$V_{s_2}(\mathcal{D}_1) = V_{s_2}(\mathcal{D}_2), \tag{16}$$

где s_2 — некоторая вариация набора s по переменным, связанным в $\mathcal{A}_1$.

Доказательство проведем индукцией по числу k логических и кванторных связок в формуле $\mathcal{D}_1$, не принадлежащих заменяемым вхождениям формулы $\mathcal{B}$. Если $k = 0$, то $\mathcal{D}_1$ обязана совпадать с одним из заменяемых вхождений $\mathcal{B}$ в $\mathcal{A}_1$, при этом $\mathcal{D}_2$ — это $\mathcal{C}$. По каждому набору s_2 можно построить набор s_1, в котором переменные $y_1, \ldots, y_m$ принимают такие же значения, как в наборе s_2, а все остальные переменные — как в наборе s. Если при этом переменная x не связана в формуле $\mathcal{A}_1$, то ее значения во всех трех наборах s, s_1 и s_2 одинаковы. Таким образом, наборы s_1 и s_2 могут различаться лишь значениями переменных, связанных в формуле $\mathcal{A}_1$, но не входящих в список $y_1, \ldots, y_m$, т. е. не свободных ни в $\mathcal{B}$, ни в $\mathcal{C}$. По теореме 1.2.3 $V_{s_2}(\mathcal{B}) = V_{s_1}(\mathcal{B})$, $V_{s_2}(\mathcal{C}) = V_{s_1}(\mathcal{C})$, и в силу равенства (15) $V_{s_2}(\mathcal{B}) = V_{s_2}(\mathcal{C})$. Но это и есть равенство (16) для рассматриваемого случая.

Пусть $k > 0$. Предположим, что для всех формул, которые содержат меньше чем k интересующих нас связок, равенство, аналогичное равенству (16), справедливо. Если главная связка формулы $\mathcal{D}_1$ — это '$\neg$', то $\mathcal{D}_1$ имеет вид $\neg\mathcal{D}'_1$. Все вхождения $\mathcal{B}$ в $\mathcal{D}_1$ входят и в $\mathcal{D}'_1$, число связок в формуле $\mathcal{D}'_1$, не принадлежащих этим вхождениям, равно $k-1$. Формула $\mathcal{D}_2$ имеет вид $\neg\mathcal{D}'_2$, где $\mathcal{D}'_2$ получается из $\mathcal{D}'_1$ так же, как $\mathcal{D}_2$ из $\mathcal{D}_1$. По предположению индукции $V_{s_2}(\mathcal{D}'_1) = V_{s_2}(\mathcal{D}'_2)$, но отсюда сразу следует справедливость равенства (16).

Сходными рассуждениями устанавливается его справедливость и тогда, когда главная связка в формуле $\mathcal{D}_1$ — это одна из двухместных логических связок '$\wedge$', '$\vee$', '$\supset$' или '$\equiv$'.

Пусть теперь главная связка в формуле $\mathcal{D}_1$ — это кванторная связка $\mathrm{Q}\,x$, где Q — это '$\forall$' или '$\exists$', так что равенство (16) имеет вид $V_{s_2}(\mathrm{Q}\,x\mathcal{D}'_1) = V_{s_2}(\mathrm{Q}\,x\mathcal{D}'_2)$. Значение в левой (или в правой) части этого равенства определяется по совокупности значений $V_{s'_2}(\mathcal{D}'_1)$ (соответственно $V_{s'_2}(\mathcal{D}'_2)$), где s'_2 — произвольная x-вариация набора s_2. Переменная x связана в формуле $\mathcal{D}_1$, а следовательно, и в $\mathcal{A}_1$, так как $\mathcal{D}_1$ — подформула формулы $\mathcal{A}_1$. Это значит, что s'_2, так же как набор s_2, является вариацией набора s по переменным, связанным в формуле $\mathcal{A}_1$. По предположению индукции $V_{s'_2}(\mathcal{D}'_1) = V_{s'_2}(\mathcal{D}'_2)$, следовательно, значение $V_{s_2}(\mathrm{Q}\,x\mathcal{D}'_1)$ определяется по той же самой совокупности значений, что и значение $V_{s_2}(\mathrm{Q}\,x\mathcal{D}'_2)$. Отсюда вытекает справедливость равенства (16) и в этом случае.

По индукции равенство (16) должно быть верно всегда, в том числе и тогда, когда $\mathcal{D}_1$ — это сама формула $\mathcal{A}_1$. При этом $\mathcal{D}_2$ — это формула $\mathcal{A}_2$ и равенство (16) принимает вид $V_{s_2}(\mathcal{A}_1) = V_{s_2}(\mathcal{A}_2)$. Но набор s_2 может не отличаться от набора s. Возникает противоречие с равенством (14), которое и доказывает теорему. $\lhd$

Теорема 1.2.9. *Если (при тех же обозначениях) в некоторой интерпретации* I *истинна формула* $\mathcal{B} \equiv \mathcal{C}$, *то и формула* $\mathcal{A}_1 \equiv \mathcal{A}_2$ *истинна в* I.

Если, кроме того, в I *истинна формула* $\mathcal{A}_1$, *то формула* $\mathcal{A}_2$ *также истинна в интерпретации* I.

Доказательство. Теорема непосредственно следует из теорем 1.2.3, 1.2.8 и следствия 1.2.5 (первая часть). $\lhd$

Следствие 1.2.10. *Если (все в тех же обозначениях) формула* $\mathcal{B} \equiv \mathcal{C}$ *общезначима, то и формула* $\mathcal{A}_1 \equiv \mathcal{A}_2$ *общезначима. Если, кроме того, формула* $\mathcal{A}_1$ *общезначима, то и формула* $\mathcal{A}_2$ *общезначима.*

Доказательство очевидно. $\lhd$

Определение 1.2.1. *Если формула* $\mathcal{A}_1 \equiv \mathcal{A}_2$ *общезначима, то формулы* $\mathcal{A}_1$ *и* $\mathcal{A}_2$ *называются логически эквивалентными.*

Теорема 1.2.11 (о цепях эквивалентностей)**.** *Пусть в последовательности формул* $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ *формула* $\mathcal{A}_i$ *для всех* $i > 0$ *получается из* $\mathcal{A}_{i-1}$ *заменой какого-либо вхождения подформулы* $\mathcal{B}_{i-1}$ *формулы* $\mathcal{A}_{i-1}$ *на формулу* $\mathcal{C}_i$, *логически эквивалентную формуле* $\mathcal{B}_{i-1}$. *Тогда формула* $\mathcal{A}_0 \equiv \mathcal{A}_n$ *общезначима.*

Доказательство. Докажем индукцией по i, что формула $\mathcal{A}_0 \equiv \mathcal{A}_i$ общезначима. Для $i = 0$ это утверждение очевидно. Пусть установлено, что формула $\mathcal{A}_0 \equiv \mathcal{A}_{i-1}$ общезначима. По следствию 1.2.10 формула $\mathcal{A}_{i-1} \equiv \mathcal{A}_i$ также общезначима. По свойству транзитивности эквивалентности формула

$$(\mathcal{A}_0 \equiv \mathcal{A}_{i-1}) \supset ((\mathcal{A}_{i-1} \equiv \mathcal{A}_i) \supset (\mathcal{A}_0 \equiv \mathcal{A}_i))$$

общезначима. Применяя дважды следствие 1.2.10, убеждаемся, что формула $\mathcal{A}_0 \equiv \mathcal{A}_i$ общезначима — шаг индукции завершен.

При $i = n$ получаем, что формула $\mathcal{A}_0 \equiv \mathcal{A}_n$ общезначима. $\lhd$

Упражнение 1.2.4. Доказать теорему о цепях импликаций: если последовательность $\mathcal{A}_0, \ldots, \mathcal{A}_n$ такова, что формула $\mathcal{A}_{i-1} \supset \mathcal{A}_i$ общезначима для всех $i > 0$, то формула $\mathcal{A}_0 \supset \mathcal{A}_n$ общезначима.

Последовательности формул, удовлетворяющие условиям теоремы о цепях эквивалентностей или импликаций, обычно пишут в виде $\mathcal{A}_0 \Leftrightarrow \mathcal{A}_1 \Leftrightarrow \ldots \Leftrightarrow \mathcal{A}_n$ или $\mathcal{A}_0 \Rightarrow \mathcal{A}_1 \Rightarrow \ldots \Rightarrow \mathcal{A}_n$. Понятен и смысл записи вида $\mathcal{A}_0 \Leftrightarrow \mathcal{A}_1 \Rightarrow \mathcal{A}_2 \Leftrightarrow \mathcal{A}_3$ (частный случай), когда для некоторых, но не всех i общезначима формула $\mathcal{A}_{i-1} \equiv \mathcal{A}_i$, а для остальных — $\mathcal{A}_{i-1} \supset \mathcal{A}_i$. Ясно, что при этом можно обосновать общезначимость только слабой формулы $\mathcal{A}_0 \supset \mathcal{A}_n$. Если же для последовательности формул $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ общезначимы не только формулы $\mathcal{A}_{i-1} \supset \mathcal{A}_i$ для $i = 1, \ldots, n$, но и формула $\mathcal{A}_n \supset \mathcal{A}_0$, то формула $\mathcal{A}_i \equiv \mathcal{A}_j$ общезначима для любых i и j ($0 \leqslant i, j \leqslant n$). Действительно, пусть $i < j$. Тогда общезначимость формулы $\mathcal{A}_i \supset \mathcal{A}_j$ следует из цепи импликаций $\mathcal{A}_i \Rightarrow \mathcal{A}_{i+1} \Rightarrow \ldots \Rightarrow \mathcal{A}_j$, а общезначимость формулы $\mathcal{A}_j \supset \mathcal{A}_i$ — из цепи $\mathcal{A}_j \Rightarrow \ldots \Rightarrow \mathcal{A}_n \Rightarrow \mathcal{A}_0 \Rightarrow \ldots \Rightarrow \mathcal{A}_i$. Остается использовать общезначимую формулу $(\mathcal{A}_i \supset \mathcal{A}_j) \supset ((\mathcal{A}_j \supset \mathcal{A}_i) \supset (\mathcal{A}_i \equiv \mathcal{A}_j))$ и следствие 1.2.10 (дважды).

Коллекция общезначимых формул, приведенная в разд. 1.2.7, и теоремы о цепях эквивалентностей и импликаций позволяют устанавливать общезначимость многих формул, не рассматривая никаких интерпретаций.

Упражнение 1.2.5. Доказать, что формула $((\mathcal{A} \supset \mathcal{B}) \supset \mathcal{A}) \supset \mathcal{A}$ общезначима.

Конечно, этот метод не универсален. Например, общезначимость формулы (1.2.7–11), занимающей в формальной логике одно из центральных мест, так не устанавливается. Завершим этот раздел доказательством ее общезначимости.

Лемма 1.2.12. *Пусть $u(x)$ — некоторый терм, возможно содержащий вхождения переменной x, $u(t)$ обозначает результат подстановки некоторого другого терма t вместо всех вхождений переменной x в терм $u(x)$. Тогда $V_{s_1}(u(x)) = V_s(u(t))$, где s — произвольный набор, s_1 — набор, получающийся из s заменой значения переменной x значением терма t на наборе s, так что $V_{s_1}(x) = V_s(t)$, $V_{s_1}(y) = V_s(y)$ для любой переменной y, отличной от x.*

Доказательство (индукция по числу n вхождений функциональных букв в терм u). Если $n = 0$, то $u(x)$ — константа или переменная. Возможны следующие случаи:

а. $u(x)$ — константа, тогда $u(t)$ — та же константа, а так как значение константы не зависит от набора, то $V_{s_1}(x) = V_s(t)$;

б. $u(x)$ — переменная y, отличная от x, тогда $u(t)$ — та же переменная y, а так как значения всех переменных, кроме x, на наборах s_1 и s совпадают, то $V_{s_1}(x) = V_s(t)$;

в. $u(x)$ — переменная x, тогда $u(t)$ — это терм t, а при этом имеет место $V_{s_1}(u(x)) = V_{s_1}(x) = V_s(t) = V_s(u(t))$.

Во всех случаях утверждение верно.

Пусть $n > 0$ и утверждение леммы верно для всех термов, содержащих менее n вхождений функциональных букв. При этом условии терм $u(x)$ имеет вид $f_i^k(t_1(x), \ldots, t_k(x))$, и все термы $t_1, \ldots, t_k$, вместе взятые, содержат $n-1$ вхождение функциональных букв. Любое вхождение x в один из термов t_j — это его вхождение также в $u(x)$ и наоборот, так что $u(t)$ имеет вид $f_i^k(t_1(t), \ldots, t_k(t))$. По предположению индукции $V_{s_1}(t_j(x)) = V_s(t_j(t))$ для $1 \leqslant j \leqslant k$. Но тогда

$$\begin{aligned} V_{s_1}(u(x)) &= f_i^{k*}(V_{s_1}(t_1(x)), \ldots, V_{s_1}(t_k(x))) \\ &= f_i^{k*}(V_s(t_1(t)), \ldots, V_s(t_k(t))) = V_s(u(t)). \end{aligned}$$ ◁

Лемма 1.2.13. *Пусть $\mathcal{B}(x)$ — некоторая формула, t — терм, свободный для переменной x в этой формуле, $\mathcal{B}(t)$ — результат подстановки терма t вместо всех свободных вхождений переменной x в формулу $\mathcal{B}(x)$. Тогда*

$$V_{s_1}(\mathcal{B}(x)) = V_s(\mathcal{B}(t)), \tag{17}$$

где s_1 и s — такие же наборы, как в лемме 1.2.12.

Доказательство (индукция по числу m логических и кванторных связок в формуле $\mathcal{B}(x)$). Если $m = 0$ (база индукции), то $\mathcal{B}(x)$ — элементарная функция. Возможны следующие случаи.

а. $\mathcal{B}(x)$ — логическая константа или переменная, тогда $\mathcal{B}(t)$ совпадает с $\mathcal{B}(x)$ и равенство (17) следует из того, что значение логической переменной не зависит от набора, а значение логической константы — даже от интерпретации.

б. $\mathcal{B}(x)$ имеет вид $P_i^k(t_1(x), \ldots, t_k(x))$, тогда $\mathcal{B}(t)$ — это $P_i^k(t_1(t), \ldots, t_k(t))$. По лемме 1.2.12 $V_{s_1}(t_j(x)) = V_s(t_j(t))$ для $j = 1, \ldots, k$, так что

$$V_{s_1}(\mathcal{B}(x)) = P_i^{k*}(V_{s_1}(t_1(x)), \ldots, V_{s_1}(t_k(x))) = P_i^{k*}(V_s(t_1(t)), \ldots, V_s(t_k(t))) = V_s(\mathcal{B}(t)),$$

где P_i^{k*} — функция, сопоставленная предикатной букве P_i^k в рассматриваемой интерпретации.

Других случаев при $m = 0$ быть не может.

Шаг индукции. Пусть $m > 0$ и для всех формул, содержащих меньше m связок, утверждение леммы справедливо. Возможны следующие случаи в зависимости от главной связки формулы $\mathcal{B}(x)$.

а. $\mathcal{B}(x)$ имеет вид $\neg\mathcal{C}(x)$ или $\mathcal{C}(x)\theta\mathcal{D}(x)$, где θ обозначает одну из двухместных логических связок. Каждое свободное вхождение переменной x в формулу $\mathcal{B}(x)$ — это вхождение, также свободное, переменной x в $\mathcal{C}(x)$ или в $\mathcal{D}(x)$ и наоборот. Поэтому $\mathcal{B}(t)$ имеет вид $\neg\mathcal{C}(t)$ или $\mathcal{C}(t)\theta\mathcal{D}(t)$. Формулы $\mathcal{C}(x)$ и $\mathcal{D}(x)$ содержат менее m связок. По предположению индукции $V_{s_1}(\mathcal{C}(x)) = V_s(\mathcal{C}(t))$ и $V_{s_1}(\mathcal{D}(x)) = V_s(\mathcal{D}(t))$. Таким образом, значения $V_{s_1}(\mathcal{B}(x))$ и $V_s(\mathcal{B}(t))$ получаются в результате применения одной и той же операции ($\neg$ или θ) к одинаковым значениям частей формулы. Поэтому они также оказываются одинаковыми.

б. $\mathcal{B}(x)$ имеет вид $\forall x\mathcal{C}(x)$ или $\exists x\mathcal{C}(x)$. При этом все вхождения переменной x в формулу $\mathcal{B}(x)$ — связанные, и $\mathcal{B}(t)$ совпадает с $\mathcal{B}(x)$. Но по теореме 1.2.3 значение формулы $\mathcal{B}(x)$ не зависит от значения переменной x, не свободной в этой формуле. А так как s_1 — это x-вариация набора s, то

$$V_{s_1}(\mathcal{B}(x)) = V_s(\mathcal{B}(x)) = V_s(\mathcal{B}(t)).$$

в. $\mathcal{B}(x)$ имеет вид $Qy\mathcal{C}(x)$, где Q — символ '$\forall$' или '$\exists$', а y — переменная, отличная от x. Все свободные вхождения переменной x в формулу $\mathcal{B}(x)$ являются также свободными вхождениями x в $\mathcal{C}(x)$ и наоборот. Поэтому $\mathcal{B}(t)$ имеет вид $Qy\mathcal{C}(t)$. При этом терм t не может содержать вхождений переменной y, иначе он не был бы свободен для x в формуле $\mathcal{B}(x)$. Значение формулы $\mathcal{B}(x)$ на наборе s_1 определяется совокупностью значений формулы $\mathcal{C}(x)$ на всевозможных y-вариациях s_1' набора s_1. Точно так же значение формулы $\mathcal{B}(t)$ на наборе s определяется совокупностью значений формулы $\mathcal{C}(t)$ на y-вариациях s' набора s. Наборы s_1' и s' можно объединить в пары так, чтобы значения переменной y у них совпадали. Тогда в каждой паре наборы s_1' и s' будут различаться лишь значением переменной x. При этом $V_{s_1'}(x) = V_{s_1}(x) = V_s(t) = V_{s'}(t)$. Последнее равенство в этой цепочке следует из того, что терм t не содержит вхождений переменной y, а набор s' — это y-вариация набора s.

Итак, в любой паре наборы s_1' и s' удовлетворяют условию леммы, а формула $\mathcal{C}(x)$ содержит на одну связку меньше, чем формула $\mathcal{B}(x)$. По предположению индукции $V_{s_1'}(\mathcal{C}(x)) = V_{s'}(\mathcal{C}(t))$. Следовательно значение $V_{s_1}(\mathcal{B}(x))$ определяется такой же совокупностью значений, что и значение $V_s(\mathcal{B}(t))$, таким образом, эти значения совпадают.

Этим завершается обоснование индукционного шага. $\triangleleft$

Теорема 1.2.14. *Если терм t свободен для переменной x в формуле $\mathcal{A}(x)$, то формулы*

$$\forall x\mathcal{A}(x) \supset \mathcal{A}(t), \tag{18}$$

$$\mathcal{A}(t) \supset \exists x\mathcal{A}(x) \tag{19}$$

общезначимы.

Доказательство. Если формула (18) не общезначима, то она не истинна в некоторой интерпретации I, т. е. найдется такой набор s значений переменных, что $V_s(\forall x\mathcal{A}(x) \supset \mathcal{A}(t)) = F$ (все значения берутся в интерпретации I). Это возможно лишь при $V_s(\forall x\mathcal{A}(x)) = T$, $V_s(\mathcal{A}(t)) = F$. Первая из этих формул означает, что $V_{s'}(\mathcal{A}(x)) = T$ для любой x-вариации s' набора s. Возьмем в качестве набора s' такой набор s_1, что $V_{s_1}(x) = V_s(t)$. Наборы s_1 и s удовлетворяют условиям леммы 1.2.13, поэтому $V_{s_1}(\mathcal{A}(x)) = V_s(\mathcal{A}(t))$. Но левая часть этого равенства должна иметь значение T, а правая — значение F, что невозможно.

Аналогично устанавливается общезначимость формулы (19). $\lhd$

Упражнение 1.2.6. Доказать общезначимость формулы (19), исходя из общезначимости формулы (18) методом цепей эквивалентностей.

Упражнение 1.2.7. Доказать общезначимость формулы

$$\forall x\mathcal{A}(x) \wedge \forall y\mathcal{B}(y) \equiv \forall x\forall y(\mathcal{A}(x) \wedge \mathcal{B}(y))$$

при условии, что формула $\mathcal{A}(x)$ не содержит свободных вхождений y, а $\mathcal{B}(y)$ — свободных вхождений x.

1.3. Исчисление высказываний

1.3.1. Аксиоматика

Термином «исчисление высказываний» обозначается класс логических формальных теорий, отличающихся от теорий, рассмотренных выше, в разделах 1.2.1–1.2.8, тем, что их язык не содержит предикатных и функциональных букв, предметных переменных и констант, а также кванторов.

Существуют различные варианты построения исчисления высказываний, которые по форме довольно сильно отличаются друг от друга, но практически совпадают по существу результатов, т. е. по своим основным свойствам. Поэтому целесообразно остановиться на таком варианте исчисления, который был бы прост и компактен как по своей форме, так и по доказательствам утверждений о свойствах этого исчисления.

Рассмотрим вариант (будем называть его теорией $\mathbf{P}$), в котором не используются пропозициональные константы T и F, а из логических связок применяются только $\neg$ и $\supset$. Элементарные формулы теории $\mathbf{P}$ — это пропозициональные буквы. Любая пропозициональная буква является также формулой теории. Если $\mathcal{F}$ и $\mathcal{G}$ — уже имеющиеся в наличии формулы, то $\neg\mathcal{F}$ и $(\mathcal{F}\supset\mathcal{G})$ — также формулы. Формул другого вида в теории $\mathbf{P}$ нет. Самые внешние скобки у импликации будем опускать. Формулы исчисления высказываний иначе называются, как уже упоминалось, пропозициональными формами. Раз в теории $\mathbf{P}$ нет функциональных и предикатных букв, то незачем упоминать о наборах переменных. В этой теории можно и должно говорить о значении формулы лишь в интерпретации, поскольку именно интерпретация определяет значения всех пропозициональных букв, а вместе с ними — и значение любой формулы.

Множество аксиом теории $\mathbf{P}$ задается тремя схемами аксиом:

A1. $\mathcal{A}\supset(\mathcal{B}\supset\mathcal{A})$,
A2. $(\mathcal{A}\supset(\mathcal{B}\supset\mathcal{C}))\supset((\mathcal{A}\supset\mathcal{B})\supset(\mathcal{A}\supset\mathcal{C}))$,
A3. $(\neg\mathcal{A}\supset\neg\mathcal{B})\supset(\mathcal{B}\supset\mathcal{A})$.

Таким образом, аксиомой теории $\mathbf{P}$ является любая формула, получающаяся из одной из схем A1–A3 в результате подстановки каких-либо пропозициональных форм вместо метапеременных $\mathcal{A}$, $\mathcal{B}$ и $\mathcal{C}$. Разумеется, вместо всех вхождений одной и той же метапеременной должна быть подставлена одна и та же форма.

Так, формула $B_1\supset(B_2\supset B_1)$ — это аксиома, построенная по схеме A1, $(\neg(B_3\supset\neg B_5)\supset\neg\neg(B_1\supset B_3))\supset(\neg(B_1\supset B_3)\supset(B_3\supset\neg B_5))$ — аксиома по схеме A3, тогда как выражение $P_1^1(x_1)\supset(P_3^2(a_2,\,x_4)\supset P_1^1(x_1))$ не является не только аксиомой, но даже формулой теории $\mathbf{P}$, так как содержит буквы, не принадлежащие алфавиту этой теории.

В теории $\mathbf{P}$ имеется лишь одно правило вывода, согласно которому формула вида $\mathcal{B}$ является непосредственным следствием формул $\mathcal{A}$ и $\mathcal{A}\supset\mathcal{B}$ (каковы бы ни были формулы $\mathcal{A}$ и $\mathcal{B}$). Коротко это правило записывается в виде схемы

$$\frac{\mathcal{A}\qquad\mathcal{A}\supset\mathcal{B}}{\mathcal{B}}\,.$$

Оно называется правилом *modus ponens* (термин берет свое начало от одного из правил силлогистики Аристотеля) или правилом заключения и сокращенно обозначается MP.

Перефразируя общее определение вывода применительно к теории $\mathbf{P}$, получаем следующий вариант. Выводом формулы $\mathcal{F}$ в исчислении высказываний $\mathbf{P}$ называется такая последовательность $\mathcal{A}_1, \ldots, \mathcal{A}_n$ формул этой теории, что формула $\mathcal{A}_n$ совпадает с $\mathcal{F}$ и при любом i, $1 \leqslant i \leqslant n$, формула $\mathcal{A}_i$ является либо одной из аксиом теории $\mathbf{P}$, либо непосредственным следствием двух других формул вывода $\mathcal{A}_j$ и $\mathcal{A}_k$ $(j, k < i)$ по правилу MP. При этом формула $\mathcal{A}_i$ тогда и только тогда непосредственно следует из формул $\mathcal{A}_j$ и $\mathcal{A}_k$ по правилу MP, когда $\mathcal{A}_k$ имеет вид $\mathcal{A}_j \supset \mathcal{A}_i$. Очень существенно, что все аксиомы теории $\mathbf{P}$ общезначимы, а правило MP обладает свойством, сформулированным в следствии 1.2.5. Подробнее на эту тему — в разд. 1.3.4.

Пример. Пусть $\mathcal{A}$ — произвольная формула теории $\mathbf{P}$. Тогда следующая последовательность из пяти формул представляет собой вывод формулы $\mathcal{A} \supset \mathcal{A}$ в теории $\mathbf{P}$:

1) $\mathcal{A} \supset ((\mathcal{A} \supset \mathcal{A}) \supset \mathcal{A})$, (аксиома по схеме A1)
2) $(\mathcal{A} \supset ((\mathcal{A} \supset \mathcal{A}) \supset \mathcal{A})) \supset ((\mathcal{A} \supset (\mathcal{A} \supset \mathcal{A})) \supset (\mathcal{A} \supset \mathcal{A}))$, (аксиома по схеме A2)
3) $(\mathcal{A} \supset (\mathcal{A} \supset \mathcal{A})) \supset (\mathcal{A} \supset \mathcal{A})$, (следует из 1) и 2) по правилу MP)
4) $\mathcal{A} \supset (\mathcal{A} \supset \mathcal{A})$, (аксиома по схеме A1)
5) $\mathcal{A} \supset \mathcal{A}$. (из 4) и 3) по правилу MP)

Таким образом справедлива следующая

Лемма 1.3.1. $\vdash \mathcal{A} \supset \mathcal{A}$.

Здесь и далее, пока речь будет идти о варианте $\mathbf{P}$ исчисления высказываний, вместо знака $\vdash_{\mathbf{P}}$ применяется знак '$\vdash$', так что метаформула $\vdash \mathcal{F}$ неявно означает: «формула $\mathcal{F}$ выводима в теории $\mathbf{P}$».

1.3.2. Теорема о дедукции

В содержательной математике утверждение «Из $\mathcal{A}$ следует $\mathcal{B}$» обычно доказывается по следующей схеме. Предполагаем, что $\mathcal{A}$ справедливо, посредством некоторой цепочки рассуждений устанавливаем, что и $\mathcal{B}$ должно быть при этом справедливо, и заключаем, что из $\mathcal{A}$ следует $\mathcal{B}$. Кроме того, в рассуждениях может участвовать ряд дополнительных предположений. Формализацией этого способа доказательства является следующая теорема, называемая теоремой о дедукции (для исчисления высказываний).

Теорема 1.3.2. *Пусть* Γ *— некоторый список формул. Тогда если* $\Gamma, \mathcal{A} \vdash \mathcal{B}$, *то* $\Gamma \vdash \mathcal{A} \supset \mathcal{B}$, *или подробнее — если формула* $\mathcal{B}$ *выводима из списка гипотез* Γ, *дополненного формулой* $\mathcal{A}$, *то формула* $\mathcal{A} \supset \mathcal{B}$ *выводима из списка гипотез* Γ.

Доказательство. Пусть $\mathcal{B}_1, \ldots, \mathcal{B}_n$ — вывод формулы $\mathcal{B}$ из списка гипотез Γ, $\mathcal{A}$. Индукцией по номеру i докажем, что формула $\mathcal{A} \supset \mathcal{B}_i$ для $i = 1, \ldots, n$

выводима из списка гипотез Γ. Более точно — если существует некоторый вывод из совокупности гипотез Γ, который уже содержит все формулы вида $\mathcal{A} \supset \mathcal{B}_j$ для $1 \leqslant j < i$, то он может быть продолжен до формулы $\mathcal{A} \supset \mathcal{B}_i$.

Рассмотрим поочередно возможные основания для включения формулы $\mathcal{B}_i$ в исходный вывод $\mathcal{B}_1, \ldots, \mathcal{B}_n$.

а. Формула $\mathcal{B}_i$ — аксиома. Тогда продолжаем вывод, который мы строим, следующими шагами:

$\mathcal{B}_i$ (аксиома)
$\mathcal{B}_i \supset (\mathcal{A} \supset \mathcal{B}_i)$ (аксиома по схеме A1)
$\mathcal{A} \supset \mathcal{B}_i$ (правило MP)

б. Формула $\mathcal{B}_i$ — гипотеза из списка Γ. Очередные шаги нового вывода — те же, что в случае а, но основанием для первого шага является принадлежность $\mathcal{B}_i$ списку Γ.

в. Формула $\mathcal{B}_i$ — гипотеза $\mathcal{A}$. В этом случае формула $\mathcal{A} \supset \mathcal{B}_i$, совпадающая с $\mathcal{A} \supset \mathcal{A}$, может быть включена в вывод на основании леммы 1.3.1. Более точно, в вывод должны быть включены все шаги вывода формулы $\mathcal{A} \supset \mathcal{A}$, существующего согласно этой лемме.

г. Формула $\mathcal{B}_i$ — это непосредственное следствие формул $\mathcal{B}_j$ и $\mathcal{B}_k$ ($j, k < i$) по правилу MP. По предположению индукции формулы $\mathcal{A} \supset \mathcal{B}_j$ и $\mathcal{A} \supset \mathcal{B}_k$, т. е. $\mathcal{A} \supset (\mathcal{B}_j \supset \mathcal{B}_i)$, уже содержатся в новом выводе. Продолжаем его шагами:

$(\mathcal{A} \supset (\mathcal{B}_j \supset \mathcal{B}_i)) \supset ((\mathcal{A} \supset \mathcal{B}_j) \supset (\mathcal{A} \supset \mathcal{B}_i))$ (A2)
$(\mathcal{A} \supset \mathcal{B}_j) \supset (\mathcal{A} \supset \mathcal{B}_i)$ (правило MP)
$\mathcal{A} \supset \mathcal{B}_i$ (правило MP)

Итак, для всех возможных случаев показано, как продолжить вывод до формулы $\mathcal{A} \supset \mathcal{B}_i$. По индукции формула $\mathcal{A} \supset \mathcal{B}_i$ выводима из списка гипотез Γ при любом i, $1 \leqslant i \leqslant n$. В частности, выводима и формула $\mathcal{A} \supset \mathcal{B}_n$, совпадающая с $\mathcal{A} \supset \mathcal{B}$. ◁

Следствие 1.3.3. $\mathcal{A} \supset \mathcal{B}, \mathcal{B} \supset \mathcal{C} \vdash \mathcal{A} \supset \mathcal{C}$.

Доказательство. Выписываем вывод:

1) $\mathcal{A} \supset \mathcal{B}$ (гипотеза)
2) $\mathcal{B} \supset \mathcal{C}$ (гипотеза)
3) $\mathcal{A}$ (гипотеза)
4) $\mathcal{B}$ (3,1, правило MP)
5) $\mathcal{C}$ (4,2, правило MP)

Таким образом, доказано, что $\mathcal{A} \supset \mathcal{B}, \mathcal{B} \supset \mathcal{C}, \mathcal{A} \vdash \mathcal{C}$. По теореме о дедукции, приняв за Γ список $\mathcal{A} \supset \mathcal{B}$ и $\mathcal{B} \supset \mathcal{C}$, получаем требуемый результат. ◁

Следствие 1.3.4. $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C}), \mathcal{B} \vdash \mathcal{A} \supset \mathcal{C}$.

Доказательство. Вывод:

1) $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})$ (гипотеза)
2) $\mathcal{B}$ (гипотеза)

3) $\mathcal{A}$ (гипотеза)
4) $\mathcal{B} \supset \mathcal{C}$ (3, 1, правило MP)
5) $\mathcal{C}$ (2, 4, правило MP)

и теорема о дедукции приводят к цели. $\lhd$

Следствие 1.3.5. $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C}) \vdash \mathcal{B} \supset (\mathcal{A} \supset \mathcal{C})$.

Доказательство. Еще раз применяем теорему о дедукции. $\lhd$

Эти следствия могут рассматриваться как основание для использования в выводах в теории **P** производных правил вывода:

$$\frac{\mathcal{A} \supset \mathcal{B} \qquad \mathcal{B} \supset \mathcal{C}}{\mathcal{A} \supset \mathcal{C}}$$ (правило силлогизма или правило S),

$$\frac{\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C}) \quad \mathcal{B}}{\mathcal{A} \supset \mathcal{C}}$$ (правило I — исключения промежуточной посылки),

$$\frac{\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})}{\mathcal{B} \supset (\mathcal{A} \supset \mathcal{C})}$$ (правило R — перестановки посылок).

Упражнение 1.3.1. Вывести в теории **P** без дополнительных (производных) правил вывода формулу $\mathcal{B} \supset (\mathcal{A} \supset \mathcal{C})$ из формулы $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})$, точнее — из списка гипотез, содержащего одну эту формулу.

Теорема о дедукции и следствия 1.3.3–1.3.5 верны для любой формальной теории, содержащей связку '$\supset$', схемы аксиом A1 и A2, правило MP и не содержащей других правил вывода. Если же в формальной теории есть другие правила вывода, то в вывод формулы $\mathcal{A} \supset \mathcal{B}$ из Γ может быть преобразован такой вывод формулы $\mathcal{B}$ из $\Gamma, \mathcal{A}$, в котором эти правила не применяются. Более сильные утверждения нуждаются в особом доказательстве (например, в доказательстве возможности преобразовать шаг вывода формулы $\mathcal{B}$ из $\Gamma, \mathcal{A}$, на котором использовано правило, отличное от MP, в шаг вывода формулы $\mathcal{A} \supset \mathcal{B}$ из Γ).

1.3.3. Некоторые леммы о выводимости

Леммы этого раздела имеют вспомогательный характер, они предназначены для использования в разд. 1.3.4. В доказательствах показаны некоторые приемы сокращения длины выводов в теории **P**.

Лемма 1.3.6. $\vdash \neg\mathcal{A} \supset (\mathcal{A} \supset \mathcal{B})$.

Доказательство. Строим вывод:

1) $\neg\mathcal{A} \supset (\neg\mathcal{B} \supset \neg\mathcal{A})$ (A1)
2) $(\neg\mathcal{B} \supset \neg\mathcal{A}) \supset (\mathcal{A} \supset \mathcal{B})$ (A3)
3) $\neg\mathcal{A} \supset (\mathcal{A} \supset \mathcal{B})$ (1, 2, правило S)

Выводимость доказана. $\lhd$

Л е м м а 1.3.7. $\vdash \neg\neg\mathcal{A} \supset \mathcal{A}$.

Доказательство.

1) $\neg\neg\mathcal{A}$ (гипотеза)
2) $\neg\neg\mathcal{A} \supset (\neg\mathcal{A} \supset \neg\neg\neg\mathcal{A})$ (лемма 1.3.6)
3) $\neg\mathcal{A} \supset \neg\neg\neg\mathcal{A}$ (1, 2, MP)
4) $(\neg\mathcal{A} \supset \neg\neg\neg\mathcal{A}) \supset (\neg\neg\mathcal{A} \supset \mathcal{A})$ (A3)
5) $\neg\neg\mathcal{A} \supset \mathcal{A}$ (3, 4, MP)
6) $\mathcal{A}$ (1, 5, MP)

По теореме о дедукции (при пустом списке Γ) получаем $\vdash \neg\neg\mathcal{A} \supset \mathcal{A}$. ◁

Упражнение 1.3.2. Формула $\neg\neg\mathcal{A} \supset \mathcal{A}$ возникла уже на пятом шаге вывода. Нельзя ли было на этом остановиться?

Л е м м а 1.3.8. $\vdash \mathcal{A} \supset \neg\neg\mathcal{A}$.

Доказательство.

1) $\neg\neg\neg\mathcal{A} \supset \neg\mathcal{A}$ (лемма 1.3.7)
2) $(\neg\neg\neg\mathcal{A} \supset \neg\mathcal{A}) \supset (\mathcal{A} \supset \neg\neg\mathcal{A})$ (A3)
3) $\mathcal{A} \supset \neg\neg\mathcal{A}$ (1, 2, MP)

Вывод построен. ◁

Л е м м а 1.3.9. $\vdash (\neg\mathcal{A} \supset \mathcal{B}) \supset (\neg\mathcal{B} \supset \mathcal{A})$.

Доказательство.

1) $\neg\mathcal{A} \supset \mathcal{B}$ (гипотеза)
2) $\mathcal{B} \supset \neg\neg\mathcal{B}$ (лемма 1.3.8)
3) $\neg\mathcal{A} \supset \neg\neg\mathcal{B}$ (1, 2, S)
4) $(\neg\mathcal{A} \supset \neg\neg\mathcal{B}) \supset (\neg\mathcal{B} \supset \mathcal{A})$ (A3)
5) $\neg\mathcal{B} \supset \mathcal{A}$ (3, 4, MP)

Применение теоремы о дедукции завершает доказательство. ◁

Аналогично доказываются

Л е м м а 1.3.10. $\vdash (\mathcal{A} \supset \neg\mathcal{B}) \supset (\mathcal{B} \supset \neg\mathcal{A})$.

Л е м м а 1.3.11. $\vdash (\mathcal{A} \supset \mathcal{B}) \supset (\neg\mathcal{B} \supset \neg\mathcal{A})$.

Аксиома A3 и леммы 1.3.9, 1.3.10 и 1.3.11 представляют все четыре вида закона контрапозиции.

Л е м м а 1.3.12. $\vdash \mathcal{A} \supset (\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B}))$.

Доказательство.

1) $\neg\neg(\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{B})$ (лемма 1.3.7)
2) $\mathcal{A} \supset (\neg\neg(\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B})$ (1, R)
3) $(\neg\neg(\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B}) \supset (\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B}))$ (лемма 1.3.9)
4) $\mathcal{A} \supset (\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B}))$ (2, 3, S)

Вывод завершен. ◁

Лемма 1.3.13. $\vdash (\neg\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B})$.

Доказательство.

1) $\mathcal{A} \supset (\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B}))$	(лемма 1.3.12)
2) $\neg\mathcal{B} \supset (\mathcal{A} \supset \neg(\mathcal{A} \supset \mathcal{B}))$	(1, R)
3) $(\neg\mathcal{B} \supset (\mathcal{A} \supset \neg(\mathcal{A} \supset \mathcal{B}))) \supset ((\neg\mathcal{B} \supset \mathcal{A}) \supset (\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B})))$	(A2)
4) $(\neg\mathcal{B} \supset \mathcal{A}) \supset (\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B}))$	(2, 3, MP)
5) $(\neg\mathcal{B} \supset \neg(\mathcal{A} \supset \mathcal{B})) \supset ((\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B})$	(A3)
6) $(\neg\mathcal{B} \supset \mathcal{A}) \supset ((\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B})$	(4, 5, S)
7) $(\neg\mathcal{A} \supset \mathcal{B}) \supset (\neg\mathcal{B} \supset \mathcal{A})$	(лемма 1.3.9)
8) $(\neg\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{A} \supset \mathcal{B}) \supset \mathcal{B})$	(7, 6, S)

Требуемая формула выведена. $\triangleleft$

Упражнение 1.3.3. Доказать, что
а. $\vdash (\neg\mathcal{A} \supset \mathcal{A}) \supset \mathcal{A}$,
б. $\vdash (\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{A} \supset \neg\mathcal{B}) \supset \neg\mathcal{A})$.

1.3.4. Теорема о полноте

Логическая формальная теория называется *полной*, если любая формула этой теории выводима тогда и только тогда, когда она общезначима. Полнота — очень привлекательное качество теории, она гарантирует, что теория содержит все необходимое и ничего лишнего для формального вывода содержательно истинных утверждений. Однако исторический опыт развития математики говорит, что существуют очень трудные теоремы, поиск доказательства которых требует больших творческих усилий. Есть некоторые утверждения, вроде большой (или великой) теоремы Ферма, которая, будучи весьма правдоподобной, до недавнего времени оставалась недоказанной. Это заставляет подозревать, что полнота — не очень распространенное свойство логических формальных теорий. Так оно и есть на самом деле.

Исчисление высказываний является одним из немногих исключений. Это нетрудно предвидеть, так как для формул исчисления высказываний есть методы их проверки на общезначимость, позволяющие провести такую проверку за конечное (хотя, может быть, и большое) число шагов. Хорошо построенная формальная теория (с достаточно богатым запасом аксиом и правил вывода) должна позволять строить формальный вывод формулы всякий раз, когда установлена ее общезначимость. Тот вариант исчисления высказываний, который мы изучаем (теория **P**), этим свойством обладает.

Однако, если сравнить методы проверки общезначимости формул, продемонстрированные в разд. 1.2.7, с примерами доказательств выводимости в теории **P** не очень сложных формул из предыдущего раздела, то бросается в глаза заметно бóльшая прозрачность и эффективность первых. Это свидетельствует, что полнота, даже если она имеет место, вовсе не гарантирует простоту поиска вывода.

Прежде чем доказать теорему о полноте теории **P**, условимся о некоторых обозначениях, которые потребуются как в этом доказательстве, так и при формулировке и доказательстве предшествующей ей леммы.

Пусть $\mathcal{A}$ — некоторая формула исчисления высказываний, I — интерпретация, в которой определены (см. начало разд. 1.3.1) значения всех пропозициональных букв, входящих в формулу $\mathcal{A}$, и следовательно определено значение $V(\mathcal{A})$ этой формулы. Далее индекс I у буквы V будем подразумевать. Обозначим через $\mathcal{A}'$ формулу, совпадающую с формулой $\mathcal{A}$, если $V(\mathcal{A}) = T$, и имеющую вид $\neg\mathcal{A}$, если $V(\mathcal{A}) = F$. Таким образом $V(\mathcal{A}') = T$. Аналогичный смысл имеет обозначение $\mathcal{B}'$ для любой подформулы $\mathcal{B}$ формулы $\mathcal{A}$, вплоть до букв B_i: B_i' совпадает с B_i, если $V(B_i) = T$, и имеет вид $\neg B_i$, если $V(B_i) = F$. Если все пропозициональные буквы, входящие в $\mathcal{A}$, содержатся среди $B_1, \ldots, B_m$, то набор формул $B_1', \ldots, B_m'$ полностью характеризует интерпретацию I в той части, которая связана с вычислением $V(\mathcal{A})$, — зная, что все B_i' принимают значение T в интерпретации I, можно определить значения букв B_i, а по ним и значение $\mathcal{A}$. Оказывается, что приняв список формул $B_1', \ldots, B_m'$ за исходную совокупность гипотез, можно вывести из нее формулу $\mathcal{A}'$ (которая, как было отмечено, также принимает значение T в интерпретации I).

Лемма 1.3.14. *Во введенных обозначениях для произвольной интерпретации* I *имеет место* $B_1', \ldots, B_m' \vdash \mathcal{A}'$.

Доказательство (индукция по числу n логических связок в формуле $\mathcal{A}$).

База индукции ($n = 0$). При этом $\mathcal{A}$ — элементарная формула, т. е. состоит из одной пропозициональной буквы B_i. Значение $\mathcal{A}$ совпадает со значением B_i, поэтому $\mathcal{A}'$ совпадает с B_i'. Вывод формулы $\mathcal{A}'$ из $B_1', \ldots, B_m'$ состоит из одной формулы $\mathcal{A}'$ (она же B_i'), которая входит в список гипотез.

Индукционный шаг ($n > 0$). Пусть для всех формул $\mathcal{B}$, содержащих менее n связок, утверждение леммы справедливо.

Случай 1. Формула $\mathcal{A}$ имеет вид $\neg\mathcal{B}$. Формула $\mathcal{B}$ содержит ровно $n - 1$ связку, и по предположению индукции $B_1', \ldots, B_m' \vdash \mathcal{B}'$. Если $V(\mathcal{B}) = T$, то $V(\mathcal{A}) = F$, формула $\mathcal{B}'$ имеет вид $\mathcal{B}$, формула $\mathcal{A}'$ — вид $\neg\mathcal{A}$, т. е. $\neg\neg\mathcal{B}$. По лемме 1.3.8 выводима формула $\mathcal{B} \supset \neg\neg\mathcal{B}$. Следовательно, имея вывод $\mathcal{B}'$ (т. е. $\mathcal{B}$) из $B_1', \ldots, B_m'$, можно приписать к нему вывод формулы $\mathcal{B} \supset \neg\neg\mathcal{B}$, а затем — по правилу MP — формулу $\neg\neg\mathcal{B}$ (т. е. $\mathcal{A}'$). Если же $V(\mathcal{B}) = F$, то $V(\mathcal{A}) = T$, поэтому $\mathcal{B}'$ имеет вид $\neg\mathcal{B}$, а $\mathcal{A}'$ совпадает с $\mathcal{A}$, т. е. также имеет вид $\neg\mathcal{B}$. Вывод $\mathcal{B}'$ из $B_1', \ldots, B_m'$ одновременно является и выводом $\mathcal{A}'$ из той же совокупности гипотез.

Случай 2. Формула $\mathcal{A}$ имеет вид $\mathcal{B} \supset \mathcal{C}$. Каждая из формул $\mathcal{B}$ и $\mathcal{C}$ содержит менее n связок, так что $B_1', \ldots, B_m' \vdash \mathcal{B}'$ и $B_1', \ldots, B_m' \vdash \mathcal{C}'$. Четыре возможные комбинации значений формул $\mathcal{B}$ и $\mathcal{C}$ укладываются в три подслучая.

2а. Если $V(\mathcal{B}) = F$, то, независимо от $V(\mathcal{C})$, $V(\mathcal{A}) = T$, $\mathcal{B}'$ имеет вид $\neg\mathcal{B}$, а $\mathcal{A}'$ совпадает с $\mathcal{A}$, т. е. с $\mathcal{B} \supset \mathcal{C}$. К выводу $\mathcal{B}'$ из $B_1', \ldots, B_m'$ на основании леммы 1.3.6 можно приписать вывод формулы $\neg\mathcal{B} \supset (\mathcal{B} \supset \mathcal{C})$, а затем (по MP) — формулу $\mathcal{B} \supset \mathcal{C}$.

2б. Если $V(\mathcal{C}) = T$, то снова $V(\mathcal{A}) = T$, $\mathcal{C}'$ совпадает с $\mathcal{C}$, а $\mathcal{A}'$ — с $\mathcal{B} \supset \mathcal{C}$. К выводу $\mathcal{C}'$ из $B'_1, \ldots, B'_m$ приписываем аксиому $\mathcal{C} \supset (\mathcal{B} \supset \mathcal{C})$ (A1) и формулу $\mathcal{B} \supset \mathcal{C}$ (по MP).

2в. Если $V(\mathcal{B}) = T$ и $V(\mathcal{C}) = F$, то $V(\mathcal{A}) = F$, $\mathcal{B}'$ совпадает с $\mathcal{B}$, $\mathcal{C}'$ — с $\neg\mathcal{C}$, а $\mathcal{A}'$ — с $\neg(\mathcal{B} \supset \mathcal{C})$. Выписываем подряд выводы $\mathcal{B}'$ и $\mathcal{C}'$ из $B'_1, \ldots, B'_m$, затем вывод формулы $\mathcal{B} \supset (\neg\mathcal{C} \supset \neg(\mathcal{B} \supset \mathcal{C}))$ (лемма 1.3.12), потом формулы $\neg\mathcal{C} \supset \neg(\mathcal{B} \supset \mathcal{C})$ и $\neg(\mathcal{B} \supset \mathcal{C})$ (оба раза по MP).

Итак, во всех возможных случаях можно построить вывод формулы $\mathcal{A}'$ из совокупности гипотез $B'_1, \ldots, B'_m$. $\lhd$

Теорема 1.3.15 (о полноте исчисления высказываний). *Формула $\mathcal{A}$ теории $\mathbf{P}$ выводима в этой теории тогда и только тогда, когда она общезначима.*

Доказательство. 1 (часть «тогда»). Пусть формула $\mathcal{A}$ общезначима и $B_1, \ldots, B_m$ — список всех входящих в $\mathcal{A}$ пропозициональных букв. Возможны 2^m интерпретаций, сопоставляющих этим буквам различные комбинации значений. Из леммы 1.3.14 следует $B'_1, \ldots, B'_m \vdash \mathcal{A}$, поскольку формула $\mathcal{A}$ истинна в любой из этих интерпретаций, и $\mathcal{A}'$ совпадает с $\mathcal{A}$. Всего может быть выписано 2^m таких утверждений о выводимости. Будем последовательно сокращать список гипотез, применяя следующий прием. Сгруппируем интерпретации попарно, так, чтобы в каждой паре они различались только значением пропозициональной буквы B_m. Соответствующие утверждения о выводимости таковы: $B'_1, \ldots, B'_{m-1}, B_m \vdash \mathcal{A}$ и $B'_1, \ldots, B'_{m-1}, \neg B_m \vdash \mathcal{A}$. Теорема о дедукции позволяет переписать эти утверждения в виде $B'_1, \ldots, B'_{m-1} \vdash B_m \supset \mathcal{A}$ и $B'_1, \ldots, B'_{m-1} \vdash \neg B_m \supset \mathcal{A}$. По лемме 1.3.13 выводима формула $(\neg B_m \supset \mathcal{A}) \supset ((B_m \supset \mathcal{A}) \supset \mathcal{A})$. Объединяя все три вывода в одну последовательность и приписывая к ней (на основании правила MP) формулы $(B_m \supset \mathcal{A}) \supset \mathcal{A}$ и $\mathcal{A}$, получим вывод формулы $\mathcal{A}$ из совокупности гипотез $B'_1, \ldots, B'_{m-1}$. Итак, $B'_1, \ldots, B'_{m-1} \vdash \mathcal{A}$.

Повторяя этот прием еще $m-1$ раз, установим, что формула $\mathcal{A}$ выводима в исчислении высказываний $\mathbf{P}$ без привлечения каких бы то ни было гипотез.

2 (часть «только тогда»). Пусть формула $\mathcal{A}$ выводима, $\mathcal{B}_1, \ldots, \mathcal{B}_n$ — вывод $\mathcal{A}$ в теории $\mathbf{P}$. Докажем по индукции, что каждая из формул $\mathcal{B}_i$, $i = 1, \ldots, n$, общезначима. Пусть это уже доказано для всех формул $\mathcal{B}_j$ при $j < i$.

Все схемы аксиом теории $\mathbf{P}$ встречались в разд. 1.2.7 среди схем формул, общезначимость которых была обоснована в разд. 1.2.6 или легко может быть обоснована изложенными там методами. Поэтому, если $\mathcal{B}_i$ — аксиома, то она общезначима.

Если $\mathcal{B}_i$ непосредственно следует по правилу MP из формул $\mathcal{B}_j$ и $\mathcal{B}_k$, где $j, k < i$, то формулы $\mathcal{B}_j$ и $\mathcal{B}_k$, имеющая вид $\mathcal{B}_j \supset \mathcal{B}_i$, общезначимы по предположению индукции. Тогда по следствию 1.3.3 и формула $\mathcal{B}_i$ общезначима.

По индукции это верно для любого i, в частности и при $i = n$. Но $\mathcal{B}_n$ — это формула $\mathcal{A}$, следовательно формула $\mathcal{B}_n$ общезначима. $\lhd$

Логическая формальная теория называется *непротиворечивой*, если не существует такой формулы $\mathcal{A}$, что в этой теории $\vdash \mathcal{A}$ и $\vdash \neg\mathcal{A}$. Следствием теоремы о полноте исчисления высказываний является

Теорема 1.3.16. *Исчисление высказываний* $\mathbf{P}$ *непротиворечиво.*

Доказательство. Если бы нашлась такая формула $\mathcal{A}$ теории $\mathbf{P}$, что в этой теории $\vdash \mathcal{A}$ и $\vdash \neg\mathcal{A}$, то по теореме о полноте (2-я часть) как формула $\mathcal{A}$, так и формула $\neg\mathcal{A}$ были бы общезначимы, т. е. истинны в любой интерпретации, что невозможно. Это и значит, что исчисление высказываний непротиворечиво. $\lhd$

1.3.5. Другие аксиоматики исчисления высказываний

Построенная выше теория $\mathbf{P}$ — исчисление высказываний — использует лишь две логические связки: '$\neg$' и '$\supset$'. В разд. 1.2.7 говорилось о возможности выразить одни логические связки через другие. Это служит основанием для следующего определения.

Определение 1.3.1. *Формула* $\mathcal{B}$, *содержащая пропозициональные буквы, скобки и логические связки '*$\neg$*', '*$\supset$*', '*$\wedge$*', '*$\vee$*' и '*$\equiv$*', называется сокращением логической формулы* $\mathcal{A}$, *содержащей те же символы, кроме связок '*$\wedge$*', '*$\vee$*' и '*$\equiv$*', если* $\mathcal{A}$ *может быть получена из* $\mathcal{B}$ *в результате конечного числа подстановок формул:*

$\neg(\mathcal{C} \supset \neg\mathcal{D})$ *вместо* $\mathcal{C} \wedge \mathcal{D}$,
$\neg\mathcal{C} \supset \mathcal{D}$ *вместо* $\mathcal{C} \vee \mathcal{D}$,
$(\mathcal{C} \supset \mathcal{D}) \wedge (\mathcal{D} \supset \mathcal{C})$ *вместо* $\mathcal{C} \equiv \mathcal{D}$,

где $\mathcal{C}$ *и* $\mathcal{D}$ *— произвольные формулы.*

Сокращенные формулы не принадлежат формальной теории $\mathbf{P}$, но принадлежат языку, пополненному связками '$\wedge$', '$\vee$' и '$\equiv$' и соответствующими правилами образования формул. Между общезначимостью сокращенных формул и выводимостью в теории $\mathbf{P}$ их прототипов существует простая связь.

Теорема 1.3.17. *Формула* $\mathcal{B}$, *являющаяся сокращением формулы* $\mathcal{A}$, *не содержащей связок '*$\wedge$*', '*$\vee$*' и '*$\equiv$*', общезначима тогда и только тогда, когда* $\mathcal{A}$ *выводима в теории* $\mathbf{P}$.

Доказательство. $\mathcal{B}$ общезначима $\Leftrightarrow$ $\mathcal{A}$ общезначима $\Leftrightarrow$ $\mathcal{A}$ выводима. Первая эквивалентность следует из теоремы об эквивалентной замене, вторая — из теоремы о полноте исчисления высказываний. $\lhd$

Естественно, что существует и другой, более прямой, способ введения связок, отличных от $\supset$ и $\neg$, в исчисление высказываний. Достаточно, например, изменить аксиоматику так, чтобы новые аксиомы описывали все характерные свойства новых связок. Часто рассматривается, например, формальная теория $\mathbf{P}_1$ со связками '$\supset$', '$\wedge$', '$\vee$' и '$\neg$', с единственным правилом вывода MP и со следующими схемами аксиом:

A1$_1$. $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{A})$,
A2$_1$. $(\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})) \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C}))$,
A3$_1$. $\mathcal{A} \wedge \mathcal{B} \supset \mathcal{A}$,
A4$_1$. $\mathcal{A} \wedge \mathcal{B} \supset \mathcal{B}$,

$\mathrm{A5}_1$. $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{A} \wedge \mathcal{B})$,
$\mathrm{A6}_1$. $\mathcal{A} \supset \mathcal{A} \vee \mathcal{B}$,
$\mathrm{A7}_1$. $\mathcal{B} \supset \mathcal{A} \vee \mathcal{B}$,
$\mathrm{A8}_1$. $(\mathcal{A} \supset \mathcal{C}) \supset ((\mathcal{B} \supset \mathcal{C}) \supset (\mathcal{A} \vee \mathcal{B} \supset \mathcal{C}))$,
$\mathrm{A9}_1$. $(\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{A} \supset \neg\mathcal{B}) \supset \neg\mathcal{A})$,
$\mathrm{A10}_1$. $\neg\neg\mathcal{A} \supset \mathcal{A}$.

Первые две схемы аксиом — те же, что в теории $\mathbf{P}$ — определяют главнейшие свойства связки '$\supset$' и позволяют, в частности, доказать в теории $\mathbf{P}_1$ лемму 1.3.1 и теорему 1.3.2 о дедукции. Схемы аксиом $\mathrm{A3}_1$–$\mathrm{A5}_1$ описывают свойства связки '$\wedge$', схемы $\mathrm{A6}_1$–$\mathrm{A8}_1$ — связки '$\vee$', а схемы $\mathrm{A9}_1$ и $\mathrm{A10}_1$ — связки '$\neg$'. Все эти схемы встречались в разд. 1.2.7 в качестве примеров схем общезначимых формул.

Для теории $\mathbf{P}_1$ верна теорема 1.3.15 о полноте. Однако из-за бóльшего числа связок и аксиом доказательство этой теоремы и основной леммы к ней требует рассмотрения бóльшего числа случаев и использования бóльшего количества вспомогательных лемм.

Упражнение 1.3.4. Доказать, что $\vdash_{\mathbf{P}_1} (\neg\mathcal{B} \supset \neg\mathcal{A}) \supset (\mathcal{A} \supset \mathcal{B})$.

Упражнение 1.3.5. Пусть теория $\mathbf{P}_2$ отличается от $\mathbf{P}_1$ лишь тем, что схемы аксиом $\mathrm{A9}_1$ и $\mathrm{A10}_1$ заменены схемой $\mathrm{A9}_2$: $(\neg\mathcal{B} \supset \neg\mathcal{A}) \supset (\mathcal{A} \supset \mathcal{B})$. Доказать, не опираясь на теорему о полноте, что множества теорем теорий $\mathbf{P}_1$ и $\mathbf{P}_2$ совпадают.

1.4. Формальные теории первого порядка

Следующий обширный класс формальных теорий, которые мы рассмотрим, — это так называемые *формальные теории первого порядка* (ФТПП). В отличие от исчисления высказываний ФТПП — это, как правило, не чисто логические теории, почему — будет сказано несколько позже. Теории порядка выше первого допускают переменные, значениями которых служат функции и (или) предикаты, а главное — кванторы по таким переменным. Подобные теории нам в этой книге не встретятся.

1.4.1. Аксиоматика

Мы будем рассматривать лишь такие формальные теории первого порядка, в которых из всех символов, перечисленных в разд. 1.2.1, используются две логические связки: '$\neg$' и '$\supset$' и один кванторный символ '$\forall$'. Это позволит, как и для исчисления высказываний, сократить некоторые определения и доказательства, ничего не теряя по существу. В алфавите ФТПП нет пропозициональных букв и пропозициональных констант. Элементарная формула ФТПП может иметь только вид $P_i^k(t_1, \ldots, t_k)$, где $t_1, \ldots, t_k$ — термы. Термы строятся обычным образом, но если множество функциональных букв теории пусто, то термом может быть только предметная переменная или константа, а если пусто и множество констант, то только предметная переменная. Множество предметных переменных x_i предполагается счетным, что позволяет не ограничиваться в числе возможных термов.

Множество предикатных букв P_i^k должно быть непусто, так как в противном случае не из чего было бы строить элементарные формулы. Неэлементарные формулы могут иметь вид $\neg\mathcal{A}$, $(\mathcal{A} \supset \mathcal{B})$ или $\forall x_i\mathcal{A}$. Иногда будут встречаться формулы, содержащие другие связки. Но при этом формулы, содержащие логические связки, отличные от '$\neg$' и '$\supset$', будем рассматривать как сокращения формул, не содержащих этих связок, по тем же правилам (см. разд. 1.3.5), что и для исчисления высказываний. Формулу вида $\exists x_i\mathcal{A}$ будем считать сокращенной записью эквивалентной ей формулы $\neg\forall x_i\neg\mathcal{A}$.

Множество аксиом любой ФТПП делится на два класса: *логических* и *собственных* аксиом. Первые строятся по одним и тем же для всех ФТПП схемам:

A1. $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{A})$,
A2. $(\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})) \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C}))$,
A3. $(\neg\mathcal{A} \supset \neg\mathcal{B}) \supset (\mathcal{B} \supset \mathcal{A})$,
A4. $\forall\mathcal{A}(x) \supset \mathcal{A}(t)$,
A5. $\forall x\,(\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \forall x\mathcal{B})$.

В схеме A4 терм t должен быть свободен для переменной x в формуле $\mathcal{A}(x)$ и, как всегда, $\mathcal{A}(t)$ обозначает результат подстановки терма t вместо каждого из свободных вхождений переменной x в формулу $\mathcal{A}(x)$. В схеме A5 формула $\mathcal{A}$ не должна содержать свободных вхождений переменной x.

Совпадение схем логических аксиом, естественно, не означает, что множество самих аксиом для всех ФТПП одно и то же. Эти множества различаются (точнее, могут различаться) из-за несовпадения наборов предметных констант, функциональных и предикатных букв, входящих в запись формул (а значит, и аксиом).

Множество собственных аксиом — свое для каждой ФТПП. Содержательно, собственные аксиомы описывают свойства объектов, функций и отношений (предикатов) некоторой математической (или мягче — математизированной) теории, представленных предметными константами, функциональными и предикатными буквами формальной теории. Именно в собственных аксиомах сосредоточена, таким образом, индивидуальность каждой ФТПП.

Пример 1. Язык формальной арифметики (в одном из вариантов) содержит одну предметную константу a_1 (обозначающую число нуль), одну одноместную функциональную букву f_1^1 ($f_1^1(x)$ обозначает натуральное число, следующее за x), две двухместные функциональные буквы f_1^2 и f_2^2, обозначающие операции сложения и умножения, и одну предикатную букву P_1^2, обозначающую отношение равенства. Для наглядности будем, как это обычно делается, писать: 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 $\neg\,(t=u)$.

Множество собственных аксиом формальной арифметики состоит из восьми явно заданных аксиом:

S1. $x_1=x_2\supset(x_1=x_3\supset x_2=x_3)$,
S2. $x_1=x_2\supset x_1'=x_2'$,
S3. $0\neq x_1'$,
S4. $x_1'=x_2'\supset x_1=x_2$,
S5. $x_1+0=x_1$,
S6. $x_1+x_2'=(x_1+x_2)'$,
S7. $x_1\times 0=0$,
S8. $x_1\times x_2'=(x_1\times x_2)+x_1$,

и одной схемы аксиом (аксиом индукции):

S9. $\mathcal{A}(0)\supset(\forall x\,(\mathcal{A}(x)\supset\mathcal{A}(x'))\supset\forall x\mathcal{A}(x))$,

где $\mathcal{A}(x)$ — произвольная формула.

Пример 2. Приведем, весьма схематически, первые шаги построения формальной теории множеств. Объекты, о которых пойдет речь в этой теории, будут только множествами.

Во многих разделах математики стало привычным говорить о таких множествах, как множество натуральных или вещественных чисел, множество точек на прямой или на плоскости. Но на их элементы (числа или точки) обычно не смотрят как на множества. Тем не менее все подобные объекты могут быть определены в рамках теории множеств, и в разд. 1.5 кое-что в этом направлении будет сделано. Поэтому, когда в примерах упоминаются некие абстрактные объекты, обозначенные буквами a, b, c, ..., имеется в виду, что эти обозначения заменяют собой предметные константы для множеств. Другие же буквы, например, x, y, u, v, A, B, M, X, и т. д. будут использоваться в роли предметных пере-

менных нашей формальной теории и обозначать, следовательно, произвольные множества. Те или иные обозначения выбираются в чисто мнемонических целях, формально же все эти обозначения совершенно равноправны и служат синонимами обозначений x_1, x_2, ... из разд. 1.2.1.

Единственное первичное отношение P_1^2 между объектами — это отношение принадлежности. Будем писать $x \in A$ вместо $P_1^2(x, A)$. Такая запись читается «x принадлежит множеству A» или «x является элементом множества A» или «множество A содержит элемент x». При этом, повторяем, сам объект x — это тоже некоторое множество.

Будем пользоваться обозначениями типа $\{a, b, c\}$ для множества, содержащего объекты a, b, c и только их, или $\{1, 3, 5, \ldots\}$ для множества всех нечетных натуральных чисел (ответа на вопрос — а что это за множества: 1, 3, 5, и что означает здесь многоточие? — мы не даем, но намеком ответим ниже, в разд. 1.5). Нужно всегда помнить, что a и $\{a\}$ — это совершенно разные множества. Об элементах множества a мы ничего не знаем, а про $\{a\}$ можем твердо сказать, что это множество, единственным элементом которого является a.

В терминах отношения принадлежности можно выразить другие отношения между множествами. Вводя новое двухместное отношение (и новый символ R, его обозначающий), обычно будем выписывать формулу $\mathcal{F}$ со свободными переменными x и y, истинную тогда и только тогда, когда объекты x и y находятся между собой в отношении R. Таким образом, формула $\mathcal{F}$ представляет собой расшифровку вновь вводимой краткой записи xRy. Соответственно будем писать «$xRy \equiv \mathcal{F}$», что достаточно хорошо отражает суть дела.

Так, отношение включения между двумя множествами, обозначаемое $\subseteq$, имеет определение

$$x \subseteq y \equiv \forall z(z \in x \supset z \in y),$$

т. е. отношение $x \subseteq y$ выполнено в том и только в том случае, если любой элемент z множества x принадлежит также множеству y.

Отношение равенства между множествами характеризуется свойствами:

$$x = y \equiv \forall z(z \in x \equiv z \in y), \tag{1}$$

$$x = y \supset \forall z(x \in z \equiv y \in z). \tag{2}$$

Формула (1) принимается за определение равенства. Она утверждает, что множества равны в том и только в том случае, когда каждый элемент одного из них принадлежит и другому. Вторая формула (свойство *объемности* или *экстенсиональности*) говорит, что если одно из двух равных множеств принадлежит некоторому множеству z, то и второе является элементом z. Эти два утверждения не вытекают одно из другого. Формула (2) принимается за аксиому, постулирующую дополнительное свойство равенства. В определении (1) говорится об элементах z как бы уже имеющихся множеств x и y. В формуле (2) на произвольные множества x и y налагается требование одновременно быть или не быть элементами любого множества z, которое когда-либо будет определено. Это разъяснение довольно туманно, тем более, что в рассматриваемой логике категория времени, напомним, отсутствует. Но вспомним также, что попытки докопаться до

сути математических определений всегда наталкиваются на едва ли преодолимые трудности. Конец примера.

В любой ФТПП имеются два правила вывода: уже знакомое правило *modus ponens* (MP)

$$\frac{\mathcal{A} \qquad \mathcal{A} \supset \mathcal{B}}{\mathcal{B}}$$

и *правило обобщения* (Gen)

$$\frac{\mathcal{A}}{\forall x \mathcal{A}},$$

говорящее, что непосредственным следствием произвольной формулы $\mathcal{A}$ является формула $\forall x \mathcal{A}$, получаемая из $\mathcal{A}$ навешиванием квантора всеобщности по произвольной переменной x. Если формула $\forall x \mathcal{A}$ включена в некоторый вывод на основании правила Gen (для этого ранее в выводе должна содержаться формула $\mathcal{A}$), то говорят, что в этом выводе использовано правило обобщения по переменной x.

Общее определение вывода переформулируется для ФТПП естественным образом. Выводом формулы $\mathcal{A}$ в теории первого порядка $\mathbf{Q}$ называется последовательность формул этой теории $\mathcal{B}_1, \ldots, \mathcal{B}_n$, где $\mathcal{B}_n$ совпадает с $\mathcal{A}$, и каждая формула $\mathcal{B}_i$ для $i = 1, \ldots, n$ либо является аксиомой (логической или собственной), либо непосредственно следует из формул $\mathcal{B}_j$ и $\mathcal{B}_k$ ($j, k < i$) по правилу MP, или же из формулы $\mathcal{B}_j$ ($j < i$) по правилу Gen. Аналогично определяется вывод из списка гипотез Γ (добавляется, что $\mathcal{B}_i$ может быть одной из гипотез).

Если существует вывод формулы $\mathcal{A}$ в ФТПП $\mathbf{Q}$, то $\mathcal{A}$ называется выводимой формулой или теоремой теории $\mathbf{Q}$ и, как обычно, это коротко записывается в виде $\vdash_{\mathbf{Q}} \mathcal{A}$ или $\vdash \mathcal{A}$, если ясно, какая теория $\mathbf{Q}$ имеется в виду. Аналогичный смысл имеет запись $\Gamma \vdash_{\mathbf{Q}} \mathcal{A}$.

Формальная теория первого порядка без собственных аксиом называется *исчислением предикатов первого порядка* (ИППП). Каждой ФТПП соответствует некоторое ИППП — ее логическое ядро — получающееся, если из множества аксиом этой теории удалить все собственные аксиомы. Конечно, такая операция выхолащивает содержание теории, так как теряется информация о свойствах ее объектов, операций и отношений. Но если множество собственных аксиом теории $\mathbf{Q}$ может быть заменено конечным списком формул Γ, из которого в соответствующем ИППП $\mathbf{Q}'$ выводимы все собственные аксиомы, (в частности, если это множество само конечно), то каждое утверждение вида $\vdash_{\mathbf{Q}} \mathcal{A}$ может быть заменено утверждением $\Gamma \vdash_{\mathbf{Q}'} \mathcal{A}$, так как вывод формулы $\mathcal{A}$ в теории $\mathbf{Q}$ преобразуется в вывод той же формулы из списка Γ в $\mathbf{Q}'$, если в его начало вставить вывод всех необходимых собственных аксиом из Γ.

Пусть $\mathcal{B}$ — пропозициональная форма, содержащая только пропозициональные буквы $B_1, \ldots, B_s$. Если каждое вхождение буквы B_i в форму $\mathcal{B}$ заменить одной и той же формулой $\mathcal{A}_i$ формальной теории первого порядка $\mathbf{Q}$ для всех $i = 1, \ldots, s$ (для $j \neq i$ формулы $\mathcal{A}_j$ и $\mathcal{A}_i$ могут, конечно, различаться), то получится некоторая формула $\mathcal{A}$ теории $\mathbf{Q}$. Это легко доказывается индукцией

по числу логических связок в форме $\mathcal{B}$. Полученная формула $\mathcal{A}$ называется *частным случаем* пропозициональной формы $\mathcal{B}$. Переход от $\mathcal{B}$ к $\mathcal{A}$ естествен, так как буквы B_i содержательно обозначают некоторые высказывания, а формулы $\mathcal{A}_i$ представляют собой запись каких-то высказываний средствами теории $\mathbf{Q}$.

Частный случай некоторой общезначимой пропозициональной формы называется также *частным случаем тавтологии* или просто *тавтологией*.

Между выводимостью в исчислении высказываний и в ФТПП существует связь (односторонняя), выраженная следующей теоремой.

Теорема 1.4.1. *Любой частный случай тавтологии выводи́м в теории первого порядка.*

Доказательство. Пусть формула $\mathcal{A}$ теории $\mathbf{Q}$ — частный случай общезначимой пропозициональной формы $\mathcal{B}$. По теореме 1.3.15 о полноте исчисления высказываний форма $\mathcal{B}$ выводима. Пусть $\mathcal{B}_1, \ldots, \mathcal{B}_m$ — вывод $\mathcal{B}$ в исчислении высказываний. Согласно определению, $\mathcal{A}$ получается из $\mathcal{B}$ подстановкой некоторых формул $\mathcal{A}_1, \ldots, \mathcal{A}_s$ теории $\mathbf{Q}$ вместо пропозициональных букв $B_1, \ldots, B_s$, входящих в $\mathcal{B}$. Сделаем такую же подстановку в каждой из формул вывода. Если эти формулы помимо букв $B_1, \ldots, B_s$ содержат другие пропозициональные буквы, то вместо каждой из них подставим всюду одну и ту же (для этой буквы) произвольную формулу теории $\mathbf{Q}$. Последовательность формул $\mathcal{B}_1, \ldots, \mathcal{B}_m$ преобразуется в последовательность формул $\mathcal{C}_1, \ldots, \mathcal{C}_m$ теории $\mathbf{Q}$. Поскольку $\mathcal{B}_m$ совпадает с $\mathcal{B}$, то формула $\mathcal{C}_m$, в которую она переходит после описанных подстановок, совпадает с $\mathcal{A}$ ($\mathcal{B}$ не может содержать других пропозициональных букв кроме $B_1, \ldots, B_s$). Если $\mathcal{B}_i$, $i = 1, \ldots, m$, — аксиома, то она построена по одной из схем A1, A2 или A3 и после подстановки преобразуется в аксиому $\mathcal{C}_i$ теории $\mathbf{Q}$, построенную по той же схеме. Если $\mathcal{B}_i$ непосредственно следует из $\mathcal{B}_j$ и $\mathcal{B}_k$ по правилу MP, то $\mathcal{B}_k$ имеет вид $\mathcal{B}_j \supset \mathcal{B}_i$ и после подстановок $\mathcal{B}_i$, $\mathcal{B}_j$ и $\mathcal{B}_k$ переходят в $\mathcal{C}_i$, $\mathcal{C}_j$ и $\mathcal{C}_k$ (последняя — вида $\mathcal{C}_j \supset \mathcal{C}_i$), так что $\mathcal{C}_i$ следует из $\mathcal{C}_j$ и $\mathcal{C}_k$ по правилу MP. Все это означает, что последовательность $\mathcal{C}_1, \ldots, \mathcal{C}_m$ — вывод формулы $\mathcal{A}$ в теории $\mathbf{Q}$. $\lhd$

Эта теорема оправдывает предпринятое в разд. 1.3 изучение сравнительно слабой логической теории — исчисления высказываний.

Мы доказали выводимость в ФТПП довольно обширного класса формул этих теорий. Но, разумеется, этот класс далеко не охватывает совокупность всех выводимых в этих теориях формул. Расширить его может помочь, прежде всего, теорема о дедукции.

1.4.2. Теорема о дедукции

Эту теорему нельзя перенести на формальные теории первого порядка в том же виде, в каком она была доказана для исчисления высказываний. Вот простейший противоречащий пример. Рассмотрим совокупность гипотез, состоящую из одной произвольной формулы $\mathcal{A}$. Можно построить вывод:

1) $\mathcal{A}$ (гипотеза)
2) $\forall x_i \mathcal{A}$ $(1, \mathrm{Gen})$

Таким образом, имеет место выводимость $\mathcal{A} \vdash \forall x_i \mathcal{A}$. С другой стороны, если в некоторой формальной теории существует формула $\mathcal{A}$, в которую переменная x_i входит свободно, и эта теория допускает модель, в которой найдутся такие набор s и его x_i-вариация s', что: $V_s(\mathcal{A}) = T$, $V_{s'}(\mathcal{A}) = F$, то $V_s(\forall x_i \mathcal{A}) = F$, $V_s(\mathcal{A} \supset \forall x_i \mathcal{A}) = F$. Это означает, что формула $\mathcal{A} \supset \forall x_i \mathcal{A}$ не истинна в такой модели. В начале разд. 1.4.3 (теорема 1.4.10) будет показано, что это невозможно, если формула $\mathcal{A}$ выводима в этой теории.

Существуют, однако, формулировки, в которых теорема о дедукции может быть доказана и для ФТПП. Вот одна из них, хотя и не самая общая.

Теорема 1.4.2. *Если в формальной теории первого порядка* **Q** *существует вывод формулы* $\mathcal{B}$ *из совокупности гипотез* $\Gamma, \mathcal{A}$, *в котором правило обобщения* Gen *не применяется ни по какой из переменных, свободных в формуле* $\mathcal{A}$, *то в теории* **Q** *формула* $\mathcal{A} \supset \mathcal{B}$ *выводима из совокупности гипотез* Γ. *При этом существует вывод формулы* $\mathcal{A} \supset \mathcal{B}$ *из* Γ, *в котором правило* Gen *применяется только по тем же переменным, по которым оно применялось в исходном выводе формулы* $\mathcal{B}$ *из* $\Gamma, \mathcal{A}$.

Доказательство. Пусть

$$\mathcal{B}_1, \ldots, \mathcal{B}_n \tag{3}$$

— вывод формулы $\mathcal{B}$ из совокупности гипотез $\Gamma, \mathcal{A}$, существование которого оговорено в условии теоремы. Покажем, что для каждого i, $i = 1, \ldots, n$, может быть построен вывод формулы $\mathcal{A} \supset \mathcal{B}_i$ из Γ, удовлетворяющий требованиям теоремы и содержащий формулы $\mathcal{A} \supset \mathcal{B}_j$ для любого $j < i$. Доказательство будем вести индукцией по i. Переходя к очередному значению i, будем предполагать, что для $i - 1$ такой вывод уже построен. В частности, для начального значения $i = 1$ никаких предварительных условий не требуется. Рассмотрим различные возможные случаи появления формулы $\mathcal{B}_i$ в выводе (3).

Если формула $\mathcal{B}_i$ — это аксиома, или одна из гипотез (из списка Γ или же гипотеза $\mathcal{A}$), или получена в выводе (3) как следствие предшествующих ей формул по правилу MP, то продолжаем вновь строящийся вывод до появления в нем формулы $\mathcal{A} \supset \mathcal{B}_i$ точно так же, как это делалось в исчислении высказываний при доказательстве теоремы 1.3.2. Правило Gen при этом не применяется.

Если же формула $\mathcal{B}_i$ возникла в выводе (3) как непосредственное следствие формулы $\mathcal{B}_j$, где $j < i$, по правилу Gen, то формула $\mathcal{B}_i$ должна иметь вид $\forall x_k \mathcal{B}_j$, и по условию теоремы переменная x_k, по которой было применено правило Gen, не должна входить в формулу $\mathcal{A}$ свободно. Добавляем к строящемуся выводу, который по индуктивному предположению содержит формулу $\mathcal{A} \supset \mathcal{B}_j$, следующие шаги:

$\forall x_k (\mathcal{A} \supset \mathcal{B}_j)$ (Gen)
$\forall x_k (\mathcal{A} \supset \mathcal{B}_j) \supset (\mathcal{A} \supset \forall x_k \mathcal{B}_j)$ (A5)
$\mathcal{A} \supset \forall x_k \mathcal{B}_j$, т. е. $\mathcal{A} \supset \mathcal{B}_i$ (MP)

На первом из них правило Gen применено по той же переменной x_k, по которой оно применялось в выводе (3). Формула, включенная в новый вывод на среднем

шаге, действительно является аксиомой по схеме A5, так как переменная x_k не входит в $\mathcal{A}$ свободно.

Во всех случаях вновь построенная последовательность формул завершается формулой $\mathcal{A} \supset \mathcal{B}_i$ и представляет собой вывод этой формулы из совокупности гипотез Γ, удовлетворяющий условиям теоремы. Полагая $i = n$, получаем то, что требовалось доказать. $\lhd$

Пример 1. Докажем следующее утверждение: $\vdash \forall x_1 \forall x_2 \mathcal{F} \supset \forall x_2 \forall x_1 \mathcal{F}$. Здесь, как и всюду в дальнейшем, если не оговорено что-либо иное, имеется в виду выводимость в любом исчислении предикатов первого порядка и, следовательно, в любой формальной теории первого порядка. Собираясь воспользоваться теоремой о дедукции, примем формулу $\forall x_1 \forall x_2 \mathcal{F}$ за гипотезу (единственную).

1) $\forall x_1 \forall x_2 \mathcal{F}$ (гипотеза)
2) $\forall x_1 \forall x_2 \mathcal{F} \supset \forall x_2 \mathcal{F}$ (A4)
3) $\forall x_2 \mathcal{F}$ (1, 2, MP)
4) $\forall x_2 \mathcal{F} \supset \mathcal{F}$ (A4)
5) $\mathcal{F}$ (3, 4, MP)
6) $\forall x_1 \mathcal{F}$ (5, Gen)
7) $\forall x_2 \forall x_1 \mathcal{F}$ (6, Gen)

Этот вывод показывает, что $\forall x_1 \forall x_2 \mathcal{F} \vdash \forall x_2 \forall x_1 \mathcal{F}$. Правило обобщения применялось дважды — по переменным x_1 и x_2, связанным в формуле $\forall x_1 \forall x_2 \mathcal{F}$. Поэтому можно воспользоваться теоремой о дедукции (взяв в качестве $\mathcal{A}$, $\mathcal{B}$ и Γ соответственно формулы $\forall x_1 \forall x_2 \mathcal{F}$, $\forall x_2 \forall x_1 \mathcal{F}$ и пустое множество гипотез) и заключить, что формула $\forall x_1 \forall x_2 \mathcal{F} \supset \forall x_2 \forall x_1 \mathcal{F}$ выводима.

Пример 2. $\vdash \forall x(\mathcal{A} \supset \mathcal{B}) \supset (\forall x \mathcal{A} \supset \forall x \mathcal{B})$.

1) $\forall x(\mathcal{A} \supset \mathcal{B})$ (гипотеза)
2) $\forall x \mathcal{A}$ (гипотеза)
3) $\forall x \mathcal{A} \supset \mathcal{A}$ (A4)
4) $\mathcal{A}$ (2, 3, MP)
5) $\forall x(\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{B})$ (A4)
6) $\mathcal{A} \supset \mathcal{B}$ (1, 5, MP)
7) $\mathcal{B}$ (4, 6, MP)
8) $\forall x \mathcal{B}$ (7, Gen)

Таким образом, $\forall x(\mathcal{A} \supset \mathcal{B}), \forall x \mathcal{A} \vdash \forall x \mathcal{B}$. Поскольку в выводе 1)–8) правило Gen применялось только по переменной x, которая не свободна в формуле $\forall x \mathcal{A}$, по теореме о дедукции существует вывод формулы $\forall x \mathcal{A} \supset \forall x \mathcal{B}$ из совокупности гипотез $\{\forall x(\mathcal{A} \supset \mathcal{B})\}$, в котором правило Gen применяется только по переменной x. Но эта переменная не входит свободно и в оставшуюся гипотезу $\forall x(\mathcal{A} \supset \mathcal{B})$. Поэтому теорему о дедукции можно применить еще раз, так что формула $\forall x(\mathcal{A} \supset \mathcal{B}) \supset (\forall x \mathcal{A} \supset \forall x \mathcal{B})$ выводима (без гипотез).

Из теоремы о дедукции можно получить следствия, более слабые, чем сама теорема 1.4.2, но из-за простоты условий более удобные в применении.

Следствие 1.4.3. *Если существует вывод $\mathcal{B}$ из $\Gamma,\mathcal{A}$, в котором правило обобщения по свободным переменным формулы $\mathcal{A}$ не применяется, то $\Gamma \vdash \mathcal{A} \supset \mathcal{B}$.*

Следствие 1.4.4. *Если $\Gamma,\mathcal{A} \vdash \mathcal{B}$ и формула $\mathcal{A}$ замкнута, то $\Gamma \vdash \mathcal{A} \supset \mathcal{B}$.*

Выполнение условий теоремы 1.4.2 в обоих случаях легко проверяется.

Следствие 1.4.5. (Правило S). $\mathcal{A} \supset \mathcal{B}, \mathcal{B} \supset \mathcal{C} \vdash \mathcal{A} \supset \mathcal{C}$.

Следствие 1.4.6. (Правило I). $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C}), \mathcal{B} \vdash \mathcal{A} \supset \mathcal{C}$.

Следствие 1.4.7. (Правило R). $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C}) \vdash \mathcal{B} \supset (\mathcal{A} \supset \mathcal{C})$.

Доказательство этих следствий теоремы о дедукции полностью совпадает с доказательством аналогичных следствий для исчисления высказываний. Таким образом, производными правилами вывода S, I и R можно пользоваться и в ФТПП.

В ФТПП существуют и свои специфические производные правила вывода, не связанные с теоремой о дедукции.

Теорема 1.4.8. $\forall x \mathcal{A}(x) \vdash \mathcal{A}(t)$, *если терм t свободен для переменной x в формуле $\mathcal{A}(x)$.*

Доказательство. Последовательность шагов:

1) $\forall x \mathcal{A}(x)$ (гипотеза)
2) $\forall x \mathcal{A}(x) \supset \mathcal{A}(t)$ (A4)
3) $\mathcal{A}(t)$ (1, 2, MP)

является требуемым выводом. $\lhd$

Производное правило вывода

$$\frac{\forall x \mathcal{A}(x)}{\mathcal{A}(t)}$$

(при указанном ограничении) называется *правилом индивидуализации* или правилом A.

Теорема 1.4.9. $\mathcal{A}(t) \vdash \exists x \mathcal{A}(x)$, *если терм t свободен для переменной x в формуле $\mathcal{A}(x)$.*

Напомним, что под $\exists x \mathcal{A}(x)$ следует понимать формулу $\neg \forall x \neg \mathcal{A}(x)$.

Доказательство. Выписываем схему вывода.

1) $\mathcal{A}(t)$ (гипотеза)
2) $\forall x \neg \mathcal{A}(x) \supset \neg \mathcal{A}(t)$ (A4)
3) $(\forall x \neg \mathcal{A}(x) \supset \neg \mathcal{A}(t)) \supset (\mathcal{A}(t) \supset \neg \forall x \neg \mathcal{A}(x))$ (тавтология)
4) $\mathcal{A}(t) \supset \neg \forall x \neg \mathcal{A}(x)$ (2, 3, MP)
5) $\neg \forall x \neg \mathcal{A}(x)$, т. е. $\exists x \mathcal{A}(x)$ (1, 4, MP)

Формула 3) включена в схему вывода на основании теоремы 1.4.1 о выводимости частного случая тавтологии. В данном случае имеется в виду тавтология

$(B_1 \supset \neg B_2) \supset (B_2 \supset \neg B_1)$. В дальнейшем на вид подразумеваемой тавтологии ссылаться не будем. $\triangleleft$

При том же ограничении производное правило вывода

$$\frac{\mathcal{A}(t)}{\exists x \mathcal{A}(x)}$$

называется *правилом существования* или правилом E.

1.4.3. Некоторые теоремы об истинности и общезначимости

Никаких общих утверждений типа теоремы о полноте для произвольных ФТПП доказано быть не может из-за наличия в них собственных аксиом. Что касается ИППП, то одна часть теоремы о полноте доказывается просто (настолько просто, что эту часть неудобно называть половиной теоремы).

Теорема 1.4.10. *Любая теорема любого исчисления предикатов первого порядка общезначима.*

Доказательство. Все логические аксиомы общезначимы. Это следует из разд. 1.2.7 и теоремы 1.2.14.

Любая формула $\mathcal{B}_i$, получаемая из общезначимых формул $\mathcal{B}_j$ и $\mathcal{B}_k$ по правилу MP, сама общезначима. Любая формула $\mathcal{B}_i$, получаемая из общезначимой формулы $\mathcal{B}_j$ по правилу Gen, общезначима. Оба эти утверждения вытекают из следствий 1.2.5 и 1.2.7 теорем 1.2.4 и 1.2.6. Таким образом, в выводе любой формулы в произвольном ИППП не может возникнуть ни одной необщезначимой формулы. Следовательно, общезначима и заключительная формула такого вывода. Разумеется, сказанное выше — это намек на то, что более аккуратное доказательство следует вести по индукции. $\triangleleft$

Может быть доказано и обратное утверждение: если формула некоторого ИППП общезначима, то она выводима в этом исчислении. Из этого утверждения и доказанной теоремы следует теорема о полноте любого ИППП. Но доказательства второй части этой теоремы ввиду его сложности мы не даем. Кроме того, оно неконструктивно — не существует никакого метода, который по данной замкнутой формуле некоторого ИППП позволял бы установить или опровергнуть ее выводимость в этом исчислении.

Моделью ФТПП называется интерпретация этой теории, в которой истинны все ее собственные аксиомы.

Теорема 1.4.11. *Любая теорема теории первого порядка* **Q** *истинна в любой модели этой теории.*

Доказательство. Собственные аксиомы теории **Q** истинны в любой ее модели по определению, логические аксиомы — в силу их общезначимости. Если некоторые формулы истинны в какой-либо интерпретации теории **Q**, то истинны (по следствиям 1.2.5 и 1.2.7 теорем 1.2.4 и 1.2.6) и непосредственные следствия этих формул по правилам MP и Gen. Следовательно, в любой модели теории **Q** истинна любая формула, в том числе — и заключительная, любого вывода. $\triangleleft$

Опять-таки может быть, хотя и с большими сложностями, доказано обратное — если некоторая формула истинна в любой модели некоторой ФТПП, то она выводима в этой теории.

1.4.4. Непротиворечивость

Формальная теория первого порядка $\mathbf{Q}$ называется *непротиворечивой*, если не существует такой формулы $\mathcal{A}$, что $\vdash_{\mathbf{Q}} \mathcal{A}$ и $\vdash_{\mathbf{Q}} \neg\mathcal{A}$.

Из тавтологии $\neg B_1 \supset (B_1 \supset B_2)$ следует, что в противоречивой ФТПП выводима любая формула $\mathcal{B}$ (выводим формулу $\neg\mathcal{A} \supset (\mathcal{A} \supset \mathcal{B})$, а затем — дважды по MP — формулу $\mathcal{B}$). Существование невыводимых формул (хотя бы одной) можно принять за альтернативное определение непротиворечивости.

Теорема 1.4.12. *Любое исчисление предикатов первого порядка* $\mathbf{Q}$ *непротиворечиво.*

Доказательство. Если бы нашлась формула $\mathcal{A}$, такая что и $\mathcal{A}$ и $\neg\mathcal{A}$ выводимы в ИППП $\mathbf{Q}$, то по теореме 1.4.10 обе эти формулы были бы общезначимы, что невозможно. $\lhd$

Лемма 1.4.13. *Если замкнутая формула* $\neg\mathcal{A}$ *невыводима в ФТПП* $\mathbf{Q}$, *то формальная теория* $\mathbf{Q}'$, *отличающаяся от* $\mathbf{Q}$ *лишь тем, что к множеству ее аксиом добавлена формула* $\mathcal{A}$, *непротиворечива.*

Доказательство. Предположим, что $\mathbf{Q}'$ противоречива. Тогда в ней выводима любая формула, в частности формула $\neg\mathcal{A}$. Вывод формулы $\neg\mathcal{A}$ в теории $\mathbf{Q}'$ одновременно является выводом $\neg\mathcal{A}$ из списка, содержащего единственную гипотезу $\mathcal{A}$, в теории $\mathbf{Q}$, так что $\mathcal{A} \vdash_{\mathbf{Q}} \neg\mathcal{A}$. Формула $\mathcal{A}$ замкнута вместе с $\neg\mathcal{A}$ и не содержит поэтому никаких свободных переменных. Таким образом, можно применить следствие 1.4.4 и получить $\vdash_{\mathbf{Q}} \mathcal{A} \supset \neg\mathcal{A}$. Из тавтологии $(B_1 \supset \neg B_1) \supset \neg B_1$ следует, что $\vdash_{\mathbf{Q}} (\mathcal{A} \supset \neg\mathcal{A}) \supset \neg\mathcal{A}$, а вместе с тем и $\vdash_{\mathbf{Q}} \neg\mathcal{A}$, что противоречит условию теоремы. Следовательно, $\mathbf{Q}'$ не может быть противоречивой. $\lhd$

Лемма 1.4.14. *Если формальная теория первого порядка* $\mathbf{Q}$ *имеет хотя бы одну модель, то она непротиворечива.*

Доказательство. Предположим, что $\mathbf{Q}$ противоречива. Тогда найдется формула $\mathcal{A}$ такая, что $\vdash_{\mathbf{Q}} \mathcal{A}$ и $\vdash_{\mathbf{Q}} \neg\mathcal{A}$. По теореме 1.4.11 как формула $\mathcal{A}$, так и формула $\neg\mathcal{A}$ истинны в имеющейся у теории $\mathbf{Q}$ модели, чего быть не может. $\lhd$

1.4.5. Теоремы о выводимости

Следующий вариант теоремы об эквивалентности можно назвать синтаксическим, так как в нем, в отличие от теоремы 1.2.8, речь пойдет о преобразовании текстов формул, а не об их интерпретации.

Теорема 1.4.15. *Если формула* $\mathcal{A}'$ *получается из формулы* $\mathcal{A}$ *заменой нескольких вхождений подформулы* $\mathcal{B}$ *на формулу* $\mathcal{C}$, *и если* $y_1, \ldots, y_k$ — *список*

свободных переменных формул $\mathcal{B}$ и $\mathcal{C}$, которые связаны в формуле $\mathcal{A}$, то в любом исчислении предикатов первого порядка $\vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{A} \equiv \mathcal{A}')$.

Доказательство (индукция по числу n логических и кванторных связок в формуле $\mathcal{A}$, не входящих в заменяемые вхождения подформулы $\mathcal{B}$).

База индукции ($n = 0$). Рассмотрим два возможных вида элементарной формулы $\mathcal{A}$.

а. $\mathcal{A}$ совпадает с $\mathcal{B}$. В этом случае $\mathcal{A}'$ совпадает с $\mathcal{C}$, и требуется установить выводимость формулы $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{B} \equiv \mathcal{C})$. Исходя из гипотезы $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C})$, применим k раз правило индивидуализации для $t = y_1, \ldots, y_k$. На последнем шаге получим формулу $\mathcal{B} \equiv \mathcal{C}$. По теореме о дедукции интересующая нас формула выводима.

б. $\mathcal{A}$ — элементарная формула, не совпадающая с $\mathcal{B}$. При этом $\mathcal{A}'$ совпадает с $\mathcal{A}$, формула, выводимость которой требуется установить, имеет вид $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{A} \equiv \mathcal{A})$. Это частный случай тавтологии $B_1 \supset (B_2 \equiv B_2)$, он выводим по теореме 1.4.1.

Индукционный шаг. Пусть для любой формулы, содержащей меньше n связок, соответствующее ей утверждение справедливо.

а. Формула $\mathcal{A}$ имеет вид $\neg\mathcal{D}$, формула $\mathcal{A}'$ — вид $\neg\mathcal{D}'$. По предположению, $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{D} \equiv \mathcal{D}')$. Тавтология $(\mathcal{D} \equiv \mathcal{D}') \supset (\neg\mathcal{D} \equiv \neg\mathcal{D}')$ вместе с правилом S дает $\vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\neg\mathcal{D} \equiv \neg\mathcal{D}')$, а это и есть то, что нам нужно.

б. Формула $\mathcal{A}$ имеет вид $\mathcal{D} \supset \mathcal{E}$, формула $\mathcal{A}'$ — вид $\mathcal{D}' \supset \mathcal{E}'$. По предположению индукции

$$\vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{D} \equiv \mathcal{D}'), \qquad \vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{E} \equiv \mathcal{E}').$$

Исходя из гипотезы $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C})$ и тавтологии

$$(\mathcal{D} \equiv \mathcal{D}') \supset ((\mathcal{E} \equiv \mathcal{E}') \supset ((\mathcal{D} \supset \mathcal{E}) \equiv (\mathcal{D}' \supset \mathcal{E}'))),$$

дважды применяя правило MP, а затем правило S и теорему дедукции, получим требуемый результат.

в. Формула $\mathcal{A}$ имеет вид $\forall x\mathcal{D}$, тогда $\mathcal{A}'$ должна иметь вид $\forall x\mathcal{D}'$. По предположению индукции $\vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{D} \equiv \mathcal{D}')$, если x содержится среди $y_1, \ldots, y_k$, и $\vdash \forall x\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{D} \equiv \mathcal{D}')$, если переменная x входит свободно в $\mathcal{B}$ или в $\mathcal{C}$ и не связана в $\mathcal{D}$. В обоих случаях x не входит свободно в посылку импликации. Применяя правило обобщения по x, получим $\vdash \forall x(\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{D} \equiv \mathcal{D}'))$ или $\vdash \forall x(\forall x\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset (\mathcal{D} \equiv \mathcal{D}'))$.

С помощью аксиомы по схеме A5 и правила MP получаем

$$\vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset \forall x(\mathcal{D} \equiv \mathcal{D}') \tag{4}$$

или

$$\vdash \forall x\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset \forall x(\mathcal{D} \equiv \mathcal{D}'). \tag{5}$$

По правилу Gen $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \vdash \forall x\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C})$, а так как x не входит свободно в $\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C})$ (потому что эта переменная либо не входит свободно ни в $\mathcal{B}$ ни в $\mathcal{C}$, или же связана в $\mathcal{A}$), то применима теорема о дедукции: $\vdash \forall y_1, \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C}) \supset \forall x\forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C})$. По правилу S случай (5) сводится к (4).

Вспомним, что формула $\mathcal{F} \equiv \mathcal{G}$ служит сокращением для $(\mathcal{F} \supset \mathcal{G}) \wedge (\mathcal{G} \supset \mathcal{F})$. Используя тавтологии $B_1 \wedge B_2 \supset B_1$, $B_1 \wedge B_2 \supset B_2$ и $B_1 \supset (B_2 \supset B_1 \wedge B_2)$, можно обосновать выводимость $\vdash \forall x(\mathcal{D} \equiv \mathcal{D}') \supset (\forall x\mathcal{D} \equiv \forall x\mathcal{D}')$, действуя по образцу примера 1.4.2–2. Отсюда и из (4) получаем то, что требуется. $\lhd$

Следствие 1.4.16 (теорема об эквивалентной замене)**.** *В условиях теоремы об эквивалентности, если* $\vdash \mathcal{B} \equiv \mathcal{C}$, *то* $\vdash \mathcal{A} \equiv \mathcal{A}'$, *а если, кроме того,* $\vdash \mathcal{A}$, *то* $\vdash \mathcal{A}'$.

Доказательство. Если $\vdash \mathcal{B} \equiv \mathcal{C}$, то $\vdash \forall y_1 \ldots \forall y_k(\mathcal{B} \equiv \mathcal{C})$ (требуется k раз применить правило Gen). Далее, по MP получаем $\vdash \mathcal{A} \equiv \mathcal{A}'$, а с помощью тавтологии $(B_1 \equiv B_2) \supset (B_1 \supset B_2)$ — $\vdash \mathcal{A} \supset \mathcal{A}'$. Если, сверх того, $\vdash \mathcal{A}$, то $\vdash \mathcal{A}'$ (по MP). $\lhd$

Упражнение 1.4.1. Если формула $\mathcal{A}$ не содержит свободных вхождений переменной y, то $\forall y(\mathcal{B} \supset \mathcal{A}) \equiv (\exists y\mathcal{B} \supset \mathcal{A})$.

Формулы $\mathcal{A}(x)$ и $\mathcal{A}(y)$ называются *подобными*, если $\mathcal{A}(y)$ содержит свободные вхождения y в точно тех же местах, где x свободно входит в $\mathcal{A}(x)$. Подробнее — если $\mathcal{A}(y)$ получается из $\mathcal{A}(x)$ подстановкой переменной y вместо всех свободных вхождений переменной x, и при этом y свободна для x в $\mathcal{A}(x)$ и в $\mathcal{A}(x)$ нет свободных вхождений y.

Упражнение 1.4.2. Если формулы $\mathcal{A}(x)$ и $\mathcal{A}(y)$ подобны, то $\vdash \mathcal{A}(x) \equiv \mathcal{A}(y)$.

В логике удобно пользоваться правилом, которое разрешает по ходу вывода из формулы

$$\exists x\mathcal{A}(x) \tag{6}$$

получать следствие вида $\mathcal{A}(b)$, где b — предметная константа, обозначающая некоторый объект, существование которого утверждает формула (6). Эта константа вводится только на время вывода и не должна присутствовать в формуле, завершающей вывод, чтобы произвол в выборе объекта b не отражался на сущности выведенного утверждения.

Это правило называется *правилом выбора* или правилом C (от английского choice — выбор).

Пример. Формулу $\exists x(\mathcal{A}(x) \supset \mathcal{B}(x)) \supset (\forall x\mathcal{A}(x) \supset \exists x\mathcal{B}(x))$ можно вывести, используя правило C, следующим образом:

1) $\exists x(\mathcal{A}(x) \supset \mathcal{B}(x))$ (гипотеза)
2) $\forall x\mathcal{A}(x)$ (гипотеза)
3) $\mathcal{A}(b) \supset \mathcal{B}(b)$ (1; правило C)
4) $\mathcal{A}(b)$ (2; A)
5) $\mathcal{B}(b)$ (4, 3; MP)
6) $\exists x\mathcal{B}(x)$ (5; E)

Итак, $\exists x(\mathcal{A}(x) \supset \mathcal{B}(x)), \forall x\mathcal{A}(x) \vdash \exists x\mathcal{B}(x))$. Двукратное применение теоремы о дедукции приводит к желаемому результату.

Следующая теорема показывает, что правило C не позволяет вывести что-либо, не выводимое без помощи этого правила. Но предварительно необходимо точно определить понятие вывода с правилом C (C-вывода) в теории первого порядка $\mathbf{Q}$. Запись $\Gamma \vdash_{\mathrm{C}} \mathcal{A}$ означает, что существует такая последовательность формул $\mathcal{B}_1, \ldots, \mathcal{B}_n$, с $\mathcal{B}_n = \mathcal{A}$, что для любого i, $1 \leqslant i \leqslant n$, формула $\mathcal{B}_i$ — это либо аксиома теории $\mathbf{Q}$, либо взята из списка Γ, либо следует из предшествующих ей в выводе формул по правилу MP или Gen, либо, наконец, получена из уже содержащейся в выводе формулы $\exists x \mathcal{C}(x)$ по правилу C, т. е. имеет вид $\mathcal{C}(b)$, где b — новая предметная константа. При этом: (а) в аксиомы, используемые в выводе, могут входить предметные константы, вводимые при использовании правила C, (б) применение правила Gen по переменным, свободным хотя бы в одной формуле $\exists x \mathcal{C}(x)$, к которой ранее было применено правило C, не допускается, (в) формула $\mathcal{A}$ не должна содержать ни одной новой предметной константы, введенной при использовании правила C.

Теорема 1.4.17. *Если* $\Gamma \vdash_{\mathrm{C}} \mathcal{A}$, *то* $\Gamma \vdash \mathcal{A}$. *При этом, если во втором выводе (без правила* C*) применяется правило* Gen *по некоторой переменной, то оно применялось по той же переменной и в* C*-выводе.*

Доказательство. Пусть $\exists y_1 \mathcal{C}_1(y_1), \ldots, \exists y_k \mathcal{C}_k(y_k)$ — список (в порядке использования) формул, к которым в C-выводе формулы $\mathcal{A}$ из списка гипотез Γ было применено правило C, а формула $\mathcal{C}_i(b_i)$ для $i = 1, \ldots, k$ — результат соответствующего применения. Обозначим $\Gamma_k = \Gamma, \mathcal{C}_1(b_1), \ldots, \mathcal{C}_k(b_k)$, тогда тот же C-вывод можно считать обычным выводом формулы $\mathcal{A}$ из списка Γ_k (меняется лишь основание для включения формул вида $\mathcal{C}_i(b_i)$ в вывод). По теореме 1.4.2 (а пункт (б) в определении C-вывода позволяет ею воспользоваться) получаем $\Gamma_{k-1} \vdash \mathcal{C}_k(b_k) \supset \mathcal{A}$. Если всюду в соответствующем выводе заменить b_k на нигде в выводе не появлявшуюся переменную y, то убедимся, что $\Gamma_{k-1} \vdash \mathcal{C}_k(y) \supset \mathcal{A}$, а по правилу Gen получим $\Gamma_{k-1} \vdash \forall y(\mathcal{C}_k(y) \supset \mathcal{A})$. Воспользуемся общезначимой (см. упражнение 1.4.1) формулой $\forall y(\mathcal{B} \supset \mathcal{A}) \equiv (\exists y \mathcal{B} \supset \mathcal{A})$. Это дает $\Gamma_{k-1} \vdash (\exists y_k(\mathcal{C}_k(y_k) \supset \mathcal{A})$. Но формула $\exists y_k \mathcal{C}_k(y_k)$ была самой последней использована в исходном выводе, так что $\Gamma_{k-1} \vdash \exists y_k \mathcal{C}_k(y_k)$, и по правилу MP $\Gamma_{k-1} \vdash \mathcal{A}$. Применяя описанный прием еще $k-1$ раз, можно исключить из списка гипотез все оставшиеся в нем формулы вида $\mathcal{C}_i(b_i)$ и прийти к заключению $\Gamma \vdash \mathcal{A}$. $\lhd$

1.4.6. Теории первого порядка с равенством

Понятие равенства занимает в математике одно из центральных мест. Мы с этим столкнулись уже в разд. 1.1.1. Поэтому в большинстве ФТПП предикат равенства вводится с самого начала (см. пример 1.4.1–1). На роль символа равенства отводится предикатная буква P_1^2, запись $P_1^2(t, u)$ сокращается до $t = u$, а $\neg P_1^2(t, u)$ — до $t \neq u$. В примере 1.4.1–2 был указан способ построения полноценного заменителя предиката равенства исходя из первичного предиката '$\in$'.

Теория **Q** называется *теорией первого порядка с равенством*, если в ней выводимы (в простейшем варианте — входят в число ее аксиом) в дополнение к списку A1–A5 из разд. 1.4.1 формула

A6. $\forall x\,(x = x)$ (рефлексивность равенства),

и формулы, построенные по схеме

A7. $x = y \supset (\mathcal{A}(x, x) \supset \mathcal{A}(x, y))$ (подстановочность равенства).

В схеме A7 имеется в виду, что $\mathcal{A}$ — это произвольная формула, а все свободные вхождения в нее переменной x разделены на две группы (первая может быть пустой). Формула $\mathcal{A}(x, y)$ получается из $\mathcal{A}(x, x)$ заменой вхождений второй группы на y, причем переменная y должна быть свободна для x во всех этих заменяемых вхождениях.

Теорема 1.4.18. *Во всякой формальной теории первого порядка с равенством*

а. $\vdash t = t$ *для любого терма* t,

б. $\vdash x = y \supset y = x$ (*симметричность равенства*),

в. $\vdash x = y \supset (y = z \supset x = z)$ (*транзитивность равенства*).

Доказательство. а. Следует из A6 по правилу индивидуализации.

б. Формула $x = x$ выводима согласно пункту а. Пусть $\mathcal{A}(x, x)$ — это $x = x$ и $\mathcal{A}(x, y)$ — это $y = x$ (т. е. замене на y подлежит первое вхождение x в $x = x$). Схема A7 дает $x = y \supset (x = x \supset y = x)$. Из этих двух формул требуемая формула получается по правилу I (исключения промежуточной посылки).

в. Опираясь на схему A7 (с переменой ролей x и y), можно убедиться, что формула $y = x \supset (y = z \supset x = z)$ выводима. С помощью пункта б. получаем требуемый результат по правилу S. $\lhd$

Лемма 1.4.19. *Если для формулы* $\mathcal{A}$ *формула* A7 (*т.е. формула, построенная по этой схеме*) *выводима, то формула вида* $x = y \supset (\mathcal{A}(x, y) \supset \mathcal{A}(x, x))$ *также выводима.*

Доказательство. Из пункта б. теоремы 1.4.18 вытекает: $\vdash x = y \supset y = x$, а по схеме A7 можно обосновать выводимость $\vdash y = x \supset (\mathcal{A}(x, y) \supset \mathcal{A}(x, x))$, так как $\mathcal{A}(x, x)$ получается из $\mathcal{A}(x, y)$ заменой некоторых вхождений y на x. Требуемый результат получается отсюда по правилу S. $\lhd$

В схеме A7 можно и не требовать, чтобы формула $\mathcal{A}$ была совершенно произвольна. Это показывает, например, следующая

Теорема 1.4.20. *Если в теории первого порядка* **Q** *теоремами являются формула* A6 *и, для любой элементарной формулы* $\mathcal{A}$, *формула* A7, *то* **Q** — *теория первого порядка с равенством.*

Доказательство. Для такой теории теорема 1.4.18 остается верной, так как при ее доказательстве схема A7 применялась только для элементарных формул в роли формулы $\mathcal{A}$. Надо доказать, что формула A7 является теоремой теории **Q**, если $\mathcal{A}$ — произвольная формула. Сделаем это индукцией по числу n логических

и кванторных связок в формуле $\mathcal{A}$. База индукции — случай $n = 0$ — вытекает непосредственно из условия теоремы. Предположим, что для формул с менее чем n связками требуемое утверждение справедливо.

а. Пусть $\mathcal{A}(x, x)$ имеет вид $\neg\mathcal{B}(x, x)$. Тогда $\mathcal{A}(x, y)$ — это $\neg\mathcal{B}(x, y)$. По предположению индукции формула, построенная по схеме A7, но с $\mathcal{B}$ вместо $\mathcal{A}$, выводима. Тогда, по лемме 1.4.19, $\vdash x = y \supset (\mathcal{B}(x, y) \supset \mathcal{B}(x, x))$. Используя тавтологию $(\mathcal{B}(x, y) \supset \mathcal{B}(x, x)) \supset (\neg\mathcal{B}(x, x) \supset \neg\mathcal{B}(x, y))$, по правилу S можно вывести формулу $x = y \supset (\mathcal{A}(x, x) \supset \mathcal{A}(x, y))$, что и требуется.

б. Пусть $\mathcal{A}(x, x)$ имеет вид $\mathcal{B}(x, x) \supset \mathcal{C}(x, x)$. Тогда $\mathcal{A}(x, y)$ — это формула $\mathcal{B}(x, y) \supset \mathcal{C}(x, y)$. По предположению индукции $\vdash x = y \supset (\mathcal{B}(x, y) \supset \mathcal{B}(x, x))$ (на основании леммы 1.4.19) и $\vdash x = y \supset (\mathcal{C}(x, x) \supset \mathcal{C}(x, y))$. Если принять формулы $x = y$ и $\mathcal{B}(x, x) \supset \mathcal{C}(x, x)$, т. е. $\mathcal{A}(x, x)$, за гипотезы, то нетрудно, применяя по два раза правила MP и S, вывести формулу $\mathcal{B}(x, y) \supset \mathcal{C}(x, y)$, т. е. $\mathcal{A}(x, x)$. Двукратное применение теоремы о дедукции дает $\vdash x = y \supset (\mathcal{A}(x, x) \supset \mathcal{A}(x, y))$.

в. Пусть $\mathcal{A}(x, x)$ имеет вид $\forall z\mathcal{B}(x, x, z)$, а $\mathcal{A}(x, y)$ — вид $\forall z\mathcal{B}(x, y, z)$. По предположению индукции $\vdash x = y \supset (\mathcal{B}(x, x, z) \supset \mathcal{B}(x, y, z))$. Комбинируя правила MP, A и Gen, легко из гипотез $x = y$ и $\mathcal{A}(x, x)$ вывести формулу $\mathcal{A}(x, y)$. Теорема о дедукции снова приводит к желаемой цели. $\lhd$

Количество вариантов формулы A7, выводимость которых должна быть обоснована, можно сократить еще больше, если выполнено условие подстановочности равенства для термов: для каждой функциональной буквы f_j^k, для каждого набора переменных $z_1, \ldots, z_k$ и для любого s, $1 \leqslant s \leqslant k$, имеет место выводимость

$$\vdash z_s = w \supset f_j^k(z_1, \ldots, z_k) = f_j^k(z_1, \ldots, z_{s-1}, w, z_{s+1}, \ldots, z_k).$$

Легко доказывается (индукцией по числу вхождений функциональных букв в терм t) следующая

Лемма 1.4.21. *Если выполнено условие подстановочности равенства для термов, то для произвольного терма t имеет место $\vdash x = y \supset t(x, x) = t(x, y)$, где терм $t(x, y)$ получается из $t(x, x)$ заменой части вхождений x на y.*

Ясно, что замена любого числа вхождений переменных может быть сведена к их последовательной замене по одному.

Теорема 1.4.22. *Пусть в теории первого порядка* **Q** *выводима формула* A6 *и все формулы вида* A7, *в которых формула $\mathcal{A}(x, x)$ элементарна, не содержит вхождений функциональных букв и $\mathcal{A}(x, y)$ получается из нее заменой лишь одного вхождения x на y. Если, кроме того, выполнено условие подстановочности равенства для термов, то* **Q** *— теория первого порядка с равенством.*

Доказательство. И в этих условиях теорема 1.4.18 остается верной. Опираясь на лемму 1.4.21 и применяя правило индивидуализации A, можно установить, что выполнены условия теоремы 1.4.20. $\lhd$

Покажем более конкретно, как это делается, на простом примере — выведем формулу $x = y \supset \mathcal{A}(f(x)) \supset \mathcal{A}(f(y))$:

1) $x = y \supset f(x) = f(y)$ (по подстановочности равенства для термов)
2) $\forall u \forall v\,(u = v \supset (\mathcal{A}(u) \supset \mathcal{A}(v)))$ (выводима по условию теоремы)
3) $\forall v\,(f(x) = v \supset (\mathcal{A}(f(x)) \supset \mathcal{A}(v)))$ (2, A)
4) $f(x) = f(y) \supset (\mathcal{A}(f(x)) \supset \mathcal{A}(f(y)))$ (3, A)
5) $x = y \supset (\mathcal{A}(f(x)) \supset \mathcal{A}(f(y)))$ (1, 4, S)

Если в теории первого порядка **Q** имеется предикат $\mathcal{E}(x,\, y)$ (будем писать $t = u$ вместо $\mathcal{E}(t,\, u)$), для которого формула A6 и все формулы A7 выводимы, то говорят, что в этой теории равенство может быть определено. Надо лишь условиться, что если термы t и u не свободны для переменных x и y в формуле $\mathcal{E}(x,\, y)$, то $t = u$ заменяет собой не формулу $\mathcal{E}(t,\, u)$, а формулу $\mathcal{E}'(t,\, u)$, получающуюся из $\mathcal{E}(t,\, u)$ переименованием связанных переменных (см. упражнение 1.4.2) так, чтобы t и u стали свободными для x и y в формуле $\mathcal{E}'(x,\, y)$. Теоремы 1.4.20 и 1.4.22 остаются в силе, если формулы A7 выводимы в теории **Q** для формул $\mathcal{E}'(x,\, y)$ того вида, о котором говорится в условиях этих теорем.

Так, в примере 1.4.1–2 предикат, введенный определением (1.4.1–1), где сразу же использовано сокращенное обозначение '$=$' для этого предиката, обладает требуемыми свойствами. Действительно, в этом примере имеется единственный первичный предикат $\in$, все остальные определяются через него. Определение (1.4.1–1) и аксиома (1.4.1–2) обеспечивают выполнение условий теоремы 1.4.22. Без аксиомы (1.4.1–2) определения (1.4.1–1) было бы недостаточно, чтобы ввести в теорию множеств полноценный предикат равенства. Она заранее обеспечивает корректность всех последующих определений, вводимых в этой теории, в которых прямо или косвенно используется понятие равенства множеств.

В математической теории, где существует понятие равенства, часто заходит речь о единственности объекта x, обладающего свойством $\mathcal{A}(x)$. Этот оборот речи выражается формулой $\exists x \mathcal{A}(x) \wedge \forall x \forall y\,(\mathcal{A}(x) \wedge \mathcal{A}(y) \supset x = y)$. Употребительным сокращением для этой формулы служит $\exists_1 x \mathcal{A}(x)$ или $\exists! x \mathcal{A}(x)$.

Если доказано, что для некоторых $x_1,\, \ldots,\, x_n$ существует единственный объект u со свойством $\mathcal{A}(u,\, x_1,\, \ldots,\, x_n)$, то можно ввести новую функциональную букву f, имея в виду, что условие $\mathcal{A}(f(x_1, \ldots, x_n),\, x_1,\, \ldots,\, x_n)$ выполняется для любых $x_1, \ldots, x_n$. Аналогично (при $n = 0$) можно ввести новую предметную константу b со свойством $\mathcal{A}(b)$. Подобные определения, хотя и облегчают изложение теории, не расширяют ее возможностей — все ее результаты могут быть получены и без их помощи (есть некоторая аналогия с использованием правила C). На многочисленных подробностях, связанных с использованием таких определений, останавливаться не будем.

1.5. Теория множеств

Теория множеств вместе с математической логикой (и теорией алгоритмов) является одним из краеугольных камней современной математики. Почти каждая математическая теория описывает свои основные, первоначальные понятия в терминах теории множеств. Мы увидим, что в рамках теории множеств могут быть определены такие понятия, как отношение, функция, порядок, натуральное число и др. Эта тенденция описывать математические понятия на теоретико-множественной основе привела к тому, что первичные понятия самой теории множеств стали общеизвестны и появились, в частности, в школьных программах. Поэтому мы, по крайней мере — на первых порах, будем не столько разъяснять эти понятия, сколько записывать их на языке логических формул, что укрепит навыки в пользовании им.

В то же время теория множеств будет излагаться не на формальном, а на содержательном уровне, примерно так, как подходил к ней главный из ее создателей Г. Кантор. В таком изложении теория множеств называется «наивной». Об аксиоматике теории множеств будет дано лишь общее представление.

Сам термин «множество» никак не определяется. Можно процитировать (взято из [41]) Кантора: «Множество или совокупность — это собрание определенных и различных объектов нашей интуиции или интеллекта, мыслимое в качестве целого (единого)», — но эти слова являются, в лучшем случае, пояснением, а не определением этого понятия. Отметим также следующее. При популярном изложении теории множеств часто приводят такие примеры, как множество волос на голове, множество солдат в роте или жирафов, живущих в Антарктиде, однако эти примеры не следует считать удачными. Мы занимаемся математикой, все понятия которой абстрактны. Недаром Кантор отнес их к «объектам нашей интуиции или интеллекта». Волосы, солдаты и жирафы — это не математические объекты.

1.5.1. Основные понятия

Мы уже начали формальное (наполовину) изложение теории множеств в примере 1.4.1–2. Напомним, там говорилось, что все объекты теории являются множествами и ничем иным.

Были введены обозначения типа $\{a, b, c\}$ для множества, состоящего из элементов a, b, c, или $\{1, 3, 5, \ldots\}$ для множества всех нечетных натуральных чисел.

Более общий способ обозначения множеств имеет вид:

$$A = \{x \mid \text{Требование к элементам } x \text{ множества } A\}. \tag{1}$$

Это частный случай еще более общего способа обозначения, когда множество составляется из значений терма, значения переменных в котором должны удовлетворять некоторым условиям:

$$A = \{t(x_1, \ldots, x_k) \mid \text{Ограничение на значения } x_1, \ldots, x_k\} \tag{2}$$

Так, например, множество $M = \{1, 3, 5, \ldots\}$ более формально задается одним из следующих способов (предполагается, что множество $\mathbf{N} = \{0, 1, 2, \ldots\}$ всех натуральных чисел уже было определено):

$$M = \{x \mid x = 2n + 1, \quad n \in \mathbf{N}\},$$
$$M = \{2n + 1 \mid n \in \mathbf{N}\}.$$

Существенно, что подобное определение однозначно устанавливает для каждого множества x, является ли оно элементом определяемого множества A.

Пустым называется множество $\emptyset$, не содержащее элементов:

$$\forall x \neg x \in \emptyset.$$

Было сказано, что через единственное первичное отношение между объектами — отношение принадлежности $\in$ — выражаются все другие отношения между множествами. Двухместное отношение R определяется с помощью формулы $\mathcal{F}$ со свободными переменными x и y, что записывается в виде «$xRy \equiv \mathcal{F}$», в словесной формулировке — «xRy является сокращением для $\mathcal{F}$».

Было определено отношение непринадлежности $\notin$ элемента множеству формулой

$$x \notin y \equiv \neg x \in y,$$

и отношение включения $\subseteq$ между двумя множествами с определением

$$x \subseteq y \equiv \forall z(z \in x \supset z \in y),$$

Следует четко различать отношения принадлежности и включения. Например, множество $\{a\}$ принадлежит множеству $A = \{\{a\}, b\}$, но не является его подмножеством, а множество $\{b\}$ является подмножеством, но не элементом A. Элементы множества $B = \{a, b, \{a, b\}\}$ — это a, b и $\{a, b\}$, причем $\{a, b\}$ является, кроме того, и подмножеством множества B. Другие подмножества B — это, например, $\{a\}$ и $\{b, \{a, b\}\}$, но не a и не b.

Отношение равенства между множествами было задано определением

$$x = y \equiv \forall z(z \in x \equiv z \in y).$$

Чтобы так определенное отношение обладало всеми свойствами равенства (см. разд. 1.4.6), потребовалась аксиома *объемности* (*экстенсиональности*):

$$x = y \supset \forall z(x \in z \equiv y \in z).$$

Упражнение 1.5.1. Доказать, что $x = y \equiv (x \subseteq y \wedge y \subseteq x)$.

Дадим еще два определения:

$$x \neq y \equiv \neg x = y,$$
$$x \subset y \equiv x \subseteq y \wedge x \neq y.$$

Упражнение 1.5.2. Доказать, что

$$x \subset y \equiv \forall z(z \in x \supset z \in y) \wedge \exists z(z \in y \wedge z \notin x).$$

Определяя результат некоторой операции над множествами, будем использовать схемы, представленные формулами (1) и (2).

Так, хорошо известная операция $\cup$ объединения множеств определяется формулой

$$A \cup B = \{x \mid x \in A \vee x \in B\}. \tag{3}$$

Аналогично определяется операция пересечения множеств (знак '$\cap$'):

$$A \cap B = \{x \mid x \in A \wedge x \in B\}. \tag{4}$$

Разность $A\backslash B$ множеств A и B имеет определение

$$A\backslash B = \{x \mid x \in A \wedge x \notin B\}. \tag{5}$$

Благодаря отмеченному выше свойству определений вида (1) и (2), новые операции определяются корректно (см. сказанное в конце разд. 1.4.6 о роли аксиомы (1.4.1–2)).

Симметрическая разность $A \triangle B$ может быть определена несколько иным, но столь же корректным способом — как композиция уже определенных операций:

$$A \triangle B = (A\backslash B) \cup (B\backslash A). \tag{6}$$

Из определений (3) и (4) видно, что теоретико-множественные операции объединения и пересечения тесно связаны с логическими операциями дизъюнкции и конъюнкции. Они обладают теми же свойствами:
идемпотентности:

$$A \cup A = A, \qquad A \cap A = A,$$

коммутативности:

$$A \cup B = B \cup A, \qquad A \cap B = B \cap A,$$

ассоциативности:

$$A \cup (B \cup C) = (A \cup B) \cup C, \qquad A \cap (B \cap C) = (A \cap B) \cap C,$$

дистрибутивности друг по отношению к другу:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C), \qquad A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

Доказываются эти свойства очень просто. Рассмотрим, например, последнее. Возьмем произвольный объект (т. е. множество) x. Имеем:

$x \in A \cap (B \cup C)$
$\Leftrightarrow x \in A \wedge x \in (B \cup C)$ (по определению пересечения)
$\Leftrightarrow x \in A \wedge (x \in B \vee (x \in C)$ (по определению объединения)
$\Leftrightarrow (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)$ (по дистрибутивности конъюнкции по отношению к дизъюнкции)
$\Leftrightarrow (x \in A \cap B) \vee (x \in A \cap C)$ (по определению пересечения)
$\Leftrightarrow x \in ((A \cap B) \cup (A \cap C))$ (по определению объединения).

Итак, $x \in A \cap (B \cup C) \equiv x \in ((A \cap B) \cup (A \cap C))$. Но так как x был взят произвольно, то верна и формула

$$\forall x(x \in A \cap (B \cup C) \equiv x \in ((A \cap B) \cup (A \cap C))).$$

На основании определения равенства множеств ее можно переписать так:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

а это и есть то равенство, которое требовалось доказать.

Ассоциативность позволяет говорить об объединении $A_1 \cup A_2 \cup \ldots \cup A_n$ или пересечении $A_1 \cap A_2 \cap \ldots \cap A_n$ любого конечного числа множеств. Любая расстановка скобок в этих выражениях приводит к одному и тому же результату.

Понятие объединения (или пересечения) конечной совокупности явно указанных множеств $M = \{A_1, \ldots, A_n\}$ можно обобщить на случай произвольного, в том числе бесконечного, множества M.

Объединением (всех элементов) множества M называется множество, обозначаемое $\bigcup M$, которое определяется формулой

$$\bigcup M = \{x \mid \exists A(A \in M \wedge x \in A)\}.$$

Пересечение (всех элементов) непустого множества M обозначается $\bigcap M$ и определяется формулой

$$\bigcap M = \{x \mid \forall A(A \in M \supset x \in A)\}.$$

Рассмотрим, что получается в случае, когда M — пустое множество. При этом, каково бы ни было A, формула $A \in M$ ложна, следовательно, конъюнкция $A \in M \wedge x \in A$ также ложна, а импликация $A \in M \supset x \in A$ истинна независимо от x. Для объекта $\bigcup M$ результат ясен — это пустое множество. Что касается объекта $\bigcap M$, то ему должно принадлежать любое произвольно взятое множество. Понятие множества всех множеств оказывается противоречивым (подробнее см. ниже, в разд. 1.5.7), поэтому мы и потребовали в определении $\bigcap M$, чтобы множество M было непусто.

Если элементы множества M имеют обозначения A_c, где $c \in C$ и C — произвольное множество индексов (иначе — *меток* или *имен*), то обозначение $\bigcup M$ можно заменить на $\bigcup_{c \in C} A_c$, а $\bigcap M$ — на $\bigcap_{c \in C} A_c$. Например, если множество M определено как $M = \{F(x) \mid x \in X\}$, где F — некоторая функция, то $\bigcup M$ — это то же самое, что $\bigcup_{x \in X} F(x)$, так что $\bigcup_{x \in X} F(x) = \{y \mid \exists x\,(x \in X \wedge y \in F(x))\}$.

1.5.2. Пары и кортежи (n-ки)

Одна из фундаментальных операций в теории множеств — это построение множества с двумя заданными элементами. *Неупорядоченной парой* (чаще — просто *парой*) с элементами x и y называется множество, обозначаемое $\{x, y\}$, со следующим свойством:

$$u \in \{x, y\} \equiv u = x \vee u = y.$$

Таким образом, какова бы ни была природа объектов x и y (это могут быть множества произвольного состава), пара $\{x, y\}$ содержит не более двух элементов. Если $x = y$, то пара превращается в одноэлементное множество $\{x, x\}$=$\{x\}$. При этом (по идемпотентности дизъюнкции):

$$u \in \{x\} \equiv u = x.$$

Ясно, что $\{x, y\}$=$\{y, x\}$ (по коммутативности дизъюнкции). Это равенство показывает, в частности, что места в паре (как и в любом множестве) неразличимы.

Но очень часто требуется составить из двух элементов такой объект, в котором эти элементы различались бы по занимаемому ими месту (1-й и 2-й элементы). Такой объект называется *упорядоченной парой* и обозначается (x, y), где x и y — входящие в него элементы. Другое обозначение — $\langle x, y\rangle$. Утверждение

$$(x, y) = (u, v) \equiv (x = u) \wedge (y = v). \tag{7}$$

можно принять за аксиому, характеризующую главное свойство упорядоченных пар. В разд. 1.5.7 будет показано, как в формальной теории можно определить упорядоченную пару через неупорядоченную и доказать справедливость утверждения (7), не вводя специальной аксиомы.

Кортежем или *n-кой* (тройкой, четверкой и т. д.) или *конечной последовательностью* называется объект, составленный из n элементов $x_1, \ldots, x_n$, занимающих каждый свое место в кортеже, чем он и отличается от множества, состоящего из тех же элементов. Кортеж обозначается $(x_1, \ldots, x_n)$ или $\langle x_1, \ldots, x_n\rangle$, число n его элементов называется *длиной* кортежа. При $n = 2$ кортеж — это то же самое, что упорядоченная пара. При $n = 1$ считается, что $(x_1) = x_1$. При $n > 2$ кортеж определяется рекурсивно, как упорядоченная пара, составленная из кортежа длины $n - 1$ и элемента x_n:

$$(x_1, \ldots, x_n) = ((x_1, \ldots, x_n - 1), x_n).$$

Основное свойство кортежей выражается следующей теоремой, обобщающей свойство (7) упорядоченных пар.

Теорема 1.5.1.

$$(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \equiv x_1 = y_1 \wedge \ldots \wedge x_n = y_n,$$

т. е. два кортежа равны тогда и только тогда, когда их длины равны, а соответствующие (стоящие на одинаковых местах) элементы совпадают.

Доказательство (индукция по длине кортежа). Для $n = 1$ утверждение тривиально, при $n = 2$ оно следует из аксиомы (7).

Пусть $n > 2$ и уже установлено, что

$$(x_1, \ldots, x_{n-1}) = (y_1, \ldots, y_{n-1}) \equiv x_1 = y_1 \wedge \ldots \wedge x_{n-1} = y_{n-1}.$$

Имеем:

$$\begin{aligned}
&(x_1, \ldots, x_n) = (y_1, \ldots, y_n)\\
&\quad \Leftrightarrow ((x_1, \ldots, x_n - 1), x_n) = ((y_1, \ldots, y_n - 1), y_n)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(по определению кортежа)}\\
&\quad \Leftrightarrow (x_1, \ldots, x_{n-1}) = (y_1, \ldots, y_{n-1}) \wedge x_n = y_n \qquad \text{(по аксиоме (7))}\\
&\quad \Leftrightarrow x_1 = y_1 \wedge \ldots \wedge x_{n-1} = y_{n-1} \wedge x_n = y_n\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(по предположению индукции).}
\end{aligned}$$

$\triangleleft$

Понятия упорядоченной пары и кортежа могут быть обобщены на случай, когда их элементы принадлежат не одному и тому же, а разным множествам (их можно называть *типами*).

Пусть задана упорядоченная пара (A_1, A_2) типов (которые могут, впрочем, совпадать). Потребуем, чтобы первый элемент упорядоченной пары (x_1, x_2) принадлежал множеству (имел тип) A_1, а второй — множеству A_2. Множество всех

таких пар называется декартовым произведением множеств A_1 и A_2 и обозначается $A_1 \times A_2$:

$$A_1 \times A_2 = \{(x_1, x_2) \mid x_1 \in A_1 \wedge x_2 \in A_2\}.$$

Длина n кортежа $(A_1, \ldots, A_n)$ типов может быть произвольной. Тогда типизированные кортежи $(x_1, \ldots, x_n)$ определяются аналогично бестиповым кортежам, рассмотренным выше:

$$(x_1, \ldots, x_n) = ((x_1, \ldots, x_{n-1}), x_n), \quad x_n \in A_n$$

Декартовым произведением $A_1 \times A_2 \times \ldots \times A_n$ называется множество

$$A_1 \times A_2 \times \ldots \times A_n = \{(x_1, \ldots, x_n) \mid x_1 \in A_1, \ldots, x_n \in A_n\}$$

всех таких кортежей.

1.5.3. Отношения

В этом разделе мы существенно сузим мир рассматриваемых объектов, ограничив его множеством всех подмножеств некоторого декартова произведения $M = A_1 \times A_2 \times \ldots \times A_n$. Предметом рассмотрения будут также кортежи $(x_1, \ldots, x_n)$, являющиеся элементами этих подмножеств, и элементы x_i этих кортежей. Подмножества R множества M, они же — множества кортежей указанного вида, называются *отношениями* (на кортеже $(A_1, \ldots, A_n)$, но чаще это уточнение подразумевается, не всегда используется и более полное обозначение $(R, A_1, \ldots, A_n)$ данного отношения). Само множество M называется *полным* отношением на $(A_1, \ldots, A_n)$. Длина n кортежей называется *вместимостью* отношения, отношение с вместимостью n — *n-местным*. Если $A_1 = \ldots = A_n = A$, то говорят, что отношение задано на множестве A.

Над отношениями, как и над множествами любого вида, можно выполнять все знакомые операции: строить их объединение (знак операции '$\cup$'), пересечение ('$\cap$'), разность ('$\setminus$'), симметрическую разность ('$\triangle$'). К ним добавляется одноместная операция образования *дополнения* $\overline{R} = M \setminus R$ любого отношения R.

Если для произвольных множеств выражение $a \cup \{a\}$ было, пусть не всегда осмыслено, но все же допустимо, то сейчас подобный объект, скорее всего, окажется незаконным из-за несоблюдения требований к виду кортежей отношения или к типам их элементов.

Для отношений (R, A, B) и (R_1, B, C) их *произведение* (RR_1, A, C) определяется формулой

$$RR_1 = \{(x, z) \mid x \in A \ \wedge z \in C \ \wedge \exists y\, (y \in B \wedge xRy \wedge yR_1z))\}.$$

На произвольном множестве A определено *отношение тождества* I_A с помощью формулы $I_A = \{(x, x) \mid x \in A\}$. Для отношения R на A *степени* отношения определяются так:

$$R^0 = I_A,\ R^{n+1} = R^n R \quad \text{для} \ \ n = 0, 1, \ldots,$$

в частности, $R^1 = R$ (доказать!) и $R^2 = RR$. Из ассоциативности операции $\wedge$ следует ассоциативность операции умножения, которая обычным образом позволяет доказать, что $R^{n+1} = RR^n$. Для того же отношения R его *транзитивное замыкание* R^+ задается формулой

$$R^+ = \bigcup_{n\geqslant 1} R^n,$$

а *рефлексивно-транзитивное замыкание* R^* — формулой

$$R^* = \bigcup_{n\geqslant 0} R^n.$$

Принадлежность кортежа $(x_1, \ldots, x_n)$ отношению R может быть записана в виде $(x_1, \ldots, x_n) \in R$, но чаще используется запись $R(x_1, \ldots, x_n)$, представляющая собой утверждение: элементы $x_1, \ldots, x_n$ (в указанном порядке) находятся между собой в отношении R. Мы уже прибегали к сходной записи и к самому понятию k-местного отношения (предиката) в разд. 1.2.1.

Простейший случай — одноместное (*унарное*) отношение $R(x)$, где $x \in A$. Оно называется *свойством* на A и сводится к проверке принадлежности x некоторому подмножеству множества A.

Важный класс отношений составляют двухместные, чаще говорят *бинарные*, отношения R — подмножества декартова произведения $\mathrm{M}{=}A_1 \times A_2$. Будем писать A вместо A_1, B вместо A_2 и xRy вместо $(x, y) \in R$. Часто «отношение» понимается именно как «бинарное отношение». Примеры: $x = y$, $x < y$, $x \in A$, $A \subset B$.

Если $R \subseteq A \times B$, то *обратное* отношение $R^{-1} \subseteq B \times A$ определяется формулой

$$R^{-1} = \{(y, x) \mid x \in A \wedge y \in B \wedge xRy\}.$$

Характерные свойства бинарных отношений для случая $B = A$ таковы: отношение R

- *рефлексивно*, если $\forall x\,(xRx)$,
- *антирефлексивно*, если $\forall x\,(x\overline{R}x)$,
- *симметрично*, если $\forall x\,\forall y\,(xRy \supset yRx)$,
- *асимметрично*, если $\forall x\,\forall y\,(xRy \supset y\overline{R}x)$,
- *антисимметрично*, если $\forall x\,\forall y\,(xRy \wedge yRx \supset x = y)$,
- *транзитивно*, если $\forall x\forall y\forall z\,(xRy \wedge yRz \supset xRz)$,
- *связно*, если $\forall x\,\forall y\,(x = y \vee xRy \vee yRx)$.

Упражнение 1.5.3. Доказать, что асимметричное отношение антирефлексивно и антисимметрично.

Примеры. Для некоторых отношений результаты операций дополнения и обращения представлены в таблице

	$=$	$\neq$	$<$	$\leqslant$	$\equiv (\mathrm{mod}\ )$
Дополнение	$\neq$	$=$	$\geqslant$	$>$	$\not\equiv (\mathrm{mod}\ )$
Обращение	$=$	$\neq$	$>$	$\geqslant$	$\equiv (\mathrm{mod}\ )$

Для тех же отношений в следующей таблице помечено, какими свойствами они обладают.

	$=$	$\neq$	$<$	$\leqslant$	$\equiv (\mathrm{mod}\ )$
Рефлексивность	+	−	−	+	+
Антирефлексивность	−	+	+	−	−
Симметричность	+	+	−	−	+
Асимметричность	−	−	+	−	−
Антисимметричность	+	−	+	+	−
Транзитивность	+	−	+	+	+
Связность	−	+	+	+	−

Рефлексивное, симметричное и транзитивное отношение называется отношением *эквивалентности*.

С эквивалентностью тесно связано понятие *разбиения* множества *на классы*. Пусть C — некоторое конечное или бесконечное множество, его элементы c играют роль имен классов. Разбиением множества A на классы A_c назовем множество $M = \{A_c \mid c \in C\}$, удовлетворяющее условиям:

1. $\forall c\,(c \in C \supset A_c \subseteq A)$,
2. $A = \bigcup_{c \in C} A_c$,
3. $\forall c \forall c_1\,(c \in C \wedge c_1 \in C \wedge c \neq c_1 \supset A_c \cap A_{c_1} = \varnothing)\}$.

Примеры. 1. Тривиальное разбиение: $M = \{\{x\} \mid x \in A\}$. Здесь каждый элемент x множества A служит именем класса, единственным элементом которого является одноэлементное множество $\{x\}$.

2. Множество $\mathbf{Z}$ всех целых чисел при произвольном $n > 0$ разбивается на n классов $A_i = \{m \mid m \equiv i \pmod{n}\}$ для $0 \leqslant i < n$.

Множество C имен классов обычно формируется из однозначно выбранных представителей каждого класса, так что $\forall c\,(c \in C \supset c \in A_c)$. Так было сделано в примерах 1 (где выбора не было) и 2 (где выбор определяется условием, наложенным на имя i).

Пример. Пусть на плоскости задана некоторая прямая p. Две точки плоскости отнесем к одному классу, если они обе лежат на прямой, параллельной p. Каждая такая прямая образует класс разбиения множества точек плоскости. В качестве имени класса можно взять точку пересечения такой прямой с какой-нибудь прямой q, пересекающейся с прямой p. Прямая q становится множеством имен классов.

Отступление. В геометрии, откуда был взят последний пример, без особого насилия над разумом (впрочем, для кого — как) можно отождествить прямую с множеством лежащих на ней точек. Несколько сложнее воспринимается отождествление точки с множеством проходящих через нее прямых. Но если пойти на это, то возникает обобщенный тип «геометрический объект», разбитый на два класса: «точек» и «прямых», и, например, одно и то же отношение *инцидентности* точки A и прямой p может быть записано с использованием одного и того же символа: либо как $A \in p$, либо в виде $p \in A$. Есть у этого соглашения и другие достоинства.

Теорема 1.5.2. *Если на множестве A задано отношение эквивалентности R, то существует такое разбиение множества A на классы, что xRy тогда и только тогда, когда x и y принадлежат одному классу. Обратно, если задано разбиение множества A на классы A_c, то отношение «x и y принадлежат одному и тому же классу» рефлексивно, симметрично и транзитивно.*

Доказательство. I. Обозначим $A_x = \{y \mid y \in A \wedge xRy\}$. Рассмотрим множество $M = \{A_x \mid x \in A\}$. Заранее очевидно, что за исключением разбиения, представленного примером 1, запас имен превышает потребности.

Выполнение условия 1 очевидно.

Доказываем, что выполнено условие 2. Пусть $x \in A$. Тогда $x \in A_x$ и $x \in \bigcup_{y\in A} A_y$, откуда $A \subseteq \bigcup_{y\in A} A_y$. Обратное включение $\bigcup_{y\in A} A_y \subseteq A$ следует из условия 1.

Убедимся, что и условие 3 выполняется. Пусть $x \in A$ и $y \in A$. Тогда либо $A_x = A_y$, либо $A_x \cap A_y = \emptyset$.

Действительно, если $y \in A_x$, то xRy. По симметричности отношения R верно и yRx. Пусть $z \in A_y$, т. е. yRz, тогда xRz по транзитивности R и $z \in A_x$. Пусть $z \in A_x$, т. е. xRz, тогда имеет место yRz, т. е. $z \in A_y$. Следовательно $A_y = A_x$.

Если же $y \notin A_x$, то $x\overline{R}y$. Пусть найдется такой элемент $z \in A$, что $z \in A_x \cap A_y$. Тогда $z \in A_x$, т. е. xRz, и $z \in A_y$, отсюда yRz и zRy, так что xRy, что невозможно. Следовательно такого z не существует, т. е. $A_x \cap A_y = \emptyset$. Осталось из многочисленных имен, первоначально данных классам по любому из содержащихся в классе элементов, выбрать одно, как это было сделано выше в примере 2, или назначить новые имена.

II. Пусть $\{A_c\}$ — разбиение множества A на классы. Определим отношение R на множестве A так: $xRy \equiv \exists c\,(x \in A_c \wedge y \in A_c)$.

Убедимся, что отношение R рефлексивно. Пусть $x \in A$, тогда для некоторого c выполнено $x \in A_c$, откуда вытекает xRx.

Симметричность отношения R очевидна.

Покажем, что R транзитивно. Пусть xRy, yRz и $y \in A_c$. Но тогда и $x \in A_c$, а также $z \in A_c$, т.е. xRz. $\lhd$

Равенство обладает всеми свойствами эквивалентности (теорема 1.4.18). Довольно часто все элементы одного класса эквивалентности могут рассматриваться как тождественные или равные, если в рамках того или иного исследования можно отвлечься от некоторых их свойств. Такой вид эквивалентности называют *изоморфизмом*.

1.5.4. Графы и деревья

В этом разделе даются лишь основные определения (в основном взятые из [18]) и некоторые простейшие результаты содержательной математической дисциплины — теории графов.

Понятие бинарного отношения допускает геометрическую интерпретацию, охотно используемую в программировании из-за ее наглядности. Пусть V — множество, обычно конечное, A — заданное на этом множестве бинарное отношение.

Будем называть элементы v множества V *вершинами*, а пары $\langle u, v\rangle$ — *дугами* (vertex — вершина, arc — дуга (англ.)) графа $G = (V, A)$, представляющего отношение A на множестве V. Принято изображать вершины графа точками (реально — маленькими кружками) на плоскости, а дугу $a = \langle u, v\rangle$ — стрелкой, иногда изогнутой, ведущей от точки u к точке v. Про дугу a в этом случае говорят, что она *исходит* из вершины u и *заходит* в вершину v, а вершины называют соответственно *началом* и *концом* дуги (иногда просто ее *концами*) и *смежными* друг другу. Граф *полон*, если все его вершины смежны.

В графическом представлении нетрудно было бы начертить две или более дуг с одинаковыми началом и концом. Но поскольку A — это множество и по определению не может содержать повторяющихся элементов, наличие подобных (*кратных*) дуг не допускается.

Про дугу и любой из ее концов говорят, что они *инцидентны*. На множестве $V \cup A$ отношение инцидентности симметрично: слова «вершина v инцидентна дуге a» и «дуга a инцидентна вершине v» означают одно и то же.

Если для дуги $a = \langle u, v\rangle$ в графе присутствует *встречная* дуга $\langle v, u\rangle$, то понятие направленности дуги a теряет смысл. Пару встречных дуг называют *ребром*, а на схеме изображают ее отрезком, не слишком изломанной ломаной или не слишком искривленной кривой. Дуга $\langle v, v\rangle$ неотличима от встречной ей дуги, т. е. автоматически оказывается ребром. Такое ребро называется *петлей* (при вершине v). Граф антирефлексивного отношения — это граф *без петель*.

Из сказанного следует, что граф асимметричного отношения не содержит ребер, такой граф будем называть *ориентированным*. В графе симметричного отношения любые две вершины могут быть соединены только ребром, и граф называется *неориентированным*. В общем случае произвольного бинарного отношения в графе могут присутствовать как дуги (без встречных), так и ребра — замаскированные пары встречных дуг. Такой граф тоже можно было бы считать ориентированным. Понятия смежности и инцидентности естественным образом распространяются на вершины и ребра неориентированного графа.

Степенью $d(v)$ вершины v называется число инцидентных ей дуг, а для неориентированного графа — ребер. Для вершины v ориентированного графа *полустепень исхода* $d_o(v)$ (*захода* $d_i(v)$) — это число исходящих из нее (заходящих в нее) дуг. Таким образом, для числа m_e ребер неориентированного графа и числа m_a дуг ориентированного графа имеют место зависимости:

$$m_e = 1/2 \sum_{v \in V} d(v), \tag{8}$$

$$m_a = \sum_{v \in V} d_o(v) = \sum_{v \in V} d_i(v) \tag{9}$$

(при таком суммировании каждое ребро засчитывается дважды, а каждая дуга в любой из сумм — лишь один раз).

При $d(v) = 1$ вершина v называется *висячей*, как и единственное инцидентное ей ребро.

В графе $G = (V, A)$ последовательность вершин $v_1, \ldots v_k$, в которой для $1 \leqslant i < k$ одна из пар $\langle v_i, v_{i+1}\rangle$ или $\langle v_{i+1}, v_i\rangle$ (или обе) принадлежит A, т. е. представляет собой дугу, вместе с этими дугами называется *цепью* с *концами* v_1 и v_k и *длиной* k. Если все дуги цепи имеют вид $\langle v_i, v_{i+1}\rangle$, то цепь называется *путем* с *начальной* вершиной v_1 и *конечной* вершиной v_k, все прочие вершины v_i называются *внутренними*. Замкнутая цепь, концы которой — это одна и та же вершина, называется *циклом*, а замкнутый путь — *контуром*. Цепь (путь, цикл, контур) называется *простой* (простым), если в ней (в нем) нет повторяющихся вершин. *Бесконтурный* ориентированный граф (не содержащий контуров, но, возможно, содержащий циклы) обычно называется *ациклическим*.

Граф $G' = (V', A')$ называется *частью* графа $G = (V, A)$ или его *подграфом* в слабом смысле, если $V' \subseteq V$, $E' \subseteq E$. Если, кроме того, потребовать, чтобы для любых двух вершин $u, v \in V'$, таких, что $\langle u, v\rangle \in A$, эта пара была бы дугой в G' ($\langle u, v\rangle \in A'$), то G' называется подграфом графа G в сильном смысле. Далее термин «подграф» будет использоваться именно в этом смысле. *Суграф* графа G — это такая его часть G', для которой $V' = V$.

Вершина v ориентированного графа *достижима* из вершины u, если существует путь из u в v. Вершины u и v *бисвязанны*, если они взаимно достижимы. В ориентированном графе вершина называется *источником*, если из нее достижимы все остальные вершины, и *стоком*, если она достижима из любой вершины.

Ориентированный граф называется *слабо связным*, если для любых двух его вершин существует соединяющая их цепь, и *бисвязным*, если любые две его вершины взаимно достижимы. Неориентированный граф *связен*, если любые две его вершины соединены цепью.

Подграф ориентированного графа G называется *бикомпонентой* или *компонентой сильной связности*, если он бисвязен и не является подграфом никакого другого бисвязного подграфа графа G (т. е. бикомпонента — это максимальный бисвязный подграф).

Граф (ориентированный) G' называется фактор-графом бикомпонент графа G (его *графом Герца*), если вершины G' — это бикомпоненты G и две различные вершины G' соединены дугой в том и только в том случае, если в G существует дуга, ведущая от вершины первой компоненты к вершине второй.

Лемма 1.5.3. *Фактор-граф G' бикомпонент любого ориентированного графа G не содержит контуров.*

Доказательство. Контуров длины 1 граф G' не содержит по построению. Предположим, что в G' существует контур длины не меньше 2, содержащий вершины A и B. Пусть a — любая вершина графа A, b — любая вершина B. Тогда в G существует путь из a в b и путь из b в a, следовательно, объединение графов A и B — это бисвязный подграф графа G. Это противоречит тому, что A — бикомпонента. $\lhd$

Следствие 1.5.4. *Если в ориентированном графе некоторый путь начинается и оканчивается в одной и той же бикомпоненте, то он целиком лежит внутри нее.*

Доказательство. Действительно, каждому пути в графе G можно сопоставить путь в фактор-графе G', выделяя те дуги, которые ведут из одной бикомпоненты в другую и строя путь из соответствующих дуг графа G'. Если бы путь, о котором идет речь, проходил через несколько бикомпонент, то ему был бы сопоставлен контур в графе G', что невозможно. $\triangleleft$

Неориентированный граф без циклов (а значит, и без петель) называется *лесом*, а если он еще и связен, то *деревом*.

Лемма 1.5.5. *Для любых двух вершин u и v дерева существует только одна соединяющая их цепь.*

Доказательство. Одна такая цепь обязана существовать в связном графе, а две цепи образовывали бы цикл. $\triangleleft$

Лемма 1.5.6. *При исключении из дерева T любого ребра $e = (u, v)$ граф перестает быть связным, а вершины u и v попадают в разные компоненты связности.*

Доказательство. Любая цепь, связывающая вершины u и v в получившемся графе, вместе с ребром e образовывала бы цикл в графе T. $\triangleleft$

Если дерево T не содержит ребра $e = (u, v)$, то такое ребро называется *хордой*.

Лемма 1.5.7. *Добавление хорды $e = (u, v)$ к дереву T порождает граф G с одним циклом.*

Доказательство. Хорда e замыкает в цикл цепь, связывающую вершины u и v в графе T. По лемме 1.5.5 других циклов возникнуть не может. $\triangleleft$

Лемма 1.5.8. *Число m ребер в дереве T на 1 меньше числа n его вершин*:

$$m = n - 1 \tag{10}$$

Доказательство (индукция). Для *вырожденного* дерева, состоящего всего из одной вершины, это очевидно, как и то, что добавление новой вершины и ребра, связывающего ее с уже имеющимся деревом, не нарушает соотношения (10). $\triangleleft$

Любая вершина дерева может быть объявлена его *корнем*. После этого дерево становится *корневым*, а все висячие вершины, отличные от корня, — *листьями* этого дерева. *Уровень* вершины — это расстояние (длина цепи) от корня до этой вершины, *высота дерева* — наибольший из уровней его листьев.

Под *ориентированным* деревом обычно понимается дерево одного из двух типов: *выходящее* — с источником в корне (с ориентацией дуг от корня) или *входящее* со стоком в корне (с противоположной ориентацией дуг). В выходящем дереве T вершина u называется *предком* вершины v (ее *потомка*), если в T имеется путь от u к v. Если длина такого пути равна 1, то предок и потомок — это *отец* и *сын*. Сыновья одного и того же отца называются *братьями*. Вершина u выходящего дерева T вместе со всеми своими потомками и дугами, попарно всех их соединяющими, образует *поддерево* дерева T с корнем в u.

Если все множества братьев в дереве упорядочены, то дерево становится *упорядоченным*. Если при этом у каждой вершины имеется не более двух сыновей, то дерево называется *бинарным*.

Разбиением на участки пути (контура) $u_1\ u_2\ \ldots u_n$ называется последовательность путей $(u_1\ \ldots u_\alpha,\ u_\alpha\ \ldots u_\beta,\ \ldots,\ u_\psi\ \ldots u_n)$, $1 \leqslant \alpha \leqslant \beta \leqslant \ldots \leqslant n$ (как правило, для любого участка $u_\mu\ \ldots\ u_\nu$ его длина $\nu - \mu$ больше 0, случай $\mu = \nu$ соответствует участку нулевой длины, состоящему из одной вершины).

Путь p может быть *собран* из путей $p_1, \ldots, p_m$, если эти пути можно разбить на участки так, что, объединяя и переставляя после объединения элементы этих разбиений, можно получить разбиение пути p.

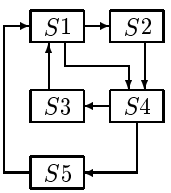


Рис. 1

Например, путь p = 5 1 2 4 3 1 4 3 1 2 4 5 1 в графе, показанном на рисунке, может быть собран из пути p_1 = 5 1 4 5 1 и дважды взятого контура p_2 = 2 4 3 1 2. Действительно, для пути p_1 имеется разбиение (5 1, 1 4, 4 5 1), а для контура p_2 — разбиения (2 4 3 1, 1 2) и (2 4, 4 3 1 2). Если объединить все эти три разбиения и переставить элементы в полученном объединении, то это даст разбиение (5 1, 1 2, 2 4 3 1, 1 4, 4 3 1 2, 2 4, 4 5 1) рассматриваемого пути p.

Л е м м а 1.5.9. *Любой путь может быть собран из простого пути с тем же началом и концом и нескольких простых контуров.*

Доказательство (индукция по длине пути). Путь длины 1 — это либо простой путь, либо простой контур (если он замкнут). Пусть утверждение леммы верно для всех путей длины меньше n. Возьмем путь $p = u_0\ u_1\ \ldots u_n$ длины n. Если он не простой (иначе доказывать нечего), то пусть u_μ — наименьший из повторяющихся номеров вершин. Тогда путь p имеет вид $u_0\ \ldots u_\mu\ \ldots u_\nu\ \ldots u_n$, где $u_\nu = u_\mu$ и $0 \leqslant \mu < \nu \leqslant n$. Его можно собрать из пути $p' = u_0\ \ldots u_\mu\ u_{\nu+1}\ \ldots u_n$ (при $\nu = n$ участок $u_{\nu+1}\ \ldots u_n$ отсутствует) с теми же началом u_0 и концом u_n, что и у пути p, и контура $p'' = u_\mu\ \ldots u_\nu$. По предположению индукции путь p' можно собрать из простого пути p_1 с началом u_0 и концом u_n и нескольких простых контуров, а контур p'' — из контура с началом и концом $u_\mu = u_\nu$ и еще нескольких простых контуров. Но тогда и путь p можно собрать из простого пути p_1 с началом u_0 и концом u_n и нескольких простых контуров. $\lhd$

В каждом ориентированном графе множество простых контуров конечно, поскольку длины их ограничены числом вершин графа. Пусть c — один из этих контуров. В графе, имеющем как источник, так и сток, можно найти простой путь p_1 от источника до какой-нибудь вершины v контура c и простой путь p_2 от вершины v до стока, а затем собрать путь p из простого пути $p_1 p_2$ и простого контура c. Ясно, что путь p не может быть собран из простого пути и простых контуров никаким другим способом (с точностью до места входа в контур c).

Упражнение 1.5.4. Доказать это утверждение.

Иными словами, база контуров для сборки путей должна совпадать с множеством всех простых контуров рассматриваемого ориентированного графа.

Каркасом (остовным деревом) связного графа G называется дерево, в котором сохранены все вершины G. Для ориентированного графа с источником u представляет интерес выходящий каркас с корнем в u, а для графа со стоком v — входящий каркас с корнем v.

Сечением ациклического ориентированного графа с источником u и стоком v называется такое множество его вершин, что каждый путь из u в v проходит через одну из вершин этого множества, но ни одно из его подмножеств уже не обладает этим свойством.

Если множество вершин неориентированного графа разбито на k непустых непересекающихся подмножеств так, чтобы никакие две смежные вершины не попали в одно подмножество, то такое разбиение называется *правильной раскраской* (вершин) графа, вершины, попавшие в одно подмножество, — *одноцветными*, а сам граф — *k-раскрашиваемым* или *k-дольным*. Например, граф отношения инцидентности на множестве $V \cup A$ двудолен, его доли — это V и A.

1.5.5. Соответствия и отображения

Отношение (R, A, B) часто называют *соответствием* с *областью отправления* A и *областью прибытия* B или соответствием на $A \times B$. При $B = A$ используют термин отношение или соответствие *на* A. Желая подчеркнуть, что отношение рассматривается как соответствие, букву R заменяют на F или иную удобную (по мнемонике) букву, вместо xFy пишут $y = F(x)$, вместо $F \in A \times B$ — $F : A \to B$, и называют «$A \to B$» *типом* соответствия.

Пусть отношения (F, A, B) и (G, B, C) трактуются как соответствия, и $(H, A, C) = (FG, A, C)$ — их произведение (в указанном порядке). Тогда, для $x \in A$, $z \in C$ зависимость $z = H(x)$ в согласии с определением произведения означает, что для некоторого $Y \in B$ имеет место $xFy \wedge yGz$ или, в записи, принятой для соответствий, $y = F(x) \wedge z = G(y)$, короче: $z = G(F(x))$. Порядок появления букв изменился по сравнению с записью, принятой для отношений. Соответствие H, называемое *композицией* соответствий F и G, обозначается с учетом этого порядка так: $H = G \circ F$, так что $z = (G \circ F)(x)$. Пишут также $z = G \circ F(x)$, если это не приводит к недоразумению.

Для соответствия $F : A \to B$ и любого $x \in A$ множество $\hat{F}(x) = \{y \mid y \in B \wedge xFy\}$ называется *полным образом* элемента x, а для любого $y \in B$ множество $\check{F}(y) = \{x \mid x \in A \wedge xFy\}$ — это *полный прообраз* элемента y при том же соответствии. Эти понятия могут быть перенесены на подмножества $X \subseteq A$ и $Y \subseteq B$ множеств A и B. Полный образ X — это множество $\hat{F}(X) = \bigcup_{x \in X} \hat{F}(x)$, а полный прообраз Y — множество $\check{F}(Y) = \bigcup_{y \in Y} \check{F}(y)$.

Подмножество $\mathrm{Dom}(F)$ области отправления соответствия F

$$\mathrm{Dom}(F) = \{x \mid x \in A \wedge \exists y (y \in B \wedge xFy)\},$$

состоящее из тех элементов множества A, для которых в B содержится хотя бы

один соответствующий им элемент, называется *областью определения* соответствия F, а подмножество $\mathrm{Rng}(F)$ области прибытия соответствия F

$$\mathrm{Rng}(F) = \{y \mid y \in B \wedge \exists x(x \in A \wedge xFy)\},$$

состоящее из тех элементов множества B, которые соответствуют хотя бы одному элементу из A, называется *областью значений* F.

Если $\mathrm{Dom}(F) = A$, то говорят, что соответствие F *всюду определено*. Если $\mathrm{Rng}(F) = B$, то говорят, что соответствие F *сюръективно*. Пусть x и x_1 — произвольные элементы из A, y и y_1 — из B. Тогда соответствие F называется *однозначным*, если $y = F(x) \wedge y_1 = F(x) \supset y = y_1$. Оно называется *инъективным*, если $y = F(x) \wedge y = F(x_1) \supset x = x_1$.

Из попарного сходства последних определений следует, что сюръективность соответствия F равносильна определенности F^{-1} всюду, а инъективность первого — однозначности второго.

Упражнение 1.5.5.
а. Для соответствия $F : A \to B$ доказать, что $\mathrm{Rng}(F) = \hat{F}(A)$, $\mathrm{Dom}(F) = \check{F}(B)$.
б. Доказать, что соответствие $F : A \to B$ сюръективно, если $\hat{F}(A) = B$.
в. Доказать, что соответствие $F : A \to B$ однозначно, если верно, что:
$\forall y \forall y_1\, (y \in B \wedge y_1 \in B \wedge y \neq y_1 \supset \check{F}(y) \cap \check{F}(y_1) = \emptyset)$.

Соответствие, одновременно сюръективное и инъективное, называется *биективным* или *взаимно однозначным*. Часто два множества называют *эквивалентными*, если между ними можно установить биективное соответствие. Эквивалентные конечные множества — это множества с одинаковым числом элементов. Множество, эквивалентное множеству $\mathbf{N}$ всех натуральных чисел, иными словами, такое, что все его элементы можно пронумеровать, называется *счетным*. Про множество, конечное или счетное в зависимости от контекста, говорят, что оно *не более чем счетно*.

Пусть $F : A \to B$ — произвольное соответствие, X — подмножество области определения соответствия F. *Ограничением* соответствия F *областью* (множеством) X называется множество $F_X = F \cap (X \times B) = \{(x, F(x)) \mid x \in X\}$, т. е. в F_X отбираются все те пары $(x, y) \in F$, для которых x принадлежит области отправления X соответствия F_X.

Однозначное всюду определенное соответствие называется *отображением* или *функцией*, а однозначное, но не обязательно всюду определенное — *частичным* отображением или *частичной* функцией.

Теорема 1.5.10. *Если отображение $f : A \to B$ инъективно, X и Y — подмножества A, то справедливы следующие утверждения*:

а. *Ограничение f_X отображения f множеством X — биективное отображение X на $\hat{f}(X)$, его обращение производится по правилу $(f_X)^{-1} = (f^{-1})_{\hat{f}(X)}$*;

б. *Из $X \subset Y$ следует $\hat{f}(X) \subset \hat{f}(Y)$*;

в. *Полные образы результатов основных теоретико-множественных операций над подмножествами множества A являются результатами тех же опе-*

раций над полными образами операндов: $\hat{f}(X \cup X_1) = \hat{f}(X) \cup \hat{f}(X_1)$, $\hat{f}(X \cap X_1) = \hat{f}(X) \cap \hat{f}(X_1)$, $\hat{f}(X \backslash X_1) = \hat{f}(X) \backslash \hat{f}(X_1)$, $\hat{f}(X \triangle X_1) = \hat{f}(X) \triangle \hat{f}(X_1)$;

г. *Если элементы множества* M *являются подмножествами множества* A, *то* $\hat{f}(\bigcup M) = \bigcup(\hat{f}(M))$ *и* $\hat{f}(\bigcap M) = \bigcap(\hat{f}(M))$.

Доказательство. а. Если отображение f определено всюду на A, то f_X определено всюду на X, при этом $\forall x\,(x \in X \supset \exists y\,(y \in \hat{f}(X) \wedge y = f(x)))$ и $\forall y\,(y \in \hat{f}(X) \supset \exists x\,(x \in X \wedge y = f(x)))$. Отсюда $\mathrm{Dom}(f_X^{-1}) = \mathrm{Rng}(f_X) = \hat{f}(X)$ и $\mathrm{Rng}(f_X^{-1}) = \mathrm{Dom}(f_X) = X$ (см. упражнение 1.5.5.а).

б. Следует прямо из определений.

в. Для каждого $x \in X$ при $X \subseteq A$ найдется такой $y \in Y$, что $y = f(x)$, причем по инъективности f никакого $x' \neq x$, такого что $y = f(x')$, быть не может. Таким образом, $x \in X \equiv f(x) \in \hat{f}(X)$ и $x_1 \in X_1 \equiv f(x_1) \in \hat{f}(X_1)$. Остальное непосредственно следует из определений операций.

г. Пусть $X \in M$. Тогда рассуждения пункта в. переносятся и на данный пункт. $\lhd$

Конечной последовательностью (элементов из A) называется отображение $a : \{1, 2, \ldots, n\} \to A$ (нумерацию часто начинают не с 1, а с 0). *Счетной последовательностью* или просто *последовательностью* называют отображение множества $\mathbf{N}$ всех натуральных чисел в A. Вместо $x = a(i)$ обычно пишут $x = a_i$. Последовательность k натуральных чисел называется *монотонно неубывающей*, если $i < j \supset k_i \leqslant k_j$, *монотонно возрастающей* — если $i < j \supset k_i < k_j$, *монотонно невозрастающей* — если $i < j \supset k_i \geqslant k_j$, *монотонно убывающей* — если $i < j \supset k_i > k_j$. Общее название всех таких последовательностей — *монотонные*. *Подпоследовательностью* последовательности a называется композиция $b = a \circ k$ последовательности a с монотонно возрастающей последовательностью k индексов отобранных из a членов.

Упражнение 1.5.6. Проверить, что если $b = a \circ k$, то $x = b_i \equiv x = a_{k_i}$.

Упражнение 1.5.7. Выписать таблицу значений всех отображений из множества $\{1, 2\}$ в множество $\{1, 2, 3\}$.

Решение этого упражнения, как и ряд таблиц из разд. 1.3, убеждает, что число различных отображений m-элементного множества в n-элементное равно n^m. По аналогии принято обозначать множество всех отображений множества X в множество Y (каждое может быть бесконечным) через Y^X.

Пусть Y — это двухэлементное множество $\{0, 1\}$. Отождествляя его с числом 2 (резон для такого отождествления дан ниже, в разд. 1.5.7 в связи с аксиомой (1.5.7–14)), принято обозначать множество всех отображений X в Y через 2^X. Элементы b этого множества — отображения X в $\{0, 1\}$ — находятся во взаимнооднозначном соответствии с элементами S множества $\mathrm{Powerset}(X)$ всех подмножеств множества X. Действительно, можно считать, что множество S состоит из элементов $x \in X$, для которых $b(x) = 1$ (т. е. $S = \{x \in X \mid b(x) = 1\}$). Отображение b называется *характеристической функцией* подмножества S. Обозначение $\mathrm{Powerset}(X)$ (сокращенно $\mathrm{Pow}(X)$) принято, видимо, по ассоциации с 2^X.

1.5.6. Отношения порядка

Отношение R на множестве A называется отношением *строгого порядка* (или, короче, — строгим порядком), если оно транзитивно и антирефлексивно. Отношение R на множестве A называется отношением *нестрогого порядка* (нестрогим порядком), если оно транзитивно, рефлексивно и антисимметрично.

Если на множестве A введено отношение порядка R (безразлично какое), то множество A называется *упорядоченным* по этому отношению (или R-упорядоченным).

Если в упорядоченном множестве A элементы x и y связаны отношением R, т. е. имеет место xRy, то говорят, что x *предшествует* (R-*предшествует*, если необходима явная ссылка на R) элементу y, а y *следует* за x. Если $x \in A$, $y \in A$, xRy и не существует элемента $z \in A$, такого что xRz и zRy (*лежащего между* x и y), то говорят, что x *непосредственно предшествует* y, а y *непосредственно следует* за x.

Теорема 1.5.11. *Любое отношение порядка (строгого или нестрогого) транзитивно и антисимметрично. Транзитивное и антисимметричное отношение является строгим порядком, если оно антирефлексивно, и нестрогим порядком, если оно рефлексивно.*

Доказательство. Почти все утверждения этой теоремы прямо следуют из определений. В дополнительном обосновании нуждается лишь утверждение, что отношение строгого порядка антисимметрично. Пусть R — строгий порядок, и для некоторых x и y имеет место xRy. Предположим, что выполнено также yRx. Тогда xRx (по транзитивности), но это противоречит антирефлексивности отношения R. Итак, из xRy следует $y\overline{R}x$, т. е. отношение R асимметрично, а отсюда (см. упражнение 1.5.3) вытекает, что оно антисимметрично. $\lhd$

Как показывает следующая теорема, транзитивное и антисимметричное отношение легко может быть преобразовано как в строгий, так и в нестрогий порядок.

Теорема 1.5.12. *Пусть R — транзитивное и антисимметричное отношение на множестве A. Тогда $R\backslash I_A$ — строгий порядок, а $R \cup I_A$ — нестрогий порядок.*

Доказательство. Проверим выполнение требуемых свойств отношений $R\backslash I_A$ и $R \cup I_A$.

1. Отношение $R\backslash I_A$ транзитивно. Пусть x, y, z таковы, что имеет место $x(R\backslash I_A)y \wedge y(R\backslash I_A)z$, подробнее — $xRy \wedge \neg(xI_Ay) \wedge yRz \wedge \neg(yI_Az)$ (по определению разности отношений). Требуется показать, что $x(R\backslash I_A)z$. Имеем: xRz (по транзитивности R) и $x \neq y$ (по определению I_A). Предположим, что xI_Az, т. е. что $x = z$. Тогда yRz преобразуется в yRx, а из xRy и yRx по антисимметричности R следует $x = y$, что противоречит доказанному. Следовательно, $\neg(xI_Az)$ и вместе с xRz это приводит к цели.

2. Отношение $R\backslash I_A$ антирефлексивно. Допустим, что для некоторого x имеет место $x(R\backslash I_A)x$, т. е. $xRx \wedge \neg(xI_Ax)$, откуда $x \neq x$, чего быть не может. Это и доказывает антирефлексивность отношения $R\backslash I_A$.

Из пунктов 1 и 2 следует, что отношение $R\backslash I_A$ — строгий порядок.

3. Отношение $R \cup I_A$ транзитивно. Пусть для некоторых x, y, z выполнено $x(R \cup I_A)y \wedge y(R \cup I_A)z$. Отсюда $(xRy \vee x = y) \wedge (yRz \vee y = z)$ (по определению объединения отношений и определению I_A). Далее:

$$(xRy \wedge yRz) \vee (xRy \wedge y = z) \vee (x = y \wedge yRz) \vee (x = y \wedge y = z)$$

(по дистрибутивности), $xRz \vee xRz \vee xRz \vee x = z$ (первый член — по транзитивности R, во втором, третьем и четвертом членах замещаем y равным ему значением), $x(R \cup I_A)z$ (по идемпотентности дизъюнкции и по определению объединения отношений).

4. Отношение $R \cup I_A$ рефлексивно, т. е. $x(R \cup I_A)x$. Следует из рефлексивности отношений R и I_A и определения объединения отношений.

5. Отношение $R \cup I_A$ антисимметрично. Пусть $x(R \cup I_A)y \wedge y(R \cup I_A)x$. Так же, как в пункте 3, получаем

$$(xRy \wedge yRx) \vee (xRy \wedge y = x) \vee (x = y \wedge yRx) \vee (x = y \wedge y = x).$$

Из каждого члена этой дизъюнкции следует равенство $x = y$ (из первого — по антисимметричности отношения R, из остальных — очевидным образом).

Из пунктов 3–5 следует, что отношение $R \cup I_A$ — нестрогий порядок. $\lhd$

Отношение порядка R называется *линейным* (строгим или нестрогим) порядком, если оно связно. Если хотят подчеркнуть, что порядок R — не обязательно линейный, то говорят, что R — *частичный* (строгий или нестрогий) порядок.

Примеры. Отношения $<$, $>$ (на любом числовом множестве), $\subset$ (на множестве всех подмножеств некоторого множества M) — это отношения строгого порядка. Оба требуемые свойства — транзитивность и антирефлексивность — легко проверяются. Отношения $\leqslant$, $\geqslant$ и $\subseteq$ — нестрогие порядки. Это легко установить как непосредственно, так и с помощью теоремы 1.5.12, зная, что $<$, $>$ и $\subset$ — отношения строгого порядка. При этом любой из порядков $<$, $>$, $\leqslant$, $\geqslant$ — линейный, а порядки $\subset$ и $\subseteq$ не линейны, если множество M содержит не менее двух элементов. Действительно, если $a \in M$, $b \in M$ и $a \neq b$, то ни одно из множеств $\{a\}$ и $\{b\}$ не является подмножеством другого.

Пусть множество M упорядочено по отношению R, A — подмножество M. Элемент $x \in M$ называется *нижней R-гранью* подмножества A, если справедлива формула $\forall y\,(y \in A \wedge y \neq x \supset xRy)$. Аналогично определяется верхняя R-грань. Элемент $x \in A$ называется *R-наименьшим* в A, если x — нижняя R-грань подмножества A.

Примеры. Для подмножеств $A = \{x \mid x > 0\}$ и $B = \{x \mid x \geqslant 0\}$ множества $\mathbf{R}$ всех вещественных чисел как число -1, так и число 0 является нижней $<$-гранью. В множестве A нет $<$-наименьшего элемента, а в множестве B $<$-наименьший элемент — это 0.

Упражнение 1.5.8. Доказать, что в множестве не может быть более одного наименьшего элемента.

Элемент $x \in A$ называется R-*минимальным* в множестве A, если верно, что $\forall y\,(y \in A \wedge y \neq x \supset y\overline{R}x)$. В любом множестве A R-наименьший элемент, если он существует, является R-минимальным. Действительно, пусть x — R-наименьший элемент в A, тогда из $y \in A \wedge y \neq x$ следует xRy, но при этом yRx невозможно, иначе в силу антисимметричности R было бы $x = y$.

Если R — линейный порядок, то верно и обратное утверждение. Действительно, пусть x — R-минимальный элемент множества A, $y \in A \wedge y \neq x$. Из связности отношения R следует, что $xRy \vee yRx$ истинно. Но yRx ложно в силу R-минимальности x. Следовательно, xRy истинно, а это и значит, что x — R-наименьший элемент A.

Упражнение 1.5.9. Может ли множество содержать более одного R-минимального элемента?

Множество A, упорядоченное отношением R, называется *вполне упорядоченным* по этому отношению (или вполне R-упорядоченным), если любое его непустое подмножество A имеет R-наименьший элемент.

Например, множество всех натуральных чисел вполне $<$-упорядочено, но не вполне $>$-упорядочено. Множества всех целых, рациональных или вещественных чисел не вполне $<$-упорядочены, так как ни одно из этих множеств не содержит наименьшего в обычном смысле (т. е. $<$-наименьшего) элемента. Нетрудно построить вполне упорядочивающее отношение R для множества всех целых или всех рациональных чисел. Для множества целых чисел определим это отношение следующим образом: $xRy \equiv |x| < |y| \vee (|x| = |y| \wedge x > y)$. Тогда в любом непустом множестве целых чисел R-наименьшим будет число, наименьшее по абсолютной величине, а если их два, то выбирается положительное. Иначе говоря, целые числа располагаются в такую последовательность

$$0,\ 1,\ -1,\ 2,\ -2,\ \ldots,\ n,\ -n,\ \ldots$$

и R-наименьшим в любом множестве целых чисел является число, имеющее наименьший номер в этой последовательности.

Два вполне упорядоченных множества A и A' с отношениями порядка R и R' соответственно называются *подобными*, если существует биективное отображение $F : A \to A'$, *сохраняющее порядок*, — такое, что

$$\forall x \forall y\,(x \in A \wedge y \in A \wedge xRy \supset F(x)R'F(y)).$$

Упражнение 1.5.10. Доказать, что:
а. Отношение R, вполне упорядочивающее множество A, связно.
б. Отношение подобия между вполне упорядоченными множествами — это отношение эквивалентности (разд. 1.5.3).

1.5.7. О парадоксах теории множеств

В теории множеств Г. Кантора (прозванной «наивной ») считается, что любое множество, о котором у человека сложилось достаточно четкое, хотя бы интуитивное, представление, может быть определено словами. Уже упоминалось,

что такой взгляд приводит к противоречиям, иначе называемым антиномиями или парадоксами (в той мере, в какой противоречие удается устранить) теории множеств.

Вот простейшие примеры.

Парадокс Рассела. Рассуждения в этом парадоксе проводятся по схеме, называемой *диагональным методом*. Если желательно построить объект, отличающийся по какому-либо признаку от объектов, представленных строками конечной или бесконечной таблицы, то этого можно достичь, собирая его из элементов, полученных изменением диагональных элементов этой таблицы: i-й элемент нового объекта отличается по этому признаку от i-го (в упрощенном варианте — от любого) элемента i-й строки таблицы, а потому полученный объект не может совпасть ни с одним из объектов, представленных в таблице.

Пусть X — множество всех «хороших» множеств, не содержащих себя в качестве элемента: $X = \{x \mid x \notin x\}$. Спросим, множество X — «хорошее» или нет. Предположим, что $X \in X$, тогда по определению X должно быть $X \notin X$. Это противоречие заставляет заключить, что $X \notin X$, но тогда, опять же по определению X, соотношение $X \notin X$ должно быть ложно. Дальше в рамках любой логики, претендующей на непротиворечивость, деваться уже некуда. В данном случае применен как раз упрощенный вариант диагонального метода: в одну «строку» собраны все «хорошие» множества, в другую — все остальные. Для множества X не находится места ни в одной строке.

Сам же Рассел предложил популярный вариант этого парадокса, известный как парадокс деревенского парикмахера. Этот парикмахер объявил, что он будет брить всех тех и только тех жителей деревни, которые не бреются сами. Легко понять, что это правило запрещает ему как бриться, так и не бриться самому. На самом деле положение не безвыходно: условие будет соблюдено, если он живет в другой деревне или же носит бороду и в бритье не нуждается.

Но есть и более интересный подход к разрешению этого парадокса. Человек многолик. В быту он ест, спит, развлекается, по утрам — умывается, чистит зубы, бреется и собирается на работу. На работе, если он — парикмахер, он принимает клиентов, стрижет их, бреет, моет им головы и умащает благовониями. Парикмахер нарушил бы свой обет, если бы случилось, что он явился сам к себе в качестве клиента.

Можно возразить, что все сказанное — игра смысловыми понятиями, почерпнутыми из человеческого быта и отраженными в естественном языке. Но и Кантор охарактеризовал понятия множества и принадлежности с помощью слов: «собрание», «интуиция», «интеллект», «целое (единое)», отличающихся от бытовых разве что чуть более высоким стилем. Другого способа просто нет. Укладывается ли в эту систему понятий возможность для множества оказаться своим собственным элементом? Может быть, это ничуть не лучше, чем принять самого себя в парикмахерской в качестве клиента?

Прежде чем перейти к другому парадоксу, напомним понятие мощности множества. Сказать, что такое мощность множества, трудно, за исключением, пожалуй, случая конечных множеств (см. пример с девочками и яблоками в

самом начале книги), где мощность множества — это число элементов в нем. Проще говорить о сравнении множеств по мощности, хотя и здесь без затруднений не обойтись.

Говорят, что мощность множества A не превосходит мощности множества B, и пишут $\overline{\overline{A}} \leqslant \overline{\overline{B}}$, если существует инъективное отображение $F : A \to B$. Мощности множеств A и B равны (обозначение $\overline{\overline{A}} = \overline{\overline{B}}$), если существует биективное отображение $F : A \to B$. Таким образом, эквивалентность множеств, определенная в разд. 1.5.5, — это эквивалентность по мощности. Мощность множества A меньше мощности множества B ($\overline{\overline{A}} < \overline{\overline{B}}$), если $\overline{\overline{A}} \leqslant \overline{\overline{B}}$ и неверно, что $\overline{\overline{A}} = \overline{\overline{B}}$, т. е. не существует биективного отображения A на B.

Интуитивно ясно, что для конечного множества A обозначение $\overline{\overline{A}}$ можно трактовать как число элементов A, и что при такой трактовке формулы $\overline{\overline{A}} \leqslant \overline{\overline{B}}$, $\overline{\overline{A}} = \overline{\overline{B}}$ и $\overline{\overline{A}} < \overline{\overline{B}}$ соответствуют словесным определениям отношений между мощностями. Но для конечных множеств существуют и другие соотношения между мощностями, которые не выполняются для бесконечных множеств. Например, если A и B — непустые конечные множества без общих элементов, то $\overline{\overline{A}} < \overline{\overline{A \cup B}}$. С другой стороны, пусть A — множество всех четных, а B — всех нечетных натуральных чисел. Тогда $A \cup B = \mathbf{N}$ и $A \cap B = \varnothing$. Отображение $f : \mathbf{N} \to A$, где $f(n) = 2n$, биективно, следовательно $\overline{\overline{A \cup B}} = \overline{\overline{A}}$.

Упражнение 1.5.11. Доказать, что $\overline{\overline{\mathbf{N}^2}} = \overline{\overline{\mathbf{N}}}$.

Упражнение 1.5.12. Доказать, что $\overline{\overline{A^*}} = \overline{\overline{\mathbf{N}}}$ для любого конечного алфавита A.

Подобные примеры дают основание спросить, а существуют ли бесконечные множества A и B, такие что $\overline{\overline{A}} < \overline{\overline{B}}$? Ответ утвердителен, как показывает

Теорема 1.5.13 (Кантор). *Для любого множества A выполняется отношение $\overline{\overline{A}} < \overline{\overline{\mathrm{Pow}(A)}}$.* (Отображение Pow определено в разд. 1.5.5.)

Доказательство. 1. Докажем, что $\overline{\overline{A}} \leqslant \overline{\overline{\mathrm{Pow}(A)}}$. Это следует из того, что отображение $f : A \to \mathrm{Pow}(A)$, для которого $\forall a\,(a \in A \supset f(a) = \{a\})$, инъективно.

2. Докажем, что $\overline{\overline{A}} \neq \overline{\overline{\mathrm{Pow}(A)}}$. Предположим обратное, т. е. что $\overline{\overline{A}} = \overline{\overline{\mathrm{Pow}(A)}}$, другими словами, что существует биективное отображение $F : A \to \mathrm{Pow}(A)$. Если x — элемент A, то $F(x)$ — элемент $\mathrm{Pow}(A)$, т. е. подмножество A. При этом x может либо принадлежать, либо не принадлежать этому подмножеству. Определим множество X как множество таких элементов x множества A, которые не принадлежат $F(x)$: $X = \{x \in A \mid x \notin F(x)\}$ (снова упрощенный вариант диагонального метода). Ясно, что $X \subseteq A$, т. е. $X \in \mathrm{Pow}(A)$. Из биективности отображения F следует, что существует единственный элемент a множества A, для которого $X = F(a)$.

Предположим, что $a \in X$, тогда по определению множества X должно быть $a \notin F(a)$, т. е. $a \notin X$. Это противоречие доказывает, что $a \notin X$. Но тогда, опять

же по определению X, не должно иметь места $a \notin F(a)$. т. е. $a \in F(a)$, или же $a \in X$. Вновь возникло противоречие, но теперь оно опровергает уже наше первое предположение о том, что $\overline{\overline{A}} = \overline{\overline{\mathrm{Pow}(A)}}$. Таким образом, $\overline{\overline{A}} \neq \overline{\overline{\mathrm{Pow}(A)}}$.

Из пунктов 1 и 2 следует, что $\overline{\overline{A}} < \overline{\overline{\mathrm{Pow}(A)}}$. $\lhd$

Упражнение 1.5.13. Не противоречат ли теореме Кантора случаи, когда A — пустое или одноэлементное множество?

Упомянем еще одну теорему, не приводя ее доказательства.

Теорема 1.5.14. ((Кантор-)Бернштейн-Шрёдер) *Из* $\overline{\overline{A}} \leqslant \overline{\overline{B}}$ *и* $\overline{\overline{B}} \leqslant \overline{\overline{A}}$ *следует* $\overline{\overline{A}} = \overline{\overline{B}}$.

Парадокс Кантора. Пусть V обозначает «множество» всех множеств. Если все множества — элементы V, то, в частности, любое подмножество V должно быть его элементом, откуда $\mathrm{Pow}(V) \subseteq V$. Отсюда следует, что существует инъективное отображение (тождественное) $f : \mathrm{Pow}(V) \to V$ и, следовательно, $\overline{\overline{\mathrm{Pow}(V)}} \leqslant \overline{\overline{V}}$. Но по теореме 1.5.13 $\overline{\overline{V}} < \overline{\overline{\mathrm{Pow}(V)}}$ т. е. $\overline{\overline{V}} \leqslant \overline{\overline{\mathrm{Pow}(V)}}$ и $\overline{\overline{V}} \neq \overline{\overline{\mathrm{Pow}(V)}}$. Получаем противоречие с теоремой 1.5.14.

В первой половине XX века было предпринято много попыток устранить противоречия из теории множеств. В частности, фон Нейман, Бернайс и Гёдель с участием некоторых других математиков создали формальную, аксиоматическую, теорию множеств, называемую, по инициалам ее авторов, теорией NBG (см. в [34]). В ней произвольные «объекты нашей интуиции или интеллекта» (и то с некоторыми ограничениями) называются *классами*. Аксиомы теории разрешают отдельным классам, а именно — элементам других классов, именоваться множествами, прочие же остаются *собственными* классами. В частности, собрание всех множеств (тем более, всех классов) — это собственный класс, который принято называть *универсальным*. Он и придуман как деревня для поселения в нее всех неудобных парикмахеров и диковинных зверей. Объект X из парадокса Рассела — также собственный класс. Он не может стать своим собственным элементом и превратиться тем самым в множество.

Ограничения на образование классов в теории NBG призваны допустить появление в качестве их элементов всех множеств, в использовании которых заинтересованы математики, но не таких, чтобы вместе с ними возникали противоречия.

Чтобы сделать формулы менее громоздкими, введем ряд соглашений и обозначений. Прописные буквы X, Y, Z, ... будем использовать как переменные для произвольных классов, которые могут оказаться и множествами. Внешние кванторы всеобщности по любым переменным будем опускать (трактуя, следовательно, любую формулу как эквивалент ее замыкания по всем свободным переменным).

Сохраним обозначение $\in$ в его обычном употреблении и содержательном смысле. Определения других привычных символов: '$\notin$', '$=$', '$\subseteq$', '$\subset$' и следствия из них, вроде $X = Y \equiv (X \subseteq Y \wedge Y \subseteq X)$, частично уже были даны.

Понятия множества и собственного класса и символы (предикатные буквы) для их обозначения формально вводятся в теорию NBG так:

$\mathrm{Set}(X) \equiv \exists Y\,(X \in Y)$, т. е. формула $\mathrm{Set}(X)$ утверждает, что при упомянутом выше условии класс X есть множество,

$\mathrm{Pr}(X) \equiv \neg\mathrm{Set}(X)$ — X есть собственный класс.

Строчные буквы x, y, z, u, ... будут служить переменными только для множеств, так что формула $\exists x\,\mathcal{A}(x)$ является сокращением для $\exists X\,(\mathrm{Set}(X) \wedge \mathcal{A}(X))$.

Приведем лишь часть аксиом теории NBG.

Аксиома объемности $X = Y \supset (X \in Z \equiv Y \in Z)$ вместе с другими свойствами обозначения '=' позволяет считать эту теорию теорией первого порядка с равенством, о чем уже говорилось.

Аксиома пустого множества $\exists x \forall y\,(y \notin x)$ утверждает существование множества, не содержащего никаких элементов. Вместе с аксиомой объемности она позволяет доказать (точнее, вывести в теории NBG формулу, утверждающую) единственность такого множества. Это позволяет (см. конец разд. 1.4.6) ввести предметную константу 0 для его обозначения (ниже будет разъяснено, почему такое обозначение предпочтительнее, чем $\varnothing$).

Аксиома пары $\exists z \forall u\,(u \in z \equiv (u = x \vee u = y))$ постулирует для любых двух множеств x и y существование такого множества z, что x и y являются его единственными элементами. Это множество будем обозначать $\{x, y\}$ (что служит сокращением для $\mathrm{Pair}(x, y)$, где Pair — новая функциональная буква).

Одноэлементное множество $\{x\}$ можно определить как $\{x, x\}$ и доказать (выводимость в теории NBG формулы) $\{x\} = \{y\} \supset x = y$.

Чтобы в общих чертах познакомиться с методами этой теории, приведем из нее определение упорядоченной пары, не требующее введения новых сущностей (аксиом):

$$(x, y) = \{\{x, y\}, \{x\}\}$$

Таким образом упорядоченная пара — это пара, один из элементов которой — неупорядоченная пара $\{x, y\}$, определяющая состав упорядоченной, а другой — одноэлементное множество, выделяющее из этой пары один из ее элементов (онто и считается первым в упорядоченной паре).

Работать с таким искусственным и громоздким определением неудобно. Для иллюстрации докажем теорему, фиксирующую основное свойство упорядоченных пар:

Теорема 1.5.15. *Две упорядоченные пары равны тогда и только тогда, когда равны их соответствующие элементы*:

$$(x, y) = (u, v) \equiv x = u \wedge y = v.$$

Доказательство. В левую сторону — пусть $x = u$ и $y = v$. Докажем, что $\{x, y\}$=$\{u, v\}$. Действительно,

$t \in \{x, y\}$
$\Leftrightarrow t = x \vee t = y$ (по определению пары)
$\Leftrightarrow t = u \vee t = v$ (по условию)
$\Leftrightarrow t \in \{u, v\}$ (по определению пары).

Еще проще доказывается, что $\{x\}=\{u\}$.

Заменим в доказанном равенстве $\{x, y\}=\{u, v\}$ элемент x на пару $\{x, y\}$, а элемент u — на равную ей пару $\{u, v\}$, элемент y — на $\{x\}$, а элемент v — на множество $\{u\}$, равное $\{x\}$. Получим равенство $\{\{x,y\},\{x\}\} = \{\{u,v\},\{u\}\}$, т. е. $(x, y) = (u, v)$.

В правую сторону — пусть $(x, y) = (u, v)$, или, в более подробной записи: $\{\{x,y\},\{x\}\} = \{\{u,v\},\{u\}\}$. Так как любой элемент одной из двух равных пар (неупорядоченных) должен быть равен одному из элементов другой пары, то можно выписать цепочки равенств:

$$\{x, y\} = \{u, v\} \vee \{x, y\} = \{u\}, \tag{11}$$

$$\{x\} = \{u, v\} \vee \{x\} = \{u\}, \tag{12}$$

$$\{u, v\} = \{x, y\} \vee \{u, v\} = \{x\}. \tag{13}$$

Из равенства $\{x\}=\{u,v\}$ следует $u = x \wedge v = x$, а из $\{x\}=\{u\}$ — $u = x$. Таким образом, формулу (12) можно переписать в виде

$$(u = x \wedge v = x) \vee (u = x).$$

По закону поглощения должен быть сохранен лишь более слабый член дизъюнкции, т. е. $u = x$. Тем самым доказана истинность первого члена конъюнкции $x = u \wedge y = v$. Переходя к ее второму члену, рассмотрим два случая.

а. Если $x = y$, то формулу (13) можно свести к $\{u,v\}=\{y\}$, откуда следует $v = y$, т. е. $y = v$.

б. Если $x \neq y$, то $y \neq u$, а это значит, что равенство $\{x,y\}=\{u\}$ невозможно. Но тогда из (11) вытекает, что $\{x,y\}=\{u,v\}$, откуда, в частности, $y = u \vee y = v$. Но случай $y = u$ уже исключен, так что $y = v$.

Итак, равенство $y = v$ доказано в обоих возможных случаях. $\lhd$

Аксиома объединения $\exists y\, \forall z\, (z \in y \equiv \exists u\, (z \in u \wedge u \in x))$ утверждает для любого множества x существование множества $y = \bigcup x$ — объединения всех элементов множества x.

Аксиома множества всех подмножеств $\exists y\, \forall z\, (z \in y \equiv z \subseteq x)$ постулирует для любого множества x существования множества $y = \mathrm{Pow}(x)$ всех его подмножеств.

В теории NBG имеются, уже для произвольных классов, аксиомы существования ряда более элементарных, чем $\bigcup$ и Pow, операций и отношений, что дает возможность ввести для них в дополнение к уже имеющимся новые функциональные буквы $\cap$, $\overline{\ }$ (дополнение к классу), $\cup$, $\setminus$, Dom (область определения отношения) и новую константу V (универсальный класс — дополнение к пустому множеству).

Имеется также аксиома бесконечности

$$\exists x\, (0 \in x \wedge \forall u\, (u \in x \supset (u \cup \{u\}) \in x)), \tag{14}$$

постулирующая существование хотя бы одного бесконечного (по построению) множества x, содержащего элемент 0 и для любого элемента u, принадлежащего x, элемент $u \cup \{u\}$.

Каждое такое множество содержит, согласно этой аксиоме, множество 0, множество $0 \cup \{0\} = \{0\}$, содержащее единственный элемент 0 и обозначаемое 1,

множество $1 \cup \{1\} = \{0, 1\}$ с двумя элементами и обозначением 2 и т. д. Множество, получающее обозначение n (можно условиться, что этот символ представляет собой десятичную запись числа), состоит из n элементов: $n = \{0, 1, \ldots, n-1\}$. Именно так вводятся в этой теории натуральные числа. Для множества всех натуральных чисел используется обозначение ω. Но та же аксиома (14) дает право говорить о существовании множества $\omega \cup \{\omega\}$, которое принято обозначать $\omega + 1$, множества $\omega + 2 = \omega + 1 \cup \{\omega + 1\}$ и т. д. Таким образом, за каждым бесконечным множеством выстраивается вереница следующих за ним (в соответствии с порядком их образования) множеств.

Второй способ определения множеств интересующего нас класса — это объединение на основании соответствующей аксиомы некоторого бесконечного и не содержащего наибольшего элемента множества имеющихся в этом классе множеств. Так было образовано множество ω. При этом $\omega = \bigcup \omega$, но есть прием, позволяющий определить ω не через себя (грубо говоря, как минимальное среди всех множеств, существующих согласно аксиоме бесконечности (14)). Все множества, построенные (условно, так как говорить о конструктивности построения здесь не приходится) одним из этих двух способов, называются *порядковыми числами*, а начиная с ω, — *трансфинитными числами*. Числа $0, 1, \ldots, n, \ldots$ — это *конечные* порядковые числа.

В конце разд. 1.5.6 было определено отношение подобия между вполне упорядоченными множествами, являющееся отношением эквивалентности (см. упражнение 1.5.10). Можно убедиться, что каждое порядковое число вполне упорядочено отношением $\in$, а также, что каждое вполне упорядоченное множество подобно некоторому порядковому числу. Благодаря этому, каждое порядковое число может выступать в роли представителя класса подобных ему (и друг другу) вполне упорядоченных множеств. Для доказательства свойств порядковых чисел существует метод *трансфинитной индукции*. Этот метод обобщает на класс трансфинитных чисел метод математической индукции для натуральных чисел (см. схему аксиом S9 из примера 1.4.1–1). Он естественным образом распространяется на доказательство свойств других вполне упорядоченных множеств. Подробнее на этом задерживаться не станем.

Построить формальную теорию множеств в объеме, позволяющем ей стать полноценной базой для создания других математических теорий, можно лишь с помощью более сложных аксиом.

Так, говоря о сравнении мощностей множеств A и B, мы рассмотрели случаи: $\overline{\overline{A}} < \overline{\overline{B}}$, $\overline{\overline{B}} < \overline{\overline{A}}$, а также $\overline{\overline{A}} \leqslant \overline{\overline{B}}$ и $\overline{\overline{B}} \leqslant \overline{\overline{A}}$ одновременно, что приводит к $\overline{\overline{A}} = \overline{\overline{B}}$. Осталось недоказанным, что хотя бы одно из соотношений $\overline{\overline{A}} \leqslant \overline{\overline{B}}$ и $\overline{\overline{B}} \leqslant \overline{\overline{A}}$ обязано выполняться, т. е. не был исключен случай несравнимости мощностей. Однако невозможность этого случая и не может быть доказана. Поэтому она, а чаще — некоторое эквивалентное ей утверждение (аксиома Цермело, лемма Цорна), принимается в теории NBG за аксиому.

Существуют и другие аксиоматизации теории множеств, другие, говоря шире, формальные методы построения оснований математики.

1.5.8. Еще раз о двух математиках

Но и аксиоматический подход оставляет ряд проблем, волнующих умы части математиков. Основная из них — это отсутствие модели у формальной теории множеств. Само понятие интерпретации формулировалось в разд. 1.2.2 в терминах теории множеств, поскольку в нем фигурировало некоторое множество, называемое предметной областью интерпретации. Если для теории собственно множеств более общую теорию с классами можно считать метатеорией и выделить в последней класс, который можно назвать предметной областью для первой, то что же может послужить предметной областью этой метатеории? Здесь или где-то рядом приходится опираться на естественный язык и здравый смысл для обоснования исследуемых понятий.

Коснемся также понятия континуума и методов его исследования.

Континуумом называется множество $\mathbf{R}$ всех вещественных чисел. Что можно сказать о мощности континуума?

Теорема 1.5.16. $\overline{\overline{\mathbf{R}}} = \overline{\overline{\mathrm{Pow}(\mathbf{N})}}$.

Доказательство. Непрерывная монотонная функция g_1, где

$$g_1(x) = \frac{2x-1}{x(1-x)}$$

для $0 < x < 1$, биективно отображает открытый промежуток $]0, 1[$ на $\mathbf{R}$. Выберем из этого множества последовательность $x_i = 1/(i+1)$, $i = 1, 2, \ldots$, сдвинем все ее элементы на два места вправо, а на освободившиеся места поместим числа 0 и 1. Это дает нам способ биективно отобразить $[0, 1]$ на $]0, 1[$, а вместе с тем и на $\mathbf{R}$, следовательно, все три множества равномощны.

Множество $\mathrm{Pow}(\mathbf{N})$ или $2^{\mathbf{N}}$ — это, в соответствии с общим определением из разд. 1.5.3, множество всех отображений множества $\mathbf{N}$ в двухэлементное множество $\{0, 1\}$, иначе говоря, — множество всех бесконечных последовательностей $b = b_0, b_1, b_2, \ldots$, где $b_i \in \{0, 1\}$ для $i = 0, 1, 2, \ldots$

Построим отображение $g : [0, 1] \to 2^{\mathbf{N}}$. Для этого разложим любое число $x \in [0, 1]$ в двоичную дробь. Если оно имеет два разложения: $0, b_0 \ldots b_{k-1}1000\ldots$ и $0, b_0 \ldots b_{k-1}0111\ldots$, то выберем второе из них. В частности, числу 1 сопоставим дробь $0{,}111\ldots$ Все такие разложения для $0 \leqslant x \leqslant 1$ имеют вид $0, b_0 b_1 b_2 \ldots$, где $b_i \in \{0, 1\}$ для всех i. Положим $g(x) = b_0, b_1, b_2, \ldots$. Отображение g всюду определено, однозначно (благодаря сделанному выбору) и инъективно, так как по двоичному разложению число восстанавливается однозначно. Таким образом, $\overline{\overline{[0, 1]}} \leqslant \overline{\overline{2^{\mathbf{N}}}}$.

Упражнение 1.5.14. Превратить отображение g в биективное.

Решение этого упражнения завершает доказательство. $\lhd$

Итак, континуум вместе с множеством $\mathrm{Pow}(\mathbf{N})$ по теореме Кантора несчетен. Каков смысл понятия несчетности? Любое счетное множество, например множество натуральных чисел, более или менее обозримо — мы легко допускаем, что каждое натуральное число можно записать цифрами на бумаге (оставляя в

стороне вопрос — хватит ли у человечества запасов бумаги, чтобы записать в десятичной системе число $10^{10^{10}}$). В несколько ином положении находится множество $\mathrm{Pow}(x)$, если x бесконечно, пусть всего лишь счетно. По теореме 1.5.13 $\overline{\overline{x}} < \overline{\overline{\mathrm{Pow}(x)}}$, т. е. $\mathrm{Pow}(x)$ — несчетно. Однако записать в виде конечного слова в любом конечном алфавите A можно лишь счетное множество слов. Поэтому, изобретая любые способы конечного описания элементов $\mathrm{Pow}(x)$, мы сможем описать (You must not forget about paper, Best Beloved) лишь счетное множество Y этих элементов. Неописанным останется несчетное множество $\mathrm{Pow}(x)\backslash Y$.

Естественно, что математики, называющие себя интуиционистами или конструктивистами, таким безликим существованием подавляющего большинства элементов множества $\mathrm{Pow}(x)$ не удовлетворяются (признавая тем самым, что их интуиция уступает канторовой). Реально существующими они согласны считать лишь такие объекты, которые можно как-то охарактеризовать индивидуально (сконструировать). Несчетность с таких позиций — понятие бессодержательное.

Следует честно признать, что этого мнения придерживается сравнительно небольшая часть математиков, и ряды их редеют. Остальные — «классические» — математики признают реальность несчетных множеств и даже значительно более абстрактных понятий. Причина такого подхода все та же — многие формулировки понятий классической математики оказываются при этом проще, а об интерпретации результатов абстрактной теории пусть заботятся другие.

Упомянем в этой связи еще некоторые обстоятельства. В рамках классического подхода может быть доказана теорема Лёвенгейма-Скулема: если формальная теория $\mathbf{Q}$ имеет какую-нибудь бесконечную модель, то она имеет и модель со счетной предметной областью. В частности имеет такую модель и теория (не обязательно полная теория множеств), в которой может быть доказано существование несчетных множеств. Этот результат не приводит к противоречию (в предметной области такой модели есть объект, соответствующий несчетному множеству в целом, но нет объектов, соответствующих каждому его элементу, да нет и средств указать элемент, не имеющий соответствия в модели). Но он все же делает понятие несчетности весьма условным.

К этому вопросу мы еще вернемся в разд. 1.6.

Возникают и совсем субъективные мысли. Человек, чем бы он ни занимался — программированием или математикой — не может не иметь личной точки зрения на предмет своих занятий, на сам предмет, а не на возможности его сбыта.

Вероятно, у каждого исследователя есть своя точка зрения, например, на континуум — то ли это множество, «составленное» каким-то образом из всех вещественных чисел, то ли нечто иное. Можно представить себе числовую ось как своего рода память, в которую записаны все рациональные числа, а между ними остаются свободные места для записи по мере необходимости новых, иррациональных, чисел при их появлении в рассуждениях. Ссылка на меру необходимости позволяет не задаваться вопросом о счетности или несчетности совокупности этих свободных мест. Можно занять часть из них, разместив все алгебраические числа (их множество все еще счетно), константы вроде π и т. д. — ситуация не изменится.

Если исследователь попытается как-то сформулировать это представление, то получится описание некой счетной совокупности — других описаний не бывает. Он излагает это представление своему собеседнику, а тот говорит: «Но ведь оно неполно — я могу, отталкиваясь от этого описания, охарактеризовать объект, не совпадающий ни с одним из объектов, в него входящих» (в этом суть диагонального метода). Реакция может быть двоякой: «Да, пожалуй, ты прав», или «Твой объект никого, кроме тебя, не интересует, во всяком случае, — не меня». На это тоже можно реагировать по-разному, но проблема конструктивности и убедительности диагонального метода и даже шире — проблема личностной точки зрения на науку и исследуемые ею сущности остается.

1.6. Вероятности и информация

Теория вероятностей — вездесущая (от карточной игры до космологии) математическая дисциплина, оперирующая с понятием случая. Вездесущность ассоциируется с фундаментальностью, а случайность — с неуловимостью. Попробуем разобраться с сутью дела.

1.6.1. Случайные события

Будем рассматривать некоторые множества (*пространства*) X, Y и др. с элементами x_k, y_i и т. д., называемыми *событиями*. В результате некоторого действия (*испытания*, выбора) из множества X выделяется один элемент (наступает событие) x_k. События, составляющие пространство X, должны быть попарно *несовместимыми* — если в некотором испытании наступило событие x_k, то никакое другое событие x_i $(i \neq k)$ не может наступить при данном испытании. Каждому событию x_k каким-то образом приписывается число $P(x_k)$ — вероятность (степень ожидания) его наступления в условиях, которыми обставлены испытания. При этом должны соблюдаться ограничения:

$$P(x_k) \geqslant 0 \qquad \text{для любого } x_k \text{ из } X, \tag{1}$$

$$\sum_{x_k \in X} P(x_k) = 1. \tag{2}$$

Из этих ограничений следует, что $P(x_k) \leqslant 1$ для любого x_k. Говорят, что тем самым (а чем именно, если это сделано «каким-то образом»?) на множестве X задано *распределение* вероятностей (иначе — вероятностная *мера*), а множество вместе с заданным распределением вероятностей называют *ансамблем* событий.

Пусть из множества X возможных исходов испытания выделено подмножество Q с элементами x_{k_1}, x_{k_2}, ... Попадание события x_k, наступившего при испытании, в множество Q — это также событие, обозначаемое $x_k \in Q$. Его вероятность должна быть равна

$$P(x_k \in Q) = P(x_{k_1}) + P(x_{k_2}) + \ldots$$

или иначе

$$P(x_k \in Q) = \sum_{x_i \in Q} P(x_i). \tag{3}$$

Это свойство называется *аддитивностью* вероятностной меры. Оно имеет место только для несовместимых, как было оговорено, событий.

В реальной обстановке ключом к определению вероятностей событий служит понятие равновероятности событий. Если ансамбль X состоит из конечного числа m событий и обстановка не дает оснований считать какие-то из этих событий более, а какие-то — менее вероятными, то все события x_k объявляются *равновероятными* и на основании формулы (2) вероятность любого из них должна быть равна $P(x_k) = 1/m$.

Допускается возможность одновременного наступления в одном испытании двух событий из разных ансамблей: x_k из X и y_j из Y. Фактически исходом

такого испытания является пара событий (x_k, y_j) и вероятность его наступления должна быть обозначена $P(x_k, y_j)$. Ансамбль таких пар обозначим XY.

Для ансамбля XY можно интересоваться только одним, например первым, событием пары. Выделим из XY подмножество Q, состоящее из элементов (x_k, y_j), где x_k — интересующий нас элемент ансамбля X, а y_j — произвольный элемент ансамбля Y. Попадание пары (x_k, y_j) в это множество исчерпывает все случаи наступления события x_k при испытаниях, связанных с ансамблем XY. По формуле (3) получаем $P(x_k) = \sum_{(x_k, y_j) \in Q} P(x_k, y_j)$ или

$$P(x_k) = \sum_{y_j \in Y} P(x_k, y_j). \tag{4}$$

Аналогично

$$P(y_j) = \sum_{x_k \in X} P(x_k, y_j). \tag{5}$$

События из ансамблей X и Y называются *независимыми*, если для любой пары (x_k, y_j) таких событий действует *правило умножения* вероятностей $P(x_k, y_j) = P(x_k)\, P(y_j)$. Для независимых событий равенства (4) и (5) превращаются в тождества в силу формулы (2).

События можно объединять не только в пары, но и в более крупные наборы. Соответственно изменятся понятия условной вероятности и независимости событий (правило умножения распространяется на весь набор).

Если свойство независимости событий нарушается, то появляется смысл говорить об условной вероятности: $P(y_j|x_k)$ — вероятности наступления события y_j при условии, что произошло событие x_k. Она определяется формулой

$$P(y_j|x_k) = P(x_k, y_j)/P(x_k). \tag{6}$$

Аналогичный смысл имеет условная вероятность

$$P(x_k|y_j) = P(x_k, y_j)/P(y_i). \tag{7}$$

Для условных вероятностей необходимое свойство меры:

$$\sum_{y_j \in Y} P(y_j|x_k) = 1, \quad \sum_{x_k \in X} P(x_k|y_j) = 1, \tag{8}$$

аналогичное свойству (2), прямо следует из формул (4) и (5). Это позволяет рассматривать «условные» ансамбли: $Y|X$, распадающийся на ансамбли $Y|x_k$, и $X|Y$. Ансамбль $Y|x_k$ состоит из тех же событий y_j, что Y, но с иной вероятностной мерой, учитывающей, что событие x_k уже наступило.

Так называемая *формула Байеса*

$$P(x_k)\, P(y_j|x_k) = P(y_j)\, P(x_k|y_j) \tag{9}$$

вытекает из формул (6) и (7) благодаря возможности выразить вероятность $P(x_k, y_j)$ совмещенного события двумя способами. Полученную зависимость можно разрешить относительно любой из четырех входящих в нее вероятностей. В условиях, когда ансамбли X и $Y|X$ считаются первичными, а все остальные — вторичными, вероятности $P(x_k)$ называют *априорными*, а следствие

$$P(x_k|y_j) = P(x_k)\, P(y_j|x_k)/P(y_j)$$

формулы (9) — теоремой Байеса для *апостериорных* вероятностей $P(x_k|y_j)$.

События, выстроенные в последовательность или каким-то иным способом упорядоченные во времени, образуют случайный процесс (см. ниже, разд. 1.6.4). Распределение вероятностей при каждом очередном событии может зависеть или не зависеть от того, что было раньше. Статистика результатов последовательных испытаний осмыслена лишь тогда, когда все испытания проводятся в одинаковых условиях и попарно (и во всей их совокупности) не зависят друг от друга, так что вероятности исходов не меняются.

1.6.2. Случайные величины

Пусть имеется величина r, которая в разных случаях может принимать разные значения, и $A = \{a_1, a_2, \dots\}$ — полный набор этих возможных значений. Случай, когда значение r равно a_k, назовем событием $r = a_k$, а вероятность $P(r = a_k)$ этого события обозначим p_k. Если все эти вероятности известны, говорят, что для *случайной величины* r задан *закон распределения*. Из формулы (2) и из того, что набор A содержит все возможные значения, следует формула

$$\sum p_k = 1$$

Иногда (но не при подсчете средней температуры тела больных в больнице) есть смысл говорить о *среднем значении* (иначе — *математическом ожидании*) $E(r)$ величины r, вычисляемом по формуле

$$E(r) = \sum_k P(r = a_k)\, a_k \tag{10}$$

(сокращенно — $E(r) = \sum p_k\, a_k$), т. е. $E(r)$ — это взвешенная сумма всех возможных значений a_k с весами p_k.

Пусть теперь наличествуют две величины: r и s с возможными значениями $\{a_1, a_2, \dots\}$ и $\{b_1, b_2, \dots\}$ и есть смысл говорить о сумме $r + s$. Пусть p_{kl} — вероятность события $(r = a_k, s = b_l)$. Не будем исключать возможность взаимной зависимости событий $r = a_k$ и $s = b_l$. Условную вероятность $P(s = b_l | r = a_k)$ обозначим $p_{l|k}$, аналогично $p_{k|l}$ — это $P(r = a_k | s = b_l)$.

Формула (10) для рассматриваемого случая дает:

$$E(r + s) = \sum_{k,l} p_{kl}\, (a_k + b_l).$$

Формулы (6)–(8) принимают вид: $p_{l|k} = p_{kl}/P(r = a_k)$, $p_{k|l} = p_{kl}/P(s = b_l)$ и $\sum_k p_{l|k} = \sum_k p_{k|l} = 1$, что открывает дорогу следующим выкладкам:

$$\begin{aligned} E(r + s) &= \sum_k \sum_l p_{l|k}\, P(r = a_k)\, a_k + \sum_l \sum_k p_{k|l}\, P(s = b_l)\, b_l \\ &= \sum_k P(r = a_k)\, a_k + \sum_l P(s = b_l)\, b_l, \end{aligned}$$

так что

$$E(r + s) = E(r) + E(s).$$

Этот результат — теорема о среднем значении суммы случайных величин — может быть без труда распространен на сумму любого числа случайных величин:

$$E\Big(\sum_j r_j\Big) = \sum_j E(r_j). \tag{11}$$

Важный частный случай. Пусть при некотором испытании событие наступает с вероятностью p (и, следовательно, не наступает с вероятностью $q = 1 - p$). Для случайной величины r — числа наступления этого события при данном испытании — формула (10) дает

$$E(r) = p \cdot 1 + q \cdot 0 = p$$

Пусть теперь проводится серия из n подобных испытаний в условиях их независимости, т. е. неизменности вероятности p, и m — общее число наступлений рассматриваемого события во всей серии. По теореме о среднем значении суммы

$$E(m) = n\,p$$

Поскольку m/n — это частота наступления события в данной серии, полученный результат означает, что средняя частота совпадает с вероятностью наступления события. Это был простейший пример накопления статистики при независимых испытаниях.

Со случайностью тесно связано понятие информации, один из вариантов которого мы сейчас и рассмотрим.

1.6.3. Об измерении информации

Исходным здесь является понятие процесса получения информации. В начале процесса имеется представление о множестве его возможных *исходов* (ожидаемых событий вместе с их вероятностями). Это множество — то же самое, что ансамбль. Процесс завершается с наступлением одного из этих событий. Начальная неопределенность полностью устраняется. Это и означает, что получена некоторая информация. Процессы могут протекать и по более сложным схемам. Теория информации имеет целью установить «естественную» меру количества информации, получаемой в результате (иногда — и по ходу) процесса.

Начнем с простейшего случая m равновероятных несовместимых исходов. Примем за меру количества полученной по завершении такого процесса информации величину $I = \log m$. Функция log появилась здесь потому, что для случая двух осуществляющихся совместно (не важно, последовательно или параллельно), но независимо друг от друга процессов, характеризуемых, соответственно, ансамблями $\{x_k\}$, $k = 1, \ldots, m_1$, и $\{y_i\}$, $i = 1, \ldots, m_2$, и количеством информации I_1 и I_2, естественно требование аддитивности меры: $I = I_1 + I_2$. Число m исходов для их общего результата — наступления пары событий (x_k, y_i) — равно $m_1\,m_2$, а $\log(m_1\,m_2) = \log m_1 + \log m_2$, что и требуется (других непрерывных и гладких функций f со свойством $f(x_1\,x_2) = f(x_1) + f(x_2)$ не существует).

Логарифм можно брать по любому основанию, это равносильно выбору единицы измерения количества информации. В простейшей ситуации выбора между всего лишь двумя возможностями ($m = 2$, на поставленный вопрос ожидается ответ «да» или «нет») количество получаемой информации «естественно» принять за 1. Это приводит к соглашению, что логарифмы следует брать по основанию 2 и можно не указывать на это явно.

Из постулата об аддитивности меры информации (подразумевается — ее количества) следует, что если процесс протекает в два этапа, причем информация, полученная в результате всего процесса равна I, а в результате второго этапа — I_2, то первому этапу следует приписать количество информации $I_1 = I - I_2$. Например, если в начале процесса было m равновероятных исходов, а в начале второго этапа их стало только m_2 (не обязательно образующих подмножество первоначальных), то $I_1 = \log(m/m_2)$.

Найдем теперь подход к случаю неравновероятных исходов. Пусть в начале процесса имеется m возможных равновероятных исходов, разбитых на n групп, причем j-я группа содержит m_j исходов, так что $\sum_j m_j = m$. Процесс протекает в два этапа. На первом этапе выясняется только, к какой из групп принадлежит окончательный исход. Можно сказать, что этот этап завершается наступлением одного из n событий (промежуточных исходов) C_j, попарно несовместимых и составляющих ансамбль событий с вероятностями $p_j = m_j/m$, причем $\sum_j p_j = 1$, как и положено.

После наступления одного из событий C_j на втором этапе выясняется, какая из m_j оставшихся возможностей фактически реализовалась. Особенность рассматриваемого процесса в том, что второй этап, как и количество $\log m_j$ получаемой на нем информации, зависит (более существенно, чем оказывалось до сих пор) от того, каким из n событий C_j завершился первый этап. Это позволяет говорить лишь о среднем значении $E(I_2)$ этой информации, вычисляемом по обычной формуле $I_2 = \sum_j p_j (\log\ m_j)$.

Так как информация, получаемая в результате всего процесса, равна $I = \log\ m$, то на долю первого этапа, представляющего для нас главный интерес, приходится (опять же в среднем) количество информации $I_1 = I - I_2 = \log m - \sum_j p_j (\log\ m_j)$, что легко приводится к виду $I_1 = \sum_j p_j \log(m/m_j)$. Это и есть, после еще одного небольшого преобразования, фундаментальная *формула Шеннона*:

$$I = -\sum_j p_j \log(p_j) \tag{12}$$

для среднего количества информации, получаемой в результате наступления одного из n событий, имеющих, в общем случае, различные вероятности p_j. Если же, в частности, эти вероятности одинаковы: $p_j = 1/n$, то $\log(p_j) = -\log(n)$, а количество I_0 информации, получаемой в этом частном случае, равно $I_0 = \log(n)$, как и должно быть. Легко убедиться, что $I \leqslant I_0$ и это неравенство переходит в равенство только в случае равных вероятностей p_j.

Перенос формулы Шеннона на еще более общий случай вероятностей, выражаемых иррациональными числами, осуществляется путем предельного перехода. Вид формулы при этом не изменяется, правда, конечная сумма может превратиться, хотя и не обязательно, в бесконечный ряд.

Понятие информации можно применить к случаю зависимых событий. Пусть X, Y, XY и $Y|X$ имеют тот же смысл, что и выше, в разд. 1.6.1. Среднее количество информации, которое несет наступление события из ансамбля XY, дается все той же формулой Шеннона, принимающей вид

$$I(XY) = -\sum_{k,i} P(x_k, y_i) \log P(x_k, y_i) \tag{13}$$

Если ансамбли X и Y фактически независимы, то

$$P(x_k, y_i) = P(x_k)\, P(y_i),$$

откуда

$$\begin{aligned} I(XY) &= -\sum_{k,i} P(x_k)\, P(y_i) \log P(x_k) - \sum_{k,i} P(x_k)\, P(y_i) \log P(y_i) \\ &= -\sum_{k} P(x_k) \log P(x_k) - \sum_{i} P(y_i) \log P(y_i), \end{aligned}$$

поскольку $\sum_i P(y_i) = \sum_k P(x_k) = 1$. Таким образом

$$I(XY) = I(X) + I(Y). \tag{14}$$

Это согласуется с постулатом об аддитивности меры информации.

В случае зависимости ансамблей X и Y можно рассмотреть процесс, в котором реализуется сначала событие из ансамбля X, поставляющее информацию $I(X)$, а в полученной ситуации — событие из ансамбля Y, в результате чего общее количество информации возрастает до $I(XY)$. Дополнительное количество информации, поставляемое на втором этапе, можно вычислить как

$$I(Y|X) = I(XY) - I(X) \tag{15}$$

или

$$I(Y|X) = -\sum_{k,i} P(x_k, y_i) \log P(x_k, y_i) + \sum_{k} P(x_k) \log P(x_k).$$

Первый член преобразуется сначала к виду

$$-\sum_{k,i} P(x_k)\, P(y_i|x_k) \log P(x_k) - \sum_{k,i} P(x_k)\, P(y_i|x_k) \log P(y_i|x_k)$$

на основании формулы (6), а затем — к виду:

$$-\sum_{k} P(x_k) \log P(x_k) - \sum_{k} P(x_k) \sum_{i} P(y_i|x_k) \log P(y_i|x_k),$$

на основании первой из формул (8). Окончательно,

$$I(Y|X) = \sum_{k} P(x_k)\, I(Y|x_k), \tag{16}$$

где

$$I(Y|x_k) = -\sum_{i} P(y_i|x_k) \log P(y_i|x_k) \tag{17}$$

— информация, приносимая ансамблем Y при условии, что на первом этапе реализовалось именно событие x_k. Иначе говоря, $I(Y|x_k)$ — это условная информация, соответствующая упомянутому в разд. 1.6.1 ансамблю $Y|x_k$. И здесь получение такой формулы можно было предвидеть.

Изложенный выше вариант обоснования формулы Шеннона математически корректен. Его можно найти в литературе (например, в [48], но там он гораздо более многословен). Да и увидеть в формуле Шеннона среднее значение количества информации по возможным ветвям процесса — «не просто, а очень просто».

Беда с формулой Шеннона — типичная для красивых формул — состоит в том, что для определения входящих в нее вероятностей p_j необходим гораздо более сложный информационный процесс, чем тот, к которому относится эта формула.

Могут ли приведенные выше соображения помочь при оценке ценности сообщения для человека? Некоторые авторы, например Л. Бриллюэн в [7], с первых же страниц отказываются рассматривать человеческую оценку информации. Кое в чем можно согласиться с этим. Однако ничто не мешает говорить о множестве ожидаемых событий при получении человеком некоторого сообщения. Например, большинство людей, не знающих французского языка, просто отложат в сторону попавшую к ним в руки французскую книгу. Но некоторые попытаются — по обложке, по фамилии автора, по отдельным знакомым словам и т. п. — составить какое-то представление о книге. Человек, владеющий французским, может просмотреть или прочесть аннотацию или предисловие, а затем и всю книгу. По той же причине — разнообразия возможных реакций — врач способен извлечь гораздо больше сведений из медицинской книги, чем программист. Да и смысл, вложенный в книгу автором, будет значительно полнее воспринят специалистом, чем профаном. Все это и означает, что количество информации, извлекаемой человеком из сообщения, неплохо поддается оценке изложенными выше методами.

1.6.4. Случайные процессы

Понятие случайного процесса весьма многостороннее. Познакомимся с ним на двух простых примерах.

Цепи Маркова. Они (названы по имени А. А. Маркова-старшего — отца) представляют собой бесконечные последовательности событий, заключающихся в переходе от одного ансамбля к другому, возможно совпадающему с предыдущим. Множество различных ансамблей s_j, иначе называемых *состояниями*, конечно. Пусть m — число его элементов. Цепь Маркова полностью характеризуется $m\times m$-матрицей P, элемент p_{jk} которой — это вероятность перехода из состояния s_j в состояние s_k. Процесс переходов из состояния в состояние часто называют *блужданием*. Вероятности p_{jk} не меняются на протяжении всего процесса.

Матрице P можно сопоставить ориентированный граф G с m вершинами. Дуга $\langle j,\, k\rangle$ присутствует в этом графе только при $p_{jk} > 0$. Граф G' компонент сильной связности графа G характеризует довольно существенную особенность блуждания — невозможность вернуться в уже покинутую компоненту. Поэтому при переходе из одной компоненты в другую статистические характеристики процесса довольно резко меняются. Граф G' ацикличен, а потому должен содержать хотя бы одну тупиковую вершину — с полустепенью исхода, равной нулю. При попадании в эту компоненту процесс вступает в *стационарную* фазу. Если граф G' состоит всего из одной компоненты, то процесс стационарен с самого начала. Если же в нем есть тупиковая компонента, содержащая лишь одну вершину графа G, то соответствующее состояние называется *поглощающим* — можно считать, что переход в это состояние завершает блуждание.

Не теряя общности, сольем все поглощающие состояния в одно — s_m. В матрице $Q = (p_{jk})_{j,k=1}^{m-1}$ сосредоточена вся существенная информация о процессе, поскольку $p_{mk} = 0$ при $k < m$, а $p_{jm} = 1 - \sum_{k=1}^{m-1} p_{jk}$ при $j \leqslant m$.

Сопоставим каждому однократному попаданию в состояние s_i для $i \leqslant m$ некоторый вес V_i — разновидность информационной меры на множестве состояний. Эта мера может сильно отличаться от меры I из предыдущего раздела, но она должна оставаться аддитивной. Несколько слов об этом будет сказано в конце. Обозначим через W_i среднее значение суммарного веса, накапливаемого при блуждании из исходного состояния s_i. Примем, что V_i не зависит от того, на каком шаге блуждания наступило состояние s_i. Из-за неизменности вероятностей перехода W_j также не будет от этого зависеть ни при каком j.

Поясним сказанное примером: если $V_j = 1$ при $j < m$, а $V_m = 0$, то W_i — это средняя продолжительность (число пройденных состояний) блуждания из состояния s_i до прихода в поглощающее состояние s_m. Другой вариант — если $V_j = 1$ для некоторого j, но $V_k = 0$ при $k \neq j$, то W_i — это среднее число повторений состояния s_j при таком блуждании.

Пример наводит на мысль, что из W_i полезно выделить вклад W_{ij} каждого из состояний s_j по ходу блуждания из состояния s_i. Значения W_{ij} должны удовлетворять уравнениям

$$W_{ii} = V_i + \sum_{k=1}^{m-1} p_{ik} W_{ki}, \qquad W_{ij} = \sum_{k=1}^{m-1} p_{ik} W_{kj} \quad \text{при } j \neq i,$$

для $1 \leqslant i, j < m$. Действительно, при $j \neq i$ первое состояние цепи заведомо отлично от s_j, а при $j = i$ — совпадает с s_j (т. е. с s_i) и дает одномоментный вклад V_i. Затем с вероятностью p_{ik} процесс переходит в состояние s_k, и с учетом этой вероятности дает вклад $p_{ik} W_{kj}$ в W_{ij} (при $k < m$). С вероятностью p_{im} сразу из состояния s_i можно попасть в поглощающее состояние, и вес навсегда перестанет возрастать.

В матричной форме: $W = V + QW$, где W — это матрица $\left(W_{ij}\right)_{i,j=1}^{m-1}$, V — вектор-столбец $V = (V_1, \ldots, V_{m-1})'$, а E — единичная матрица того же размера. Отсюда: $(E - Q)W = V$, так что

$$W = (E - Q)^{-1} V. \tag{18}$$

Матрица Q в указанном выше смысле управляет шагами блуждания, в массиве V собраны информационные характеристики состояний, а матрица W дает довольно полную информационную картину всего процесса блуждания при произвольном начальном состоянии.

А. Н. Колмогоров [24] предложил в качестве меры информации, связываемой с исполнением вычислительного процесса, минимальную длину алгоритма, описывающего этот процесс. Однако самое короткое описание может оказаться не самым экономным по числу исполняемых действий, а кратчайшая длина композиции алгоритмов — меньше суммы их длин, что нарушает требование аддитивности меры. Впрочем в марковской модели каждое состояние рассматривается независимо от предшествующих. Содержательно это означает запрет на использование накопленного опыта, а это — аргумент в пользу аддитивности.

По-видимому, именно труд, затрачиваемый на получение результата, должен служить мерилом информационной сложности процесса. Понятия информации и сложности настолько близки по смыслу и по свойствам, что один термин часто заменяют другим.

Процесс Пуассона. Процесс состоит в том, что в какие-то моменты непрерывного времени t подаются независимо друг от друга некие мгновенные импульсы. Вероятность $p(t_0, \Delta t)$ поступления импульса в промежутке времени $[t_0, t_0 + \Delta t]$ не зависит от t_0 и равна $\nu \Delta t$, где ν — единственный параметр процесса. Ясно, что это выражение не может быть точным, иначе, задав достаточно большое Δt, можно было бы получить для вероятности значение больше 1. Его точность повышается с уменьшением Δt. Независимость от t_0 позволяет не указывать в дальнейшем аргумент t_0 функции p. В пределе при $\Delta t \to 0$ для $p(t)$ получаем линейное дифференциальное уравнение $dp(t) = \nu dt$. Его решение: $p(t) = -e^{-\nu t} + C$. Из начального условия $p(0) = 0$ (или из условия $\lim_{t\to\infty} p(t) = 1$) получаем $C = 1$, так что $p(t) = 1 - \exp(-\nu t)$.

Полученный непрерывный закон распределения вероятностей называется распределением Пуассона. Разобьем интервал $T = [0, t]$ на n мелких интервалов. Вероятность поступления импульса на любом из них равна $\nu\, t/n$, вероятностью поступления более чем одного импульса на очень малом интервале можно пренебречь. Среднее значение числа импульсов, поступивших на всем интервале T, равно νt (ср. с концом разд. 1.6.2). Итак, параметр ν — это средняя частота поступления импульсов не только на коротких интервалах времени, как следовало из определения процесса, но и на интервалах произвольной длины.

1.7. Теория вычислимости

Люди вычисляют много, но не то чтобы охотно, а по необходимости — есть присловье «Деньги счет любят». Без скрупулезных расчетов в наше время невозможно построить ни одно сооружение, спроектировать ни одну машину. Арифметике детей учат с первого класса и норовят учить еще раньше. Чтобы избавить взрослых от непосильных вычислений, пришлось создать компьютеры.

Бухгалтерский баланс мог бы сойтись до копейки лишь при абсолютно точном счете деньгам. Но в технике и в естественных науках, даже там, где требуется «астрономическая точность», уже измерения, поставляющие исходные данные для любых вычислений, неограниченную точность не обеспечивают. С античных времен известно, что диагональ и сторона квадрата несоизмеримы — их отношение не может быть записано конечным числом цифр ни в какой системе счисления. Кстати, история появления и развития систем счисления — это еще одно свидетельство важности вычислений в жизни человека.

По названным причинам абсолютно точные вычисления невозможны, да и не нужны. Математики удовлетворяются обозначениями вроде $\sqrt{2}$ для уже упомянутого отношения диагонали квадрата к его стороне или π — для отношения длины окружности к ее диаметру.

А практики довольствуются приближенными результатами вычислений. Вычислительная математика создала ряд важных понятий — от округления результата до неустойчивости вычислительного процесса и т. п. — и добилась с их помощью серьезных результатов, претендующих на фундаментальность. На стыке теории вычислений и вычислительной математики возникла теория сложности вычислений, которая едва ли не затмила своих предков.

Практики — народ своеобразный. Вслед за Архимедом (отнюдь не чистым практиком) они готовы бегать нагишом по улицам родного города с криком «Эврика!», когда им удается найти практичный, хотя теоретически и невозможный, — «эвристический» — метод решения волнующей их задачи. Но и теоретики не остаются в долгу, продолжая обнаруживать все новые «алгоритмически неразрешимые» задачи. Мудрый судья говорит им: «Ты прав» и «Ты тоже прав», а растерявшемуся обывателю, замечающему, что не могут оба быть правы, отвечает: «И ты тоже прав».

Иметь в виду столь универсальную точку зрения на правильность утверждений отнюдь не вредно при чтении литературы по основаниям математики. Реальная сложность мира выше сложности любых абстракций, что не может не отражаться на позициях и высказываниях разных лиц.

А вычисления в излагаемой ниже теории понимаются гораздо шире, чем в вычислительной математике.

1.7.1. Введение

В этом разделе кратко излагается теория процессов *вычисления*, которые, отправляясь от некоторых *исходных данных*, завершаются в благоприятном случае получением соответствующего *результата*. Должен быть точно определен

способ осуществления таких процессов — без этого никакую теорию построить невозможно. Для этого описывается некоторая *абстрактная машина*, работающая с данными — преобразующая их — по определенным правилам. Вид данных и правила их преобразования — это все, что надо знать о машине.

Данные могут быть чрезвычайно, даже неограниченно, разнообразны по содержанию. Но их форма должна быть определена коротко и понятно — не для машины, а для человека, который избирает некий вид данных (например, письменную речь, графические схемы, программы на алгоритмических языках и т. п.), чтобы зафиксировать свои мысли и знания о мире, событиях, фактах и передать их в этом виде другим людям, владеющим тем же или сходным представлением о связи подобных данных с возможными мыслями и знаниями.

Примерно то же самое можно сказать и о правилах работы машины. При всем разнообразии (желательно — предельном) процессов, которые машина должна быть способна исполнить, совокупность правил должна быть обозримой. Иначе было бы невозможно ни предписать машине какой-либо образ действий, ни убедиться самому, что эти действия способны привести к желаемой цели. Для этого процесс вычисления обычно расчленяется на *шаги*. На каждом шаге машина исполняет одно *элементарное действие*, руководствуясь одним выбранным правилом. При этом используются *текущие* данные — те, которыми машина располагает к началу шага, — возможно даже, лишь их небольшая часть. Текущими данными для самого первого шага служат исходные данные процесса.

Каждый шаг исполняется по такой схеме: выбирается или окончательно уточняется правило для данного шага, изменяются в соответствии с этим правилом текущие данные или их выделенная часть, проверяется — не закончен ли процесс, если еще нет, то подготавливается следующий шаг. Если процесс закончился, то полученные текущие данные становятся результатом его исполнения.

Нет никакой гарантии, что на каком-либо шаге проверка на окончание процесса даст положительный результат (это и был бы упомянутый в самом начале благоприятный случай). В неблагоприятном случае процесс не сможет завершиться никогда. Но машина ничего «знать» об этом не может — если бы могла, то это дало бы ей повод прекратить исполнение процесса. Даже если за работой машины наблюдает человек (который, в соответствии со сказанным, не имеет права вмешиваться в работу машины, но может за ней следить), предсказать дальнейший ход процесса ему в общем случае трудно, а то и невозможно.

Правила работы абстрактной машины для решения некоторой *задачи* — получения результата, связанного определенным образом с исходными данными, — называются *алгоритмом* ее решения.

Для примера опишем абстрактную *машину Тьюринга*. Данные, с которыми она работает, представлены на бесконечной *ленте*, разбитой на *ячейки*. В каждой ячейке может быть записана *буква* из заранее фиксированного *алфавита* этой машины. Над одной из ячеек находится *головка* машины. Задан также конечный набор возможных *состояний* машины. Эти состояния никак не связываются с содержимым ленты.

Содержимое ленты (записанное на ней слово), состояние машины и положение головки вместе составляют то, что было названо текущими данными.

Правила работы машины определяются *программой*, состоящей из конечного набора *команд*. Выбор команды для исполнения зависит от текущего состояния машины и от буквы, записанной в ячейке, над которой находится головка (ячейка может оказаться и пустой — пробел на ленте считается одной из букв алфавита). Для каждой такой пары в программе должно содержаться не более одной команды. Если команды не оказалось, то работа машины на этом *завершается*. Если команда нашлась, то в ней должны быть указаны: буква, *записываемая* в ячейку под головкой (содержимое остальных ячеек не меняется), *перемещение* головки (оставаться над той же ячейкой, или сдвинуться на одну ячейку вправо или влево), состояние, в которое *переходит* машина.

Перед началом работы машина находится в выделенном *начальном* состоянии, головка расположена над самой левой из непустых ячеек ленты, на ленте, начиная с этой ячейки, записано *исходное слово*, остальные ячейки пусты. Слово, записанное на ленте в момент завершения работы машины (все ячейки слева и справа от него должны быть пусты), считается *результатом* работы машины по данной программе над данным исходным словом.

Это неформальное определение не лишено недостатков, которые довольно легко устранить. Например, понятие бесконечной ленты неконструктивно. Хуже того, на такой ленте невозможно найти ни один из концов слова-результата — за любой группой пустых ячеек могут снова начаться непустые, принадлежащие этому слову. От привлечения этого понятия можно избавиться, считая, что по обоим концам исходного слова на конечной ленте находятся (не входящие в основной алфавит) символы конца слова. Если при сдвиге головки она попадает на такой символ, то лента пополняется пустой ячейкой между этим символом и той частью ленты, на которой записано текущее слово. Над этой пустой ячейкой и устанавливается головка. В каждый момент работы машины лента конечна, сохраняется лишь возможность неограниченно наращивать ее длину (*потенциальная бесконечность* ленты).

Устройство машины Тьюринга, как и многих других абстрактных машин, чрезвычайно примитивно. Поэтому составление для этой машины программ, решающих даже очень простые задачи, оказывается довольно сложным делом. Не видно общего способа строить из таких программ программы решения более сложных задач. Тем не менее эта модель абстрактной машины весьма популярна в традиционных монографиях, статьях и руководствах по теории вычислимости.

Ниже, в разд. 1.7.2, будет рассмотрена абстрактная машина с языком более «высокого» уровня. Этот язык частично основан на идеях и обозначениях, предложенных А. Черчем, которому практически одновременно с А. Тьюрингом и некоторыми другими авторами удалось формализовать понятие алгоритма.

Вскоре выяснилось (было математически строго доказано), что все предложенные схемы абстрактных машин обладают одинаковыми возможностями. Было обнаружено также, что на практике не удается указать вычислительный процесс, который не мог бы быть описан любым из предложенных способов. Это

дало повод высказать так называемый *тезис Черча* о том, что чисто интуитивное понятие алгоритма обрело и строгую математическую формулировку. Сам тезис Черча не может быть строго доказан, поскольку это интуитивное понятие в нем сохраняется.

Среди предложенных формализаций упомянем понятие частично рекурсивной функции, возникшее как результат попыток не только формализовать арифметику, но и *арифметизировать* логику — представить натуральными числами и действиями над ними как утверждения из области теоретической арифметики, так и связанные с ними логические построения.

Машины типа машин Тьюринга, имеющие дело со словами в некотором алфавите, заметно упростили аппарат. В *канонических системах* Э. Поста (1943 г., описаны в [35], гл. 12) и в *нормальных алгорифмах* А. А. Маркова (идея которых возникла еще в 1947 г., если не раньше, см. [31]) слова преобразуются не побуквенно, а по значительно более общей схеме. Это позволило сделать действия над словами и их результаты более обозримыми. Легче стало доказывать различные утверждения о свойствах алгоритмов. Но и этот шаг был не последним.

1.7.2. Язык Лисп

Этот язык был предложен Дж. Мак-Карти через двадцать с лишним лет после появления работ Черча, Тьюринга и других авторов. В нем учтен появившийся к тому времени опыт построения языков программирования для компьютеров. Но в отличие от многих других таких языков в Лиспе используется *функциональный стиль* программирования (подробнее о нем см. разд. 2.5.1), достаточно удобный как для написания алгоритмов решения многих прикладных задач — задач обработки символьных выражений, составляющих основу современной математической нотации, так и для описания абстрактной машины с более обозримыми, чем, например, у машины Тьюринга, и проще составляемыми программами. Кроме того, именно функции, но не в их классическом, теоретико-множественном, а в программистском понимании — как способ вычислить (когда это удается) значение функции при заданном значении аргумента, фактически служат одним из вариантов понятия алгоритма.

Базовый алфавит языка состоит из следующих 40 букв: прописные латинские от A до Z, цифры от 0 до 9, знак '–', ограничители '(', ')' и ','.

Первые 37 из них используются для записи слов, воспринимаемых как *атомы* (неделимые символы) — слова, состоящие из латинских букв и цифр, но начинающиеся обязательно с латинской буквы, а также отдельные цифры и знак '–'. Примеры (записанные без знаков препинания между атомами):

LISP *YEAR*2000 *QUOTE* 7 –

Выражения Лиспа бывают двух типов: атомы и списки. *Списком* называется заключенная в круглые скобки конечная последовательность выражений — *элементов* списка — разделяемых запятыми или пробелами, например:

(*A*, *B*, *C*)
(1 (2 3) (4 (5 6)))

(X)
$(\,)$
$(((HAPPY\ BIRTHDAY\ TO\ YOU)))$

Число элементов списка называется его *длиной*. Например, только что выписанные списки имеют следующие длины: 3, 3, 1, 0, 1 (но не 4!).

В Лиспе допускается большая свобода в записи списков там, где это не приводит к недоразумениям. В частности, под пробелом понимается и перенос продолжения списка на другую строку текста. Например,

$(A,\ B,\ C)$ $(A\ B\ C)$ $(A$ $(A\ B$
B $C)$
$C)$

— это четыре способа записи одного и того же списка. Условимся впредь никогда не пользоваться запятыми для разделения элементов списка.

Пробелы можно добавлять в любом месте списка, но не внутри атомов. Если хотя бы один из двух соседних элементов списка сам является списком, то пробел между ними можно опускать, так что список (1 (2 3) (4 (5 6))) может быть записан так: (1 (2 3) (4 (5 6))) или так: (1(2 3)(4(5 6))).

Будем прибегать к ряду *метаобозначений* (т. е. обозначений внешнего уровня, не входящих в символику самого Лиспа): *atom* (атом), *list* (список), *expr* (выражение), '|' (заменяет русское слово «или»), $\sim$ *sequence* (последовательность объектов вида $\sim$, разделенных пробелами — с учетом сказанного о них выше). В этих обозначениях некоторые из приведенных выше правил записываются так:

expr : *atom* | *list*
list : (*expr sequence*)

Выражения в Лиспе часто обладают *значениями*, которыми могут быть другие выражения. *Формой* называется выражение, которое обладает значением, если оно само и его окружение записаны правильно, или от которого этого хотя бы можно ожидать (если его вычисление завершается). В метаобозначениях связь между формой *form* и ее значением *expr* записывается так:

form $\to$ *expr*

Значение атома может быть обычным, если данный атом — это *константа*, или *функциональным*, если атом используется в программе как *имя функции* (*fname*). Атом еще одного класса — *переменная* (метаобозначение *var*) может иметь значение любого из этих типов, подробнее это будет разъяснено ниже. Символом $\to$ будем пользоваться независимо от типа значения. Начнем с констант. Наиболее употребительные из них — это

$NIL \to NIL$
$T \to T$

имеющие, таким образом, в качестве значений сами себя. Тем же свойством обладают все цифры, а также знак '$-$':

$0 \to 0$ ... $9 \to 9$ $- \to -$

Атом *NIL* трактуется двояко — как синоним пустого списка:

$$() \to NIL \quad \text{и} \quad NIL \to ()$$

и как логическое значение «ложь». Принято считать, что любое выражение, отличимое от *NIL*, представляет логическое значение «истина». Но стандартным изображением этого логического значения служит атом T.

Форма в виде списка — это обращение к функции (*fcall*), имеющее вид

fcall : (*function arg sequence*)
function : *defexpr* | *fname*
arg : *expr*

где *defexpr* — это *определяющее выражение* функции, вид которого вскоре будет описан, *fname* — имя функции, определенной вместе с языком Лисп (*встроенной* в него), или атом, имеющий некоторое определяющее выражение своим функциональным значением, *arg* — выражение, служащее *аргументом* обращения.

Функции бывают *обычными* и *особыми* (*псевдофункциями*). Значение обращения (*function* arg_1 ... arg_n) к обычной функции вычисляется по следующим правилам:

1) поочередно вычисляются значения аргументов $arg_1, \ldots, arg_n$,

2) к этому набору значений применяется функция с определяющим выражением, заданным явно или под именем *fname*.

Часто, особенно в примерах, некоторые значения требуется задать явно, как бы в кавычках. Делается это в виде обращения (*QUOTE expr*) к особой функции *QUOTE*. Значение этой формы вычисляется по правилу:

$$(QUOTE\ expr) \to expr$$

где *expr* в обоих вхождениях — одно и то же выражение. Обычно такая форма записывается в сокращенном виде: *'expr*. Примеры:

$(QUOTE\ X) \to X$
$'(F\ G\ H) \to (F\ G\ H)$

Появление в программе атома X или списка $(F\ G\ H)$ без кавычки означало бы, как было сказано, требование найти значение этого атома или списка. Атомы *NIL*, T, все цифры и знак '$-$' не составляют исключения, но приведенное выше соглашение позволяет записывать эти формы без кавычек.

Функция *QUOTE* — это одна из встроенных функций (см. выше). Но написать нетривиальную программу можно лишь с помощью функций, определенных в ней самой. Вместо «способ вычисления ее значения» можно было бы сказать «функциональное значение ее имени» или «ее семантика».

Начнем с правил вычисления некоторых обычных встроенных функций. Значением единственного аргумента функций *CAR* и *CDR* должен быть непустой список. Функция *CAR* выделяет из этого списка первый элемент, а функция *CDR* — остаток списка после отбрасывания первого элемента. Пусть $X \to (expr_1\ expr_2\ \ldots\ expr_n)$. Тогда:

$(CAR\ X) \to expr_1$
$(CDR\ X) \to (expr_2\ \ldots\ expr_n)$

Например,

$$(CAR\ \ '(Y\ 2\ K)) \to Y$$
$$(CDR\ \ '(Y\ 2\ K)) \to (2\ K)$$

Докапываясь до элементов из глубины списочной структуры, нередко приходится использовать сложные композиции функций CAR и CDR. Для них применяются сокращенные обозначения: $CAAR$ (двукратное обращение к CAR), $CDAR$ (обращение сначала к CAR — внутреннее обращение, а затем к CDR) и т. д. Так, например,

$$(CADAAR\ '(((A\ B)(C\ D))((E\ F)(G\ H))) \to B$$
$$(CADADR\ '(((A\ B)(C\ D))((E\ F)(G\ H))) \to (G\ H)$$

Полезно помнить, что функции CAR, $CADR$, $CADDR$, ... выделяют соответственно 1-й, 2-й, 3-й и т. д. элементы значения своего аргумента (разумеется, если это значение — список достаточной длины).

Функция $CONS$ с двумя аргументами вставляет значение (вид его произволен) своего первого аргумента в начало списка, являющегося значением второго аргумента. Пусть $X \to expr$, $Y \to (expr_1\ \ldots\ expr_n)$. Тогда:

$$(CONS\ X\ Y) \to (expr\ expr_1\ \ldots\ expr_n)$$

Например,

$$(CONS\ 'A\ '(B\ C)) \to (A\ B\ C)$$
$$(CONS\ '(1\ 2)\ '(3\ 4)) \to ((1\ 2)\ 3\ 4)$$
$$(CONS\ 'X\ NIL) \to (X)$$

В последнем примере атом X вставляется в начало пустого списка.

Функции, проверяющие наличие или отсутствие некоторого свойства у набора значений своих аргументов и вырабатывающие, соответственно, значение T или NIL, называются *предикатами*, а обращения к таким функциям — *логическими выражениями* ($bool$). Поскольку за истину принимается не только T, но и любое значение, отличное от NIL, роль предикатов могут исполнить очень многие функции.

Предикат $ATOM$ проверяет — не атом ли значение его аргумента:

$$(ATOM\ \ (CAR\ \ '(A))) \to T$$
$$(ATOM\ \ (CDR\ \ '(A))) \to T$$

(пустой список ( ) считается атомом, поскольку это то же, что NIL),

$$(ATOM\,(CONS\,X\ Y)) \to NIL$$

каковы бы ни были значения X и Y.

Предикат EQ проверяет, совпадают ли между собой два атома, являющиеся значениями его аргументов. Он дает значение NIL (вполне разумное) и в том случае, когда значение только одного из аргументов — список. Но проверить тождественность двух списков этот предикат не может. Примеры:

$$(EQ\ \ T\ \ NIL) \to NIL$$
$$(EQ\ T\ 'T) \to T$$
$$(EQ\ \ (CAR\ '(A\ A))\ (CDR\ '(A\ A))) \to NIL$$

Значение (*EQ expr expr*) равно T, если значение *expr* — атом, и не определено, если это значение — список.

Имя *COND* дано особой функции, предназначенной для разветвления вычислений, точнее — для выбора одного из возможных путей их продолжения. Обращение к *COND* (или *COND-конструкция*) имеет вид:

$$(COND\ (bool_1\ expr_1)\ \ldots\ (bool_n\ expr_n))$$

и вычисляется по следующим правилам:

1) в *ветвях* $(bool_i\ expr_i)$ поочередно, начиная с $i = 1$, вычисляются значения *условий* $bool_i$,

2) если получено значение *NIL*, то переходим к следующей ветви,

3) если значение $bool_i$ отлично от *NIL*, то i-я ветвь считается *выбранной*, и следующие ветви уже не рассматриваются, вычисляется значение выражения $expr_i$ в выбранной ветви, оно становится значением всей *COND*-конструкции,

4) если ни одна ветвь не была выбрана, то за значение конструкции принимается *NIL*.

Обычно последний случай стараются предотвратить и в последней ветви в качестве условия $bool_n$ ставят T. Пример: выражение

$$(COND\ \ ((EQ\ expr\ NIL)\ T)\ (T\ \ NIL))$$

имеет значение T, если значение *expr* равно *NIL*, и значение *NIL* при любом другом значении *expr*. Можно считать, что это — проверка списка *expr* на пустоту, а можно смотреть на него как на отрицание высказывания *expr* в языке Лисп. Однако записано оно в чересчур сложном виде. В данной *COND*-конструкции не нужна вторая ветвь (действует правило 4), а в первой ветви нужное значение вырабатывает уже условие (*EQ expr NIL*). Им то и рекомендуется заменить всю конструкцию.

Упоминавшееся выше определяющее выражение функции (*LAMBDA-конструкция*) имеет вид:

$$(LAMBDA\ (var_1\ \ \ldots\ var_m)\ expr)$$

Оно обозначает функцию, вычисляемую с помощью выражения *expr* (*тела* определяющего выражения) после того, как переменные $var_1\ \ \ldots\ var_m$ (*параметры функции*) получат значения аргументов обращения к этой функции. В обращении к функции вида:

$((LAMBDA\ (var_1\ \ \ldots\ var_m)\ expr)\ arg_1\ \ \ldots\ arg_k)$ должно быть $k \leqslant m$.

Часто требуется одно и то же выражение использовать в программе многократно. Поэтому желательно связать его с именем, под которым оно и будет упоминаться. Определение константы

$$(CSETQ\ atom\ expr)$$

связывает атом *atom* с обычным значением — значением выражения *expr*, а в определении функции

$$(DEFINE\ fname\ defexpr)$$

атом *fname* связывается с явно заданным функциональным значением *defexpr*. Функциональное значение любой встроенной функции не имеет явного представления в языке.

С учетом сказанного выше функцию *NULL*, проверяющую список L на пустоту, можно определить так:

(*DEFINE NULL* (*LAMBDA* (*L*) (*EQ L NIL*)))

Но эта же функция меняет логическое значение высказывания P на противоположное, поэтому может появиться желание дать ей второе имя — *NOT*:

(*DEFINE NOT* (*LAMBDA* (*P*) (*NULL P*)))

Принципиально важным способом задания функций является *рекурсия* — обращение к функции из ее собственного определения. В качестве примера определим рекурсивную функцию *APPEND*, вставляющую в начало списка Y (ее второго параметра) все элементы списка X, сохраняя их порядок:

(*DEFINE APPEND* (*LAMBDA* (*X Y*) (*COND*
((*NULL X*) *Y*)
(*T* (*CONS* (*CAR X*) (*APPEND* (*CDR X*) *Y*))))))

Порядок проверок можно изменить:

(*DEFINE APPEND* (*LAMBDA* (*X Y*) (*COND*
(*X* (*CONS* (*CAR X*) (*APPEND* (*CDR X*) *Y*)))
(*T Y*))))

Определение стало на один атом короче, но на столько же непонятнее.

(*APPEND*
'(*RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS*)
'(*AND THEIR COMPUTATION BY MACHINE*))
→ (*RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS*
AND THEIR COMPUTATION BY MACHINE)

(название первой публикации Дж. Мак-Карти по Лиспу (1960)).

Рекурсия может быть и косвенной, когда определения нескольких функций содержат обращения друг к другу, замыкающиеся в один или несколько циклов. При этом ни одна из них может не обращаться сама к себе прямо. Обращение к функции называется *рекурсивным*, если оно прямо или косвенно делается из определения этой функции.

Во время вычисления формы (*function* arg_1 ... arg_k) *уровнем рекурсии* называется число исполняющихся и еще не завершенных рекурсивных обращений к функции *function*. Согласно этому определению при обращении к нерекурсивной функции уровень рекурсии может быть равен только единице. *Глубиной рекурсии* при обращении к функции с конкретными значениями аргументов называется максимальный уровень рекурсии, достигаемый при этом обращении.

Число k аргументов обращения может быть меньше числа m параметров в определении функции, когда для вычисления функции требуются переменные, которым не обязаны соответствовать никакие аргументы обращения. Эти переменные получают при обращении к функции значение *NIL*.

Пример. Функция *REVERSE* переставляет элементы списка в обратном порядке, X — еще не перевернутая, а Y — перевернутая часть списка, она-то в конце вычислений и становится их результатом.

(*DEFINE REVERSE* (*LAMBDA* (*X Y*) (*COND*
((*NULL X*) *Y*)
(*T* (*REVERSE* (*CDR X*) (*CONS* (*CAR X*) *Y*))))))
(*REVERSE* '(1 (2 3) (4 (5 6)))) → ((4 (5 6)) (2 3) 1)

Но к функции *REVERSE* можно обратиться и с двумя аргументами, тогда она делает нечто большее, чем было сказано:

(*REVERSE* '(*C B A*) '(*D E F*)) → (*A B C D E F*)

Базовым Лиспом, который мы будем далее использовать в качестве абстрактной машины, принято называть набор, состоящий из атомов *NIL*, *T* и некоторых других, функций или конструкций: *QUOTE* (или '), *CAR*, *CDR*, *CONS*, *ATOM*, *EQ*, *COND*, *LAMBDA*, *CSETQ* и *DEFINE* и их композиций (см. [29] и любые другие публикации по Лиспу).

Программа на Лиспе (Лисп-программа) состоит из произвольной последовательности форм — обращений как к функциям и константам базового Лиспа, так и к тем, что были определены в самой программе до обращения к ним. «До обращения» не значит, что обращение к функции (или к константе) не может стоять в тексте программы выше места ее определения — ограничение относится только к временно́й последовательности исполнения определений и обращений.

Хороший стиль программирования на Лиспе предполагает, что в начале программы определяются все необходимые для ее работы функции, а затем с их помощью вычисляются одна или несколько форм. Отступление от этого правила ничего не дает, лишь затрудняя понимание программы.

Базовый алфавит из-за нерасчленимости атомов не следует считать алфавитом программы, написанной на Лиспе. Фактически алфавит программы представлен совокупностью всех атомов, используемых в программе.

Список — это обобщение понятия слова. Списки, составленные только из атомов, полностью аналогичны словам. Возможность включать списки в некоторый список в качестве его элементов придает ему структурированность, иерархичность (см. разд. 1.5.4), не присущую слову согласно его определению, но хорошо отображаемую графами-деревьями. Это также способствует обозримости программ и данных. В то же время на метауровне любой список можно считать словом в базовом алфавите. Такое слово можно при желании записать на ленте машины Тьюринга и подвергнуть его обработке по той или иной программе, например, проверяющей правильность расстановки скобок или даже исполняющей Лисп-программу, представленную этим словом.

Несколько слов об использовании функциональных аргументов, т. е. аргументов, обладающих функциональными значениями. Параметры определяющего выражения не делятся на обычные и функциональные. Большей частью параметр, которому передано функциональное значение, используется в теле определяющего выражения как имя функции. Вот пример:

```
(DEFINE COMPOSITION (LAMBDA (F G X)
  (F (G X)) ))
```

Здесь в общем виде описано, что понимается под композицией двух функций. Так как в этом описании параметры F и G используются в качестве имен функций, то соответствующие им аргументы должны быть функциональными, как, например, в следующих определениях:

```
(DEFINE MCADR (LAMBDA (X)
     (COMPOSITION 'CAR 'CDR X) ))
(DEFINE MCDDR (LAMBDA (X)
     (COMPOSITION 'CDR 'CDR X) ))
(DEFINE MCDDADR (LAMBDA (X)
     (COMPOSITION 'MCDDR 'MCADR X) ))
```

Эти функции действуют точно так же, как *CADR*, *CDDR* и *CDDADR*:

```
(MCADR '(A B C)) → B
(MCDDR '(A B C)) → (C)
(MCDDADR '(A (B C D))) → (D)
```

Но если функциональный аргумент — это не имя встроенной функции, т. е. если его значение имеет явное лисповское представление в виде некоторого определяющего выражения, то с ним можно обращаться и как с обычным выражением — выделять его элементы, исследовать его структуру и т. п. Эта возможность будет использована ниже, в разд. 1.7.5.

1.7.3. Модель арифметики

Убедимся, что целые числа и все основные виды операций над ними могут быть смоделированы средствами базового Лиспа. Каждое натуральное (неотрицательное целое) число n предста́вим в этой модели списком, содержащим n элементов, каждый из которых — это атом 1: ( ) изображает число 0, (1) — число 1, (1 1) — число 2 и т. д. Отрицательное число $-m$ изобразим, включив в начало списка атом '−': (− 1) изображает число −1, (− 1 1) — число −2 и т. д.

Определим сначала арифметические операции над натуральными числами. При принятых соглашениях функция *APPEND* с успехом моделирует сложение натуральных чисел:

```
(DEFINE PLUS (LAMBDA (M N) (APPEND M N)))
```

Функция *DIFFERENCE* с определением

```
(DEFINE DIFFERENCE (LAMBDA (M N) (COND
  ((NULL N) M)
  ((NULL M) (CONS - N))
  (T (DIFFERENCE (CDR M) (CDR N))) )))
```

вычисляет разность натуральных чисел. Ее результат может быть отрицательным целым.

С умножением (функция *TIMES*) все достаточно просто:

```
(DEFINE TIMES (LAMBDA (M N) (COND
  ((NULL N) NIL)
  (T (PLUS (TIMES M (CDR N)) M)) )))
```

Прежде чем заняться делением с остатком, определим арифметический предикат *LESSP*, проверяющий, что значение первого аргумента меньше значения второго

```
(DEFINE LESSP (LAMBDA (M N) (COND
  ((NULL N) NIL)
  ((NULL M) T)
  (T (LESSP (CDR M) (CDR N))) )))
```

Будем считать также, что значение делителя N положительно. В противном случае обе последующие функции (*QUOTIENT* — частное и *REMAINDER* — остаток) вовлекаются в безысходную рекурсию. В *QUOTIENT* потребуется вспомогательная переменная L, где будет накапливаться значение частного:

```
(DEFINE QUOTIENT (LAMBDA (M N L) (COND
  ((LESSP M N) L)
  (T (QUOTIENT (DIFFERENCE M N) N (CONS 1 L))) )))
(DEFINE REMAINDER (LAMBDA (M N) (COND
  ((LESSP M N) M)
  (T (REMAINDER (DIFFERENCE M N) N)) )))
```

которые довольно естественно объединяются в одну:

```
(DEFINE DIVIDE (LAMBDA (M N Q) (COND
  ((LESSP M N) (CONS Q (CONS M NIL)))
  (T (DIVIDE (DIFFERENCE M N) N (CONS 1 Q))) )))
```

Очень просто описывается алгоритм Евклида для нахождения наибольшего общего делителя двух положительных целых чисел:

```
(DEFINE GCD (LAMBDA (M N) (COND
  ((NULL N) M)
  (T (GCD N (REMAINDER M N))) )))
```

Переходя к арифметике целых чисел без ограничений на знаки, запасемся предикатом *MINUSP*, отличающим отрицательные числа от неотрицательных:

```
(DEFINE MINUSP (LAMBDA (P) (COND
  ((NULL P) NIL)
  (T (EQ (CAR P) -)) )))
```

и одноместной функцией *MINUS*, меняющей знак числа на противоположный:

```
(DEFINE MINUS (LAMBDA (P) (COND
  ((NULL P) P)
  ((EQ (CAR P) -) (CDR P))
  (T (CONS - P)) )))
```

Операция *ZPLUS* сложения целых чисел описывается так:

```
(DEFINE ZPLUS (LAMBDA (P Q) (COND
  ((NULL P) Q) ((NULL Q) P)
  ((MINUSP P)
    (COND
      ((MINUSP Q) (MINUS (PLUS (CDR P) (CDR Q))))
      (T (MINUS (DIFFERENCE (CDR P) Q))) ))
  ((MINUSP Q) (MINUS (DIFFERENCE (CDR Q) P)))
  (T (PLUS P Q)) )))
```

После этого уже совсем просто описать способ вычисления разности целых чисел:

```
(DEFINE ZDIFFERENCE (LAMBDA (P Q) (ZPLUS P (MINUS Q))))
```

На этом мы оборвем построение нашей модели. Не должно показаться удивительным, что все эти и многие другие арифметические функции встроены в «настоящий» Лисп, при реализации которого используются все возможности современных компьютеров, начиная с естественного для них представления чисел.

1.7.4. Моделирование машины Тьюринга

Составим Лисп-программу, полностью моделирующую работу машины Тьюринга (см. разд. 1.7.1). Тем самым будет показано, что язык Лисп с теми возможностями, которые были описаны в разд. 1.7.2 и будут использованы в настоящем разделе, является не менее мощной абстрактной машиной, чем другие. Тезис Черча не позволяет ожидать большей мощности, но, чтобы окончательно в этом убедиться, следовало бы промоделировать на одной из этих машин язык Лисп.

Чтобы зафиксировать конкретный вариант машины Тьюринга, в программу следует включить определения двух констант по следующей схеме:

```
(CSETQ ALPHABET '(atom sequence))
```

где атомы — это коды букв алфавита машины в нашей программе, и

```
(CSETQ STATESLIST '(atom sequence))
```

где тем же способом представлены все возможные состояния машины.

Для машины с таким алфавитом и списком состояний можно писать разные программы. Чтобы остановиться на одной из них, в Лисп-программе следует определить еще одну константу, для которой примем следующую структуру:

```
(CSETQ PROGRAM '(instlist sequence))
```

где все команды (инструкции), допустимые в некотором состоянии *state*, собраны в список *instlist* вида

```
(state instrest sequence)
```

а *instrest* (остаток команды) — это список вида

```
(letter newstate newletter headshift)
```

Здесь *state* и *letter* — это текущие состояние и буква под головкой, которые служат для выбора исполняемой команды, *newstate* и *newletter* — новое состояние

и записываемая буква, а *headshift* — направление сдвига головки (*STILL* — стоять на месте, *RIGHT* — сдвиг вправо, *LEFT* — влево).

Программа для машины Тьюринга требует задания исходного состояния и исходного слова — списка букв (элементов списка *ALPHABET*) в том порядке, в каком они записаны на ленте. Условимся, что исходное состояние стоит первым в списке *STATESLIST*. Запуск машины организует функция *TURING*:

```
(DEFINE TURING (LAMBDA (INPUTWORD)
  (FINDSTATE (CAR STATESLIST) NIL
    (CAR INPUTWORD) (CDR INPUTWORD) PROGRAM) ))
```

Функции *FINDSTATE* и *FINDINSTRUCTION* подготавливают очередной шаг. Первая из них по заданному текущему состоянию *STATE*, перевернутому (чтобы все изменения происходили в начале, а не в конце списка) списку *LEFTPART* содержимого ленты слева от головки, букве *LETTER* под головкой, списку *RIGHTPART* букв ленты справа от головки и еще не просмотренной части *PROGRAMREST* программы находит в этой части список инструкций для текущего состояния. В случае неуспеха этого поиска работа машины завершается, а функция *RESULT* компонует из частей ленты результирующее слово.

```
(DEFINE FINDSTATE
  (LAMBDA (STATE LEFTPART LETTER
      RIGHTPART PROGRAMREST)
    (COND
      ((NULL PROGRAMREST)
        (RESULT LEFTPART LETTER RIGHTPART) )
      ((EQ STATE (CAAR PROGRAMREST))
        (FINDINSTRUCTION STATE LEFTPART LETTER
          RIGHTPART (CDAR PROGRAMREST) ) )
      (T (FINDSTATE STATE LEFTPART LETTER
        RIGHTPART (CDR PROGRAMREST) )) )))
```

Функция *FINDINSTRUCTION* ищет в списке команд *INSTRUCTIONSLIST* для состояния *STATE* команду, соответствующую букве *LETTER*. Если команда нашлась, то происходит обращение к функции *STEP* для исполнения очередного шага работы машины. При неудаче вызывается функция *RESULT* — работа программы завершена.

```
(DEFINE FINDINSTRUCTION
  (LAMBDA (STATE LEFTPART LETTER RIGHTPART
      INSTRUCTIONSLIST)
    (COND
      ((NULL INSTRUCTIONSLIST)
        (RESULT LEFTPART LETTER RIGHTPART) )
      ((EQ LETTER (CAAR INSTRUCTIONSLIST))
        (STEP (CADAR INSTRUCTIONSLIST) LEFTPART
          (CADDAR INSTRUCTIONSLIST) RIGHTPART
          (CADDDAR INSTRUCTIONSLIST) ) )
```

```
(T (FINDINSTRUCTION STATE LEFTPART LETTER
   RIGHTPART (CDR INSTRUCTIONSLIST) )) ))))
```

Функция *STEP* записывает под головкой новую букву *NEWLETTER*, сдвигает, если значение параметра *SHIFT* этого требует, головку вправо или влево и переводит машину в состояние *NEWSTATE*. При этом, если часть ленты в направлении сдвига представлена пустым списком, то под головкой размещается пустая клетка, как бы выделенная из этого списка, остающегося пустым. Условимся, что в списке *ALPHABET* соответствующая буква стоит первой.

```
(DEFINE STEP (LAMBDA (NEWSTATE LEFTPART
    NEWLETTER RIGHTPART SHIFT)
  (COND
    ((EQ SHIFT 'STILL)
      (FINDSTATE NEWSTATE LEFTPART NEWLETTER
        RIGTHPART PROGRAM) )
    ((EQ SHIFT 'RIGHT)
      (FINDSTATE NEWSTATE (CONS NEWLETTER LEFTPART)
        (COND
          ((NULL RIGHTPART) (CAR ALPHABET))
          (T (CAR RIGHTPART)) )
        (COND ((NULL RIGHTPART) NIL) (T (CDR RIGHTPART)))
        PROGRAM) )
    ((EQ SHIFT 'LEFT)
      (FINDSTATE NEWSTATE
        (COND ((NULL LEFTPART) NIL) (T (CDR LEFTPART)))
        (COND
          ((NULL LEFTPART) (CAR ALPHABET))
          (T (CAR LEFTPART)) )
        (CONS NEWLETTER RIGHTPART) PROGRAM) ) )))
```

Функция *RESULT* совсем проста по сравнению с предыдущими:

```
(DEFINE RESULT (LAMBDA (LEFTPART LETTER RIGHTPART)
  (CONS 'MAKES
    (REVERSE LEFTPART (CONS LETTER RIGHTPART)) ) ))
```

В качестве примера приведем слегка упрощенную программу сложения двух натуральных чисел из [34] (§ 5.2, пример 4). В данном варианте число $m \geqslant 0$ представлено участком ленты, на котором записано слово из m букв *ONE*. Итак:

```
(CSETQ ALPHABET '(ZERO ONE AST)),
```

где буква *ZERO* представляет пустую клетку, *AST* — разделитель слагаемых,

```
(CSETQ STATESLIST '(Q0 Q1 Q2)).
```

В начальном состоянии $Q0$ разыскивается и заменяется на *ONE* разделитель *AST*, в состоянии $Q1$ разыскивается правый конец ленты, в состоянии $Q2$ лишняя единица, возникшая в результате замены *AST*, стирается и работа программы завершается.

```
(CSETQ PROGRAM
  '((Q0 (ONE Q0 ONE RIGHT) (AST Q1 ONE STILL))
    (Q1 (ONE Q1 ONE RIGHT) (ZERO Q2 ZERO LEFT))
    (Q2 (ONE Q2 ZERO STILL)) ) )
```

Результаты нескольких обращений к программе *TURING* с различными вариантами исходного слова:

```
(TURING '(ONE ONE ONE ONE AST)) →
    (MAKES ONE ONE ONE ONE ZERO ZERO)
```

Одна из двух пустых клеток за концом слова-результата появилась при сдвиге за правый конец текущего слова в состоянии $Q1$, вторая — при замене *ONE* на *ZERO* в состоянии $Q2$.

```
(TURING '(ONE ONE ONE AST ONE)) →
    (MAKES ONE ONE ONE ONE ZERO ZERO)

(TURING '(AST ONE ONE ONE ONE)) →
    (MAKES ONE ONE ONE ONE ZERO ZERO)

(TURING '(AST)) → (MAKES ZERO ZERO)
```

1.7.5. Семантика рекурсивных функций

Идея *денотационной*, или *математической*, семантики языка программирования выглядит просто — описать результат исполнения программы в целом и ее частей не в терминах работы соответствующей абстрактной машины, а с помощью традиционных математических средств. В то же время желательно, чтобы эти средства оставались *финитными*, т. е. имели бы дело лишь с поддающимися построению объектами и с обозримыми, поддающимися описанию, множествами таких объектов, отображениями таких множеств одно в другое и т. п. Имеется в виду описание на естественном языке, но не допускающее двух толкований.

Для реальных компьютеров сделать это нетрудно в силу их конечности. Для абстрактных машин ситуация несколько сложнее, поскольку любая из них воплощает идею *потенциальной бесконечности*. Например, в Лиспе множество атомов в принципе бесконечно: хотя в любой программе используется лишь конечный набор атомов, но всегда сохраняется возможность пополнить его. Рассмотренная выше модель арифметики для любого натурального числа n допускает переход от списка, изображающего это число, к списку, изображающему число $n+1$. При описании модели машины Тьюринга в разд. 1.7.4 было рассказано, как с помощью конечных списков моделируется бесконечная в обе стороны лента.

В Лисп-программе любой список или выражение конечны, а множество этих объектов потенциально бесконечно. Потенциально бесконечное множество атомов служит, как уже было сказано, моделью бесконечного (также потенциально) алфавита, а списки, состоящие только из атомов, — это не что иное, как слова в этом алфавите. Списки общего вида моделируют иерархические структуры с

произвольной, но конечной глубиной иерархии, например, математические формулы. Любой внутренний список представляет часть или подформулу формулы, представленной объемлющим списком. Сказанное означает, что почти все средства базового Лиспа не выходят за рамки финитной математики.

Более того, к указанной аналогии и сводится денотационная семантика языка Лисп. Предикат *ATOM* отличает буквы от выражений (термов, формул, выводов и пр.), предикат *EQ* распознает тождественные буквы, функции *CAR* и *CDR* отделяют начало — букву или подвыражение — текущего читаемого выражения от его остатка. Функция *CONS* моделирует процесс записи структурированного текста — написать его первый элемент, а потом все остальное. Функция *COND* осуществляет последовательный конечный перебор возможных случаев, типичный для доказательств математических утверждений. Функция *LAMBDA* выделяет в лисповском выражении некоторые атомы в качестве имен параметров, превращая его в описание алгоритма, а *DEFINE* закрепляет за этим алгоритмом имя. Псевдофункция *QUOTE* «закавычивает» выражение, позволяя явно передавать его в качестве значения. Эта семантика легко распространяется на любые композиции базовых функций и на все нерекурсивные — никогда, ни прямо, ни косвенно, не обращающиеся к самим себе — функции. В принципе эта база мало отличается, например, от аппарата примитивно рекурсивных функций в теоретической арифметике (см. [34], § 3 главы 3). Громоздкость и неудобства этого аппарата поражают воображение программиста. Но математика-профессионала увлекает возможность интерпретации логики в терминах арифметики.

Финитным и укладывающимся в рамки описанной выше семантики можно считать определение таких рекурсивных функций, для которых предельная глубина рекурсии легко определяется. Например, для функции *APPEND* из раздела 1.7.2 она совпадает с длиной первого аргумента. Иерархичность структуры текста позволяет на каждом уровне просматривать его более крупными кусками. Общая сложность просмотра при этом не уменьшается.

Почти так же просто подсчитывается число обращений к функции *PLUS* из разд. 1.7.3, возникающих на всех уровнях рекурсии при обращении к *TIMES*.

Если, как в последнем примере, учитываются обращения ко всем вовлекаемым в рекурсию функциям, то ограниченность суммарной глубины рекурсии при вычислении функции F означает возможность *статически* — исходя из текста программы и входных данных, без ее фактического исполнения — установить, что исходное обращение к F завершается за конечное время. В таких случаях функцию F характеризуют, говоря, что она частична, но область ее определения *задана эффективно*.

А можно ли, а если можно, то как, установить ограниченность глубины рекурсии для произвольной рекурсивной функции F при произвольном значении ее аргумента? В теории вычислимости этот вопрос занимает одно из центральных мест, поскольку именно здесь финитность оказывается под угрозой.

Для ответа на него опишем две вспомогательные функции. Функция

```
(DEFINE CATCH (LAMBDA ( ) (COND
  (NIL T) (T (CATCH)))))
```

увязает в цикле рекурсивных обращений навечно, а функция

$$(DEFINE\ BREAK\ (LAMBDA\ ()\ (COND\ (NIL\ (BREAK))\ (T\ T))))$$

немедленно вырывается со значением T из этого цикла.

Предположим, что может быть описана функция $STOPSP$ с одним параметром, которому при обращении к ней соответствует аргумент $expr$, вырабатывающая значение T, если вычисление $expr$ благополучно завершается, и значение NIL — в противном случае. Так, например, значением $(STOPSP\ '(CATCH))$ должно быть NIL, а значением $(STOPSP\ '(BREAK))$ — T. В традиционной теории вычислимости доказывается, что сделанное предположение приводит к противоречию.

Идея доказательства такова. Описывается функция, назовем ее $TRICK$, которая с помощью $STOPSP$ предсказывает исход обращения к ней самой. Далее вычисление направляется по ветви, состоящей из обращения к $CATCH$, если предсказано успешное завершение, или по ветви с обращением к $BREAK$ — при противоположном предсказании. В обоих случаях поведение ветвей и функции в целом противоречит предсказанию, которое оказывается ошибочным. Итак, из сделанного предположения о свойствах функции $STOPSP$ логически выводится тождественно ложный результат:

$$\neg((STOPSP\ '(TRICK)) \to T)\ \wedge \neg\neg((STOPSP\ '(TRICK)) \to T)$$

что приводит к выводу о невозможности существования функции $STOPSP$ с требуемым свойством.

Это доказательство можно было бы считать безупречным, если бы оказалось возможным описать функцию $TRICK$, делающую в точности то, что о ней сказано выше. По-видимому, это описание должно выглядеть так:

$$(DEFINE\ TRICK\ (LAMBDA\ ()\ (COND\ ((STOPSP\ '(TRICK))\ (CATCH))\ (T\ (BREAK)))))$$

Каково значение входящего сюда выражения $(STOPSP\ '(TRICK))$? Действуя в лоб, можно попытаться заменить это выражение на $(TRICK)$, поскольку значением $TRICK$ при успешном завершении вычисления может быть только T. Но в действительности никакой результат вообще не может быть получен, так как такое обращение к $TRICK$ вязнет уже в первом условии $COND$, так что ни одна из ветвей не может быть выбрана.

Однако не помогают и любые косвенные подходы. Относительно функции $STOPSP$ следует рассмотреть две возможности: вырабатывает ли она свое значение статически, лишь анализируя текст аргумента $expr$, или динамически, пытаясь запустить вычисление $expr$ и проследить за ходом вычисления. Но при аргументе $'(TRICK)$ это приведет к бесконечному повторению попыток возобновить вычисление $(STOPSP\ '(TRICK))$ — так в соответствии с идеологией диагонального метода устроена функция $TRICK$.

Предположение, что результат может быть выяснен статически, должно быть отвергнуто — именно оно позволяет вывести тождественно ложное высказывание. А бесконечная рекурсия при вычислении $(STOPSP\ '(TRICK))$ губит

замысел, заключающийся в получении результата, вступающего в противоречие с ответом на поставленный вопрос. Задать пресловутый вопрос так, чтобы получить на него какой-нибудь ответ, оказывается невозможным (парикмахер носит бороду и вопрос — сам ли он бреется или нет, теряет смысл).

Коварство замысла — составить функцию *TRICK* так, чтобы она вела себя вопреки сделанному предположению, — здесь ни при чем. Если вызовы *CATCH* и *BREAK* поменять местами, сделав поведение функции соответствующим предсказанию, это никак не отразится на возможности, вернее — на невозможности, осуществить само предсказание. Любая функция, обращающаяся сама к себе для выработки суждения о наличии у нее некоторого свойства, не защищена от зацикливания.

Таким образом, диагональный метод приведения к противоречию, столь заманчивый в теории, на практике не сработал. Вспоминаются разногласия между двумя собеседниками в конце разд. 1.5.8 по поводу того же метода. Однако в литературе он остается основным способом доказательства утверждений о несуществовании алгоритмов решения многих задач, называемых на этом основании *алгоритмически неразрешимыми*.

Замечание. Утверждение о диагональном методе как об основном при доказательстве алгоритмической неразрешимости массовых проблем несколько преувеличено. В действительности непосредственно этим методом доказывается неразрешимость двух-трех проблем, которые, как оказывается, могут быть сведены ко множеству других, неразрешимых в силу этой сводимости.

Рассмотрим в более или менее общем виде проблему распознавания свойств функций. Ограничимся функциями с одним параметром и будем говорить о свойствах пар вида (F, A), где F — функция, A — аргумент обращения к ней, поскольку для аргументов разных классов функция F может вести себя совершенно по-разному.

Свойство P *нетривиально*, если можно указать и пару (G, B), обладающую этим свойством, и пару (H, C), не обладающую им. Свойство P *инвариантно*, если любое условное выражение вида (*COND* (*expr expr1*) (*T expr2*)) обладает или не обладает им вместе с той ветвью (*expr*1 или *expr*2), которая выбирается при вычислении этого выражения. Так, инвариантно свойство завершаемости вычисления или свойство вырабатывать значение *NIL*, поскольку вычисление выражения сводится к вычислению выбранной ветви.

Свойство P *распознаваемо*, если может быть описана всюду определенная функция S с аргументами F и A, вырабатывающая значение, отличное от *NIL*, если пара (F, A) обладает свойством P, и значение *NIL* в противном случае. Теорема Райса (или Успенского-Райса, см., например, [31], § 56) утверждает, что любое инвариантное нетривиальное свойство нераспознаваемо, т. е. что функцию S с указанными возможностями описать нельзя. Ее традиционное доказательство строится по той же диагональной схеме. В предположении, что распознающая функция S существует, описывается функция D:

(*DEFINE D* (*LAMBDA* (*X*) (*COND* ((*S X X*) (*H C*))(*T* (*G B*)))))

Если функция S всюду определена, то она должна работать и в случае, когда X — это описание некоторой функции, которое может быть использовано и в качестве функционального аргумента (первый аргумент обращения к S). Попытка выяснить наличие свойства P у пары $(D,\ D)$ приводит к противоречию: эта пара обладает свойством P тогда и только тогда, когда она в силу требований, предъявляемых к S, не обладает им.

В этом доказательстве молчаливо предполагается, что функция S вырабатывает свое значение статически — лишь на основании анализа текста аргументов F и A. Это может быть сделано во многих случаях — практически для всех задач на программирование, предлагаемых студентам младших курсов, не говоря о школьниках. Таким образом доказано, что не для всех пар $(F,\ A)$ наличие или отсутствие свойства P можно установить статически — для пары $(D,\ D)$ сделать это невозможно.

Но в запасе есть резерв. В теории вычислений (см., например, теорему о временно́й иерархии в [43]) доказывается, что в общем случае распознать наличие свойства P у пары $(F,\ A)$ можно только динамически — прослеживая весь процесс вычисления значения F при аргументе A. Подробно это будет обосновано в разд. 3.2.4 и 3.2.5. Чтобы проследить ход процесса, надо его запустить, сопровождая основные действия, предусмотренные алгоритмом вычисления F, действиями по выявлению свойства P.

В случае, когда процесс запускается для пары $(D,\ D)$, происходит уже знакомое нам зацикливание: вычисление (лисповского) выражения $(D\ D)$ начинается с вычисления условия $(S\ D\ D)$, которое требует повторного запуска в вычисление выражения $(D\ D)$ и т. д. Зацикливание именно в этом месте — при вычислении условия $(S\ D\ D)$ — означает, что до выбора ветви: $(H\ C)$ или $(G\ B)$ дело не доходит, а следовательно, и противоречие, опровергающее возможность существования функции S, не возникает.

Зацикливание появилось в результате самоприменения функции D (ее применения к собственному описанию). А это самоприменение — неустранимый, по-видимому, элемент диагонального метода. По этому поводу М. Минский в [35] замечает: «Мы не касаемся в данном случае вопроса о том, чем интересны такие вычисления “с самоанализом”; все же нет ничего абсурдного в ситуации, когда человек размышляет над описанием своего собственного мозга». Пусть так, но абсурдно предположение, что мозг человека поможет ему до конца разобраться в этом описании, если оно исчерпывающе полно.

В традиционной теории вычислений проблема самоприменимости функции выдвигается на первый план среди всех массовых проблем распознавания. Функция называется *самоприменимой* или *несамоприменимой* в зависимости от ее применимости к собственному описанию (или номеру в некоторой нумерации). Ставится задача: построить функцию K, применимую лишь к описаниям любых несамоприменимых функций (вариант парадокса Рассела). Невозможность построения доказывается тем же диагональным методом.

Предположение, что K самоприменима, т. е. применима к своему описанию K', означало бы, в силу требования к K, что K' — это описание несамоприме-

нимой функции. Возникшее противоречие приводит к выводу о несамоприменимости K. А здесь противоречие выйти на поверхность уже не может. Несамоприменимость K означает бесконечное ожидание результата применения K к K', т. е. невозможность получить второй исход предсказания. Так и перед расселовским парикмахером вопрос: брить или не брить самого себя в качестве клиента, — не стоит и встать никогда не может.

Любопытно, что в [31] (замечание к § 47.2.1) отмечается недопустимость слишком большой свободы в рамках теоретико-множественной концепции — нельзя, не впадая в противоречия, любым способом объединять 'объекты' в 'множества', которые сами будут рассматриваться как 'объекты'. Однако вопрос о том, можно ли без всяких ограничений строить 'слова' — описания 'алгорифмов' и подавать такие 'слова' на вход любых алгорифмов, почему-то не возникает.

Может быть, и в рамках теоретико-алгорифмического подхода свобода может оказаться чрезмерной? Если, например, нормальный алгорифм написан в расчете на входное слово, состоящее из двух частей с некоторым разделителем между ними, то каковы основания рассматривать возможность его применения к слову, не содержащему такого разделителя? (Иначе говоря, не следовало бы разрешать обращаться с одним аргументом к функции двух параметров — это простейшее ограничение на типы, естественно учитывать и более сложные.)

Говорят, что некоторое свойство объекта *полуразрешимо*, если возможна функция, выявляющая за конечное время наличие у объекта этого свойства, но зацикливающаяся при его отсутствии, или наоборот. Из «диагонального» доказательства несуществования всюду определенного разрешающего алгорифма S следует, что для функций, подобных D, невозможно и полуразрешение. Однако существование объекта с некоторым свойством вполне может оказаться неразрешимым или полуразрешимым свойством самого свойства.

По-видимому, неразрешимые свойства существуют. Но выявление таких свойств оказывается трудной математической задачей, а диагональный метод при этом — плохим помощником. Поэтому будем ниже говорить не об алгоритмической неразрешимости, а о *неразрешимости стандартными средствами*.

Взглянем на ту же проблему с еще одной точки зрения, которая, впрочем, приведет нас к почти тому же самому результату.

1.7.6. Теория неподвижной точки

Механизм рекурсивного определения функций, принятый в Лиспе, может показаться недостаточно строгим по сравнению с традиционным для теории вычислений подходом.

Изложим его основные черты. Пусть X — произвольное множество, $S = \{s \mid s : X \to X\}$ — множество всех частичных функций на множестве X, $s0$ — пустая функция (ее область определения $\mathrm{Dom}(s0) = \emptyset$).

Будем рассматривать множество функций

$$H = \{h \mid h : U \to S\},$$

всюду определенных на произвольном множестве U.

Полезно заранее иметь в виду, что в дальнейшем (разд. 3.4) будет рассматриваться интерпретация, в которой U — это множество программных единиц u (например, операторов и выражений) некоторого языка программирования; X — множество состояний w программы, так что S — множество семантик s (правил преобразования состояний); H — семантика языка, набор семантических функций h, сопоставляющих единицам их семантики. Цель построения теории — создать метод распространения множества H, первоначально определенного лишь для нерекурсивных единиц, на рекурсивные.

В случае языка Лисп под U следует понимать множество всех форм, под w — множество форм с уже вычисленными значениями, а под X — множество всех выражений (в частности, форм и их значений), так что S — это множество правил вычисления значений форм. Основу для построения семантик рекурсивных функций составляют семантики функций базового Лиспа и функций, определимых в нем без рекурсии или со статически оцениваемой глубиной рекурсии. В разд. 1.7.5 пояснялось, почему семантику таких функций можно считать финитной.

Обозначим через $h0$ функцию, ставящую в соответствие каждому элементу, которым пополняется множество U, функцию $s0$. Ясно, что $h0 \in H$.

Пусть $s1, s2 \in S$, $h1, h2 \in H$. Определим отношения

$$s1 \sqsubseteq s2 \equiv \forall w\,(w \in \mathrm{Dom}(s1) \supset s1(w) = s2(w)), \tag{1}$$

т. е. $s2$ — это расширение $s1$,

$$h1 \subseteq h2 \equiv \forall u\,(u \in U \supset h1(u) \sqsubseteq h2(u)). \tag{2}$$

Легко убедиться, что отношения $\sqsubseteq$ на S и $\subseteq$ на H — это частичные порядки, для которых соответственно $s0$ и $h0$ — наименьшие элементы.

Пусть φ — отображение типа $H \to H$ (оператор над H). *Неподвижной точкой* оператора φ называется функция h, удовлетворяющая уравнению $h = \varphi(h)$. Чтобы ее найти, можно использовать метод последовательных приближений. Для этого определим последовательность $\langle h_n \rangle$, *порождаемую* оператором φ:

$$h_0 = h0, \quad h_{n+1} = \varphi(h_n) \text{ для } n = 0, 1, \ldots$$

Отображение φ *монотонно* по порядку $\subseteq$, если для любых $h1, h2 \in H$:

$$h1 \subseteq h2 \supset \varphi(h1) \subseteq \varphi(h2), \tag{3}$$

т. е. если φ сохраняет порядок $\subseteq$. Содержательно: если функция $h2(u)$ определена на более обширном множестве значений w, чем $h1(u)$, то и $\varphi(h2)(u)$ определена шире, чем $\varphi(h1)(u)$.

Теорема 1.7.1. *Пусть оператор $\varphi : H \to H$ монотонен. Тогда*

$$\forall n\,(h_n \subseteq h_{n+1}),$$

т. е. последовательность $\langle h_n \rangle$ монотонна по порядку $\subseteq$.

Доказательство (индукция по n). База индукции $h_0 \subseteq h_1$ очевидна. Пусть уже установлено, что $h_{n-1} \subseteq h_n$. Тогда $\varphi(h_{n-1}) \subseteq \varphi(h_n)$ (по монотонности φ), т. е. $h_n \subseteq h_{n+1}$. $\lhd$

Замечание. Функция h в отличие от каждой из h_n описывает свойство оператора φ в целом, а не отдельного случая его применения. Если о решении уравнения $h = \varphi(h)$ ничего не известно заранее, то именно для монотонного оператора есть смысл принять за начальное приближение h_0 функцию $h0$ и надеяться, что каждый очередной элемент h_n последовательности, который не меняет значений функции для тех значений аргумента u, где была определена функция h_{n-1}, и, возможно, доопределяет ее для новых значений, будет ближе нее к искомому решению задачи, а в идеале совпадет с ним при некотором n.

Любая последовательность $\langle g_n\rangle$, где $g_n \in H$, обладающая свойством монотонности: $\forall\, n\,(g_n \subseteq g_{n+1})$, называется *цепью* (над H). Теорема 1.7.1 означает, что последовательность $\langle h_n\rangle$ — это цепь, а $\langle\varphi(h_n)\rangle$ — та же самая цепь, но без начального элемента h_0.

Пусть последовательность $\langle g_n\rangle$ — это цепь над H. *Наименьшей верхней гранью* (Least Upper Bound) цепи $\langle g_n\rangle$ называется соответствие $\mathrm{LUB}(g) : U \to S$, такое что:

$$\forall\, w \forall\, w_1\,(\mathrm{LUB}(g)(u)(w) = w_1 \;\equiv\; \exists\, n\,(g_n(u)(w) = w_1)) \tag{4}$$

для любого $u \in U$.

Формально $\mathrm{LUB}(g)(u)$ определена как соответствие, возможно многозначное. Убедимся, что $\mathrm{LUB}(g)(u)$ — это функция и, следовательно, $\mathrm{LUB}(g)$ принадлежит H. Пусть $u \in U$.

Возможны два случая:

а. $\forall\, n\,(\mathrm{Dom}(g_n(u)) = \emptyset)$, при этом функция $\mathrm{LUB}(g)(u)$ определена однозначно — это $s0$.

б. Пусть для каких-то w, w_1 и w_2 имеет место $\mathrm{LUB}(g)(u)(w) = w_1$, а также $\mathrm{LUB}(g)(u)(w) = w_2$. Это значит, что найдутся такие $n1$ и $n2$, что $g_{n1}(u)(w) = w_1$, $g_{n2}(u)(w) = w_2$ и, например, $n1 \leqslant n2$. Тогда: $g_{n1} \subseteq g_{n2}$ — по монотонности цепи $\langle g_n\rangle$; $g_{n1}(u) \sqsubseteq g_{n2}(u)$ — по определению (2) отношения $\subseteq$, $w_1 = w_2$ — по определению (1) отношения $\sqsubseteq$.

Итак, соответствие $\mathrm{LUB}(g)(u)$ однозначно в своей области определения, а так как u — произвольный элемент U, то $\mathrm{LUB}(g)$ — это функция из H.

Обозначение $\mathrm{LUB}(g)$ отвечает функциональному взгляду на $\langle g_n\rangle$. Теоретико-множественной трактовке функций g_n (которой мы следовали в только что завершенном доказательстве) лучше отвечает другое обозначение — $\bigcup_{n\geqslant 0} g_n$ — того же объекта. Можно также считать $\bigcup_{n\geqslant 0} g_n$ пределом последовательности $\langle g_n\rangle$, что дает основания принять следующее определение.

Монотонный оператор φ называется *непрерывным*, если для соответствующей ему последовательности $\langle h_n\rangle$ имеет место $\varphi(\bigcup_{n\geqslant 0}\ h_n) = \bigcup_{n\geqslant 0}\ \varphi(h_n)$.

Теорема 1.7.2. *Если φ — непрерывный оператор, а $\langle h_n\rangle$ — соответствующая ему цепь, то*

$$LUB(h) = \varphi(LUB(h)), \tag{5}$$

т. е. $LUB(h)$ — неподвижная точка оператора φ, а также

$$\forall\, h'\,(h' = \varphi(h') \;\supset\; LUB(h) \subseteq h'), \tag{6}$$

т. е. из всех неподвижных точек этого оператора точка $LUB(h)$ — наименьшая.

Доказательство. Оператор φ непрерывен, поэтому

$$\varphi(\bigcup_{n\geqslant 0} h_n) = \bigcup_{n\geqslant 0} \varphi(h_n).$$

Продолжаем цепочку равенств: $\bigcup_{n\geqslant 0} \varphi(h_n) = \bigcup_{n\geqslant 1} h_n$, и так как $h_0 \subseteq h_1$, то $\bigcup_{n\geqslant 1} h_n = \bigcup_{n\geqslant 0} h_n$.

Итак, $\varphi(\mathrm{LUB}(h)) = \mathrm{LUB}(h)$, т. е. утверждение (5) доказано.

Пусть функция $h' \in H$ такова, что $h' = \varphi(h')$. Индукцией по n докажем, что для всех n имеет место $h_n \subseteq h'$. База индукции — $h_0 \subseteq h'$ — очевидна. Допустим, что $h_n \subseteq h'$ при некотором n. Непрерывность φ предполагает его монотонность, так что $\varphi(h_n) \subseteq \varphi(h')$, т. е. $h_{n+1} \subseteq h'$. Утверждение доказано.

Пусть $u \in U$ и $s = \mathrm{LUB}(h)(u)$. Тогда, по определению $\mathrm{LUB}(h)$, либо $s = s0$, при этом $s \sqsubseteq h'(u)$ при всяком h', либо для любого w из $\mathrm{Dom}(s)$ найдется такое n, что $s(w) = h_n(u)(w)$, при этом $w \in \mathrm{Dom}(h_n(u))$. Из доказанного выше утверждения следует $h_n(u) \sqsubseteq h'(u)$ — по определению (2) и $h_n(u)(w) = h'(u)(w)$ — по определению (1), иначе: $s(w) = h'(u)(w)$. В силу того же определения (1) и в этом случае $s \sqsubseteq h'(u)$. Но тогда $\mathrm{LUB}(h) \subseteq h'$ согласно определению (2). Справедливость формулы (6) подтверждена. $\lhd$

Замечания. 1. Непрерывность оператора φ была использована только в первой части этого доказательства — при установлении истинности равенства (5). Во второй части потребовалась лишь его монотонность.

2. Теоремы 1.7.1 и 1.7.2 отражают основное содержание «первой теоремы о рекурсии» и «теоремы о рекурсии» Клини ([21], § 66) и построений, использованных при их доказательстве.

Изложенный подход рассмотрим на типичном для Лиспа описании рекурсивной функции F:

```
(DEFINE F (LAMBDA (X) (COND
  ((C X) (F (B X)))
  (T (A X)))))
```

где A, B и C — это функции с уже определенной семантикой или даже просто композиции базовых функций Лиспа. Воспользуемся *LAMBDA*-конструкцией из этого определения, чтобы построить оператор ψ, преобразующий одну нерекурсивную функцию G в другую — $H = \psi(G)$:

```
(DEFINE H (LAMBDA (X) (COND
  ((C X) (G (B X))) (T (A X)))))
```

При этом функция F удовлетворяет уравнению $F = \psi(F)$, т. е. оказывается неподвижной точкой оператора ψ. Этот оператор порождает потенциально бесконечную последовательность *нерекурсивных* функций F_n, $n \geqslant 0$:

```
(DEFINE F_n (LAMBDA (X) (COND
  ((C X) (F_{n-1} (B X))) (T (A X)))))
```

За F_0 примем ограничение F множеством тех значений X, для которых $(C\ X) \rightarrow NIL$. При этом первая ветвь *COND* блокируется, так что описание функции F_0 можно свести к виду:

```
(DEFINE F_0 (LAMBDA (X) (A X)))
```

В силу сказанного в разд. 1.7.5 семантику функций F_n можно считать финитной.

Лемма 1.7.3. *Оператор ψ монотонен, а функция F — наименьшая из его неподвижных точек.*

Доказательство. Определим последовательность $\langle X_0, X_1, X_2, \ldots\rangle$ (конечную или бесконечную) следующим образом. За X_0 примем значение X, с которым произошло начальное (или «внешнее» — на *нулевом* уровне рекурсии) обращение к функции F.

Установим, что функции F и F_n одинаково вычисляют начальный отрезок $\langle X_0, X_1, \ldots, X_k\rangle$ этой последовательности до $k = n$.

Пусть X_k уже определено. Если $k = n$, то, как будет видно из последующего, $(F_n\ X_0) = (F_0\ X_n)$, так что вычисление F_n завершается со значением $(A\ X_n)$, но вычисление F, где обращение к *COND* сохранено, продолжается.

Возможны следующие случаи (для F_n — при $k < n$):

а. $(C\ X_k)$ не определено, тогда не определены как $(F\ X_0)$, так и $(F_n\ X_0)$ (оно же, как будет видно, $(F_{n-k}\ X_k)$ для $1 \leqslant k \leqslant n$);

б. $(C\ X_k) \to NIL$, $(A\ X_k)$ не определено, тогда $(F\ X_0)$ и $(F_n\ X_0)$ также не определены;

в. $(C\ X_k) \to NIL$, $(A\ X_k) \to Y$ — единственный случай, когда вычисление $(F\ X_0)$ и $(F_n\ X_0)$ завершается, притом с одним и тем же значением Y;

г. $(C\ X_k) \to T$, $(B\ X_k)$ не определено, тогда и $(F\ X_0)$, и $(F_n\ X_0)$ не определены;

д. $(C\ X_k) \to T$, $(B\ X_k) \to Y$, тогда полагаем $X_{k+1} = Y$; при вычислении F значение $(F\ X_0)$ определяется по прежней схеме, но уже при известном значении X_{k+1}; при вычислении F_n значение $(F_n\ X_0)$ определяется как значение (индукция по k) $(F_{n-1}\ X_1) = \ldots = (F_{n-k}\ X_k) = (F_{n-k-1}\ X_{k+1})$ по одной и той же схеме для всех этих значений.

Итак, установлено, что F_n — это ограничение F множеством тех значений X, для которых глубина рекурсии при вычислении $(F\ X)$ не превышает n. Для F_0 это следует из того, что рекурсивное обращение заблокировано, а для F_n при $n > 0$ — из совпадения процессов вычисления F и F_n до завершения второго из них. Вместе с равенством $(F_n\ X_0) = (F_{n-k}\ X_k)$ (см. п. «д») это доказывает монотонность оператора ψ.

Вычисление F может завершиться (случай «в») только при завершении вычисления F_n для некоторого n. Это означает, что $F = \bigcup_{n \geqslant 0} F_n$.

То, что неподвижная точка F оператора ψ — наименьшая, следует из второй части доказательства теоремы 1.7.2, где непрерывность оператора не используется. $\lhd$

Если заранее известно, что необходимая глубина рекурсии ограничена или определяется статически, то тем самым обеспечено наступление случая «в» с завершением этого процесса. Одновременно определяются совпадающие между собой значения $(F\ X)$ и $(F_n\ X)$ для некоторого n, а это означает, что полностью определена семантика функции F с произвольным значением аргумента X. Но в общем случае на завершение процесса при произвольном значении аргумента X можно только надеяться, т. е. процесс оказывается потенциально бесконечным.

Для тех значений аргумента X, когда случай «в» все же когда-то наступает, значение $(F\ X)$ определяется только в этот момент. Чтобы дождаться этого, надо быть бесконечно терпеливым. Если это все же случится, то это значение одновременно оказывается значением $(F_n\ X)$ для некоторого n и для всех бóльших значений n.

Но значения X, для которых процесс не обрывается никогда, не могут быть выявлены и при потенциально бесконечном терпении. Таким образом, предикат «вычисление $(F\ X)$ при данном X завершается» *полуразрешим* (см. разд. 1.7.5). Эта половинчатость мешает признать функцию F конструктивным объектом, а ее семантику — финитной. Но похоже, что не все конструктивисты с этим согласны (см. [31], § 55.3.2°).

1.7.7. На общем фоне

Со времен Ньютона в физике и родственных ей науках господствует тенденция описывать законы природы языком дифференциальных уравнений — от обыкновенных в случае механики системы материальных точек до уравнения Шредингера в волновой механике и далее. Однако эти уравнения оказываются сравнительно простыми лишь в весьма абстрактных моделях, таких как материальная точка, идеальный газ, несжимаемая жидкость, абсолютно твердое тело, изолированная система и т. п. При этих допущениях удается установить наличие у исследуемого объекта некоторых общих свойств вроде законов сохранения энергии и импульса, неубывания энтропии и пр.

Но и эти упрощенные уравнения редко удается решить до конца в некотором условном смысле. Например, «проинтегрировать в квадратурах» уравнения задачи двух тел (читай — точек), «взять» эти квадратуры, т. е. представить интегралы через элементарные функции и получить законы Кеплера, или же решить уравнение Шредингера для атома водорода. Но уже для случая трех тел, будь то в классической или в квантовой механике, подобное решение получить невозможно. А если нечто подобное возможно, то либо квадратуры не берутся, либо многие из найденных композиций элементарных функций оказываются весьма громоздкими. Короче говоря, решение физической задачи обычно не приводится к желаемому виду, а сам этот вид далеко не столь обозрим и легко анализируем, как считается.

Приходится, оставаясь на уровне дифференциальных уравнений, искать их приближенные решения численными методами. Нередко этот путь на практике даже более эффективен из-за его стандартности, чем возня с общими аналитическими решениями. Численные методы обычно основаны на замене дифференциальных уравнений близкими к ним в том или ином смысле разностными, т. е. алгебраическими. Их преимущество в том, что решения можно вычислять по шагам, а посему — поручить компьютеру. Не будем останавливаться на проблемах оценки близости решений дифференциальных уравнений и аппроксимирующих их разностных, на сходимости и устойчивости численных методов и т. п.

Основной вывод из сказанного — найти общее, аналитическое, решение уравнений, описывающих физическое явление, большей частью невозможно и прихо-

дится анализировать не это решение, а само уравнение. Если же решение найдено, то воспользоваться им в практических целях также редко удается.

А можно ли надеяться, что для задач, изначально предполагающих разностную форму описания (например, в виде уравнения $h = \varphi(h)$ — см. сказанное на с. 139 и замечание на с. 140) и пошаговый способ решения (см. разд. 1.7.1), положение окажется существенно иным? По-видимому, нет. Результаты предыдущих разделов подтверждают этот вывод, который будет развит и дополнительно обоснован в главе 3.

Диагональный метод позволяет установить, что общие свойства алгоритмов далеко не всегда могут быть найдены без их — алгоритмов — исполнения, в результате применения некоторого другого алгоритма к тексту исследуемого и к тексту исходных данных для его работы. В общем же случае выявить свойства результата вычислений можно лишь прослеживая и анализируя работу алгоритма при конкретном варианте исходных данных. Именно в такой формулировке можно принять утверждение об алгоритмической неразрешимости (или о неразрешимости стандартными средствами) почти всех массовых проблем теории вычислимости.

Библиографическая справка

В списке литературы [22, 34] — это фундаментальные учебники, [7, 12, 21, 31, 32, 35, 41, 47] — монографии, [48] — популярная, но содержательная книга.

По книгам [21, 22, 34, 41] читатель сможет составить гораздо более полное представление о круге задач, рассматриваемых в современной математической логике и теории множеств, и о подходах к их решению. Наше изложение математической логики в основном следовало по пути, принятому Э. Мендельсоном в [34], но лишь до того места, где интересы программистов и логиков начинают осязаемо расходиться.

Источники [12, 31, 32, 41] ознакомят читателя со взглядами на основания математики различных представителей интуиционистского и конструктивного направлений. Комментарии А. А. Маркова существенно обогащают книгу [12].

В книге [47] излагаются другой подход к базовым понятиям математической логики и основы тех методов, которые с 60-х годов начали применяться для автоматического доказательства теорем, а позже — для автоматизации поиска алгоритмов решения задач по формальным описаниям последних.

Вопросы вероятности и информации освещены в [7, 48], книга [18] — это весьма полное справочное пособие по теории графов.

2. Основные понятия и конструкции языков программирования

В этой главе дан обзор основных понятий, встречающихся в некоторых современных языках программирования, таких как «Паскаль», ПЛ/1, Алгол 68, Си, «Ява» (Java) и др. (Многие пишут Паскаль и Ява без кавычек, не проявляя должного уважения ни к великому мыслителю, ни к знаменитому острову.) Главное внимание здесь уделено не деталям и частным случаям, которые в изобилии встречаются в этих языках, особенно в ПЛ/1 и других «фирменных» языках, а наиболее общим понятиям, взятым в достаточно простой форме. Основная цель — расширить кругозор читателя, находящегося в плену представлений, почерпнутых из так называемых новых информационных технологий, дать ему возможность ознакомиться с традиционными программистскими понятиями, так или иначе присутствующими в этих технологиях часто в сильно искаженном и неудобном для пользования виде, и применять их для составления более обозримых, проще и лучше организованных программ. Предварительное знание какого-либо из этих языков от читателя не требуется. Из машинных понятий полезно знать только, что такое память, разряд (бит), ячейка (один или несколько восьмиразрядных байтов), адрес (номер ячейки или байта). Глава не рассчитана на первоначальное обучение — предполагается, что читатель уже испытывал потребность обратиться к компьютеру для решения какой-либо возникшей перед ним задачи и начал испытывать затруднения при работе с предлагаемымиему программными средствами. А чтобы написать работающую программу, ему предстоит точно узнать, как описанные здесь понятия воплощены в выбранном им языке.

2.1. Программы

2.1.1. Данные и информация

В обиходе сплошь и рядом говорится о том, что компьютеры имеют дело с информацией — получают ее, хранят, обрабатывают и передают. Но так ли это на самом деле?

В разд. 1.6.3 количество информации, заключенной в сообщении, связывалось с разнообразием реакций получателя на это сообщение. В случае компьютера его реакция на вводимые данные однозначна — первым делом занести их в память, не теряя по возможности ни одного знака, и ждать дальнейших команд. На каждую команду компьютер реагирует вполне определенным, заранее известным, образом — будь это не так, человеку было бы от него мало пользы. Обработка данных — трансляция программы, вычисления по ней и пр. — также должна приводить ко вполне определенному результату, который один к одному может быть получен вновь при повторной обработке. Изменение результата при иной установке флажков, при перенастройке системы — не в счет, просто изменились попутные данные.

В [7] утверждается, что, обрабатывая информацию, машина ничего к ней не добавляет, не создает никакой новой информации, а лишь придает ей новую форму (как в любом ином канале передачи сообщений при кодировании и декодировании данных). С этим нельзя не согласиться с одной лишь оговоркой — выходные данные, как правило, имеют бóльшую ценность для человека, чем исходные. Но это зависит уже от разнообразия реакции человека на те и другие данные — см. конец разд. 1.6.3. Именно в этом разнообразии кроется, к нему сводится способность человека извлекать информацию из данных, делать для себя определенные выводы и поступать в соответствии с этими выводами. Разные люди извлекают больше или меньше информации из одних и тех же данных. Разные компьютеры, если они вообще способны обработать некоторые данные, должны делать это практически одинаково — это и подразумевается, когда заходит речь об их (вместе с установленным на них программным обеспечением) совместимости.

Итак, данные — сигналы, записанные теми или иными знаками на какомлибо носителе — адресованы компьютеру. Компьютер их обрабатывает, преобразуя в другие данные, и обязан это делать как можно более единообразно.

Информация, заключенная в данных (если она там действительно присутствует), адресована одним человеком другому и может быть воспринята только человеком. Иногда получатель способен извлечь из данных даже больше информации, чем имел в виду отправитель. Пример: отправитель экспериментальных данных — технический сотрудник, а получатель — квалифицированный специалист. Читайте также детективные романы.

Можно было бы посетовать на искажение истинного смысла в таких терминах, как «обработка информации», «школьная информатика», «информационные технологии», но в естественных языках это наблюдается сплошь да рядом.

2.1.2. Языки программирования

Формально язык программирования — это открытое для пополнения множество текстов, записанных с помощью некоторого набора символов — *алфавита* языка (ср. с разд. 1.1.2). Содержательно, по основному своему назначению, язык программирования — это средство общения между человеком, или, как еще говорят, *пользователем* языка и вычислительной машиной (компьютером). Но сразу после своего возникновения языки программирования приобрели и другие функции — стали средством общения (обмена программами) между людьми и даже средством мышления, причем очень важным, для людей, которым по роду их деятельности постоянно приходится составлять программы для решения разнообразных задач и заставлять компьютеры работать по этим программам.

При оценке возможностей, которые язык программирования предоставляет пользователю, на первое место выступают именно качества языка как источника представлений о том, как работает программа, как связаны между собой ее составные части. Хорошо организованный язык с четко продуманной системой понятий, достаточно общих, но в то же время простых и ясных, хорошо увязан-

ных друг с другом, помогает пользователю организовать свои мысли, отделить главное от второстепенного на каждом этапе составления программы, последовательно двигаясь к намеченной цели. Поэтому во всем мире наиболее опытные и авторитетные программисты высоко оценивают языки, в полной мере обладающие отмеченными выше качествами, и скептически относятся к языкам Фортран, ПЛ/1, даже Си и им подобным. Достоинство языка заключается не в обилии включенных в него средств, а именно в концептуальной продуманности, хотя и то, и другое обычно имеют в виду, говоря об уровне языка. Хороший программист-профессионал должен владеть языками любого уровня, а первоначальное обучение идеям и методам программирования следует вести на примере языка, в наибольшей степени дисциплинирующего и систематизирующего мышление.

Применительно к знаковым системам, в частности к языкам программирования, говорят об их *синтаксисе* — правилах образования текстов, *семантике* — правилах истолкования текстов тем, кому они адресованы, и *прагматике*, сопоставляющей тексты целям и намерениям того, от кого они исходят. В случае языков программирования синтаксис — это совокупность требований, которым должна удовлетворять любая осмысленная программа. Впрочем, не исключен вариант, когда синтаксически правильная программа не имеет никакого содержательного смысла, например (этот и следующие два примера написаны на языке «Паскаль»)

```
program Nonsense;
  var i, n : integer;
begin for i := 1 to 100 do n := i;
  n := n;  Writeln(i)
end.
```

Семантика языка программирования — это правила, определяющие, какие операции и в какой последовательности должна исполнить машина, работая по произвольной заданной ей программе. Не исключено, что некоторой синтаксически правильной программе не может быть сопоставлена никакая последовательность исполнимых действий, например

```
program Nonexecutable;
  var i, n : integer; u, v : real; x : array [1 .. 10] of real;
begin n := 20;  u := 10.0;  v := 1E1;
  for i := -n to n do x[i] := i/(u - v)
end.
```

Говорят, что такие программы содержат семантические ошибки или что их исполнение семантически не определено.

Прагматика языка программирования — это, по сути дела, методология программирования, т. е. описание принципов, методов и приемов, позволяющих исходя из постановки задачи составить программу ее решения. Некоторые семантически правильные программы могут оказаться прагматически совершенно неприемлемыми. Например, приведенная ниже программа — не слишком утрированный пример того, как некоторые начинающие программисты решают задачу о вычислении числа $e = 2.71828...$ с точностью до 10^{-5}.

```
program Calculate_e;
  var i,j,k,n : integer; x,y : real;
      a,b,c : array[1..100000]of real;
begin n := 100000; a[0] := 1; b[0] := 1; c[0] := 1;
  x := 1;
  for i := 1 to n do
  begin a[i] := 1; for j := 1 to i do a[i] := a[i] * x end;
  for i := 1 to n do
  begin b[i] := 1; for k := 1 to i do b[i] := b[i] * k end;
  for i := 1 to n do c[i] := a[i]/b[i];
  y := 0; for i := 0 to n do y := y + c[i];
  Writeln(y:8:5);
end.
```

Упражнение 2.1.1. Как бы Вы решили эту задачу?

Некоторые массовые (т. е. допускающие различные варианты исходных данных) задачи, хотя и формулируются крайне просто, не имеют никакого общего способа (алгоритма) их решения, а потому называются *неразрешимыми стандартными средствами* (см. разд. 1.7.5). Такова, например, задача: проверить на семантическую правильность любую заданную программу. Она неразрешима не потому, что в «Паскале» недостаточно четко определено, какую программу следует считать семантически правильной, и не потому, что в ней идет речь о проверке семантической правильности любой программы. Даже если для одной конкретной программы требуется проверить, возможно ли в ней деление на нуль или выход значений индексов за границы, заданные в описании массивов, то обычно это нельзя сделать, не зная, с какими исходными данными она должна исполняться. Да и располагая этими данными, возможность возникновения этих неприятностей часто можно установить, лишь доведя исполнение программы до получения значения делителя или индекса.

Существование задач, неразрешимых стандартными средствами, делает предмет прагматики несколько расплывчатым и неопределенным — нельзя в общем случае дать никаких рекомендаций, которые от постановки задачи гарантированно приводили бы к программе, ее решающей. Кроме того, из-за разнообразия решаемых с помощью компьютеров задач, те рекомендации, которые могут быть даны, либо носят чересчур общий характер, либо, наоборот, слишком конкретны, относятся к узкому классу задач.

Следует указать на один важный принцип прагматического характера — составление программы не только должно начинаться с четкой постановки задачи, но и проходить на всех его этапах с оглядкой на эту постановку. Инженеры из всех отраслей техники любят говорить по этому поводу: правильно поставить задачу — значит наполовину ее решить.

Теория программирования, хотя и может гордиться рядом значительных достижений, ориентирована больше на разработчиков языков программирования и общего программного обеспечения компьютеров, чем на массового пользователя.

Многие ее результаты представляют собой лишь постановки задач, решение которых потребует от теоретиков и практиков программирования еще многих лет труда, если вообще когда-либо будет достигнуто. Поэтому бо́льшую роль в овладении навыками составления программ играет разбор примеров разной степени сложности из разных классов задач.

2.1.3. Описание синтаксиса языков

Ввиду такого положения с прагматикой большее внимание при описании языков программирования уделяется их синтаксису и семантике. Способы описания того и другого довольно разнообразны. Однако для задания синтаксических правил наибольшей популярностью пользуется аппарат *форм Бакуса-Наура* (*БНФ*). Их основное назначение — определить, какие именно последовательности символов считаются программами в данном языке программирования. Это достигается указанием, из каких составных частей и каким способом могут быть построены программы. Для обозначения этих составных частей и программ в целом служат особые символы, называемые *металингвистическими переменными* или, короче, *метапеременными*. Они представляют собой слова или группы русских слов, заключенные в специальные кавычки ‘<’ и ‘>’, например:

<программа>
<арифметическое выражение>
<идентификатор>
<знак операции типа сложения>

Возможны более краткие, но менее мнемоничные обозначения.

Должен быть задан также перечень (алфавит) *первичных символов* языка, называемых также *основными*, из которых в конечном счете состоят программы. Под *значением* метапеременной понимается некоторая конечная последовательность основных символов.

Каждая форма Бакуса-Наура, иначе — *металингвистическая формула*, а в контексте синтаксических описаний — просто формула, задает одно или несколько правил образования возможных значений метапеременной. Формула состоит из определяемой метапеременной, следующего за ней *метасимвола* ‘::=’ (или иного — в разд. 1.7.2 использовался символ ‘:’) и одной или нескольких (по числу задаваемых правил) последовательностей, образованных из основных символов и метапеременных. Эти последовательности, если их больше одной, разделяются метасимволами ‘|’. Любое возможное значение определяемой метапеременной получается, если в одной из таких последовательностей заменить все метапеременные какими-либо их возможными, уже найденными, значениями.

Сейчас мы приступаем к описанию синтаксиса и семантики некоторых конструкций, присутствующих в том или ином виде почти во всех языках программирования. Употребляемые при этом слова «иногда», «возможно», «случается, что» и т. п. будут указывать на варианты, принятые лишь в некоторых из этих языков. Условимся, что в качестве основных символов будут использоваться латинские буквы, цифры, знаки препинания, арифметические знаки и некоторые

сочетания этих знаков, а также слова, набранные полужирным шрифтом, вроде **div** или **mod**. Все эти символы неделимы, так что символ '<=' не следует воспринимать как стоящие рядом символы '<' и '=' (имеющие свое собственное значение), а буква **d**, если бы она появилась в качестве основного символа, не имела бы никакого отношения к вхождению той же буквы в символы **div** и **mod**. Переходим к примерам.

Формула

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

задает десять очевидных правил, которые по отдельности могли бы быть заданы формулами

<цифра> ::= 0 . . . <цифра> ::= 9

Разумеется, эти формулы удовлетворяют определению БНФ лишь потому, что цифры 0, . . . , 9 были объявлены выше основными символами языка. Поскольку правые части этих формул не содержат метапеременных, возможными значениями метапеременной <цифра> являются односимвольные последовательности

0 1 . . . 9

и только они.

Формула

<целое без знака> ::= <цифра> | <целое без знака> <цифра>

задает два правила. Согласно первому из них любое значение метапеременной <цифра> может также быть значением метапеременной <целое без знака>. Короче, любая цифра является целым без знака. Второе правило означает, что, приписав справа к целому без знака любую цифру, мы получим еще одно целое без знака. Таким образом, возможные значения метапеременной <целое без знака> — это произвольные конечные последовательности цифр.

Металингвистические формулы ничего не говорят о том, какой смысл в языке приписывается таким последовательностям — это задача семантики. Семантику целых без знака, опираясь на опыт человека, знакомящегося с языком программирования, можно определить, сказав, что эти последовательности имеют обычный смысл. Но, желая дать формальное определение, можно пояснить, что если <целое без знака> в правой части второго правила изображает целое m, а <цифра> — число d, то значение целого без знака в левой части правила — это $10\,m + d$. Уже этот примитивный пример показывает, что металингвистическая формула, задавая структуру части программы, представленной метапеременной из левой части правила, может также подсказывать (но не более) способ исполнения этой части программы (ее семантику), если для метапеременных в правой части этот способ уже известен.

Дадим более содержательный пример. Слегка поступаясь истиной, строение арифметического выражения в языке «Паскаль» можно описать так:

<одноместная операция изменения знака> ::= −
<двухместная операция типа сложения> ::= + | −
<двухместная операция типа умножения> ::= ∗ | / | **div** | **mod**
<выражение типа произведения> ::=

<первичное арифметическое выражение>
| <выражение типа произведения>
<двухместная операция типа умножения>
<первичное арифметическое выражение>
<арифметическое выражение> ::=
<выражение типа произведения>
| <одноместная операция изменения знака>
<выражение типа произведения>
| <арифметическое выражение>
<двухместная операция типа сложения>
<выражение типа произведения>
<первичное арифметическое выражение> ::=
<число без знака> | <переменная>
| (<арифметическое выражение>)

где **div** и **mod** — знаки операций деления целых чисел с остатком и взятия остатка от такого деления, так что

$$n = (n \,\mathbf{div}\, m) * m + n \,\mathbf{mod}\, m.$$

Эти формулы задают правила записи арифметических выражений. Но одновременно задается структурная связь между выражением в целом и его частями, а это служит косвенным указанием на то, что операции типа умножения исполняются раньше операций со знаками '$-$' и '$+$', одноместная операция изменения знака над значением выражения типа произведения — раньше операций типа сложения, а значение арифметического выражения в скобках должно быть найдено раньше, чем будут исполняться какие-либо операции вне этих скобок. Операции одного типа исполняются в порядке слева направо, так что выражение $x/y * z$ соответствует в обычной записи выражению $\frac{x}{y}\, z$, а дробь $\frac{x}{yz}$ должна быть записана в виде $x/(y * z)$.

Допустим, нам известно, что x, y и z — это переменные, а числа без знака записываются в более или менее естественном виде. Нетрудно убедиться, что

$$x \qquad -(x + y) * z \qquad x + y * z$$

— это арифметические выражения, а последовательности символов

$$y * z/ \qquad x + -y * z$$

арифметическими выражениями не являются. Более того, из металингвистических формул следует, что арифметическое выражение $-(x + y) * z$ содержит в своем составе переменные (они же первичные арифметические выражения, они же выражения типа произведения) x, y и z, арифметические выражения x и $x + y$, первичное арифметическое выражение (оно же выражение типа произведения) $(x + y)$ и выражение типа произведения $(x + y) * z$. Переменную y также можно было бы воспринять как арифметическое выражение, но это не помогло бы, а скорее помешало в проделанном нами *синтаксическом разборе* выражения $-(x + y) * z$.

Обычно металингвистические формулы строятся так, что синтаксический разбор программы и всех ее частей протекает однозначно.

Некоторые достаточно простые конструкции бывает трудно описать, строго соблюдая требования к БНФ. Например, в программах часто содержатся *примечания* — тексты, адресованные человеку, читающему программу, а не машине, ее исполняющей. Формула

<примечание> ::= {<последовательность символов, отличных от '{' и '}'>}

не является БНФ. Она, по-видимому, не должна вызывать затруднений у человека, изучающего описание языка. Но компьютер нуждается в особых правилах обработки (точнее, игнорирования) примечаний.

2.1.4. Описание семантики

Семантика языка программирования в целом задается указанием:

1) используемых в языке *типов* (т. е. множеств) *простых значений*, например целых и нецелых чисел;

2) способов построения *составных значений*, например комплексных чисел, векторов и матриц или их аналогов, из нескольких простых;

3) *операций* над простыми, а иногда и над составными значениями и других *действий*, отличающихся от операций тем, что они не вырабатывают новых числовых или иных значений (например, действие, называемое присваиванием, заключается в том, что за некоторым вычисленным или выбранным значением закрепляется некоторое обозначение);

4) способов задания *сложных действий*, складывающихся из ряда простых, выполняемых в определенной последовательности.

Простые и составные значения и действия — таков основной арсенал семантических понятий.

Семантика в отличие от синтаксиса чаще всего описывается словесно. Объясняется это тем, что, несмотря на многочисленные попытки, пока еще не удалось предложить столь же простой, выразительный и гибкий аппарат для описания семантических понятий, каким являются БНФ для задания синтаксиса. С другой стороны, многие требования синтаксического характера не могут быть описаны с помощью БНФ и тоже, как правило, формулируются словесно. Таковы, например, требования, чтобы для каждого появления какой-либо переменной в программе можно было бы найти в другом месте программы определение ее типа, или чтобы векторы были описаны как одномерные массивы, а матрицы — как двумерные, а элементы векторов и матриц выделялись в соответствии с этими описаниями (о массивах и их размерностях будет рассказано в разд. 2.2.2).

В общем случае под *реализацией* языка программирования на некотором компьютере понимается программа, переводящая программы с этого языка на машинный язык данного компьютера, подкрепленная набором обслуживающих программ, важнейшей из которых следует считать программу *подсказок* или *помощи*, которая в трудных ситуациях, когда нормальная работа программы не может продолжаться, объясняет пользователю причину этого, а также служит кратким пособием по синтаксису и семантике языка.

2.2. Структуры данных

2.2.1. Простые значения и их представление

Обычно в языках программирования существует несколько типов простых значений, хотя теоретически можно было бы обойтись и одним.

Простые значения — это числа, логические значения, литеры и ссылки. Типы простых значений часто называют *простыми типами*. В некоторых языках встречаются и другие простые типы.

Принято различать два основных типа *чисел*: *целые* и так называемые *вещественные*. Внутри этих типов числа иногда различаются по диапазонам допустимых значений. Диапазоны зависят не от самого языка программирования, а от его реализации на конкретном компьютере, точнее — от того, сколько байтов памяти отводится под одно числовое значение из того или иного диапазона (каков размер необходимой для этого ячейки памяти).

Если операнды двухместной операции принадлежат различным диапазонам, то они приводятся к более широкому диапазону из двух. Над целыми числами любая операция должна исполняться точно, если только она вообще выполнима внутри рассматриваемого диапазона. Если результат операции не укладывается в этот диапазон, то последствия могут быть различными: результат оказывается искаженным, результат рассматривается как число следующего по ширине диапазона или как вещественное число, работа программы прерывается с сообщением о невозможности исполнить операцию. Выбор между этими возможностями чаще всего также зависит от реализации языка.

Для вещественных чисел точность исполнения даже самых простых операций не гарантируется, т. е. допускается, что результат любой операции может быть получен с некоторой погрешностью — ошибкой округления. Величина этой ошибки зависит от операции, значений операндов, диапазона, которому принадлежат эти значения, и, разумеется, от компьютера, на котором исполняется программа, поскольку вид компьютера определяет способ представления чисел внутри каждого из допустимых диапазонов. Если результат операции не представим в соответствии со способом представления ее операндов, то опять же либо он переводится в следующий диапазон, либо исполнение программы прерывается с выдачей диагностического сообщения. Поэтому более точно этот тип можно было бы охарактеризовать термином «приближенные числа».

Таким образом, уже в самом начале изложения семантики языков программирования мы сталкиваемся с проникновением в нее понятий из вычислительной практики — на уровне абстракции, свойственном традиционной математике, удержаться не удается. Это и естественно, так как программирование — это не столько математическая, сколько инженерная дисциплина, своего рода технология вычислительных процессов, осуществляемых на компьютерах. Отвлекаться от машинных аспектов программирования особенно опасно с позиций прагматики, т. е. разумного построения программ решения различных задач.

Под программным *представлением* чисел или иных значений понимается принятый способ их обозначения, тесно связанный со способом хранения. Пред-

ставления чисел в различных языках программирования почти идентичны. Опишем синтаксис обозначений, принятый в «Паскале». Для целых чисел без знака он уже был приведен в разд. 2.1.2. Далее:

<целое> ::= <целое без знака> | + <целое без знака>
| − <целое без знака>
<число без порядка> ::= <целое> | <целое> . <целое без знака>
<порядок> ::= e <целое> | E <целое>
<число> ::= <число без порядка>
| <число без порядка> <порядок>

где точка отделяет целую часть числа от дробной, а порядок равен 10^n, если <целое> — это n.

Таким образом, обозначения

$0.1\,e+6 \qquad 100000\,E\,0 \qquad 100000.0$

представляют одно и то же вещественное число. Однако в реализации результаты преобразования этих обозначений в машинное представление могут и не совпасть из-за ошибок округления.

На словах добавим, что целые без знака в «Паскале» (но не в составе изображений вещественных чисел) могут записываться в шестнадцатеричной системе счисления цифрами 0, ..., 9, A, B, C, D, E, F. Такой записи должен предшествовать символ '$'. Например, последовательность $\$FF$ изображает десятичное число $15 \times 16 + 15$, т. е. 255 в десятичной записи.

Следующий тип простых значений — это *логические* или *булевы* значения. Значения эти называются «истина» и «ложь». Они служат для того, чтобы зафиксировать один из двух возможных ответов на некоторый вопрос, один из двух возможных исходов некоторой проверки, одно из двух состояний некоторого объекта. Часто бывает безразлично, какое именно логическое значение связывается с данным ответом — все зависит от соглашения, от формы поставленного или подразумеваемого вопроса. Важно, чтобы возможных ответов было ровно два.

Логические значения «истина» и «ложь» изображаются обычно английскими словами *true* и *false*, записанными строчными или прописными буквами, сокращенно T и F, символами **true** и **false**, цифрами 1 и 0 и т. п.

В «Паскале» существует тип, представляющий своего рода обобщение логического типа на произвольное конечное множество фиксированных значений — идентификаторов из заранее составленного списка. Такой список называется *шкалой*. Идентификаторы из любой шкалы сами себя обозначают, других обозначений не требуется. Обозначением типа шкалы служит список всех ее идентификаторов, взятый в скобки. Например, тип *weekday* может быть обозначен так:

(*mon*, *tue*, *wed*, *thu*, *fri*, *sat*, *sun*)

В языках программирования признано необходимым иметь дело с текстами, записываемыми с помощью некоторого набора печатных знаков — *литер*. Этот набор в языке обычно не фиксируется и зависит от реализации языка. В частности, в разных странах учитываются особенности национальных алфавитов. Обычно набор литер совпадает с набором знаков, доступных (с теми или иными

ухищрениями) на клавиатурах, экранах мониторов и в принтерах (печатающих устройствах). В число литер обязательно входит пробел — место в тексте, не занятое никакой видимой литерой. В ту давнюю пору, когда авторы программ сами не набирали их текст на клавиатуре, а пользовались услугами технического персонала, изображение пробелов приводило к затруднениям, которые сейчас вспоминаются с улыбкой. Значения из доступного набора литер относятся к типу *литерный*. Это еще не тексты, а только знаки, из которых составляются тексты — конечные последовательности литер, называемые обычно *строками* или *цепочками*. Строки иногда причисляют к простым значениям (типа *строковый*), иногда — к составным. Подробнее об этом мы скажем ниже, в разд. 2.2.2 и 2.3.2.

Строки изображаются последовательностями литер, из которых они состоят, заключенными в кавычки: непарные (‘’’ реже ‘"’) или парные (‘«’ и ‘»’). Применяются и другие формы кавычек. Отдельные литеры, если появилась надобность в их представлении, изображаются так же, как однолитерные строки.

Если строки заключаются в парные кавычки, то кавычками можно выделять и стро́ки внутри строки́. Например: «Ваня» и «Маня» — это строки внутри строки ««Ваня» и «Маня» — это строки». В случае непарных кавычек понятие внутренней строки, если не принять дополнительных соглашений, теряет смысл. Поэтому внутри строки одну (формально не имеющую пары) кавычку уславливаются изображать в виде сдвоенной кавычки $''$, например: $'''\text{Ваня}'' + ''\text{Маня}'' = \text{любовь}'$. Это не приводит к недоразумениям, так как в программах две строки подряд, без какого-либо разделителя между ними, встречаться не должны.

Есть более универсальный способ. Некоторому символу, например ‘@’, приписывается внутри строки роль *признака исключения*. Это значит, что сам он не входит в последовательность литер строки, а следующий за ним символ не несет своей обычной функции, а является рядовой литерой. Таким образом, две литеры ‘@’’ обозначают одну (непарную) кавычку, не закрывающую строку, а из двух литер ‘@@’ только первая служит символом исключения, а вторая изображает сама себя. Этот метод широко используется не только в языках программирования, но и во многих средствах работы с текстами (редакторах и пр.)

Наконец, еще один простой тип — это тип *ссылки*. Ссылка — это косвенное имя, придаваемое некоторому значению, и само являющееся значением. В повседневной жизни косвенные имена встречаются чаще, чем прямые. Большинство окружающих нас предметов, явлений, даже людей занимают свое место в нашем сознании благодаря тем или иным связям с другими предметами, явлениями или людьми, а их прямые имена (если они есть) отступают на задний план. Часто, например, услышав фамилию актера, мы спрашиваем: «А где он играл?» — созданные им образы значат для нас больше, чем фамилия. Не обнаружив стула на своем рабочем месте, мы возмущаемся: «Кто взял мой стул!», а не «Кто взял стул инв. № 2816». В авиабилет нам вписывают номер рейса, а не заводской и не бортовой номер самолета, выполняющего этот рейс.

Такие косвенные связи часто бывают случайными и непредсказуемыми. Например, пассажир, если он специально не заглянет в расписание, обычно не знает

номера нужного ему рейса, а кассир не знает, какой самолет будет назначен на этот рейс. Преподаватель, впервые придя в группу, бывает вынужден обратиться к студенту примерно так: «Вот Вы — в третьем ряду у окна — как Ваша фамилия?». Поэтому и приходится прибегать к косвенным именам.

По мере расширения сферы использования компьютеров росла и потребность в отображении косвенных, ассоциативных связей между объектами и явлениями (процессами). Косвенные имена в языках программирования — одно из главных средств, обслуживающих эту потребность.

Семантику ссылок можно легко объяснить в машинных терминах. Значение типа ссылки — это адрес в памяти компьютера, по которому открыт или в подходящий момент откроется доступ к некоторому другому значению или объекту, например, адрес ячейки, где записано (или будет записано) некоторое число. Пользователю обычно интересен не адрес, а само число. Но у адреса есть преимущество — он короче, занимает меньше места, чем значение, хранящееся в памяти по этому адресу. Это преимущество особенно возрастает, когда объект ссылки занимает в памяти не одну, а много ячеек.

В машинных языках адрес играет роль имени содержимого ячейки. Например, в командах арифметических операций лишь в специальных случаях выписываются числа, над которыми производится операция, обычно же — адреса ячеек, где хранятся эти числа. В языках программирования машинные адреса не используются. Их появление там противоречило бы основной идее — создать язык, ориентированный не на машину, а на человека, понятный и удобный ему. Их роль и выполняют ссылки, которые возникают во время работы программы вместе с объектом ссылки, могут быть сохранены, как всякие другие значения, и в нужный момент использованы для доступа к объекту. В процитированном выше обращении преподавателя к студенту тоже использован своего рода адрес. Дальнейшие разъяснения даны в разд. 2.3.4 вместе с примером, помогающим уяснить некоторые практические приемы работы со ссылками.

Хотя для ссылок и не требуется никакого явного обозначения, среди значений типа ссылки необходимо иметь значение, ни на что фактически не ссылающееся. Аналогами этого понятия могут служить целое число нуль (отсутствие предметов счета), пустое (лишенное элементов) множество, пробел внутри строки. Для этого значения как раз требуется явное обозначение. Обычно им служит слово *NIL* или символ **nil**.

Важный класс обозначений, встречающихся в любом языке программирования, образуют идентификаторы:

<идентификатор> ::= <буква>
| <идентификатор> <буква>
| <идентификатор> <цифра>

В частности, они могут служить синонимами любого из рассмотренных обозначений. Например, синоним для приближенного значения константы π может быть создан с помощью следующего *определения константы*

$pi = 3.1415926536$

где слева от знака равенства стоит вводимый синоним, а справа — исходное изображение константы. Для пущей наглядности, перед определением константы или перед группой таких определений требуется поместить символ **const**, завершая каждое определение точкой с запятой, например

const $n = 25$; $e = 2.7182818285$;

А что получится, если завести синоним для ссылки, и как это можно сделать? Ответ на первую половину вопроса довольно ясен — в нашем распоряжении окажется явное имя, обладающее теми же возможностями, что и ссылка — имя неявное. По этому имени, как объяснялось выше, можно получить доступ к некоторому уже существующему объекту или объекту, появление которого еще только ожидается (например, будущие родители могут решить, что своего сына, если Бог не пошлет им дочку, они назовут Васей, и говорить «Наш Вася» вместо «Наш будущий сын»). На вторую половину вопроса ответа пока нет, поскольку ссылки не имеют явных изображений и в определении константы справа от знака равенства мы сейчас написать ничего не можем.

К этой теме мы вернемся в разд. 2.3.1 и 2.3.4.

2.2.2. Составные значения и их типы

В современных языках программирования могут использоваться значения сложной структуры, теоретически — сколь угодно сложной. Надо уметь задавать эту структуру, называемую также *типом* значения.

В основе классификации типов лежат уже упоминавшиеся простые типы. Для их обозначения служат символы: **int** (целый), **real** (вещественный), **bool** (логический), **ref** (тип ссылки), **char** (литера), а если строки считаются простыми значениями, то и **string** (строка). Для обозначения типов чисел, принадлежащих разным диапазонам, используются два приема. В «Паскале» произвольный *диапазон* целых чисел может быть задан так:

<диапазон> ::= <целое> .. <целое>

В таком задании первое <целое> m не должно превосходить второе — n: $m \leqslant n$ (иначе диапазон не содержит ни одного значения). Число $n-m+1$ значений в данном диапазоне называется его *длиной*.

Кроме того, для некоторых диапазонов введены индивидуальные обозначения:

shortint — для диапазона $-128\,..\,127$
byte — для диапазона $0\,..\,255$
integer — для диапазона $-32768\,..\,32767$
word — для диапазона $0\,..\,65535$
longint — для диапазона $-2147483648\,..\,2147483647$

Легко проверить, что для чисел типов **shortint** и **byte** достаточно одного байта памяти, типов **integer** и **word** — двух, а типа **longint** — четырех байтов. Аналогично типы **single**, **real**, **double** и **extended** относятся соответственно к четырех-, шести-, восьми- и десятибайтовым вещественным числам. При этом

число байтов влияет, главным образом, на точность представления. Но меняется также количество разрядов, выделенных под десятичный порядок числа, а вместе с ним — диапазон возможных значений для данного типа. Во всех случаях в диапазон входит число ноль, а абсолютная величина положительных или отрицательных значений может лежать в диапазоне от $1.3e-45\,..\,3.4e38$ для типа **single** до $3.4e-4932\,..\,1.1e4932$ для типа **extended**.

Другой прием заключается в приписывании к основному типу **int** или **real** одного или нескольких префиксов **long** для указания на желательность расширить диапазон. Возможность и степень расширения диапазона фактически зависят от компьютера и реализации языка.

В дальнейшем будем считать, что <тип>, <тип1>, <тип2>,... — это входящие в программу обозначения некоторых типов. Попутно с рассказом о типах составных значений будем пояснять, как строятся их обозначения.

В большинстве случаев можно было бы обойтись всего двумя способами образования составных значений. Первый способ — это объединение нескольких значений одного и того же типа в одномерный *массив* (вектор). Объединяемые значения — *элементы* массива нумеруются целыми числами (*индексами*). Но, в отличие от векторов в алгебре, нумерация элементов не обязана начинаться с 1. Индексы могут быть и отрицательными. Их диапазон задается так, как было сказано в начале раздела, и часто называется *граничной парой*, состоящей из двух *границ*: *нижней* и *верхней*. Итак:

<тип массива> ::= **array**[<диапазон>] **of** <тип элементов>
<тип элементов> ::= <тип>

Служебные символы **array** и **of** избыточны, но они делают обозначение типа более наглядным, позволяя, в частности, сразу увидеть, что определен именно тип массива.

Строго говоря, массив оказывается одномерным только в том случае, если его элементы — не массивы. Массив n-мерных массивов (иначе — массивов *размерности* n) при любом $n \geqslant 1$ — это уже $(n+1)$-мерный массив. В алгебре аналогом двумерного массива служит матрица — прямоугольная таблица. Элементами этой таблицы следует считать ее строки, сами являющиеся одномерными массивами. Трехмерный массив можно уподобить пачке листов, каждый из которых содержит по матрице — все одного и того же размера, т. е. с одинаковым числом строк и столбцов.

Обозначение типа многомерного массива можно записывать в сокращенном виде, заменяя последовательности символов

]of array[

между соседними граничными парами на запятые. В результате обозначение типа, например, вещественной $m \times n$-матрицы вместо

array$[1\,..\,m]$**of array**$[1\,..\,n]$**of real**

примет вид

array$[1\,..\,m, 1\,..\,n]$**of real**

Как такое изменение синтаксиса может повлиять на семантику массива, будет сказано в разд. 2.3.1.

Строку длины n можно рассматривать как массив литер (значений типа **char**). Тип такой строки обозначается

array $[1\,..\,n]$ **of char**

где вместо n должно стоять положительное целое (если идентификатор n не был определен в качестве синонима этого целого). Но чаще такое обозначение сокращается до

string $[n]$

тогда как просто **string** обозначает тип строки максимально возможной в данной реализации длины.

Второй способ образования составных значений — это объединение некоторого фиксированного числа k значений, возможно, различных, но также фиксированных типов <тип1>, ... , <тип2>, в одно (разумеется, «тип2» — это k-й по порядку тип). Такое объединение принято называть *записью* (реже *структурой*), а объединяемые значения — *полями* этой записи. Чтобы можно было выделять из записи отдельные поля, они снабжаются индивидуальными наименованиями — *идентификаторами полей*, которые входят в обозначение типа записи:

<тип записи> ::= **record** <список полей> **endrec**
<список полей> ::= <идентификатор>:<тип>
| <список полей>; <идентификатор>:<тип>

Классический пример типа записи — тип комплексного числа, имеющий обозначение

record re : **real**; im : **real endrec**

Несколько полей одного типа могут объединяться в группы, тогда их общий тип указывается лишь один раз:

record re, im : **real endrec**

Оба обозначения представляют тип записи с двумя вещественными полями, снабженными идентификаторами re и im. В разд. 2.3.1 будет показано, как для записи z такого типа можно построить обозначения ее полей, аналогичные традиционным $\mathrm{Re}\,z$ и $\mathrm{Im}\,z$ для вещественной и мнимой части комплексного числа z.

Тип записи для календарной даты может быть таким:

record *day* : **byte**; *month* : **string** [8]; *year* : **word endrec**

(с учетом «проблемы 2000 года»), а одна строка экзаменационной ведомости в студенческой группе может быть охарактеризована типом

record *name*, *mark* : **string** [17] **endrec**

в расчете на то, что фамилия студента не окажется длиннее слова «удовлетворительно». Тип всей ведомости может быть обозначен примерно так:

record *faculty* : **string**;
group, *day* : **byte**; *month* : **string** [8]; *year* : **word**;
subject : **string**;

marklist : **array** [1 . . 30]**of**
record *name*, *mark* : **string** [17] **endrec**
endrec

Этот пример показывает, что обозначения типов вполне естественно могут иметь довольно длинные общие части. Для них обычно создаются имена (своего рода ти́повые константы). Как это делается, покажем на примерах:

type *compl* = **record** *re*, *im* : **real endrec**
type *date* = **record** *day* : **byte**; *month* : **string** [8]; *year* : **word endrec**
type *markrec*=**record** *name*, *mark* : **string** [17] **endrec**
type *examlist*=
record
faculty : **string**; *group* : **byte**; *examdate* : *date*; *subject* : **string**;
marklist : **array** [1 . . 30] **of** *markrec*
endrec

В общем случае, такое *определение типа* имеет синтаксис:

type <идентификатор>=<тип>

где <тип>, как уже было сказано, может быть простым типом, типом массива или записи, именем типа, введенным с помощью подобного определения, или одним из типов, о которых еще пойдет речь в дальнейшем.

Некоторые типы по самой идее их использования должны быть *рекурсивными*. Это значит, что написать обозначение для такого типа можно, только используя прямо или косвенно его имя, вводимое с помощью определения типа. Простейший, но в то же время чрезвычайно важный и распространенный случай — это тип звена так называемого списка. Такое звено состоит из содержательной информации — значения некоторого типа <тип> и ссылки на следующее звено списка. Тип такого звена естественно определить так:

type *link* = **record** *value* : <тип>; *next* : **ref** *link* **endrec**

Здесь **ref** *link* — обозначение типа ссылки на объект типа *link*, и идентификатор *link* оказывается частью собственного определения, что и делает этот тип рекурсивным.

Другой пример дают два связанные между собой определения:

type a = **array**[1 . . n] **of** b
type b = **record** s : **string**; r : **ref** a **endrec**

Если идентификаторы типов *compl*, *date* и др. в приведенных ранее примерах вводились лишь для сокращения других обозначений, то типы *link*, a и b невозможно определить, не пользуясь их идентификаторами.

Появление ссылок в определениях рекурсивных типов отнюдь не случайно. Причина этого проясняется при попытке подсчитать размер участка памяти, занятого значением того или иного типа. Для каждого простого типа этот размер фиксирован, об этом уже говорилось. Для значения типа массива размер равен произведению длины массива на размер участка, требуемого для размещения каждого элемента, с некоторой постоянной добавкой на весь массив. Для

значения типа записи — сумме длин участков под все поля также с некоторой добавкой. Если бы мы попытались определить тип *link*, не пользуясь ссылками:

type *link* = **record** *value* : <тип>; *next* : *link* **endrec**

то это напоминало бы попытку вложить друг в друга две матрешки одинакового размера — длина требуемого участка превышала бы сама себя. Если же тип определен так, как это было сделано, то длина участка памяти под каждое звено лишь на константу (место под ссылку и для организации хранения всей записи) превышает размер памяти, необходимой для хранения одного значения более простого типа <тип>.

Общее правило проверки допустимости некоторого набора определений таково. Выбросим из каждого определения (только ради проверки, такие определения уже перестают быть синтаксически правильными) имена типов, стоящие после **ref**. Если после этого в правых частях не останется имен определяемых типов, то такой набор допустим. Если останутся, то будем поочередно заменять эти имена измененными правыми частями соответствующих определений. Этот процесс должен закончиться за конечное число шагов. Если же некоторые имена станут появляться вновь и вновь, то такой набор определений недопустим. Это случится, например, с последним вариантом определения типа *link*. А с определениями типов a и b мы сначала получим:

type a = **array**$[1\,..\,n]$ **of** b
type b = **record** s : **string**; r : **ref endrec**

а затем:

type a = **array**$[1\,..\,n]$ **of record** s : **string**; r : **ref endrec**
type b = **record** s : **string**; r : **ref endrec**

на чем проверка допустимости благополучно и закончится.

Итак, под типом объекта одновременно понимается как структура возможного значения объекта, так и все множество этих значений. На примере типов **shortint** и **byte** можно было увидеть, что эти множества могут частично пересекаться. Но обычно два типа или не пересекаются, или один из них целиком содержится в другом (образует его подмножество), как, например, **shortint**, **integer** и **longint** в этом порядке, или любой целый тип — в любом вещественном (но лишь в математическом смысле, так как машинные представления у них различны).

Тип **string**$[n]$ при любом значении n является подмножеством типа **string**, поскольку места, отводимого под строку максимально возможной длины, достаточно для любой строки. На том же основании тип **array** $[1\,..\,m]$ **of** <тип> следует считать подмножеством типа **array** $[1\,..\,n]$ **of** <тип> при $m \leqslant n$. Эта идея, хотя и завуалированно, используется в языке «Паскаль», где, например, процедура решения систем линейных алгебраических уравнений порядка n может быть применена для решения систем любого порядка m при $m \leqslant n$. Плата за такую «универсальность» процедуры — избыточная трата памяти и некоторые неудобства, вызванные наличием в матрице «лишних» элементов.

В других языках идея включения одних типов в другие проводится более гибко. Например, считается, что тип **array** $[1\,..\,n]$ **of** <тип> при любом целочис-

ленном значении n является подмножеством типа **array** [1 . .] **of** <тип>, а тип **array** [m . . n] **of** <тип> при любых m и n — это подмножество типа **array** [ ] **of** <тип>. Это соглашение автоматически распространяется на массивы любой размерности. Символ **ref** считается синтаксически правильным изображением типа, объединяющего все типы вида **ref**<тип> при любом <типе> объекта ссылки.

Общепризнано, что в мало-мальски развитых языках без объединения различных типов в один обойтись невозможно. Даже в таком «строго типизированном» языке, как «Паскаль», для этой цели введен аппарат записей с вариантами. Одно из полей такой записи объявляется *селектором варианта*, часть прочих полей — общими для всех вариантов, а остальные распределяются по вариантам. Можно определить, например, объединение не имеющих между собой ничего общего типов <тип1>, <тип2> и <тип3> в один следующим образом (синтаксис «Паскаля» слегка «подправлен»).

type *united* = **record case** *selector* : 1 . . 3 **of**
1 : (*field1* :<тип1>);
2 : (*field2* :<тип2>);
3 : (*field3* :<тип3>) **endrec**

Считается, что запись типа *united*, кроме поля *selector*, содержит при *selector*=1 только поле *field1* типа <тип1> и т. д.

Если тип поля записи, переменной или иного объекта задан с помощью изображения типа, отличного от символа или идентификатора, то возникает вопрос — всегда ли такое изображение задает один и тот же тип (*проблема тождества типов*)? Ясно, что для ответа на этот вопрос требуется, как минимум, проанализировать это изображение и учесть контекст, в котором тип был задан. Чтобы упростить анализ, в некоторых языках, например в «Паскале», тип подобного объекта следует задавать только с помощью имени типа, а если использовано более сложное изображение, считается, что тип объекта отличен от всех других типов. Алгол 68 впадает в противоположную крайность: если определены виды (**mode**):

mode m1 = **struct** (**int** p, q) и **mode m2** = **struct** (**int** p, q)

то объявления **m1** a и **m2** b создают переменные a и b одного вида.

Мы ниже будем считать, что проблема тождества (или включения) типов, заданных произвольными изображениями, решается без осложнений, и что различные имена представляют различные типы, если иное не указано явно: **type** $t1 = t2$.

2.3. Структуры действий

2.3.1. Переменные и их объявления

В конце разд. 2.2.1 мы вплотную подошли к понятию *переменной* как идентификатора — синонима ссылки на объект, способный в разные моменты исполнения программы принимать различные значения одного предписанного типа. Короче, переменная, в отличие от ссылки, — это явное имя объекта в программе. Значение объекта ссылки принято называть *текущим значением* переменной (хотя, строго говоря, значением переменной, как и всякой иной константы, следовало бы считать ее неявный синоним).

В разд. 2.2.1 было также сказано, что в машинной терминологии объект ссылки можно отождествить с участком (в простейшем случае — ячейкой) памяти, в который записываются сменяющие друг друга значения переменной. Адрес этого участка явно в программе не фигурирует. Про сам участок памяти обычно тоже не вспоминают, но и забывать о нем не следует.

Переменная возникает в результате исполнения ее *объявления*, имеющего синтаксис:

<объявление переменной> ::= **var** <идентификатор> : <тип>

где <идентификатор> — это собственно переменная, а <тип> — тип ее возможного значения. Очень часто несколько таких объявлений с разными идентификаторами, но с одним типом, объединяют в одно, например:

var x, y, z : **real**

Символ **var** включен в объявление лишь для большей наглядности и многие языки на его присутствии не настаивают.

Если в объявлении переменной <тип> — это тип составного значения, то именно такое значение — массив или запись — является текущим. Но очевидно нужно иметь доступ и к компонентам этого значения — элементам массива или полям записи, а если те — сами составные, то и к их компонентам, и т. д. вплоть до компонент простого типа. Аналогично, если значение переменной — это ссылка, то надо обеспечить доступ и к объекту этой ссылки. Поэтому понятие переменной не может сводиться лишь к идентификатору. Расширено оно может быть следующим образом:

<переменная> ::= <идентификатор>
| <переменная> <выделитель>
<выделитель> ::= [<индексное выражение>]
| . <выделитель поля> | ↑

Выделитель первого вида наращивает переменную со значением типа массива. В нем <индексное выражение> — это арифметическое выражение (см. разд. 2.1.2) со значением целого типа. Оно задает значение индекса того элемента массива, который требуется выделить, и поэтому должно лежать в пределах, установленных граничной парой в определении типа массива.

Во втором случае наращивается переменная со значением типа записи. Выделитель поля должен быть идентификатором из списка полей в определении этого типа.

Третий случай относится к переменной со значением типа ссылки. Символ '↑' означает, что по этой ссылке требуется перейти к значению объекта ссылки.

Если вид выделителя не соответствует типу переменной, или не соблюдены другие из перечисленных требований, то переменная не имеет смысла, программа, в которой она встретилась, не может быть исполнена. Некоторые из ограничений могут быть проверены во время трансляции (перевода) программы с языка программирования на машинный язык (говорят — проверены *статически*). Об их нарушении должен сообщать хорошо написанный транслятор. Другие ограничения, например, на значение индекса, иногда можно проверить только *динамически* — во время исполнения программы. От транслятора зависит, способен ли он включить в компьютерную программу проверку нарушения таких ограничений, а от пользователя — хочет ли он, чтобы такая проверка проводилась (по прошествии некоторого времени работы с программой эти проверки становятся уже мало актуальными, однако любое изменение, внесенное в программу, может повернуть ситуацию вспять).

Если идентификатор объявлен в качестве имени составного значения, то вместе с этим именем автоматически создаются имена описанного выше вида для всех его компонент и подкомпонент любого уровня (в дальнейшем те и другие будем называть просто подкомпонентами).

Прежде чем воспользоваться в любых целях именем, содержащим в своем составе индексные выражения, следует вычислить их значения и зафиксировать эти значения в связи с таким именем.

Применим сказанное к записи типа *examlist* из разд. 2.2.2. Пусть объявление переменной *table* имеет вид

var *table* : *examlist*

Идентификатор *table* обозначает всю ведомость — объект типа записи. Переменные

table . *faculty*
table . *group*

типа, соответственно, **string** и **byte** обозначают наименование факультета и номер группы. Переменная

table . *marklist*

имеет тип **array** [1 . . 30]**of** *markrec* и обозначает список студентов группы вместе с полученными оценками. Строку, относящуюся к одному из студентов, можно получить с помощью обозначения

table . *marklist*[*i*]

(если текущее значение переменной i лежит в пределах $1 \leqslant i \leqslant 30$) или, например,

table . *marklist*[7]

Тип этой строки — *markrec*. Узнать фамилию i-го по порядку в списке студента и полученную им оценку можно, взяв значения (типа **string**) переменных

table . marklist[*i*] . *name*
table . marklist[*i*] . *mark*

Обозначения

table[*i*]
table . marklist[*i*]*mark*

интуитивно, может быть, и понятны, но формально не допустимы, так как в первом случае выделитель приписывается к переменной не того типа, а во втором — с нарушением синтаксиса.

Для многомерных массивов можно разрешить замену двух оказавшихся рядом символов '][' на запятую. В некоторых языках требуется, чтобы такое сокращение было согласовано с аналогичным сокращением (см. разд. 2.2.2) в определении типа многомерного массива.

Но существуют и языки, где есть средства выделить из матрицы не только строку, но и произвольный столбец, или даже — из произвольного массива подмассив той же или меньшей размерности с меньшими длинами по всем или некоторым измерениям. Подробностей касаться не будем.

2.3.2. Операции и выражения

Операция в языках программирования — это действие, вырабатывающее некоторое значение, называемое ее *результатом*. Исходные данные для операции — ее *операнды* — это одно или два значения. Операция соответственно называется *одноместной* или *двухместной*. Как операнды, так и результаты операций — это обычно простые значения. Операции делятся на классы в зависимости от типов операндов и результата.

Операции с числовыми операндами и результатом называются *арифметическими*. Их типичный набор был приведен в разд. 2.1.2 в связи с примером описания синтаксиса арифметического выражения. Иногда к этому набору добавляют операцию возведения числа в степень с целым неотрицательным показателем. Тип результата при этом совпадает с типом основания степени.

Операции с числовыми операндами и логическим результатом называются *отношениями*. Существуют два стиля написания символов (знаков операций) отношений. Общепринятый набор отношений и их символов в обоих стилях таков:

равно	=	**eq**
меньше	<	**lt**
больше	>	**gt**
не равно	<>	**ne**
больше или равно	>=	**ge**
меньше или равно	<=	**le**

Синтаксис отношений прост и очевиден:

<отношение> ::=
<арифметическое выражение>
<знак операции отношения>
<арифметическое выражение>

Естественно, что результат операции отношения, будучи логическим значением, не может быть операндом другой такой операции, что и отражено в этой синтаксической формуле. По этой причине запись ограничения на значение переменной x в привычном виде

$0 <= x <= 1$

является недопустимой вольностью в любом языке программирования.

Операции с логическими операндами и результатом называются *логическими*. Их типичный набор в порядке старшинства (т. е. в том порядке, в каком они должны исполняться при вычислении значений логических выражений, если иное не предписано скобками):

отрицание	$\neg$	**not**
конъюнкция	$\wedge$	**and**
дизъюнкция	$\vee$	**or**
импликация	$\supset$	**imp**
эквивалентность	$\equiv$	**equ**

Значения логических выражений вычисляются в соответствии со следующей таблицей (той же, что в разд. 1.2.1), где u и v — значения операндов операции:

u	v	**not** u	u **and** v	u **or** v	u **imp** v	u **equ** v
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*		*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*		*false*	*false*	*true*	*true*

Логическое выражение обычно называют тем же термином, что и его главную операцию — операцию, которая при вычислении значения этого выражения исполняется последней. Так, например, конъюнкция

$0 <= x \wedge x <= 1$

— это правильная запись ограничения на значение переменной x, имевшегося в виду выше. Противоположное ограничение записывается в виде дизъюнкции

$x < 0 \vee 1 < x$

а утверждение о том, что эти два ограничения противоположны, — в виде эквивалентности (при другом стиле обозначений)

x **lt** 0 **or** 1 **lt** x **equ not** $(0$ **le** x **and** x **le** $1)$

Ограничимся этими примерами и подробный синтаксис логических выражений приводить не будем. Старшинство операций в разных языках определено по-разному. Поэтому при малейшем сомнении порядок исполнения операций в

выражении лучше всего задавать скобками, а чтобы избежать сомнений, следует тщательно изучить правила старшинства в языке, которым вы собираетесь воспользоваться.

Операции '=' и '<>' (совпадение и несовпадение значений), как правило, определены над значениями любого типа, даже над ссылками. Никакие другие операции над ссылками смысла не имеют.

Над литерами, кроме этих двух операций, иногда определяют отношение предшествования в некотором алфавитном порядке. Если алфавит кроме латиницы содержит и буквы национальных алфавитов, то реализовать этот порядок единообразно бывает затруднительно из-за большого разнобоя в машинной (двоичной) кодировке литер.

Если строки рассматриваются как простые значения, то над ними полезно определить следующие операции: выделение первой литеры (одноместная операция с результатом типа **char**), отбрасывание первой литеры (одноместная, тип результата **string**), сцепление двух строк, т. е. выстраивание литер сначала одной, а затем другой строки в одну последовательность (двухместная с результатом типа **string**), взятие длины строки (одноместная, дает результат типа **int**). Первые две операции соответствуют обычному порядку чтения текстов слева направо: сначала выделяется для анализа первая литера, затем она отбрасывается и работа продолжается с остатком строки. Часто в языки включаются и более сложные операции над строками.

Основное назначение логических выражений (а отношения считаются первичными логическими выражениями) — направить дальнейшее исполнение программы по одному из двух направлений. Во многих языках такая возможность используется, в частности, при вычислении значений выражений:

<условное выражение> ::=
<условие> <выражение> **else** <выражение> **endif**

где

<условие> ::= **if** <логическое выражение> **then**

Вычисление значения условного выражения происходит так. Проверяется выполнение условия, т. е. вычисляется значение логического выражения в нем. Если получено значение «истина» (условие выполнено), то выбирается первое внутреннее выражение, стоящее сразу за условием, если условие не выполнено, то второе — следующее за **else**. Оба выражения должны обладать значением одного и того же типа, определяющего тип всего условного выражения. Значение выбранного выражения становится значением условного выражения, значение другого даже не вычисляется (т. е. здесь разрешено использовать и такое выражение, вычислить которое было бы невозможно при этом исходе проверки).

2.3.3. Операторы и структура программы

Операторы, в отличие от выражений, — это конструкции, предназначенные для организации действий, более сложных, чем вычисление новых значений. Эти

действия сводятся к нескольким основным категориям: создать новое имя для дальнейшего использования в программе (уже знакомое нам *объявление*), вычислить значение некоторого выражения и дать ему одно из предусмотренных в программе имен, сохранив это значение для использования в будущем (*присваивание*), установить последовательность, в которой должны исполняться другие действия (*управление*).

Программа в большинстве языков организуется по *блочному* принципу и сама представляет собой *блок*, часто снабженный *заголовком*, где указано имя программы и кое-какие другие сведения для связи со *внешней средой*. Эта среда служит для программы источником исходных данных и потребителем результатов ее работы.

Блоки могут быть целиком вложены один в другой или вовсе не иметь общих частей. В начале блока — его *преамбуле* — размещаются определения констант и типов, объявления переменных и процедур. За преамбулой следует оператор — *тело блока*. Каждое из определений и объявлений вводит в употребление некоторый идентификатор. Этот идентификатор считается *локализованным* во всем блоке, за исключением тех внутренних блоков, в которых определен или объявлен объект с таким же идентификатором. Объект, представленный в данном блоке локальным идентификатором, может использоваться в нем только в полном соответствии со своим определением или объявлением. Внешнему блоку этот объект не доступен. Если во внутреннем блоке этот (точнее, такой же) идентификатор не локализован заново, то в этом блоке он считается *глобальным* и продолжает использоваться в прежней роли.

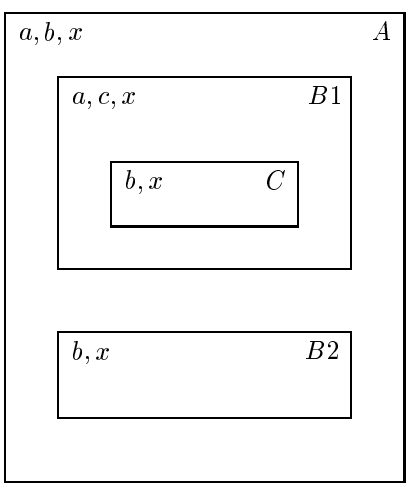


Рис. 1

Пример. Пусть в блоке A содержатся два внутренних блока: $B1$ и $B2$, а в блоке $B1$ — внутренний блок C. Пусть также в блоке A локализованы идентификаторы a, b и x, в блоке $B1$ — a, c и x, в блоках $B2$ и C — b и x. Тогда идентификатор a будет локальным в блоках A и $B1$ (обозначая в последнем новый объект с новыми свойствами) и глобальным в блоке C, где он наследует свои свойства от блока $B1$, и в блоке $B2$, с наследованием от блока A. Идентификатор b локален в блоках A, $B2$ и C, обозначая в каждом разные объекты, и глобален в блоке $B1$. Идентификатор c локален в блоке $B1$, глобален в C и не существует для блоков A и $B2$. Идентификатор x локален во всех четырех блоках, всюду со своим значением.

Кроме программы, блоки возникают в связи с каждым объявлением процедуры. Иногда разрешается создавать блоки в качестве других частей программы, о чем мы еще немного скажем в конце раздела.

Синтаксис (оператора) *присваивания*:

<присваивание> ::= <переменная> := <выражение>

где ':=' (или просто '=') называется символом присваивания, <переменная> — *левой частью* или *получателем*, а <выражение> — *источником* (*правой частью*). Получатель — это и есть то имя, которое в результате исполнения присваивания приобретает новое значение — вычисленное значение источника, безвозвратно теряя старое. Так, при исполнении оператора $n := n + 1$ новое текущее значение переменной n становится на 1 больше предыдущего.

Тип значения источника должен быть подмножеством (см. конец разд. 2.2.2) типа получателя. Например, разрешено присваивать вещественным переменным целочисленные значения. Реализация языка должна предусматривать преобразование значений из одного представления в другое. В более сложных случаях свойства нового значения получателя могут стать менее общими, чем это следует из правил его вычисления. В языках, где используются обобщенные типы (например, массивы без явного задания границ индексов), нужны специальные средства для уточнения этих свойств (для определения границ индексов присвоенного значения типа массива). Каковы эти средства — уточнять не будем.

При присваивании одной переменной значения другой создается копия этого значения. Поэтому значения обеих переменных после присваивания никак не связаны друг с другом. Новое присваивание одной из них не сказывается на значении другой. Об особенности, связанной с использованием ссылок, будет сказано в разд. 2.3.4.

Любые присваивания можно делать только локальным или глобальным переменным исполняемого в данный момент блока. По завершении исполнения блока глобальные переменные сохраняют полученные в нем значения, а локальные перестают существовать вместе со своими значениями. Если такие же идентификаторы были локализованы и за пределами завершившегося блока, то их текущие значения на момент входа в этот блок восстанавливаются.

Если требуется присвоить одно и то же значение нескольким переменным, то не хотелось бы многократно вычислять значение одного и того же выражения (к тому же повторное вычисление может дать уже другое значение, как показывает последний пример). Не очень хочется и передавать значение от одного имени к другому: $m := n$. Поэтому часто допускаются присваивания с несколькими получателями и одним общим источником, например

$$m := n := n + 1$$

Иногда также разрешается ради краткости совмещать объявление переменной с присваиванием ей начального значения по схеме

var n : **int** := 0

Разумеется, перед присваиванием нового значения переменной с индексами (на любом уровне) следует зафиксировать их значения (см. разд. 2.3.1).

Упражнение 2.3.1. Пусть перед присваиванием $a[a[2]] := 1$ переменные $a[1]$ и $a[2]$ имеют значение 2. Каким значением будут обладать переменные $a[1]$, $a[2]$ и $a[a[2]]$ после исполнения присваивания?

Если присваивания переменным с составным значением разрешены (не во всех языках это так), то при присваивании такой переменной нового значения изменятся значения всех ее подкомпонент. Если же присваивание делается переменной, обозначающей подкомпоненту составного значения, то меняются значения лишь подкомпонент, содержащих эту подкомпоненту в своем составе. Так, в примере из разд. 2.2.2, если присваивание *table.marklist*[*i*].*mark*:= «отлично» исполняется при текущем значении i, равном 7, то считается, что меняется содержимое всей экзаменационной ведомости, но эту оценку получит, увы, только студент по фамилии *mtable.marklist*[7].*name*.

Присваивание входит как составная часть в исполнение некоторых других операторов. Похожую функцию выполняют и *операторы ввода*. Обо всем этом речь пойдет ниже в настоящем и последующих разделах.

Рассмотрим основные виды *операторов управления*. Простейший из них — *пустой оператор*. Часто он изображается каким-либо символом, например **skip**, но отсутствие всякого текста выразительнее напоминает о том, что при его исполнении ничего делать не надо.

Другие операторы управления называются также *композициями*. *Последовательная* композиция:

<последовательность операторов> ::= <оператор>
| <последовательность операторов>; <оператор>

предписывает, чтобы ее операторы исполнялись подряд слева направо.

Условная композиция ставит исполнение одного из двух операторов в зависимость от результата проверки некоторого условия:

<условный оператор> ::=
<условие> <оператор> **else** <оператор> **endif**

При положительном исходе проверки выбирается для исполнения оператор, непосредственно следующий за условием, при отрицательном — оператор, следующий за символом **else**.

Если этот последний оператор пуст, то и символ **else** обычно не ставится (краткая форма условного оператора). В программах нередко случается, что выбор действия зависит от результата нескольких следующих друг за другом проверок — до первой завершившейся успешно.

Пусть, например, переменной *sgnx* требуется присвоить значение 1 при положительном значении переменной x, -1 — при отрицательном, и 0 — при $x = 0$. Для этого можно написать оператор

if $x > 0$ **then** *sgnx* := 1
else if $x < 0$ **then** *sgnx* := -1
else *sgnx* := 0
endif
endif

Но часто в подобных случаях разрешается каждую пару стоящих рядом символов **else if** заменять одним символом **elif**, а группу оказавшихся рядом символов **endif** заменять одним:

if $x > 0$ **then** $sgnx := 1$
elif $x < 0$ **then** $sgnx := -1$
else $sgnx := 0$
endif

что и в одну строчку смотрится неплохо:

if $x > 0$ **then** $sgnx := 1$ **elif** $x < 0$ **then** $sgnx := -1$ **else** $sgnx := 0$ **endif**

В близком родстве с условными операторами, особенно в только что рассмотренной форме, состоят *операторы выбора* (по метке). Предполагается, что исполнение программы разветвляется в соответствии с одной из нескольких заранее известных ситуаций, снабженных именами в виде целочисленных значений, строк или идентификаторов из некоторой шкалы (см. разд. 2.2.1).

<оператор выбора> ::=
case <выражение> **of**
<последовательность ветвей>
endcase
<последовательность ветвей> ::= <ветвь>
| <последовательность ветвей>; <ветвь>
<ветвь> ::= <метка>:<оператор> | <метка>:<ветвь>
<метка> ::= <обозначение значения>

Выбирающее выражение между **case** и **of** должно вырабатывать значение того типа, которому принадлежат все метки, среди которых не должно быть одинаковых. Исполнение оператора выбора начинается с вычисления значения этого выражения. Затем находится и исполняется оператор из ветви, содержащей полученное значение в качестве одной из меток.

Как завершить исполнение, если значение выбирающего выражения среди меток ветвей не встретилось? Обычно при этом ничего больше не делается. Однако к последовательности ветвей может быть добавлена в конце особая ветвь, своеобразной меткой которой служит символ **else**. Оператор из этой ветви исполняется лишь в указанной ситуации.

Циклическая композиция предписывает повторное исполнение некоторого внутреннего оператора до тех пор, пока существуют условия для этого. Внутренний оператор называют также *телом цикла*. Программный текст, окаймляющий тело, определяет условие продолжения или завершения исполнения цикла. Все вместе составляет *оператор цикла*. Существует несколько видов циклов.

<цикл с предпроверкой> ::=
while <логическое выражение> **do**
<оператор>
enddo

Здесь, как и ниже для циклов других видов, оператор — это тело цикла. Исполнение такого цикла начинается с вычисления значения выражения между **while** и **do**. Если это — «истина», то исполняется тело цикла, и все повторяется вновь до тех пор, пока не будет получено значение «ложь», на чем исполнение цикла завершается.

<цикл с послепроверкой> ::=
repeat
<оператор>
until <логическое выражение>

Здесь логическое выражение попало в самый конец оператора. Исполнение цикла начинается с исполнения тела, после чего значение «истина» логического выражения означает, что исполнение цикла следует завершить, а значение «ложь» возвращает исполнение на новый виток.

<цикл с параметром> ::=
for <параметр> := <начальное значение>
step <шаг>
to <предельное значение>
do <оператор>
enddo
<параметр> ::= <идентификатор>

Параметр цикла рассматривается в его теле как переменная типа **int**. «Начальное значение», «шаг» (изменения параметра) и «предельное значение» должны быть арифметическими выражениями с целочисленными значениями. С их вычисления начинается исполнение цикла, сразу после чего параметру присваивается указанное начальное значение. Значение шага не должно быть нулевым. Оно, как и предельное значение, вычисляется однократно. Цикл завершается, если текущее значение параметра перешло через предельное значение (с учетом знака шага). Подробнее условие завершения выглядит так:

шаг $>$ 0 **and** параметр $>$ предельное значение **or**
шаг $<$ 0 **and** параметр $<$ предельное значение

Отсюда видно, что цикл может завершиться, даже если его тело не исполнится ни разу (например, если начальное значение больше предельного при положительном шаге). Если же условие завершения не выполнено, то исполняется тело цикла (с текущим значением параметра), затем значение параметра изменяется на величину шага и вновь проверяется условие завершения.

Мы описали один из возможных вариантов синтаксиса и семантики операторов цикла. В различных языках все делается не совсем так, но основа сохраняется.

Оператор вывода не принадлежит к числу операторов управления, но по своему действию он (если не программировать особо изощренно, т. е. крайне неразумно) эквивалентен пустому оператору, так как не меняет значений переменных программы и не сказывается на порядке исполнения операторов.

К средствам управления исполнением программы следует причислить и *процедуры*. Они, в отличие от средств композиции, далеко относят в тексте программы место описания действия от места или мест, где инициируется его исполнение (см. разд. 2.4.1).

В еще большей степени этим грешат не случайно упоминаемые нами лишь в последнюю очередь *операторы перехода*:

<оператор перехода> ::= **go to** <метка>
<оператор> ::= <непомеченный оператор>
| <метка>:<оператор>

Семантика оператора перехода — найти в программе оператор, помеченный указанной меткой, и перейти к его исполнению, тем самым безвозвратно, в отличие от процедур, нарушив естественный порядок исполнения операторов в их последовательности.

Бурная дискуссия конца 60-х о хорошем, «структурированном», стиле программирования привела к выводу, что последовательная, условная и циклическая композиции вместе с аппаратом процедур составляют основу этого стиля. Но бывают случаи, когда соблазн призвать на помощь операторы перехода весьма велик.

Остановимся для примера на задаче слияния двух упорядоченных массивов a и b в один — массив c. Когда один из массивов a или b исчерпан, остается переслать в массив c остаток другого массива. Пока этого не случилось, надо пересылать в c меньший из еще остающихся элементов исходных массивов. Пусть их длины — m и n. Соответствующий участок программы может быть таким:

$i := 1;\ j := 1;\ k := 1;$
while $k <= m + n$ **do**
  if $i > m$ **then go to** BC
  elif $j > n$ **then go to** AC
  elif $a[i] < b[j]$ **then**
$AC:$ $c[k] := a[i];\ i := i + 1$
  else
$BC:$ $c[k] := b[j];\ j := j + 1$
  endif;
  $k := k + 1$
enddo

Здесь явно отражены два пути, приводящие к необходимости пересылки в c из a, и сходные пути, требующие пересылки из b. Поскольку переходы происходят из одной ветви условного оператора в другую, можно надеяться, что исполнение обеих ветвей закончится благополучно (в «Паскале» так и происходит). Но переходить внутрь любой композиции извне настоятельно не рекомендуется.

Попробуем написать тело цикла в рассматриваемом участке программы без операторов перехода:

if $j > n$ **or** $i <= m$ **and** $a[i] < b[j]$ **then**
  $c[k] := a[i];\ i := i + 1$
else
  $c[k] := b[j];\ j := j + 1$
endif;
$k := k + 1$

Довольно легко удалось не только устранить операторы перехода, но и сократить программу. Правда, условие может показаться более сложным. Но при

написании условий следует придерживаться простого правила: если пути, ведущие к некоторому оператору, сначала расходятся, то условия прохождения по этим путям соединяются логической связкой **or**, а если вдоль пути требуется проверить несколько условий, то их следует объединить с помощью **and**. Существенно, чтобы при этом значения логических выражений вычислялись экономно — как только выясняется, что $j > n$, альтернативная ветвь выражения уже не может изменить значение дизъюнкции, она и не вычисляется, то же самое относится к бесполезной проверке отношения $a[i] < b[j]$ при $i > m$. Реализация языка обязана предусматривать эту экономию бесполезных, а то и невозможных, вычислений. Если соблюсти правильный порядок членов в дизъюнкциях и конъюнкциях — коммутативный закон в программах не работает, то переходы, явно включенные в предыдущий вариант программы, в машинной программе появятся сами.

Продолжая эту мысль, укажем, что бывает естественно сделать в программе блоком ветвь условного оператора или оператора выбора, поскольку ветви становятся независимыми друг от друга. Можно говорить также о наличии некоторых неявных блоков. Например, оператор цикла с параметром иногда считают блоком, в котором объявлена одна переменная — параметр цикла, остальные объекты остаются глобальными. При таком соглашении значение параметра цикла становится недоступным по завершении исполнения цикла.

2.3.4. Работа со ссылками

Формальная семантика ссылок была определена в разд. 2.2.1. Дадим некоторые дополнительные разъяснения.

Наглядно представить характер значений и действий, в которых принимают участие ссылки, помогает следующая графическая схема. Допустим, что мы имеем дело с простыми значениями лишь двух типов: **real** (числа) и **ref** (ссылки), а также с векторами и записями, составленными из таких же значений. Будем изображать число длинным прямоугольником (рис. 1 а), а ссылку — коротким (рис. 1 б).

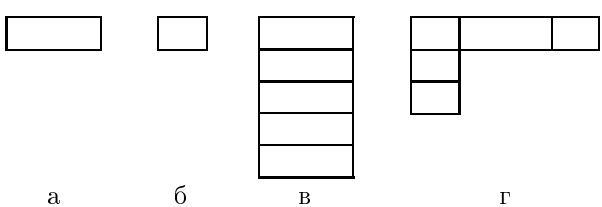


Рис. 1

Вектора и записи будем изображать фигурами, составленными из фигур, изображающих их компоненты и расположенных соответственно одна под другой или одна рядом с другой по горизонтали. На рис. 1в изображен вектор из 5 чисел, а на рис. 1г — запись с тремя полями, где первое поле — вектор из трех ссылок, второе — число, третье — ссылка.

Связь между именем (идентификатором или ссылкой) и обозначаемым им значением изобразим в виде стрелки, ведущей от имени к значению. Имена на схемах могут быть явные — идентификаторы и косвенные — ссылки. На рис. 2а изображена переменная с объявлением **var** x : **real**, а на рис. 2б — переменная с объявлением **var** p : **ref real** (или **var** p : **ref** T, где T — любое обозначение типа). Значение, присвоенное вещественной переменной, можно вписать внутрь соответствующего прямоугольника (рис. 1в), но обычно мы не будем этого делать.

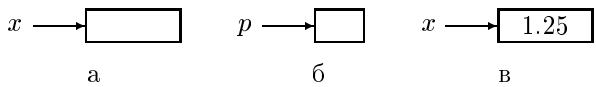


Рис. 2

Переменная типа **ref** T имеет своим значением ссылку (косвенное имя). Если это имя действительно ссылается на некоторое другое значение, то ссылку изобразим точкой, поставленной внутри прямоугольника, изображающего переменную, и от этой точки проведем стрелку к фигуре, изображающей значение. Так, на рис. 3а показано, что значение переменной p именует число 3.14. Если же имя ни на что не ссылается (такую ссылку мы условились обозначать **nil**), то прямоугольник, изображающий это значение, перечеркнем наискось (рис. 3б).

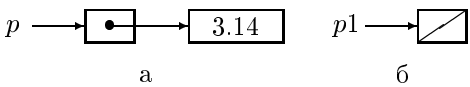


Рис. 3

Что меняется в подобной схеме после исполнения присваивания? Объект-получатель представлен на схеме некоторой фигурой, к которой ведет стрелка от его имени — идентификатора или ссылки. Изменение значения объекта отображается изменением содержимого прямоугольников, из которых составлена эта фигура, поскольку присваивание значений составным объектам сводится к присваиваниям подкомпонентам этих объектов вплоть до простых.

Рис. 2в отображает результат присваивания $x := 1.25$, а рис. 3б — результат присваивания $p1$:= **nil** (такое присваивание допустимо, если переменная $p1$ имеет тип **ref** T, каков бы ни был тип T). Что касается рис. 3а, то возможны два пути возникновения подобной ситуации.

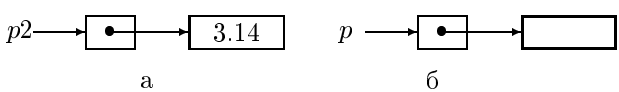


Рис. 4

Либо некоторая переменная $p2$ того же типа **ref real**, что и p, уже имела значением ссылку на число 3.14 (рис. 4а) и было исполнено присваивание $p := p2$, либо значением p была ссылка на какое-то число (рис. 4б) и оператор $p\uparrow := 3.14$

заменил его на 3.14. В первом случае изменилось содержимое левого (короткого) прямоугольника, во втором — правого (длинного).

Однако рис. 4а не вполне отображает результат присваивания $p := p2$. Значение $p2$ копируется при таком присваивании, но значение $p2\uparrow$ не копируется, поэтому рис. 5 более сооответствует возникшей ситуации.

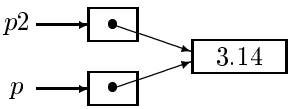


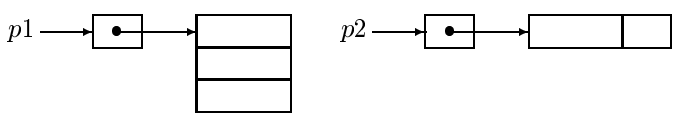
Рис. 5

Теперь значения p и $p2$ — это одна и та же ссылка (имя содержимого длинного прямоугольника). Поэтому очередное присваивание одной из переменных $p\uparrow$ или $p2\uparrow$, например $p\uparrow := 1.25$, изменит значение и другой переменной (т. е. $p2\uparrow$).

Поскольку объявление переменной обычно не задает ее начального значения, ясно, что описание **var** p : **ref real** способно породить ситуацию, отображенную на рис. 2б, но не на рис. 4б. Последняя впервые может возникнуть только в результате присваивания некоторой переменной значения выражения, называемого *генератором*:

<генератор> ::= $new(< mun >)$

Каждое исполнение генератора создает объект указанного типа с неопределенным значением и ссылку на этот объект, которая и является значением генератора. Чтобы это значение тут же не пропало, оно должно быть присвоено некоторой переменной. В частности, рис. 4б отражает результат исполнения присваивания $p := new(\textbf{real})$, рис. 6а — присваивания $p1 := new(\textbf{array}\,[1\,..\,3]\,\textbf{of real})$,



а б

Рис. 6

а рис. 6б — результат присваивания $p2 := new(\textbf{record}\ a : \textbf{real};\ b : \textbf{ref}\ \mathrm{T}\ \textbf{endrec})$. В последнем случае идентификаторы a и b могут быть и иными, а тип T — произвольным.

Объекты, содержащие в своем составе ссылки, принято называть *динамическими*. Имеется в виду, что при присваивании ссылкам значений генераторов в графической схеме объекта возникают новые части, а при присваивании значения **nil** — из вида исчезают старые. При появлении в объекте новых ссылок возникают и новые возможности преобразования схемы, разрастание которой ничем не ограничено.

Более сложный пример программы, интенсивно работающей со ссылками, содержится в разд. 2.5.1.

2.4. Более сложные средства

2.4.1. Процедуры

Аппарат процедур позволяет дать обозначение некоторому оператору — *телу процедуры*, с тем, чтобы этот оператор мог быть исполнен как бы в том месте программы, где возникла потребность в описываемых им действиях, и где для этой цели содержится *обращение к процедуре* (в другой терминологии — ее *вызов*). Тело процедуры обычно содержит некоторые идентификаторы, называемые (*формальными*) *параметрами* процедуры. При обращении к процедуре, непосредственно перед исполнением ее тела, эти параметры либо получают значения соответствующих *аргументов* обращения (*фактических параметров*), либо заменяются этими аргументами. Первый вариант называется *вызовом* параметров *значением*, второй — вызовом *по имени*. Во втором случае в современных языках требуется, чтобы аргумент и был именем (т. е. переменной, идентификатором процедуры или файла и т. п.), хотя иногда допускается замена параметров аргументами более общего вида.

Процедуры делятся на два класса: *собственно процедуры*, обращения к которым рассматриваются как операторы (да и являются ими), и *процедуры-функции*, для которых обращения выступают в роли выражения и служат для вычисления значения. Для процедур обоих классов действует описанная выше общая схема обращения, хотя детали несколько различаются. Главное отличие — в теле процедуры-функции должен содержаться оператор специального вида, вырабатывающий требуемое значение функции. Таких операторов может быть даже несколько в разных ветвях тела процедуры-функции, но только один из них должен быть выполнен при исполнении этого тела.

Процедуры (обоих классов), которые во время исполнения обращения к ним могут прямо или косвенно (через другие процедуры) обратиться сами к себе, называются *рекурсивными*. Споры о допустимости использования в языках рекурсивных процедур остались далеко в прошлом.

Процедуры, которые предполагается использовать в некотором блоке (но не за его пределами), должны быть объявлены в начале (преамбуле) этого блока. Синтаксис *объявления процедуры* продемонстрируем на примере языка «Паскаль» с небольшими отступлениями.

<объявление процедуры> ::=
  proc <идентификатор> (<список параметров>); <блок>
<список параметров> ::= <спецификация параметра>
  | <список параметров>; <спецификация параметра>
<спецификация параметра> ::=
  <идентификатор>:<обозначение типа>
  | **var**<идентификатор>:<обозначение типа>

Идентификатор в объявлении процедуры — это ее идентификатор, в спецификации параметра — идентификатор этого параметра. Спецификация без символа **var** относится к параметру, вызываемому значением, а с этим символом — к параметру, вызываемому по имени. Разрешены процедуры без параметров, в этом

случае список последних отсутствует вместе со скобками. Разрешена группировка параметров одного класса (по способу вызова) и одного типа, как это делается в объявлениях переменных.

Объявление процедуры-функции отличается тем, что сразу за списком параметров, если он есть, а если нет — то за идентификатором процедуры, следует символ ':' и обозначение типа значения функции (ее *результата*). Для большей выразительности символ **proc** заменяется на **function**.

Оператор, вырабатывающий значение функции, внешне не отличается от присваивания, в котором получателем служит идентификатор данной функции. Однако это сходство — чисто внешнее, значение этого получателя уже не может быть использовано в пределах тела функции, а только в качестве значения вызова функции.

Блок в объявлении процедуры — ее *тело*, состоит, как уже говорилось, из преамбулы — перечня описаний и объявлений объектов, локализуемых в блоке, и оператора (чаще всего, последовательной композиции), заключенного между символами **begin** и **end**. Описания и объявления отделяются друг от друга и от этого оператора точками с запятой.

Оператор процедуры выглядит так:

<оператор процедуры> ::=
<идентификатор процедуры> (<список аргументов>)

Идентификатор (глобальный или локальный в блоке, содержащем данный оператор) указывает, к какой процедуре происходит обращение. Список аргументов содержит столько же элементов, сколько параметров у этой процедуры (т. е. может и отсутствовать). Соответствие между аргументами и параметрами устанавливается по порядку их задания. Аргумент должен быть выражением при вызове значением и именем — при вызове по имени. Аргумент должен быть согласован по типу с соответствующим ему параметром. Это значит, что типы аргумента и параметра должны совпадать, или тип аргумента должен быть подмножеством типа параметра в той мере, в какой это принято в данном языке (см. конец разд. 2.2.2). Вызов функции имеет точно такой же вид, но семантически является выражением.

В различных языках можно встретить много мелких отклонений от этих правил записи и исполнения процедур и функций. Иногда эти детали переходят из одного языка в другой, иногда изобретаются заново. Есть причины, по которым изложенные выше правила следует считать недостаточными. Возможен *конфликт идентификаторов* — тех, которые локальны или глобальны в теле процедуры, с теми, которые попадают в него при вызове по имени. Обращение к рекурсивной процедуре может встретиться в тексте программы раньше, чем ее объявление. Хорошо было бы к этому моменту просмотра текста знать, как может быть истолковано такое обращение. Объявление процедуры-функции с одним или двумя параметрами без особого труда может быть преобразовано в описание операции — одноместной или двухместной. Ограничимся этим и не будем подробнее останавливаться на деталях.

2.4.2. Алгоритмы над графами

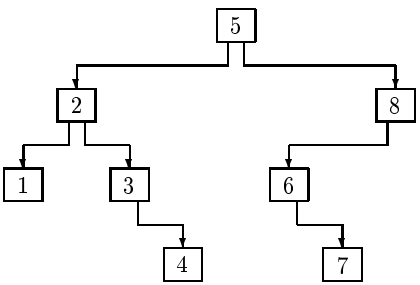
Для закрепления сведений, изложенных в предыдущих разделах, приведем примеры алгоритмов, на которые будем ссылаться в разд. 3.

Пусть узел двоичного дерева представлен (на языке «Паскаль») записью типа

$Rec =$ **record** $val : valtype$; $left$, $right : RefRec$ **end**

Поле val каждого узла хранит значение типа $valtype$, связанное с этим узлом, поля́ $left$ и $right$ — это указатели на корни левого и правого поддеревьев. Переменная $tree$ типа **type** $RefRec = \uparrow Rec$ содержит ссылку $ad1$ на корень дерева.

На рис. 1 изображен пример дерева и соответствующей ему структуры памяти.



ad	val	$left$	$right$
$ad1$	5	$ad2$	$ad4$
$ad2$	2	$ad5$	$ad3$
$ad3$	3	**nil**	$ad7$
$ad4$	8	$ad6$	**nil**
$ad5$	1	**nil**	**nil**
$ad6$	6	**nil**	$ad8$
$ad7$	4	**nil**	**nil**
$ad8$	7	**nil**	**nil**

Рис. 1

Рекурсивная процедура $Traverse$ левостороннего обхода дерева с обработкой значений, сопоставленных узлам, в порядке их второго посещения, т. е. по схеме:

1) обход левого поддерева,
2) обработка данных в корне,
3) обход правого поддерева,

может выглядеть так:

```
procedure Traverse(ptr : RefRec);        {ptr — указатель на корень}
begin
  if ptr ≠ nil then    {Первое посещение корня}
  begin
    Traverse(ptr↑.left);
    Visit(ptr↑);    {Второе посещение}
    Traverse(ptr↑.right)
  end
end {Traverse}
```

Текст процедуры снабжен примечаниями в фигурных скобках. Подробнее о примечаниях будет сказано в разд. 2.4.4.

Здесь процедура $Visit$ должна описывать действия, исполняемые при втором посещении корня. Если обработка сводится, например, к печати значения,

то оператор $Visit(ptr\uparrow)$ можно заменить на $Write(ptr\uparrow.val)$. Если у узла (как, например, у любого листа) нет левого поддерева, то оператор $Traverse(ptr\uparrow.left)$ срабатывает впустую, так что второе посещение следует непосредственно за первым, и процедура *Visit* обследует и такие узлы.

Обход всего дерева начинается при обращении к процедуре с помощью оператора $Traverse(tree)$.

О вариантах этой процедуры будет рассказано в разд. 3.2.9.

Теперь опишем алгоритм (тело процедуры) построения входящего каркаса, имеющего корень в стоке *Sout* ориентированного графа с множеством вершин U и множеством дуг T (в разд. 3.1 такой граф будет операторной схемой некоторого алгоритма, U — множеством ее операторов, T — множеством переходов).

Описание дадим на паскале-подобном языке. Тип переменных U, P, Q, R, $P1$ — множество вершин графа, переменные S, $S1$ и $S2$ имеют тип «вершина», переменные T, $N1$ и $N2$ — тип «множество дуг» (пар вершин). Каркас будем строить по уровням, P — последний уже построенный уровень, $P1$ — строящийся уровень, R — объединение всех построенных уровней, Q — множество вершин, еще не отнесенных ни к одному из уровней. Главная задача алгоритма — построить множество $N1$ дуг каркаса и множество $N2$ выбрасываемых из T (не включаемых в каркас) дуг.

В выражениях используются операции над множествами со знаками, взятыми из языка «Паскаль». Цикл с заголовком **for** S **in** Q **do** — это цикл по всем элементам S множества Q.

```
P := R := [Sout]; Q := U − R; N1 := N2 := [ ];
repeat P1 := [ ];
  for S in Q do
    for S1 in P do
      if (S, S1) in T then
      begin P1 := P1 + [S]; R := R + [S]; Q := Q − [S];
        N1 := N1 + [(S, S1)];
        for S2 in R do
          if (S2, S) in T then N2 := N2 + [(S2, S)];
        for S2 in (P − [S1]) + P1 do
          if (S, S2) in T then N2 := N2 + [(S, S2)]
      end;
  P := P1
until (Q = [ ]) ∨ (P = [ ])
```

Алгоритм построения выходящего остовного дерева с корнем во входе получим, если *Sout* заменим на *Sin* и переставим операторы в каждой паре: $(S, S1)$ заменим на $(S1, S)$ и т. д.

В этих двух примерах были использованы два способа представления графов: в виде динамических ссылочных систем (для частного случая двоичных деревьев) и в виде множеств — в обоих случаях не вполне корректно.

Для графов общего вида каждая вершина может быть представлена записью примерно следующего типа

Vertex = **record** *val* : *valtype*; *outarcs*, *inarcs* : *RefArc* **end**

где *val* — по-прежнему, сведения о вершине, *outarcs* и *inarcs* — указатели типа *RefArc*=↑*Arc* на начала (т. е. на первые элементы) списков исходящих и заходящих дуг (второй список нужен лишь для того, чтобы получить возможность перемещаться по дугам от вершины к вершине в любом направлении). Тип элемента списка может быть таким:

Arc = **record** *mark* : *marktype*; *neighbour* : *RefVert*; *nextarc*: *RefArc* **end**

где *mark* — сведения о дуге (ее пометка), *neighbour* — указатель на второй конец дуги, *nextarc* — указатель на следующую дугу в данном списке.

Даже в «Паскале» — одном из немногих языков, где существует понятие множества, невозможно создавать множества с элементами произвольных типов, в частности, таких, которые были использованы во втором примере. Их приходится заменять массивами, что приводит к более громоздкой записи операций над множествами, но и к более экономному их исполнению.

Есть и другие способы представления графов, из которых остановимся на двух. *Матрицей смежности* ориентированного графа G с n вершинами и m дугами называется логическая $n \times n$-матрица *adj*, для которой *adj*[*i*, *j*] = *true* означает, что в графе имеется дуга $\langle i, j \rangle$. Аналогично можно определить матрицу достижимости *att*. Процедура *Attainability* по матрице смежности строит матрицу достижимости, удваивая (по меньшей мере) на каждом витке основного цикла длину возможных путей от одной вершины к другой.

```
type diap = 1..n; AdjType = array[diap, diap] of Boolean;
var adj, att: AdjType;
procedure Attainability(n: byte; adj: AdjType; var att: AdjType);
  var i, j, k: byte; q: Boolean;
begin att := adj;                          {Все смежные вершины достижимы}
  begin for k := 1 to n do att[k, k] := true;          {Сама вершина также}
  repeat q := true;              {Пока что нет новых недостижимых вершин}
    for i := 1 to n do for j := 1 to n do          {Цикл по элементам adj}
        if adj[i,j] then
         {Достижимость обнаружена на прошлом или на данном витке}
        begin adj[i,j] := false;              {Пометка с вершины снимается}
          for k := 1 to n do          {Ищем новые достижимые вершины}
            if (k <> i) and not att[i,j] and (k <> j) and att[j,k] then
                {Если k-я вершина достижима из j-й, то и из i-й тоже}
            begin q := false; adj[i,k] := true; att[i,k] := true;
              {Впервые обнаружилась достижимость вершины att[i,k]}
            end {for k}
        end {for i и j}
  until q              {Выход, если новых достижимых вершин не нашлось}
end {Attainability};
```

Текст весьма скупых примечаний, призванных обосновать правильность алгоритма, сравним по длине с текстом программы. Ниже, в разд. 3.1.2, 3.2.2, 3.2.8 и др., мы убедимся, что строгое доказательство правильности еще более объемно, а в общем случае может быть проведено только параллельно с исполнением программы (ср. с разд. 1.7.5).

В эксперименте при $n = 20$ матрица достижимости строится (с контролем) за три витка цикла **repeat** ... **until**. Идея алгоритма принадлежит В. В. Мартынюку [33], впрочем в наши дни найти первооткрывателя подобных идей — дело почти безнадежное.

И, наконец, *матрица инцидентности* для графа G — это целочисленная $n \times m$-матрица inc, причем $inc[i, j] = 1$ означает, что j-я дуга исходит из i-й вершины, а $inc[i, j] = -1$ — что эта дуга заходит в нее.

2.4.3. Файлы и операторы для работы с ними

Файл (от английского file — папка для подшивки бумаг, скоросшиватель) — это еще одна разновидность объектов с составными значениями. В машинных терминах — это объекты, размещаемые на *внешних носителях* данных: дискетах, магнитных дисках и лентах и т. п. Иначе все это называется внешней памятью компьютера. Доступ к компонентам файлов по адресу, как и для данных во внутренней, собственно машинной, памяти, невозможен или затруднен. Применяются другие механизмы доступа, реализуемые специальными *операторами обмена* данными между машинной и внешней памятью. С некоторой натяжкой к данным, размещаемым в файлах, относят и все то, что поступает в компьютер из внешней среды (см. разд. 2.3.3) или передается в нее из компьютера.

Способы работы с файлами довольно разнообразны, соответственно классифицируются и сами файлы. Различают два основных *метода доступа* к файлам: последовательный и прямой. При *последовательном* доступе все компоненты файла упорядочиваются в линейную последовательность. Добавлять к ней новые компоненты можно только в конце, а извлекать — поочередно, начиная с первой. При *прямом* доступе такой упорядоченности компонент файла нет, но каждая компонента снабжается *признаком* или *ключом*, вместе с которым она помещается в файл и по которому впоследствии может быть извлечена из него.

По *режиму работы* с ним файл может быть предназначен либо только для выборки или, как чаще говорят, *чтения* его компонент, либо только для занесения или *записи* в него новых компонент, либо, наконец, для *переработки* (записи и чтения вперемежку в некотором допустимом порядке). Для файлов с прямым доступом этот порядок практически произволен. Эти режимы работы могут быть как постоянными — на все время существования файла для программы (физически файл может существовать на своем носителе данных до начала работы программы с ним и после окончания этой работы), так и временными — на какой-то период исполнения программы. При любом режиме работы обмен данными (их чтение или запись) может происходить только отдельными компонентами, но не их группами и не частями. Особых неудобств это не создает, так как размер

компоненты можно назначать в достаточно широких пределах. Если эти компоненты — составные значения, то перед записью в файл они должны быть предварительно сформированы целиком. При чтении компоненты тоже извлекаются целиком и становятся значением одной из переменных программы. Потом подкомпоненты этого значения можно извлекать уже традиционными средствами.

Файлы могут быть оснащены *буфером* — особой переменной, только через которую возможен обмен данными с файлом. При отсутствии буфера должен быть явно указан тот программный объект, значение которого участвует в данной операции обмена (обычно это переменная, но можно записывать в файл и значение выражения без промежуточного присваивания этого значения какой-либо переменной).

Метод доступа, режим работы и буферизация (наличие буфера) называются *характеристиками* файла. Они имеют специальные обозначения и должны быть заданы в описании типа файла или же (это относится только к режиму) в операторе «открыть» файл (см. ниже):

<тип файла> ::=
file of <тип компонент> <доступ> <режим> <буферизация>
<доступ> ::= **seq** | **dir** <тип ключа>
<режим> ::= | **in** | **out** | **update**
<буферизация> ::= | **buf**

Символ **seq** обозначает последовательный доступ к файлу, а **dir** — прямой с указанием на тип ключа, по которому осуществляется доступ. Режим работы обозначается символами **in** (для записи), **out** (для чтения) или **update** (для переработки). Допускается (что представлено пробелом между символами '::=' и '|') отсутствие какого-либо из этих символов. Это означает, что режим может меняться во время исполнения программы и должен быть задан при каждом открытии файла. Символ **buf** означает, что файл снабжен буфером и весь обмен данными между программой и файлом может идти только через него. Если этот символ в типе файла не указан, то у файла буфера нет и обмен идет так, как это было сказано выше.

Файлы, работа с которыми предполагается в том или ином блоке программы, должны быть доступны (локальны или глобальны) в нем.

<объявление файла> ::=
const <имя файла>:<тип файла>=<внешнее имя файла>

Здесь символ **const** подчеркивает, что объявленные характеристики файла меняться уже не могут, <имя файла> — это идентификатор, под которым файл становится известен программе, он локализуется в блоке подобно другим идентификаторам. Файлы, входящие компонентами в составные значения, отдельно не объявляются, их типы задаются в качестве типов этих компонент.

<Внешнее имя файла> — это то имя, под которым файл (существование которого, как было сказано выше, не связывается с одной определенной программой и, тем более, с одним ее исполнением) известен во внешней среде, что-либо вроде $'C:\backslash MYDATA\backslash FILE\_A'$.

Для файлов с последовательным доступом имеет смысл говорить об *общем числе компонент* файла и о *числе обработанных* (после открытия файла) *компонент*. И то и другое число может меняться, соотношение между ними определяет возможность фактического чтения или записи очередной компоненты. Сразу после исполнения объявления файла число обработанных компонент равно нулю, а общее число компонент не определено. Такое соглашение объясняется именно тем, что в программе могут использоваться ранее созданные файлы. Для таких файлов некоторые их характеристики (внешнее имя, средства защиты, общее число компонент и т. п.) могут храниться во внешней среде, образуя вместе то, что мы будем называть *паспортом* файла.

Тип компонент файла не должен предусматривать присутствие в них ссылок ни на каком уровне. Это ограничение вызвано тем, что ссылки не имеют внешнего представления, а их внутреннее представление (машинные адреса) тесно связано с каждым отдельным исполнением программы. Файлы не могут быть компонентами других файлов, это противоречило бы принципу обмена данными между программой и файлами только отдельными компонентами, но не файлами в целом.

Из файлов можно образовывать массивы и включать их в записи. Однако на использование массивов и записей, содержащих файлы в качестве своих компонент, накладываются ограничения — как и сами файлы, они не могут быть компонентами файлов, не могут участвовать в присваиваниях (поскольку их значение в целом недоступно), на них не могут делаться ссылки (так как объекты ссылок должны размещаться во внутренней машинной памяти). Таким образом, файлы — это составные значения особого рода, во многих отношениях неравноправные с обычными массивами и записями. Все это отражается и на использовании имени файла в качестве переменной.

Компоненты файлов прямых обозначений (имен) не имеют. Символ **buf** в объявлении файла говорит о существовании буфера файла. Буфер — это переменная, не требующая иного объявления и обладающая почти всеми свойствами обычных переменных. Ее тип совпадает с типом компонент файла. Ее явное имя получается из имени файла приписыванием к нему выделителя '[]'. Если компонента файла — составное значение, то к этому имени можно приписывать дальнейшие выделители. Все полученные таким образом имена, включая само имя буфера, можно помещать в любой из частей оператора присваивания. Если имя относится к объекту с простым значением, оно может входить в выражения в качестве имени операнда. Приведем примеры.

Пусть имеются следующие объявления (в них характеристики файлов опущены, всюду предполагается, что файлы буферизованы):

```
const fc : file of char;    fi : file of int;
  fac : file of array [1..80] of char;
var i : int;    b : bool;    ac : array [1..80] of char;
  af : array [1..3] of file of
    record k : int; s : array [1..72] of char endrec;
```

Допустимы следующие выражения и операторы (все они рассматриваются порознь и не образуют осмысленный участок программы; символ '<>', как в «Паскале», заменяет '≠'):

$fc[\,]$ $\quad$ $fc[\,] <> \,'"'$
$fi[\,] := 0$ $\quad$ $fac[\,][i] := 'A'$
for $i := 1$ **step** 1 **to** 80 **do** $ac[i] := fac[\,][i]$
$ac := fac[\,]$

Результат исполнения двух последних операторов один и тот же.

Следующая группа операторов присваивает переменной b значение **true**, если содержимое буфера файла $af[1]$ совпадает с содержимым буфера файла $af[2]$, и **false** — в противном случае

$b := af[1]\,[\,].k = af[2]\,[\,].k;\ i := 0;$
while $(i < 72) \wedge b$ **do**
$\quad i := i + 1;\ b := af[1]\,[\,].s[i] = af[2]\,[\,].s[i]$
enddo

Операторы, обслуживающие все потребности работы с файлами, делятся на две группы. К первой группе относятся операторы «открыть» и «закрыть» файл, начинающие и завершающие очередной этап работы с файлом в целом. Как будет видно, они особенно важны для файлов с последовательным доступом. Вторая группа — это операторы «писать», «читать», «пропустить» и «исключить», обеспечивающие работу с отдельными компонентами файла, а точнее, исполняющие операции обмена и родственные им. Тесно связаны с этими операторами встроенные функции *eof* (сокращение от end of file — конец файла) и *key* (ключ), с помощью которых можно проконтролировать, успешно ли прошла операция обмена, не встретились ли при ее исполнении особые условия, препятствующие обмену. Эти функции будут описаны ниже на с. 188.

Подготавливает файл к очередному этапу работы с ним оператор «открыть (файл)»:

<оператор открыть> ::= **open** <имя файла> <режим>

Как и во всех других операторах этого раздела, <имя файла> — это идентификатор или переменная, выделяющая компоненту типа файла из некоторого составного значения. При открытии файла исполняются некоторые операции с его паспортом (находится внешнее имя файла, проверяются права доступа и т. д.), который к этому моменту должен быть уже создан — файл должен быть объявлен. Режим может быть и не указан, это значит, что он должен был быть задан при объявлении файла. Если режим указан в обоих местах, причем поразному, то оператор не исполни́м. В остальных случаях устанавливается заданный режим работы с данным файлом.

При открытии файла с последовательным доступом общее число компонент файла берется из его паспорта. Для вновь создаваемого файла это число равно нулю. Если файл открывается для чтения или для переработки, то число обработанных компонент становится равным нулю, т. е. файл «устанавливается на

начало». Если же файл открывается для записи, то он, напротив, устанавливается «на конец» — число обработанных компонент принимается равным общему числу компонент.

При открытии файла с прямым доступом и заданным при объявлении режимом в его состоянии ничего не меняется. Иногда оператор «открыть» воспринимается как разрешение работать с файлом — применять к нему другие операторы, пока файл не будет закрыт.

Оператор «закрыть (файл)» запрещает (до нового открытия) работу с файлом, заносит в паспорт файла (с последовательным доступом) общее число его компонент, делает неопределенным число обработанных компонент и (для файлов с возможностью смены режима) режим работы файла:

<оператор закрыть> ::= **close** <имя файла>

Основные операторы, фактически исполняющие операции обмена, — это операторы «писать» и «читать»

<оператор писать> ::= **write** <имя файла>
<источник> <источник ключа>
<источник> ::= | **from** <выражение>
<источник ключа> ::= | **key from** <выражение>

Источник определяет значение, которое записывается в файл, источник ключа — значение ключа (признака), которым снабжается записываемая компонента. Как источник, так и источник ключа могут быть и не заданы.

Источник задается обязательно для не буферизованных явно файлов. Выражение в нем должно вырабатывать значение, тип которого совпадает с типом компонент файла. Источник не должен задаваться для буферизованных файлов, так как записываемое значение всегда берется из буфера.

Задание источника ключа обязательно для файлов с прямым доступом. Тип выражения в нем должен совпадать с типом, указанным в типе файла. Для файлов с последовательным доступом источник ключа, как правило, не указывается. В языке может быть разрешено указывать источник ключа и для таких файлов. При этом ключ записывается в файл вместе с очередной компонентой в следующих целях: обеспечить возможность дополнительного контроля при чтении; перейти в режим последовательного доступа только после записи в файл или обнаружения в нем компоненты с указанным значением ключа. В обоих случаях характеристика доступа в описании типа файла должна быть заменена на **seq**<тип ключа>.

При исполнении оператора «писать» обычно в файле появляется новая компонента и (для файлов с последовательным доступом) общее число компонент и число обработанных компонент увеличиваются на единицу.

Однако оператор «писать» записывает компоненту в файл не всегда, а при определенных условиях. Режим работы с файлом должен быть «для записи» или «для переработки». Кроме того, для файлов с последовательным доступом требуется, чтобы запись происходила в конец файла: число обработанных компонент должно совпадать с общим числом компонент файла. Это условие удовлетворя-

ется автоматически для вновь создаваемых файлов. Для файлов с прямым доступом необходимо, чтобы в файле не было компоненты с тем значением ключа, которое задано в операторе. При нарушении этих условий запись новой компоненты не производится, а само нарушение может быть выявлено с помощью логических встроенных функций *eof* и *key* (см. с. 188). Содержимое буфера файла при исполнении оператора «писать» не изменяется.

Синтаксис оператора «читать»:

<оператор читать> ::= **read** <имя файла>
<назначение> <ключ>
<назначение> ::= | **to** <переменная>
<ключ> ::= | **key** <выражение>

Назначение указывает, какой переменной должно быть присвоено значение читаемой компоненты файла. Тип переменной должен совпадать с типом компоненты. Для буферизованных файлов назначение не указывается, так как чтение всегда происходит в буфер. <Ключ> содержит выражение, значение которого берется в качестве ключа для поиска в файле читаемой компоненты. Тип значения выражения должен совпадать с типом ключа, указанным в характеристике доступа в описании типа файла. Для файлов с последовательным доступом в языке может быть разрешен поиск компоненты по ключу с автоматической корректировкой числа обработанных компонент. Последующее чтение компонент может происходить уже в чисто последовательном порядке. В программе может также быть предусмотрена проверка, снабжена ли очередная компонента файла именно этим ключом. Если названные возможности не предусмотрены, то ключ в операторе «читать» не задается.

Прочитанная компонента сохраняется в файле. Для файлов с последовательным доступом число обработанных компонент увеличивается на 1, общее число компонент не меняется.

Условия, при которых фактически читается компонента, заключаются в следующем. Режим работы с файлом должен быть «для чтения» или «для переработки». В файле с прямым доступом должна существовать компонента с заданным значением ключа. Для файла с последовательным доступом число обработанных компонент должно быть меньше общего числа компонент. О возможном контроле по ключу для таких файлов уже говорилось. Нарушение этих условий обнаруживается после исполнения оператора «читать» с помощью функций *eof* или *key*.

Иногда при работе с последовательным файлом в режиме чтения или переработки одна или несколько очередных компонент заведомо не представляют интереса. Через них можно перескочить, используя оператор «пропустить»:

<оператор пропустить> ::= **skip** <имя файла>
| **skip**(<выражение>) <имя файла>

Выражение в этом операторе должно вырабатывать целочисленное значение k. Форма **skip** <имя файла> оператора «пропустить» неявно задает $k = 1$. Если $k > 0$ и число обработанных компонент в сумме с k не превышает общего

числа компонент, то k компонент считаются как бы прочитанными, т. е. число обработанных компонент заменяется этой суммой. Значения пропущенных компонент никуда не попадают, содержимое буфера (если он есть) не меняется. Если $k \leqslant 0$, то состояние файла не меняется. Если упомянутая сумма оказывается больше общего числа компонент, то все имеющиеся компоненты считаются обработанными, а возникшая ситуация обнаруживается с помощью функции *eof*.

Из файла с прямым доступом можно удалить ненужную компоненту с помощью оператора «исключить»:

<оператор исключить> ::= **delete** <имя файла> <ключ>

<Ключ> задается так же, как в операторе «читать». Если в файле нет компоненты с таким значением ключа, то в нем ничего не меняется, а возникшая ситуация может быть обнаружена с помощью функции *key*.

Обращения к встроенным логическим функциям *eof* и *key* имеют вид *eof*(<имя файла>) и *key*(<имя файла>). Функция *eof* применяется только к последовательным файлам. Она имеет значение *true*, если для последнего выполнявшегося над указанным файлом оператора «писать», «читать» или «пропустить» нарушено ограничение на число обработанных компонент. Если ограничение соблюдается, то значение функции — *false*.

Таким образом, если файл f с последовательным доступом ранее использовался в режиме записи, а на очередном этапе работы программы надо прочитать и как-то обработать все компоненты этого файла, то это можно сделать с помощью операторов:

open f **in**; **read** f;
while $\neg$ *eof*(f) **do** S; **read** f **enddo**

где оператор S обрабатывает значение прочитанной компоненты, не производя никаких операций над файлом f.

Если требуется продолжить запись в ранее созданный и закрытый файл f, уже содержащий некоторое число компонент, то достаточно лишь открыть его на запись:

open f **out**

Если же файл f открыт для переработки, часть записей (возможно, даже все) из него прочитана и надо вновь начать запись в этот файл, то оператор

repeat skip f **until** *eof*(f)

создаст все необходимые для этого условия.

Функция *key* применима к файлам, компоненты которых в соответствии с его типом снабжены индивидуальным ключом. Функция имеет значение *true*, если для последнего оператора «писать», «читать» или «исключить», исполнявшегося над таким файлом, было нарушено условие возможности фактической записи, чтения или исключения компоненты, а для файла с последовательным доступом — если при исполнении оператора «читать» в режиме чтения ключ очередной компоненты не совпал со значением, заданным в операторе. В остальных случаях значение функции *key* — *false*.

С целью экономии внешней памяти все значения записываются в файлы во внутреннем (машинном) представлении. Просмотреть содержимое таких файлов с помощью обычного текстового редактора невозможно, даже если тип компонент — литера или строка. В ином положении оказываются так называемые *текстовые* файлы. Весь набор характеристик в описании текстового файла может быть заменен одним символом **text**. Подразумевается, что доступ к текстовому файлу — последовательный, а буфер отсутствует. Режим работы, как для любого файла, может быть задан оператором «открыть». Считается, что содержимое текстового файла — это последовательность литер, но разбитая на строчки (это не то же самое, что строки) произвольной, но обычно все же ограниченной длины. Операторы «писать» и «читать» служат для записи в текущую строчку или прочтения из нее значений типов **char** или **string**. Имя файла может отсутствовать, тогда имеется в виду, что данные для записи в файл поступают с клавиатуры, а вывод при чтении производится на экран монитора. Клавиатура и экран могут быть заменены иными специально предусмотренными устройствами.

Для текстовых файлов операторы «писать» и «читать» имеют наряду с обычными особые формы. Оператор **writeln** <имя файла> <источник> записывает в указанный текстовый файл значение источника вместе с признаком конца строчки, а при пустом источнике — только этот признак. Оператор **readln** <имя файла> <назначение> читает из файла данные в соответствии с указанным назначением, а остаток строчки до признака конца — пропускает. При пустом назначении весь остаток строчки читается «в воздух».

Пример 1. Продемонстрируем простейшие приемы работы с последовательными файлами. Тип компонент рассматриваемых файлов задается определением вида

type *arecord*=**record** *rkey*:**int**;... **endrec**

Из всех полей компоненты нас будет интересовать лишь поле *rkey* типа **int**, служащее ее характеристическим признаком. Сходство этого поля с признаком, по которому различаются компоненты файлов с прямым доступом, — лишь кажущееся. Признак *rkey* не используется ни при выборке компоненты из файла, ни при записи в него. Это обычное поле записи, и его особая роль вытекает лишь из постановки задачи, к описанию которой мы и перейдем.

Считаем, что перед началом работы имеются два файла с компонентами типа (*arecord*). В каждом из файлов различные компоненты обладают различными значениями признака *rkey*. Один из них — файл *old* — хранит некоторый запас данных (представленных компонентами файла), подлежащий частичной корректировке. Другой — файл *corr* — содержит набор всех изменений, которые должны быть внесены в файл *old*. В результате работы алгоритма (т. е. процедуры, которую нам предстоит написать) должен быть создан файл *new* с компонентами того же типа, в который внесены все эти изменения.

Изменения вносятся по следующим правилам. Если файлы *old* и *corr* содержат компоненты с одним и тем же значением признака *rkey*, то в файл *new* должна быть записана компонента из файла *corr* (исправленная запись). Если в файле *old* есть компонента с некоторым значением признака, а в файле *corr*

компоненты с таким значением признака нет, то компонента из файла *old* переносится в файл *new* без изменения. Если, наоборот, в файле *corr* есть компонента со значением признака, которым не обладает ни одна из компонент файла *old*, то эта компонента добавляется к файлу *new* в качестве нового элемента данных. Иначе говоря, файл *new* должен быть составлен из всех компонент файла *corr* и тех компонент файла *old*, для которых в файле *corr* нет компонент с тем же значением признака *rkey*.

Если компоненты в файлах *old* и *corr* располагаются хаотически, то для каждой компоненты файла *old* потребуется просмотреть весь файл *corr*, чтобы решить — включать или не включать эту компоненту в файл *new*. Гораздо более экономный алгоритм можно построить, если потребовать, чтобы оба файла были упорядочены, т. е. чтобы в любой паре компонент файла компонента с бóльшим порядковым номером имела бы большее значение признака, чем у компоненты с меньшим порядковым номером.

В рамках нашей задачи файлы *old* и *corr* будут использоваться только в режиме чтения, а файл *new* — только в режиме записи. Однако в других частях программы режим работы с этими файлами может быть иным, поэтому в описаниях файлов режим работы указывать не следует. Чтобы иметь возможность сравнивать значения признаков очередных компонент файлов *old* и *corr*, для этих компонент надо выделить место в памяти. Обычно проще всего для этой цели использовать буферы. Однако для файла *new* буфер заводить нежелательно, так как в качестве очередной компоненты этого файла иногда берется компонента файла *old*, а иногда — файла *corr*. Если файл *new* не имеет буфера, то брать для записи очередную компоненту можно прямо оттуда, где она находится. Если же снабдить этот файл буфером, то перед записью потребуется лишняя пересылка данных в этот буфер (т. е. пересылка в пределах оперативной памяти).

По смыслу задачи откорректированный файл *new* со временем станет корректируемым (окажется в роли *old*). Поэтому припишем файлам *old* и *new* тип **file** (*arecord*) **seq**, а файлу *corr* — тип **file** (*arecord*) **seq buf**. Очередную компоненту файла *old* будем считывать в переменную *nextrec* типа (*arecord*).

Остальные пояснения дадим в виде примечаний.

```
proc update (old, new : file (arecord) seq; corr : file (arecord) seq buf);
  var nextrec : (arecord);
begin open old in; open corr in; open new out;
```

{при открытии файлов заданы режимы работы в соответствии с их ролью в процедуре}

```
  read old to nextrec; read corr;
```

{закончена подготовительная часть алгоритма, теперь переменные *nextrec* и *corr*[ ] содержат очередные (пока первые) компоненты файлов *old* и *corr*}

```
  repeat
    if nextrec.rkey < corr[ ].rkey then
```

{очередное исправление относится к одной из последующих компонент файла

old, его текущая компонента должна быть записана в файл *new* и заменена следующей}

begin write *new* **from** *nextrec*; **read** *old* **to** *nextrec* **end**
else
begin write *new* **from** *corr*[ ];

{записано очередное исправление в файл *new*}

if *nextrec* . *rkey* = *corr*[ ] . *rkey* **then read** *old* **to** *nextrec*;

{исправление относилось к текущей компоненте файла *old*, которая больше интереса не представляет}

read *corr* {прочитано следующее исправление}
end
until *eof*(*old*) ∨ *eof*(*corr*);

{выход из основного цикла происходит, когда обнаруживается, что один из файлов *old* или *corr* исчерпан}

if *eof*(*corr*) **then**

{надо переписать в файл *new* оставшиеся компоненты файла *old* (если они есть)}

while ¬ *eof*(*old*) **do**
begin *write new* **from** *nextrec*; **read** *old* **to** *nextrec* **end**
else

{в файл *new* должны быть записаны оставшиеся исправления}

repeat *write new* **from** *corr*[ ]; **read** *corr*
until *eof*(*corr*);

{файлы *old* и *corr* исчерпаны, файл *new* заполнен}

close *old*; **close** *corr*; **close** *new*;
end {конец процедуры *update*}

2.4.4. Примечания в программах

По определению (см. конец разд. 2.1.3) примечание — это такая вставка в текст программы, выбрасывание которой почти никак не влияет на результат работы программы. Таким образом к основной функции языка — служить средством общения между человеком и машиной — примечания не имеют отношения. Но их роль чрезвычайно велика, когда с программой имеет дело человек, будь то автор программы или кто-либо иной, собирающийся воспользоваться ею в своих целях. Программа — это детальный план действий, исполнение которых должно привести к решению некоторой задачи. Составляя этот план, автор должен убедить прежде всего самого себя в том, что в результате исполнения программы задача действительно будет решена. Еще труднее убедить в этом других людей, настроенных более скептически и не доверяющих энтузиазму автора.

Программы часто разрабатываются не одним человеком, а группой авторов. Каждый из них должен понимать ход мыслей члена группы, написавшего ту или

иную часть программы, должен знать, что́ эта часть требует от других частей и на какие свойства данной части можно рассчитывать при составлении этих других частей.

Программа, предназначенная для публикации или открытая в качестве полуфабриката для разработки других программ, должна быть понятна читателю и потенциальному пользователю.

Объяснить принцип работы программы можно в некотором сопроводительном документе, не пересекающемся с ее текстом. Но, указывая в самой программе назначение каждого оператора или хотя бы не слишком больших по размеру композиций операторов, можно существенно дополнить такой документ. Пояснения, относящиеся к одному оператору, носят такой же характер, как пояснения ко всей программе — описывается, в каких условиях начинает исполняться оператор и как результаты его исполнения связаны с этими начальными условиями.

Пояснения, о которых идет речь, могут быть более или менее детальными. Писать подробные примечания никому не хочется, слишком краткие примечания не достигают цели (подобно словам «очевидно, что» в математических публикациях, хотя самому автору публикации это стало очевидным после дней или даже недель напряженных раздумий). Если в программе реализован хорошо известный метод решения задачи, то может оказаться достаточным сослаться на источник, указав на связь используемых там обозначений с обозначениями в программе и на другие параллели между структурой метода и частями программы.

Противоположная ситуация возникает, когда в тексте примечаний не используются никакие достаточно известные понятия, а обозначения в основном не выходят за пределы терминологии, используемой в программе. Определенный круг понятий, терминов и обозначений все же должен быть использован. Это простейшие общематематические понятия и соответствующие им изобразительные средства, например, символы min, max, ‘$\wedge$’, ‘$\exists$’, ‘$\sum$’ и т. п.

Примечания мы по-прежнему будем выделять фигурными скобками ‘{’ и ‘}’, но часто используются и другие ограничители. Важно лишь, чтобы они не встречались в тексте примечаний. Текст примечаний записывается на естественном языке или с привлечением более или менее формальных средств, с широким использованием математической символики. Естественнее всего, как уже было сказано, помещать примечание, относящееся к одному из операторов программы, перед началом или за концом этого оператора, характеризуя соответственно начальные условия для исполнения оператора или результат его исполнения. Жестких ограничений обычно не существует, но попытка поместить примечание в середину идентификатора, строки или записи числа ни к чему хорошему не приведет.

Три простых примера. Запись

$$\{i > 0\} i := i - 1 \{i \geqslant 0\}$$

выражает достаточно понятное утверждение, что после уменьшения на 1 положительного значения целочисленной переменной i ее новое значение окажется неотрицательным.

В следующем примере

if $a < b$ **then** $x := a$ **else** $x := b$ **endif** $\{x = \min(a, b)\}$

утверждение $x = \min(a, b)$, которое можно рассматривать как сокращенную запись логической формулы $x \leqslant a \wedge x \leqslant b \wedge (x = a \vee x = b)$, оказывается истинным после исполнения оператора, каковы бы ни были исходные значения a, b и x.

В книге [26], гл. 7, § 5, в результате некоторого обсуждения задачи о вычислении частной суммы ряда

$s = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$

с точностью (приблизительно) до ε была получена программа, где в примечаниях сконцентрированы итоги и суть обсуждения:

$j := -2; \quad k := 0; \quad u := x; \quad s := x; \quad \{i := 0\}$
$v := -x \times x;$
while $abs(u) > eps$ **do**
begin $\{j = j_i = 8i - 2 = j_{i-1} + 8,$
$k = k_i = 2i(2i+1) = k_{i-1} + j_i,$
$u = u_i = (-1)^i x^{2i+1}/(2i+1)! = u_{i-1}(-x^2)/k_i,$
$s = s_i = \sum_{j=0}^{i} u_i = s_{i-1} + u_i\}$
$j := j + 8; \quad k := k + j; \quad u := u \times v/k; \quad s := s + u \quad \{i := i + 1\}$
end
$\left\{s = \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!}, \quad \left|\frac{x^{2n+1}}{(2n+1)!}\right| \leqslant \varepsilon\right\}$

В примечаниях здесь фигурируют переменные i и n, которые в программе явно не используются. Такие переменные-«призраки», как назвал их Г. С. Цейтин [46], часто очень упрощают запись примечаний и проясняют идею программы. Зависимость «реальных» переменных j, k, u и s от i описана в виде примечания в начале тела цикла. «Призрачные» операторы $i := 0$ и $i := i + 1$ явно не исполняются, но предшествующие каждому из них четыре оператора присваивают реальным переменным значения, соответствующие этим фиктивным присваиваниям.

Оговорка «почти» в первой фразе раздела не случайна. Некоторые примечания специального вида иногда служат для связи программы с внешней средой — точнее, с транслятором во время его работы над текстом программы. Они устанавливают режим трансляции программы, а как следствие, и ее исполнения, желательный для автора программы на данном этапе работы с ней. Типичный пример: с помощью таких примечаний можно включать или выключать контроль попадания вычисленных значений индексов или подобных им величин в заданный диапазон.

Такая погоня за двумя зайцами — довольно грязный стиль построения программного обеспечения. Mais c'est la vie.

2.5. Старые новые веяния

2.5.1. О функциональном стиле программирования

Большинство языков программирования ориентировано на *операторный* (он же *процедурный*) стиль программирования. Для него характерно использование следующих средств: переменных, операторов присваивания для изменения их значений, последовательной, условной и циклической композиций операторов, собственно процедур (не функций).

Функциональный стиль программирования основан, как показывает его название, на использовании только процедур-функций. Роль переменных исполняют параметры функций, присваивание значений осуществляется только при задании аргументов в обращениях к функциям, последовательная композиция операторов заменяется поочередным вычислением аргументов при вызове функции, условная композиция операторов — такой же композицией выражений, циклическая композиция — рекурсией.

Может показаться, что арсенал средств программирования сильно обедняется. Но это далеко не так. Программы, написанные в функциональном стиле, обычно оказываются короче, обозримее и проще для понимания. К тому же они надежнее — «переменная» (новый экземпляр параметра функции) создается в момент присваивания ей значения и не меняет его во все время своего существования. Использование рекурсии при вызове функций хорошо сочетается с рекурсивным стилем описания типов значений и таких конструкций, как выражение, в частности — обращение к функции (или наоборот, ибо выражение — это не что иное, как разветвленная композиция обращений к встроенным в язык функциям — операциям). В разд. 1.7.2–1.7.4 были описаны простейшие средства классического функционального языка Лисп и показаны его богатые возможности даже при скромном арсенале средств.

Конечно, у каждой медали есть оборотная сторона. Если список полей в типе записи даже внешне очень похож на список параметров процедуры, то тип массива имеет скорее последовательный, чем параллельный характер объединения компонент в составное значение. Обработка массивов гармонирует с использованием циклов с параметром, но последние заметно хуже сочетаются с рекурсией, чем циклы с пред- или послепроверкой. Поэтому программы и процедуры вычислительной математики, тесно связанные с линейной алгеброй, вряд ли можно посоветовать писать в функциональном стиле. Но и в традиционных языках программирования, подобных «Паскалю», имеются средства, поддерживающие этот стиль. Они и будут использованы ниже.

Знать о существовании функционального стиля программисту весьма полезно, чтобы не сказать больше. Но автор больших программ с разнообразной и сложной структурой данных не должен считать его панацеей от всех бед.

Пример 1. Процедура дифференцирования элементарных функций, заданных своими аналитическими выражениями, но представленными в виде динамических объектов. Здесь мы проиллюстрируем и разъясним многое из того, что было сказано в разд. 2.2.2, 2.3.3, 2.3.4 и 2.4.1 по поводу работы с обобщен-

ными (объединенными) типами, их подмножествами и вариантами, а также со ссылками.

Функции, называемые элементарными в узком смысле термина, задаются перечислением. Так, одноместные элементарные функции — это '−' (изменение знака), exp (показательная функция), ln (натуральный логарифм), sin, ..., а двухместные функции (операции) — это '+', '−', '×', ... В широком смысле элементарная функция, а точнее, ее аналитическое выражение, — это либо константа (целое или вещественное число или его обозначение, вроде π, а в программах — *pi*), либо переменная (идентификатор, возможно, однобуквенный), либо одна из перечисленных функций (элементарных в узком смысле), примененная к одному или двум выражениям любого из названных подклассов. Вот примеры на все четыре случая:

$3.14 \qquad x \qquad \exp(x)$

$2\,\sin(a \times x + 0.13) - \ln(\cos(x - 3\,y^x))$

где в некоторых случаях знаки '×' подразумеваются.

Можно было бы, разумеется, представлять такие выражения строками. Но должным образом расчленять эти строки на части, представляющие подвыражения, а потом собирать из этих частей и других символов строку, изображающую результат дифференцирования, было бы весьма кропотливым занятием. Гораздо проще, да и нагляднее, один раз для выражения, заданного строкой, построить соответствующий динамический объект, пользуясь аппаратом ссылок, а потом заняться построением по этому объекту другого — для производной.

Создадим шкалу подклассов

type *exprs* = (*const*, *var*, *un*, *bin*)

шкалы имен одноместных (унарных) и двухместных (бинарных) функций (последние представлены наименованиями знаков их операций)

type *unnames* = (*unminus*, *exp*, *ln*, *sin*, ...)
type *binnames* = (*plus*, *minus*, *times*, *div*, ...)

и опишем класс всех выражений для элементарных функций как тип записи:

```
type ident : string [8]; exprptr=ref
  node = record
    case expr : exprs of
      const : (val : real);
      var : (id : ident);
      un : (unname : unnames; opnd : exprptr);
      bin : (binname : binnames; opnd1, opnd2 : exprptr) endrec
```

При этом всем константам, даже целочисленным, приписан наиболее общий тип **real** — это ничему не противоречит, а константы-обозначения, такие как *pi*, следует присоединить к подклассу переменных, не допуская совпадения идентификаторов переменных с этими закрепленными обозначениями.

Нашим четырем примерам элементарных функций различных подклассов соответствуют графические схемы, представленные на рис. 1а и 1б для константы и переменной

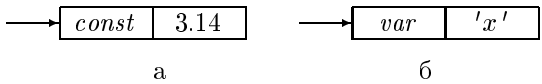


Рис. 1

на рис. 2 для выражения $\exp(x)$

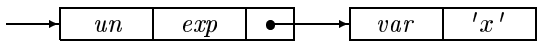


Рис. 2

и на рис. 3 для выражения $2\,\sin(a \times x + 0.13) - \ln(\cos(x - 3\,y^x))$

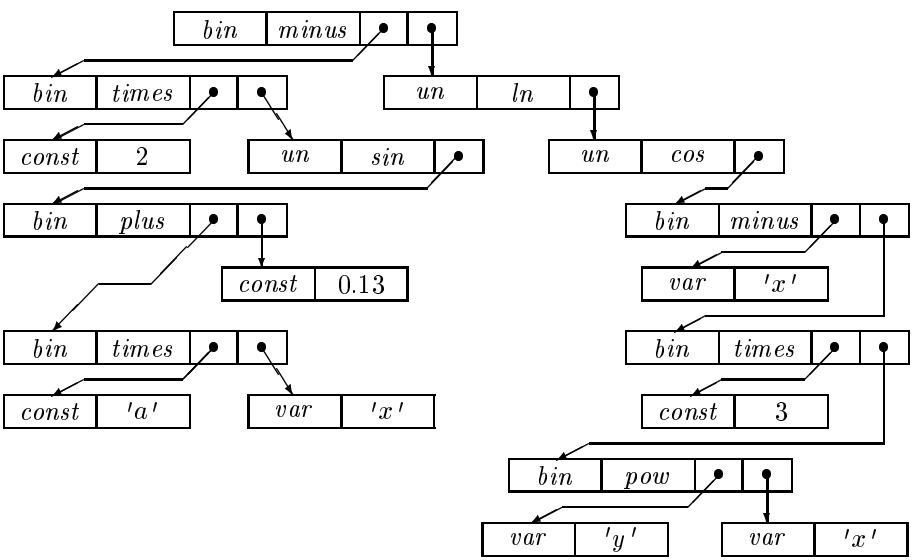


Рис. 3

Поскольку при дифференцировании выражения необходимо дифференцировать его компоненты, которые сами могут быть сложными выражениями, алгоритм дифференцирования естественно описать в виде рекурсивной процедуры-функции. Здесь уже использована основная идея функционального стиля программирования. В остальном будем для начала придерживаться операторного стиля. Параметрами процедуры должны быть ссылка, представляющая дифференцируемое выражение, и строка — наименование переменной, по которой производится дифференцирование. Значение функции — ссылка на выражение для производной. Не будем выписывать тело процедуры целиком, ограничимся лишь

несколькими характерными ветвями. Откажемся от упрощения полученного выражения, оставляя в его составе такие подвыражения, как $0+x$, $x\times 1$, x^0, $x+2\times x$ и т. п. Все необходимые типы были описаны выше.

```
function deriv(y : exprptr; x : ident) : exprptr;
  var p, p1, p2, p3 : exprptr;
begin p := new (node);
  case y ↑. expr of
  const : begin p ↑. expr := const; p ↑. val := 0
    {производная константы равна нулю} end {const};
  var : begin p ↑. expr := const;
    if y ↑. id = x then p ↑. val := 1 else p ↑. val := 0 endif
      {производная переменной по себе самой равна 1,
      по другой переменной — 0} end {var};
  un : begin p1 := y ↑. opnd;
    case y ↑. unname of
    unminus : begin p ↑. expr := un;
      p ↑. unname := unminus; p ↑. opnd := deriv(p1, x)
        {d(−z)/dx = −dz/dx} end {unminus};
    ln : begin p ↑. expr := bin; p ↑. binname := div;
      p ↑. opnd1 := deriv(p1, x); p ↑. opnd2 := p1
        {d(ln z)/dx = (dz/dx)/z} end {ln};
    sin : begin p ↑. expr := bin; p ↑. binname := times;
      p2 := new(node); p2 ↑. expr := un; p2 ↑. unname := cos;
      p2 ↑. opnd := p1; p ↑. opnd1 := p2; p ↑. opnd2 := deriv(p1, x)
        {d(sin z)/dx = cos z × (dz/dx)} end {sin}; ...
    endcase
  end {un};
  bin : begin p1 := y ↑. opnd1; p2 := y ↑. opnd2;
    case y ↑. binname of
    plus, minus : begin p ↑. expr := bin; p ↑. binname := y ↑. binname;
      p ↑. opnd1 := deriv(p1, x); p ↑. opnd2 := deriv(p2, x)
        {d(z ± w)/dx = dz/dx ± dw/dx} end {plus, minus};
    times : begin p ↑. expr := bin; p ↑. binname := plus;
      p3 := new(node); p3 ↑. expr := bin; p3 ↑. binname := times;
      p3 ↑. opnd1 := deriv(p1, x); p3 ↑. opnd2 := p2; p ↑. opnd1 := p3;
      p3 := new(node); p3 ↑. expr := bin; p3 ↑. binname := times;
      p3 ↑. opnd1 := p1; p3 ↑. opnd2 := deriv(p2, x); p ↑. opnd2 := p3
        {d(z w)/dx = dz/dx w + z dw/dx} end {times}; ...
    endcase
  end {bin}
  endcase;
  deriv := p
end {deriv}
```

Для печати в более или менее читаемом виде значений динамических объектов воспользуемся процедурой

```
procedure writeexpr(point : exprptr);
begin case point↑. expr of
  const : write(point↑. val, ' ');
  var : write(point↑. id, ' ');
  un : begin write(point↑. unname, ' ');
       writeexpr(point↑. opnd) end {un};
  bin : begin write(point↑. binname, ' ');
       writeexpr(point↑. opnd1); writeexpr(point↑. opnd2) end {bin}
  endcase
end {writeexpr}
```

где *write* — встроенная процедура печати с произвольным числом аргументов – строк.

Значения распечатываются в одну строчку в так называемой бесскобочной записи Лукасевича — сначала символ функции (знак операции), а затем ее аргументы (операнды). Скобки при этом, действительно, не нужны, так как число аргументов или операндов всегда известно, и привязать их к функции или операции не составляет труда.

Чтобы отпечатать выражение $x\,ln(x) - x$ и его производную по x, надо исполнить следующие операторы:

```
p0 := new(node); p0↑. expr := bin; p0↑. binname := minus;
p1 := new(node); p1↑. expr := bin; p1↑. binname := times;
p2 := new(node); p2↑. expr := varnm; p2↑. id :='x';
p1↑. opnd1 := p2;
p2 := new(node); p2↑. expr := un; p2↑. unname := ln;
p3 := new(node); p3↑. expr := varnm; p3↑. id :='x';
p2↑. opnd := p3; p1↑. opnd2 := p2;
p0↑. opnd1 := p1;
p1 := new(node); p1↑. expr := varnm; p1↑. id :='x';
p0↑. opnd2 := p1;
writeln; write('Expr: '); writeexpr(p0);
writeln; write('Deriv: '); writeexpr(deriv(p0,'x'));
```

На бумаге результат исполнения этих операторов выглядит так:

```
Expr :   minus times x ln x x
Deriv :    minus plus times 1 ln x times x div 1 x 1
```

В более привычном виде запись результата дифференцирования принимает вид $1 \times ln(x) + x \times (1/x) - 1$, что без труда может быть сведено к $ln(x)$.

Попытаемся теперь в максимально возможной степени переписать части нашей программы в функциональном стиле. Опишем четыре функции, создающие динамические объекты используемых нами классов:

```
function makeconst(v : real ) : exprptr;
```

```
  var p : exprptr;
begin p := new(node); p↑.expr := const; p↑.val := v;
  makeconst := p end {makeconst};
function makevar(nm : ident) : exprptr;
  var p : exprptr;
begin p := new(node); p↑.expr := var;
  p↑.id := nm; makevar := p end {makevar};
function makeun(nm : unnames; opd : exprptr) : exprptr;
  var p : exprptr;
begin p := new(node); p↑.expr := un; p↑.unname := nm;
  p↑.opnd := opd; makeun := p end {makeun};
function makebin(nm : binnames; opd1, opd2 : exprptr) : exprptr;
  var p : exprptr;
begin p := new(node); p↑.expr := bin; p↑.binname := nm;
  p↑.opnd1 := opd1; p↑.opnd2 := opd2; makebin := p end {makebin};
```

и саму функцию дифференцирования:

```
function deriv(y : exprptr; x : ident) : exprptr;
  var p, p1, p2 : exprptr; v : real;
begin
  case y↑.expr of
    const : deriv := makeconst(0) {производная константы равна нулю};
    var : begin
      v := if y↑.id = x then 1 else 0 endif;
      deriv := makeconst(v)
        {производная переменной по себе самой равна 1,
        по другой переменной — 0} end {var};
    un : begin p1 := y↑.opnd;
      case y↑.unname of
        unminus : deriv := makeun(unminus, deriv(p1, x))
          {d(−z)/dx = −dz/dx};
        ln : deriv := makebin(bdiv, deriv(p1, x), p1)
          {d(ln z)/dx = (dz/dx)/z};
        sin : deriv := makebin(times, makeun(cos, p1), deriv(p1, x));
          {d(sin z)/dx = cos z × (dz/dx)}; ... endcase
    end {un};
    bin : begin p1 := y↑.opnd1; p2 := y↑.opnd2;
      case y↑.binname of
        plus, minus :
          deriv := makebin(y↑.binname, deriv(p1, x), deriv(p2, x))
          {d(z ± w)/dx = dz/dx ± dw/dx};
        times : deriv := makebin(plus, makebin(times, deriv(p1, x), p2),
                                       makebin(times, p1, deriv(p2, x)))
          {d(zw)/dx = dz/dx w + z dw/dx}; ... endcase
```

```
        end {bin}
    endcase
end {deriv};
```

Наконец, процедура (увы, не функция) печати исходного выражения и результата дифференцирования:

```
procedure writederiv(point : exprptr; varnm : ident);
begin writeln; write('Expr : '); writeexpr(point);
  writeln; write('Deriv :'); writeexpr(deriv(point, varnm))
end {writederiv};
```

Весь этот аппарат приводится в действие вызовом процедуры *writederiv*:

```
writederiv(makebin(minus,
                   makebin(times,
                           makevar('x'),
                           makeun(ln, makevar('x'))),
                   makevar('x')),'x');
```

Краткие комментарии таковы. Функциональный стиль, действительно, приводит к более выразительным программам. Но выдержать его до конца не удается. Последовательность тестов никаким естественным образом не может быть преобразована в один вызов функции. Операторный стиль невозможно вытеснить и на нижнем уровне — из тел функций *makeconst*, ..., *makebin*, а также *writederiv*. Объяснение довольно просто — машинные программы представляют собой последовательности команд, и чем ближе мы к этому уровню, тем труднее преодолеть этот последовательный характер описания действий, да и не всегда стоит этого добиваться.

Упражнение 2.5.1. Дописать недостающие ветви процедуры *deriv*.

Упражнение 2.5.2. Составить процедуру, преобразующую выражение для элементарной функции в виде строки в объект, пригодный в качестве аргумента для обращения к *deriv*.

Упражнение 2.5.3. Составить процедуру упрощения выражения на основе тождеств вида $x \pm 0 = x$, $x \times 1 = x$, $x^0 = 1$ и т. п.

2.5.2. Объекто-ориентированное программирование

Остановимся на основных понятиях того направления в программировании, которое принято называть объекто-ориентированным (некоторые предпочитают говорить «объектно-ориентированное») и обозначать ООП.

Объекты в программировании существовали всегда. Например, в машинных языках программирования — это регистры, сумматор, ячейки памяти, числа, команды и т. д. В языках, о которых мы говорили ранее, под объектами понимались простые и составные значения, в меньшей степени — многие другие из описанных выше понятий. При излагаемом сейчас подходе объекты могут не только быть операндами операций или аргументами процедур, получать имена

при их присваивании переменным, — главное в том, что они обладают присущим им *поведением*, более сложным, чем, например, поведение динамических объектов (см. разд. 2.3.4). Понятие объекта постоянно развивалось и обогащалось новыми качествами примерно в следующем порядке.

Развитые типы данных («записи» или «структуры») появились, наряду с примитивными типами (**word**, **integer**, **real** и т. п.), уже в Коболе (1960), а с середины 60-х гг. — почти во всех универсальных языках («Паскаль», PL/1, Алгол 68 и др.).

Следующие примеры написаны на языке «Паскаль».

Пример 1. В этом примере объект — комплексное число, а из процедур для работы с такими объектами описана только процедура умножения.

```
type Complex= record re,im : real end;
procedure MultCompl(z1,z2 : Complex; var w : Complex);
  begin w.re := z1.re * z2.re - z1.im * z2.im;
    w.im := z1.re * z2.im + z1.im * z2.re end;
var c,c1,c2: Complex;
begin c1.re := 1.0; c1.im := 2.5; c2.re := 1.2; c2.im := 2.0;
  MultCompl(c1,c2,c); Writeln(c.re,c.im)
end.
```

Пример 2. Здесь объектом служит стек. Описан достаточно полный набор процедур и функций, относящихся к таким объектам.

```
const StackSize= 100;
type StackElement = word {или real и т. п.};
  StackPtr=↑Stack;
  Stack=
    record AStack:array [1..StackSize] of StackElement;
         StackTop: word;
    end;
procedure InitStack(var St :Stack);
  begin St. StackTop := 0 end;
procedure CreateStack(var StPtr: StackPtr);
  begin new(StPtr); InitStack(StPtr↑) end;
```

Процедура *InitStack* инициализирует (заполняет единственное в данном случае поле) уже существующий объект, а *CreateStack* создает динамический объект (см. разд. 2.3.4), располагая лишь указателем на него.

```
function StackFull(var St :Stack):boolean;
  begin StackFull:= (St. StackTop = StackSize) end;
function StackEmpty(var St : Stack) : boolean;
  begin StackEmpty:= (St. StackTop = 0) end;
procedure PutOn(x : StackElement; var St : Stack);
  begin with St do
```

```
        begin inc(StackTop); AStack[StackTop] := x end
    end;
  function Top(var St : Stack) : StackElement;
    begin Top:= St.AStack[St.StackTop] end;
  procedure DeleteFrom(var St : Stack);
    begin dec(St.StackTop) end
```

Пример 3. В этом примере описан тот же объект, что и в предыдущем, с теми же действиями над ним, но при представлении в виде списка, а не массива элементов.

```
  const StackSize = 100;
  type StackElement=word;
    ElmPtr=↑StElm; StackPtr=↑Stack;
    StElm=record Elm: StackElement; Prev:ElmPtr end;
    Stack= record First: ElmPtr; CurrSize:word end;
  procedure InitStack(var St : Stack);
    begin St.First := nil; St.CurrSize := 0 end;
  procedure CreateStack(var StPtr: StackPtr);
    begin new(StPtr); InitStack(StPtr↑) end;
  function StackFull(var St : Stack) : boolean;
    begin StackFull:=(St.CurrSize = StackSize) end;
  function StackEmpty(var St : Stack) : boolean;
    begin StackEmpty:=(St.CurrSize = 0) end;
  procedure PutOn(x : StackElement; var St : Stack);
    var SPtr: ElmPtr;
    begin with St do
      begin new(SPtr); SPtr↑.Elm := x;
        SPtr↑.Prev := First; First := SPtr;
          inc(CurrSize) end
    end;
  function Top(var St : Stack) : StackElement;
    begin list: Top:=St.First↑.Elm end;
  procedure DeleteFrom(var St : Stack);
    begin St.First := St.First↑.Prev;
      dec(St.CurrSize) end
```

Стоит обратить внимание на то, что с типом объекта (*Compl* или *Stack*) не просто соседствуют, а органично сочетаются процедуры, характеризующие его возможности и поведение.

Процедуры и функции в качестве элементов данных вошли в обиход несколько позже, примерно начиная с Алгола 68. Вызвано это, по-видимому, тем, что в сознание не только пользователей, но и разработчиков языков с трудом укладывалось равноправие процедур (описаний действий) с элементами обрабатываемых данных. Но когда этот психологический барьер был преодолен,

понятие типа данных стало существенно богаче. Появился даже новый термин *класс* (объектов).

Полиморфизм операций. Один и тот же знак операции может обозначать в зависимости от типа операндов разные действия. Например в «Паскале» знак '+' может обозначать сложение чисел (типа **integer** или **real**), сцепление строк или объединение множеств. Здесь борются друг с другом две тенденции: просто экономить число различных символов и обозначать единым символом только операции, аналогичные по существу (с одним или почти одним и тем же набором аксиом, сказали бы математики). Надо, конечно, тщательно следить за тем, чтобы обозначения операций толковались однозначно — чтобы не возникало сомнений, каковы операнды операции в данном контексте («скупой платит дважды»).

Наследование свойств в простейшем виде представлено в записях с вариантами в языке «Паскаль». Наряду с наследуемыми свойствами — полями записи, общими для всех объектов данного типа, имеются свойства, присущие лишь объектам из некоторого варианта, объектам из подвариантов и т. д.

Модульность (инкапсуляция). Сравнение примеров 2 и 3 показывает, что одна и та же цель (в данном случае — возможность работать со стеками) может быть достигнута разными программными средствами. Для пользователя выбор средств не очень существен. Ему важно, чтобы сама цель была ему понятна, а выбранные средства обеспечивали ее достижение во всей полноте.

Пример 4. Пользователь, зная, что в его распоряжении имеются тип *Stack*, процедуры *InitStack*, *PutOn* и *DeleteFrom* и функции *Top*, *StackFull* и *StackEmpty*, может описать переменные

var k : **word**; St : *Stack*

и составить программу из операторов:

InitStack(St); $k := 0$;
repeat *inc*(k); *PutOn*(k, St) **until** *StackFull*(St);
repeat *Writeln*(k,' ', *Top*(St)); *DeleteFrom*(St); *dec*(k)
until *StackEmpty*(St);

В обоих вариантах реализации названных выше процедур и функций результат получается по существу одним и тем же — именно таким, каким он должен быть.

В условиях примера 2 пользователь, зная механизм реализации, мог бы вместо оператора *PutOn*(k, St) написать

inc(St . *StackTop*); St . *AStack*[St . *StackTop*] := k

и получить тот же результат. Однако в условиях примера 3 в точности такое же изменение программы вступило бы в противоречие с реализацией объектов.

Во втором представлении переполнение стека не грозит большими неприятностями, но константа *StackSize* и функция *StackFull* сохранены ради единообразия, позволяющего сменить это представление на первое, не опасаясь за последствия.

Часто бывает, что механизм реализации некоторого понятия разрабатывается одним программистом, а используется другим — его коллегой или сторонним пользователем. При этом второму вовсе не обязательно, а то и вредно знать, какой механизм выбран, — достаточно владеть самим понятием и видеть заголовки процедур и функций, к которым он может обращаться. В чистом «Паскале» скрыть этот механизм от пользователя невозможно. В Турбо-Паскале это можно сделать, например, следующим образом.

Пример 5. Объединение двух описанных выше в примерах 2 и 3 представлений стека в одном модуле.

```
unit Stacks;
interface {информация для пользователя о наличных средствах}
  const StackSize = 100;
  type repr=(arr,list); StackElement = word;
    ElmPtr=↑StElm; StackPtr =↑Stack;
    StElm=record Elm: StackElement; Prev: ElmPtr end;
    Stack=
      record
        case rr : repr of
        arr: (AStack:array [1..StackSize] of StackElement;
          StackTop: word);
        list: (First: ElmPtr; CurrSize: word)
      end;
  var rr : repr;
  procedure InitStack(var St : Stack);
  procedure CreateStack(var StPtr: StackPtr);
  function StackFull(var St : Stack) : boolean;
  function StackEmpty(var St: Stack): boolean;
  procedure PutOn(x : StackElement; var St : Stack);
  function Top(var St: Stack): StackElement;
  procedure DeleteFrom(var St : Stack);
implementation {скрытая от пользователя реализация типа
    данных}
  procedure InitStack;
    begin case rr of
      arr: St.StackTop := 0;
      list: begin St.First := nil; St.CurrSize := 0 end
    end end;
  procedure CreateStack;
    begin new(StPtr); InitStack(StPtr↑) end;
  function StackFull;
    begin case rr of
      arr: StackFull:=(St.StackTop = StackSize);
```

```
        list: StackFull:=(St . CurrSize = StackSize)
    end end;
  function StackEmpty;
    begin case rr of
      arr: StackEmpty:=(St . StackTop = 0);
      list: StackEmpty:=(St . CurrSize = 0)
    end end;
  procedure PutOn;
    var SPtr: ElmPtr;
    begin case rr of
      arr: with St do
        begin inc(StackTop); AStack[StackTop] := x end;
      list: with St do
        begin new(SPtr); SPtr↑.Elm := x;
          SPtr↑.Prev := First; First := SPtr;
            inc(CurrSize) end
    end end;
  function Top;
    begin case rr of
      arr: Top:=St.AStack[St.StackTop];
      list: Top:=St.First↑. Elm
    end end;
  procedure DeleteFrom;
    begin case rr of
      arr: dec(St.StackTop);
      list: begin St.First := St.First↑. Prev;
        dec(St.CurrSize) end
    end end;
```

Ресурсы — это все то, что обеспечивает поведение объектов в соответствии с их назначением. С некоторой натяжкой можно провести параллели между различными видами ресурсов в обыденной жизни и в программировании:

Сырье	Исходные данные
Энергоснабжение	Подключение к процессору
Транспорт	Каналы передачи данных
Связь	Прерывания и запросы
Площади (складские, производственные, жилые и др.)	Память различных уровней и разновидностей

Объекты конкурируют друг с другом за обладание ресурсами. Полагаться на стихийные рыночные отношения весьма рискованно — нужны средства и

механизмы управления ресурсами. Чтобы управление было эффективным, традиционных прерываний и запросов мало: нужна более развитая система связи — обмена сообщениями. Сообщения могут посылаться как в общую среду — «Всем, всем, всем!» — так и по конкретному адресу. Для объектов каждого класса должны быть определены способы (методы, процедуры) их реакции на сообщения.

Общая среда (имеется в виду среда, создаваемая самой программой во время ее работы, а не внешняя среда, о которой говорилось в разд. 2.1.2 и 2.4.3) сама может состоять из множества взаимодействующих объектов. Осмысленное поведение объектов в среде возможно лишь при условии, что каждый объект постоянно дает знать о своем существовании и состоянии и сам ведет себя, считаясь с существованием и действиями других объектов.

Роль универсального ресурса — платежного средства — играет время во всех его ипостасях («Время — деньги»). Статическое распределение ресурса (его постоянное закрепление за объектом) соответствует передаче имущества в собственность, динамическое — сдаче в аренду или в прокат.

Эти параллели при всей их условности помогают уяснить понятие ресурса и роль ресурсов при функционировании объектов. К сожалению, демонстрация всего сказанного на примере, даже содержательно не очень сложном, заняла бы слишком много места.

Абстрактные типы данных. Самое существенное здесь — аксиоматическое определение типа и всех операций, которые можно выполнять над объектами этого типа. Одно время бытовало мнение, что можно чисто логическими средствами устанавливать свойства отдельных операторов и программ в целом, что эта возможность, если не полностью заменит традиционные методы отладки (никогда не дающие полной гарантии правильности программы), то существенно их дополнит. Предполагалось, что принципиально возможно автоматизировать процесс составления программ по их формальным спецификациям. Для написания спецификаций нужны особые языки, но ничего общепринятого или хотя бы достаточно распространенного здесь пока не возникло. Да и идея «доказательства правильности» программ в значительной мере исчерпала себя, так как выяснилось, что в общем случае невозможно установить свойства результата работы программы или процедуры, не исполнив ее до конца. Поэтому любое свойство программы, зависящее от конкретных исходных данных, неразрешимо стандартными средствами.

Базы данных (реляционные и др.) — это множества объектов, представляющих собой массивы записей (таблицы). Были созданы разнообразные языки для работы с базами данных, учитывающие возможные связи между объектами. Это позволило придать всей работе с данными бóльшую четкость и наглядность. До ООП в его современной трактовке оставалось сделать всего один шаг.

Моделирование. Но и этот шаг фактически был уже сделан в связи с использованием ЭВМ для моделирования процессов, происходящих в разнообразных технических, природных и социальных системах. Интуитивно ясно, что описание этих процессов имеет характер, отличающийся от традиционного представления об алгоритмах. Основная причина этого — недетерминированность

процессов (значительная степень случайности событий), бóльшая или меньшая независимость поведения участников процесса, хотя и с оглядкой на происходящее. Понадобились новые языки программирования и они были созданы. Первым и, пожалуй, наиболее ярким из них был язык Симула ([13], 1966 г.) и его последующие версии.

ООП в понимании автора — это не столько новый стиль написания программ или новый класс языков, сколько новая парадигма (концепция) в программировании, требующая от участников существенной ломки сознания, навыков и собственного поведения. Эта перестройка — пустяк по сравнению с переходом всей страны от распределительной экономики к рыночной. Но выяснилось, что и она трудна, не каждому доступна и, несомненно, для многих нежелательна.

Основные черты этого подхода:

- понятие типа данных заменяется более общим понятием класса объектов,
- объектам свойственно индивидуальное поведение в соответствии с их ролью в системе,
- объекты действуют и взаимодействуют в общей для всех среде,
- взаимодействие объектов становится возможным, если только они способны посылать в эту среду и друг другу сообщения и своевременно и правильно их воспринимать.

Обращаться к процедурам, описанным при определении того или иного класса, можно во многих традиционных языках. Но посылка сообщения — это не вызов процедуры, а воздействие на общую среду, в которой функционируют все объекты. Наивно утверждать, что включение зеленого цвета на светофоре — это вызов процедуры перехода улицы у части пешеходов. Обучение человека правилам дорожного движения (и наказание за их нарушение) — это попытка привить ему желаемый тип поведения. Различие между парадигмами (казарменной процедурной и демократической объектной) проявляется в этом примере достаточно ярко. Фундаментальность этого различия означает, помимо всего прочего, сложность задачи переучивания, переориентации людей, свыкшихся с первой парадигмой, на вторую. Поэтому один-два примера не могут прояснить всех особенностей и достоинств объектной парадигмы.

Уяснив себе эти понятия, можно, конечно, воспользоваться любым привычным языком для составления соответствующей, т. е. объекто-ориентированной программы. Но лучше иметь под руками язык, в который необходимые средства органично встроены и где они сочетаются с более традиционными средствами.

Языки ООП. Лидером среди объекто-ориентированных языков (ООЯ) признается язык C++. Во всяком случае, это наиболее распространенный из таких языков, но на пятки ему наступает Java. Язык C++ возник в результате эволюции достаточно традиционного языка C, который сам появился как следствие потребности иметь в языке высокого уровня средства, ориентированные на машину.

Другой известный язык — Smalltalk (что, между прочим, переводится как светская беседа) в отличие от C++ является «чистым» языком ООП. Главное,

что отличает Smalltalk (как и некоторые другие языки моделирования и спецификаций) от традиционных языков программирования — это отсутствие фиксированной семантики на нижнем уровне или, что точнее, ее простота. Основной проектируемый в том или ином приложении объект характеризуется в терминах его поведения и характера его взаимодействия с объектами прочих классов. С этими объектами проделывается то же самое и т. д. Процесс обрывается на объектах, поведение которых полностью определяется встроенными в них (и закрытыми от пользователя) возможностями. Характер этого поведения внешне проявляется только в посылаемых ими ответных сообщениях.

Но «чистыми» могут быть идея, а не ее воплощение, язык, а не основанная на нем система программирования. Последняя должна включать в себя редактор текстов программ, удобные средства графики, создания баз данных, взаимодействия с базовой операционной системой, включения в программы частей, написанных на других языках и т. д. Все это уже существовало к моменту появления языка Smalltalk, если еще не в C++, то в C, и сразу же начало врастать в системы программирования на базе этого языка и процесс еще идет.

Библиографическая справка. Язык программирования ПЛ/1 описан в [40], Алгол 68 — в [8, 30, 37], «Паскаль» — в [10, 39, 54] и многих других источниках.

За последние 10–15 лет тема ООП стала весьма модной. Обширная информация по ООП содержится в материалах ежегодно проводимых конференций по системам, языкам и приложениям ООП. При довольно тщательном просмотре тома материалов Десятой конференции [55] встретились лишь три упоминания о языке Smalltalk. При этом язык C++ упоминается в томе десятки, если не сотни, раз, имеются довольно частые ссылки еще на дюжину или около того языков и иных средств ООП. Номер журнала Communications of the ACM [49], целиком посвящен опыту и тенденциям в ООП. Многие авторы этого номера довольно благосклонно относятся к языку Smalltalk.

Фундаментальная публикация по языку C — это [20]. По языку C++ опубликованы сотни книг, десятки — в русском переводе. Наиболее подробная русская публикация — это [38]. В [36] описан не только язык, но и система Turbo Vision как иллюстрация идей ООП, отнюдь не просто учебная. Книга [6] — это хорошее введение в языки C и C++, а поиск опечаток в ней — неплохое упражнение на усвоение материала. Однако примеры программ, по-видимому, тщательно проверены на компьютере.

Достаточно популярен у издателей язык «Ява» (Java) — [2] и др.

По языку Smalltalk [52] на русском языке удалось обнаружить одну, и то не очень удачную, публикацию [42].

3. Анализ свойств программ

Наиболее популярными разделами того, что можно назвать теорией программирования, являются:

- структуры данных, их представление в памяти ЭВМ;
- информационный поиск, упорядочивание линейных массивов и файлов;
- формальные языки, грамматики, автоматы и другие абстрактные машины;
- синтаксический анализ программ;
- оценка трудоемкости и теория сложности алгоритмов;
- эквивалентные преобразования алгоритмов, экономия памяти;
- спецификации задач, доказательство свойств программ, автоматический синтез программ;
- семантика языков программирования (теория моделей программ).

В этой главе рассматриваются некоторые из этих вопросов, в том числе и разные способы описания структуры программы, начиная с одного из самых ранних — операторных схем. Много внимания уделено так называемой математической семантике языков программирования. Вновь рассматриваются структуры данных и совершаемых над ними действий, но с более абстрактных позиций, чем в главе 2. Языки, грамматики и абстрактные машины обсуждались в главе 1.

3.1. Операторные схемы

Операторная схема — это одна из разновидностей модели алгоритма (программы). В операторной схеме отражаются многие важные свойства алгоритмов, однако такие понятия, как условные операторы, циклы и, в особенности, подпрограммы, прямо в ней не представлены.

Содержательно оператор — это часть алгоритма, всегда выполняющая одни и те же вычисления над значениями некоторых из переменных программы — *аргументов* оператора. При этом вырабатываются значения некоторых других (частично, возможно, тех же) переменных — *результатов* оператора. В зависимости от значений аргументов после выполнения этих вычислений происходит *переход* от одного оператора (*предшественника*) к другому (*преемнику*).

В теории операторных схем не принимается во внимание существо операций, выполняемых оператором.

Пример 1. Отыскание минимума m функции f для целых значений аргумента от 1 до n.

Рассмотрим алгоритм, записанный в виде фрагмента программы на алголоподобном языке.

$m := f(1)$;
for $i := 2$ **step** 1 **until** n **do**

if $f(i) < m$ **then** $m := f(i)$;
$Write(m)$

или подробнее — на языке почти машинного уровня, операторы которого хорошо отображаются на операторных схемах (справа в фигурных скобках проставлены обозначения операторов):

$m := f(1)$; $i := 2$;	$\{S1\}$
A: **if** $i > n$ **then go to** E;	$\{S2\}$
$u := f(i)$;	$\{S3\}$
if $u \geqslant m$ **then go to** B;	$\{S4\}$
$m := u$;	$\{S5\}$
B: $i := i + 1$; **go to** A;	$\{S6\}$
E: $Write(m)$	$\{S7\}$

На рис. 1 изображена так называемая блок-схема этого алгоритма, к которой добавлены указания, какие переменные использует (кружки с исходящими из них стрелками) и вырабатывает (кружки с заходящими в них стрелками) каждый оператор.

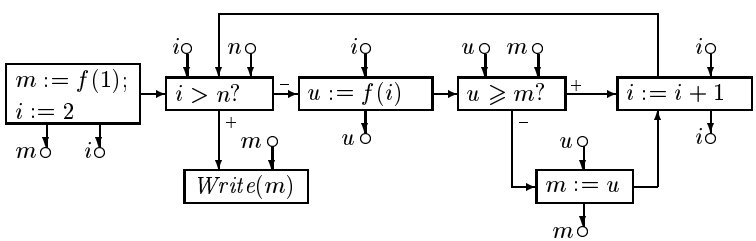


Рис. 1

Если описания действий, содержащиеся внутри прямоугольников, заменить обозначениями операторов и стереть знаки ‘+’ и ‘−’, обозначающие направление перехода в зависимости от результата проверки, то получится операторная схема алгоритма в ее графическом изображении (рис. 2).

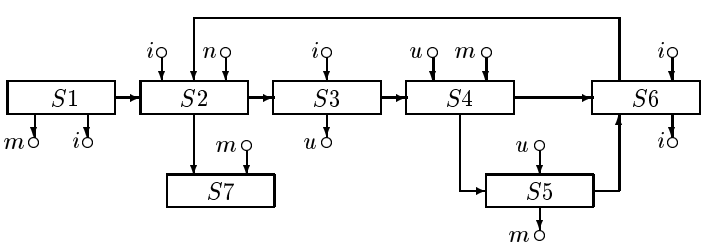


Рис. 2

Все подобные графы никогда не содержат кратных дуг (см. разд. 1.5.4).

Пример 2. Слияние упорядоченных массивов.

$Read(m, n, a, b);$ $\{S0\}$
$i := j := k := 1;$ $\{S1\}$
C: **if** $i > m$ **then go to** B; $\{S2\}$
if $j > n$ **then go to** A; $\{S3\}$
if $a[i] > b[j]$ **then** $\{S4\}$
B: **begin** $c[k] := b[j]; j := j + 1$ **end** $\{S5\}$
else
A: **begin** $c[k] := a[i]; i := i + 1$ **end**; $\{S6\}$
$k := k + 1;$ $\{S7\}$
if $k <= m + n$ **then go to** C; $\{S8\}$
$Write(c)$ $\{S9\}$

Операторная схема этого фрагмента дана на рис. 3.

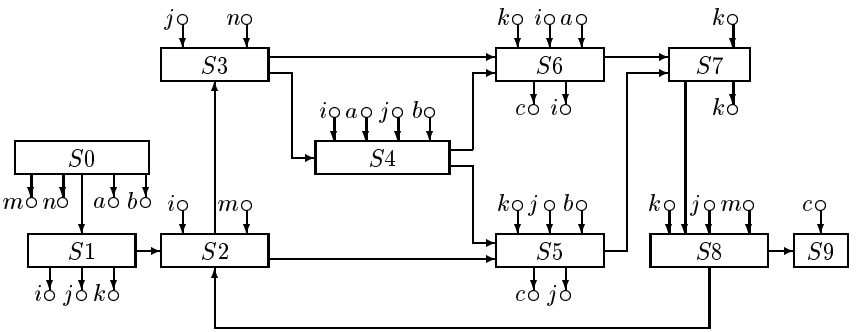


Рис. 3

Обычно мы будем опираться на теоретико-множественное описание, согласно которому операторная схема задается пятеркой $\langle U, V, A, R, T\rangle$, где

$U = \{S_1, \ldots, S_m\}$ — множество операторов S_j,
$V = \{x_1, \ldots, x_n\}$ — множество переменных x_i,
$A \subseteq V \times U$ — отношение «быть аргументом» ($x_i\ A\ S_j$),
$R \subseteq U \times V$ — отношение «иметь результатом» ($S_j\ R\ x_i$),
$T \subseteq U \times U$ — отношение между оператором-предшественником и оператором-преемником ($S_j\ T\ S_k$), т. е. множество возможных переходов.

Схема объединяет в себе ориентированные графы всех трех отношений A, R и T. Граф $\langle U \cup V, A \cup R\rangle$ — двудольный. Тип (U или V) начала и конца каждой дуги определяет, какому из отношений соответствует эта дуга. Граф $\langle U, T\rangle$ будем называть *графом переходов*.

В разных задачах представляют интерес разные стороны понятия операторной схемы. Поэтому данное выше определение будет либо обобщаться, либо уточняться.

Входом операторной схемы назовем оператор S_{in} со свойствами:

$\forall x\,(x \in V \supset x\,\overline{A}\,S_{\text{in}})$ — S_{in} не имеет аргументов,

$\forall S\,(S \in U \supset S\,\overline{T}\,S_{\text{in}})$ — S_{in} не имеет предшественников.

Выход операторной схемы — это оператор S_{out} со свойствами:

$\forall x\,(x \in V \supset S_{\text{out}}\,\overline{R}\,x)$ — S_{out} не имеет результатов,

$\forall S\,(S \in U \supset S_{\text{out}}\,\overline{T}\,S)$ — S_{out} не имеет преемников.

В дальнейшем x всегда обозначает переменную, S — оператор, и можно писать короче: $\forall x\,(x\,\overline{A}\,S_{\text{in}})$, $\forall S\,(S\,\overline{T}\,S_{\text{in}})$ и т. п. Кроме того, граф переходов должен удовлетворять следующему условию связности: любая вершина (т. е. оператор) должна быть достижима из входа — в алгоритме, представленном схемой, не должно быть заведомо неисполняемых операторов, а выход должен быть достижим из любой вершины — в алгоритме не должно быть явных тупиков.

В теории операторных схем рассматриваются следующие основные проблемы: оценка трудоемкости алгоритма; методы исследования свойств алгоритма (завершаемость, связь между аргументами и результатами схемы в целом и т. п.); экономия памяти (преобразования схемы, связанные с переобозначением переменных); эквивалентность операторных схем при преобразованиях, затрагивающих отношение T (меняющих порядок исполнения операторов).

3.1.1. Оценка трудоемкости алгоритмов

В этой задаче не принимаются во внимание множество V и отношения A и R. Каждому оператору S_j сопоставляется трудоемкость (время, временна́я сложность) его исполнения — вещественное число t_j. Задаются также правила, часто́ты или вероятности перехода от любого оператора S_j к произвольному S_k; частота p_{jk} отлична от нуля лишь при условии, что этот переход возможен. Требуется определить время исполнения всего алгоритма или математическое ожидание этого времени.

Марковская схема. Одна из простейших схем оценки трудоемкости алгоритма получается, если рассматривать операторную схему как цепь Маркова (см. разд. 1.6.4), приписывая каждой паре операторов (S_j, S_k) вероятность перехода p_{jk}, причем $p_{jk} = 0$, если $S_j\,\overline{T}\,S_k$.

Вероятности p_{jk} должны удовлетворять условию

$$\sum_{k=1}^{m} p_{jk} = 1 \quad \text{для } j = 1, \ldots, m.$$

Однако если в схеме есть хотя бы один выход S_{out}, то это условие не может быть выполнено, так как для выхода

$$\sum_{k=1}^{m} p_{\text{out},k} = 0.$$

Чтобы обойти это затруднение, иметь возможность промоделировать завершение работы алгоритма, представленного операторной схемой, и остаться в рамках теории цепей Маркова, введем в схему дополнительный (поглощающий) оператор S_{m+1} со следующими свойствами:

$$p_{m+1,k} = 0 \quad \text{для } k = 1, \ldots, m, \qquad p_{m+1,m+1} = 1.$$

Такой оператор естественно называть *оператором останова*, поскольку исполнение программы моделируется в этой схеме блужданием до прихода в поглощающее состояние. Для любого выхода S_{out} операторной схемы следует принять $p_{\text{out},m+1} = 1$, для остальных операторов $p_{j,m+1} = 0$. Теперь условие $\sum_{k=1}^{m+1} p_{jk} = 1$ выполнено для любого j от 1 до $m+1$.

Исполнение программы обычно начинается с оператора входа. Но мы за компанию определим средние значения W_i времени от начала исполнения любого оператора S_i до завершения работы программы.

Уравнение (1.6.4–18):

$$W = (E - Q)^{-1} t. \tag{1}$$

остается в силе (с точностью до разницы на 1 в размерах векторов и матриц и более конкретного обозначения: t вместо V) и дает искомую оценку, поскольку время W_i — это i-й элемент вектора W.

Установим важное свойство оценки трудоемкости, получаемой по марковской схеме. Допустим, что в операторной схеме со входом S_1 и выходом S_m задан путь (последовательность $\langle j_1, j_2, \ldots, j_n, j_{n+1} \rangle$ номеров операторов), соответствующий так называемой *истории исполнения* алгоритма при некоторой совокупности исходных данных. Пусть в этом пути пара соседних номеров (j,k) встречается m_{jk} раз (при этом, разумеется, $m_{jk} > 0$, только если $S_j T S_k$ истинно). Вычислим число повторений M_j оператора S_j по формуле $M_j = \sum_{k=1}^{m+1} m_{jk}$ и в качестве вероятностей p_{jk} примем частоты переходов, вычисляемые по формуле

$$p_{jk} = m_{jk}/M_j \qquad \text{для } j = 1, \ldots, m, \quad k = 1, \ldots, m+1.$$

Предположим, что нам известны только значения p_{jk}, и по формуле (1) мы вычислили значения W_k элементов вектора W. Формула была получена в разд. 1.6.4 из уравнения $W = t + Q\,W$, а из него можно найти тождества, которым должны удовлетворять значения W_k:

$$W_k = \delta_{ik} + \textstyle\sum_{j=1}^{m} p_{jk}\, W_j, \quad k = 1, \ldots, m,$$

где δ_{ik} — символ Кронекера (1 при $k = i$, 0 при $k \neq i$).

Рассмотрим теперь сумму $\sum_{j=1}^{m} m_{jk}$. Она выражает число переходов к оператору S_k от всех других операторов схемы. Это число равно M_k при $k \neq i$ и $M_i - 1$ при $k = i$, так как самое первое исполнение оператора S_i происходит без перехода на него. Отсюда

$$M_k = \delta_{ik} + \textstyle\sum_{j=1}^{m} m_{jk} = \delta_{ik} + \sum_{j=1}^{m} p_{jk}\, M_j.$$

Сравнивая тождества для W_k и M_k, видим, что значения этих величин обязаны совпадать, т. е. по частотам p_{jk} числа M_k восстанавливаются однозначно. В терминологии математической статистики формула (1) дает несмещенную оценку для M_k.

Отсюда следует, что без особого искажения действительности к цепям Маркова могут быть сведены и такие операторные схемы, где переходы имеют вероятности, подчиняющиеся более сложным закономерностям, и даже детерминированные схемы (встречающиеся достаточно часто), где направление перехода

однозначно определяется текущими значениями переменных (например, параметров циклов).

Пример 3.1–1 (продолжение). Изложенный метод здесь неприменим, так как вероятность перехода от оператора $S4$ к оператору $S5$ уменьшается (как правило) с ростом числа повторений цикла. Но трудоемкость этого алгоритма легко оценивается.

Оператор $S2$ исполняется первый раз при $i = 2$, затем — при $i = 3,4, \ldots$ Последний раз он исполняется при $i = n + 1$, и после перехода на оператор $S7$ работа алгоритма завершается. Таким образом, оператор $S2$ исполнится ровно n раз, операторы $S1$ и $S7$ — по одному, а операторы $S3$, $S4$ и $S6$ — по $n - 1$ разу. Число исполнений оператора $S5$ однозначно определить нельзя, если ничего не знать о свойствах последовательности значений $f(1), f(2), \ldots, f(n)$.

Предположим, что эти значения случайны и независимы. (На практике это очень редкий и малоправдоподобный случай.) Заметим, что в операторе $S5$ значение u равно $f(i)$, а m — это наименьшее значение среди $f(1), \ldots, f(i-1)$. Неравенство $u < m$ означает, что среди i чисел $f(1), \ldots, f(i)$ наименьшим является $f(i)$. При нашем предположении вероятность этого равна $1/i$. Следовательно, математическое ожидание числа исполнений оператора $S5$ равно $\sum_{i=2}^{n} 1/i$.

Если t_j — время исполнения оператора S_j, то средняя трудоемкость всего алгоритма равна

$$t_1 + t_7 + nt_2 + (n-1)(t_3 + t_4 + t_6) + t_5 \sum_{i=2}^{n} 1/i.$$

Метод Кирхгофа. Изложим, следуя Д. Кнуту [23], довольно общий метод извлечения всей информации о трудоемкости алгоритма, которая фактически воплощена в графе переходов операторной схемы в сочетании с набором чисел $t_1, \ldots, t_m$. Метод основан на уравнениях и способе их решения, принадлежащих Г. Р. Кирхгофу.

Наша цель — для $1 \leqslant j \leqslant m$ найти число n_j исполнений каждого оператора S_j, с тем чтобы подсчитать общее время t_0 работы алгоритма: $t_0 = \sum_{j=1}^{m} n_j t_j$. Но сначала определяется число переходов по каждой из дуг графа переходов.

Можно не требовать, чтобы в схеме существовали вход и выход в точном смысле данного выше определения. Но необходимо, чтобы были указаны один *начальный* оператор, с которого начинается исполнение алгоритма, и один *заключительный* оператор, на котором оно заканчивается.

Добавим к графу переходов еще одну дугу a_0, исходящую из заключительного оператора и заходящую в начальный, и припишем ей число переходов, равное 1. Дуги исходного графа обозначим $a_1, \ldots, a_s$, а число переходов по дуге a_i — g_i. Тогда для каждого оператора S_j число n_j его исполнений за время работы программы может быть подсчитано суммированием либо чисел переходов на оператор S_j (по дугам вида (S_k, S_j)), либо чисел переходов от S_j (по дугам (S_j, S_k)):

$$\sum_{a_i \longrightarrow S_j} g_i = \sum_{a_i \longleftarrow S_j} g_i \;=\; n_j$$

при $j = 1, \ldots, m$ (закон Кирхгофа), где $a_i \longrightarrow S_j$ означает, что дуга a_i заходит в вершину S_j, а $a_i \longleftarrow S_j$ — что дуга a_i исходит из вершины S_j.

Рассмотрим для иллюстрации операторную схему примера 3.1–2. Ее граф переходов изображен на рис. 1 (пока не следует обращать внимание на то, что некоторые стрелки толще других). Штриховой линией изображена упомянутая выше дуга от заключительного оператора к начальному.

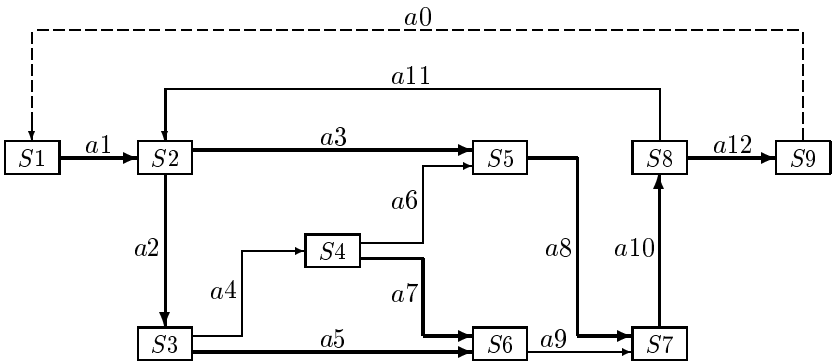


Рис. 1

Для этого графа $m = 9, s = 12$, и, используя в лучших программистских традициях вместо переменных с индексами идентификаторы $g0 - g12$, можно выписать следующие девять уравнений Кирхгофа:

$$\begin{array}{rrr} g0 = g1, & g4 = g6 + g7, & g8 + g9 = g10, \\ g1 + g11 = g2 + g3, & g3 + g6 = g8, & g10 = g11 + g12, \\ g2 = g4 + g5, & g5 + g7 = g9, & g12 = g0, \end{array}$$

а также делающее эту систему неоднородной уравнение

$$g0 = 1.$$

В общем случае получается $m+1$ уравнение для определения $s+1$ величины $g_0, g_1, \ldots, g_s$. Легко заметить также, что уравнения Кирхгофа не независимы. Так как каждая дуга исходит ровно из одного и заходит ровно в один оператор, то, складывая уравнения Кирхгофа, получим тождество, содержащее сумму $\sum_{i=0}^{s} g_i$ как в левой, так и в правой части. Таким образом, среди $s+1$ уравнения есть не более m независимых, и общее решение системы должно зависеть не менее чем от $s + 1 - m$ параметров. Мы убедимся, что при условии связности, наложенном в разд. 3.1 на граф переходов, число независимых уравнений действительно равно m.

Разумеется, можно найти эти уравнения и решить систему, пользуясь обычными для линейных алгебраических уравнений методами. Но метод Кирхгофа позволяет получить общее решение этой системы гораздо экономнее — лишь из теоретико-графовых соображений.

Выбросим из графа переходов дугу a_0 и еще $s - m + 1$ дугу так, чтобы оставшиеся $m-1$ дуга вместе со всеми m вершинами составляли каркас исходного графа.

В нашем примере (рис. 1) один из возможных вариантов такого каркаса показан утолщенными стрелками.

Добавив к каркасу любую из выброшенных дуг — хорд каркаса, получим (см. разд. 1.5.4) граф, содержащий ровно один цикл. Выпишем эти циклы для каждой из хорд. Если при обходе цикла направление обхода совпадает с направлением дуги, то предваряем ее знаком '+', иначе — знаком '−'. Направление обхода выбираем так, чтобы хорда, определяющая цикл, получила знак '+'.

Для нашего примера получим следующие циклы:

$a0 + a1 + a3 + a8 + a10 + a12$	$g0$
$a4 + a7 - a5$	$g4$
$a6 - a3 + a2 + a5 - a7$	$g6$
$a9 - a8 - a3 + a2 + a5$	$g9$
$a11 + a3 + a8 + a10$	$g11$

Выписанные циклы образуют систему циклов, называемых *базисными*. Эта система зависит от выбора каркаса и поэтому определяется неоднозначно. Пригоден любой выбор. О предпочтительном выборе будет сказано ниже.

Каждый базисный цикл содержит, по построению, только одну из ранее выброшенных дуг — хорду a_i. Выпишем против каждого базисного цикла соответствующую переменную g_i, как это и было сделано выше. Назовем и эти переменные базисными.

Придадим какой-нибудь из базисных переменных g_i значение 1, остальным — значение 0, выделив тем самым один из базисных циклов. Каждой переменной g_k, соответствующей дуге a_k, вошедшей в этот цикл, придадим значение $+1$ или -1, выбирая знак так, чтобы он совпадал со знаком, предваряющим дугу a_k в данном цикле. Остальным переменным g_j придадим значение 0. Ясно, что такое присваивание значений соответствует однократному обходу выделенного базисного цикла в выбранном направлении. Ясно также, что уравнения Кирхгофа при этом удовлетворяются. Такое решение уравнений Кирхгофа назовем базисным.

Базисные решения независимы, так как все они, кроме одного, присваивают значение 0 одной из базисных переменных g_i и, следовательно, это одно не может быть линейной комбинацией остальных. Любая линейная комбинация базисных решений с произвольными значениями базисных переменных g_i для $i > 0$, также будет решением системы уравнений Кирхгофа. Значение $g_0 = 1$ должно быть фиксировано.

Значения базисных переменных g_i однозначно определяют решение уравнений Кирхгофа. Действительно, если бы имелись два различных решения $g_0, \ldots, g_s$ и $g_0{}', \ldots, g_s{}'$ с одинаковыми значениями этих переменных, то разность этих решений также была бы решением, в котором были бы отличны от нуля только значения переменных, соответствующих некоторым дугам каркаса, что невозможно (если выбросить из каркаса дуги, которым сопоставлены значения $g_k = 0$, и рассмотреть один из листьев оставшегося леса, то убедимся, что в нем уравнение Кирхгофа не удовлетворяется). Итак, решение системы (включая уравнение $g_0 = 1$) зависит точно от $s - m + 1$ параметра.

Легко явно выразить значения g_k для дуг каркаса через значения базисных переменных g_i. В выражение для g_k каждое g_i входит с таким же коэффициентом ($+1$, -1 или 0), с каким дуга a_k входит в базисный цикл, содержащий дугу a_i.

Для нашего примера:

$$g1 = g0, \qquad g7 = g4 - g6,$$
$$g2 = g6 + g9, \qquad g8 = g0 + g11 - g9,$$
$$g3 = g0 - g6 + g11 - g9, \qquad g10 = g0 + g11,$$
$$g5 = -g4 + g6 + g9, \qquad g12 = g0,$$

причем $g0 = 1$, а $g4$, $g6$, $g9$ и $g11$ могут быть заданы произвольно.

Однако не всякое решение системы уравнений Кирхгофа с дополнительным условием $g0 = 1$ соответствует возможной истории исполнения алгоритма. Действительно, во-первых, все g_i должны быть целыми неотрицательными числами. Например, решение с $g4 = 1$, $g6 = 2$ не годится, так как дает $g7 = -1$.

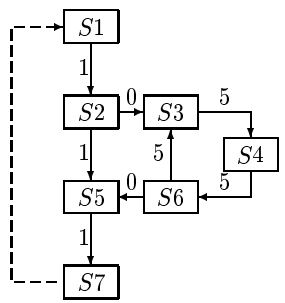


Рис. 2

Во-вторых, граф, в котором сохранены лишь дуги со строго положительными значениями g_i, должен быть связен, так что решение, изображенное на рисунке (не для нашего примера), хотя и удовлетворяет уравнениям Кирхгофа и условию неотрицательности, но также не годится.

Если же оба эти условия соблюдены, то решение определяет путь по схеме от начального оператора к заключительному, который может реализоваться при исполнении алгоритма (однако учет значений, принимаемых переменными, мог бы воспрепятствовать реализации такого пути).

Хотя общее решение уравнений Кирхгофа может быть получено по любому каркасу, два типа каркасов: выходящий и входящий ориентированные каркасы (по поводу их построения см. разд. 2.4.2) приводят к решениям, обычно более благоприятным для оценки трудоемкости. Листья входящего каркаса расположены в местах разветвлений при движении по схеме от ее входа. Это удобно, так как именно в этих местах легче всего бывает сказать, как часто происходит переход по той или иной дуге графа переходов.

Можно использовать также следующий критерий построения каркаса. Из графа переходов выбрасываются те дуги, переход по которым связан с какой-нибудь содержательной характеристикой алгоритма, так что число переходов отражает значение такой характеристики. Но этот критерий уже не укладывается в понятие операторной схемы.

3.1.2. Доказательство свойств программ

Детально рассмотреть эту проблему, оставаясь в рамках теории операторных схем, невозможно. Подробнее это будет сделано ниже, в разд. 3.2.2, а сейчас обсудим схему доказательства свойств программ только в самых общих чертах.

Для этого потребуется лишь граф переходов по операторной схеме, но с размеченными дугами.

Свойство программы — это некоторое утверждение о состоянии переменных программы, т. е. о совокупности значений переменных из множества V. Но в определении операторной схемы участвуют лишь имена переменных, а не их значения. Поэтому исследование содержательной стороны проблемы не может основываться только на этом общем определении. Будем говорить просто о *состояниях* (не уточняя — чего) и о множестве (пространстве) W состояний.

Утверждение о состоянии — это предикат, определенный на множестве состояний, т. е. отображение множества W в множество $\mathrm{BV} = \{true, false\}$ логических значений.

Состояния будем обозначать буквами $w, \ldots$, предикаты — буквами $p, q, \ldots$ (возможно, со штрихами и индексами). Таким образом, если w — это состояние ($w \in \mathrm{W}$), то $p(w)$ — логическое значение ($p(w) \in \mathrm{BV}$).

Операторы преобразуют состояния (неведомым способом). Кроме того, состояние, в котором исполняется оператор, определяет дугу графа T, по которой происходит переход к следующему исполняемому оператору.

Таким образом, о состояниях и их свойствах, т. е. о предикатах, которым удовлетворяют состояния, целесообразно говорить не во время исполнения операторов, а в промежутках между исполнением оператора-предшественника и оператора-преемника. Поэтому предикаты естественно сопоставлять дугам графа переходов — размечать их этими предикатами. Пусть дуге a_i сопоставлен предикат p_i. Совокупность всех предикатов p_i назовем *предикатной разметкой* дуг схемы.

Обычно важнее всего установить связь между исходным и заключительным состояниями программы. Потребуем, чтобы из единственного входа S_{in} операторной схемы исходила лишь одна дуга a_{in} к некоторому другому оператору, с исполнения которого и начинается преобразование состояний (оператор S_{in} может быть оператором ввода исходных данных, определяющим тем самым начальное состояние). Этой дуге сопоставляется предикат p_{in}, называемый начальным предикатом. Интерес представляют только такие состояния, вырабатываемые оператором S_{in}, которые удовлетворяют начальному предикату p_{in}. Будем их называть *допустимыми начальными состояниями*.

Аналогично, потребуем, чтобы в выход S_{out} операторной схемы заходила единственная дуга, которой сопоставляется заключительный предикат p_{out}.

Пара предикатов $(p_{\mathrm{in}}, p_{\mathrm{out}})$ называется *спецификацией* программы, а доказательство того, что исполнение программы, начавшееся в допустимом начальном состоянии, завершается в состоянии, удовлетворяющем предикату p_{out}, — доказательством правильности программы (по отношению к данной спецификации).

Излагаемый ниже подход к доказательству свойств программ предполагает возможность доказывать некоторые утверждения о свойствах каждого оператора схемы в отдельности, исходя из предикатной разметки.

Обозначим через $p(S)$ дизъюнкцию предикатов, сопоставленных всем дугам, заходящим в оператор S. Если некоторый оператор S_j начинает исполняться в состоянии w, удовлетворяющем дизъюнкции $p(S_j)$, в результате его исполнения вырабатывается состояние w_1 и происходит переход по дуге a_i, исходящей из S_j, причем состояние w_1 удовлетворяет предикату p_i, сопоставленному дуге a_i, то будем говорить, что данное исполнение оператора S корректно (по отношению к выбранной разметке).

Теорема 3.1.1. *Пусть про каждый оператор S_j операторной схемы (кроме ее входа и выхода) известно, что каждое его исполнение корректно по отношению к выбранной разметке. Пусть $a'_0\,(=a_{\text{in}})$, a'_1, ... — последовательность (конечная или бесконечная) дуг, соответствующая некоторому исполнению алгоритма, представленного схемой, причем состояние w_0 после исполнения оператора S_{in} допустимо, т. е. удовлетворяет предикату p_{in}, сопоставленному дуге a_{in}. Тогда состояние, соответствующее любой дуге a'_i этой последовательности, удовлетворяет предикату p'_i, сопоставленному этой дуге.*

В частности, если эта последовательность конечна и завершается дугой $a'_n = a_{\text{out}}$, заходящей в оператор S_{out}, то состояние в момент перехода по этой дуге удовлетворяет предикату p_{out}.

Доказательство (индукция по номеру i дуги a'_i). Для дуги a'_0 утверждение теоремы выполнено по условию. Пусть дуга a'_{i-1} заходит в оператор S_j и текущее состояние w удовлетворяет предикату p'_{i-1}. Тогда оно удовлетворяет и дизъюнкции $p(S_j)$. Дуга a'_i — это одна из дуг, исходящих из S_j, причем по определению последовательности $a'_0, a'_1, \ldots, a'_{i-1}, a'_i, \ldots$ после исполнения оператора S_j в состоянии w произошел переход именно по дуге a'_i. В силу корректности исполнения оператора S_j, состояние w_1, выработанное этим оператором, должно удовлетворять предикату p'_i, сопоставленному дуге a'_i. По индукции это верно для любого i. Если последовательность $a'_0, a'_1, \ldots$ конечна, то это верно и для ее последней дуги a'_n, которая по условию теоремы заходит в оператор S_{out} и поэтому может быть только дугой a_{out}. Следовательно, состояние при переходе по этой дуге удовлетворяет предикату p_{out}. $\triangleleft$

Замечания. 1. Теорема достаточно тривиальна. Ее роль лишь в том, чтобы служить обоснованием введенных здесь понятий и требований к разметке схемы и к реализации алгоритма, представленного схемой.

2. Ни в формулировке, ни в доказательстве теоремы не требовалось, чтобы алгоритм был детерминированным, т. е. чтобы состояние w, в котором начинает исполняться оператор, однозначно определяло состояние w_1 после исполнения оператора и дугу a, по которой происходит переход. Важно лишь, чтобы любой возможный исход удовлетворял требованию корректности исполнения оператора по отношению к принятой разметке схемы. Если алгоритм недетерминирован, то последовательность $a'_0, a'_1, \ldots$ не определяется однозначно начальным состоянием, но утверждение теоремы остается верным для любой возможной последовательности.

Для иллюстрации вернемся к алгоритму слияния двух упорядоченных массивов (пример 3.1–2 и рис. 3.1–1). Необходимые предикаты имеют вид:

$p1$: $m \geqslant 0 \wedge n \geqslant 0 \wedge m + n \geqslant 1$
$\wedge \forall_{i1=1}^{m-1} a_{i1} \leqslant a_{i1+1} \wedge \forall_{j1=1}^{n-1} b_{j1} \leqslant b_{j1+1}$
$\wedge i = 1 \wedge j = 1 \wedge k = 1$

(массивы a и b могут быть пустыми, но хотя бы один из них не пуст, эти массивы упорядочены, заданы начальные значения переменных i, j и k);

$p11$: $\forall_{i1=i}^{m-1} a_{i1} \leqslant a_{i1+1} \wedge \forall_{j1=j}^{n-1} b_{j1} \leqslant b_{j1+1} \wedge \forall_{k1=1}^{k-2} c_{k1} \leqslant c_{k1+1}$
$\wedge 1 \leqslant i \leqslant m+1 \wedge 1 \leqslant j \leqslant n+1 \wedge k = i + j - 1$
$\wedge (i \leqslant m \supset c_{k-1} \leqslant a_i) \wedge (j \leqslant n \supset c_{k-1} \leqslant b_j)$
$\wedge k \leqslant m + n$

(остатки массивов a и b и накопленная часть массива c упорядочены, даны ограничения на текущие значения переменных, последний элемент массива c не превышает очередных элементов массивов a и b, массив c не заполнен);

$p2$: $p11 \wedge i \leqslant m$,
$p3$: $p11 \wedge i = m + 1$,
$p4$: $p11 \wedge i \leqslant m \wedge j \leqslant n$,
$p5$: $p11 \wedge i \leqslant m \wedge j = n + 1$

(предикаты дополнительно ограничивают текущие значения переменных i и j);

$p6$: $p4 \wedge a_i > b_j$,
$p7$: $p4 \wedge a_i \leqslant b_j$

(возможные соотношения между очередными элементами массивов a и b);

$p8$: $\forall_{i1=i}^{m-1} a_{i1} \leqslant a_{i1+1} \wedge \forall_{j1=j}^{n-1} b_{j1} \leqslant b_{j1+1} \wedge \forall_{k1=1}^{k-1} c_{k1} \leqslant c_{k1+1}$
$\wedge 1 \leqslant i \leqslant m+1 \wedge 1 \leqslant j \leqslant n+1 \wedge k = i + j - 2$
$\wedge (i \leqslant m \supset c_k \leqslant a_i) \wedge (j \leqslant n \supset c_k \leqslant b_j) \wedge k \leqslant m + n$

(предикат выражает то же самое, что $p11$, но при иной связи между текущими значениями переменной k и переменных i и j);

$p9$: идентичен $p8$;
$p10$: $\forall_{i1=i}^{m-1} a_{i1} \leqslant a_{i1+1} \wedge \forall_{j1=j}^{n-1} b_{j1} \leqslant b_{j1+1} \wedge \forall_{k1=1}^{k-2} c_{k1} \leqslant c_{k1+1}$
$\wedge 1 \leqslant i \leqslant m+1 \wedge 1 \leqslant j \leqslant n+1 \wedge k = i + j - 1$
$\wedge (i \leqslant m \supset c_{k-1} \leqslant a_i) \wedge (j \leqslant n \supset c_{k-1} \leqslant b_j)$
$\wedge k \leqslant m + n + 1$

(предикат выражает то же самое, что $p11$, но при иной связи между текущими значениями переменной k и переменных m и n);

$p12$: $p10 \wedge k > m + n$
или
$\forall_{k1=1}^{m+n-1} c_{k1} \leqslant c_{k1+1}$
$\wedge k = m + n + 1 \wedge i = m + 1 \wedge j = n + 1$

(массив c весь упорядочен и заполнен, массивы a и b исчерпаны).

Здесь предикат $p1$ описывает требования, предъявляемые к исходным данным (действуя строго, следовало бы добавить к алгоритму начальный оператор $S0$: $Read(m, n, a, b)$ и сопоставить дуге, ведущей от него к оператору $S1$, «чистый» начальный предикат, отличающийся от $p1$ отсутствием трех последних членов).

Угадать вид предиката $p11$ не слишком просто (легко кое-что не предусмотреть, а в чем-то и ошибиться, но тогда не все условия теоремы 3.1.1 окажутся выполненными или не все существенные свойства алгоритма — доказанными).

Дуга $a11$ зацикливает схему, поэтому предикаты, сопоставляемые подобным дугам, принято называть *инвариантами циклов*.

В правильном подборе инвариантов циклов состоит суть доказательства свойств программ, а поскольку явно или неявно этот инвариант должен иметься в виду при написании программы, то в этом же заключается само искусство программирования. Некоторые программисты могут и не согласиться с этим тезисом, имея на то резон — не всякий набор сведений может быть воплощен в форму общего (квантифицированного) утверждения, каким является инвариант.

После того как предикат $p11$ подобран, нетрудно убедиться, что $p1 \supset p11$ (так что $(p1 \vee p11) \equiv p11$), из дизъюнкции $p3 \vee p6$ логически следует $p8$ (с учетом новых значений j и c_k, вырабатываемых оператором $S5$), из $p5 \vee p7$ аналогичным образом следует $p9$, а из $p8 \vee p9$ — $p10$. Корректность исполнения операторов ветвления $S2$, $S3$, $S4$ и $S8$ еще более очевидна.

Приведенный пример показывает, что полная запись доказательства свойств программы может оказаться более громоздкой, чем сама программа. Это одна из причин непопулярности идеи доказательства правильности программ у программистов-практиков, больше полагающихся на свою интуицию и на отладку.

С громоздкостью можно частично бороться, вводя специальные предикаты, например $Ord(a,i,m)$, означающий, что часть массива a от a_i до a_m упорядочена (Ord — сокращение от $Ordered$), доказывая заранее их свойства, как-то

$$Ord(a,i,m) \supset Ord(a,i+1,m),$$
$$Ord(c,1,k) \wedge c_{k+1} \geqslant c_k \supset Ord(c,1,k+1),$$

и используя эти свойства при доказательстве свойств (корректности исполнения) операторов.

При составлении спецификации программы надо быть достаточно внимательным и не упустить существенных свойств. Так, в рассмотренном примере не доказано, что массив c получен объединением массивов a и b. Если все элементы массивов a и b различны в совокупности, то в заключительный предикат следовало бы включить условие $\forall_{k=1}^{m+n}(\exists_{i=1}^{m} c_k = a_i \vee \exists_{j=1}^{n} c_k = b_j)$ и доказать, что оно выполняется. Если же элементы в массивах a и b могут повторяться или быть общими, то условие должно быть еще более громоздким.

На деле ситуация может оказаться еще сложнее. Не исключено, что предикат p_i должен быть связан не только с фактом перехода по дуге a_i в данный момент исполнения программы, но и с обстоятельствами, предшествующими этому переходу. В самом общем, худшем с точки зрения объема хранимых данных, случае утверждение о текущем состоянии может зависеть от всей истории исполнения — описания последовательности всех предшествующих переходов и всех промежуточных состояний. Это означает, что доказательство свойств такой программы не может быть статическим и его следует вести динамически — параллельно с ее исполнением при конкретных исходных данных.

3.1.3. Завершаемость алгоритмов

Ощутимым недостатком теоремы 3.1.1 является то, что она устанавливает связь между начальным и заключительным состояниями лишь при условии, что работа алгоритма завершается (за конечное число шагов — после исполнения конечного числа операторов, после конечного числа переходов). То, что она действительно завершается, приходится доказывать отдельно.

Частично здесь могут помочь оценки времени исполнения алгоритма. Так, если матрица $E - Q$ в методе цепей Маркова не особая, то $(E - Q)^{-1}$ существует и среднее время исполнения алгоритма конечно. Иначе говоря, вероятность зацикливания алгоритма равна нулю. Метод Кирхгофа также позволяет вычислить или оценить сверху время работы алгоритма и тем самым доказать, что оно конечно.

Однако интерес представляют более прямые методы доказательства завершаемости алгоритмов. Ниже излагается один из таких методов, основанный на использовании функций, аналогичных функциям, введенным А. М. Ляпуновым в его теории устойчивости движения (знаменитая «вторая метóда»).

Теорема 3.1.2. *Пусть для операторной схемы задана предикатная разметка дуг и найдена функция состояния F с целочисленными значениями, такая, что*: 1) *для каждого оператора S и для каждого состояния w, удовлетворяющего предикату $p(S)$, верно, что $F(w_1) < F(w)$, где w_1 — состояние, вырабатываемое оператором S в состоянии w*; 2) *для каждой дуги a'_i верно, что $p'_i \supset F(w) \geqslant 0$, где w — текущее состояние при прохождении дуги a'_i*; 3) *соблюдаются условия теоремы* 3.1.1. *Тогда алгоритм, начинающий работу из допустимого состояния w_0, завершается за конечное число шагов.*

Доказательство. Последовательность дуг $a'_0, a'_1, \ldots$, соответствующая некоторому исполнению алгоритма, не может быть бесконечной, если начальное состояние допустимо. В самом деле, по теореме 3.1.1 переход по каждой дуге a'_i этой последовательности происходит при состоянии w_i, удовлетворяющем предикату p'_i. По свойству 1) $F(w_0) > F(w_1) > \ldots$, откуда $F(w_n) \leqslant F(w_0) - n$. Если $n > F(w_0)$, то $F(w_n) < 0$, что противоречит свойству 2). $\lhd$

Замечания. 1. Содержательно функция F и ей подобные характеризуют продвижение к цели, ради которой написана программа. Соображения на этот счет, вероятно, есть у автора любой программы. Однако никакого общего метода построения подобных функций не существует, так как задача распознавания завершаемости алгоритма неразрешима стандартными средствами (разд. 1.7.5).

2. Если найдена подходящая функция F, то $F(w_0)$ дает верхнюю оценку длины пути от входа до выхода, что позволяет получить верхнюю оценку и для максимальной трудоемкости алгоритма, например $t_0 \leqslant F(w_0) \max_{j=1}^{m} t_j$ (обозначения те же, что в разд. 3.1.1).

3. Функцию F построить трудно еще и потому, что требование 1) слишком жестко. Однако некоторые «напрашивающиеся» обобщения теоремы неверны.

Верное обобщение дадим, используя некоторые понятия разд. 1.5.4.

Предварительно заметим, что каждой бикомпоненте графа переходов операторной схемы соответствует часть алгоритма, исполняемая не более одного раза при любом исполнении данного алгоритма. Примеры — прямой и обратный ход в некоторых алгоритмах линейной алгебры, проходы (этапы) компиляторов. Завершаемость исходного алгоритма вытекает из завершаемости каждой из этих частей, что позволяет доказывать завершаемость поэтапно.

Теорема 3.1.3. *Пусть для операторной схемы задана предикатная разметка дуг и найдена функция состояния F с целочисленными значениями, такая, что*: 1) *для каждого оператора S и для каждого состояния w, удовлетворяющего предикату $p(S)$, верно, что $F(w_1) \leqslant F(w)$, где w_1 — состояние, вырабатываемое оператором S в состоянии w*; 2) *для каждого простого контура графа переходов схемы можно указать такой оператор S, через который проходит этот контур, что $F(w_1) < F(w)$*; 3) *для каждой дуги a_i верно, что $p_i \supset F(w) \geqslant 0$, где w — состояние, в котором проходится эта дуга*; 4) *соблюдаются условия теоремы* 3.1.1. *Тогда алгоритм, начинающий работу в допустимом состоянии w_0, завершается за конечное число шагов.*

Доказательство. Пусть по ходу исполнения алгоритма, представленного данной операторной схемой, из допустимого начального состояния операторы встречаются в последовательности $S'_0\ S'_1\ \ldots\ S'_n$, возможно с повторениями. Этот путь p по графу переходов может быть составлен из простого пути с теми же началом S'_0 и концом S'_n и s простых контуров графа (при подсчете s учитывается кратность вхождения каждого контура). Длина простого пути не может быть больше числа m операторов в схеме. Пусть l' — максимальная длина простых контуров. Для длины n пути p получаем оценку: $n \leqslant m + l's$.

При исполнении алгоритма каждый оператор, принадлежащий любому простому контуру, вошедшему в состав пути p, исполнится столько раз, сколько этот контур вошел в представление. Уменьшение значения функции F не может быть меньше s. С другой стороны, оно не может быть больше $F(w_0)$, где w_0 — состояние, выработанное оператором S'_0, так что $s \leqslant F(w_0)$. Отсюда следует оценка для возможной длины пути: $n \leqslant m + l'F(w_0)$. $\lhd$

3.1.4. Структурированные схемы

Структурированность программ — стиль их написания. Перефразируя Бюффона, можно сказать, что программист — это стиль написанных им программ. Стиль программирования, как и его элементы, — понятие неформальное. Разные люди имеют разное мнение по поводу того, что считать хорошим стилем. Тем не менее уже в 60-е годы широко распространилось убеждение, что один из важнейших признаков хорошего стиля — это структурированность программы. Опишем это понятие в терминах операторных схем.

Фрагменты структурированных схем программ. Структурированная схема строится из фрагментов, каждый из которых имеет одну входную и одну выходную стрелку. Простейший фрагмент — пустой — состоит из одной стрелки,

входной и выходной одновременно (рис. 1 а0), за ним идет фрагмент из одного оператора (рис. 1 а).

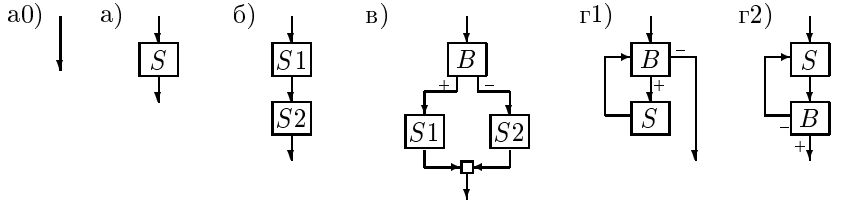


Рис. 1

Фрагменты остальных видов (структурированные) получаются композицией двух или трех операторов. Фрагменты любого вида обычно могут заменять внутренние фрагменты этих фрагментов.

При последовательной композиции (рис. 1 б) выходная стрелка одного из двух сочленяемых операторов совпадает со входной стрелкой другого оператора.

Оставшиеся виды композиции предусматривают наличие в составе фрагмента псевдооператора B, проверяющего выполнение некоторого условия и не осуществляющего иных действий, а поэтому не меняющего состояние программы (значений ее переменных). Такой оператор имеет две выходные стрелки. Переход по одной из них происходит, если условие удовлетворяется, по другой — если нет.

При альтернативной композиции (рис. 1 в) одна из стрелок ведет к внутреннему оператору $S1$, другая — к оператору $S2$. Выходные стрелки этих операторов обязательно сливаются в одну и не имеют права вести к разным фрагментам схемы. Слияние стрелок не предусмотрено в определении операторной схемы. Можно считать, что схема неявно содержит еще один оператор, не производящий никаких действий. На рис. 1в он изображен в виде маленького квадрата. На нем и сливаются стрелки. Возможны варианты альтернативной композиции, в которых один из внутренних фрагментов пуст.

Последняя композиция — циклическая (рис. 1 г1 и 1 г2). Переход по одной из стрелок после проверки условия означает выход из данного фрагмента. Другая стрелка ведет к внутреннему оператору S, выходная стрелка которого ведет вновь к проверке условия. Два подвида этой композиции различаются тем, что входная стрелка фрагмента ведет в первом случае к проверке условия (цикл с предпроверкой), во втором — к оператору S (цикл с послепроверкой). В обоих случаях фрагмент с оператором S не может быть пустым.

Ациклические фрагменты. Если во фрагменте ни на какой глубине не встречается циклическая композиция, то такой фрагмент — ациклический граф. Все пути по нему от входной до выходной стрелки — простые. Интересно подсчитать их число.

По пустому фрагменту или фрагменту из одного неструктурированного оператора проходит лишь один путь, состоящий из этого единственного фрагмента.

Число путей по фрагменту с последовательной композицией равно произведению чисел путей по его внутренним фрагментам. Здесь заключен источник быстрого роста числа различных путей.

Число путей по фрагменту с альтернативной композицией равно сумме чисел путей по двум внутренним фрагментам.

О числе контуров в базе контуров структурированной схемы. Циклическая композиция порождает на внешнем уровне один простой путь, состоящий из одного псевдооператора B для цикла с предпроверкой и из двух операторов $S\,B$ — для цикла с послепроверкой. На этом же уровне порождается один простой контур: $B\,S\,B$. Что касается числа простых путей на самом внутреннем уровне (уровне неструктурированных операторов), то оно остается равным 1 для случая предпроверки и совпадает с числом простых путей по внутреннему фрагменту при послепроверке. Наконец, число простых контуров — это сумма числа простых контуров во внутреннем фрагменте S (они остаются простыми и в композиции) и числа простых путей по этому же фрагменту (они встраиваются в контур $B\,S\,B$).

Сечения ациклического графа и их роль в доказательстве завершаемости алгоритма. Фактически доказано, что базовые простые контуры схемы находятся во взаимно однозначном соответствии с простыми путями по внутренним фрагментам всех циклических фрагментов этой схемы. Выделив множество Z этих внутренних фрагментов, можно иметь дело только с этими ациклическими графами и максимальными (не содержащимися ни в каком другом пути) путями по ним. В теореме 3.1.3 ссылки на простые контуры графа переходов можно заменить ссылками на пути (по умолчанию — максимальные) по графам из множества Z.

Число таких путей может быть значительно больше числа элементов множества Z, поэтому желательно найти способ переформулировать условие этой теоремы, ограничившись упоминанием только графов из множества Z, а не путей по ним.

Вершины рассматриваемых графов — это операторы. Если требование 2) в условии теоремы 3.1.3 заменить на:

> $2'$) для каждого графа из множества Z можно выбрать такое его сечение (см. разд. 1.5.4), что для любого оператора из этого сечения имеет место $F(w_1) < F(w)$,

то и требование 2) будет выполняться, так что теорема останется справедливой.

В сечение, удовлетворяющее требованию $2'$, не могут входить операторы, не меняющие значений переменных, т. е. ни операторы (типа B) проверки условий, ни пустые операторы. Например, во фрагменте

$i := 1;\ j := 1;$	$\{S1\}$
while $i > 0$ **do**	$\{B1\}$
if $j < 0$ **then**	$\{B2\}$
$j := j + 1$	$\{S2\}$

сечение с требуемым свойством для внутреннего фрагмента ($B2\,S2$) единственного цикла построить невозможно. Действительно, этот фрагмент — альтернативный оператор, одна из ветвей которого (**else**-ветвь) пуста. Простой путь, содержащий оператор-условие $B2$ и (неявно) эту пустую ветвь, не может быть рассечен

требуемым образом. Поэтому теорема 3.1.3 при замене условия 2) на 2′) не может обосновать завершаемость этого алгоритма. Функция F для этой программы не может быть построена и никаким другим способом, так как исполнение алгоритма действительно зацикливается.

Сечения структурированных фрагментов. Рассмотренный пример помогает уяснить следующие правила построения сечений (приемлемых для использования в теореме) для различных ациклических фрагментов.

Пустой фрагмент не имеет сечений. Фрагмент из одного неструктурированного оператора имеет единственное сечение, состоящее из этого оператора, если он меняет значения каких-либо переменных, и не имеет сечений, если он только проверяет условие.

В качестве сечения фрагмента, полученного последовательной композицией, можно взять сечение любой из компонент. Здесь переход от путей к сечениям особенно выгоден.

Фрагмент, полученный альтернативной композицией, не имеет сечений, если ими не обладает хотя бы одна из компонент. В противном случае сечение такого фрагмента получается объединением любого сечения одной компоненты с любым сечением другой компоненты.

Иных типов структурированных ациклических фрагментов быть не может.

Сечения неструктурированных фрагментов. Сечения ациклических графов, в которых данные выше правила структурирования не соблюдаются, должны строиться иными методами. Обычно это достаточно просто сделать, непосредственно опираясь на определение сечения.

Вернемся к ациклическому графу, состоящему из операторов $S2$–$S7$ примера 3.1–2 (тело единственного в этом примере цикла). Из рис. 3.1–1 легко усмотреть, что из операторов, меняющих состояние программы, можно построить два сечения: $\{S5\ S6\}$ и $\{S7\}$. Первому сечению соответствует функция F, вычисляемая по формуле $m + n - i - j + 2$, второму — по формуле $m + n - k + 1$. Завершаемость алгоритма интуитивно очевидна, но эта интуиция как раз и опирается на неявное использование одной из упомянутых функций (обращение в ноль первой функции свидетельствует об исчерпании массивов a и b, второй — о заполнении массива c).

3.1.5. Экономия памяти

Переменная (идентификатор) в программе на алгоритмическом языке или в операторной схеме и адрес ячейки в машинной программе — это если не одно и то же, то кровные сестра и брат. Переменная — это наименование, приданное на время некоторому значению, адрес для машины служит наименованием содержимого некоторой ячейки памяти, т. е. тоже значения.

Значение, нуждающееся в именовании, чтобы им можно было воспользоваться в другой точке программы и в другой момент ее исполнения, будем называть *величиной*.

Не последнее место в программировании занимает проблема экономии памяти, сводящаяся, грубо говоря, к поочередному использованию одного и того же участка памяти (и, следовательно, одного и того же адреса) в разных, содержательно не связанных целях. Программа работала бы точно так же, если бы это были разные участки. С позиции понятия «переменная» эта проблема звучит так: в каких случаях можно совместить обозначения содержательно разных величин, используемых программой? А также — в каких случаях можно считать, что различные величины лишь случайно, без веских оснований, получили одинаковое наименование?

В общем случае ответить на эти вопросы невозможно или очень трудно. Приведем пример, ставший классическим.

Пусть требуется обратить матрицу A. Игнорируя сложности, связанные с плохой обусловленностью матрицы и с желательной минимизацией числа действий, можно предложить следующий способ организации вычислений.

Обратная матрица — это решение уравнения $A\,X = E$, где E — единичная матрица того же порядка n, что и матрица A. Решение можно найти за n шагов, выполняя на i-м шаге следующие действия: делим i-ю строку расширенной матрицы $(A\;E)$ на $a[i,i]$, из j-й строки вычитаем полученную строку, умноженную на $a[j,i]$, для $j = 1, \ldots, n, \quad j \neq i$. В итоге этих преобразований исходного уравнения в равносильные ему получим единичную матрицу на месте A и обратную матрицу на месте E. Вот фрагмент соответствующей программы в предположении, что единичная матрица в правой половине расширенной матрицы $a[1\,..\,n, 1\,..\,2\,n]$ уже была сформирована:

```
for i := 1 to n do
begin r := 1/a[i,i];
  for k := 1 to n+n do a[i,k] := a[i,k] * r;
  for j := 1 to n do
    if j <> i then
    begin r := a[j,i];
      for k := 1 to n+n do a[j,k] := a[j,k] - r * a[i,k]
    end
end
```

До i-го шага $a[j,k] = 0$ при $k > n+i$, $k \neq n+j$, поэтому столбцы расширенной матрицы с такими номерами k не изменяются, т. е. остаются столбцами единичной матрицы. После i-го шага то же самое можно сказать про столбцы с номерами k для $1 \leqslant k \leqslant i$. Окончательное значение элемента $a[i, n+i]$ возникает в тот же момент, когда элемент $a[i,i]$ обращается в 1 (или чуть позже), а окончательные значения элементов $a[j, n+i]$ для $j \neq i$ — когда элементы $a[j,i]$ обнуляются.

Все это означает, во-первых, что вычисления во внутреннем цикле можно вести лишь для значений k от $i+1$ до $n+i$, и, во-вторых, что столбцы правой половины расширенной матрицы можно по мере их вычисления помещать в памяти на место ставших известными независимо от матрицы A, а значит — ненужными, столбцов левой половины матрицы. В измененной программе:

```
for i := 1 to n do
begin r := 1/a[i,i];
  for k := 1 to n do a[i,k] := a[i,k] * r;
  a[i,i] := r;
  for j := 1 to n do
    if j <> i then
    begin r := a[j,i];
      for k := 1 to n do a[j,k] := a[j,k] - r * a[i,k];
      a[j,i] := -r
    end
end
```

вместо массива с $2n^2$ элементами удается обойтись массивом размером n^2.

Без словесных разъяснений, предваряющих этот вариант программы, не очень просто понять, что именно она делает и на какой идее основана.

Пример этот типичен не только для алгоритмов линейной алгебры. Достаточно типичен и отказ от бесполезных вычислений (от получения единичной матрицы в левой половине массива) как предпосылка для применения этого приема.

Ограничимся этой иллюстрацией приемов экономии памяти для программ, работающих с массивами. Дальше в этом разделе будем рассматривать программы только со скалярными (неструктурированными) переменными.

Если некоторое значение помещено в ячейку памяти, т. е. присвоено некоторой переменной, то оно должно находиться там до тех пор, пока оно еще может быть использовано. Слова «до тех пор» означают — во время исполнения всех операторов, находящихся на пути от оператора, присвоившего переменной это значение, до оператора, который его использует. Ни один из этих операторов не имеет права поместить в эту ячейку другое значение, т. е. присвоить значение переменной с тем же именем. Это — основное ограничение на переименование переменных в программе. Оно не мешает такому оператору использовать текущее значение переменной для своих нужд.

Путь по графу переходов операторной схемы от оператора, вырабатывающего некоторую величину (т. е. значение), до оператора, использующего это значение, назовем *маршрутом* этой величины. Маршруты, берущие начало от одного и того же оператора, безусловно являются маршрутами одной и той же величины. Будем говорить, что такие маршруты составляют *пучок*.

Другой оператор, вырабатывающий значение той же переменной, потенциально является родоначальником пучка маршрутов другой величины. Однако если обнаруживается, что какие-нибудь маршруты таких двух пучков встречаются в месте использования данной переменной, то следует признать, что эти маршруты, а с ними — и пучки, *зацепляются* — относятся к одной и той же величине.

Объединение всех зацепляющихся друг за друга пучков одной и той же величины называется *областью действия* соответствующей переменной, т. е. данного этой величине обозначения. (Речь не идет о блочной структуре программ, где тот же термин выражает иной, хотя и родственный по содержанию, смысл.)

Если оператор имеет переменную как аргументом, так и результатом, то это само по себе не является признаком зацепления маршрутов. Область действия переменной захватывает оператор, имеющий эту переменную своим результатом, только в части этого результата. Промежуточные операторы любого маршрута, попавшего в эту область, захватываются полностью, т. е. данная переменная рассматривается в качестве неявного аргумента и результата такого оператора. Оператор, для которого данная переменная является только аргументом (концевой оператор непродолжаемого маршрута), захватывается лишь в части этого аргумента.

Техника построения пучка маршрутов переменной может быть близка к той, которая была использована в разд. 2.4.2 для построения каркаса графа. Однако здесь возникает затруднение. Строить пучок следует, начиная с одного из операторов, вырабатывающего значение этой переменной. Оператор, использующий значение переменной, служит концом одного из ее маршрутов, но продолжаем этот маршрут или нет — сказать трудно. Поэтому, действуя с запасом, сначала следует построить пучок ациклических путей, берущих начало в выбранной вершине. Каждый такой путь заканчивается либо оператором, вырабатывающим значение рассматриваемой переменной (такой оператор не может стать промежуточным оператором маршрута), либо оператором, все преемники которого (если они есть) уже захвачены пучком. В полученном множестве кандидатов в промежуточные операторы маршрутов переменной следует сохранить лишь те операторы, которые достижимы из выявленных концов маршрутов при движении навстречу стрелкам графа переходов.

Если при написании программы ее автор стремился давать каждой содержательно обособленной величине свое наименование, то каждая переменная будет иметь (в меру этого старания) лишь одну область действия. Это создает наибольшую свободу для последующей экономии памяти — сокращения числа используемых переменных. Если автор предпринял какие-то усилия по совмещению обозначений разных величин, то перед оптимизацией программы по числу обозначений полезно нейтрализовать последствия этих усилий, выявив области действия переменных.

Пусть множество всех областей действия всех переменных программы уже построено. Из сказанного выше следует, что про исходные обозначения величин можно забыть и считать, что области действия соответствуют уже не переменным, а величинам. Построим неориентированный граф, имеющий эти области своими вершинами. Две вершины соединяются ребром, если найдется оператор, захваченный как той, так и другой областью в части какого-либо аргумента или результата. Этот граф назовем *графом несовместимости* величин схемы. Найдем его правильную раскраску (см. разд. 1.5.4).

Достаточно очевидно, что величинам, для которых соответствующие им вершины графа несовместимости соединены ребром, не могут быть приданы одинаковые наименования (они не могут храниться в одной ячейке). При правильной раскраске они получают разные цвета. Напротив, всем величинам, которым соответствуют одноцветные вершины, можно дать одно и то же наименование.

Таким образом, найти наиболее экономное распределение памяти или же минимальную раскраску графа несовместимости величин — это одна и та же задача. Эта эквивалентность задач экономии памяти и раскраски вершин графа вызывает скорее огорчение, чем радость: давно известно, что вторая задача относится к числу переборных (трудно решаемых), для всего класса которых многолетние усилия математиков не привели к обнаружению эффективного алгоритма их решения.

Мораль для авторов программ: если у вас есть веские основания для совмещения обозначения величин, то действуйте в соответствии с ними, не слишком опасаясь того, что ваши усилия будут кем-то или чем-то нейтрализованы.

3.2. Формализация семантики языков программирования

Есть такая наука семиотика — общая теория знаковых систем (к которым, бесспорно, относятся и языки программирования). В этой теории, как и в некоторых других, наряду с тремя источниками: формальная логика, лингвистика и философия, принято выделять три составные части: *синтаксис* (или синтактику), трактующий правила построения текстов исследуемого языка, *семантику*, изучающую связи между текстами и смыслом, который они выражают, и *прагматику*, исследующую отношения между знаковыми системами и теми, кто ими пользуется. У нас прагматический в этом смысле характер носят замечания, появляющиеся время от времени по ходу изложения.

Первая попытка формализовать описание синтаксиса языка программирования — Алгола 60 — оказалась весьма удачной и стимулировала многочисленные исследования как теоретического, так и прикладного характера в области формальных грамматик и методов синтаксического анализа.

Семантике повезло меньше, но уже к концу 60-х годов сложились три основных подхода к формальному описанию семантики языков программирования.

1. *Операционный* подход: семантика описывается в терминах некоторой машины, чаще всего — воображаемой (абстрактной). Одним из наиболее известных и характерных примеров такого подхода служит описание семантики Алгола 68. Следует упомянуть и так называемый венский метод описания семантики. Фактически этот подход реализуется всякий раз, когда создается «базовый» интерпретатор языка, служащий эталоном правильности любой другой реализации. Пионером здесь был Дж. Мак-Карти с его Лисп-интерпретатором, написанным на Лиспе же.

В процессе работы машины меняется *состояние программы* — совокупность значений ее переменных. Результат исполнения программы — это состояние, при котором машина завершает работу по данной программе. Естественно, оно зависит от начального состояния — того, в котором машина начала работать.

2. *Аксиоматический* или *деривационный* подход, основанный на системе аксиом, постулирующих свойства основных конструкций языка, и правил вывода, позволяющих получать исходя из этих аксиом свойства любых программ и их фрагментов (П. Наур, Р. Флойд, Ч. А. Р. Хоор и др.). Таким образом, строится некая формальная теория — «исчисление программ», формулы которой выражают утверждения о состоянии программы.

В разд. 3.1.2 мы немного познакомились с этим подходом (в ином оформлении) на примере программ, представленных своими схемами, дуги которых размечены предикатами (точнее, логическими формулами), описывающими состояние программы при прохождении данной дуги.

3. *Денотационный* подход, получивший также название математической семантики. Здесь, как и в операционном подходе, в основу кладется модель (интерпретация) алгоритма, но она описывается не в терминах машины, а более традиционными для математики средствами — в терминах некоторой совокуп-

ности множеств, отношений и отображений. По замыслу эти средства должны описывать результаты исполнения машиной программы в целом и ее отдельных частей, точнее — связь между начальным и заключительным состоянием. Основы этого подхода заложили Д. Скотт, К. Стрейчи, Я. В. де Баккер и др. Не будет преувеличением утверждать, что эти авторы вознамерились приблизить абстрактные машины теории вычислимости к реальным компьютерам, а главное — поднять строгость описания семантики реальных языков программирования до уровня, сложившегося в этой теории. Для языка Лисп обе эти цели достижимы.

Речь шла о динамическом выводе утверждений о свойствах программ. Статический подход, когда рассматривается не конкретный вид исходных данных, а некие общие условия, которым эти данные должны удовлетворять, более интересен. Но что-либо доказать о свойствах результата вычислений можно, лишь убедившись, что вычисления когда-либо завершаются, фигурально выражаясь — когда бесконечность, даже потенциальная, может быть изгнана из семантики программ.

Внутри каждого из подходов можно выделить ряд различных приемов и изобразительных средств.

Названные подходы не исключают, а дополняют друг друга, освещают семантику языков программирования с разных сторон. Операционный подход ближе других стоит к задаче создания как интерпретаторов языков программирования, так и компиляторов для них. По традиции будем считать его первичным, отражающим содержательную сторону программирования. Однако существует и иная позиция по отношению к операционному — якобы «фон-неймановскому» — взгляду на программирование, отдающая приоритет функциональному, денотационному подходу (Дж. Бакус и др.).

Предельная строгость и четкость — качества, которыми программисты (как и плоды их трудов — программы, спецификации и пр.) должны обладать в значительно большей степени, чем даже математики. Последние вправе рассчитывать на интеллект и интуицию своих коллег, тогда как первые имеют дело с компьютерами — шкафами и ящиками, нашпигованными всего лишь электроникой. Машинно-ориентированные понятия и термины (ячейка, адрес и пр.) в чем-то помогают установить требуемую четкость при описании семантики, но основная цель — добиться ясности понимания самих программных конструкций, языковых средств, используемых при их написании. Отсюда и берет начало стремление отойти от операционного подхода в направлении к двум другим.

Аксиоматический подход в большей степени помогает (по крайней мере, призван помочь) составлять правильные программы, не полагаясь на их отладку методом «тыка и ляпа» (на опыт, как мерило справедливости, — сказал бы Фейнман). Однако, как и для традиционных логических исчислений, надо уметь обосновывать общезначимость аксиом и состоятельность правил вывода исчисления программ. Такую возможность предоставляет (как многие надеются) денотационный подход, в котором на основе правил работы абстрактной машины в операционной семантике используется хорошо развитый и изученный аппарат теории множеств и формальной логики.

Остаток главы будет посвящен различным способам описания семантики некоторого модельного языка и последствиям выбора того или иного способа.

3.2.1. Модельный язык и его операционная семантика

Пусть Z — множество целых чисел, Z = $\{c, \ldots\}$; BV = $\{true, false\}$ — множество логических значений; D = Z ∪ BV, D = $\{d, \ldots\}$; $d?$ — неопределенное значение, $d? \notin$ D; D′ = D ∪ $\{d?\}$. Множество D назовем *предметной областью*. Здесь и далее многоточие означает возможность снабжать названные буквы индексами и штрихами. Буквы $c, d, \ldots$ будут служить обозначениями произвольных элементов соответствующих множеств.

Далее, IV = $\{v, \ldots\}$ — множество переменных; IE = $\{e, u, \ldots\}$ — множество выражений с целочисленными значениями; BE = $\{b, \ldots\}$ — множество логических выражений. В нотации Бакуса–Наура, известной нам по разд. 2.1.3, классы IE и BE определяются металингвистическими формулами:

IE ::= Z | IV | IE + IE | ... | **if** BE **then** IE **else** IE **fi** ,
BE ::= BV | IE < IE | ... | ¬ BE | BE ∧ BE| ...
| **if** BE **then** BE **else** BE **fi**.

Здесь многоточия означают, что кроме знаков ‘+’, ‘<’, ‘∧’ можно пользоваться и другими знаками операций (арифметических, операций отношения и логических).

Полагаем, что содержательная семантика операций этих трех классов известна (вернее, что абстрактная машина умеет их выполнять). Разумеется, все эти операции должны быть детерминированными и иметь следующие типы:

$$\text{ArOp} : \text{Z}\times\text{Z} \to \text{Z}, \qquad \text{RelOp} : \text{Z}\times\text{Z} \to \text{BV}, \qquad \text{BoolOp} : \text{BV}\times\text{BV} \to \text{BV}$$

соответственно их классу. Логическое отрицание ¬ : BV → BV — единственная одноместная операция.

Семантика условного выражения **if** b **then** $e1$ **else** $e2$ **fi** такова. Вычисляется значение выражения b. Это значение (*true* или *false*) определяет выбор между внутренними выражениями: $e1$ или $e2$. Выбранное выражение вычисляется, полученное значение становится значением всего условного выражения.

Пока — до разд. 3.4 — вызовы функций в качестве выражений не допускаются. Это позволяет считать, что вычисление значения выражений не изменяет состояние, как говорят, — не сопровождается побочным эффектом.

Для синтаксического класса St пока что примем следующее определение:

St ::=	**skip**	— пустой оператор,
|	**abort**	— неисполнимый оператор,
|	IV := IE	— оператор присваивания,
|	*Write*(*ExprList*)	— оператор вывода,
|	*Read*(*VarList*)	— оператор ввода,
|	St ; St	— последовательность операторов,
|	**if** BE **then** St **else** St **fi**	— условный оператор,
|	**while** BE **do** St **od**	— оператор цикла,
|	*Decl*	— оператор объявления,
|	**begin** *BlTail* **end**	— блок,

где

VarList	::=	IV | *VarList* , IV	— список переменных,
ExprList	::=	IE | *ExprList* , IE	— список выражений,
BlTail	::=	St | *Decl* ; *BlTail*	— остаток блока,
Decl	::=	**loc** IV	— объявление переменной.

Опишем содержательную (операционную) семантику операторов. Вместе с упомянутой выше семантикой выражений ее следует рассматривать как естественную интерпретацию формул исчисления программ и их денотационной семантики. По отношению к последним она играет ту же роль, что интерпретации логических исчислений и языков в традиционной математической логике. Наиболее существенное отличие операционной семантики от этих традиционных интерпретаций заключается в ее динамизме — исполнении операторов программы в определенной последовательности. Различные интерпретации одной и той же программы получаются при рассмотрении процесса ее исполнения из разных начальных состояний. В ходе этого процесса состояния сменяют друг друга, тогда как в обычной логике интерпретация формальной теории — это фиксированное, статичное множество значений переменных. Естественно, что именно этому процессу смены состояний, а не различию начальных состояний, будет уделено особое внимание. Можно сказать, что этот процесс обладает своего рода эргодичностью: наблюдая за сменой состояний в ходе исполнения одной, достаточно сложной, чтобы оказаться типичной, программы, можно увидеть все то общее, что характеризует исполнение этой и даже всех других программ из разных начальных состояний. Подробнее на этом останавливаться не будем и никакую «эргодическую теорему» формулировать, тем более — доказывать, не станем.

Начнем с первичных операторов, не содержащих вхождений других операторов.

Пустой оператор **skip** исполни́м из любого состояния, и его исполнение завершается в этом же состоянии.

Невыполнимый оператор **abort** не исполни́м (не завершается) ни из какого состояния.

Оператор присваивания S вида $v := e$ исполняется так. В текущем состоянии вычисляется значение выражения e. Если завершить это вычисление не удалось, то оператор S не исполни́м. Не исполни́м он и в случае, если переменная v не была ранее объявлена. Если же все благополучно, то программа переходит в новое состояние, в котором переменная v получает вычисленное значение выражения e, а все остальные переменные сохраняют прежние значения.

Ниже, в разд. 3.3.1, индексы, заключаемые в квадратные скобки '[' и ']', органически войдут в модельный язык. А в метаязыке индексы у различных букв и даже идентификаторов будем записывать в обычном математическом стиле.

Оператор вывода $Write(e_1, \ldots, e_m)$ ничего не меняет в состоянии программы, а только передает некоторую информацию о нем во внешнюю среду. С этой точки зрения его семантика совпадает с семантикой пустого оператора.

Исполнение оператора ввода $Read(v_1, \ldots, v_n)$ приводит к изменению значений переменных $v_1, \ldots, v_n$. Новые значения берутся извне. Они никак не связаны ни с прежними значениями этих переменных, ни с текущими значениями других переменных программы, остающимися без изменения.

Далее следуют структурированные операторы (композиции операторов).

Составной оператор S имеет вид $S1; S2$. Его семантика — из текущего состояния исполнить оператор $S1$. Если он не исполни́м, то не исполни́м и оператор S. Если исполнение оператора $S1$ завершилось, то из полученного состояния исполняется оператор $S2$. Полученный результат будет и результатом исполнения оператора S.

Условный оператор S имеет вид **if** b **then** $S1$ **else** $S2$ **fi**, где b — логическое выражение, $S1$ и $S2$ — операторы. Его содержательная семантика: вычислить значение b, если оно равно *true*, то исполнить $S1$, если *false* — то исполнить $S2$, если значение b не определено, то оператор S исполнен быть не может. Допускается сокращенная форма условного оператора: **if** b **then** $S1$ **fi**, означающая в точности то же, что и **if** b **then** $S1$ **else skip fi**.

Оператор цикла S имеет вид **while** b **do** $S1$ **od**, где b — логическое выражение, $S1$ — оператор (тело цикла). Содержательная семантика: в текущем состоянии вычисляется значение выражения b. Если это значение не определено, то оператор S не исполни́м. Если оно равно *false*, то исполнение оператора S завершается, при этом состояние не изменяется. Если значение b равно *true*, то исполняется тело цикла — оператор $S1$ — и из полученного состояния все повторяется. Таким образом, b — это условие повторения цикла.

При исполнении объявления **loc** v входящей в него переменной v придается новая роль в соответствии с этим объявлением. Ее значение становится неопределенным. Прежние роль и значение на время исполнения блока забываются. Ни одна переменная не должна быть объявлена дважды в одном и том же блоке. Поскольку исполнение объявления сопряжено с действиями, существенно меняющими состояние программы, есть все основания считать объявление одной из разновидностей оператора.

Мы несколько забежали вперед, так как все переменные пока что играют одну и ту же роль — получать и хранить до поры до времени целочисленные значения. Ниже (в разд. 3.3.1) роли станут существенно разнообразнее.

Содержательная семантика блока S, имеющего вид

begin loc v_1; ... ; **loc** v_n; $S1$ **end**

описывается следующим образом. В начале исполнения блок открывается — возникает новое состояние, в котором все переменные сохраняют присвоенные им значения.

Затем поочередно исполняются все объявления переменных. Вслед за этим исполняется оператор $S1$. Если исполнить его не удалось, то не исполни́м и оператор S. В ходе исполнения оператора $S1$ как объявленным (локализованным) в блоке переменным, так и объявленным ранее и не переобъявленным в нем (глобальным) переменным могут быть присвоены новые значения.

По завершении исполнения оператора $S1$ блок закрывается. При этом переменные, объявленные в блоке, перестают существовать. Для всех одноименных переменных, объявленных к началу исполнения блока S, восстанавливается их прежняя роль, а для тех, которые были переобъявлены в нем, — и значения. Для глобальных переменных сохраняются значения, присвоенные им при исполнении оператора $S1$.

3.2.2. Исчисление программ

Аксиоматическая (деривационная) семантика имеет своим предметом логическое исчисление, использующее наряду с обычными логическими формулами исчисления высказываний или исчисления предикатов (обычно первого порядка) формулы специального вида, содержащие внутри себя операторы некоторого алгоритмического языка и описывающие их логические свойства — связь между состояниями переменных программы до и после исполнения этих операторов.

Рассмотрим, не вдаваясь в очевидные детали, пример подобного исчисления программ на материале ранних работ Р. Флойда и Ч. А. Хоора, в которых были заложены основы аксиоматической семантики программ.

Будем использовать операторные формулы двух видов: слабые формулы $p\,\{S\}\,q$ и сильные формулы $p\,[S]\,q$, где S — оператор, p и q — логические выражения, называемые, соответственно, предусловием и постусловием для оператора S. В этом контексте их класс будет расширен до формул некоторого логического исчисления 1-го порядка. Присутствующие в них предикаты описывают состояние программы с помощью некоторых общих понятий. Это предпочтительнее, чем говорить лишь о значениях отдельных переменных, но не всегда оказывается возможным.

Слабая формула имеет следующий содержательный смысл: если исполнение оператора S началось из состояния w, удовлетворяющего условию p, и завершилось (что не гарантируется) в некотором состоянии w_1, обозначаемом также $s(w)$, то последнее удовлетворяет условию q:

$$(p\,\{S\}\,q) \;\equiv\; \forall\, w\,(p(w) \supset \forall\, w_1\,(w_1 = s(w) \supset q(w_1))). \qquad (1)$$

Смысл сильной формулы: если оператор S начал исполняться из состояния w, удовлетворяющего условию p, то его исполнение обязательно завершается и любое выработанное в результате состояние w_1 удовлетворяет условию q:

$$(p\,[S]\,q) \;\equiv\; \forall\, w\,(p(w) \;\supset\; \exists\, w_1\,(w_1 = s(w)) \;\wedge\; \forall w_2\,(w_2 = s(w) \supset q(w_2))). \qquad (2)$$

Операторные формулы можно было бы записывать и в более традиционной форме: $q = \mathrm{Week}(p, S)$ — для слабых формул, $p = \mathrm{Strong}(S, q)$ — для сильных, где Week и Strong — символы соответствий (в общем случае).

Свойства первичных операторов описываются очевидными аксиомами:

(A1) $p\,[\mathbf{skip}]\,p$,
(A2) $p\,\{\mathbf{abort}\}\,false$

при любой логической формуле p. Для оператора присваивания:

(A3) $p(v)\,\{v := e(v)\}\,\exists\, i\,(p(i) \;\wedge\; v = e(i))$, (Флойд)
(A4) $p(e)\,[v := e]\,p(v)$. (Хоор)

Здесь i — обозначение, отличное от всех обозначений переменных в программе, $p(i)$ — формула, возможно содержащая это обозначение, $p(v)$ и $p(e)$ — результат подстановки переменной v или, соответственно, выражения e вместо i в эту формулу. Аксиома Флойда означает, что если формула $p(v)$ была истинна до исполнения оператора S, то после его завершения можно утверждать, что при некотором значении i переменной v (очевидно, при ее старом значении) формула p остается истинной, а новое значение переменной v равно $e(i)$. Аксиома Хоора действует как бы в обратном направлении — чтобы формула $p(v)$ была истинна после завершения оператора присваивания, перед исполнением этого оператора она должна быть истинна при том значении переменной v, которое ей присваивается. Обе аксиомы в общих чертах отражают содержательную семантику оператора присваивания, а в детали мы условились не вдаваться. Напомним, что в реальных языках программирования правила исполнения оператора присваивания содержат достаточно много подробностей и оговорок.

Квантор существования в постусловии приносит мало радости (если, вернувшись домой, вы обнаружили, что ваша квартира ограблена, то вы знаете, что кто-то сделал это, но будет ли он пойман?). Если соотношение $v = e(i)$, рассматриваемое как уравнение относительно i, имеет при любом v, удовлетворяющем условию $p(v)$, конечное множество решений: $e_1, \ldots, e_k$, число k которых известно и неизменно, то аксиому А3 (не будем заключать в скобки имена аксиом, встречающиеся в тексте) можно заменить аксиомой

(А3′) $\quad p(v)\,\{v{:=}e(v)\}\,p(i) \;\wedge\; (i = e_1 \vee \ldots \vee i = e_k) \;\wedge\; v = e(i).$

Здесь каждое из выражений $e_j, \quad j = 1, \ldots, k$, зависит от v и каждое из равенств $i = e_j$ влечет за собой равенство $v = e(i)$, так что последний член в этой формуле не обязателен. Особенно благоприятен случай $k = 1$.

Если же ни условие p, ни правая часть e оператора присваивания не зависят от v, то аксиома еще более упрощается:

(А3″) $\quad p\,\{v := e\}\,p \;\wedge\; v = e.$

Семантика оператора вывода $Write(e_1, \ldots, e_m)$, не меняющего, как уже говорилось, состояния программы, совпадает с семантикой пустого оператора:

(А5) $\quad p\;[Write(e_1, \ldots, e_m)]\;p.$

Напротив, исполнение оператора ввода $Read(v_1, \ldots, v_n)$ вносит значительную неопределенность в состояние программы. Значения переменных $v_1, \ldots, v_n$ меняются, вместе с ними могут измениться значения любых выражений или предикатов, зависящих от этих переменных. Поэтому предикат, описывающий состояние программы после исполнения оператора ввода, в общем случае должен быть написан автором программы, отвечающим и за подготовку вводимых данных, значения которых удовлетворяли бы этому предикату. Написать аксиому, исчерпывающим образом описывающую семантику оператора ввода, оказывается невозможным.

Тем не менее есть одно утверждение, всегда истинное после исполнения оператора ввода. Обозначим через $v0_1, \ldots, v0_n$ константы, фиксирующие вводимые этим оператором значения переменных $v_1, \ldots, v_n$. Тогда формулу

(A6) $\quad true\ \{Read(v_1, \ldots, v_n)\}\ v_1 = v0_1 \wedge \ldots \wedge v_n = v0_n$

можно принять за аксиому, не полностью, но в довольно существенной части описывающую семантику оператора ввода. Иногда, особенно если этот оператор лишь однократно исполняется в начале работы программы, этого оказывается достаточно, чтобы доказать все необходимое о свойствах программы.

Замечание. Для любого выражения $e(v_1, \ldots, v_n)$, содержащего вхождения переменных v_i, подстановочность равенства обеспечивает справедливость формулы $e(v_1, \ldots, v_n) = e(v0_1, \ldots, v0_n)$ сразу после исполнения оператора ввода. Поэтому подобные формулы можно добавлять в качестве конъюнктивных членов к постусловию аксиомы A6, не занимаясь их выводом.

Правила вывода в исчислении программ удобно записывать в привычном виде:

$$\frac{F_1 \quad \ldots \quad F_k}{F}$$

где $F_1, \ldots, F_k$ — логические или операторные формулы (*посылки* правила), F — формула, чаще всего, операторная (*заключение* правила). Правило, как обычно, означает, что если вывод (строящаяся по шагам последовательность формул) уже содержит формулы $F_1, \ldots, F_k$, то на очередном шаге к нему может быть приписана в конце формула F.

Правило вывода

$$\text{(R1)} \quad \frac{p \supset p1 \quad p1\,\{S\}\,q1 \quad q1 \supset q}{p\,\{S\}\,q}$$

позволяет усиливать предусловие и ослаблять постусловие операторной формулы. Бывает удобнее использовать варианты этого правила:

$$\text{(R1}'\text{)} \quad \frac{p \supset p1 \quad p1\,\{S\}\,q}{p\,\{S\}\,q}, \qquad \text{(R1}''\text{)} \quad \frac{p\,\{S\}\,q1 \quad q1 \supset q}{p\,\{S\}\,q}.$$

Следующие правила для структурных операторов соответствуют их содержательной семантике:

$$\text{(R2)} \quad \frac{p\,\{S\}\,p1 \quad p1\,\{S1\}\,q}{p\,\{S; S1\}\,q},$$

$$\text{(R3)} \quad \frac{p \wedge b\,\{S1\}\,q \quad p \wedge \neg\, b\,\{S2\}\,q}{p\,\{\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}\}\ q},$$

$$\text{(R4)} \quad \frac{p \wedge b\,\{S1\}\,p}{p\,\{\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}\}\ p \wedge \neg b}.$$

В последнем правиле формула (условие) p называется *инвариантом цикла*. Единственная посылка правила требует, чтобы это условие, если оно было истинно перед исполнением тела $S1$ оператора цикла, оставалось истинным и после завершения этого тела (отсюда и название — инвариант). Тривиальный (слабейший) инвариант *true* существует всегда, а инвариант, во всем существенном отражающий свойства оператора цикла при данных b и p, найти очень нелегко, если вообще возможно (ср. с разд. 3.1.2). В посылке зафиксировано также, что тело исполняется лишь при соблюдении условия b, входящего в состав оператора

цикла. Заключение правила гласит, что если весь цикл начал исполняться при истинном инварианте p, то и по завершении цикла инвариант остается истинным, а условие повторения b становится ложным.

В доказательстве свойств программ участвуют все традиционные аксиомы и правила вывода соответствующего логического исчисления и той области (например, арифметики), к которой относится задача, решаемая программой.

Пример. Программа вычисления факториала:

$Read(n);\ f := 1;$
while $n > 1$ **do** $f := f * n;\ n := n - 1$ **od**

Пусть $n0$ — это, как выше, значение переменной n, вводимое оператором $Read(n)$. Мы хотим доказать, что по окончании исполнения этой программы переменная f приобретает значение $n0!$. Начинаем строить вывод. Общезначимые арифметические формулы будем включать в него на правах аксиом.

1) $true\ \{Read(n)\}\ n = n0 \wedge n! = n0!$ (аксиома А6 + замечание)
2) $n = n0 \wedge n! = n0!\ \{f := 1\}\ n = n0 \wedge n! = n0! \wedge f = 1$ (аксиома А3″)
3) $true\ \{Read(n);\ f := 1\}\ n = n0 \wedge n! = n0! \wedge f = 1$
(результат шагов 1, 2; правило R2)

Далее наступает самый ответственный момент в построении вывода — в программе стоит оператор цикла, и надо угадать (найти, определить) вид его инварианта. Попробуем не спешить с догадками и запишем его в виде $p(n, f)$. Это означает, что формула p содержит вхождения переменных n и f. Запись $p(e, u)$ обозначает результат подстановки выражений e и u вместо этих вхождений.

Чтобы воспользоваться правилом R4, надо суметь вывести посылку

$$p(n, f) \wedge n > 1\,\{f := f * n; n := n - 1\}\,p(n, f)$$

этого правила. А для этого сначала надо определить вид постусловия для оператора $f := f * n$ при предусловии $p(n, f) \wedge n > 1$:

$$p(n, f) \wedge n > 1\{f := f * n\}p(n, f/n) \wedge n > 1 \qquad \text{(аксиома А3′)}$$

Аналогично для оператора $n := n - 1$:

$$p(n, f/n) \wedge n > 1\{n := n - 1\}p(n + 1, f/(n + 1)) \wedge n + 1 > 1 \qquad \text{(аксиома А3′)}$$

Теперь все готово для следующего шага:

$$p(n, f) \wedge n > 1\{f := f * n; n := n - 1\}\ p(n + 1, f/(n + 1)) \wedge n \geqslant 1 \qquad \text{(правило R2)}$$

Но этого еще мало. Правило R4 заработает, если предварительно удастся вывести формулу:

(Н) $p(n + 1, f/(n + 1)) \wedge n \geqslant 1 \supset p(n, f)$.

Кроме того, использовать результат шага 3) удастся лишь при условии справедливости формулы

(Н′) $n = n0 \wedge n! = n0! \wedge f = 1 \supset p(n, f)$,

а заключение правила R4 приведет нас к цели, если формула

(Н″) $p(n, f) \wedge n \leqslant 1 \supset f = n0!$

окажется истинной. Приглядевшись к формулам (H)–(H″), можно предположить, что формула $p(n, f)$ должна иметь структуру $p'(n, f) \wedge n \geqslant 1$. При этом левая часть формулы (H″) примет вид $p'(n, f) \wedge n \geqslant 1 \wedge n \leqslant 1$ и сможет стать началом следующей цепочки импликаций (здесь и далее в цепочках эквивалентностей и (или) следований эти отношения обозначены символами $\Leftrightarrow$ и $\Rightarrow$):

$$p'(n, f) \wedge n \geqslant 1 \wedge n \leqslant 1 \Rightarrow p'(n, f) \wedge n = 1 \Rightarrow p'(1, f).$$

Формула (H″) окажется выводимой, если цепочка допускает продолжение

$$p'(1, f) \Rightarrow f = n0!$$

Теперь еще один взгляд на формулу (H′) приводит к мысли, что формула $p'(n, f)$ должна быть вида $F(n, f) = n0!$, а конкретнее — вида $G(n)\, f = n0!$. С учетом всего этого последняя формула, а также формулы (H) и (H′) приводятся к виду:

$$G(1)\, f = n0! \supset f = n0!,$$
$$G(n+1)\, (f/(n+1)) = n0! \wedge n \geqslant 1 \wedge n \geqslant 1 \supset G(n)\, f = n0! \wedge n \geqslant 1,$$
$$n = n0 \wedge n! = n0! \wedge f = 1 \supset G(n)\, f = n0! \wedge n \geqslant 1.$$

Отсюда после упрощений последовательно получаем:

$$G(1) = 1,$$
$$G(n+1)/(n+1) = G(n) \quad \text{или} \quad G(n+1) = G(n)\,(n+1),$$

т. е. $G(n) = n!$, и

$$n = n0 \wedge n! = n0! \wedge f = 1 \supset n!\, f = n0! \wedge n \geqslant 1.$$

Последняя импликация истинна при дополнительном условии $n0 \geqslant 1$, которое следует наложить на вводимое в начале работы программы значение переменной n и включить его в качестве конъюнктивного члена в аксиому (A3) и в пред- и постусловия ее следствий.

Замечание. Внимательный читатель обнаружит, что достаточно условия $n0 \geqslant 0$, но либо программа будет выполнять лишнее умножение f на 1, либо работать с инвариантом будет чуточку сложнее.

Выпишем теперь шаги вывода, следующие за шагом 3):

4) $n = n0 \wedge n! = n0! \wedge f = 1 \wedge n0 \geqslant 1$
$\supset n!\, f = n0! \wedge n \geqslant 1$ (общезначима)

5) $true\, \{Read(n);\ f := 1\}\, n!\, f = n0! \wedge n \geqslant 1$ (3, 4; R1″)

6) $n!\, f = n0! \wedge n \geqslant 1 \wedge n > 1 \{f := f * n\}\, n!\, f/n = n0! \wedge n > 1$ (A3′)

7) $(n-1)!\, f = n0! \wedge n > 1 \{n := n - 1\}\, n!\, f = n0! \wedge n + 1 > 1$ (A3′)

8) $n!\, f = n0! \wedge n \geqslant 1 \wedge n > 1 \{f := f * n;\, n := n - 1\}$
$n!\, f = n0! \wedge n \geqslant 1$ (6, 7; R2)

9) $n!\, f = n0! \wedge n \geqslant 1 \{\textbf{while}\ n > 1\ \textbf{do}\ f := f * n;\, n := n - 1\ \textbf{od}\}$
$n!\, f = n0! \wedge n \geqslant 1 \wedge n \leqslant 1$ (8; R4)

10) $n!\, f = n0! \wedge n \geqslant 1 \wedge n \leqslant 1 \supset f = n0!$ (общезначима)

11) $n!\, f = n0! \wedge n \geqslant 1 \{\textbf{while}\ n > 1\ \textbf{do}\ f := f * n;\, n := n - 1\ \textbf{od}\}$
$f = n0!$ (9, 10; R1″)

12) $true \{Read(n);\ f := 1;\ \textbf{while}\ n > 1\ \textbf{do}\ f := f * n;\ n := n - 1\ \textbf{od}\} f = n0!$
(5, 11; R2)

То, что мы проделали, представляется чудовищно громоздким для доказательства «очевидного» свойства очень простенькой программы. Но не будем забывать, что аппарат исчисления предикатов первого порядка — это своего рода машинный язык для математиков. Доказывая свои теоремы, они («работающие математики», как сами они любят говорить) пользуются языками гораздо более высокого уровня, а главное — значительно менее формализованными (что хорошо в меру, так как ошибки в доказательствах становятся менее заметными). «Работающие программисты» также должны развивать интуицию, позволяющую им убеждаться в правильности своих программ до начала их отладки и тестирования (как известно, далеко не гарантирующих желаемого) и быстро находить и устранять ошибки в ходе отладки. Исчисление программ, основы которого были здесь изложены, а в последующем (разд. 3.2.7, 3.2.8) будут расширены и дополнены, может послужить лишь отправным пунктом или дорожным указателем на пути к развитию программистской интуиции.

3.2.3. Состояния и преобразователи состояний

В программах, написанных на любом алгоритмическом языке, существуют связи программных объектов, разделенных как по месту их расположения в программе, так и по времени их исполнения. Типичный пример — это связь между объявлением переменной, присваиванием ей значения и ее использованием в качестве выражения (т. е. использованием ее текущего значения). По существу то же самое — связь между заголовком описания функции (или процедуры), где вводятся обозначения ее параметров, вызовом функции, где задаются аргументы — значения, придаваемые параметрам при данном обращении к ней, и телом функции, где параметры выступают в роли операндов некоторых операций или аргументов других обращений. Этот вариант характерен для языков, называемых функциональными, но он используется и в языках операторного стиля. В любом варианте необходимо сохранить некоторые сведения об объекте, будь то переменная или параметр, или о его значении и донести их до места, где эти сведения используются и, возможно, пополняются (присваивание значения).

Для этого и служат состояния множества переменных программы. *Состояние* $w : \mathrm{IV} \to \mathrm{Z} \cup \{d?\}$ — это частичная функция, сопоставляющая каждой объявленной в программе переменной v ее текущее значение $w(v)$, возможно неопределенное (обозначаемое символом $d?$). Множество состояний обозначим буквой W.

Будем обозначать область определения (отправления) соответствия s через $\mathrm{Dom}(s)$, а множество значений (область прибытия) — через $\mathrm{Rng}(s)$.

Так, область определения состояния $\mathrm{Dom}(w) = \{\, v \mid \exists\, d\,(d = w(v))\,\}$ — это множество переменных, объявленных (доступных) в состоянии w.

Другой пример информации, которую надо как-то сохранять, — это состояние на момент входа в блок. Не располагая ею, невозможно осуществить выход из блока так, как этого требует содержательная семантика блока (см. разд. 3.2.1).

Каждому оператору S из множества операторов St сопоставим его семантику — *преобразователь состояний* — некоторое соответствие на множестве состояний W:

$$\mathrm{Sem}(S) : \mathrm{W} \to \mathrm{W},$$

которое в общем случае не обязано быть ни всюду определенным, ни детерминированным. Недетерминированность может возникнуть, например, если в программе используются операторы ввода в иных целях, кроме ввода начальных значений, или датчик случайных чисел для задания некоторых значений. Такие программы здесь рассматриваться не будут, если иное не сказано прямо.

Функция Sem имеет тип $\mathrm{St} \to (\mathrm{W} \to \mathrm{W})$. Ограничимся случаем, когда соответствие $\mathrm{Sem}(S)$ однозначно (иначе оператор S не был бы детерминирован), это позволяет называть $\mathrm{Sem}(S)$, как и другие преобразователи состояний, *семантическими функциями*. Множество всех таких функций обозначим Semant.

Семантику оператора S ($S1$, S' и т. п.) часто будем кратко обозначать вместо $\mathrm{Sem}(S)$ строчной буквой s ($s1$, s' и т. д.).

Иногда и на формализованном (денотационном) уровне семантику $\mathrm{Sem}(S)$ будем называть преобразователем состояний.

Множества состояний P, Q, ... занимают важное место в развиваемой теории. Каждому множеству P можно взаимно однозначно сопоставить его характеристическую функцию — такое всюду определенное отображение $p : \mathrm{W} \to \mathrm{BV}$, что

$$p(w) \equiv (w \in P).$$

Ниже, во всем разделе 3.2, предикатами будем называть характеристические функции множеств состояний.

Выделим класс предикатов, представленных логическими выражениями рассматриваемого языка. Однако значения этих выражений могут быть вычислены не в любом состоянии. Поэтому сопоставим логическому выражению b два множества состояний: B^+, в котором это выражение истинно, т. е. имеет значение *true*, и B^-, в котором оно ложно. Характеристические функции этих множеств обозначим b^+ и b^-. Функция b^+ отличается от функции, представленной выражением b, тем, что она определена (но принимает, как и b^-, значение *false*) там, где значение b не определено.

3.2.4. Целые и логические выражения

Пусть $\mathrm{Expr} = \mathrm{IE} \cup \mathrm{BE}$ — множество $\{e, \ldots\}$ выражений любого типа. Денотационная семантика выражений задается своим для каждого выражения отображением $e : \mathrm{W} \to \mathrm{D}'$, которое для состояния w дает значение данного выражения в этом состоянии. Таким образом, программный текст e выражения принят за имя этого отображения в денотационной семантике, а $e(w)$ обозначает значение выражения e в состоянии w.

Замечание. Корректнее было бы дать функции, определяющей значение любого выражения e в любом состоянии w, единое имя, например — $\mathrm{Value} : \mathrm{Expr} \times \mathrm{W} \to Val$. Можно считать, что мы так и поступили, и рассматривать запись $e(w)$ как удобное сокращение от $\mathrm{Value}(e, w)$.

Надо определить семантику выражений модельного языка — правила определения значений $e(w)$.

Введем обозначение $\mathrm{OpSym} = \{+, \ldots, <, \ldots, \wedge, \ldots\}$ для множества (синтаксического класса) имеющихся в языке знаков двухместных операций, Ops — для множества самих операций, точнее, их реализаций на рассматриваемой абстрактной машине. В обобщенной записи выражений (т. е. в левых частях определений функций e) элементы множества OpSym будем обычно заменять символом op, а в описании семантики выражения (в правых частях тех же определений) соответствующую операцию из множества Ops обозначим символом Op.

Замечание. Множество Expr состоит из фрагментов текста программы, которые мы не станем заключать в какие-либо кавычки, помня, что, например, имя функции e в записи $e(w)$ — это всегда текст. Некоторыми деталями синтаксиса, например скобками, выделяющими из выражения его подвыражения, будем пренебрегать, как и соглашениями о старшинстве операций, позволяющими в ряде случаев обходиться без этих скобок.

Идентичность двух текстов обозначим символом $\approx$, так что формула $e \approx v$ означает, что e — это не произвольное выражение, а переменная с обозначением v. Аналогично, формула $e \approx e1\ op\ e2$ означает, что выражение e представляет собой текст, состоящий из выражения $e1$, знака операции op и выражения $e2$, выписанных подряд. Символ $\not\approx$ обозначает несовпадение (существенное) текстов, которые он связывает.

Замечание. Несовпадение, к примеру, текстов $n-1+1$ и n следует считать несущественным. Обобщая этот пример, мы должны признать совпадающими арифметические (логические) выражения $e1$ и $e2$ всякий раз, когда формула $e1 = e2$ ($e1 \equiv e2$) выводима в формальной арифметике.

Для любого состояния w принимаем, что $e(w) = d?$ в следующих неблагоприятных ситуациях:

$$
\begin{aligned}
&e \approx v \ \wedge\ v \notin \mathrm{Dom}(w),\\
&e \approx e1\ op\ e2 \ \wedge\ (e1(w) = d? \vee e2(w) = d?),\\
&e \approx \neg\, b \ \wedge\ b(w) = d?,\\
&e \approx \textbf{if } b \textbf{ then } e1 \textbf{ else } e2 \textbf{ fi}\\
&\quad \wedge (b(w) = d? \ \vee\ b(w) \ \wedge\ e1(w) = d? \ \vee \neg\, b(w) \ \wedge\ e2(w) = d?).
\end{aligned}
$$

В остальных же (нормальных) случаях:

$$
\begin{aligned}
&d(w) = d, \text{ (в частности, } \mathit{true}(w) = \mathit{true}, \mathit{false}(w) = \mathit{false}),\\
&v(w) = w(v),\\
&(e1\ op\ e2)(w) = e1(w)\ Op\ e2(w),\\
&(\neg\, b)(w) = \neg\, b(w).
\end{aligned}
$$

Замечание. Здесь, следуя принятому курсу, мы проигнорировали проблему синтаксического анализа. Имя e в записи $e(w)$ — это текст выражения. Надо определить, какого типа этот текст (для начала — арифметическое ли это выражение или логическое), правильно ли он построен, выделить его компоненты (например, $e1$ и $e2$ в случае, когда e — это $e1 + e2$) для передачи их в качестве имен функций, входящих в правую часть соответствующего определения. Для этого нужны другие, синтаксические, функции. Выше их использование подразумевалось, но явно не обозначалось. Так же будем поступать и далее, чтобы не

затемнять нашу основную задачу — описание семантики программ — важными, но прямо не относящимися к ней деталями.

И, наконец, значение условного выражения:

$$(\textbf{if } b \textbf{ then } e1 \textbf{ else } e2 \textbf{ fi})(w) = d \equiv (b(w) \wedge d = e1(w) \vee \neg\, b(w) \wedge d = e2(w)).$$

Иначе, если не стесняться перемежать формальные выражения словами естественного языка:

$$(\textbf{if } b \textbf{ then } e1 \textbf{ else } e2 \textbf{ fi})(w) = \begin{cases} e1(w), & \text{если } b(w), \text{ т. е. если } b(w) = \mathit{true}, \\ e2(w), & \text{если } \neg\, b(w), \text{ т. е. если } b(w) = \mathit{false}. \end{cases}$$

Определение семантики выражений мы начали с констант d и переменных v — выражений, не содержащих внутри себя конструкций того же класса. Затем семантику сложных выражений мы определили через семантику их явных компонент — выражений, текстуально входящих в рассматриваемое выражение. Иначе говоря, для них дается не столько сама семантика, сколько схема или правило ее построения. Процесс построения семантики от вложенных конструкций к объемлющим на основе подобных правил назовем ее *регулярным расширением*. Построение семантик — это и суть те дополнительные действия по установлению свойств алгоритмов, о которых говорилось в разд. 1.7.5 на с. 137.

Л е м м а 3.2.1. *Значение $e(w)$ выражения e определено однозначно* (*т. е. e — детерминированное отображение*).

Доказательство (индукция по структуре выражения). Первое определение из этой группы однозначно дает значения констант (элементов множества $\mathrm{Z} \cup \mathrm{BV}$), второе, также однозначно, — значения переменных в текущем состоянии.

В остальных случаях выражение содержит собственные вхождения других выражений (его подвыражений). Пусть уже доказано, что значения подвыражений определены однозначно. Тогда значения выражений вида $e1$ *op* $e2$ или $\neg\, b$ также определены однозначно, поскольку все двухместные операции Op и одноместная операция '$\neg$' детерминированы. Если же выражение имеет вид **if** b **then** $e1$ **else** $e2$ **fi**, то значение b однозначно определяет выбор между $e1$ и $e2$, а значение выбранного выражения — значение всего выражения. Соглашения о значении $e(w)$ в особых ситуациях и о смысле обозначения $d?$ делают функции e формально однозначно определенными и в этих ситуациях. Поскольку состояние w произвольно, то лемма доказана. $\lhd$

3.2.5. Преобразователи состояний для операторов

1. Пустому оператору $S \approx$ **skip** соответствует тождественное преобразование I на множестве состояний:

$$\mathrm{Sem}(\textbf{skip}) = I \quad \text{или} \quad \forall\, w\, (s(w) = w). \tag{3}$$

2. Для невыполнимого оператора $S \approx$ **abort** преобразователь состояний — пустое соответствие:

$$\mathrm{Sem}(\textbf{abort}) = \varnothing \quad \text{или} \quad \forall\, w \forall\, w_1\, (s(w) \neq w_1). \tag{4}$$

В дальнейшем нам часто придется описывать преобразования функций. Введем в этих целях следующее обозначение. Пусть $f : X \to Y$ — некоторая функция, $x \in X$, $y \in Y$. Обозначим $f\{y/x\}$ функцию, получающуюся из f при изменении ее значения в точке x на y:

$$f\{y/x\}(x) = y, \tag{5}$$

$$f\{y/x\}(x1) = f(x1) \qquad \text{при} \quad x1 \in \mathrm{Dom}(f), \text{ но } x1 \neq x. \tag{6}$$

Здесь $x1 \neq x$ означает $\neg(x1 = x)$, где '=' обозначает отношение равенства, принятое на множестве X. Случай $x \notin \mathrm{Dom}(f)$ не исключается, при этом $\mathrm{Dom}(f\{y/x\}) = \mathrm{Dom}(f) \cup \{x\}$.

3. При определении денотационной семантики оператора $S \approx v := e$ только что введенное преобразование будет применено к функции w. Состояние $w\{d/v\}$ — это функция (как и состояние w), возникающая из w в результате присваивания значения d переменной v. В данном случае, поскольку переменные — это тексты, под $v1 \neq v$ следует понимать $v1 \not\approx v$. Формулы (5) и (6) однозначно определяют новое состояние.

Таким образом:

$$\mathrm{Sem}(v := e)(w) = w_1 \equiv \exists\, d\,(d = e(w) \ \wedge \ d \neq d? \ \wedge \ w_1 = w\{d/v\}). \tag{7}$$

В случае $e(w) = d?$ значение $s(w)$ не определено.

Л е м м а 3.2.2. *Семантика оператора присваивания* $Sem(v := e)$ *определена однозначно.*

Доказательство. Значение выражения e в состоянии w и операция $w\{d/v\}$ определены однозначно. $\lhd$

4. Оператор вывода $S \approx \mathit{Write}(e_1, \ldots, e_m)$ передает в той или иной форме значения выражений $e_1, \ldots, e_m$ во внешнюю среду — см. разд. 3.2.1, но не меняет состояние программы, так что

$$\mathrm{Sem}(\mathit{Write}(e_1, \ldots, e_m)) = \mathrm{I},$$

как и для оператора **skip**. Если, однако, ввести в рассмотрение абстрактный аналог внешних запоминающих устройств на дисках, лентах и т. п., то можно было бы описать содержательную сторону операций вывода на такие устройства (см. разд. 2.4.3) и формально определить их денотационную семантику. Здесь многое напоминало бы семантику обычного присваивания — посылки данных на хранение в оперативную память машины. Но развивать эту тему не станем.

5. Подобно содержательной, денотационная семантика оператора ввода $S \approx \mathit{Read}(v_1, \ldots, v_n)$ не может быть описана полностью, поскольку новые значения переменных $v_1, \ldots, v_n$ зависят от состояния внешней среды, а не самой программы или машины, на которой она исполняется. Случай ввода этих значений из устройств внешней памяти мог бы быть формализован наряду с выводом на такие устройства, но только что было обещано этого не делать.

В духе сказанного в разд. 3.2.2 переменным $v_1, \ldots, v_n$ целесообразно сопоставить константы $v0_1, \ldots, v0_n$, получающие при исполнении оператора ввода те же значения, но впоследствии их не меняющие (до следующего ввода). Константы $v0_i$ не могут появляться в основном тексте программы. С их помощью в пред-

и постусловиях или в часто играющих их роль примечаниях к программе можно описывать (см. тот же раздел) свойства программы, связывающие результаты ее исполнения с введенными извне значениями.

Появляется также возможность описать семантику оператора ввода в виде

$$\mathrm{Sem}(Read(v_1, \ldots, v_n))(w) = w\{v0_1/v_1\} \ldots \{v0_n/v_n\},$$

где в результирующем состоянии фиксируется связь значений переменных v_i непосредственно после исполнения оператора с константами $v0_i$.

Перейдем к структурированным операторам различных типов. Они содержат внутри себя другие операторы. Наша задача — выразить преобразователи состояний для структурированных операторов через преобразователи для их внутренних операторов.

В отличие от предыдущих случаев, где была дана семантика конкретных операторов, здесь еще острее требуется именно схема построения семантики. Для структурированного оператора семантика может быть построена по такой схеме только после построения семантики его внутренних операторов.

Эта ситуация типична. В предыдущем разделе мы уже назвали процесс построения семантики некоторой структурированной программной конструкции, исходя из семантики входящих в нее неструктурированных фрагментов, регулярным расширением семантики. Если там расширялась совокупность функций $e(w)$ (подразумеваемая функция $\mathrm{Value}(e, w)$), то сейчас — функция Sem.

6. Семантика составного оператора $S \approx S1; S2$ такова. Пусть w — текущее состояние, тогда

$$\mathrm{Sem}(S1; S2)(w) = \mathrm{Sem}(S2)(\mathrm{Sem}(S1)(w)) \tag{8}$$

или короче:

$$s = s2 \cdot s1\,. \tag{9}$$

У формулы (9) есть очевидное следствие: если оператор S имеет вид $S_1; S_2; \ldots; S_n$, то

$$s = s_n \cdot \ldots \cdot s_2 \cdot s1. \tag{10}$$

Лемма 3.2.3. *Если функции Sem(S1) и Sem(S2) детерминированы, то и функция Sem(S1; S2) детерминирована.*

Утверждение очевидно. $\lhd$

Прежде чем начать рассматривать условные операторы и циклы, введем понятие композиции pS условия (логического выражения) p и оператора S. Содержательно это означает, что оператор S можно исполнить только из состояния, удовлетворяющего условию p, в противном случае исполнение композиции не определено. Формально, композиция — это преобразователь состояний $\mathrm{Sem}(pS)$ или, кратко, ps, который задается формулой

$$\mathrm{Sem}(pS)(w) = w_1 \;\equiv\; p(w) \wedge w_1 = \mathrm{Sem}(S)(w).$$

Для случая, когда p — одна из характеристических функций b^+ или b^-, которые выше, в самом конце разд. 3.2.3, были сопоставлены логическому выражению b, используется обозначение $\mathrm{Sem}(bS)$ или bs вместо $\mathrm{Sem}(b^+S)$, и $\mathrm{Sem}(\overline{b}S)$ или $\overline{b}s$ вместо $\mathrm{Sem}(b^-S)$.

Нам понадобится также композиция Sp оператора S и условия p (именно в таком порядке, в отличие от pS). Здесь условие p — это ограничение, наложенное на состояние, возникшее после исполнения оператора S. Преобразователь состояний $\mathrm{Sem}(Sp)$, а короче — sp, соответствующий такой композиции, задается формулой

$$\mathrm{Sem}(Sp)(w) = w_1 \;\equiv\; w_1 = \mathrm{Sem}(S)(w) \wedge p(w_1).$$

Опять-таки вместо $\mathrm{Sem}(Sb^+)$ будем писать $\mathrm{Sem}(Sb)$ или sb, а вместо $\mathrm{Sem}(Sb^-)$ — $\mathrm{Sem}(S\overline{b})$ или $s\overline{b}$.

Теперь перейдем к оставшимся типам структурированных операторов.

7. Для условного оператора $S \approx$ **if** b **then** $S1$ **else** $S2$ **fi**, где b — логическое выражение, $S1$ и $S2$ — операторы, преобразователь состояний может быть задан формулой

$$s = bs1 \cup \overline{b}s2, \tag{11}$$

где $bs1$ и $\overline{b}s2$ — композиции, определенные немного выше. Подробнее:

$$s(w) = w_1 \;\equiv\; b^+(w) \wedge\; w_1 = s1(w) \vee b^-(w) \wedge\; w_1 = s2(w).$$

Действительно, если начальное состояние w удовлетворяет предикату b^+, то результирующее состояние w_1 вырабатывается оператором $S1$, а если w удовлетворяет b^-, то оператором $S2$. Если же $b(w)$ не определено, то никакое состояние w_1 не может быть выработано, так как в этом случае обе функции b^+ и b^- принимают значение *false*.

Л е м м а 3.2.4. *Если семантика операторов $S1$ и $S2$ определена однозначно, то семантика оператора* ***if*** b ***then*** $S1$ ***else*** $S2$ ***fi*** *также определяется однозначно.*

Доказательство. Уже доказано (разд. 3.2.4), что значение выражения b определено однозначно. Тем самым в текущем состоянии определен выбор между операторами $S1$ и $S2$. Семантика выбранного оператора детерминирована по условию. Остальное очевидно. $\lhd$

8. Займемся оператором цикла $S \approx$ **while** b **do** $S1$ **od**, где b — логическое выражение, $S1$ — оператор с семантикой $s1$.

Назовем числом повторений цикла число исполнений оператора $S1$. Таким образом, если оператор цикла завершается после n повторений, то условие b проверяется (значение анализатора b вычисляется) $n+1$ раз, причем первые n раз вырабатывается значение *true*, а в $(n+1)$-й раз — значение *false*.

Пусть $f : X \to X$ — произвольное соответствие. Как обычно, через f^0 обозначим тождественное соответствие I на множестве X, т. е. $f^0(x) = x$ для любого $x \in X$, а через f^n — соответствие $f \cdot f \cdot \ldots \cdot f$ (n раз), иначе говоря, $f^0 = \mathrm{I}$ и для любого $n \geqslant 0$ и любых x, y из X

$$f^{n+1}(x) = y \;\equiv\; y = f(f^n(x)).$$

Если цикл начался в состоянии $w = w_0$, то состояние w_1 после первого повторения удовлетворяет условию $w_1 = bs1(w)$, подробнее: $b^+(w)$ (иначе и одного повторения не потребовалось бы) и $w_1 = s1(w)$, что очевидно. Состояние после n повторений удовлетворяет условию $w_n = ((bs1)^n)(w)$ (доказывается по

индукции). Если цикл завершился после n повторений, то заключительное состояние w_1 удовлетворяет только что написанному условию, а также условию завершения $b^-(w_1)$. Все вместе:

$$w_1 = (((bs1)^n)\overline{b})(w).$$

Будем обходиться без части скобок:

$$w_1 = (bs1^n\overline{b})(w).$$

Оператор цикла завершается, если требуемое число повторений конечно:

$$s(w) = w_1 \;\equiv\; \exists\, n\,(n \geqslant 0 \wedge w_1 = (bs1^n\overline{b})(w)). \tag{12}$$

В более краткой записи:

$$s = \bigcup_{n\geqslant 0} bs1^n\overline{b}. \tag{13}$$

Замечания. 1. Переход от семантики $bs1^n$ к семантике $bs1^{n+1}$ — это необходимое в общем случае (см. разд. 1.7.5) прослеживание свойств алгоритма, выполняемое на каждом витке цикла. Чтобы избавиться от такого прослеживания, надо суметь свернуть нефинитное (из-за присутствия $\bigcup_{n\geqslant 0}$) выражение (13) во что-нибудь более компактное — математика располагает богатым набором понятий и изобразительных средств для такой свертки.

2. Регулярное расширение связано с еще одной трудностью — если оператор $S1$ сам является циклом или содержит циклы внутри себя, то семантика $s1$ не может быть построена из-за отсутствия соответствующего правила. Поэтому построение семантики оператора цикла надо начинать с самых внутренних циклов, а справедливость формулы (13) доказывать индукцией по глубине вложенности циклов.

Этот прием построения и обоснования семантических правил (правил расширения функции Sem) следует использовать всякий раз, когда приходится иметь дело с операторами новых видов, явно или неявно содержащими внутри себя операторы цикла или операторы этого нового вида. Будем в дальнейшем пользоваться этим методом, специально этого не оговаривая.

Из формулы (13) вытекает

Следствие 3.2.5. *Оператор* $S \approx$ ***while*** b ***do*** $S1$ ***od*** *эквивалентен оператору* S', *имеющему вид* ***if*** b ***then*** $S1$; S ***else skip fi***.

Доказательство. Найдем преобразователь состояний s' для оператора S'. Последовательно получаем:

$$\begin{aligned} s' &= b(s\cdot s1)\cup\overline{b}\mathrm{I} = b\Big(\big(\textstyle\bigcup_{n\geqslant 0}(bs1^n\overline{b})\big)\cdot s1\Big)\cup \mathrm{I}\overline{b} \\ &= \big(\textstyle\bigcup_{n\geqslant 0}(bs1^n\overline{b})\cdot bs1\big)\cup bs1^0\overline{b} = \textstyle\bigcup_{n>0}(bs1^n\overline{b})\cup bs1^0\overline{b} \\ &= \textstyle\bigcup_{n\geqslant 0}(bs1^n\overline{b}) \;=\; s. \end{aligned}$$

Здесь использованы формулы (11), (8), (3), (13), а также

$$b(f\cdot g) = f\cdot bg, \tag{14}$$

$$f\overline{b}\cdot g = (f\cdot g)\overline{b}, \tag{15}$$

$$\Big(\bigcup_{n\geqslant 0} f_n\Big)\cdot g = \bigcup_{n\geqslant 0}(f_n\cdot g) \tag{16}$$

и другие, еще более очевидные, равенства. $\lhd$

Упражнение 3.2.1. Доказать истинность равенств (14) и (16).

Л е м м а 3.2.6. *Если семантика оператора $S1$ определена однозначно, то и семантика оператора $S \approx$ **while** b **do** $S1$ **od** определяется однозначно.*

Доказательство (индукция по n). Состояние w_0 определено однозначно. Пусть $n \geqslant 0$ и для всех $i \leqslant n$ доказано, что состояние w_i определено однозначно и при $i < n$ значение $b(w_i)$ (определяемое однозначно) равно *true*. Тогда если $b(w_n) =$ **false**, то состояние w_n — заключительное. Если же $b(w_n) =$ **true**, то однозначно определится состояние w_{n+1} и уже для $i \leqslant n+1$ будет истинно сформулированное выше утверждение. Если для всех $n \geqslant 0$ оказывается, что $b(w_n) = true$, то условие $\exists\, n\,(n \geqslant 0 \wedge (w_1 = (bs1^n\overline{b})(w)))$ не выполняется и состояние $\mathrm{Sem}(S)(w)$ не определено. $\lhd$

С практической точки зрения дефект доказательства состоит в том, что завершаемость цикла после конечного числа повторений этим способом не устанавливается. В отдельных случаях она может быть установлена какими-то иными, нестандартными, средствами.

9. Семантика оператора объявления **loc** v, если его рассматривать изолированно от контекста, дается формулой

$$\mathrm{Sem}(\mathbf{loc}\, v)(w) = w\{d?/v\},$$

где вновь использовано введенное выше обозначение $f\{y/x\}$. Эта формула говорит, что переменная v получает неопределенное значение независимо от того, была ли она (а точнее — переменная с таким же идентификатором) объявлена вне текущего блока или объявляется впервые. Во втором случае область определения $\mathrm{Dom}(w)$ состояния w автоматически расширяется.

Однако при описании семантики блока S, имеющего вид

begin loc v_1; ... ; **loc** v_n; $S1$ **end**

где $S1$ — оператор (тело блока), уже не содержащий объявлений (не входящих во внутренние блоки), необходимо формировать и учитывать контекст, включающий в себя множество *dvars* переменных, объявленных в блоке, и состояние w_0, каким оно было на момент входа в блок. Без этого невозможно отразить описанную в разд. 3.2.1 содержательную семантику блока. При обработке объявления **loc** v_i надо также следить, чтобы переменная v_i не входила в накопленное на этот момент множество *dvars*.

Определим поэтому рекурсивные функции SemIn и SemOut, которые описывают преобразование состояний при входе в блок и при выходе из него для одной из переменных, объявленных в блоке:

$$\begin{aligned}
&\mathrm{SemIn}(\mathbf{loc}\ v; S',\ dvars,\ w_0)(w)\\
&\quad = \mathrm{SemOut}(v,\ w_0)(\mathrm{SemIn}(S', dvars \cup \{v\}, w_0)(w\{d?/v\})), \quad \text{если } v \notin dvars,\\
&\forall\, w'\, \mathrm{SemIn}(\mathbf{loc}\ v;\ S',\ dvars,\ w_0)(w) \neq w', \quad \text{если } v \in dvars,\\
&\mathrm{SemIn}(S1,\ dvars,\ w_0)(w) = \mathrm{Sem}(S1)(w),\\
&\qquad \text{если } S1 \not\approx \mathbf{loc}\, v;\ S' \text{ при любом операторе } S'.
\end{aligned}$$

В этом описании состояние преобразуется так же, как было сказано в начале пункта. Результат передается в качестве аргумента преобразователю, создава-

емому той же функцией SemIn, но для оператора, в котором объявление **loc** v отброшено. Результат работы этого преобразователя становится аргументом преобразователя, создаваемого функцией SemOut:

$$\mathrm{SemOut}(v, w_0)(w) = \begin{cases} w\{w_0(v)/v\}, & \text{если } v \in \mathrm{Dom}(w_0), \\ w \setminus \langle v, w(v)\rangle, & \text{если } v \notin \mathrm{Dom}(w_0). \end{cases}$$

Функция SemOut в полном соответствии с содержательной семантикой действий по закрытию блока либо восстанавливает то значение переменной v, которым она обладала в состоянии w_0 перед входом в блок (как бы отображая присваивание $v := w_0(v)$), либо исключает пару ⟨переменная, значение⟩ из текущего состояния, постепенно восстанавливая область определения состояния на момент входа в блок.

Благодаря передаче аргумента w_0 от одного рекурсивного обращения к функции SemIn другому, а также функции SemOut, обеспечивается сохранение информации о состоянии программы на момент входа в блок. О необходимости этого говорилось в начале разд. 3.2.3.

Теперь полная семантика блока описывается уже просто:

$$\mathrm{Sem}(\textbf{begin } S \textbf{ end})(w_0) = \mathrm{SemIn}(S, \varnothing, w_0)(w_0).$$

Следует также уточнить смысл обозначения $d?$ — под ним можно понимать либо фиксированное, но не входящее в множество $\mathrm{Z} \cup \mathrm{BV}$ значение, либо произвольное (случайно выбираемое) значение из $\mathrm{Z} \cup \mathrm{BV}$. Обе трактовки имеют право на жизнь, так как в реальных машинах может существовать код, обозначающий неопределенное значение, но чаще такого кода не существует и в ячейке, сопоставленной переменной v, может находиться что угодно, в лучшем случае — весьма редко используемое значение из Z.

При первой трактовке справедлива

Лемма 3.2.7. *Семантика оператора* ***loc****v определена однозначно.*

Утверждение очевидно, так как ограничение на неповторяемость объявлений в блоке соблюдается. ◁

При второй трактовке обозначения $d?$ последствия исполнения объявления, напротив, крайне неоднозначны. В оставшейся части раздела будем следовать первой трактовке.

Лемма 3.2.8. *Если семантика тела блока $S1$ детерминирована, то детерминирована и семантика блока в целом. В результате исполнения блока область определения текущего состояния* (*состояния на момент входа в блок*) *не изменяется.*

Доказательство. Лемма является следствием предыдущих лемм и однозначного определения операции $f\{y/x\}$ (в данном случае — $w\{d?/v\}$) в разд. 3.2.5. Инвариантность области определения состояния обеспечивается преобразователем SemOut. ◁

Программа в целом — это блок. Его исполнение начинается из пустого состояния, где ни одна переменная (кроме, может быть, стандартных, определенных в самом языке) не объявлена.

Теорема 3.2.9. *Если программа не содержит операторов и выражений никакого другого вида, кроме рассмотренных в этом разделе выше, а оператор ввода используется лишь один раз и только для ввода начальных значений некоторых переменных, то результат ее исполнения с момента завершения ввода определен однозначно.*

Доказательство. Утверждение теоремы — следствие доказанных выше лемм о свойствах семантики отдельных операторов и выражений. $\lhd$

3.2.6. Преобразователи предикатов

Каждому соответствию $R : X \to X$ можно сопоставить индуцированные отображения: $R{\uparrow}\colon 2^X \to 2^X$ — полный образ подмножества P множества X при соответствии R и $R{\downarrow}\colon 2^X \to 2^X$ — полный прообраз подмножества Q множества X при том же соответствии. Они задаются формулами:

$$R{\uparrow}(P) = \{\, R(w) \mid w \in P \,\}, \qquad \text{для любого } P \subseteq X,$$
$$R{\downarrow}(Q) = \{\, w \mid R(w) \in Q \,\}, \qquad \text{для любого } Q \subseteq X.$$

Пусть X — это множество W состояний программы, а R — преобразователь состояний $\mathrm{Sem}(S)$ для некоторого оператора S.

Для любой реальной машины множество W практически бесконечно, так что множество 2^{W} практически несчетно. (Понимание термина «практически» — дело интуиции как автора, так и читателя.) Поэтому среди множеств из 2^{W} те, которые могут быть описаны обозримыми логическими формулами (будем далее называть их предикатами), составляют ничтожное меньшинство. Однако прагматические (см. начало разд. 3.2) соображения заставляют нас ограничиться этим меньшинством. С предикатами мы уже встречались в разд. 3.1.2 и 3.2.2.

В терминах предикатов отображения $\mathrm{Sem}(S){\uparrow}$ и $\mathrm{Sem}(S){\downarrow}$ описываются так:

$$\mathrm{Sem}(S){\uparrow}(p)(w_1) \;\equiv\; \exists\, w\, (p(w) \wedge w_1 = \mathrm{Sem}(S)(w)), \tag{17}$$
$$\mathrm{Sem}(S){\downarrow}(q)(w) \;\equiv\; \exists\, w_1\, (q(w_1) \wedge w_1 = \mathrm{Sem}(S)(w)). \tag{18}$$

Здесь $\mathrm{Sem}(S){\Uparrow}(p)$ и $\mathrm{Sem}(S){\Downarrow}(q)$, так же как p и q, представляют собой логические формулы (тексты). Их назначение — выделять из множества W интересующие нас подмножества. Всюду определенные на множестве предикатов отображения $\mathrm{Sem}(S){\uparrow}$ и $\mathrm{Sem}(S){\downarrow}$ будем называть *прямым* и *обратным преобразователями предикатов* (для оператора S, если $\mathrm{Sem}(S)$ — преобразователь состояний для этого оператора).

Как правило, будем писать $s\uparrow$ вместо $\mathrm{Sem}(S){\uparrow}$ и $s\downarrow$ вместо $\mathrm{Sem}(S){\downarrow}$ (аналогично для $S1$, S' и т. д.):

$$s{\uparrow}(p)(w_1) \;\equiv\; \exists\, w\, (p(w) \wedge w_1 = s(w)), \tag{19}$$
$$s{\downarrow}(q)(w) \;\equiv\; \exists\, w_1\, (q(w_1) \wedge w_1 = s(w)). \tag{20}$$

Используя введенные обозначения, слабую операторную формулу $p\ \{S\}\ q$ можно переписать в виде

$$\forall\, w_1\, (s{\uparrow}(p)(w_1) \supset q(w_1)). \tag{21}$$

Действительно, пусть формула $s\uparrow(p)(w_1)$ истинна. Согласно формуле (19) это значит, что найдется состояние w, такое, что истинно утверждение $p(w)$ и $w_1 = s(w)$. Но тогда из формулы (3.2.2–1) вытекает истинность формулы $q(w_1)$.

Формула (21) означает, что $s\uparrow(p)$ — это наиболее сильный предикат, который можно использовать в качестве q в формуле $p\ \{S\}\ q$.

Можно считать, что использование слабых формул предполагает получение свойств программы в направлении от предусловия p к постусловию q, от предыдущего состояния w — к последующему w_1, как для каждого оператора, так и для программы в целом.

Для сильных операторных формул вида $p\,[S]\,q$ потребуем, чтобы оператор S был детерминирован. Пусть при этом условии такая формула справедлива. Это значит, что для любого состояния w, в котором истинна формула p, однозначно определено состояние $w_1 = s(w)$ и в этом состоянии обязана быть истинной формула q. В силу единственности состояния w_1 правая часть определения (3.2.2–2) может быть переписана в виде

$$\forall\, w\,(p(w) \supset \exists\, w_1\,(w_1 = s(w) \wedge q(w_1))),$$

после чего ее, на основании формулы (20), можно заменить формулой

$$\forall\, w\,(p(w) \supset s{\downarrow}(q)(w)). \qquad (22)$$

Так измененная формула (3.2.2–2) означает, что $s\downarrow(q)$ — это наиболее слабый предикат, который может стоять в качестве p в формуле $p\,[S]\,q$ (для детерминированного оператора S).

Таким образом, сильные формулы как бы распространяют свойства программы и ее отдельных операторов в направлении от постусловия q к предусловию p, от последующего состояния w_1 к предыдущему w.

Можно считать также, что формулы (19) и (20) конкретизируют соответствия Week и Strong из разд. 3.2.2, превращая их в отображения $s\uparrow(p)$ и $s\downarrow(q)$.

Для недетерминированных алгоритмов введенные понятия и обозначения не дают возможности записывать утверждения относительно исхода вычислений по всем возможным путям исполнения алгоритма. Поэтому для таких алгоритмов сильная формула не имеет эквивалента в этих обозначениях.

Поясним это на примере моделирования игры, элементом которой является бросание монеты. В программе монета заменяется датчиком случайных чисел. От значения, выработанного датчиком, зависит дальнейший ход и, в конечном счете, результат игры, например, «Выиграл игрок» или «Выиграла машина».

Если изучать свойства этой программы, пользуясь слабыми формулами, то, каково бы ни было предусловие для всей программы, скорее всего, удастся доказать только такое свойство заключительного состояния:

Выиграл игрок $\vee$ Выиграла машина.

Такой результат может показаться мало содержательным, но ведь к нему мог бы добавиться еще какой-нибудь дизъюнктивный член (скажем, «Год рождения бабушки игрока = 1900»).

Что может получиться, если попытаться исследовать свойства этой программы, исходя из постусловия «Выиграл игрок» и пользуясь сильными формулами? Ответ на этот вопрос зависит от того, что в точности понимается под недетерминированным алгоритмом, в частности — под моделью игры, и под сильной формулой. Если последняя гарантирует завершаемость алгоритма и истинность постусловия в одном из возможных завершающих состояний, то весьма

вероятно, что на состояние, предшествующее бросанию монеты, будет наложено какое-либо слабое предусловие (необходимое, но не достаточное для выигрыша), может быть, просто *true*. Оно говорит об исходе игры лишь то, что для игрока еще не все потеряно. Ведь в силу своей недетерминированности программа может, если игроку повезет, завершить работу в состоянии, где он выиграет, ну а не повезет — пора задуматься, стоило ли браться за игру.

Если же сильная формула означает в соответствии с формулой (3.2.2–2), что алгоритм обязательно завершает работу и постусловие истинно в любом из возможных заключительных состояний, иначе говоря — что предусловие достаточно для достижения выигрыша, то таким предусловием может быть, скорее всего, только *false* — в силу все той же недетерминированности.

Предсказать результат игры можно было бы только для алгоритма, реализующего выигрывающую стратегию для одного из участников. В рассмотренном примере такой стратегии быть просто не может (если не пользоваться монетой с орлом на обеих сторонах).

3.2.7. Преобразователи предикатов для операторов

Исходя из выражений, полученных в разд. 3.2.5, построим соответствующие преобразователи предикатов. В настоящем разделе всюду подразумевается, что p обозначает предусловие, а q — постусловие для оператора S. Кванторы всеобщности по этим буквам также явно не выписаны.

1. Легко видеть, что преобразователи предикатов для оператора $S \approx$ **skip** тождественны:

$$s{\uparrow}(p) \;\approx\; p, \tag{23}$$
$$s{\downarrow}(q) \;\approx\; q. \tag{24}$$

2. Преобразователи предикатов для оператора $S \approx$ **abort**, как всегда, всюду определены, но сопоставляют каждому предикату из 2^{W} тождественно ложный предикат:

$$s{\uparrow}(p) \;\approx\; \mathit{false}, \tag{25}$$
$$s{\downarrow}(q) \;\approx\; \mathit{false}. \tag{26}$$

Формула (25) блокирует исполнение любого оператора вслед за оператором **abort**, а формула (26) — и самого этого оператора.

Добавим к этому, что если оператор $S0$ не исполни́м в силу любых причин, явно упомянутых или даже вовсе не упомянутых выше, то и для него верны аналогичные формулы: $\mathrm{Sem}(S0){\uparrow}(p) \approx \mathit{false}$, $\mathrm{Sem}(S0){\downarrow}(q) \approx \mathit{false}$. Это позволяет не делать в дальнейшем никаких оговорок о виде преобразователей предикатов в случаях невозможности исполнить тот или иной конкретный оператор.

3. Чтобы найти прямой преобразователь предикатов для составного оператора $S \approx S1;\ S2$, выполним преобразования:

$$\begin{aligned} s{\uparrow}(p)(w_1) \;&\Leftrightarrow \exists\, w\,(p(w) \wedge \exists\, w'\,(w' = s1(w) \wedge w_1 = s2(w'))) \\ &\Leftrightarrow \exists\, w'\,(\exists\, w\,(p(w) \;\wedge\; w' = s1(w)) \;\wedge\; w_1 = s2(w')) \\ &\Leftrightarrow \exists\, w'\,(s1{\uparrow}(p)(w') \;\wedge\; w_1 = s2(w')) \Leftrightarrow s2{\uparrow}(s1{\uparrow}(p))(w_1). \end{aligned}$$

Итак,

$$s{\uparrow}(p) \;\approx\; s2{\uparrow}(s1{\uparrow}(p)), \qquad \text{или} \qquad s{\uparrow} = s2{\uparrow}\cdot\, s1{\uparrow}. \tag{27}$$

Так же находим обратный преобразователь предикатов (при детерминированном операторе S):

$$s{\downarrow}(q) \;\approx\; s1{\downarrow}(s2{\downarrow}(q)), \qquad \text{или} \qquad s{\downarrow} = s1{\downarrow}\cdot\, s2{\downarrow}, \tag{28}$$

поскольку обратные преобразователи строятся справа налево.

Пользуясь индукцией, можно получить аналоги формул (27) и (28) для составного оператора вида $S_1;\ \ldots\ ;\ S_n$:

$$s = s_n \cdot \ldots \cdot s_1 \supset s{\uparrow} = s_n{\uparrow}\cdot \ldots \cdot\, s_1{\uparrow}, \tag{29}$$
$$s = s_n \cdot \ldots \cdot s_1 \supset s{\downarrow} = s_1{\downarrow}\cdot \ldots \cdot\, s_n{\downarrow}. \tag{30}$$

Сформулируем некоторые промежуточные утверждения. Пусть $s = s1 \cup s2$, в переводе с теоретико-множественного на логический язык:

$$s(w) = w_1 \;\equiv\; w_1 = s1(w) \vee w_1 = s2(w).$$

При этом

$$s{\uparrow}(p) \;\approx\; s1{\uparrow}(p) \vee s2{\uparrow}(p), \tag{31}$$
$$s{\downarrow}(q) \;\approx\; s1{\downarrow}(q) \vee s2{\downarrow}(q). \tag{32}$$

Упражнение 3.2.2. Доказать истинность равенства (31).

Преобразователи предикатов, соответствующие преобразователям состояний ps и sp, определяются формулами:

$$ps{\uparrow}(p') \;\approx\; s{\uparrow}(p \wedge p'), \tag{33}$$
$$ps{\downarrow}(q) \;\approx p \wedge s{\downarrow}(q), \tag{34}$$
$$sp{\uparrow}(p') \;\approx s{\uparrow}(p') \wedge p, \tag{35}$$
$$sp{\downarrow}(q) \;\approx s{\downarrow}(q \wedge p). \tag{36}$$

Упражнение 3.2.3. Доказать истинность равенств (33) и (34).

4. Для условного оператора $S \;\approx$ **if** b **then** $S1$ **else** $S2$ **fi** получаем следующие выражения для преобразователей предикатов:

$$s{\uparrow}(p) \;\approx\; s1{\uparrow}(b^+ \wedge p) \vee s2{\uparrow}(b^- \wedge p), \tag{37}$$
$$s{\downarrow}(q) \;\approx\; b^+ \wedge s1{\downarrow}(q) \vee b^- \wedge s2{\downarrow}(q). \tag{38}$$

Формула (37) следует из формул (31) и (33), а формула (38) — из (32) и (34).

5. Теперь построим предикатные преобразователи для оператора цикла $S \;\approx$ **while** b **do** $S1$ **od**. Обозначим $s_n = (bs1)^n$. К преобразователю s, определяемому формулой (3.2.5–12)

$$s(w) = w_1 \;\equiv\; \exists\, n\,(n \geqslant 0 \wedge w_1 = (s_n\overline{b})(w)),$$

можно применить обобщения формул (31) и (32) на случай дизъюнкции счетного множества формул:

$$s{\uparrow}(p) \;\approx\; \exists\, n\,(n \geqslant 0 \wedge (s_n\overline{b}){\uparrow}(p)), \quad s{\downarrow}(q) \;\approx\; \exists\, n\,(n \geqslant 0 \wedge (s_n\overline{b}){\downarrow}(q)).$$

На основании формул (35) и (36) получаем

$$(s_n\overline{b}){\uparrow}(p) \;\approx\; s_n{\uparrow}(p) \wedge b^-, \quad (s_n\overline{b}){\downarrow}(q) \;\approx\; s_n{\downarrow}(q \wedge b^-).$$

Обозначим $p_n \approx ((bs1){\Uparrow})^n(p)$, $q_n \approx ((bs1){\Downarrow})^n(q \wedge b^-)$ и воспользуемся формулами (29) и (30) для случая, когда каждое $s_i = bs1$:

$$s_n{\uparrow}(p) \;\approx\; p_n, \quad s_n{\downarrow}(q \wedge b^-) \;\approx\; q_n$$

для всех $n \geqslant 0$. Собирая все вместе, получаем

$$s\uparrow(p) \;\approx\; \exists\, n\,(n \geqslant 0 \wedge p_n \;\wedge\; b^-), \tag{39}$$

где $p_0 \;\approx\; p$, а для $n > 0$ — $p_n \;\approx\; (bs1)\uparrow(p_{n-1})$,

$$s\downarrow(q) \;\approx\; \exists\, n\,(n \geqslant 0 \wedge q_n), \tag{40}$$

где $q_0 \;\approx\; q \;\wedge\; b^-$, а для $n > 0$ — $q_n \;\approx\; (bs1)\downarrow(q_{n-1})$.

Введем обозначение $e'[e/v]$ для результата текстовой подстановки выражения e вместо переменной v в целое или логическое выражение e':

1) $d[e/v] \approx d$, где $d \in D$;
2) $(e1\ op\ e2)[e/v] \approx e1[e/v]\ op\ e2[e/v]$,

где op — знак операции ($op \in$ OpSym);

3) (**if** b **then** $e1$ **else** $e2$ **fi**)$[e/v] \approx$ **if** $b[e/v]$ **then** $e1[e/v]$ **else** $e2[e/v]$ **fi**;
4а) $v[e/v] \approx e$;
4б) $v1[e/v] \approx v1$, если $v1 \in$ IV, но $v1 \not\approx v$.

Во всех случаях решение — производить или нет фактическую подстановку выражения e вместо переменной v — либо переносится на уровень подвыражений выражения e', либо принимается статически — без учета значений переменных.

6. В разд. 3.2.5 установлено, что если w_1 — состояние, возникшее после исполнения оператора присваивания $S \approx v := e$ из состояния w, то значение $d = e(w)$ определено и $w_1 = w\{d/v\}$ или, компактнее,

$$s(w) = w\{e(w)/v\}.$$

Л е м м а 3.2.10. *Пусть $v \in$ IV, $e \in$ IE, оператор $S \approx v := e$ исполняется из состояния w и $d = e(w)$. Тогда для произвольного выражения e'*

$$e'[e/v](w) = e'(w\{d/v\}). \tag{41}$$

Доказательство (индукция по структуре выражения e'). Базу индукции составляют случаи, когда e' — константа или простая переменная. Рассмотрим их поочередно, опираясь на формулы (3.2.5–5), (3.2.5–6), определяющие, в частности, функцию $w\{d/v\}$.

а) Если e' — это константа: $e' \approx d1$, $d1 \in$ D, то
$$d1[e/v](w) = d1(w) = d1, \quad d1(w\{d/v\}) = d1.$$

б′) В случае, когда $e' \approx v$, имеем
$$v[e/v](w) = e(w) = d, \quad v(w\{d/v\}) = w\{d/v\}(v) = d.$$

б″) Пусть $e' \approx v'$, где $v' \in$ IV, но $v' \not\approx v$. При этом:
$$v'[e/v](w) = v'(w) = w(v'), \quad v'(w\{d/v\}) = w\{d/v\}(v') = w(v').$$

Во всех трех случаях формула (41) справедлива, так что база индукции установлена.

Пусть теперь выражение e' структурировано, т. е. содержит вхождения других выражений. Предположим, что для всех таких подвыражений формула вида (41) верна.

в) $e' \approx e1\ op\ e2$, где $op \in$ OpSym.

По определению $e'[e/v](w) = e1[e/v](w)\ op\ e2[e/v](w)$ и по предположению индукции

$$e'[e/v](w) = e1(w\{d/v\})\ op\ e2(w\{d/v\}) = e'(w\{d/v\}).$$

г) $e' \approx$ **if** b **then** $e1$ **else** $e2$ **fi**.

При этом:

$$(\textbf{if}\ b\ \textbf{then}\ e1\ \textbf{else}\ e2\ \textbf{fi})\ [e/v](w) = \begin{cases} e1[e/v](w), & \text{если } b[e/v](w), \\ e2[e/v](w), & \text{если } \neg\, b[e/v](w). \end{cases}$$

Но b, $e1$, $e2$ — это подвыражения выражения e'. По предположению индукции для них формула вида (41) справедлива, значит она справедлива и для e'. $\lhd$

Теорема 3.2.11 (Хоор). *Пусть $v \in$ IV, $e \in$ IE, q — предикат на множестве состояний, $q \in$ BE. Тогда для оператора $S \approx v := e$ обратный преобразователь предикатов $s{\downarrow}$ задается выражением*

$$s{\downarrow}(q) = q[e/v]. \tag{42}$$

Доказательство. Используем те же обозначения, что и в лемме 3.2.10, в частности $d = e(w)$. Согласно определению обратного преобразователя предикатов надо доказать эквивалентность (напомним, что детерминированность исполнения оператора присваивания подразумевается)

$$q[e/v](w) \ \equiv\ \exists\, w_1\ (q(w_1)\ \wedge\ w_1 = s(w)).$$

Но $s(w) = w\{d/v\}$, так что надо установить справедливость равенства

$$q[e/v](w) = q(w\{d/v\}).$$

Она следует из леммы 3.2.10. $\lhd$

Найдем вид прямого преобразователя предикатов. Для этого потребуется расширить класс логических выражений. Введем новый класс DV$=\{i, j, \dots\}$ «немых» (связанных) переменных. Условимся, что если логическое выражение q не содержит немой переменной i, то выражение вида $\exists\, i\, q[i/v]$ принадлежит классу BE. Если $v \in$ IV, то для выражения e любого класса выражения $e[i/v]$ могут появляться в качестве подвыражений (того же класса) логического выражения вида $\exists\, i\, q[i/v]$ и только в таком качестве.

Выражению $(\exists\, i\, q)(w)$ придадим обычную содержательную интерпретацию — в состоянии w найдется такое целочисленное значение переменной i, при котором q принимает значение *true*. Соответственно, определение значения выражения в состоянии w дополним правилом

$$(\exists\, i\, e)(w) = \exists\, i\ (e(w)).$$

Операцию $e'[e/v]$ (см. конец п. 5) доопределим так:

5) если $e' \approx \exists\, i\, e1$ и $i, j \in$ DV, $j \not\approx i$, то

а) $e'[e/i] \approx e'$,

б) $(\exists\, j\, e1)[e/i] \approx \exists\, j\ (e1[e/i])$, если j не входит в e,

в) $(\exists\, j\, e1)[e/i] \approx \exists\, j1\ (e1[j1/j][e/i])$, если j входит в e.

В последнем случае $j1 \in$ DV, $j1 \not\approx i, j$, $j1$ не должно входить ни в e, ни в $e1$.

Теорема 3.2.12. *Пусть* $v \in \mathrm{IV}$, $e \in \mathrm{IE}$ *и* $S \approx v := e$. *Тогда прямой преобразователь предикатов* $s\uparrow$ *для оператора* S *ставит в соответствие предикату* p *предикат* $s\uparrow(p)$, *который образуется из* p *по правилу*

$$s\uparrow(p) \approx \exists\, i\, (p[i/v] \ \wedge\ v = e[i/v]). \tag{43}$$

Содержательный смысл этой теоремы изложен в разд. 3.2.2 в связи с аксиомой Флойда А3.

Доказательство основано на простой идее — обозначим $i = v(w)$, тогда оператор $S1 \approx v := i$, выполненный из состояния w_1, восстанавливает состояние w, т. е. $w_1 = s(w) \ \equiv\ w = s1(w_1)$. Поэтому прямой преобразователь предикатов для S эквивалентен обратному преобразователю для $S1$:

$$\begin{aligned} s\uparrow(p)(w_1) &= \exists\, w\, (p(w) \ \wedge\ w_1 = s(w)) \\ &= \exists\, w\, (p(w) \ \wedge\ w = s1(w_1)) = s1\downarrow(p)(w_1). \end{aligned}$$

Важнейшее свойство этого обратного преобразователя дается формулой (41) из леммы 3.2.10. Неизвестное в состоянии w_1 значение i надо связать квантором существования. Из формулы $w_1 = s(w)$ достаточно выделить наиболее существенную ее часть: $v = e[i/v]$.

Проведем это доказательство с необходимыми подробностями. В соответствии с определением (3.2.6-19) прямого преобразователя предикатов $s\uparrow$, утверждением (43) доказываемой теоремы и выражением $s(w) = w\{e(w)/v\}$ для преобразователя состояний надо доказать, что справедлива формула

$$\exists\, i\, (p[i/v] \wedge v = e[i/v])(w_1) = \exists\, w\, (p(w) \wedge w_1 = w\{e(w)/v\}). \tag{44}$$

Левая часть этой формулы согласно правилам интерпретации формул принимает вид

$$\begin{aligned} &(\exists\, i\, (p[i/v] \ \wedge\ v = e[i/v]))(w_1) = \exists\, i\, (p[i/v] \ \wedge\ v = e[i/v])(w_1) \\ &\quad = \exists\, i\, (p[i/v](w_1) \ \wedge\ v(w_1) = e[i/v](w_1)). \end{aligned}$$

На основании леммы 3.2.10 можно утверждать, что

$$p[i/v](w_1) = p(w_2), \quad e[i/v](w_1) = e(w_2),$$

где $w_2 = w_1\{i(w_1)/v\} = w_1\{i/v\}$. Кроме того, $v(w_1) = w_1(v)$. Используя все эти зависимости и вводя состояние w_2 (переименованное в w) и информацию о нем с помощью квантора существования, приводим доказываемую формулу к виду

$$\begin{aligned} &\exists\, i\, \exists\, w\, (w = w_1\{i/v\} \ \wedge\ p(w) \ \wedge\ w_1(v) = e(w)) \\ &\qquad \equiv\ \exists\, w\, (p(w) \ \wedge\ w_1 = w\{e(w)/v\}). \end{aligned} \tag{45}$$

Покажем, что из левой части логически следует правая часть этой формулы. Из равенства $w = w_1\{i/v\}$ вытекает, что $w_1(v1) = w(v1)$ при $v1 \not\approx v$, а тогда из $w_1(v) = e(w)$ получаем $w_1 = w\{e(w)/v\}$. Таким образом,

$$\begin{aligned} &\exists\, i\, \exists\, w\, (w = w_1\{i/v\} \ \wedge\ p(w) \ \wedge\ w_1(v) = e(w)) \\ &\quad \Rightarrow \exists\, i\, \exists\, w\, (p(w) \ \wedge\ w_1 = w\{e(w)/v\}) \Rightarrow \exists\, w\, (p(w) \ \wedge\ w_1 = w\{e(w)/v\}), \end{aligned}$$

поскольку под квантором $\exists\, i$ уже не осталось вхождений переменной i.

Доказываем обратную импликацию. Из $w_1 = w\{e(w)/v\}$ следует, что $w_1(v) = e(w)$, $w_1(v1) = w(v1)$ при $v1 \not\approx v$. Обозначая неизвестное значение $w(v)$ через i, где $i \in \mathrm{DV}$, i не входит ни в p, ни в e, получаем:

$$\exists\, w\,(p(w)\ \wedge\ w_1 = w\{e(w)/v\})$$
$$\Rightarrow \exists\, i\, \exists\, w\,(w = w_1\{i/v\}\ \wedge\ p(w)\ \wedge\ w_1(v) = e(w)).$$

Таким образом, истинность формулы (45) установлена. $\triangleleft$

Все, что было сказано в разд. 3.2.2 по поводу частных случаев аксиомы A3, остается в силе и в связи с формулой (43). А именно, из теоремы 3.2.12 вытекают:

Следствие 3.2.13. *Если уравнение $v = e[i/v]$ при любом v, удовлетворяющем условию $p(v)$, имеет относительно i лишь конечное число k решений $e_1(v), \ldots, e_k(v)$ и k не зависит от v, то формула* (43) *сводится к формуле*

$$s{\uparrow}(p) \approx p[i/v] \wedge (i = e_1(v) \vee \ \ldots\ \vee i = e_k(v)). \tag{46}$$

Следствие 3.2.14. *Если ни формула p, ни выражение e не содержат вхождений переменной v, то справедлива формула*

$$s{\uparrow}(p) \approx p\ \wedge\ v = e. \tag{47}$$

7. В п. 9 разд. 3.2.5 было показано, что семантика SemIn входа в блок сводится к последовательной обработке объявлений, размещающихся в начале блока, включая контроль за отсутствием повторных объявлений одной и той же переменной, и к запоминанию исходного состояния w_0 с тем, чтобы при выходе из блока можно было восстановить значения переобъявленных переменных. Сейчас коснемся лишь преобразования предикатов в связи с исполнением объявлений.

Пусть оператор объявления $S\ \approx\ \mathbf{loc}\ v$ исполняется из состояния w и завершается в состоянии w_1. Исключим беззаконный случай попытки повторно объявить переменную v в том же блоке. Для абстрактной машины пока не видно других причин, вроде ограничений на объем памяти, мешающих завершить исполнение этого оператора. Построим прямой преобразователь предикатов $s{\uparrow}$ для оператора S.

Надо различать два случая. Если переменная с именем v не была объявлена ранее (т. е. во внешнем блоке или блоках), то предусловие p не может ее содержать. Если была и p зависит от v, т. е. содержит свободные вхождения этого имени, то они относятся не к той переменной, которая сейчас объявляется, а к ее тезке. В соответствии со сказанным в разд. 3.2.6 за $s{\uparrow}$ следует принять наиболее сильное из не зависящих от v логических следствий формулы p — формулу $p'\ \approx\ \exists\, v\, p$.

Действительно, то, что эта формула не зависит от v, — очевидно, как и то, что она логически следует из формулы p. Пусть p'' — еще одно следствие формулы p, не зависящее от v. Оно не было бы слабее формулы p', если бы в некотором состоянии w' оказалось, что формула $p' \supset p''$ ложна, т. е. значение p' — истина, а значение p'' — ложь. Но тогда нашлось бы состояние w'', отличающееся от w' только значением переменной v, в котором p истинна, а p'' по-прежнему ложна. Это означало бы, что p'' не является следствием p.

Итак,

$$\mathrm{Sem}(\mathbf{loc}\, v){\uparrow}(p)\ \approx\ \exists\, v\, p. \tag{48}$$

В частности, если p не зависит от v, то

$$\mathrm{Sem}(\mathbf{loc}\, v)\uparrow(p) \;\approx\; p. \tag{49}$$

Этим приемом удалось нейтрализовать всю информацию, содержавшуюся в предусловии p о недоступной в данном блоке тезке переменной v, ничего не добавляя относительно вновь объявленной переменной.

Перейдем к обратному преобразователю предикатов $s\downarrow$ для того же оператора. Пусть состояния w и w_1 определены так же, как выше, q — формула, которая может зависеть от v. Из того же разд. 3.2.6 следует, что за $s\downarrow(q)$ надо принять наиболее слабую из всех логических формул, не зависящих от v, истинность которых обеспечивает истинность q. Рассуждения, симметричные тем, что были приведены выше по поводу формулы $\exists\, v\, p$, показывают, что требуемая формула — это $\forall\, v\, q$.

Однако вывод, что эта формула и представляет собой искомый преобразователь, не должен быть поспешным.

Там же в разд. 3.2.6 подчеркивалось, что свойством (3.2.6–22) обладают обратные преобразователи предикатов только для детерминированных операторов S. Оператор **loc** v заменяет значение переменной v на неопределенное — $d?$, имеющее, как было сказано в разд. 3.2.5, две трактовки. При первой значение $d?$ детерминировано, но его использование в любых вычислениях запрещено. При второй оно произвольно, но тогда оператор **loc** v обладает как бы высшей степенью недетерминированности. Означает ли это, что исполнение программы не может быть продолжено?

Если формула q не зависит от v, то она эквивалентна формуле $\forall\, v\, q$ и выбор между ними в качестве предусловия для оператора **loc** v, как и опасения, связанные с неопределенностью значения v, теряют актуальность. Естественно, однако, выбрать формулу q как более простую.

Пусть теперь формула q существенно зависит от v, т. е. может быть истинна при одних значениях v и ложна при других. Тогда формула $\forall\, v\, q$ эквивалентна *false*, а такое предусловие запрещает исполнить оператор **loc** v. Это естественно, так как при неопределенном значении v невозможно утверждать, что требуемое постусловие q будет удовлетворяться. Поэтому словечко «пока» в связи с видимым отсутствием помех к исполнению объявления вроде бы не было лишним.

Но каковы шансы на то, что формула q окажется зависящей от v? Убедимся, что в хорошо построенной программе, где предусмотрена инициализация всех объявленных переменных, их нет. На каждом пути от начала тела блока найдем первое появление переменной v. Оно должно быть левой частью оператора присваивания $v := e$, инициализирующего переменную v. Выражение e не должно зависеть от v, поэтому присваивание может быть исполнено беспрепятственно. Если постусловие q' для этого оператора и зависело от v, то предусловие $p' \;\approx\; q'[e/v]$ (см. формулу (42)) зависеть уже не будет. Формула p' становится постусловием для предшествующего оператора на рассматриваемом пути (построение обратного преобразователя предикатов идет от преемника к предшественнику), и так для всех путей, если только ни на одном из них своевременная инициализация не забыта. На самом деле рассуждения должны быть несколько сложнее, так как

надо рассматривать все множество *dvars* объявленных переменных, а не одну переменную v, но это усложнение чисто техническое. Положение мало меняется, если переменная v инициализируется оператором ввода (подробности — в следующем пункте).

Итак, как бы ни трактовать символ $d?$, постусловие q для оператора **loc** v не должно зависеть от переменной v. При соблюдении этого требования обратный преобразователь предикатов — это тождественное преобразование:

$$\mathrm{Sem}(\mathbf{loc}\,v)\downarrow(q) \;\approx\; q. \tag{50}$$

Замечания. 1. Вместо $\mathrm{Sem}(\mathbf{loc}\,v)$ в формулах (48) и (50) можно было бы написать $\mathrm{SemIn}(\ \mathbf{loc}\,v;\ S',\ dvars,\ w_0)$. В начале данного пункта объяснялось, почему это не делается.

2. В некоторых языках есть возможность совмещать объявление переменной с присваиванием ей начального значения. Синтаксис такого комбинированного действия может иметь вид **loc** $v := e$. Если $e(w) = d$, то исполнение этого действия приводит к состоянию $w_1 = w\{d/v\}$. Отличие от обычного присваивания в том, что переменная v либо создается впервые, либо не имеет ничего общего со своей тезкой из внешнего блока. Читателю предлагается самому написать преобразователи предикатов для такого действия.

Преобразование предикатов при выходе из блока будет рассмотрено ниже в п. 10.

8. Инициализация переменных присваиванием им определенных значений встречается не столь уж часто. Обычно начальные значения переменных программы (исходные данные для ее работы) задаются оператором ввода.

Оператор ввода $S \approx \mathit{Read}(v_1, \ldots, v_n)$, если и устраняет неопределенность значений переменных $v_1, \ldots, v_n$, то не в полной мере. В разд. 3.2.2 уже было сказано, что постусловие для такого оператора должен написать программист. Сейчас мы подготовлены к тому, чтобы уточнить, что именно должно быть сделано автором программы и какова должна быть связь между контекстом оператора ввода (его постусловием и предусловием) и его текстом.

Новый синтаксис оператора ввода должен быть примерно таким:

$$S \;\approx\; \mathit{Read}(v_1, \ldots, v_n)[q0(v0_1, \ldots, v0_n)],$$

где $q0(v0_1, \ldots, v0_n)$ — это и есть добавленная к программе логическая формула, описывающая свойства вводимых значений $v0_i$ переменных v_i.

Содержательная семантика такого оператора — ввести извне эти значения и проверить, что они действительно удовлетворяют формуле $q0$. Если нет, то исполнение оператора S не может быть завершено.

Если оператор S предназначен только для ввода исходных данных программы и исполняется лишь однократно, то предусловие p не должно зависеть от переменных v_i, значения которых к этому моменту еще не определены. Но если значения этих переменных периодически обновляются извне, то переменные v_i могут входить в предусловие. Зависимость значения формулы p от значений этих переменных должна быть погашена подобно тому, как это делается при их переобъявлении и при обычном присваивании им новых значений.

С учетом всего сказанного здесь и в разд. 3.2.2, прямой преобразователь предикатов для оператора S должен иметь вид

$$s{\uparrow}(p) \approx (\exists\, v_1\ \ldots\ \exists\, v_n)\, p \wedge q0(v0_1, \ldots, v0_n) \\ \wedge\ v_1 = v0_1 \wedge \ldots \wedge v_n = v0_n. \tag{51}$$

Обратный преобразователь можно получить, исходя из того, что оператор S фактически эквивалентен оператору

if $q0(v0_1, \ldots, v0_n)$ **then** $v_1 := v0_1; \ldots; v_n := v0_n$ **else abort fi**

Используя формулы (38), (30), (42) и (26), после упрощений получаем

$$s{\downarrow}(q) \ \approx\ q0(v0_1, \ldots, v0_n) \wedge q[v0_1/v_1] \ldots [v0_n/v_n]. \tag{52}$$

Определяемое по формуле (52) предусловие не зависит от текущих значений переменных v_i, как это и должно быть. Своеобразие этого предусловия в том, что в нем фигурируют значения $v0_i$, еще не известные перед исполнением оператора ввода. Это обязывает выполнить ввод, определив при этом значения $v0_1, \ldots, v0_n$, найти значение $s{\downarrow}(q)$ в соответствии с формулой (52) и, только получив значение *true*, исполнить присваивания $v_1 := v0_1; \ldots; v_n := v0_n$. Если же окажется, что $s{\downarrow}(q) \approx$ *false*, завершение исполнения оператора ввода блокируется.

Пример. Пусть $S \approx \mathit{Read}(v)[v0 < 2]$, $q \approx v > 1$. Тогда формула (52) принимает вид $s{\downarrow}(q) \equiv v0 < 2 \wedge v0 > 1$, что вполне осмысленно. Если числа 1 и 2 поменять местами, то получим невыполнимое предусловие $v0 < 1 \wedge v0 > 2$. Но и здесь все разумно. Член $v0 > 2$ говорит, что введенное значение переменной v будет больше двух, а член $v0 < 1$ — что при таком значении успешное завершение программы не гарантируется. Своевременно получить подобную информацию отнюдь не вредно.

9. Ясно, что оба преобразователя предикатов для оператора вывода $S \approx \mathit{Write}(e_1, \ldots, e_m)$, как и для $S \approx$ **skip**, определяют тождественное преобразование:

$$s{\uparrow}(p) \ \approx\ p, \tag{53}$$
$$s{\downarrow}(q) \ \approx\ q. \tag{54}$$

Однако предусловие p оператора вывода представляет интерес не столько в формальной семантике программы, сколько за ее пределами — оно дает пользователю более или менее полную информацию о свойствах выводимых значений (результатов работы программы), их связи с константами $v0_i$ (данными, полученными при последнем вводе значений переменных v_i).

10. Выше, в п. 7, было начато построение преобразователей предикатов для блока $S \approx$ **begin loc** $v_1; \ldots,$ **loc** $v_n;\ S1$ **end**. Чтобы его завершить, осталось построить преобразователи предикатов для действия SemOut, выполняемого при выходе из блока в связи с наличием в блоке объявления **loc** v. В разд. 3.2.5 было сказано, что это действие как бы эквивалентно присваиванию $v := w_0(v)$, в котором v и слева и справа представляет имя переменной, а не ее значение. Исходя из этого и опираясь на формулы (42)–(47) из п. 6, а также на результаты п. 7, напишем:

$$\mathrm{SemOut}(S, v, w_0){\uparrow}(p) \ \approx\ (\exists\, v\, p) \wedge v = w_0(v). \tag{55}$$

Правда, в формуле (47) стоит p, а не $\exists\, v\, p$, но зависимость предиката p от значения v, действовавшего в блоке, следует погасить. На самом деле, в хорошей программе этот предикат не должен зависеть от локализованной в блоке переменной v, но конкретизировать это утверждение здесь нельзя.

Навешивать кванторы по переменным v_i, объявленным в блоке, и добавлять конъюнктивные члены $v_i = w_0(v_i)$ можно поодиночке, как это предусматривает формула (55). Но проверять истинность q следует только после восстановления значений всех этих переменных. Поэтому вместо формулы

$$\mathrm{SemOut}(v, w_0)\!\downarrow\!(q) \;\approx\; q[w_0(v)/v],$$

буквально отражающей сказанное выше, опишем суммарный результат этих действий для блока S так:

$$\mathrm{SemOut}(S, w_0)\!\downarrow\!(q) \;\approx\; q[w_0(v_n)/v_n] \ldots [w_0(v_1)/v_1]. \tag{56}$$

Содержательный смысл этой проверки ясен — выйти из блока можно, если восстановление не нарушает постусловия для блока в целом.

3.2.8. Обоснование правил деривационной семантики

В разд. 3.2.6 мы убедились, что из формулы $s\!\uparrow\!(p) \approx q$ логически следует формула $p\{S\}q$, а из формулы $p \approx s\!\downarrow\!(q)$ — формула $p[S]q$ (для детерминированных алгоритмов). Таким образом, каждому выражению для прямого преобразователя предикатов можно сопоставить некоторое правило вывода в исчислении программ со слабыми формулами, а каждому выражению для обратного преобразователя предикатов — правило с сильными формулами.

Теперь мы это и проделаем на основе результатов предыдущего раздела. В этом и заключается (хотелось бы верить) основная ценность денотационного подхода к семантике алгоритмических языков, тогда как все, что было сделано в разд. 3.2.3–3.2.5, — всего лишь математизированное изложение достаточно хорошо известных правил интерпретации программ и их фрагментов.

Начнем с конца. Теореме 3.2.12 соответствует аксиома

$$p\ \{v := e\}\ \exists\, i\ (p[i/v] \wedge v = e[i/v]).$$

В обозначениях разд. 3.2.2 она совпадает с аксиомой Флойда А3:

$$p(v)\ \{v := e(v)\}\ \exists\, i\ (p(i) \wedge v = e(i)).$$

Теорема 3.2.11 приводит, с точностью до обозначений, к аксиоме Хоора А4:

$$q[e/v]\ [v := e]\ q.$$

Выражению (3.2.7–37): $s\!\uparrow\!(p) \;\approx\; s1\!\uparrow\!(b^+ \wedge p) \vee s2\!\uparrow\!(b^- \wedge p)$ для оператора $S \approx$ **if** b **then** $S1$ **else** $S2$ **fi** соответствует правило вывода:

$$\frac{b\,p\ \{S1\}\ q1 \qquad \overline{b}\,p\ \{S2\}\ q2}{p\ \{\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}\}\ q1 \vee q2} \tag{57}$$

а выражению (3.2.7–38): $s \downarrow (q) \;\approx\; b^+ \wedge s1 \downarrow (q) \vee b^- \wedge s2 \downarrow (q)$ для того же оператора S — правило

$$\frac{p1\ [S1]\ q \qquad p2\ [S2]\ q}{b\,p1 \vee \overline{b}\,p2\ [\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}]\ q} \tag{58}$$

Доказательство (57). Пусть для некоторых формул p, $q1$ и $q2$ операторные формулы $b\,p\ \{S1\}\ q1$ и $\overline{b}\,p\ \{S2\}\ q2$ истинны. Это значит, что $s1{\uparrow}(b^+ \wedge p) \supset q1$ и $s2{\uparrow}(b^- \wedge p) \supset q2$.

Пусть формула $s{\uparrow}(p)(w_1)$ истинна. Отсюда последовательно вытекает:

$$\begin{aligned}&(s1{\uparrow}(b^+ \wedge p) \vee s2{\uparrow}(b^- \wedge p))(w_1)\\ &\quad\Leftrightarrow (s1{\uparrow}(b^+ \wedge p)(w_1)) \vee (s2{\uparrow}(b^- \wedge p)(w_1))\\ &\quad\Rightarrow q1(w_1) \vee q2(w_1) \Leftrightarrow (q1 \vee q2)(w_1).\end{aligned}$$

Таким образом, если оператор S начал исполняться в состоянии w, удовлетворяющем формуле p, и завершил работу в состоянии w_1, то в этом состоянии обязана быть истинной формула $q1 \vee q2$. Именно это и утверждает заключение правила (57).

Доказательство (58). Пусть операторы $S, S1$ и $S2$ детерминированы. Пусть также истинны посылки правила (58): $p1\ [S1]\ q$ и $p2\ [S2]\ q$. Это значит, что $p1 \supset s1{\downarrow}(q)$, $p2 \supset s2{\downarrow}(q)$. Для произвольного состояния w, удовлетворяющего формуле $s{\downarrow}(q)$, последовательно получаем:

$$\begin{aligned}&(b\,p1 \vee \overline{b}\,p2)(w) \Leftrightarrow b^+(w) \wedge p1(w) \vee b^-(w) \wedge p2(w)\\ &\quad\Rightarrow b^+(w) \wedge s1{\downarrow}(q)(w) \vee b^-(w) \wedge s2{\downarrow}(q)(w)\\ &\quad\Leftrightarrow (b^+ \wedge s1{\downarrow}(q))(w) \vee (b^- \wedge s2{\downarrow}(q))(w)\\ &\quad\Leftrightarrow (b^+ \wedge s1{\downarrow}(q) \vee b^- \wedge s2{\downarrow}(q))(w) \Leftrightarrow s{\downarrow}(q)(w).\end{aligned}$$

Любое из условий $b^+(w)$ или $b^-(w)$ означает, что выражение b может быть вычислено в состоянии w, а сильные формулы в качестве посылок правила (58) гарантируют завершение оператора S. Детерминированность оператора S означает, что заключительное состояние w_1 определено однозначно, а так как формула $s{\downarrow}(q)(w)$ верна, то верно и $q(w_1)$. В совокупности все это влечет за собой истинность заключения правила (58).

Правило (57) несколько отличается от правила

$$\frac{b^+ \wedge p\ \{S1\}\ q \qquad b^- \wedge p\ \{S2\}\ q}{p\ \{\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}\}\ q}, \tag{59}$$

предложенного Хоором. Но, хотя на первый взгляд правило (59) слабее правила (57), в действительности они взаимозаменяемы в исчислении, где имеются правила R1′ и R1″ из разд. 3.2.2:

$$(\mathrm{R}')\qquad \frac{p \supset p1 \quad p1\ \{S\}\ q}{p\ \{S\}\ q}, \qquad\qquad (\mathrm{R}'')\qquad \frac{p\ \{S\}\ q1 \quad q1 \supset q}{p\ \{S\}\ q}$$

и выводимы все логически общезначимые формулы исчисления предикатов.

Действительно, из посылок правила (59) его заключение выводится с помощью правила (57) следующим образом:

1) $b^+ \wedge p\ \{S1\}\ q$
2) $b^- \wedge p\ \{S2\}\ q$
3) $p\ \{\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \ \textbf{fi}\}\ q \vee q$ (1, 2; правило (57))
4) $q \vee q \supset q$ (общезначима)
5) $p\ \{\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \ \textbf{fi}\}\ q$ (3, 4; правило R1″)

Теперь из посылок правила (57) выведем его заключение с помощью правила (59):

1) $b^+ \wedge\ p\ \{S1\}\ q1$
2) $b^- \wedge\ p\ \{S2\}\ q2$
3) $q1 \supset q1 \vee q2$ (общезначима)
4) $b^+ \wedge\ p\ \{S1\}\ q1 \vee q2$ (1, 3; правило R1″)
5) $q2 \supset q1 \vee q2$ (общезначима)
6) $b^- \wedge\ p\ \{S2\}\ q1 \vee q2$ (2, 5; правило R1″)
7) $p\,\{\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}\}\ q1 \vee q2$ (4, 6; правило (59))

Обратимся к выражению (3.2.7–39):

$$s{\uparrow}(p) \approx \exists n\,(n \geqslant 0 \wedge p_n \wedge b^-),$$

где $p_0 \approx p$ и $p_{n+1} \approx bs1{\uparrow}(p_n)$ при $n \geqslant 0$, для прямого предикатного преобразователя оператора $S \approx$ **while** b **do** $S1$ **od**. Ему соответствует правило вывода:

$$\frac{p \equiv p_0 \qquad \forall n\,(n \geqslant 0 \supset (b^+ \wedge p_n)\{S1\}\,p_{n+1})}{p\ \{\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}\}\ b^- \wedge \exists n\,(n \geqslant 0 \wedge p_n)}\ . \tag{60}$$

Правило Хоора выглядит значительно проще:

$$\frac{b^+ \wedge\ p\ \{S1\}\ p}{p\ \{\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}\}\ b^- \wedge\ p}\ . \tag{61}$$

Но оказывается, что правила (60) и (61) также взаимозаменяемы.

Заключение правила (61) выводится из его посылки с помощью правила (60) за один шаг, если принять $p_n \approx p$ для любого n и учесть, что и формула $\exists n\,(n \geqslant 0 \wedge p_n)$ сводится при этом к формуле p.

Покажем, как из посылок правила (60) вывести его заключение с помощью правила (61):

1) $p \equiv p_0$
2) $\forall n\,(n \geqslant 0 \supset b^+ \wedge\ p_n\,\{S1\}\,p_{n+1})$
3) $b^+ \wedge \exists n\,(n \geqslant 0 \wedge p_n) \supset b^+ \wedge\ p_{n0}$
4) $b^+ \wedge\ p_{n0}\ \{S1\}\ p_{n0+1}$
5) $b^+ \wedge \exists n\,(n \geqslant 0 \wedge p_n)\ \{S1\}\ p_{n0+1}$ (3,4; R1′)
6) $p_{n0+1} \supset \exists n\,(n \geqslant 0 \wedge p_n)$ (общезначима)
7) $b^+ \wedge \exists n\,(n \geqslant 0 \wedge p_n)\ \{S1\}\ \exists n\,(n \geqslant 0 \wedge p_n)$ (5, 6; R1″)
8) $\exists n\,(n \geqslant 0 \wedge p_n)\{\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}\}b^- \wedge \exists n\,(n \geqslant 0 \wedge p_n)$ (7; 5′)
9) $p_0 \supset \exists n\,(n \geqslant 0 \wedge p_n)$ (общезначима)
10) $p_0\ \{\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}\}b^- \wedge \exists n\,(n \geqslant 0 \wedge p_n)$ (9, 8; R1′)
11) $p\ \{\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}\}\ b^- \wedge \exists n\,(n \geqslant 0 \wedge p_n)$ (1, 10; R1′)

Этот вывод не вполне формален. Формула 3) получена на уровне содержательных рассуждений — через $n0$ обозначено то значение n, существование которого утверждает посылка этой формулы. Формула 4) — это частный случай формулы 2). При выводе формулы 11) по правилу (R1′) использована не сама формула 1), а следствие из нее: $p \supset p_0$. В разд. 1.4.5 и 1.4.6 говорилось о возможности строго обосновать подобные приемы, сокращающие вывод.

Выражение (3.2.7–40): $s\downarrow(q) \approx \exists n\,(n \geqslant 0 \wedge q_n)$, где $q_0 \approx b^- \wedge q$, а при $n \geqslant 0$ — $q_{n+1} \approx b^+ \wedge s1\downarrow(q_n)$, приводит к правилу вывода с сильными формулами для оператора цикла. Обозначим $q'_n \approx s1\downarrow(q_n)$. Тогда

$$q_{n+1} \approx b^+ \wedge q'_n, \quad q'_{n+1} \approx s1\downarrow(b^+ \wedge q'_n), \quad \text{для } n = 0, 1, \ldots;$$
$$s\downarrow(q) \approx (b^- \wedge q) \vee (b^+ \wedge \exists n\,(n \geqslant 0 \wedge q'_n)),$$

так что правило имеет вид

$$\frac{q_0\ [S1]\ b^- \wedge q \qquad \forall n\,(n \geqslant 0 \supset q_{n+1}\ [S1]\ b^+ \wedge q_n)}{(b^- \wedge q) \vee (b^+ \wedge \exists n\,(n \geqslant 0 \wedge q_n))\ [\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od}]\ q}\ . \tag{62}$$

Предикаты q_n в правиле (62) соответствуют только что введенным предикатам q'_n, а не исходным предикатам q_n.

Для сильной операторной формулы нельзя построить аналог формулы (61), где в инварианте отсутствовала бы кванторная формула, характеризующая прогресс в ходе вычислений по мере изменения значения подкванторной переменной n, — приближение к состоянию, в котором цикл заведомо завершается (см. разд. 3.1.3).

3.2.9. Операторные схемы, рекурсия и циклы

Циклы и рекурсивные процедуры принадлежат к арсеналу средств структурного программирования наряду с условными и выбирающими операторами. Последние имеют вид (при $v \in$ IV, $d_1, d_2, \ldots, d_m \in$ Z)

case v **of** $d_1 : S_1;\ d_2 : S_2;\ \ldots;\ d_m : S_m$ **esac**

Такой оператор может быть сведен к условным операторам:

if $v = d_1$ **then** S_1
else if $v = d_2$ **then** S_2
else . . .
else if $v = d_m$ **then** S_m
fi (возможно также **else skip fi**) ... **fi fi**

На первый взгляд, операторные схемы (их графы переходов) представляют собой более общий способ описания последовательности действий. Ясно, что любую операторную схему можно описать на языке программирования, в котором разрешено пользоваться операторами перехода. Но можно ли обойтись только «структурными» средствами описания последовательности действий? Оказывается, да.

Убедимся, что графы переходов, рекурсивные процедуры и циклы представляют собой эквивалентные средства задания схемы управления в программе (в сочетании с составными, условными и выбирающими операторами).

Пусть описание процедуры P без параметров с телом S имеет вид $\langle P, S\rangle$.

Следствие 3.2.5 из семантики оператора цикла, да и сама содержательная семантика оператора цикла показывают, что при любых b и $S1$ оператор цикла **while** b **do** $S1$ **od** может быть заменен оператором рекурсивной процедуры $Loop$ с описанием

$\langle Loop,$ **if** b **then** $S1; Loop$ **else skip fi**$\rangle$

Для того же оператора на рис. 3.1.4–1 г1 был приведен соответствующий ему фрагмент операторной схемы (рис. 1).

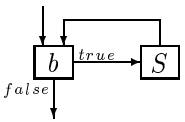


Рис. 1

Таким образом, для циклов, а следовательно, и для программ с циклами существуют эквиваленты как в классе рекурсивных программ (т. е. программ с рекурсивными процедурами), так и в классе операторных схем.

Пусть дана произвольная операторная схема с одним или несколькими выходами. Опишем способ ее преобразования в рекурсивную программу.

Добавим к переменным схемы еще одну логическую переменную *cont* (сокращение от *continue* — продолжать) и к каждому оператору выхода — присваивание $cont := false$.

Если в схеме есть операторы, например Sa и Sb, соединенные дугой (от Sa к Sb), причем от Sa и к Sb не ведут никакие другие дуги, то такие операторы, очевидно, можно объединить в один оператор $Sa; Sb$.

Все операторы, в которые заходит более одной дуги, пометим различными метками.

Если из оператора S исходит одна или несколько дуг к помеченным операторам, то это можно описать, включая в S оператор перехода (безусловного) к соответствующей метке или условный (или выбирающий) оператор, содержащий внутри себя переходы к нужным меткам.

Пусть все это проделано. Далее, двигаясь по графу переходов от одного из

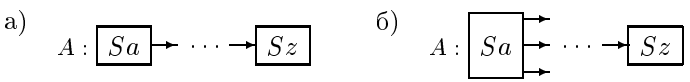


Рис. 2

операторов выхода Sz назад по дугам, дойдем до первого помеченного оператора Sa (им может быть сам оператор выхода) — рис. 2 а. Затем, двигаясь по тому же графу от вершины Sa в направлении дуг, построим максимальный фрагмент этого графа с оператором Sa в корне, непомеченными прочими вершинами и операторами выхода в качестве терминальных вершин (рис. 2 б).

Для случая, представленного на рис. 2 а (иначе потребуется очевидное усложнение), составим описание процедуры $\langle A,\ Sa;\ \ldots;\ Sz\rangle$, а в более привычном виде — **procedure** A; **begin** $Sa; \ldots; Sz$ **end**

Все операторы перехода **go to** A заменим операторами процедуры A. Это относится и к операторам **go to** A, возможно содержащимся в теле процедуры A. Из графа переходов выбросим операторы $Sa, \ldots, Sz$ и все дуги, заходящие в оператор Sa. В схеме при этом наряду с оставшимися операторами выхода могут появиться другие операторы, из которых не исходит ни одна дуга.

Если оставшаяся часть схемы содержит хотя бы один помеченный оператор, применим к ней тот же процесс, рассматривая только что упомянутые операторы без исходящих дуг наравне с операторами выхода. Когда появляется новое описание процедуры, например $\langle B,\ Sb;\ \ldots;\ Sy\rangle$, то операторы **go to** B заменяем на

оператор B не только в оставшейся части операторной схемы, но и во всех телах процедур.

За конечное число шагов этот процесс закончится. От операторной схемы останется несколько непомеченных операторов, образующих выходящее дерево с корнем в операторе входа. Эту часть операторной схемы следует считать основной программой. Остальные операторы перейдут в тела процедур. В начале основной программы (не позже первого разветвления) поместим оператор $cont := true$.

Рассмотрим в качестве примера блок-схему программы слияния упорядоченных массивов (пример 3.1–2, рис. 3.1.1–1). Последовательно выделяем следующие процедуры:

$\langle D,\ k := k + 1;\ \textbf{if}\ k <= m + n\ \textbf{then}\ C\ \textbf{else}\ Write(c);\ cont := false\ \textbf{fi}\rangle;$

$\langle A,\ c[k] := a[i];\ i := i + 1;\ D\rangle;$

$\langle B,\ c[k] := b[j];\ j := j + 1;\ D\rangle;$

$\langle C,\ \textbf{if}\ i > m\ \textbf{then}\ B$
$\quad\textbf{else if}\ j > n\ \textbf{then}\ A$
$\quad\quad\textbf{else if}\ a[i] > b[j]\ \textbf{then}\ B\ \textbf{else}\ A\ \textbf{fi}$
$\quad\quad\textbf{fi}$
$\quad\textbf{fi}\rangle;$

и основную программу:

$\textbf{begin}\ Read(m, n, a, b);\ i := 1;\ j := 1;\ k := 1;$
$\quad cont := true;\ C\ \textbf{end}$

Выполнение этой программы и каждой из процедур заканчивается либо обращением к другой процедуре (в исходных точках ветвления), либо исполнением оператора $cont := false$, означающим, что достигнут выход из операторной схемы. Так построенная совокупность основной программы и процедур эквивалентна исходной операторной схеме в отношении последовательности и условий выполнения операторов. Формального доказательства этого мы дать не можем, так как семантика оператора перехода не была формализована. Впрочем, описанный здесь и ниже прием преобразования операторной схемы в рекурсивную программу, а затем — в цикл, как раз и формализует понятие перехода, сводя его к более фундаментальным понятиям.

Итак, мы умеем преобразовывать циклы в операторные схемы, а те и другие — в рекурсивные программы с процедурами без параметров. Правда при этом возникают лишь такие рекурсивные процедуры, у которых обращение к другой процедуре возможно только в самом конце исполнения тела процедуры, так что завершение процедуры, к которой произошло обращение, означает одновременное завершение всей цепочки процедур, непосредственно или косвенно к ней обратившихся. Такая рекурсия называется *концевой*.

Покажем, что рекурсивная программа такого вида может быть превращена в циклическую. Здесь нам помогут операторы $cont := true$ и $cont := false$, которые в аппарате рекурсивных программ не нужны.

Потребуются также следующие преобразования. Снабдим все процедуры номерами от 1 до m. Заменим всюду оператор процедуры, получившей номер i, оператором $v := i$ (v — новая переменная). В конец основной программы, подвергнутой тем же переделкам, поместим оператор

while *cont* **do case** v **of** 1: S_1; ...; m: S_m **esac od**

где во внутреннем выбирающем операторе после метки i: стоит тело S_i процедуры с номером i. Сами описания процедур исключим из программы.

В нашем примере получим:

```
begin Read(m,n,a,b); i := 1; j := 1; k := 1; cont := true;
  v := 1;
  while cont do
    case v of
1:    if i > m then v := 2
      else if j > n then v := 3
           else if a[i] > b[j] then v := 2 else v := 3 fi
           fi
      fi;
2:    begin c[k] := b[j]; j := j + 1; v := 4 end;
3:    begin c[k] := a[i]; i := i + 1; v := 4 end;
4:    begin k := k + 1;
        if k <= m + n then v := 1 else Write(c); cont := false fi
      end
    esac
  od
end
```

Ясно, что в результате присваивания *cont* := *false* происходит (не только в данном примере) завершение работы программы, а в результате присваивания $v := i$ — повторный вход в выбирающий оператор и исполнение тела i-й процедуры.

Другой пример избавления от явных переходов был приведен в разд. 2.3.3 на с. 173. Но это был именно пример, а сейчас описан общий, возможно внешне неуклюжий, метод подобных преобразований.

Типичным примером неконцевой рекурсии служит процедура *Traverse* левостороннего обхода дерева, приведенная в разд. 2.4.2. Внутри этой процедуры рекурсия для обхода правого поддерева — концевая. Ее можно было бы исключить, видоизменив тело процедуры следующим образом:

```
while ptr≠ nil do
begin Traverse(ptr↑.left); Write(ptr↑.val);
  ptr := ptr↑.right end
```

Но рекурсия для обхода левого поддерева — существенно неконцевая: параметр *ptr* нельзя заменить глобальной переменной, так как ее значение должно быть сохранено к моменту возвращения после обхода левого поддерева, а во время этого обхода оно может оказаться испорченным. Кроме того, надо обеспечить

продолжение работы программы после обхода левого поддерева. Со всем этим можно справиться, введя глобальный массив (стек) для запоминания текущего значения *ptr* на время обхода левого поддерева и добавив операторы, обеспечивающие как само запоминание-восстановление, так и продвижение указателя стека. Иначе говоря, в программе можно явно проделывать то, что неявно делает любой интерпретатор языка, допускающего рекурсию. По завершении обхода дерева надо обследовать состояние стека. Если он не пуст, то закончился обход всего лишь поддерева, — надо восстановить из стека значение *ptr* и продолжить цикл исполнения тела процедуры. Если же стек пуст, то завершен обход всего дерева, — следует выйти из цикла и из процедуры.

Нерекурсивный вариант процедуры *Traverse* таков:

```
procedure Traverse1 (ptr : RefRec);
  var i : integer; pstack : array [1 .. n] of RefRec;
begin i := 0;
  repeat
    while ptr ≠ nil do
      begin i := i + 1; pstack[i] := ptr; ptr := ptr↑.left end;
    ptr:=pstack [i]; Write (ptr↑.val); ptr := ptr↑.right; dec(i)
  until (i = 0) ∧ (ptr = nil)
end
```

Равенство $i = 0$ означает, что стек пуст, условие $ptr =$**nil** имеет прежний смысл.

Подобные приемы позволяют сводить рекурсивные алгоритмы к циклическим в широком классе случаев. Однако наш пример показывает, что рекурсивные алгоритмы не только короче, но и смысл их гораздо более ясен. В случае неконцевой рекурсии так бывает почти всегда.

3.3. Денотационная семантика составных значений и указателей

До сих пор рассматривались переменные только одного типа — с целочисленными значениями. При этом в понятии типа не было необходимости и в описании переменной указывался только ее идентификатор (**loc** — это не тип, а синтаксический выделитель оператора описания вроде **begin** или **if**). В этом разделе объектам: значениям, переменным и выражениям можно будет задавать, кроме типа **int**, ряд других типов. Базовым типом остается **int**, а остальные будут подробнее описаны ниже. Пока скажем только, что они делятся на три группы: векторы (одномерные массивы), записи (или структуры) и ссылки.

3.3.1. Векторы, записи и ссылки

Вектор — это значение, состоящее из нескольких других значений, его *элементов*, имеющих произвольный, но один и тот же для всех элементов тип. Число элементов также произвольно, но при описании типа вектора оно фиксируется для данного типа. Элементы массива снабжаются *индексами*, т. е. нумеруются от 1 до верхней границы, равной числу элементов. Любой элемент может быть индивидуально упомянут (назван) по имени, составленному из имени вектора и индекса элемента. В программе индекс может быть представлен произвольным выражением, однако ниже термин «имя» мы обычно будем понимать в более узком смысле, когда вместо индекса подставляется его текущее значение — номер элемента, однозначно выделяющий этот элемент. Если номер неположителен или превосходит границу, то такое имя ничего не именует.

Запись — это значение, состоящее из нескольких других значений, ее *полей*, имеющих произвольный, свой для каждого поля, тип. Поля снабжаются индивидуальными внутренними именами — *выделителями полей*. Полный список выделителей и типов полей задается при описании типа записи. Любое поле может быть упомянуто по его (внешнему) имени, составленному из имени записи и выделителя поля. Оба имени — записи и поля (внешнее) — должны пониматься одинаково: либо в первом (программном), либо во втором (узком) смысле, когда все индексы, встречающиеся в имени записи, заменены номерами.

Ссылка — это еще одна разновидность имени. Оно дается значению определенного типа — *предмету ссылки*, но и само также является значением. Переменные с такими значениями называются *указателями*. Ссылка предназначена для использования машиной, на которой исполняется программа, а не человеком, работающим с этой программой. Для него предмет ссылки может быть косвенно представлен именем, которое строится на основе доступного ему имени этого указателя. Тип предмета ссылки задается при описании указателя. Следовательно, все ссылки, которые могут быть значениями данного указателя, обязаны именовать значения одного и того же типа.

Символ **nil** обозначает фиктивную, ничего не именующую ссылку. Сама она может быть значением любого указателя и используется в тех случаях, когда

программа требует задать это значение, но предмет ссылки отсутствует (список не имеет продолжения, в дереве нет соответствующей ветви и т. п.).

Итак, в содержательной семантике есть три вида имен — два явных (сама переменная программы и имя, получаемое из нее заменой индексных выражений на их значения) и скрытые имена (ссылки). Последние могут иметь несколько явных синонимов в виде значений указателей. Множество переменных программы обозначим через Var, множество имен в узком смысле — через Name, а множество ссылок — через Ref.

Для описания отношения между составными (структурированными) значениями — векторами и записями — и их компонентами будем пользоваться терминами *надобъект* и *подобъект*. Элементы вектора — это его подобъекты, а поля — подобъекты записи. Подобъекты подобъектов также считаются подобъектами (более низкого уровня) составного значения, а само это значение — надобъектом по отношению к любому своему подобъекту.

Предмет ссылки не считается ее подобъектом.

Когда исполняется объявление структурированной переменной (имеющей тип вектора или записи), то кроме этой переменной также создаются без явного объявления все ее переменные-компоненты (элементы вектора или поля записи). В принятой терминологии — вместе с надобъектом создаются все его подобъекты.

Способ выделения подобъекта из ближайшего надобъекта полностью определяется типом последнего. Реализация этого способа сильно зависит от машины, на которой исполняется программа, и не может быть описана в общем виде.

Два составных значения *равны* тогда и только тогда, когда они имеют в точности один и тот же тип и при этом равны как их соответствующие компоненты, так и предметы соответствующих друг другу ссылок. Это определение действует до уровня примитивных компонент — целых или логических значений, равенство которых определяется традиционным образом.

Для указателей и составных значений, содержащих указатели в качестве компонент какого-либо уровня, имеет смысл говорить также об их *эквивалентности*. Равные значения эквивалентны. Ссылки эквивалентны, если эквивалентны их предметы, а значение **nil** эквивалентно только самому себе. Одинаково составленные значения эквивалентны, если эквивалентны все их соответствующие компоненты. Иначе говоря, эквивалентность значений — это их совпадение с точностью до ссылок, ориентированных, напомним, на машину.

С точки зрения пользователя понятие эквивалентности более фундаментально, чем понятие равенства. Поэтому ниже запись вида *val*=*val'*, где *val* и *val'* — значения некоторых программных единиц, всегда будет обозначать эквивалентность этих значений.

Понятия, которые мы сейчас ввели, требуют расширения синтаксиса модельного языка и его семантики.

Новый синтаксический класс Type — типы (и одновременно обозначения типов) значений — вместе с его подклассами определяется так:

Type ::= **int** | **bool** | Vector | Record | Pointer | TypeName,
Vector ::= **vector** [N] **of** Type,

Record ::= **record** Fields **endrec**,
Fields ::= Field | Fields ; Field,
Field ::= Type FldSel,
Pointer ::= **ref** Type,

где **int** и **bool** — типы соответственно целочисленных и логических значений, Vector — типы векторов (N — граница индексов для данного вектора, Type — тип его элементов), Record — типы записей, Fields — перечень полей, Field — описание отдельного поля (Type — тип поля, FldSel — его выделитель), Pointer — типы указателей (Type — тип предмета ссылки), TypeName — множество имен типов. Эти имена (ими могут быть только идентификаторы) вводятся в определениях типов, класс которых определяется метаформулой

TypeDefn ::= **type** TypeName = Type,

где TypeName — имя, которое дается типу Type.

Всякий раз, когда необходимо определить тип объекта (переменной или значения) и обнаруживается, что он представлен именем *tpnm* с определением **type** *tpnm*= *typ*, следует заменить это имя типом *typ*. Считаем для простоты, что все типы определены перед началом исполнения программы и более не меняются, хотя во многих реальных языках это не так.

Класс FldSel — множество выделителей полей. Он формируется по мере обработки определений типов записей и делится на подклассы, возможно пересекающиеся, так что в один подкласс попадают выделители для одного типа записей.

Как уже сказано в разд. 3.2.1, в этих формулах N, Type, FldSel и др. — это названия синтаксических классов (N — класс изображений натуральных чисел, Type — класс обозначений типов, FldSel — класс идентификаторов, служащих выделителями полей и т. д.). В программе эти названия должны быть заменены элементами этих классов. Например,

type *Ar* = **vector**[3] **of** *Rec*
type *Rec* = **record int** *i*; **ref** *Ar r* **endrec**

— это определения типов, построенные в соответствии с приведенными выше метаформулами.

В связи с разнообразием типов вместо одного класса IV появляется уже упомянутый класс Var всех возможных переменных программы:

Var ::= VarName | Var [IE] | Var . FldSel | Var ↑ (1)

Класс VarName простых имен (идентификаторов) переменных образует базу для построения любых возможных имен. Три другие конструкции соответствуют упомянутым в начале раздела способам именования элементов векторов, полей записей и предметов ссылок. Назовем эти способы *индексацией* (вектора), *выделением* (поля из записи) и *разыменованием* (указателя).

Как раньше, так и сейчас, характер использования переменной v меняется в зависимости от ее позиции. В составе (или в качестве) выражения она всегда

представляет свое текущее значение (этим, в частности, обеспечивается возможность построения имен в узком смысле), а в левой части оператора присваивания — только имя этого значения. В связи с появлением ссылочных значений, в некоторых случаях переход от имени к значению (т. е. к другому имени) должен быть сделан явным. В этом и заключается суть разыменования.

Переменную класса VarName будем называть простой, несмотря на то, что она может иметь значение, сколь угодно сложное по структуре. Значение же назовем (и уже называли) примитивным, если оно не составное. В программе оно иногда может быть доступно лишь под громоздким именем. Типы примитивных значений — это **int** и **bool**. Тип указателя — **ref** Type — стоит особняком.

Класс Decl объявлений переменных, ранее имевших вид **loc** IV, теперь определяется метаформулой

Decl ::= Type VarName.

Такие объявления вводят в программу указанное простое имя в качестве простой переменной и определяют тип значений этой переменной. Нетрудно видеть, что когда все простые имена объявлены, однозначно определяется тип переменной с любым именем. Это позволяет говорить о целых переменных, переменных-векторах, переменных-записях и указателях.

3.3.2. Состояния, имена, выражения

Поскольку значениями переменных теперь могут быть не только целые числа, надо расширить некоторые понятия. Напомним, что Name обозначает множество всех имен в узком смысле, а Ref — множество ссылок.

Предметная область — это множество $D1 = D \cup Ref'$, где $Ref' = Ref \cup \{\mathbf{nil}\}$ и $\mathbf{nil} \notin (D' \cup Ref)$ (см. разд. 3.2.1 и 3.3.1). Таким образом, **nil** — это не только программный, но и денотационный символ. Обозначим также $D1' = D1 \cup \{d?\}$.

Множество Val — это объединение множества D всех примитивных значений с множеством любых составных значений и множеством ссылок.

Расчленим класс $Expr = \{e, \dots\}$ на подклассы выражений с целочисленными, логическими и ссылочными значениями:

Expr ::= IE | BE | PE,
PE ::= **nil** | Ref | **if** BE **then** PE **else** PE **fi**

(определения классов IE и BE не меняются).

В класс операторов St включим операторы нового вида *new*(*pv*) (*pv* — переменная-указатель), называемые *генераторами*. О них будет сказано ниже, в разд. 3.3.3.

В разд. 3.2.3 каждое состояние рассматривалось как функция, дающая значение переменной по его имени — идентификатору. Сейчас одной функцией нам не обойтись. Структура имени стала более сложной, в нее входят номера элементов векторов, выделители полей записей и символы разыменования. В разд. 3.3.1 уже была отмечена необходимость ввести понятие имени в узком смысле. Активно стало применяться понятие типа объекта.

Будем использовать три функции. Функция nm : Var $\to$ Name $\cup$ Ref преобразует программное имя в имя в узком смысле или в ссылку. Обозначение w дадим теперь функции w : Name $\to$ Val, которая по этому имени находит текущее значение переменной. Функция wt : Expr $\to$ Type дает тип значения любого выражения в текущем состоянии. Отсюда сразу следует, что состоянием надо называть пару $\langle w, wt\rangle$. Пары такого вида будут как аргументами, так и результатами любого преобразователя состояний.

Функция nm словесно была определена выше, в предыдущем разделе. Формально она определяется правилами:

$$nm(v) \approx v \text{ – для } v \in \text{VarName}, \tag{2}$$

$$nm(ar[e]) \approx nm(ar)\ [\ e(w)\ ] \text{ – для } wt(ar) \in \text{Vector}, e \in IE, \tag{3}$$

$$nm(rec\,.\,\mathit{flds}) \approx nm(rec)\,.\,\mathit{flds} \text{ – для } wt(rec) \in \text{Record}, \mathit{flds} \in \text{FldSel}, \tag{4}$$

$$nm(pv\uparrow) \approx nm(pv)\uparrow \text{ – для } wt(pv) \in \text{Ref}. \tag{5}$$

В этих формулах используется символ $\approx$, подчеркивающий скорее синтаксическую, чем семантическую природу функции nm. Следуя уговору — уделять синтаксису минимум внимания, не будем задерживаться на этом, не лишенном дефектов, определении.

Значение выражения e по-прежнему вычисляет функция с именем e и аргументом w. Ее определение, данное в разд. 3.2.4, сохраняется за исключением правила вычисления значения переменной v, рассматриваемой как выражение. Оно принимает вид

$$v(w) = w(nm(v)) \tag{6}$$

(так что $v(w)$ — это теперь сокращение для Value($nm(v)$,w)).

У правила вычисления значений констант появляется еще один частный случай

$$\mathbf{nil}(w) = \mathbf{nil}.$$

Функция wt также нуждается в определении на всем множестве Expr выражений языка. База для расширения дается правилами:

$$wt(c) = \mathbf{int},$$
$$wt(\mathit{true}) = wt(\mathit{false}) = \mathbf{bool},$$
$$wt(\mathbf{nil}) = \mathbf{ref}.$$

Символ **ref** не является законченным изображением типа, но он автоматически наращивается некоторым типом typ в зависимости от контекста, в котором стоит фиктивная ссылка **nil**. Подробнее об этом будет сказано в разд. 3.3.3.

На множестве Var функция wt расширяется и ограничивается динамически, по мере исполнения объявлений в блоках и завершения исполнения блоков.

В результате исполнения объявления $typ\ v$ функция $wt(v)$ приобретает значение typ на время исполнения блока, содержащего это объявление, за исключением периодов исполнения тех подблоков, в которых переменная v переобъявляется. Если typ — тип составного значения, то по нему определяются значения $wt(v[i])$ или $wt(v.\mathit{fsel})$ для любых допустимых значений индекса i или выделителя

fsel. Этот процесс косвенного объявления продолжается до уровня примитивных компонент или компонент-указателей.

Если указатель pv имеет явное или косвенное объявление **ref** *typ* pv, то функция $wt(pv\uparrow)$ приобретает значение *typ* на все время, пока сохраняется ссылка, созданная при исполнении генератора $new(pv)$ или хотя бы одна копия этой ссылки. В разд. 3.3.3 будет разъяснено, как при исполнении генераторов и присваиваний возникают оригиналы и копии ссылок.

Для каждой переменной v ее тип $wt(v)$ определяет, в частности, как может быть использовано ее имя.

Правила регулярного расширения функции wt таковы:

$$wt(e1\ op\ e2) = \begin{cases} \textbf{int}, & \text{если } op \in \mathrm{ArOp}, \\ \textbf{bool}, & \text{если } op \in (\mathrm{RelOp} \cup \mathrm{BoolOp}), \end{cases}$$
$$wt(not\ e) = \textbf{bool}.$$

Здесь должны выполняться следующие ограничения на типы аргументов:

$$wt(e1) = wt(e2) = \begin{cases} \textbf{int}, & \text{если } op \in (\mathrm{ArOp} \cup \mathrm{RelOp}), \\ \textbf{bool}, & \text{если } op \in \mathrm{BoolOp}, \end{cases}$$
$$wt(e) = \textbf{bool}.$$

Далее:

$$wt(\textbf{if}\ b\ \textbf{then}\ e1\ \textbf{else}\ e2\ \textbf{fi}) = wt(e1) = wt(e2)$$

(типы выражений $e1$ и $e2$ обязаны совпадать).

Множество всех функций wt обозначим WT.

Семантику первичных операторов — объявлений, присваиваний, генераторов и оператора ввода, т. е. соответствующие им преобразователи состояний, будем определять в терминах функции $\mathrm{Sem} : \mathrm{St} \to ((\mathrm{W} \times \mathrm{WT}) \to (\mathrm{W} \times \mathrm{WT}))$.

Это позволит нам оставить без существенного изменения все правила регулярного расширения семантики, данные в разд. 3.2.5. Пару $\langle w, wt\rangle$ обычно будем коротко обозначать wwt. Считаем, что любой преобразователь состояний $\mathrm{Sem}(S)$ может выделить из пары wwt (wwt', wwt_1, ...) как элемент w (w', w_1, ...), так и wt (wt', wt_1, ...). Со всеми разумными подробностями эти правила примут вид

$$\mathrm{Sem}(S1; S2)(wwt) = s2(s1(wwt)),$$
$$\mathrm{Sem}(pS)(wwt) = wwt_1 \ \equiv\ p(wwt)\ \wedge\ wwt_1 = s(wwt),$$
$$\mathrm{Sem}(Sp)(wwt) = wwt_1 \ \equiv\ wwt_1 = s(wwt) \wedge p(wwt_1),$$
$$\mathrm{Sem}(\textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \ \textbf{fi})(wwt) = wwt_1$$
$$\equiv\ b^+(wwt) \wedge\ wwt_1 = s1(wwt) \vee\ b^-(wwt) \wedge\ wwt_1 = s2(wwt),$$
$$\mathrm{Sem}(\textbf{while}\ b\ \textbf{do}\ S1\ \textbf{od})(wwt) = wwt_1$$
$$\equiv\ \exists n\,(n \geqslant 0 \wedge wwt_1 = (bs1^n\overline{b})(wwt))$$

(при этом $(bs1)^{n+1}(w, wt) = bs1((bs1)^n(w, wt), wt)$).

Может возникнуть вопрос — зачем аргумент wt передается от одного преобразователя состояний $\mathrm{Sem}(S)$ к другому и нигде видимым образом не используется?

Ответ таков. Среди операторов модельного языка есть операторы объявления, которые преобразуют функцию wt. Им-то и нужна эта функция как объект

преобразования. Результат преобразований после исполнения всех объявлений в блоке должен быть передан телу блока для дальнейшего использования. Никакие другие операторы (в том числе и блоки после выхода из них) не меняют wt.

Но это лишь половина ответа. Среди подробностей, неразумно отнесенных к неразумным, есть и такая — значения выражений (в первую очередь — переменных программы) не могут быть вычислены без помощи функции wt. Это видно из правила сокращения (6), подключающего к вычислению $v(w)$ функцию nm, и из правил (2)–(5) вычисления этой функции, в которых функция wt присутствует явно. Поэтому текущее значение, например, выражения p сейчас записано в виде $p(wwt)$, а не $p(w)$, как это было сделано для выражения v в формуле (6). Тем не менее следует писать $w(nm(v))$, а не $wwt(nm(v))$.

После этих разъяснений будем считать принятую систему обозначений достаточно приемлемой.

Есть у состояния еще одна, неявная, компонента — память, расчленяемая на ячейки. По сути дела, имена — это имена ячеек, а значения — то, что в этих ячейках хранится. Иногда о ячейках нам придется вспоминать, но никаких связанных с ними формальных средств, кроме значений, их имен и типов, не требуется.

Функции w и w_1 назовем эквивалентными и будем записывать это в виде $w = w_1$, если для любой простой переменной $v \in \mathrm{VarName}$ имеет место эквивалентность $w(nm(v)) = w_1(nm(v))$. Как станет ясно из следующего раздела, отсюда вытекает такая же эквивалентность значений для любой переменной v.

3.3.3. Объявления, генераторы и присваивания

Исполнение объявления $typ\ v$, где $typ \in \mathrm{Type}$, $v \in \mathrm{VarName}$, заключается в следующем. Переменной v приписывается тип typ, тем самым она причисляется к классу $dvars$ объявленных переменных и создается возможность использовать идентификатор v в качестве имени. Пока что по этому имени открыт доступ к неопределенному значению $val?(typ)$ типа typ (точнее, к ячейке, где оно хранится). В дальнейшем программа имеет возможность неоднократно изменить это значение (при присваиваниях) и воспользоваться им (в выражениях) — то и другое по имени v. Открывается доступ и к подобъектам этого значения, и к предметам ссылок, входящих в его состав, — это уже по более сложно устроенным именам.

На денотационном уровне все сказанное описывается формулой

$$\mathrm{Sem}(typ\ v)(\langle w,\ wt\rangle) = \langle w\{val?(typ)/nm(v)\},\ wt\{typ/v\}\rangle\,. \qquad (7)$$

Замечание. За класс $dvars$ объявленных в состоянии $\langle w,\ wt\rangle$ переменных следует принять множество $\mathrm{Var} \cap \mathrm{Dom}(wt)$, которое автоматически расширяется при доопределении функции wt в точке v.

Выше, в конце разд. 3.3.2, было сказано, что внутри функции w связь имени v со значением переменной v идет через вновь созданную, но явно не упоминаемую ячейку памяти. Это можно было бы отразить и явно в денотационной семантике объявлений примерно следующим образом.

Введем функцию GetCell : Type → Ref (ее зависимость от текущего состояния памяти явно не упоминается), которая находит в памяти свободную ячейку, достаточную по своему размеру для размещения значения типа *typ*, и возвращает адрес ячейки (ссылку на нее) в качестве своего результата. Через эту ссылку неявно пролегает путь от имени $nm(v)$ к значению $w(v)$ не только для объекта v, но и для его подобъектов.

Все объявления исполняются в начале исполнения блока. Поэтому функция nm в формуле (7) — это функция, уже преобразованная в соответствии с этими объявлениями.

Замечание. Методы реализации функции GetCell (динамического распределения памяти), конечно, поддаются формальному описанию, но это другая тема, далекая от проблем, которые сейчас нас интересуют. Многое здесь сильно зависит от свойств машины, на которой проводится реализация. Поэтому и было решено оставить скрытыми под капотом машины ряд особенностей ее работы и ограничить описание семантики объявлений формулой (7).

Содержательная семантика оператора $S \approx new(pv)$, где тип pv — это **ref** *typ*, такова. Создается переменная типа *typ* с неопределенным значением. Имя этой переменной (фактически, — ячейки, вновь выделяемой в памяти машины, не совпадающей и не пересекающейся с другими ячейками) — это ссылка, отличающаяся от всех уже используемых в текущем состоянии. Это имя присваивается переменной pv и автоматически включается в множество Ref.

Переменная, создаваемая генератором, называется *динамической* в отличие от *статических* переменных, создаваемых при исполнении объявлений.

Основное различие между ними заключается в том, что статическая переменная создается лишь один раз при исполнении блока, в котором она объявлена, а генератор в своем блоке может исполняться многократно, порождая каждый раз новую динамическую переменную. Правда имена этих переменных присваиваются всегда одному и тому же указателю pv, но обычно их нужно сохранить, поэтому они тут же копируются — присваиваются другим указателям, входящим компонентами в создаваемую по ходу исполнения программы динамическую (растущую, меняющуюся) структуру, — отсюда и термин.

Соответственно, семантика оператора S определяется так:

$$\begin{aligned} &\mathrm{Sem}\,(new(pv))(wwt) = wwt_1 \\ &\quad\equiv\ \exists\, typ \exists\, y\,(wt(pv) = \textbf{ref}\ typ \wedge\ y = \mathrm{GetCell}(typ)) \\ &\qquad\wedge\ wwt_1 = \langle w\{y/nm(pv)\}\{val?(typ)/y\},\, wt\rangle\,)\,. \end{aligned} \tag{8}$$

Замечание. Здесь обращение к функции GetCell для создания новой ячейки и ссылки на нее не скрывается. Но именно такова и содержательная семантика генератора.

Семантика оператора присваивания $S \approx v := e$, где $v \in \mathrm{Var}$, $e \in \mathrm{Expr}$, должна учитывать возможность присваивания переменным значений любых типов, но, конечно, с учетом описания переменной. Сначала находится имя y переменной v, затем — значение *val* выражения e. Если хотя бы одно из них не определено, или если типы переменной v и выражения e не совпадают, то оператор S не исполни́м. В противном случае значение *val* становится доступным под именем y.

Точнее — не само значение val, а его копия. Если e — это переменная, то она продолжает сохранять прежнее значение, а ссылка y впредь именует другое, эквивалентное ему значение. На уровне ссылок это копирование обрывается — ссылки копируются, а их предметы — нет.

Изменение значения подобъекта частично меняет и значение надобъекта, но не затрагивает другие, параллельные, подобъекты этого надобъекта. Изменение значения надобъекта может полностью изменить значения всех его подобъектов.

Денотационная семантика оператора S такова:

$$\begin{aligned}&\mathrm{Sem}(v := e)(wwt) = wwt_1\\ &\quad \equiv\ \exists y\,(y = nm(e) \wedge \exists val\,(val = e(w) \wedge wwt_1 = \langle w\{val/y\},\, wt\rangle\,)).\end{aligned} \tag{9}$$

Операция $w\{val/y\}$ сохраняет свой смысл — изменить функцию w с учетом того, что y впредь будет именем значения val. Но теперь уже нельзя утверждать, что значение $w\{val/y\}(y1)$ всегда равно $w(y1)$, если $y1 \neq y$.

Результат исполнения операции $w\{val/y\}$ опишем полуформально. Основное ее свойство, выраженное равенством $w\{val/y\}(y) = val$, не изменяется. Свойство $w\{val/y\}(y1) = w(y1)$ при $y1 \neq y$ сохраняется, если объект (с именем) $y1$ не является ни подобъектом, ни надобъектом по отношению к y. Если же $y1$ — подобъект y, то $w\{val/y\}(y1) = val1$, где $val1$ — подобъект значения val, занимающий в его структуре место, аналогичное месту подобъекта $y1$ в структуре y. Если $y1$ — надобъект по отношению к y, то изменение $w\{val/y\}(y1)$ по сравнению с $w(y1)$ затрагивает только подобъект y и его подобъекты, но не касается других подобъектов объекта $y1$.

В случае $wt(e) \neq wt(v)$ функция $\mathrm{Sem}(v := e)$ не определена. Из этого правила есть единственное исключение, когда $w(e) = \mathbf{nil}$, т. е. $wt(e) \approx \mathbf{ref}$. При этом требуется только, чтобы тип $wt(v)$ принадлежал классу Pointer.

Впрочем, при определении функции wt от аргумента **nil** в разд. 3.3.2 было принято соглашение о наращивании символа **ref** некоторым типом typ. Именно здесь оно и срабатывает. Тип typ извлекается из типа $wt(v) \approx \mathbf{ref}\ typ$. Таким способом и это исключение устраняется.

Когда исполняется оператор $v := e$, где v и e имеют тип **ref** typ, то, как следует из формулы (9), имя $nm(v)$ начинает именовать значение выражения e — ссылку на некоторый предмет типа typ. Тем самым открывается путь доступа к этому предмету без его перевычисления. Но при этом теряется прежнее значение переменной v — ссылка (возможно, единственная) на другой, прежний, предмет ссылки того же типа typ.

Таким образом, формула (9) полностью соответствует изложенной выше операционной семантике операторов присваивания, а также отражает содержательную семантику как статических, так и динамических переменных (см. выше).

В случае оператора $v := e\uparrow$, где v имеет тип typ, а e — **ref** typ, переменной v присваивается значение типа typ, предмет ссылки e. Это подозрительно напоминает прямое присваивание переменной v значения типа typ оператором $v := e'$, где e' — выражение этого типа. Подробнее это сходство будет разобрано в разд. 3.3.5.

Лемма 3.2.2 о детерминированности семантики оператора присваивания остается в силе.

3.3.4. Блоки

При исполнении объявлений, помещенных в начале блока, меняются на время исполнения блока состав переменных, доступных программе, и — в отличие от разд. 3.2 — их типы. Когда исполнение блока завершится, следует восстановить и то и другое. При этом значения глобальных переменных, приобретенные ими в блоке, должны быть сохранены.

В разд. 3.2.5 эти задачи (кроме действий с типами) решались достаточно простым и эффективным способом, основанным на совпадении множества всех переменных с множеством простых (в теперешней терминологии) переменных.

Сейчас переменные могут именоваться с привлечением операций индексации, выделения и разыменования и создаваться при исполнении не только объявлений, но и генераторов.

Однако базой для формирования имен переменных продолжают служить простые (объявленные) переменные. Следовательно, можно действовать так же, как прежде, — при входе в блок запомнить, а при выходе — восстановить состояние wwt (обе его компоненты) в части, относящейся к переобъявленным переменным, вместе с областью определения $\mathrm{Dom}(wt)$.

Задачу можно решить совсем просто, если махнуть рукой на засорение памяти уже ненужными значениями. Для этого достаточно описать семантику блока так:

$$\mathrm{Sem}(\textbf{begin}\ S\ \textbf{end})(wwt) = \mathrm{SemIn}(S,\ \emptyset,\ wwt)(wwt),$$

где SemIn, а за ней SemOut — семантические функции, похожие на те, что были описаны в разд. 3.2.5:

$$\begin{aligned}
&SemIn(typ\ v;\ S',\ dvars,\ wwt_0)(wwt)\\
&\quad = \mathrm{SemOut}(v,\ wwt_0)(\mathrm{SemIn}(S',\ dvars \cup \{v\},\ wwt_0)(\mathrm{Sem}(typ\ v)(wwt)),\\
&\qquad \text{если } v \notin dvars;\\
&\forall\, wwt'\ SemIn(typ\ v;\ S',\ dvars,\ wwt_0)(wwt) \neq wwt', \quad \text{если } v \in dvars;\\
&\mathrm{SemIn}(S1,\ dvars,\ wwt_0)(wwt) = Sem(S1)(wwt),\\
&\quad \text{если } S1 \not\approx typ\ v;\ S' \quad \text{при любых } typ, v \text{ и } S';\\
&\mathrm{SemOut}(v,\ wwt_0)(wwt)\\
&\quad = \begin{cases} \langle w\{w_0(v)/nm(v)\},\ wt\{wt_0(v)/v\}\rangle, & \text{если } v \in \mathrm{Dom}(wt_0),\\ \langle w \setminus \langle nm(v),\ w(nm(v))\rangle,\ wt \setminus \langle v,\ wt(v)\rangle\rangle, & \text{если } v \notin \mathrm{Dom}(wt_0). \end{cases}
\end{aligned}$$

Именно в этих описаниях интенсивно используется функция wt в составе аргумента преобразователей состояний.

Если же программа активно строит динамические структуры (см. предыдущий раздел), которые со временем становятся ненужными для ее дальнейшей работы, то хотя бы иногда следует заняться так называемой *сборкой мусора*.

Изложим принципы организации этого процесса, не стремясь объять необъятное.

Применяется лишь один из этих принципов — значение заведомо не нужно программе, если оно ей недоступно. Другие критерии ненужности вéдомы только автору программы и скрыты от машины за семью печатями.

Переменная v из множества Var доступна при данных функциях w, wt (и nm), если в этом состоянии она имеет имя $nm(v)$, построенное по правилам из разд. 3.3.1 и 3.3.2. Вкратце: базой имени должна служить простая переменная v, принадлежащая области $\mathrm{Dom}(wt)$, для индексации используются целые числа от 1 до границы индексов вектора, имя — значение указателя доступно (через функцию w) по имени самого указателя.

Если доступен объект, то доступны как все его подобъекты, так и предметы всех содержащихся в нем подобъектов-ссылок. Но если объект недоступен, то некоторые его подобъекты могут быть, тем не менее, доступными как предметы ссылок из других объектов.

Выход из блока — удобный момент для сборки мусора, так как база для построения имен сокращается, что и приводит к появлению недоступных программе значений (есть, впрочем, и другие источники появления таких значений, в частности, потеря ссылок при присваивании новых значений указателям, см. предыдущий раздел).

Необходима функция Dispose, которая строит новую функцию w (преобразует текущее состояние памяти) так, чтобы оставались неизменными значения всех переменных из области $\mathrm{Dom}(wt)$, и т о л ь к о их.

Семантика блока должна предусматривать обращение к функции Dispose для корректировки функции w с соблюдением указанного требования:

$$\mathrm{Sem}(\textbf{begin } S \textbf{ end})(wwt) = \mathrm{Dispose}(\mathrm{SemIn}(S, \varnothing, wwt)(wwt)).$$

3.3.5. Простые переменные как указатели

Понятие переменной имеет много общего с понятием указателя — оба объекта связаны со ссылкой, открывающей доступ к значению некоторого фиксированного, но произвольного типа. Эта ссылка явно упоминается в содержательной семантике указателя и скрыта в реализации функции w для обычной переменной (см. разд. 3.3.3). Покажем, что эти понятия можно сблизить еще больше.

Т е о р е м а 3.3.1. *Пусть в начале некоторого блока программы S содержится объявление typ v, а затем до конца блока следуют (среди прочих) операторы присваивания вида $v := e$. Программа S' получается из программы S в результате замены в этом блоке объявления typ v на* **ref** *typ v, включения в нее между объявлениями и собственно телом блока оператора new(v) и замены всех вхождений переменной v в левые части операторов присваивания и в любые выражения на $v\uparrow$. Тогда при исполнении обеих программ в одинаковых условиях*: а) *исполнение любого их оператора, кроме упомянутых операторов объявления, завершается при эквивалентных значениях переменных, если для S' (в отличие от S) под значением переменной v понимать значение $v\uparrow$*; б) *вычисление любого выражения дает эквивалентные результаты.*

Замечания. 1. Имеется в виду, что между операторами программ установлено естественное соответствие — каждый оператор программы S' соответствует тому оператору программы S, из которого он получен, генератор $new(\mathrm{v})$ — подразумеваемому в S пустому оператору. Одинаковые условия исполнения предполагают одинаковые начальные состояния и одинаковые значения, поставляемые соответственными операторами ввода.

2. Для программы S' значение (в обычном понимании) переменной v — это ссылка. Предмет этой ссылки, как утверждает теорема, совпадает со значением переменной v в программе S. Нижеследующее доказательство лишь уточняет детали обоснования этого утверждения. В этом и состоит обещанное сближение понятий переменной и соответствующего ей в программе S' указателя.

Доказательство теоремы проведем индукцией по суммарному числу N исполненных операторов и вычисленных выражений. При $N = 0$, т. е. перед началом работы, утверждение теоремы очевидно верно. Состояния после исполнения структурированных операторов можно не сравнивать, так как исполнение любого из них завершается вместе с исполнением одного из его подоператоров, а выбор исполняемого подоператора зависит только от значения логического выражения b в составе условного оператора или цикла.

Предположим, что все уже исполненные операторы и уже вычисленные выражения дали результаты в соответствии с утверждением теоремы.

Сравним состояния обеих программ после исполнения объявлений *typ* v и **ref** *typ* v соответственно (см. разд. 3.3.3). В программе S переменной v будет сопоставлена неявная ссылка y на вновь выделенную ячейку, предназначенную для хранения значения типа *typ*, в программе S' — явная ссылка $y1$ на ячейку под значение типа **ref** *typ*. После исполнения всех объявлений и до начала исполнения собственно тела блока программа S' исполняет включенный в нее генератор $new(v)$, а программа S ничего не делает (исполняет пустой оператор). Поскольку в S' тип переменной v — это **ref** *typ*, то генератор $new(v)$ выделяет ячейку под значение типа *typ* и ссылку y' на эту ячейку помещает в ячейку по ссылке $y1$. Ссылка y' будет теперь явно использоваться для той же цели, что и ссылка y в программе S. Как y, так и y' ссылаются пока что на неопределенное значение и пригодны лишь для того, чтобы оператор присваивания вида $v := e$ для программы S и $v\uparrow := e'$ для S' или же оператор ввода ($Read(v)$ и $Read(v\uparrow)$ соответственно) поместил по этой ссылке конкретное значение типа *typ*. До тех пор переменная v не имеет права входить ни в одно вычисляемое выражение.

Первичными компонентами вычислимого выражения могут быть константы (целые, *true*, *false*, **nil** — их значения не зависят от программы) и простые переменные (из класса VarName), уже получившие определенные значения, эквивалентные друг другу по предположению индукции. Простые переменные могут быть наращены (если их тип позволяет) цепочками из индексных выражений, выделителей полей и символов $\uparrow$. По определению эквивалентности значений каждое звено такой цепочки ведет для обеих программ к эквивалентным значениям. Операции со знаками $+$, $<$, $\wedge$ и др. из множества OpSym могут применяться только к примитивным значениям. Их эквивалентность, по тому же

определению, подразумевает равенство, так что и результаты оказываются равными. Таким образом, утверждение теоремы оказывается справедливым после вычисления очередного выражения, каков бы ни был его вид.

Операторы присваивания переменным v (для S) и $v\uparrow$ (для S') помещают эквивалентные значения по ссылкам y и y' (см. выше). Эти значения становятся значениями названных переменных (см. разд. 3.3.2). Это относится и к операторам ввода (см. замечание 1).

Другие неструктурированные операторы не меняют значений переменных. Итак, утверждение теоремы остается верным после завершения исполнения любого оператора. $\triangleleft$

Следствие 3.3.2. *Исполнение обеих программ либо одновременно прерывается из-за невозможности исполнить очередной шаг, либо одновременно завершается, либо не завершается никогда.*

Следствие 3.3.3. *Теорема справедлива, если описанное в ней преобразование программы выполнено для любого множества переменных в любом числе блоков.*

Если его выполнить для всех переменных программы, то символ $\uparrow$ непосредственно за каждым вхождением любой переменной в текст программы S' становится неизбежным и можно условиться его не писать. Внешнее отличие программы S' от S заметно уменьшается. Так сделано, в частности, в Алголе 68.

Следствие 3.3.4. *Утверждение теоремы верно для выражений, одинаково построенных из переменных программ S и S' (с заменой v на $v\uparrow$), даже если эти выражения не входят в тексты программ, например, для пред- и постусловий.*

3.3.6. Динамические типы

Во многих алгоритмических языках в той или иной форме допускаются переменные, которые во время исполнения программы могут получать значения разного типа, — переменные с *динамическим типом* значений. Иногда разнообразие типов значений переменной ограничивается и контролируется, как, например, в «Паскале» (тип записей с вариантами). Но бывают и языки, где любой переменной может быть присвоено значение любого типа, — например Лисп (см. разд. 1.7.2). Чтобы программа имела возможность узнать, как этим значением можно распорядиться, сведения о типе текущего значения переменной следовало бы хранить вместе с самим значением. В обоих названных языках это обеспечивается, в «Паскале» — наличием селектора варианта, в Лиспе, где тип значения — это его списочная структура плюс типы атомов, — встроенными в язык предикатами. Однако ответственность за использование значений в согласии с их типами целиком ложится на программиста.

Рассмотрим вариант, в котором система типов остается прежней с одним лишь отличием в определении класса Pointer:

Pointer ::= **ref**.

Соответственно, объявление (задание типа) указателя pv имеет вид **ref** pv. Любой указатель, будь то самостоятельная переменная, элемент вектора или поле записи, может ссылаться на значение любого типа, и ссылка на такое значение в любой момент может быть ему присвоена.

Всем прочим переменным — статическим или динамическим, с примитивным или составным значением — может быть присвоено значение только предписанного им типа. Тип динамической переменной определяется при возникновении этой переменной. Для этого генератор должен иметь вид $new(pv, typ)$ вместо $new(pv)$, так как тип значения, на которое может ссылаться указатель pv, теперь не определен заранее.

Второй аргумент typ генератора $new(pv, typ)$ — это типовое выражение. Им может быть любой элемент синтаксического класса Type, определенного в разд. 3.3.1. Класс этот строится в направлении от простых типов к более сложным. Но иногда требуется двинуться в противоположном направлении — от типа составного значения к типу его компонент, возможно на несколько шагов. Для этого служит типовое выражение $as\ v$. Его значение — тип переменной v.

При исполнении операций типы операндов должны контролироваться. Например, складывать можно только два значения типа **int** и сумма также получает тип **int**. Значение типа **ref**, т. е. ссылку, можно только разыменовывать — обращаться к предмету этой ссылки. Значение типа **vector**$[n]$ **of** typ' можно только индексировать, при этом значение индекса должно лежать в пределах от 1 до n. Тип результата в последних двух случаях извлекается из памяти вместе с собственно значением. При этом для элемента вектора он должен совпадать с typ'. Аналогично трактуются значения-записи.

Такой контроль необходим как при статических, так и при динамических типах значений. В первом случае он часто может быть выполнен статически (но не всегда — значение индекса, например, становится известным лишь во время исполнения программы). Во втором — его почти неизбежно следует осуществлять динамически. Если контроль типов осуществляется машиной, то труд программиста заметно облегчается, а текст программы сокращается. Для этого в денотационной семантике должны быть зафиксированы правила контроля. Кое-что в этом направлении уже было сделано в предыдущих разделах, что-то добавится ниже, но подробно описывать механизм контроля не будем.

Вкратце изложим денотационную семантику языков с динамическими типами. Состояние — это по-прежнему функция w $(w_1, \ldots)$, определяющая значение любой переменной v по имени $nm(v)$. Но теперь значение $w(nm(v))$ должно быть парой $\langle val, typ\rangle$, включающей в себя собственно значение val и его тип typ. Таким образом, тип состояния — Name $\to$ Val $\times$ Type.

Проекционные функции $v\ell$ и wt' извлекают из значения $w(nm(v))$ любой переменной v его компоненты val и typ. Функции wv : Name $\cup$ Expr $\to$ Val (аналог прежней функции w) и wt : Expr $\to$ Type строятся методом регулярного расширения исходя из формул

$$wv(e) = e(w),$$

$v(w) = v\ell(w(nm(v)))$ для $v \in \mathrm{Var}$,
$wt(v) = wt'(w(nm(v)))$ для $v \in \mathrm{Var}$.

Остальные формулы берутся из разд. 3.3.2 без изменения.

Обозначения $e(w)$ и $wt(e)$ родственны обозначениям из разд. 3.2.3 – 3.2.5, 3.3.2 и 3.3.4. Благодаря этому семантические правила занимают промежуточное положение между правилами из этих разделов, ближе к первым. Наиболее существенные из них представлены следующими ниже формулами.

$$\mathrm{Sem}(typ\ v)(w) = w\{\langle val?(typ),\ typ\rangle / nm(v)\}.$$

Самостоятельного значения эта формула, как и ее аналог в начале п. 9 разд. 3.2.5, не имеет. Но детальнее семантику блоков здесь рассматривать не станем.

Содержательная семантика генератора $new(pv, typ)$ описана выше. Ей соответствует следующее семантическое правило:

$$\mathrm{Sem}(new(pv,\ typ))(w) = w_1$$
$$\equiv \exists y\,(y = \mathrm{GetCell}(typ, w) \wedge w_1 = w\{\langle y, \mathbf{ref}\rangle / pv\}\{\langle val?(typ), typ\rangle / y\}).$$

Семантика оператора присваивания:

$$\mathrm{Sem}(v := e)(w) = w_1$$
$$\equiv \exists y\,(y = nm(v) \wedge \exists\, val\,(val = e(w) \wedge wt(v) = wt(e) \wedge w_1 = w\{val/y\})).$$

Имя $nm(v)$ начинает именовать уже сформированное значение $e(w)$ правой части, включающее в себя и тип. Проверяется совпадение типов переменной v (статической или динамической) и выражения e. При этом, если v — указатель, то и e должно быть указателем, но типы предметов ссылок неизвестны, а потому и не сравниваются.

Замечание. Теорема 3.3.1 теряет свой прямой смысл, так как объявление **ref** *typ* v перестало быть синтаксически правильным. Однако, усложнив семантические правила (скомбинировав правила из разд. 3.3.3 с правилами данного раздела), можно разрешить применять в языке объявления как такого вида, так и вида **ref** v — соответственно для переменных со статическими или динамическими типами значений. Это повысило бы гибкость языка.

3.3.7. Преобразователи предикатов для присваивания

Продолжим начатое в п. 6 разд. 3.2.7 построение прямого и обратного преобразователей предикатов для оператора присваивания, но уже для языка с составными значениями. Не гонясь за полной общностью, будем иметь дело с объектами двух типов: простыми целочисленными переменными и векторами с целочисленными элементами. Именно векторы порождают различие между программными именами и именами в узком смысле. Последствия этого различия и должны быть рассмотрены.

Идентификаторы простых переменных составляют класс VarName, а идентификаторы векторов — класс Vect $= \{ar, \ldots\}$. Синтаксический класс IV определим теперь так:

IV ::= VarName | Vect [IE].

Доопределим операцию $e_1[e/v]$ правилами для случаев, когда e_1 — это переменная с индексом (при этом $ar \in \mathrm{Vect}$):

4в) $ar[e'][e/v] \approx ar[e'[e/v]]$,
если $v \in \mathrm{VarName}$ или $v \approx ar1[e1]$, где $ar1 \not\approx ar$,

4г) $ar[e'][e/ar[e1]] \approx$ **if** $e'[e/ar[e1]] = e1$ **then** e **else** $ar[e'[e/ar[e1]]]$ **fi**.

Если в случае 4в) решение о подстановке по-прежнему принимается статически, то в случае 4г) статически отождествить переменные $ar[e']$ (с подстановкой e вместо $ar[e1]$ в индексное выражение e') и $ar[e1]$ невозможно. Использованное выражение делает это динамически — во время исполнения программы.

Имя $nm(v)$ переменной v строится по следующим упрощенным правилам:

$nm(v) = v$, если $v \in \mathrm{VarName}$,
$nm(ar[e]) = ar[e(w)]$, если $ar \in \mathrm{Vect}$, $e \in \mathrm{IE}$.

При этом множество Name имен в узком смысле задается метаформулой

Name ::= VarName | Vect [N].

Теперь это множество составляет область отправления любого состояния: $w : \mathrm{Name} \to \mathrm{Z} \cup \mathrm{BV} \cup \{d?\}$. Значение переменной v в произвольном состоянии w вычисляется по прежней формуле

$$v(w) = w(nm(v))$$

или, с учетом правил построения имени $nm(v)$:

$v(w) = w(v)$, если $v \in \mathrm{VarName}$,
$ar[e](w) = w(ar[e(w)])$.

Правила регулярного расширения обозначения $e(w)$ для значения выражения e в состоянии w сохраняются.

Семантику оператора $S \approx v := e$, где $v \in \mathrm{IV}$, исполняемого из состояния w, при обозначениях $d = e(w)$, $y = nm(v)$ можно коротко записать в виде

$$s(w) = w\{d/y\}.$$

Лемма 3.3.5. *Пусть* $v \in \mathrm{IV}$, $e \in \mathrm{IE}$, *оператор* $S \approx v := e$ *исполняется из состояния* w *и* $d = e(w)$, $y = nm(v)$. *Тогда для произвольного выражения* e'

$$e'[e/v](w) = e'(w\{d/y\}). \tag{10}$$

Доказательство получаем, продолжая доказательство леммы 3.2.10 с одним изменением — вместо $w\{d/v\}$ должно стоять $w\{d/y\}$. К случаям а–г добавляются еще два случая.

д′) В роли e' выступает $ar[e']$, в роли v — переменная из VarName или $ar1[e1]$, где $ar1 \not\approx ar$. При этом

$ar[e'][e/v](w) = ar[e'[e/v]](w) = w(ar[e'[e/v](w)])$,
$ar[e'](w\{d/y\}) = w\{d/y\}(ar[e'(w\{d/y\})])$.

По предположению индукции $e'[e/v](w) = e'(w\{d/y\})$, так что $ar[e'[e/v](w)]$ и $ar[e'(w\{d/y\})]$ — это одно и то же имя $y1$. Но $y = nm(v)$ — это либо идентификатор v, либо имя вида $ar1[d1]$, так что $y \not\approx y1$, поэтому $w\{d/y\}(y1) = w(y1)$, т. е. формула (10) верна и в этом случае.

д″) В роли e' снова $ar[e']$, но $v \approx ar[e1]$ — последний и наиболее громоздкий случай.

Обозначим $d1 = e1(w)$, $d2 = e'(w\{d/y\})$. Из предположения индукции следует $e'[e/ar[e1]](w) = e'(w\{d/y\}) = d2$, причем $y = nm(ar[e1]) = ar[e1(w)] = ar[d1]$.

Преобразуем левую часть формулы (10):

$$\begin{aligned} & ar[e'][e/ar[e1]](w) \\ & \quad = (\textbf{if } e'[e/ar[e1]] = e1 \textbf{ then } e \textbf{ else } ar[e'[e/ar[e1]]] \textbf{ fi})(w) \\ & \quad = \begin{cases} e(w), & \text{если } (e'[e/ar[e1]] = e1)(w) = true, \\ ar[e'[e/ar[e1]]](w) & \text{— в противном случае.} \end{cases} \end{aligned}$$

Но $(e'[e/ar[e1]] = e1)(w) = true \Leftrightarrow e'[e/ar[e1]](w) = e1(w) \Leftrightarrow d2 = d1$. Итак, при $d2 = d1$ левая часть формулы (10) имеет значение $e(w)$, т. е. d, а при $d2 \neq d1$ — значение $ar[e'[e/ar[e1]]](w) = w(ar[e'[e/ar[e1]](w)]) = w(ar[d2])$.

Займемся правой частью:

$$\begin{aligned} & ar[e'](w\{d/y\}) = w\{d/y\}(ar[e'(w\{d/y\})]) = w\{d/y\}(ar[d2]) \\ & \quad = \begin{cases} d & \text{при } y = ar[d2], \text{ т. е. при } d1 = d2, \\ w(ar[d2]) & \text{при } d1 \neq d2. \end{cases} \end{aligned}$$

Итак, и в этом случае значения обеих частей формулы (10) совпадают. $\triangleleft$

Теорема 3.3.6 [Хоор]. *Пусть $v \in$ IV, $e \in$ IE. Тогда для оператора $S \approx v := e$ обратный преобразователь предикатов $s{\downarrow}$ задается выражением*

$$s{\downarrow}(q) \approx q[e/v].$$

Доказательство базируется на предыдущей лемме и полностью аналогично доказательству теоремы 3.2.11. $\triangleleft$

Как и в разд. 3.2.7, будем пользоваться немыми переменными из класса DV со следующими дополнениями. Если p не содержит ни i, ни j, то выражение $p[i/ar[j]]$ принадлежит классу BE. Выражения вида $p[i/ar[j]]$, $e[i/ar[j]]$ или $v[i/ar[j]]$ будем допускать только в составе логических выражений вида $\exists\, i\, \exists\, j\, q$.

Выражение $(\exists\, i\, \exists\, j\, q)(w)$ трактуется аналогично выражению $(\exists\, i\, q)(w)$.

Теорема 3.3.7 [де Баккер]. *Пусть $v \in$ IV, $e \in$ IE и $S \approx v := e$. Тогда в прямом преобразователе предикатов $s{\uparrow}$ для оператора S предикату p соответствует предикат q, который образуется из p по правилам*:

а) *если $v \in VarName$, то $q \approx \exists\, i\, (p[i/v] \wedge v = e[i/v])$,*

б) *если $v \approx ar[e']$, то*

$$q \approx \exists\, i\, \exists\, j\, (p[i/ar[j]] \wedge ar[j] = e[i/ar[j]] \wedge j = e'[i/ar[j]]).$$

Пункт а) — это теорема 3.2.12 и в доказательстве уже не нуждается.

Пункт б) имеет следующий содержательный смысл. Если переменной с индексом $ar[e']$ присваивается значение выражения e, то найдутся такие значения j индексного выражения e' (номер компоненты, которой делается присваивание) и i — самóй компоненты $ar[j]$ вектора ar (оба значения — до присваивания), при которых p истинно, а e имеет значение, присваиваемое этой компоненте.

Доказываем этот пункт теоремы, основываясь на прежней идее и следуя, в основном, прежней схеме. Доказываемое утверждение теперь записывается так:

$$q(w_1) \;\equiv\; \exists\, w\,(p(w) \wedge\; w_1 = w\{e(w)/nm(v)\}). \qquad (11)$$

При этом $q \;\approx\; \exists\, i\, \exists\, j\,(p[i/ar[j]] \wedge\; ar[j] = e[i/ar[j]] \wedge\; j = e'[i/ar[j]])$ по условию пункта б). Поскольку $v \;\approx\; ar[e']$, в правой части формулы (11) имя $nm[v]$ — это $ar[e'(w)]$. Далее:

$$\begin{aligned}&(\exists\, i\, \exists\, j\,(p[i/ar[j]] \wedge\; ar[j] = e[i/ar[j]] \wedge\; j = e'[i/ar[j]]))(w_1)\\ &\quad= \exists\, i\, \exists\, j\,(p[i/ar[j]](w_1) \;\wedge\; ar[j](w_1) = e[i/ar[j]](w_1)\\ &\qquad\wedge\; j(w_1) = e'[i/ar[j]](w_1)\;).\end{aligned}$$

На основании формулы (10):

$$\begin{aligned}&p[i/ar[j]](w_1) = p(w_2), \quad e[i/ar[j]](w_1) = e(w_2),\\ &e'[i/ar[j]](w_1) = e'(w_2), \quad \text{где } w_2 = w_1\{i(w_1)/nm(ar[j])\}.\end{aligned}$$

Кроме того, $i(w_1) = i$, $j(w_1) = j$, $nm(ar[j]) = ar[j(w_1)] = ar[j]$, и следовательно, $w_2 = w_1\{i/ar[j]\}$; $ar[j](w_1) = w_1(ar[j])$. Таким образом, доказывать теперь надо справедливость формулы

$$\begin{aligned}&\exists\, i\, \exists\, j\, \exists\, w\,(w = w_1\{i/ar[j]\} \wedge\; p(w) \wedge\; w_1(ar[j]) = e(w) \wedge\; j = e'(w))\\ &\qquad\equiv\; \exists\, w\,(p(w) \wedge\; w_1 = w\{e(w)/ar[e'(w)]\}).\end{aligned} \qquad (12)$$

В правую сторону: из $w = w_1\{i/ar[j]\} \;\wedge\; w_1(ar[j]) = e(w) \;\wedge\; j = e'(w)$ следует, что $w_1(ar[e'(w)]) = e(w)$ и $w_1(v') = w(v')$ при $v' \;\not\approx\; ar[e'(w)]$, т. е. что $w_1 = w\{e(w)/ar[e'(w)]\}$. Поэтому

$$\begin{aligned}&\exists\, i\, \exists\, j\, \exists\, w\,(w = w_1\{i/ar[j]\} \wedge\; p(w) \wedge\; w_1(ar[j]) = e(w) \wedge\; j = e'(w)\;)\\ &\quad\Rightarrow\; \exists\, i\, \exists\, j\, \exists\, w\,(p(w) \wedge\; w_1 = w\{e(w)/ar[e'(w)]\})\\ &\quad\Rightarrow\; \exists\, w\,(p(w) \wedge\; w_1 = w\{e(w)/ar[e'(w)]\}).\end{aligned}$$

В левую сторону: если для неизвестных в состоянии w_1 значений ввести обозначения $e'(w) = j$, $ar[j](w) = i$, где $i, j \;\in$ DV, i и j не входят ни в p, ни в e, ни в e', то из $w_1 = w\{e(w)/ar[e'(w)]\}$ следует $w_1(ar[j]) = e(w)$, $w = w_1\{i/ar[j]\}$, откуда

$$\begin{aligned}&\exists\, w\,(p(w) \wedge\; w_1 = w\{e(w)/ar[e'(w)]\})\\ &\quad\supset\; \exists\, i\, \exists\, j\, \exists\, w\,(w = w_1\{i/ar[j]\}\\ &\qquad\wedge\; p(w) \wedge\; w_1(ar[j]) = e(w) \;\wedge\; j = e'(w)\;).\end{aligned}$$

Все вместе это доказывает справедливость формулы (12). $\lhd$

Пример. Пусть $p \;\approx\; (a[1] = 2 \;\wedge\; a[2] = 2)$ и исполняется оператор $S \;\approx\; a[a[2]] := 1$. При этом значение 1 получает переменная $a[2]$, переменная $a[1]$ сохраняет значение 2, так что значение $a[a[2]]$ оказывается равным $a[1]$, т. е. 2, вопреки видимому эффекту исполняемого оператора.

Пропустим этот пример через развитый выше аппарат.

$$\begin{aligned}&s{\uparrow}(p) \;\approx\; \exists\, i\, \exists\, j\,((a[1] = 2 \;\wedge\; a[2] = 2)[i/a[j]]\\ &\qquad\wedge\; a[j] = 1[i/a[j]] \wedge\; j = a[2][i/a[j]]\;)\\ &\quad\approx\; \exists\, i\, \exists\, j\,(\textbf{if } 1[i/a[j]] = j \textbf{ then } i \textbf{ else } a[1[i/a[j]]] \textbf{ fi} = 2[i/a[j]]\\ &\qquad\wedge\; \textbf{if } 2[i/a[j]] = j \textbf{ then } i \textbf{ else } a[2[i/a[j]]] \textbf{ fi} = 2[i/a[j]]\\ &\qquad\wedge\; a[j] = 1\\ &\qquad\wedge\; j = \textbf{if } 2[i/a[j]] = j \textbf{ then } i \textbf{ else } a[2[i/a[j]]] \textbf{ fi}\;)\\ &\quad\approx\; \exists\, i\, \exists\, j\,(\textbf{if } 1 = j \textbf{ then } i \textbf{ else } a[1] \textbf{ fi} = 2\\ &\qquad\wedge\; \textbf{if } 2 = j \textbf{ then } i \textbf{ else } a[2] \textbf{ fi} = 2\end{aligned}$$

$\wedge\ a[j] = 1$
$\wedge\ j = \mathbf{if}\ 2 = j\ \mathbf{then}\ i\ \mathbf{else}\ a[2]\ \mathbf{fi}\ ).$

Предположение, что $j \neq 2$, приводит к противоречию, так как $a[2] = 2$ (из 2-го члена) и $j = a[2]$ (из 4-го члена). Следовательно, $j = 2$ и $s\uparrow(p)$ сводится к $\exists\, i\,(a[1] = 2 \ \wedge\ i = 2 \ \wedge\ a[2] = 1 \ \wedge\ 2 = i)$, т. е. к $a[1] = 2 \ \wedge\ a[2] = 1$, так что из истинности $s\uparrow(p)$ следует $a[a[2]] = 2$.

Это служит иллюстрацией к тому, что после исполнения присваивания $ar[e'] := e$ не всегда можно утверждать, что в полученном состоянии верна формула $\exists\, i\,(ar[e'] = e[i/ar[e']])$, аналогичная формуле Флойда для языка без векторов.

Продолжая пример, зададимся вопросом — в каких случаях после исполнения оператора присваивания $a[a[v]] := v'$, где $v, v' \in \mathrm{VarName}$, не соблюдается равенство $a[a[v]] = v'$? Для этого достаточно найти результат p обратного преобразования предиката $a[a[v]] \neq v'$. Теорема 3.3.6 дает:

$$p \approx (a[a[v]] \neq v')[v'/a[a[v]]] \approx a[a[v]][v'/a[a[v]]] \neq v'[v'/a[a[v]]]$$
$$\approx \mathbf{if}\ a[v][v'/a[a[v]]] = a[v]\ \mathbf{then}\ v'\ \mathbf{else}\ a[a[v][v'/a[a[v]]]]\ \mathbf{fi}\ \neq v'.$$

Но

$$a[v][v'/a[a[v]]] \approx \mathbf{if}\, v[v'/a[a[v]]] = a[v]\ \mathbf{then}\, v'\ \mathbf{else}\, a[v[v'/a[a[v]]]]\ \mathbf{fi}$$
$$\approx \mathbf{if}\, v = a[v]\, \mathbf{then}\, v'\ \ \mathbf{else}\, a[v]\ \mathbf{fi}.$$

Отсюда

$$\begin{aligned} p \approx\ & \mathbf{if}\ \mathbf{if}\ v = a[v]\ \mathbf{then}\ v'\ \ \mathbf{else}\ a[v]\ \mathbf{fi} = a[v]\ \mathbf{then}\ v' \\ & \mathbf{else}\ a[\mathbf{if}\ v = a[v]\ \mathbf{then}\ v'\ \mathbf{else}\ a[v]\ \mathbf{fi}] \\ & \mathbf{fi}\ \neq v' \\ \approx\ & v = a[v] \ \wedge\ \mathbf{if}\ v' = a[v]\ \mathbf{then}\ v'\ \mathbf{else}\ a[v']\ \mathbf{fi}\ \neq v' \\ & \vee\ v \neq a[v] \ \wedge\ \mathbf{if}\ a[v] = a[v]\ \mathbf{then}\ v'\ \mathbf{else}\ a[a[v]]\ \mathbf{fi}\ \neq v' \\ \approx\ & v = a[v] \ \wedge\ v' = a[v] \ \wedge\ v' \neq v' \\ & \vee\ v = a[v] \ \wedge\ v' \neq a[v] \ \wedge\ a[v'] \neq v' \\ & \vee\ v \neq a[v] \ \wedge\ v' \neq v' \\ \approx\ & v = a[v] \ \wedge\ v' \neq v \ \wedge\ a[v'] \neq v'. \end{aligned}$$

Значения $a[1] = a[2] = 2$, $v = 2$, $v' = 1$ из первой части примера как раз удовлетворяют этому предусловию.

3.4. Денотационная семантика процедур и функций

Убедимся, что денотационная семантика вызова процедуры с параметрами-значениями и с параметрами-переменными представляет собой преобразователь, являющийся наименьшей неподвижной точкой некоторого оператора.

3.4.1. Нерекурсивные процедуры и функции

Аппарат процедур будем описывать, отправляясь от языка, рассмотренного в разд. 3.2, — с одним типом значений **int** (целый). Увязка этого аппарата со значениями других типов — векторами, записями и указателями — принципиальных трудностей не вызывает, но загромождает изложение.

Не станем раздельно рассматривать весьма родственные механизмы процедур-функций и собственно процедур, а сразу сведем их в единое понятие.

Синтаксис описания процедуры P представлен схемой

$$\langle P(v, \mathbf{var}\, y)\colon typ,\ S0\rangle,$$

где $P \in$ PName (PName — множество имен процедур), $v, y \in$ IV, $v \not\approx y$, v — (формальный) параметр-значение, y — параметр-переменная, $S0$ — оператор (тело процедуры), который не содержит вызовов ни одной процедуры, и typ — описатель типа значения, вырабатываемого процедурой, $typ \in \{\mathbf{int}, \mathbf{fict}\}$, где **fict** — еще одно значение класса Type, характеризующее процедуры, не вырабатывающие значения (т. е. «обычные»). Разумеется, число параметров каждого класса может быть и не равно 1, но никаких принципиальных изменений это не вносит.

Конструкция вида $P(e, z)$ — это вызов процедуры P, где e — аргумент (фактический параметр), поставляющий значение параметра v, второй аргумент z — это та переменная, которую представляет в теле процедуры параметр y.

Под функцией (случай $typ \approx \mathbf{int}$) понимается процедура, которая в результате исполнения своего тела вырабатывает одно значение определенного типа. Это значение считается основным результатом исполнения вызова функции. Возможное изменение состояния (побочный эффект) не существенно, в отличие от обычной процедуры, и часто считается нежелательным. Однако этот эффект при вызове функций возможен и в строгом описании семантики должен быть учтен, что и будет сделано в данном разделе.

Класс операторов St пополняется конструкцией вида

$$P := e, \qquad \text{где } P \in \text{PName},\ e \in \text{IE}.$$

У этой конструкции много общего с оператором присваивания, но надо помнить, что P — не обычная переменная, а смысл конструкции — указать, что значение выражения e становится (при $typ \approx \mathbf{int}$) значением функции. Оператор вида $P := e$ должен встречаться хотя бы один раз в теле процедуры-функции.

Содержательная семантика оператора $S \approx P(e, z)$ состоит в следующем. В текущем состоянии w вычисляется значение d аргумента e. Создаются псевдопеременная P с неопределенным значением типа typ и переменная v, которой

присваивается значение d. Затем исполняется оператор $S0[z/y]$, получающийся из оператора $S0$ заменой всех вхождений параметра y аргументом z. При этом предполагается, что в операторе $S0$ обозначение z не было локализовано, т. е. объявлено. Если это не так, то перед заменой y на z оно заменяется новым обозначением, отличным от всех других обозначений, действующих в $S0$.

Если оператор $S0[z/y]$ не может быть успешно исполнен, то не исполни́м и оператор S. Если же исполнение оператора $S0[z/y]$ завершается, то в выработанном состоянии переменная v ликвидируется (а если до начала исполнения вызова S процедуры P существовала другая переменная с этим именем, то она восстанавливается).

Для процедуры-функции ее значение при данном вызове находим как значение псевдопеременной P и ликвидируем эту псевдопеременную. Но если при исполнении оператора S не был исполнен оператор вида $P := e'$, то значение выражения $P(e, z)$ в исходном состоянии w не определено.

Полученное состояние считается результатом исполнения оператора S.

Семантически вызов (обычной) процедуры — это разновидность оператора, а вызов функции — разновидность выражения. В этой ситуации, когда одна и та же синтаксическая конструкция — вызов процедуры — может выступать в роли как выражения, так и оператора, надо отождествить эти два семантических класса.

Выберем нейтральный термин — *единица* (программная) для обозначения объединенного понятия — оператор или выражение. В синтаксических конструкциях единицы будем обозначать символами un (вместо S или e), а весь класс программных единиц обозначим Unit (вместо St ∪ Expr).

Мы продолжаем рассматривать состояние w как функцию, сопоставляющую каждой переменной ее значение. Одновременно мы вводим еще одну разновидность действий над переменными — мысленную замену формального параметра-переменной фактическим (тоже переменной).

Чтобы описать семантику замены y на z, вновь введем в рассмотрение именующую функцию nm : IV ∪ PName → Id, где Id — класс идентификаторов, и определим состояние как функцию w : Id → D′. В исходной программе функция nm тождественна (внешне, так как IV, PName и Id — разные синтаксические классы):

$$\forall\, v\, (v \in \text{IV} \cup \text{PName} \supset nm(v) \approx v),$$
$$v(w) = w(nm(v)).$$

Так определенная функция nm инъективна — разным переменным соответствуют разные имена. Это важное свойство должно быть сохранено при любых преобразованиях функции nm.

Функция nm служит неявной компонентой любого состояния w, как это было разъяснено в разд. 3.3.2.

Пусть для переменной v $(x, y, \ldots)$ соответствующее значение функции nm равно $v1$ $(x1, y1, \ldots)$. Как правило, $v1 = v$, но будем рассматривать общий случай. Чтобы при присваивании и при извлечении значения вместо переменной y

в операторе $S0[z/y]$ возникала переменная z, необходим преобразователь состояний $\mathrm{Subst}(z, y, S0)$, который менял бы значение $y1$ функции nm в точке y на $z1$ (аргумент $S0$ указывает контекст, в пределах которого действует эта замена):

$$\begin{aligned}&\mathrm{Subst}(z, y, S0)(w) = w,\\ &\mathrm{SubstNm}(z, y, nm) = nm1\\ &\quad\equiv\ (nm(z) \approx nm(y)) \ \wedge\ nm1 = nm\\ &\qquad\vee\ (nm(z) \not\approx nm(y))\\ &\qquad\quad\wedge\ \exists\, z'\,(z' \notin \mathrm{Rng}(nm) \wedge\ nm1 = nm\{z'/z\}\{nm(z)/y\}\,).\end{aligned}$$

Здесь предусмотрена и упомянутая выше замена обозначения z новым, ранее не встречавшимся в $S0$, обозначением z'. Условие $z' \notin \mathrm{Rng}(nm)$ гарантирует инъективность функции $nm1$.

При принятом способе описания преобразователя Subst текст оператора $S0$ фактически не изменяется. Однако одновременно с обращением к Subst происходит неявное обращение к SubstNm, преобразующее функцию nm — неявную компоненту состояния. Функция nm — это по существу таблица перекодировки программных обозначений переменных в коды (адресá ячеек), по которым функция w находит их значения как для изменения (присваивания), так и для использования. Следующая табличка, где x — любая переменная, отличная от y и от z, это поясняет.

Аргумент v	x	y	z
$nm(v)$	$x1$	$y1$	$z1$
$nm\{z'/z\}(v)$	$x1$	$y1$	z'
$nm\{z'/z\}\{z1/y\}(v)$	$x1$	$z1$	z'

Функция Subst определена однозначно, если избран какой-либо детерминированный метод подбора нового обозначения z'.

При формализации семантики программных единиц возникает специфическая (для математической семантики) трудность. Функция, в ее математическом понимании, не может ни обладать побочным эффектом (изменением состояния), ни двумя значениями сразу (значение d единицы и новое состояние w_1). Можно, конечно, объединить d и w_1 в пару $\langle d, w_1\rangle$, но рано или поздно придется эту пару расчленять и проблема возникнет снова. Выход таков. Для единиц параллельно строятся две последовательности семантических функций. Функции первой последовательности вычисляют значения единиц, второй — новые состояния. Для первых сохраним обозначение $un : \mathrm{W} \to \mathrm{D}'$, за вторыми оставим обозначение Sem с типом $\mathrm{Unit} \to (\mathrm{W} \to \mathrm{W})$.

Дадим новое определение семантики выражений (т. е. единиц, фактически обладающих значением). В этом определении уже нельзя считать, что это значение постоянно вычисляется в одном и том же состоянии w, поскольку вычисление одной из компонент выражения может изменить состояние, в котором должны вычисляться другие компоненты. Состояние, возникшее после вычисления выражения, становится исходным для действия, которое может непосредственно за этим последовать. Итак, формулы первой последовательности:

$$
\begin{array}{l}
d(w) = d,\\
v(w) = w(nm(v)),\\
(e1\,op\,e2)(w) = e1(w)\ Op\ e2(\mathrm{Sem}(e1)(w)),\\
(\neg\, b)(w) = \neg\, b(w),\\
(\mathbf{if}\, b\, \mathbf{then}\, e1\, \mathbf{else}\, e2\, \mathbf{fi})(w) = d\\
\quad \equiv\ b(w)\ \wedge\ d = e1(\mathrm{Sem}(b)(w)) \vee\ \neg\, b(w)\ \wedge\ d = e2(\mathrm{Sem}(b)(w)).
\end{array}
$$

Соответствующее определение семантики выражений (вторая последовательность):

$$
\begin{array}{l}
\mathrm{Sem}(d)(w) = w,\\
\mathrm{Sem}(v)(w) = w,\\
\mathrm{Sem}(e1\,op\,e2)(w) = \mathrm{Sem}(e2)(\mathrm{Sem}(e1)(w)),\\
\mathrm{Sem}(\neg\, b)(w) = \mathrm{Sem}(b)(w),\\
\mathrm{Sem}(\mathbf{if}\, b\, \mathbf{then}\, e1\, \mathbf{else}\, e2\, \mathbf{fi})(w) = w_1\\
\quad \equiv\ b(w)\ \wedge\ w_1 = \mathrm{Sem}(e1)(\mathrm{Sem}(b)(w))\\
\qquad \vee\ \neg\, b(w)\ \wedge\ w_1 = \mathrm{Sem}(e2)(\mathrm{Sem}(b)(w)).
\end{array}
$$

Второе из этих правил выглядит столь просто только потому, что при принятом ограничении переменная v может быть только идентификатором. Если бы было разрешено пользоваться переменными с индексами, а в последних — вызовами функций, то все стало бы сложнее.

Семантика операторов (единиц без значений) в основном остается прежней. Но, в отличие от разд. 3.2.5, в семантике присваивания значения переменной v и вычисления ее значения вместо v появляется имя $nm(v)$:

$$
\begin{array}{l}
\mathrm{Sem}(v := e)(w) = w_1\\
\quad \equiv\ \exists\, d\, (d = e(w) \wedge \exists\, w'\, (w' = \mathrm{Sem}(e)(w) \wedge d \neq d? \wedge w_1 = w'\{d/nm(v)\})).
\end{array}
$$

Семантика присваивания $P := e$ для $P \in \mathrm{PName}$ практически совпадает с этой семантикой и обладает теми же свойствами. Однако, поскольку P — это не обычная переменная, считается, что в теле $S0$ процедуры ее значение не определено:

$$P(w) = d?,$$

даже если присваивание $P := e$ было исполнено. Поэтому P не может быть ни выражением, ни его компонентой.

Семантика вызова грубо может быть представлена следующей последовательностью операторов:

$$v1 := e;\ \mathbf{begin}\, typ\, P;\ \ \mathbf{int}\, v;\ v := v1;\ S0[z/y];\ v1 := P\, \mathbf{end} \qquad (1)$$

Здесь переменная $v1$ (с неявным объявлением $\mathbf{int}\ v1$) использована как для передачи внутрь создаваемого блока значения аргумента e, так и для передачи вовне вычисленного значения функции. Если $typ \approx \mathbf{fict}$, то объявление $typ\ P$ и присваивание $v1 := P$ должны восприниматься как пустые операторы. Присваивание $v1 := P$ — это единственный (и то подразумеваемый) контекст, в котором используется значение псевдопеременной P.

Хотя это описание и не вполне корректно, оно указывает на родство семантики вызовов с семантикой блоков. Для последней в разд. 3.2.5 была принята

схема с парными функциями SemIn и SemOut, позволившая достаточно просто связать между собой действия, выполняемые при входе в блок и при выходе из него для каждого из повторяющихся в нем объявлений переменных. Постараемся и сейчас придерживаться этой схемы, но количество парных семантических функций возрастет:

$$\begin{aligned}&\mathrm{Sem}(P(e,z))(w)\\&\quad=\mathrm{SemReturn}(P,typ,w)\\&\qquad(\mathrm{SemArg}((e,z),(v,\mathbf{var}\,y),S0,w)(w\{val?(typ)/P\})),\end{aligned}\tag{2}$$

$$\begin{aligned}&\mathrm{SemArg}((e,z),(v,\mathbf{var}\,y),S0,w_0)(w)\\&\quad=\mathrm{SemOut}(w,w_0)(\mathrm{SemArg}((z),(\mathbf{var}\,y),S0,w_0)(w\{e(w_0)/nm(v)\})),\end{aligned}\tag{3}$$

$$\begin{aligned}&\mathrm{SemArg}((z),(\mathbf{var}\,y),S0,w_0)(w)\\&\quad=\mathrm{SemArg}((),(),S0,w_0)(\mathrm{Subst}(z,y,S0)(w)),\end{aligned}\tag{4}$$

$$\mathrm{SemArg}((),(),S0,w_0)(w)=\mathrm{Sem}(S0)(w).\tag{5}$$

Здесь формула (3) — это схема определения функции SemArg для случая, когда очередной параметр — это параметр-значение, а формула (4) — то же самое для случая параметра-переменной.

Значение псевдопеременной P в состоянии w_1, полученном в результате обращения к SemArg (и всех рекурсивных обращений), становится значением вызова $P(e,z)$:

$$P(e,z)(w)\;=\;P(\mathrm{SemArg}((e,z),(v,\mathbf{var}\,y),S0,w_0)(w\{val?(typ)/P\})).\tag{6}$$

Преобразователь SemReturn перерабатывает состояние w_1 в окончательное, а также неявно восстанавливает функцию nm:

$$\mathrm{SemReturn}(P,typ,w_0)(w)\;=\;w\setminus\langle P,w(P)\rangle\,.\tag{7}$$

Формула (6) явно представляет на денотационном уровне контекст, в котором правило $P(w_1)=d?$ не действует. Предполагается, что аргументы: typ функции SemReturn, $(v,\mathbf{var}\,y)$ и $S0$ функции SemArg при первом обращении к ней берутся из описания процедуры P. Формула (6) продолжает первую последовательность функций, упомянутых выше, остальные — вторую.

Лемма 3.4.1. *Если семантика оператора $S0$ детерминирована, то и функция $Sem(P(e,z))$ детерминирована. Оператор $P(e,z)$ не изменяет область определения текущего состояния.*

Доказательство подобно доказательству аналогичной леммы 3.2.8 для семантики блоков. $\lhd$

И в заключение — если бы в языке существовало понятие указателя, то можно было бы все переменные, в частности z, трактовать так, как это описано в конце разд. 3.3.5 (следствие 3.3.3), и отказаться от понятия параметра-переменной и всего, что с ним связано (подстановка фактического параметра вместо формального параметра-переменной, функции nm и Subst и пр.). Описание процедуры можно было бы изменить на $\langle P(\mathbf{int}\,v,\,\mathbf{ref\ int}\,y)\colon typ,\ S0\rangle$, переменная z была бы объявлена в основной программе как $\mathbf{ref\ int}\ z$, и оба параметра процедуры рассматривались бы как параметры-значения. Фиктивный оператор (1) принял бы вид

$$v1 := e; \textbf{begin}\, typ\, P; \textbf{int}\, v; \textbf{ref int}\, y; v := v1; y := z; S0\, \textbf{end}$$

В результате присваивания $y := z$ значение y (ссылка) стало бы совпадать со значением z. Эта единая ссылка имела бы и единый предмет, что обеспечило бы корректное использование как y, так и $y\uparrow$ в выражениях и в левых частях операторов присваивания в пределах оператора $S0$.

3.4.2. Рекурсивные процедуры

Процедура P рекурсивна, если ее тело $S0$ содержит вхождения единиц вида $P(e', z')$. Чтобы не затемнять рассуждения, ограничимся случаем, когда программа использует лишь одну рекурсивную процедуру.

Главная задача настоящего раздела — распространить семантическую функцию Sem на единицы, являющиеся вызовами процедур или содержащие вхождения вызовов.

Пусть единица un — это вызов процедуры, имеющий вид $un \approx P(e, z)$. Содержательная семантика этой единицы остается той же, что и в разд. 3.4.1, так что для ее формального описания можно сохранить прежние формулы (3.4.1–2)–(3.4.1–7).

Но теперь преобразователь $\mathrm{Sem}(S1)$ в формуле (5), а значит и вся правая часть определения функции $\mathrm{Sem}(P(e, z))$, зависит от преобразователей вида $\mathrm{Sem}(P(e', z'))$ для вызовов $P(e', z')$ процедуры P из ее тела $S0$ (внутренних вызовов). Эти преобразователи могут быть получены только с помощью тех же формул, так что построить преобразователь $\mathrm{Sem}(P(e, z))$ для рассматриваемого (внешнего) вызова методом регулярного расширения невозможно (за конечное число шагов, зависящее только от самого вызова $P(e, z)$ и от единицы $S0$ из описания процедуры P, но не от исходного состояния). Однако математически корректное выражение для этого преобразователя можно получить, используя технику разд. 1.7.6 (но не следуя ей слепо).

Роль множества X будет играть множество W состояний w, роль U — множество Unit единиц un. При этом в качестве множества S будет выступать множество Semant преобразователей состояний — функций s типа $\mathrm{W} \to \mathrm{W}$, а обозначение H сохраним за множеством функций Sem (Sem1, Sem′ и т. п.) типа $\mathrm{Unit} \to \mathrm{Semant}$. Наряду с термином «семантика $\mathrm{Sem}(un)$ единицы un» будем, не вполне корректно, пользоваться термином «семантика Sem».

Из формул (3.4.1–2)–(3.4.1–7) видно, что правила вычисления значения вызова, как и других единиц un, входят в семантику Sem как органическая часть (первая последовательность). Ниже обычно не будем упоминать об этих правилах отдельно.

Таким образом, H — это множество семантических функций (правил), сопоставляющих единицам их семантику. Особенность ситуации — в том, что в состав единицы, для которой строится семантика, входит (явно или неявно) точно такая же или подобная ей единица, причем этот круг в общем случае не может быть статически разорван.

Термин «статически» употреблен в прежнем смысле — без анализа процесса исполнения программы. Вернемся на время к старой терминологии. Составные и условные операторы (о циклах — речь в следующем абзаце) могут содержать внутри себя другие операторы этих же двух видов. Но, выделяя из этих внутренних операторов входящие в них операторы-компоненты, мы за несколько шагов добираемся до неструктурированных операторов. Это позволяет построить семантические правила для составных и условных операторов регулярным расширением правил для операторов иного класса, синтаксически более простого, — круг разрывается.

С ситуацией статически неразмыкаемого круга мы столкнулись в разд. 3.2.1, где уже на содержательном уровне семантика оператора цикла определялась сама через себя: «Если значение b равно *true*, то исполняется тело цикла — оператор $S1$ и в полученном состоянии все повторяется». Фактически это заставило нас доказать в разд. 3.2.5 частный случай теоремы о неподвижной точке, не вводя этого понятия.

Оператор $\varphi : H \to H$, выражающий семантическую функцию внешней единицы через такую же функцию внутренней, отражает ситуацию замкнутого круга. Центральным в разд. 1.7.6 оказалось понятие неподвижной точки такого оператора. Но если там было не вполне понятно — откуда и для чего возникла задача о неподвижной точке оператора, то сейчас смысл этой задачи проясняется — семантика вызова должна быть одной и той же, на каком бы уровне рекурсии ни исполнялся вызов.

Для этого определим оператор φ так, чтобы он по заданной семантике Sem1 внутренних вызовов процедуры P из тела $S0$ ее описания (и из выражения e, соответствующего параметру-значению v) находил семантику Sem2 ее внешнего вызова, не принимая во внимание рекурсивность этой процедуры, т. е. считая семантики Sem1 и Sem2 различными. Различать в этом смысле следует и функции SemArg1 и SemArg2. Тогда формулы (3.4.1–2)–(3.4.1–7) можно переписать так:

$$\begin{aligned}&\mathrm{Sem2}(P(e,z))(w)\\&\quad=\mathrm{SemReturn}(P,\mathit{typ},w)\\&\qquad(\mathrm{SemArg2}((e,z),(v,\mathbf{var}\,y),S0,w)(w\{\mathit{val?}(\mathit{typ})/P\})),\end{aligned}\tag{8}$$

$$\begin{aligned}&\mathrm{SemArg2}((e,z),(v,\mathbf{var}\,y),S0,w_0)(w)\\&\quad=\mathrm{SemOut}(w,w_0)\\&\qquad(\mathrm{SemArg2}((z),(\mathbf{var}\,y),S0,w_0)(w\{e(w_0)/nm(v)\})),\end{aligned}\tag{9}$$

$$\begin{aligned}&\mathrm{SemArg2}((z),(\mathbf{var}\,y),S0,w_0)(w)\\&\quad=\mathrm{SemArg2}((),(),S0,w_0)(\mathrm{Subst}(z,y,S0)(w)),\end{aligned}\tag{10}$$

$$\mathrm{SemArg2}((),(),S0,w_0)(w)=\mathrm{Sem1}(S0)(w),\tag{11}$$

$$P(e,z)(w)=P(\mathrm{SemArg2}((e,z),(v,\mathbf{var}\,y),S0,w_0)(w\{\mathit{val?}(\mathit{typ})/P\})),\tag{12}$$

$$\mathrm{SemReturn}(P,\mathit{typ},w_0)(w)\;=\;w\setminus\langle P,w(P)\rangle\tag{13}$$

Формула $\mathrm{Sem2}=\varphi(\mathrm{Sem1})$ — это просто краткая запись зависимости между семантиками Sem1 и Sem2 (в том числе и между соответствующими преобразователями состояний), выраженной формулами (8)–(13).

Замечание. Мы не вводим обозначений SemReturn1 и SemReturn2, поскольку схема построения семантики SemReturn в отличие от семантики $\mathrm{Sem}(S1)$ для тела процедуры не зависит явно от семантики вызова. Однако при переходе от семантики Sem1 к семантике Sem2 расширяется область определения преобразователей состояний для любых программных единиц, а не только для вызовов. Это будет учитываться в доказательствах многих утверждений настоящего раздела.

Задача о неподвижной точке приобретает ясный смысл — найти семантику Sem, общую для внутренних и внешних вызовов, т. е. учесть, что описана именно рекурсивная процедура. Коротко это и описывается уравнением

$$\mathrm{Sem} \;=\; \varphi(\mathrm{Sem}). \tag{14}$$

Любой внутренний вызов тут же становится внешним по отношению к еще более внутренним вызовам. Это и приводит к идее о последовательности семантик $\langle \mathrm{Sem}_n \rangle$, порождаемой оператором φ.

Эта последовательность строится точно так же, как в разд. 1.7.6, а именно, Sem_0 — это семантика с неопределенным на любом вызове значением: $\forall e \forall z \forall w \forall w' \, (\mathrm{Sem}_0(P(e,z))(w) \neq w')$. При этом значение $\mathrm{Sem}_0(un)(w)$ заведомо не определено, если при исполнении un из состояния w потребуется обратиться к процедуре P.

Для $n > 0$ переход от семантики Sem_{n-1} к семантике Sem_n описывается, как уже было сказано, оператором φ, т. е. формулами (8)–(13), в которых роль Sem1 играет семантика Sem_{n-1}, а роль Sem2 — семантика Sem_n.

Семантика Sem_n для любого n может быть распространена на единицы любого вида методом регулярного расширения. (В предыдущих разделах семантическая функция Sem произвольна, в частности это может быть и функция Sem_n.)

Поясним содержательную сторону дела. Для наглядности изобразим схему основной программы в виде

$$\textbf{begin} \ldots \; P(e,z) \; \ldots \; \textbf{end}$$

а схему описания процедуры P — в виде

$$\langle P(v, \textbf{var}\, y) \colon typ, \; \textbf{begin} \; \ldots \; P(e', z') \; \ldots \; \textbf{end} \rangle.$$

Замечание. В модельном языке операторные скобки **begin** и **end** в этом контексте не нужны. Здесь они поставлены ради большей наглядности.

Для некоторых начальных состояний может случиться так, что при исполнении основной программы содержащиеся в ней вызовы вида $P(e,z)$ вообще не исполняются, так как находятся в неисполняемых ветвях условных единиц. В таких случаях ход исполнения программы не изменится, если заменить ее программой

$$\textbf{begin} \; \ldots \; P_0(e,z) \; \ldots \; \textbf{end} \tag{15}$$

где процедура P_0 описана так:

$$\langle P_0(v, \textbf{var}\, y) \colon typ, \; \textbf{abort} \rangle.$$

Затем можно выделить множество начальных состояний, для которых фактическая глубина рекурсии при обращении к процедуре P не превышает 1, т. е.

новых обращений к ней из единицы $S0$ не происходит. Для таких начальных состояний можно было бы вызовы процедуры P в основной программе заменить вызовами нерекурсивной процедуры P_1:

begin ... $P_1(e,z)$ ... **end**

где P_1 имеет описание

$\langle P_1(v, \mathbf{var}\, y)\colon typ,\ \mathbf{begin}\ \ldots\ P_0(e',z')\ \ldots\ \mathbf{end}\rangle$.

Продолжая этот процесс исключения рекурсивных обращений, на его n-м шаге придем к программе

begin ... $P_n(e,z)$ ... **end** (16)

явно обращающейся вместо P к процедуре P_n (нерекурсивной, как и все предыдущие):

$\langle P_n(v, \mathbf{var}\, y)\colon typ,\ \mathbf{begin}\ \ldots\ P_{n-1}(e',z')\ \ldots\ \mathbf{end}\rangle$ (17)

и косвенно — к процедурам P_1, ... , P_{n-1}. При этом будут охвачены все начальные состояния, для которых глубина рекурсии при исполнении программы в ее исходном виде не превышает n.

Для конкретной процедуры P не исключено, что для всех возможных случаев ее использования фактическая глубина рекурсии ограничена, так что все эти случаи исчерпываются на каком-то шаге процесса. Но в общем случае ни гарантировать это, ни рассчитывать на это нельзя, так что последовательность $\langle P_n\rangle$ надо считать бесконечной.

В последовательности $\langle \mathrm{Sem}_0,\ \mathrm{Sem}_1,\ \ldots\ \mathrm{Sem}_n,\ \ldots\rangle$ семантика Sem_n — это семантика процедуры P_n. Как уже сказано, методом регулярного расширения любую из этих семантик можно распространить на всю программу (16) (программа (15) — это частный случай программы (16) для $n = 0$). Программа (16) должна содержать описания процедур P_0, P_1, ... , P_n.

На содержательном уровне ясно, что каждая такая семантика расширяет предыдущую. Благодаря этому в процессе построения семантики программы раз найденная информация не теряется, не обесценивается. Если уже установлено, что некая единица, исполняясь в некотором состоянии, переводит программу в другое конкретное состояние, то это знание должно сохраняться, оставаться достоверным, что бы впоследствии ни выяснялось о семантике любой единицы. В этом заключается содержательная сторона понятия монотонности (оператора и порождаемой им последовательности семантик).

Обратимся к его формальной стороне.

Л е м м а 3.4.2. *Для произвольной семантической функции Sem справедлива формула* $Sem \subseteq \varphi(Sem)$.

Доказательство. Лемма утверждает, что, согласно определению (1.7.6–2) отношения $\subseteq$,

$\forall\, un\ (\mathrm{Sem}(un) \sqsubseteq \varphi(\mathrm{Sem})(un))$, (18)

или, на основании определения (1.7.6–1) отношения $\sqsubseteq$,

$\forall un \forall w \forall w'\ (\ \mathrm{Sem}(un)(w) = w' \supset \varphi(\mathrm{Sem})(un)(w) = w')$, (19)

где un — произвольная программная единица, w, w' — состояния из W.

Если семантика $\mathrm{Sem}(un)$ единицы un в состоянии w не определена, т. е. формула

$$\mathrm{Sem}(un)(w) = w' \tag{20}$$

ложна при любом w', то импликация в формуле (19) истинна для этих un и w.

Будем рассматривать поэтому только такие un и w, для которых формула (20) истинна при некотором (однозначно определяемом) состоянии w', т. е. $w \in \mathrm{Dom}(\mathrm{Sem}(un))$. При этом для единиц unI, исполнение которых завершается раньше, чем единицы un (назовем их *опережающими* эту единицу), состояния wI, из которых они исполняются, обязаны принадлежать соответствующей области определения $\mathrm{Dom}(\mathrm{Sem}(unI))$. Надо доказать, что отсюда следует

$$\varphi(\mathrm{Sem})(un)(w) = w'. \tag{21}$$

Прибегнем к индукции по порядку завершения исполнения единиц в процессе исполнения единицы un. Предположим, что для всех единиц и состояний, опережающих единицу un, соответствующие частные случаи формулы (19) истинны:

$$\mathrm{Sem}(unI)(wI) = wI' \supset \varphi(\mathrm{Sem})(unI)(wI) = wI'. \tag{22}$$

Рассмотрим возможные случаи.

а) Единица un — это вызов $P(e,z)$. Все единицы из тела процедуры P исполняются при исполнении этого вызова в некоторой семантике Sem', такой, что $\varphi(\mathrm{Sem}') \subseteq \mathrm{Sem}$. Вызовы вида $P(e',z')$ процедуры P из ее тела опережают вызов $P(e,z)$, так что по предположению индукции

$$\forall\, wI'\, (\mathrm{Sem}'(P(e',z'))(wI) = wI' \supset\ \varphi(\mathrm{Sem}')(P(e',z'))(wI) = wI')$$

и, в силу соотношения между семантиками Sem' и Sem,

$$\forall\, wI'\, (\mathrm{Sem}'(P(e',z'))(wI) = wI' \supset\ \mathrm{Sem}(P(e',z'))(wI) = wI'). \tag{23}$$

а′) Пусть, кроме того, все эти вызовы могут быть исполнены, т. е. для каждого из них найдется такое состояние wI', что $\mathrm{Sem}'(P(e',z'))(wI) = wI'$. Из формулы (23) следует, что и $\mathrm{Sem}(P(e',z'))(wI) = wI'$. Семантики $\mathrm{Sem}'(P(e',z'))$ и $\mathrm{Sem}(P(e',z'))$ входят, наряду с семантикой прочих неструктурных единиц, в базу для построения семантик $\mathrm{Sem}(P(e,z))$ и $\varphi(\mathrm{Sem})(P(e,z))$ методом регулярного расширения. Отсюда следует, что эти последние семантики должны совпадать. Следовательно, формула (21) справедлива.

а″) Пусть хотя бы один из вызовов $P(e',z')$ в соответствующем состоянии wI не исполни́м при семантике Sem'. Тогда был бы не исполни́м и вызов $P(e,z)$ в состоянии w при семантике Sem. Но это противоречит допущению об истинности формулы (20) (иначе говоря, этот случай уже рассмотрен).

б) Пусть $un \not\approx P(e,z)$. Согласно определению оператора φ, семантики $\mathrm{Sem}(un)$ и $\varphi(\mathrm{Sem})(un)$ получаются регулярным расширением семантики предшествующих единице un вызовов вида $P(e,z)$. Но для этих вызовов в пункте а) уже установлено, что их значения в обеих семантиках совпадают. Следовательно, должны совпадать между собой $\mathrm{Sem}(un)$ и $\varphi(\mathrm{Sem})(un)$. Формула (20) по предположению истинна, поэтому должна быть истинна и формула (21). $\lhd$

Следствие 3.4.3. *Последовательность $\langle Sem_n \rangle$ монотонна.*

Утверждение очевидно, так как $\mathrm{Sem}_n = \varphi(\mathrm{Sem}_{n-1})$. $\lhd$

Монотонность последовательности $\langle \mathrm{Sem}_n \rangle$ удалось доказать без ссылки на монотонность оператора φ. Но последняя нам еще потребуется при доказательстве теоремы о наименьшей неподвижной точке для данных обстоятельств.

Т е о р е м а 3.4.4. *Оператор φ монотонен.*

Доказательство. Формула (1.7.6–3) из определения монотонности оператора в принятых сейчас обозначениях выглядит так:

$$\mathrm{Sem}' \subseteq Sem'' \supset \varphi(\mathrm{Sem}') \subseteq \varphi(\mathrm{Sem}'').$$

С учетом определения отношений $\subseteq$ и $\sqsubseteq$ ее можно переписать в виде

$$\forall un \forall w \forall w' (\mathrm{Sem}'(un)(w) = w' \supset Sem''(un)(w) = w') \\ \supset \forall un \forall w \forall w' (\varphi(Sem')(un)(w) = w' \supset \varphi(Sem'')(un)(w) = w').$$

Пусть семантики Sem' и Sem'' таковы, что $\mathrm{Sem}' \subseteq \mathrm{Sem}''$, т. е.

$$\forall un \forall w \forall w' (\mathrm{Sem}'(un)(w) = w' \supset \mathrm{Sem}''(un)(w) = w'). \tag{24}$$

Надо доказать, что для произвольной программной единицы un и любых двух состояний w и w'

$$\varphi(\mathrm{Sem}')(un)(w) = w' \supset \varphi(\mathrm{Sem}'')(un)(w) = w'. \tag{25}$$

Если $w' \neq \varphi(\mathrm{Sem}')(un)(w)$, то импликация (25) истинна. Пусть имеет место $w' = \varphi(Sem')(un)(w)$. Тогда истинна формула $w \in \mathrm{Dom}(\varphi(Sem')(un))$, и для всех единиц unI, опережающих единицу un, состояния wI, из которых они исполняются, принадлежат соответствующей области определения $\mathrm{Dom}(\varphi(Sem')(unI))$. Остается доказать, что $w' = \varphi(Sem'')(un)(w)$. Воспользуемся индукцией по порядку завершения исполнения единиц.

Предположим, что для любой единицы unI, опережающей единицу un, справедливо утверждение

$$\varphi(Sem')(unI)(wI) = wI' \supset \varphi(Sem'')(unI)(wI) = wI'.$$

Это подразумевает и совпадение значений $unI(wI)$ в семантиках $\varphi(Sem')$ и $\varphi(Sem'')$.

По лемме 3.4.2 для любой единицы un и любого состояния w в любой семантике Sem справедлива формула

$$\varphi(\mathrm{Sem})(un) = \mathrm{Sem}(un), \tag{26}$$

а значения $un(w)$ в семантиках $\varphi(\mathrm{Sem})$ и Sem совпадают.

а) Если un не содержит вхождений единиц вида $P(e,z)$, то ее семантика строится методом регулярного расширения. Используя формулу (26) дважды: для $\mathrm{Sem} = \mathrm{Sem}'$ и $\mathrm{Sem} = \mathrm{Sem}''$, получаем формулу (25) как прямое следствие формулы (24).

б) Если $un \approx P(e,z)$, то как $\varphi(Sem')(un)$, так и $\varphi(Sem'')(un)$ находятся по формулам (8)–(13), где в роли Sem1 выступает Sem' или Sem'', в роли Sem2 — соответственно $\varphi(Sem')$ или $\varphi(Sem'')$. Единица e опережает un, и, по предположению индукции, состояния $\varphi(Sem')(e)(w)$ и $\varphi(Sem'')(e)(w)$ должны совпадать, как и значения $e(w)$ в обеих семантиках. Единицы $typ\,P$, $\mathbf{int}\,v$, $v := e$ и $S0$ также опережают un. Поскольку исполнение un в семантике $\varphi(Sem')$ завершается, то и

исполнение этих единиц в семантике Sem' должно завершаться. Согласно формуле (24) должно завершаться и исполнение каждой из этих единиц в семантике Sem'', причем в том же состоянии, что и в семантике Sem'. Если единица e содержит вызовы процедуры P, то истинность формулы устанавливается сначала для этих вызовов, а уж потом для процедуры un. Таким образом, однократное применение формул (8)–(13) приводит к заключению, что семантики $\varphi(Sem')$ и $\varphi(Sem'')$ должны совпадать, т. е. что формула (25) истинна.

в) Если $un \approx unI1\ op\ unI2$, то определение семантики выражений из разд. 3.4.1 в сочетании с формулами (26) обеспечивает все необходимое для доказательства.

г) Пусть $un \approx$ **if** bI **then** $unI1$ **else** $unI2$ **fi**. Тогда, в силу формул (26), $\varphi(\mathrm{Sem})(bI) = \mathrm{Sem}(bI)$. Значение $bI(w)$, одинаковое в семантиках $\varphi(\mathrm{Sem})$ и Sem, определяет выбор между единицами $unI1$ и $unI2$, а семантика выбранной единицы становится семантикой единицы un. Дальнейшее очевидно.

Случаи

д) $un \approx unI1;\ unI2$,

е) $un \approx$ **while** bI **do** unI **od**,

ж) $un \approx typ\,v;\ unI$

рассматриваются аналогично. $\lhd$

Итак, последовательность $\langle \mathrm{Sem}_n \rangle$ — это цепь. По ней может быть построена семантическая функция

$$\mathrm{Sem0} = \mathrm{LUB}(\mathrm{Sem}) = \bigcup_{n \geqslant 0} \mathrm{Sem}_n. \tag{27}$$

Теорема 3.4.5. *Семантическая функция* Sem0 *описывает исполнение любой программы, обращающейся к процедуре P и только к ней, из произвольного начального состояния и является наименьшей неподвижной точкой оператора φ :*

$$\mathrm{Sem0} = \varphi(\mathrm{Sem0}).$$

Доказательство. Согласно определению наименьшей верхней грани, множество Unit распадается на два непересекающихся класса. Для всех единиц un первого класса семантика $\mathrm{Sem}_n(un) = s0$ при любом n (класс единиц, неисполнимых из-за невозможности выйти из рекурсии, в том числе — неисполнимых программ). Тогда и $\mathrm{Sem0}(un) = s0$. Это значит, что $\mathrm{Sem}_n(un) = \mathrm{Sem0}(un)$, а так как un произвольна (в рассматриваемом классе), то и $\mathrm{Sem}_n = \mathrm{Sem0}$ для всех n. В частности, $\mathrm{Sem}_0 = \mathrm{Sem0}$ и $\mathrm{Sem}_1 = \mathrm{Sem0}$. Но $\mathrm{Sem}_1 = \varphi(\mathrm{Sem}_0)$, так что утверждение теоремы справедливо.

Для единиц un второго класса и любого состояния w найдутся такие n и w', что $\mathrm{Sem}_n(un)(w) = w'$. Тогда и $\mathrm{Sem0}(un)(w) = w'$, откуда

$$\mathrm{Sem}_n(un)(w) = \mathrm{Sem0}(un)(w). \tag{28}$$

Здесь n зависит как от un, так и от w, поэтому отбросить аргументы w и un нельзя. Будем рассматривать в качестве un не все единицы из Unit, а только основную программу $un0$ и ее подъединицы unI. Зафиксируем состояние w_0, из которого начинается исполнение $un0$. Для каждой unI состояние wI, из которого она исполняется, однозначно зависит от w_0.

Пусть n_0 — наименьшее значение n, при котором формула (28) истинна для $un \approx un0$ и $w = w_0$. Так как при $n = n_0$ исполнение $un0$ из состояния w_0 завершается, то при этом же или меньшем значении n завершается исполнение любой единицы unI из состояния wI, т. е. все эти единицы принадлежат второму классу. Семантика этого исполнения — это Sem_n, где $n \leqslant n_0$. Если результат исполнения есть wI', т. е. $\mathrm{Sem}_n(unI)(wI) = wI'$, то, согласно формуле (28), $\mathrm{Sem0}(unI)(wI) = wI'$.

Это значит, что семантика Sem0 исчерпывающе описывает исполнение как всей программы $un0$, так и любой ее подъединицы. Это значит также, что в формулах (8)–(13) можно заменить как Sem1, так и Sem2 со всеми их частями (w, SemArg и др.) на Sem0 и ее части. Отсюда и следует, что $\varphi(\mathrm{Sem0}) = \mathrm{Sem0}$, т. е. семантика Sem0 — это неподвижная точка оператора φ.

То, что эта неподвижная точка — наименьшая, вытекает из монотонности этого оператора (см. разд. 1.7.6). $\triangleleft$

3.4.3. Преобразователи предикатов для процедур

Проблему преобразования предикатов в аппарате процедур рассмотрим на примере, без выхода в деривационную семантику, т. е. без построения правил вывода для этого аппарата.

Продолжим начатое в разд. 3.1 и 3.2 исследование задачи о слиянии двух упорядоченных массивов в один. В разд. 3.2.9 возник набор процедур, которые мы сейчас слегка переделаем, устранив ненужную переменную *cont* и превратив переменные i, j и k в параметры-значения процедур A, B, C и D:

$$\begin{array}{l}\langle C(\textbf{int } i,j,k),\\ \quad \textbf{if } i > m \textbf{ then } B(i,j,k)\\ \quad \textbf{else if } j > n \textbf{ then } A(i,j,k)\\ \qquad \textbf{else if } a[i] > b[j] \textbf{ then } B(i,j,k) \textbf{ else } A(i,j,k) \textbf{ fi}\\ \qquad \textbf{fi}\\ \quad \textbf{fi}\rangle;\end{array} \tag{29}$$

$$\langle A(\textbf{int } i,j,k), c[k] := a[i];\ D(i+1,j,k+1)\rangle; \tag{30}$$

$$\langle B(\textbf{int } i,j,k), c[k] := b[j];\ D(i,j+1,k+1)\rangle; \tag{31}$$

$$\langle D(\textbf{int } i,j,k), \textbf{if } k <= m+n \textbf{ then } C(i,j,k) \textbf{ else } \mathit{Write}(c)\rangle. \tag{32}$$

Соответственно изменится основная программа:

$$\begin{array}{l}\textbf{begin } \mathit{Read}(m,n,a,b)\,[m \geqslant 0 \wedge n \geqslant 0 \wedge \mathit{Ord}(a,\,1,\,m) \wedge \mathit{Ord}(b,\,1,\,n)];\\ \quad C(1,1,1) \textbf{ end}.\end{array} \tag{33}$$

Здесь в соответствии со сказанным в п. 8 разд. 3.2.7 выписано и постусловие $p0$ для оператора ввода — необходимое условие для правильной работы программы. Поскольку значения m и n после ввода не меняются, дублирующие константы $m0$ и $n0$ не нужны. Не меняются и значения элементов массивов a и b, поэтому условие $p0$ остается истинным на протяжении исполнения всей программы и будет входить подразумеваемой конъюнктивной компонентой во все промежуточные пред- и постусловия.

После обращения $C(1,1,1)$ процедуры исполняются циклически в следующей последовательности: C, A или B, D, снова C и т. д., пока при очередном исполнении D не произойдет вывод массива c и завершение всех вызовов и работы программы. Существенно рекурсивной (см. предыдущий раздел) оказывается только процедура C, а процедуры A, B и D можно считать ее нерекурсивными частями.

В том же порядке происходит построение семантики вызовов — как преобразователей состояний, так и преобразователей предикатов. Сначала строится SemArg (SemArg$\uparrow$ или SemArg$\downarrow$) при полных 3-элементных списках параметров и аргументов, затем при 2- и 1-элементных — во всех случаях по схеме (3.4.2–2), так как используется значение, а не имя каждого аргумента. После этого строится семантика тела процедуры, содержащего новые вызовы. Однако раскрывать эту семантику для процедуры C уже не требуется, так как, согласно разд. 3.4.2, она становится лишь аргументом Sem1 оператора φ для этой процедуры, а именно он (оператор) нас сейчас интересует.

Этот оператор описывается следующими формулами:

$$\begin{cases} i > m \wedge \mathrm{Sem2}(C(i,j,k)) = \mathrm{Sem}(B(i,j,k)) \\ \quad \vee\, i \leqslant m \wedge j > n \wedge \mathrm{Sem2}(C(i,j,k)) = \mathrm{Sem}(A(i,j,k)) \\ \quad \vee\, i \leqslant m \wedge j \leqslant n \wedge a[i] > b[i] \\ \qquad \wedge \mathrm{Sem2}(C(i,j,k)) = \mathrm{Sem}(B(i,j,k)) \\ \quad \vee\, i \leqslant m \wedge j \leqslant n \wedge a[i] \leqslant b[i] \\ \qquad \wedge \mathrm{Sem2}(C(i,j,k)) = \mathrm{Sem}(A(i,j,k)), \end{cases} \tag{34}$$

$$\mathrm{Sem}(A(i,j,k))(w) = \mathrm{Sem}(D(i+1,j,k+1))(w\{a[i](w)/nm(c[k])\}, \tag{35}$$

$$\mathrm{Sem}(B(i,j,k))(w) = \mathrm{Sem}(D(i,j+1,k+1))(w\{b[j](w)/nm(c[k])\}, \tag{36}$$

$$\begin{cases} k \leqslant m+n \wedge \mathrm{Sem}(D(i,j,k)) = \mathrm{Sem1}(C(i,j,k)) \\ \quad \vee\, k > m+m \wedge \mathrm{Sem}(D(i,j,k)) = I, \end{cases} \tag{37}$$

которые строятся почти автоматически на основе формул разд. 3.2.4 и 3.2.5.

Приступим, исходя из предусловия $p0$, к формированию постусловия для начальных вызовов процедуры C. Предположим пока, что m и n достаточно велики. Тогда при исполнении вызова $C(1,1,1)$ из основной программы потребуется сравнить значения $a[1]$ и $b[1]$. Симметрия позволяет ограничиться лишь одним исходом сравнения, например $a[1] \leqslant b[1]$. Для этого случая должен быть исполнен вызов $A(1,1,1)$ и, следовательно, присваивание $c[1] := a[1]$ и вызов $D(2,1,2)$, влекущий за собой вызов $C(2,1,2)$. В результате присваивания в предусловии для этого вызова (оно же постусловие предыдущего вызова процедуры C) появится член $c[1] = a[1]$. Итог анализа аргументов обращения породит члены $i = 2$, $j = 1$ и $k = 2$. Результат сравнения также должен войти в это условие, но с помощью члена $c[1] = a[1]$ ему можно придать форму $c[1] \leqslant b[1]$.

Теперь сравнению подлежат значения $a[2]$ и $b[1]$. Исходы сравнения уже не симметричны. Исход $a[2] \leqslant b[1]$, аналогично предыдущему, порождает члены $c[2] = a[2]$, а также $i = 3$, $j = 1$, $k = 3$ и $c[2] \leqslant b[1]$. Кроме этого, следствие $a[1] \leqslant a[2]$ предиката $Ord(a, 1, m)$, входящего в $p0$, без труда преобразуется в $c[1] \leqslant c[2]$ с помощью уже имеющихся членов и в таком виде также может быть включено в постусловие.

При исходе $a[2] > b[1]$ в постусловие войдут члены $c[2] = b[1]$, $i = 2$, $j = 2$ и $k = 3$, а уже содержащийся в нем член $c[1] \leqslant b[1]$ преобразуется в $c[1] \leqslant c[2]$. Результат сравнения может быть отражен в постусловии в форме $a[2] > c[2]$ и при случае послужить материалом для формирования сначала члена $c[2] < c[3]$, а затем — члена $c[2] \leqslant c[3]$.

Но в постусловии желательно иметь не эти разрозненные члены, а инвариант процедуры C и программы в целом, содержащий в себе формулы $i+j = k+1$, $Ord(c, 1, k-1)$, еще одну формулу, утверждающую, что для каждого k найдется либо единственное i, такое что $c[k] = a[i]$, либо единственное j, такое что $c[k] = b[j]$, но не оба эти случая вместе, и наконец, кое-какие мелочи (хотя их в программировании не бывает), позволяющие установить, что работа программы завершается при значении k, равном $m+n+1$. Для конкретных значений i, j и k выше было показано, что нетрудно получить все необходимое для вывода этих формул по индукции.

Этот достаточно примитивный пример все же отчетливо выявляет общее свойство всех задач, для которых составляются компьютерные программы. Каков бы ни был уровень описания:

- допрограммный (в нашем примере — это формулировка задачи словами «слияние двух упорядоченных массивов в один», более формальная спецификация программы состоит из предусловия $p0$ и инварианта процедуры C в той форме, которую он принимает при $k = m + n + 1$);
- программный (программа (33) с процедурами (29)–(32));
- преобразователей состояний (формулы (34)–(37));
- преобразователей предикатов (все сказанное по поводу инварианта программы),

— на этом уровне должны быть зафиксированы в свойственной ему форме одни и те же свойства задачи на фоне одной и той же, начиная с уровня формальной спецификации программы, последовательности действий.

Изложение основ денотационной семантики языков программирования операторного (императивного) типа на этом завершается.

Однако природа последнего примера почти что заставляет сделать следующее наблюдение. Как только в условном операторе, составляющем тело процедуры C, избрана одна из двух первых ветвей (пусть первая), параметр i замерзает на значении $m+1$, первая формула инварианта вырождается в $m+j = k$ и открывается возможность сократить число дальнейших проверок, заменив в этой ветви процедуру B процедурой

$$\langle B1(\mathbf{int}\ j, k), c[k] := b[j]; \mathbf{if}\ j < n\ \mathbf{then}\ B1(j+1, k+1)\ \mathbf{else}\ \mathit{Write}(c)\rangle.$$

Во второй ветви появляется вызов совершенно аналогичной процедуры $A1$. Обе процедуры существенно рекурсивны. Такое изменение особенно полезно, если программа работает в условиях, когда один из массивов a или b может быть существенно короче другого, возможно даже пуст.

Мало того, после этих изменений проверка в процедуре D всегда дает положительный результат и поэтому становится излишней. Но тогда и сама процедура

D не делает ничего полезного, кроме вызова процедуры C. Поэтому в процедурах A и B можно обращаться не к D, а прямо к C с теми же значениями аргументов.

Итак, элементарный акт — будь то присваивание или проверка условия — имеет не только локальный эффект (изменение значения одной переменной или выбор ветви), но и глобально отражается на всем состоянии программы, а главное — меняет как форму, так и содержание утверждений об этом состоянии, что, в свою очередь, позволяет преобразовать (оптимизировать) оставшуюся для исполнения часть алгоритма.

Подобные наблюдения в гораздо более общей и плодотворной форме привели А. П. Ершова (ряд статей в [19]) и других авторов [56] к понятию смешанных и частичных вычислений, выполняемых над данными в более широком понимании — не только над значениями переменных, но и, например, над текстами программ. Но это — другая история.

3.5. Послесловие. За что боролись?

Подведем итоги. Программиста-практика весь аппарат денотационной семантики интересует, вероятно, не больше, чем устройство, детали работы на машинном уровне, используемого им компьютера и его программного оснащения. Ему важно, чтобы все это хозяйство работало, чтобы его собственная программа исполнялась без помех, а если что-то мешает ее исполнению, то желательно поточнее знать — что именно.

Так и от денотационного аппарата, представляющего собой набор общих схем построения семантики конкретных программных единиц, программист ждет (предположим, что он все же проникся идеей о полезности формального описания семантики программ), чтобы он помог ему построить семантику его конкретной программы. Если это не удается, то важно узнать — в каком именно месте и по какой конкретной причине.

Денотационный аппарат, строя по шагам семантику программы и всех ее единиц, дает такую возможность. Особенно полезна, конечно, помощь в отладке программы — в выявлении содержащихся в ней семантических ошибок (в той мере, в какой это вообще возможно). Если, начиная с некоторого момента, семантика какой-либо программной конструкции (а затем и всех конструкций, ее объемлющих) оказывается не определенной, то — мы за этим следили — источник неопределенности фиксируется. Если после завершения построения семантики программы выясняется, что заключительное состояние не соответствует спецификации, то есть надежда, хотя и слабая, что источник несоответствия может быть обнаружен в семантике одного из промежуточных состояний (а не в порочности модели, на основе которой построена программа).

Предполагается, что такую помощь аппарат способен оказать. Но для этого должны быть выполнены некоторые условия. Читатель, вероятно, давно уже заметил, что от намеченной цели — описать семантику алгоритмического языка в математическом, денотационном стиле, противопоставляемом операционному, мы оказались очень далеки, если вообще к ней продвинулись. Фактически даны лишь правила, позволяющие параллельно с собственно построением программы выводить утверждения о текущем состоянии. Польза, которую эти утверждения могут оказать при отладке программы, сильно зависит от круга понятий, используемых при формулировке этих утверждений. Можно сказать, что обозримость совокупности этих утверждений более важна, чем обозримость самой программы, и даже, что вторая сводится к первой.

Проблема неопределенности программы в случаях, более всего волнующих программиста-практика, — незавершаемость циклов, невыход из рекурсии — не решается (если только не удастся выразить каждое из многих бесконечных объединений $\bigcup_{n\geqslant 0}$ в конечной, замкнутой, форме) и, как уже говорилось, в принципе едва ли может быть решена.

Остается надежда, довольно эфемерная, что денотационный язык описания семантики может оказаться удобным промежуточным языком при создании интерпретаторов и компиляторов для вновь проектируемых алгоритмических языков. Это может случиться, если сам денотационный язык окажется достаточно

эффективно реализуемым на современных компьютерах и если перевод с проектируемого языка на денотационный будет достаточно прост и поддержан необходимыми программными средствами.

В случае задач, взятых, подобно рассмотренной в разд. 3.4.3, из математики (или смежных с ней дисциплин), доалгоритмическое описание наиболее компактно. Это не удивительно, поскольку возраст математики на 1–2 порядка (десятичных) больше возраста программирования и она обросла горами полезных понятий, которые студенты худо-бедно осваивают за несколько лет обучения. Спецификации задач из слабо формализованных областей, как-то: медицина, экономика и управление, компьютерные игры и т. п., напротив, обычно пространны и маловразумительны. Здесь сами современные алгоритмические языки и иные средства программирования (информационные технологии) служат подчас наиболее приемлемым средством формального описания задачи.

Громоздкость денотационных описаний связана с тем, что они охватывают не только исполняемые действия, но и механизм их исполнения, тогда как программы на том или ином алгоритмическом языке опираются на независимое описание содержательной семантики всех языковых конструкций.

Описание семантики программ в терминах предикатов — пред- и постусловий — содержит в себе еще и описание контекста (всех промежуточных состояний), в котором исполняются эти конструкции. Если бы удалось создать систему автоматического доказательства свойств программ, то каждую новую программу нужно было бы снабдить только двумя предикатами — исходным и заключительным, т. е. дать вместе с программой еще и спецификацию решаемой задачи. Все остальные предикаты возникали бы в процессе построения логического вывода и содержались в его тексте (см. пример из разд. 3.2.2), обращаться к которому программист должен был бы лишь в случаях острой необходимости. Впрочем, ситуация здесь снова идеализирована. Как убрать многочисленные бесконечные объединения $\bigcup_{n\geqslant 0}$ из выражений для неподвижных точек (они же инварианты явных и неявных — замененных рекурсией — циклов)? Да и систем подобного рода на рынке программной продукции пока не видно.

Лишь для задач некоторых сравнительно простых типов существуют системы (например, PRIZ — авторы E. H. Tyugu и его коллеги), где пользователь пишет только спецификацию задачи, а гарантированно правильная (т. е. соответствующая этой спецификации) программа ее решения синтезируется автоматически. Правда перед этим кто-то должен очень серьезно поработать и создать модель той предметной области, из которой взята эта задача и в которой есть и много других. Модель должна содержать формальное описание всех понятий, используемых в этой области, ряд заготовок (фрагментов будущих программ) и другие данные.

Прощаясь с читателем, автор выражает робкую надежду (сменившую былое твердое убеждение), что подобным системам принадлежит будущее.

Решения упражнений

1.1.1. а. $x(y+z) = (y+z)x = yx + zx = xy + zx = xy + xz$.
б. $(x-y)z = (x+(-1)y)z = xz + ((-1)y)z = xz + (-1)yz = xz - yz$.
в. $x - x = x + (-1)x = 1 \cdot x + (-1)x = (1 + (-1))x = 0 \cdot x = x \cdot 0 = 0$.

1.1.2. $(a^2-ab)+(ba-b^2) = (a^2-ab)+(ba+(-1)b^2) = ((a^2-ab)+ba)+(-1)b^2 = = ((a^2-ab)+ba)-b^2$. Далее: $(a^2-ab)+ba = (a^2+(-1)ab)+ba = a^2+((-1)ab+ba)$. Затем $(-1)ab + ba$ преобразуется в 0 и после этого $a^2 + 0$ заменяется на a^2.

1.1.3. Достаточно воспользоваться последней интерпретацией равенства (1.1.1–3), понимая под a и b векторы, представленные полудиагоналями параллелограмма.

1.1.4. Предположим, что $|\chi_1| = |\chi_1'| = l$. Из $|\chi_1| \leqslant |\chi_1|+|\varphi|+|\chi_2| = |\psi|$ следует, что $l \leqslant k$, где $k = |\psi|$. Пусть $\chi_1 = b_1 \ldots b_l$, $\chi_1' = c_1 \ldots c_l$, $\psi = d_1 \ldots d_k$. Так как $\psi = \chi_1\varphi\chi_2$, то $b_1 = d_1, \ldots, b_l = d_l$, и аналогично $c_1 = d_1, \ldots, c_l = d_l$. Но тогда $b_1 = c_1, \ldots, b_l = c_l$, а это значит, что $\chi_1 = \chi_1'$. Но это противоречит тому, что вхождения $\chi_1\varphi\chi_2$ и $\chi_1'\varphi\chi_2'$ различны. Следовательно, $|\chi_1| \neq |\chi_1'|$.

1.1.5. Обозначим $l = |\chi_1|$, $l' = |\chi_1'|$. Пусть $l \leqslant l'$, $\chi_1 = b_1 \ldots b_l$, $\chi_1' = c_1 \ldots c_{l'}$. Так же как в упражнении 1.1.4, убеждаемся, что $b_i = c_i$ для $i = 1, \ldots, l$. Обозначим слово $c_{l+1} \ldots c_{l'}$ (может быть, пустое) через ξ. Тогда $\chi_1' = \Lambda\chi_1\xi$, т. е. χ_1 входит в χ_1'. Аналогично, если $l \geqslant l'$, то χ_1' входит в χ_1.

1.1.6. $n + 1$.

1.1.7. Если $\chi_1 a \chi_2$ и $\chi_1' a \chi_2'$ — различные вхождения буквы a в слово ψ, то длины слов χ_1 и χ_1' отличаются хотя бы на 1. Но длина однобуквенного слова равна 1, поэтому эти вхождения пересекаться не могут.

1.1.8. Формулы теории $\mathbf{L}_2$ — любые слова в алфавите $\mathbf{A}_2$, аксиомы — это x, y, z, правила вывода —

$$\frac{\varphi \quad \psi}{(\varphi + \psi)} \quad \text{и} \quad \frac{\varphi \quad \psi}{(\varphi \times \psi)}.$$

Вывод формулы $((x + y) \times (y + z))$ таков:

x, y, $(x + y)$, z, $(y + z)$, $((x + y) \times (y + z))$.

1.1.9. Любой вывод из списка Γ является, в соответствии с определением, выводом из Δ.

1.1.10. Пусть

$$\mathcal{B}_1, \ldots, \mathcal{B}_n \tag{1}$$

— вывод формулы $\mathcal{B}$ в теории $\mathbf{T}_1$, а

$$\mathcal{B}_1', \ldots, \mathcal{B}_{n'}' \tag{2}$$

— вывод формулы $\mathcal{A}$ из списка гипотез $\mathcal{C}_1, \ldots, \mathcal{C}_k$ в теории $\mathbf{T}$. По условию оба вывода существуют. Если в выводе (1) ни разу не применялось правило

$$\frac{\mathcal{C}_1 \quad \ldots \quad \mathcal{C}_k}{\mathcal{A}}, \tag{3}$$

то последовательность (1) будет выводом формулы $\mathcal{B}$ и в теории $\mathbf{T}$, т. е. $\vdash_{\mathbf{T}} \mathcal{B}$. Если же правило (3) было использовано как основание для включения в вывод (1)

формулы $\mathcal{B}_i$, то $\mathcal{B}_i$ совпадает с $\mathcal{A}$, и среди формул $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$ содержатся все формулы $\mathcal{C}_1, \ldots, \mathcal{C}_k$.

Пусть $\mathcal{B}_i$ — самая первая формула, включенная в вывод (1) на этом основании. Рассмотрим последовательность

$$\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}, \mathcal{B}'_1, \ldots, \mathcal{B}'_{n'}, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_n \tag{4}$$

Каждая из формул $\mathcal{B}'_j$, $1 \leqslant j \leqslant n'$, является либо аксиомой теории $\mathbf{T}$, либо непосредственным следствием по правилу MP предшествующих ей формул (в последовательности (2), а значит, и в последовательности (4)), либо совпадает с одной из формул $\mathcal{C}_1, \ldots, \mathcal{C}_k$. В последнем случае она может быть включена в вывод в теории $\mathbf{T}$ на том же основании, на каком в нем содержится совпадающая с $\mathcal{B}'_j$ формула из числа $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$. Таким образом, отрезок последовательности (4), кончая формулой $\mathcal{B}'_{n'}$, может рассматриваться как вывод (формулы $\mathcal{B}'_{n'}$) в теории $\mathbf{T}$. Вся последовательность (4) может еще не быть выводом в теории $\mathbf{T}$, если среди формул $\mathcal{B}_{i+1}, \ldots, \mathcal{B}_n$ содержатся другие формулы, включенные в вывод (1) на основании правила (3). Но их можно поочередно исключить тем же приемом, что формулу $\mathcal{B}_i$. Преобразованная таким образом последовательность станет выводом формулы $\mathcal{B}$ в теории $\mathbf{T}$, т. е. $\vdash_{\mathbf{T}} \mathcal{B}$.

1.2.1. Поместим в счетчик термов, которые мы еще ожидаем встретить, число 1. Будем просматривать символы слова, начиная с самого левого и продвигаясь по одному направо. Если очередной символ — k-местная функциональная буква, то увеличим содержимое счетчика на $k-1$. Если этот символ — предметная константа или переменная, то вычтем из счетчика 1. Если этот символ — какой-то иной, то слово — не терм. Легко убедиться, что по завершении просмотра всех символов какого-либо терма, входящего в данное слово, содержимое счетчика становится на 1 меньше, чем было перед началом просмотра этого терма. Поэтому если данное слово — терм, то по окончании просмотра всех символов, составляющих этот терм, в счетчике должен оказаться 0. Если 0 появится в счетчике раньше, чем окончится просмотр всего слова, или по завершении этого просмотра содержимое счетчика будет положительно, то слово — не терм.

Пример.

Слово		f_5^3	a_2	f_{10}^2	x_3	f_4^1	x_1	f_2^1	f_7^1	f_4^2	a_1	x_2
Счетчик	1	3	2	3	2	2	1	1	1	2	1	0

1.2.2. Алфавит: $\{a_1, a_2, \ldots, x_1, x_2, \ldots, f_1^1, f_2^1, \ldots, f_1^2, f_2^2, \ldots, (\ , \ ), \ , \}$. Формулы — любые слова в этом алфавите. Аксиомы: $a_1, a_2, \ldots, x_1, x_2, \ldots$ Правила вывода (по одному на каждую функциональную букву f_i^k):

$$\frac{\mathcal{A}_1 \quad \mathcal{A}_2 \quad \ldots \quad \mathcal{A}_k}{f_i^k(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k)}.$$

Впрочем, эта формальная теория не удовлетворяет требованиям, сформулированным в разд. 1.2.1 — ее алфавит бесконечен, бесконечно также множество правил вывода. Дело можно поправить, если вместо обозначений a_i, x_i, f_i^k пользоваться обозначениями вида $a||\ldots|$, $x||\ldots|$, $f||\ldots|.||\ldots|$, где число палочек на

1 меньше прежних индексов i и k. Тогда можно обойтись конечным алфавитом $\{a,\ x,\ f,\ |\ ,\ .\ ,\ (\ ,\ )\ ,\ ,\}$, аксиомами a и x и правилами вывода:

1) $\dfrac{a\mathcal{A}}{a\mathcal{A}|}$ (если a_i — терм, то и a_{i+1} — терм),

2) $\dfrac{x\mathcal{A}}{x\mathcal{A}|}$ (если x_i — терм, то и x_{i+1} — терм),

3) $\dfrac{\mathcal{D}}{f.(\mathcal{D})}$ (если $\mathcal{D}$ — терм, то и $f_1^1(\mathcal{D})$ — терм),

4) $\dfrac{f.\mathcal{B}(\mathcal{C})\quad \mathcal{D}}{f.\mathcal{B}|(\mathcal{C},\mathcal{D})}$ (если $f_1^k(\mathcal{C})$ и $\mathcal{D}$ — термы, то и $f_1^{k+1}(\mathcal{C},\mathcal{D})$ — тоже терм),

5) $\dfrac{f\mathcal{A}.\mathcal{B}}{f\mathcal{A}|.\mathcal{B}}$ (если $f_i^k\mathcal{B}$ — терм, то и $f_{i+1}^k\mathcal{B}$ — терм).

Слова $\mathcal{A}$ в правиле 5 и $\mathcal{B}$ в правиле 4 не должны (и не могут) содержать других символов, кроме $|$.

В этих обозначениях терм $f_4^2(x_1,\ a_3)$ имеет вид $f|||.|(x,\ a||)$. Его вывод таков:

x	(аксиома)
$f.(x)$	(по правилу 3 при $\mathcal{D}=x$)
a	(аксиома)
$a\|$	(по правилу 1 при $\mathcal{A}=\Lambda$)
$a\|\|$	(по правилу 1 при $\mathcal{A}=\|$)
$f.\|(x,\ a\|\|)$	(по правилу 4 при $\mathcal{B}=\Lambda,\ \mathcal{C}=x,\ \mathcal{D}=a\|\|$)
$f\|.\|(x,\ a\|\|)$	(по правилу 5 при $\mathcal{A}=\Lambda,\ \mathcal{B}=\|(x,\ a\|\|)$)
$f\|\|.\|(x,\ a\|\|)$	(по правилу 5 при $\mathcal{A}=\|$ и том же $\mathcal{B}$)
$f\|\|\|.\|(x,\ a\|\|)$	(по правилу 5 при $\mathcal{A}=\|\|$ и том же $\mathcal{B}$)

1.2.3. Для $n=2$ существует только один порядок расстановки скобок (объединение формул $\mathcal{A}_1$ и $\mathcal{A}_2$ в конъюнкцию $(\mathcal{A}_1\wedge\mathcal{A}_2)$). Для $n>2$ назовем порядок, при котором сначала объединяются $\mathcal{A}_1$ и $\mathcal{A}_2$, затем $(\mathcal{A}_1\wedge\mathcal{A}_2)$ и $\mathcal{A}_3$ и т. д., стандартным. Предположим, что для всех групп, содержащих меньше n формул, уже доказано, что произвольный порядок скобок эквивалентен стандартному, т. е. приводит к формуле, имеющей то же значение, что и формула со стандартным порядком. Докажем, что это верно и для группы, содержащей n формул. Пусть главная связка объединяет формулы, содержащие k и $m=n-k$ членов: $(\mathcal{A}_1\wedge\ldots\wedge\mathcal{A}_k)\wedge(\mathcal{A}_{k+1}\wedge\ldots\wedge\mathcal{A}_n)$ (внутренние скобки в группах не выписаны). Если $m>1$, то порядок скобок во второй группе либо уже стандартен, либо его можно изменить на стандартный: $(\mathcal{A}_1\wedge\ldots\wedge\mathcal{A}_k)\wedge((\mathcal{A}_{k+1}\wedge\ldots\wedge\mathcal{A}_{n-1})\wedge\mathcal{A}_n)$ по предположению индукции. По ассоциативности эта формула эквивалентна формуле $((\mathcal{A}_1\wedge\ldots\wedge\mathcal{A}_k)\wedge(\mathcal{A}_{k+1}\wedge\ldots\wedge\mathcal{A}_{n-1}))\wedge\mathcal{A}_n$. По тому же предположению в подформуле $(\mathcal{A}_1\wedge\ldots\wedge\mathcal{A}_k)\wedge(\mathcal{A}_{k+1}\wedge\ldots\wedge\mathcal{A}_{n-1})$ также можно заменить порядок скобок на стандартный. В результате он станет стандартным во всей формуле. При каждой перегруппировке членов формула преобразовывалась в эквивалентную, следовательно, заключительная формула (со стандартным порядком) эквивалентна исходной. По индукции утверждение верно для произвольного n.

1.2.4. Доказательство отличается от доказательства теоремы 1.2.11 лишь тем, что вместо эквивалентности всюду говорится об импликации.

1.2.5. По формулам разд. 1.2.7:

$$((\mathcal{A} \supset \mathcal{B}) \supset \mathcal{A}) \supset \mathcal{A}$$
$$\Leftrightarrow (\neg(\mathcal{A} \supset \mathcal{B}) \vee \mathcal{A}) \supset \mathcal{A} \quad \text{(выражаем импликацию через дизъюнкцию)}$$
$$\Leftrightarrow (\mathcal{A} \wedge \neg\mathcal{B} \vee \mathcal{A}) \supset \mathcal{A} \quad \text{(отрицание импликации через конъюнкцию)}$$
$$\Leftrightarrow \mathcal{A} \supset \mathcal{A} \quad \text{(по закону поглощения)}$$

Полученная формула общезначима, хоть ее и нет в разд. 1.2.7.

1.2.6. Подставим в формулу (1.2.8-7) вместо подформулы $\mathcal{A}(x)$ формулу $\neg\mathcal{A}(x)$. Получим общезначимую формулу $\forall x\neg\mathcal{A}(x) \supset \neg\mathcal{A}(t)$. Далее:

$$\Leftrightarrow \mathcal{A}(t) \supset \neg\forall x\neg\mathcal{A}(x) \quad \text{(по закону контрапозиции)}$$
$$\Leftrightarrow \mathcal{A}(t) \supset \exists x\neg\neg\mathcal{A}(x) \quad \text{(кванторный аналог закона де Моргана)}$$
$$\Leftrightarrow \mathcal{A}(t) \supset \exists x\mathcal{A}(x) \quad \text{(закон двойного отрицания).}$$

1.2.7. Обозначим: $\mathcal{A}_1 = \mathcal{A}(x)$, $\mathcal{A}_2 = \forall x\mathcal{A}_1$, $\mathcal{B}_1 = \mathcal{B}(y)$, $\mathcal{B}_2 = \forall y\mathcal{B}_1$, $\mathcal{C} = \mathcal{A}_2 \wedge \mathcal{B}_2$, $\mathcal{C}_1 = \mathcal{A}_1 \wedge \mathcal{B}_1$, $\mathcal{D}_1 = \forall y\mathcal{C}_1$, $\mathcal{D} = \forall x\mathcal{D}_1$, $\mathcal{E} = \mathcal{C} \equiv \mathcal{D}$.

Требуется установить общезначимость последней формулы.

При ограничении, установленном для $\mathcal{A}_1$ и $\mathcal{B}_1$, формулы $\mathcal{A}_2 \equiv \mathcal{A}_1$ и $\mathcal{B}_2 \equiv \mathcal{B}_1$ общезначимы, так же как $\mathcal{D}_1 \equiv \mathcal{C}_1$ и $\mathcal{D} \equiv \mathcal{D}_1$.

Проводя короткие серии эквивалентных замен, получаем цепи эквивалентностей: $\mathcal{C} \Leftrightarrow \mathcal{A}_1 \wedge \mathcal{B}_1 \Leftrightarrow \mathcal{C}_1$, откуда $\mathcal{C} \equiv \mathcal{C}_1$ общезначима, $\mathcal{D} \equiv \mathcal{C}_1$ и, наконец, $\mathcal{E} \Leftrightarrow \mathcal{C}_1 \equiv \mathcal{C}_1 \Leftrightarrow T$, т. е. $\mathcal{E}$ общезначима.

1.3.1. Вывод может быть таким:

1) $\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})$ (гипотеза)
2) $(\mathcal{A} \supset (\mathcal{B} \supset \mathcal{C})) \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C}))$ (аксиома A2)
3) $(\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C})$ (1, 2; правило MP)
4) $((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C})) \supset (\mathcal{B} \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C})))$ (A1)
5) $\mathcal{B} \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C}))$ (3, 4; MP)
6) $(\mathcal{B} \supset ((\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{A} \supset \mathcal{C}))) \supset ((\mathcal{B} \supset (\mathcal{A} \supset \mathcal{B})) \supset (\mathcal{B} \supset (\mathcal{A} \supset \mathcal{C})))$ (A2)
7) $(\mathcal{B} \supset (\mathcal{A} \supset \mathcal{B})) \supset (\mathcal{B} \supset (\mathcal{A} \supset \mathcal{C}))$ (5, 6; MP)
8) $\mathcal{B} \supset (\mathcal{A} \supset \mathcal{B})$ (A1)
9) $\mathcal{B} \supset (\mathcal{A} \supset \mathcal{C})$ (8, 7; MP)

1.3.2. Нет, так как формула $\neg\neg\mathcal{A} \supset \mathcal{A}$ выведена из непустого списка гипотез.

1.3.3. Используем некоторые из лемм данного раздела.

а. 1) $\mathcal{A} \supset \mathcal{A}$ (лемма 1)
2) $(\neg\mathcal{A} \supset \mathcal{A}) \supset ((\mathcal{A} \supset \mathcal{A}) \supset \mathcal{A})$ (лемма 6)
3) $(\neg\mathcal{A} \supset \mathcal{A}) \supset \mathcal{A}$ (2, 1; I)

б. 1) $(\mathcal{A} \supset \mathcal{B}) \supset (\neg\mathcal{B} \supset \neg\mathcal{A})$ (лемма 4б)
2) $(\neg\mathcal{B} \supset \neg\mathcal{A}) \supset ((\mathcal{B} \supset \neg\mathcal{A}) \supset \neg\mathcal{A})$ (лемма 6)
3) $(\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{B} \supset \neg\mathcal{A}) \supset \neg\mathcal{A})$ (1, 2; S)
4) $(\mathcal{B} \supset \neg\mathcal{A}) \supset ((\mathcal{A} \supset \mathcal{B}) \supset \neg\mathcal{A})$ (3; R)
5) $(\mathcal{A} \supset \neg\mathcal{B}) \supset (\mathcal{B} \supset \neg\mathcal{A})$ (лемма 4а)
6) $(\mathcal{A} \supset \neg\mathcal{B}) \supset ((\mathcal{A} \supset \mathcal{B}) \supset \neg\mathcal{A})$ (5, 4; S)
7) $(\mathcal{A} \supset \mathcal{B}) \supset ((\mathcal{A} \supset \neg\mathcal{B}) \supset \neg\mathcal{A})$ (6; R)

1.3.4. В теории $\mathbf{P}_1$ возможен следующий вывод из гипотез:

1) $\neg\mathcal{B} \supset \neg\mathcal{A}$ (гипотеза)
2) $\mathcal{A}$ (гипотеза)
3) $\mathcal{A} \supset (\neg\mathcal{B} \supset \mathcal{A})$ ($A1_1$)
4) $\neg\mathcal{B} \supset \mathcal{A}$ (2, 3; MP)
5) $(\neg\mathcal{B} \supset \mathcal{A}) \supset ((\neg\mathcal{B} \supset \neg\mathcal{A}) \supset \neg\neg\mathcal{B})$ ($A9_1$)
6) $(\neg\mathcal{B} \supset \neg\mathcal{A}) \supset \neg\neg\mathcal{B}$ (4, 5; MP)
7) $\neg\neg\mathcal{B}$ (1, 6; MP)
8) $\neg\neg\mathcal{B} \supset \mathcal{B}$ ($A10_1$)
9) $\mathcal{B}$ (7, 8; MP)

Таким образом,

$$\neg\mathcal{B} \supset \neg\mathcal{A},\ \mathcal{A} \vdash_{\mathbf{P}_1} \mathcal{B}.$$

Дважды применяя теорему о дедукции, получим

$$\vdash_{\mathbf{P}_1} (\neg\mathcal{B} \supset \neg\mathcal{A}) \supset (\mathcal{A} \supset \mathcal{B}).$$

1.3.5. Из результата предыдущего упражнения следует, что в теории $\mathbf{P}_1$ выводимы все аксиомы, а следовательно, и все теоремы теории $\mathbf{P}_2$. Теория $\mathbf{P}_2$ содержит все аксиомы теории $\mathbf{P}$, поэтому в ней выводимы все теоремы этой теории, в частности любая формула, имеющая схему $A9_1$ (упражнение 1.3.3б) или схему $A10_1$ (лемма 1.3.7). Таким образом, все аксиомы, а вместе с ними — все теоремы, теории $\mathbf{P}_1$ выводимы в теории $\mathbf{P}_2$.

1.4.1. Если формула $\mathcal{A}$ не содержит свободных вхождений переменной y, то $\mathcal{A} \equiv \forall y\mathcal{A}$ в силу правил Gen и A. Используя эту эквивалентность, правило контрапозиции, результат примера 3.2–2 и сокращение $\exists y\mathcal{B}$, можно выписать следующую цепь эквивалентностей:

$$\forall y(\mathcal{B} \supset \mathcal{A}) \Leftrightarrow \forall y(\neg\mathcal{A} \supset \neg\mathcal{B}) \Leftrightarrow \forall y\neg\mathcal{A} \supset \forall y\neg\mathcal{B} \Leftrightarrow \neg\mathcal{A} \supset \forall y\neg\mathcal{B}$$
$$\Leftrightarrow \neg\forall y\neg\mathcal{B} \supset \mathcal{A} \Leftrightarrow \exists y\mathcal{B} \supset \mathcal{A}.$$

1.4.2. Построим вывод:

1) $\forall x\mathcal{A}(x) \supset \mathcal{A}(y)$ (A4)
2) $\forall y(\forall x\mathcal{A}(x) \supset \mathcal{A}(y))$ (Gen)
3) $\forall y(\forall x\mathcal{A}(x) \supset \mathcal{A}(y)) \supset (\forall x\mathcal{A}(x) \supset \forall y\mathcal{A}(y))$ (A5)
4) $\forall x\mathcal{A}(x) \supset \forall y\mathcal{A}(y)$ (2, 3; MP)

Аналогично выводится формула $\forall y\mathcal{A}(y) \supset \forall x\mathcal{A}(x)$. Вывод результата завершаем с помощью тавтологии $B_1 \supset (B_2 \supset B_1 \wedge B_2)$, в которой вместо B_1 и B_2 подставлены две выведенные перед этим формулы, и правила MP.

1.5.1. К требуемому результату приводит цепочка эквивалентностей:

$$x = y \Leftrightarrow \forall z\,(z \in x \equiv z \in y) \Leftrightarrow \forall z\,((z \in x \supset z \in y) \wedge (z \in y \supset z \in x))$$
$$\Leftrightarrow \forall z\,(z \in x \supset z \in y) \wedge \forall z\,(z \in y \supset z \in x) \Leftrightarrow x \subseteq y \wedge y \subseteq x.$$

1.5.2.

$$x \subset y \Leftrightarrow x \subseteq y \wedge x \neq y \Leftrightarrow x \subseteq y \wedge \neg\, x = y$$
$$\Leftrightarrow x \subseteq y \wedge \neg(x \subseteq y \wedge y \subseteq x) \Leftrightarrow x \subseteq y \wedge (\neg(x \subseteq y \vee \neg\, y \subseteq x)$$
$$\Leftrightarrow x \subseteq y \wedge \neg\, x \subseteq y \vee x \subseteq y \wedge \neg\, y \subseteq x \Leftrightarrow F \vee x \subseteq y \wedge \neg\, y \subseteq x$$
$$\Leftrightarrow x \subseteq y \wedge \neg\, y \subseteq x \Leftrightarrow \forall z\,(z \in x \supset z \in y) \wedge \neg\forall z\,(z \in y \supset z \in x)$$

$\Leftrightarrow \forall z\,(z \in x \supset z \in y) \wedge \exists z\,(z \in y \wedge \neg\, z \in x)$
$\Leftrightarrow \forall z\,(z \in x \supset z \in y) \wedge \exists z\,(z \in y \wedge z \notin x)$.

1.5.3. Пусть R — асимметричное отношение на множестве A. Для произвольно взятых $x \in A$ и $y \in A$ при $x \neq y$ из $xRy \wedge yRx$ следует xRy и yRx (по определению конъюнкции), а также $y\overline{R}x$ (из xRy по асимметричности отношения R). Полученное противоречие заставляет заключить, что $x = y$ и $x\overline{R}x$ — только в этом случае импликация $xRx \supset x\overline{R}x$ истинна. Оба свойства доказаны.

1.5.4. При движении вдоль пути p_1 одна из вершин контура c (в том числе и вершина v) может встретиться раньше конца пути. Появится выбор: двигаться ли дальше вдоль пути p_1 или свернуть на контур c. При втором решении, чтобы собрать путь p, придется обойти весь контур, никуда не сворачивая (простой контур не может быть собран из других контуров), пройти остаток пути p_1 и весь путь p_2. При первом решении ситуация может повториться. Но если путь p_1 будет пройден до конца, то, чтобы не искушать судьбу (ведь путь p_2 может и не содержать вершин контура), следует обойти контур c и пройти путь p_2.

1.5.5. а. Из определений получаем:

$y \in \mathrm{Rng}(F) \Leftrightarrow y \in B \wedge \exists x\,(x \in A \wedge xFy)$
$\Leftrightarrow \exists x\,(x \in A \wedge y \in B \wedge xFy) \Leftrightarrow \exists x\,(x \in A \wedge y \in \hat{F}(x))$,
$y \in \hat{F}(X) \Leftrightarrow y \in \bigcup_{x \in X} \hat{F}(x) \Leftrightarrow \exists x\,(x \in X \wedge y \in \hat{F}(x))$.

Отсюда логически следует, что $\hat{F}(A) = \mathrm{Rng}(F)$.

б. Прямо следует из определения и пункта а.

в. Если найдется x, такой что $y = F(x)$, $y_1 = F(x)$ и $y \neq y_1$, то $x \in \check{F}(y)$, $x \in \check{F}(y_1)$ и $\check{F}(y) \cap \check{F}(y_1) \neq \varnothing$, что противоречит условию. Следовательно, $y = y_1$, т. е. соответствие F однозначно.

1.5.6. Если перейти от индексного стиля к функциональному, то $x = b_i$ примет вид $x = b(i)$, а $x = a_{k_i}$ — вид $x = a(k(i))$, иначе $x = (a \circ k)(i)$. Отсюда $b = a \circ k$.

1.5.7. Для каждого значения аргумента x функция $f(x)$ может принимать 3 значения: 1, 2 и 3. Каждое из этих значений для $x = 1$ может сочетаться с каждым для $x = 2$. Все возможные сочетания собраны в табличке

f(1)	1	1	1	2	2	2	3	3	3
f(2)	1	2	3	1	2	3	1	2	3

1.5.8. Предположим, что x и y — два различных R-наименьших элемента в множестве A. Тогда должно быть xRy (так как $y \in A$, $y \neq x$ и x — R-наименьший элемент в A) и yRx (аналогично). Но тогда по антисимметричности R должно быть $x = y$, что противоречит допущению.

1.5.9. Может. Например, в множестве X всех непустых подмножеств множества A, содержащего не менее двух элементов, любое одноэлементное подмножество множества A является $\subseteq$-минимальным элементом.

1.5.10. а. Пусть $x \in A$, $y \in A$ и $x \neq y$. В множестве $\{x, y\}$ должен найтись R-наименьший элемент. Если это x, то xRy, если y — то yRx.

б. Ясно, что каждое вполне упорядоченное множество подобно самому себе (при тождественном отображении на себя). Если биективное отображение F сохраняет порядок, то и обратное отображение биективно и сохраняет порядок: при $F(x)R'F(y)$ не может иметь места ни $x = y$ (по однозначности F), ни yRx (что вело бы к $F(y)R'F(x)$), стало быть xRy (по пункту а). Наконец, если F биективно и с сохранением порядка отображает A на A', а G — A' на A'', то композиция $G \circ F$ биективно отображает A на A'', также сохраняя порядок.

1.5.11. Множество $\mathbf{N}^2$ состоит из всех пар вида (m, n), где $m, n \in \mathbf{N}$. Рассмотрим отображение $f : \mathbf{N}^2 \to \mathbf{N}$, где $f(m, n) = 2^m(2n+1)-1$. Это отображение биективно, так как каждое положительное натуральное число единственным способом может быть представлено в виде произведения степени двойки на нечетное число. Следовательно, $\overline{\overline{\mathbf{N}^2}} = \overline{\overline{\mathbf{N}}}$.

1.5.12. Пусть $A = \{a_1, \ldots, a_m\}$. Расположим все элементы множества A^* (т. е. все конечные слова в алфавите A) в следующей последовательности:

$$\lambda,\ a_1,\ \ldots,\ a_m,\ a_1a_1,\ \ldots,\ a_1a_m,\ \ldots,\ a_ma_1,\ \ldots,\ a_ma_m,\ a_1a_1a_1,\ \ldots$$

и перенумеруем их в этом порядке. Другими словами, поставим в соответствие каждому слову $a_{i_1}a_{i_2}\ldots a_{i_n}$ число $i_1m^{n-1} + i_2m^{n-2} + \ldots + i_n$. Это и будет биективное отображение A^* на $\mathbf{N}$.

1.5.13. Нет. У пустого множества нет элементов, но есть одно подмножество — оно само. У одноэлементного множества два подмножества — оно само и $\varnothing$.

1.5.14. В отображении g отсутствуют прообразы последовательностей, содержащих, начиная с некоторого места, только нули (их множество счетно). Выделим прообразы аналогичных последовательностей, но с единицами, в их последовательности передвинем все элементы на четные места, а на нечетные поместим недостающие прообразы.

2.1.1. Формулировка задачи содержит готовый ответ. Поэтому самое большее, что имеет смысл сделать, — это:

program *Print_e*;
  const $e = 2.71828$;
begin *Writeln*$('e =', e : 8 : 5)$ **end**.

3.2.1. Доказательство формулы (3.2.5–14):

$$\begin{aligned} y = (b(f\cdot g))(x) &\Leftrightarrow\ b(x)\ \wedge\ y = (f\cdot g)(x)\\ &\Leftrightarrow\ b(x)\ \wedge\ \exists x'\,(x' = g(x) \wedge y = f(x'))\\ &\Leftrightarrow\ \exists\, x'\,(b(x)\ \wedge\ x' = g(x)\ \wedge\ y = f(x'))\\ &\Leftrightarrow\ \exists x'\,(x' = bg(x) \wedge y = f(x'))\ \Leftrightarrow\ y = (f\cdot bg)(x) \end{aligned}$$

Доказательство (3.2.5–16):

$$\begin{aligned} y = \big(\big(\textstyle\bigcup_{n\geqslant 0} f_n\big)\cdot g\big)(x)\ &\Leftrightarrow\ \exists x'\,(x' = g(x) \wedge y = \big(\textstyle\bigcup_{n\geqslant 0} f_n\big)(x'))\\ &\Leftrightarrow\ \exists x'\,(x' = g(x) \wedge \exists n\,(n \geqslant 0 \wedge y = f_n(x')))\\ &\Leftrightarrow\ \exists\, x'\,\exists n\,(n \geqslant 0 \wedge x' = g(x)\ \wedge\ y = f_n(x'))\\ &\Leftrightarrow\ \exists n\,(n \geqslant 0 \wedge (\exists x'\,(x' = g(x) \wedge y = f_n(x'))))\\ &\Leftrightarrow\ \exists n\,(n \geqslant 0 \wedge y = (f_n\cdot g)(x))\ \Leftrightarrow\ y = \big(\textstyle\bigcup_{n\geqslant 0}(f_n\cdot g)\big)(x). \end{aligned}$$

3.2.2. Доказательство (3.2.7–31):

$$\begin{aligned} s{\uparrow}(p)(w_1) &\Leftrightarrow \exists\, w\,(p(w) \wedge (w_1 = s1(w) \vee w_1 = s2(w))) \\ &\Leftrightarrow \exists\, w\,(p(w) \wedge w_1 = s1(w) \vee p(w) \wedge w_1 = s2(w)) \\ &\Leftrightarrow \exists\, w\,(p(w) \wedge w_1 = s1(w)) \vee \exists\, w\,(p(w) \wedge w_1 = s2(w)) \\ &\Leftrightarrow s1{\uparrow}(p)(w_1) \vee s2{\uparrow}(p)(w_1) \Leftrightarrow (s1{\uparrow}(p) \vee s2{\uparrow}(p))(w_1). \end{aligned}$$

3.2.3. Доказательство (3.2.7–33):

$$\begin{aligned} ps{\uparrow}(p')(w_1) &\Leftrightarrow \exists\, w\,(p'(w) \wedge w_1 = ps(w)) \\ &\Leftrightarrow \exists\, w\,(p'(w) \wedge p(w) \wedge w_1 = s(w)) \\ &\Leftrightarrow \exists\, w\,((p \wedge p')(w) \wedge w_1 = s(w)) \Leftrightarrow s{\uparrow}(p \wedge p')(w_1). \end{aligned}$$

Доказательство (3.2.7–34):

$$\begin{aligned} ps{\downarrow}(q)(w) &\Leftrightarrow \exists\, w_1\,(q(w_1) \wedge w_1 = ps(w)) \\ &\Leftrightarrow \exists\, w_1\,(q(w_1) \wedge p(w) \wedge w_1 = s(w)) \\ &\Leftrightarrow p(w) \wedge \exists\, w_1\,(q(w_1) \wedge w_1 = s(w)) \\ &\Leftrightarrow p(w) \wedge s{\downarrow}(q)(w) \Leftrightarrow (p \wedge s{\downarrow}(q))(w). \end{aligned}$$

Список литературы

[1] Алгоритмический язык Алгол-60. / Пересмотр. сообщ. Пер. с англ. — М.: Мир, 1965.

[2] Арнольд К., Гослинг Дж. Язык программирования Java. / Пер. с англ. — СПб.: Питер, 1997. — 304 с.

[3] Базисный рефал. Описание языка и основные приемы программирования. — М.: ЦНИПИАСС, 1974.

[4] Бауэр Ф. Л., Гнац Р., Хилл У. Информатика. Задачи и решения. / Пер. с нем. — М.: Мир, 1978.

[5] Бауэр Ф. Л., Гооз Г. Информатика. Вводный курс. / Пер. с нем. — М.: Мир, 1976.

[6] Березин Б. И., Березин С. Б. Начальный курс С и С++. — М.: Диалог-МИФИ, 2000. — 288 с.

[7] Бриллюэн Л. Наука и теория информации. / Пер. с англ. — М.: Наука, 1960. — 392 с.

[8] Васильев В. А. Язык Алгол-68. Основные понятия. — М.: Наука, 1972.

[9] Вирт Н. Алгоритмы и структуры данных. / Пер. с англ. — М.: Мир, 1989. — 360 с.

[10] Вирт Н. Систематическое программирование. Введение. / Пер. с англ. — М.: Мир, 1977. — 183 с.

[11] Вирт Н. Язык программирования Паскаль (пересмотр. сообщ.). / Пер. с англ. — В кн.: Алгоритмы и организация решения экономических задач. — М.: Статистика, 1977, вып. 9, с. 52–86.

[12] Гейтинг А. Интуиционизм. Введение. / Пер. с англ. под ред. и с комментариями А. А. Маркова. — М.: Мир, 1965. — 200 с.

[13] Дал У.-И. Языки для моделирования систем с дискретными событиями. — В кн.: Языки программирования. / Пер. с англ. — М.: Мир, 1972, с. 344–403.

[14] Дал У., Дейкстра Э., Хоор К. Структурное программирование. / Пер. с англ. — М.: Мир, 1975.

[15] Дал У.-И., Мюрхауг Б., Нюгорд К. СИМУЛА 67. Универсальный язык программирования. — М.: Мир, 1969.

[16] Дейкстра Э. Дисциплина программирования. / Пер. с англ. — М.: Мир, 1978.

[17] Дейкстра Э. Взаимодействие последовательных процессов. — В кн.: Языки программирования. / Пер. с англ. — М.: Мир, 1972, с. 9–86.

[18] Евстигнеев В. А., Касьянов В. Н. Толковый словарь по теории графов в информатике и программировании. — Новосибирск: «Наука», Сибирское предприятие РАН, 1999. — 291 с.

[19] Ершов А. П. Избранные труды. / ИСИ СО РАН, отв. ред. И. В. Поттосин. — Новосибирск: ВО «Наука», 1994. — 416 с.

[20] Керниган Б., Ричи Д. М. Язык программирования С. / Пер. с англ. — М.: Финансы и статистика, 1985.

[21] Клини С. К. Введение в метаматематику. / Пер. с англ. — М.: ИЛ, 1957. — 526 с.

[22] Клини С. К. Математическая логика. / Пер. с англ. — М.: Мир, 1973. — 480 с.

[23] Кнут Д. Искусство программирования для ЭВМ, т. 1, Основные алгоритмы. / Пер. с англ. — М.: Мир, 1976.

[24] Колмогоров А. Н. Три подхода к определению понятия «количество информации». — Проблемы передачи информации, 1965, **1**, вып. 1, с. 3–11.

[25] Кузнецов А. П., Адельсон-Вельский Г. М. Дискретная математика для инженера. — М.: Энергия, 1980. — 342 с.

[26] Лавров С. С. Введение в программирование. / 2-е изд. — М.: Наука, 1977. — 367 с.

[27] Лавров С. С. Основные понятия и конструкции языков программирования. — М.: Финансы и статистика, 1982. — 80 с.

[28] Лавров С. С. Лекции по теории программирования: Уч. пособие. / Министерство общего образования Российской Федерации. Санкт-Петербургский государственный технический университет. — СПб.: изд-во НЕСТОР, 1999. — 107 с.

[29] Лавров С. С., Силагадзе Г. С. Автоматическая обработка данных. Язык Лисп и его реализация / Библиотечка программиста. — М.: Наука, Физматлит, 1978. — 176 с.

[30] Линдси Ч., ван дер Мюйлен С. Неформальное введение в Алгол-68. / Пер. с англ. — М.: Мир, 1973.

[31] Марков А. А., Нагорный Н. М. Теория алгорифмов. — М.: Наука, Физматлит, 1984. — 432 с.

[32] Мартин-Лёф П. Очерки по конструктивной математике. / Пер. с англ. — М.: Мир, 1975. — 136 с.

[33] Мартынюк В. В. Экономное построение транзитивного замыкания бинарного отношения. / Ж. вычисл. матем. и матем. физ., 1962, **2**, 4, с. 723–725.

[34] Мендельсон Э. Введение в математическую логику. / Пер. с англ., 2-е изд. — М.: Наука, Физматлит, 1976. — 320 с.

[35] Минский М. Вычисления и автоматы. / Пер. с англ. — М.: Мир, 1971. — 364 с.

[36] Неформальное введение в C++ и Turbo Vision. — СПб.: галерея «Петрополь», 1992. — 383 с.

[37] Пересмотренное сообщение об Алголе 68. / Пер. с англ. — М.: Мир, 1979.

[38] Страуструп Б. Язык программирования C++. / Пер с англ. 3-е изд. — М.: Бином, СПб.: Невский диалект, 1999. — 990 с.

[39] Турбо Паскаль 7.0. / 2-е изд. — Киев: торгово-издательское бюро BHV, 1997. — 444 с.

[40] Универсальный язык программирования PL/1. / Пер. с англ. — М.: Мир, 1968.

[41] Френкель А., Бар-Хиллел И. Основания теории множеств. / Пер. с англ. — М.: ИЛ, 1966. — 555 с.

[42] Фути К., Судзуки Н. Языки программирования и системотехника СБИС. / Пер. с яп. — М.: Мир, 1988. — 224 с.

[43] Хенни Ф., Стирнз Р. Двухленточное моделирование многоленточной машины Тьюринга. / Пер. с англ. — В кн.: Проблемы математической логики / Ред. В. А. Козмидиади, А. А. Мучник. — М.: Мир, 1970.

[44] Холл П. Вычислительные структуры. Введение в нечисленное программирование. / Пер. с англ. — М.: Мир, 1978.

[45] Хоор К. Обработка записей. — В кн.: Языки программирования. Пер. с англ. — М.: Мир, 1972, с. 278–343.

[46] Цейтин Г. С. Некоторые черты языка для системы программирования, проверяющей доказательства. / В кн.: Теория программирования. Труды симпозиума. — Новосибирск: ВЦ СО АН СССР, 1972, ч. II, с. 234–248.

[47] Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем / Пер. с англ. — М.: Наука, Физматлит, 1983. — 360 с.

[48] Яглом А. М., Яглом И. М. Вероятность и информация. / 3-е изд. — М.: Наука, 1973. — 511 с.

[49] Communications of the ACM, v. 38, Oct. 1995.

[50] Dijkstra E. W. Go to statement considered harmful. / Comm. ACM, vol. 11, 1968, № 3, p. 147–148.

[51] Farber D. J. et al. SNOBOL, a string manipulation language. / J. ACM, vol. 11, 1964, № 1, p. 21–30.

[52] Goldberg A., Robson D. SMALLTALK-80 — The language and its implementation. — Reading, Mass.: Addison-Wesley, 1983.

[53] Hoare C. H. R. An axiomatic basis for computer programming. / Comm. ACM, vol. 12, 1969, № 10, p. 576–583.

[54] Jensen K., Wirth N. PASCAL: User manual and report. — New York, Heidelberg, Berlin: Springer Verlag, 1974.

[55] OOPSLA'95 — ACM SIGPLAN Notice, v. 30, № 10, Oct. 1995, xiii+480+3 pp.

[56] Partial evaluation and mixed computation, ed. by D. Bjørner, A. P. Ershov and N. D. Jones — Amsterdam a. o.: North-Holland, 1988, xxxii+625 pp.