Информатика

HEIDELBERGER TASCHENBÜCHER
SAMMLUNG INFORMATIK

Herausgegeben von F. L. Bauer, G. Goos und M. Paul

Band 80 | Band 91

INFORMATIK

Eine einführende Übersicht

FRIEDRICH L. BAUER

ord. Professor der Mathematik und Informatik
an der Technischen Universität München

GERHARD GOOS

ord. Professor der Informatik
an der Universität Karlsruhe (TH)

Erster Teil | Zweiter Teil
Kapitel 1—4 | Kapitel 5—8

Dritte Auflage
völlig neu bearbeitet und erweitert
von F. L. Bauer

Springer-Verlag
Berlin Heidelberg New York Tokyo

1982 | 1984

Ф. Л. Бауэр, Г. Гооз

ИНФОРМАТИКА

вводный курс

Второе, полностью переработанное и расширенное издание

В двух частях

Часть 1

Перевод с немецкого
М. К. Валиева, В. Г. Кербеля
и
В. К. Сабельфельда
под редакцией
А. П. Ершова

Москва «Мир» 1990

ББК 22.19 + 32.97
Б 29
УДК 681.142.2

Бауэр Ф. Л., Гооз Г.

Б 29 Информатика. Вводный курс: В 2-х ч. Ч.1. Пер. с нем.— М.: Мир, 1990. — 336 с., ил.

ISBN 5-03-002099-3

Учебный курс, представляющий собой перевод с 3-го, полностью переработанного и расширенного издания написанного известными специалистами (ФРГ) (Второе немецкое издание было переведено на русский (М Мир, 1976)) Изложение отличается глубокой методической проработанностью материала и применением концептуального подхода к преподаванию основ информатики Фундаментальные понятия программирования вводятся в книге как исходные, получают затем формализацию и раскрываются в виде конструкций языков программирования разного уровня Много содержательных и изящных примеров решения конкретных задач с использованием алгола-68 и паскаля

В ч 1 входят предисловия и гл. 1—4.

Для студентов, изучающих программирование, научных работников различных специальностей, использующих ЭВМ в своей работе.

Б $\frac{1602110000-394}{041(01)-90}$ 27—89 ББК 22.19 + 32.97

Редакция литературы по математическим наукам

ISBN 5-03-002099-3 } (русск.)
ISBN 5-03-000298-7 }
ISBN 3-540-11722-9 } (нем.)
ISBN 3-540-13121-3 }

© by Springer-Verlag Berlin Heidelberg 1971, 1973, 1974, 1982 1984
All Rights Reserved
Authorized translation from German language edition published by Springer-Verlag Berlin Heidelberg New York Tokyo
© перевод на русский язык, В К Сабельфельд М К Валиев, В Г. Кербель, 1990

Предисловие к русскому изданию

Это второе издание „Информатики" представляет собой перевод уже третьего немецкого издания, практически полностью переработанного профессором Ф. Л. Бауэром с учётом прогресса, достигнутого в программировании за последние годы.

Учебник Ф. Л. Бауэра и Г. Гооза пользуется большой популярностью в странах, говорящих по-немецки. Первое русское издание книги (М.: Мир, 1976) получило признание и у советского читателя. Успех учебника объясняется, по-видимому, тем, что авторы *одними из первых* удачно применили концептуальный подход в преподавании основ программирования. Фундаментальные понятия информатики вводятся в книге как исходные, а не как производные тех функций, которые реализуются теми или иными вычислительными устройствами. Другая принципиальная особенность курса, отличающая его от традиционных учебников по программированию, тесно связана с пропагандируемым авторами трансформационным стилем программирования.

Сегодня уже многие сознают, что преподавание основ программирования не должно ограничиваться только тем, чтобы обучать синтаксису и семантике одного или нескольких языков программирования. Вместе с освоением языковых конструкций обучаемый должен получить первые навыки разработки программ, которую авторы с полным на то основанием трактуют как процесс систематического и вполне познаваемого перехода от неалгоритмической постановки задачи к эффективной программе для её решения Методология такого систематического перехода излагается в третьей главе нового издания.

Много внимания в новом издании уделено технике рекурсивного программирования, вовсе не затрагивавшейся в предыдущих изданиях. Во второй главе изложены основные принципы функционального стиля программирования, получившего в последние годы широкое распространение; здесь же на содержательных примерах демонстрируются изящные рекурсивные формулировки для решения конкретных задач.

Удачной методической находкой следует считать описанную во второй главе „машину обработки формуляров", представляющую собой весьма простую абстрактную модель исполнителя рекурсивных программ Формуляр — это обобщение понятия

формулы, имеющее наглядное графическое представление, так что понятия, связанные с разными способами выполнения рекурсивных программ, оказываются представленными на страницах книги в „живых" картинках — при помощи таких наглядных процессов, как копирование картинок, их соединение друг с другом, заполнение граф в формулярах.

Среди других новых понятий, появившихся и подробно обсуждаемых в новом издании, особо следует отметить понятие вычислительной структуры, представляющее собой по сути дела конкретизацию понятия абстрактного типа данных. Как работает это относительно новое для практики программирования понятие, показано в книге на многочисленных примерах конкретных вычислительных структур, а также при описании способов конструирования сложных вычислительных структур из более простых.

В отличие от первого русского издания во втором была использована авторская версия языка алгол-68, без перевода служебных слов с английского на русский. Такое решение было принято с целью избежать разнобоя: в новом издании примеры программ приводится не только на алголе-68, но и на паскале, для которого русской версии вообще нет; кроме того, русская версия алгола-68 (см. [05]) не получила, к сожалению, широкого распространения.

Важно, что книга хорошо стыкуется с введённым сейчас в нашей средней школе курсом основ информатики и вычислительной техники. Поэтому хочется надеяться, что переиздание „Информатики" будет способствовать наметившимся положительным сдвигам в перестройке нашей системы образования, в какой-то степени поможет решить задачу повышения качества подготовки специалистов по применению вычислительной техники.

* * *

Каждое из трёх немецких изданий выходило в двух частях, с небольшим временны́м разрывом между первой и второй и с отдельными предисловиями к каждой части. Это издание готовилось в одном томе. Поэтому мы собрали все предисловия вместе в начале книги. Однако по техническим причинам на стадии корректуры нам всё-таки пришлось тоже разбить книгу на две части. Отсюда сплошная нумерация страниц в частях.

Перевод предисловий, приложений и глав 1, 3, 5 выполнен В. К. Сабельфельдом, глав 2, 4 и 6 — В. Г. Кербелем, глав 7 и 8 — М. К. Валиевым.

А. П. Ершов

В. К. Сабельфельд

От переводчиков

Инициатором обоих русских изданий был академик А. П. Ершов. Второе издание стало возможным во многом благодаря именно его усилиям. Учёный с мировым именем, А. П. Ершов внёс значительный и определяющий вклад в становление информатики в СССР. Недавняя смерть А. П. Ершова глубоко поразила нас. Мы посвящаем наш перевод его памяти.

М. Валиев
В. Кербель
В. Сабельфельд

Из предисловия к первому изданию части I

Informatik — это немецкое название для computer science [1] — области знания, которая сложилась в самостоятельную научную дисциплину в шестидесятые годы, прежде всего в США, а также в Великобритании. В последнее время изучение информатики в ФРГ находится на быстром подъёме, чему в немалой степени способствует Федеральное министерство образования и науки. Данная книга представляет собой вводный курс информатики, который согласуется с рекомендациями обществ GAMM и NTG [2]. Она может служить в качестве пособия для начального годового курса лекций. Она и возникла из таких лекций, которые читались на отделении математики Мюнхенского технического университета с 1967 года в связи с введением в университете регулярного изучения информатики.

Изложение построено по принципу от общего к частному. Фундаментальные понятия программирования мы вводим в качестве исходных, вместо того чтобы, как это часто делают, выводить их из специальных функций машины. Преимущество такого способа изложения, освобождённого от случайностей технического развития и тем самым позволяющего получать более „жизнестойкие“ утверждения, нельзя считать незначительным; ещё в большей степени, как нам кажется, этот способ оправдан тем, что мы заставляем начинающего задумываться, вместо того чтобы подавать ему готовые решения. Этот способ изложения недавно получил распространение и в США (под названием top-down-teaching [3]), в частности благодаря усилиям Элана Перлиса. Наконец, развитие языков программирования в сторону формализации семантики также нацелено на то, чтобы шаг за шагом сводить сложное к простому.

Наряду с общим обзором информатики в книге вскрываются связи между её отдельными специальными разделами.

В написание этой книги внесли конструктивный вклад наши коллеги К. Замельзон, П. Пауль и Ф. Пайшль. Первый вариант рукописи подготовила в 1967/68 учебном году г-жа Д. Майзон. При разработке упражнений к лекциям, особенно при составлении задач, нам помогали Р. Гнац и Х. Вальтер, последний из которых отредактировал также второй вариант курса (1968/69 учебный год). Замельзон читал этот вводный курс в 1969/70 учебном году, и мы получили от него ценную критику. Кроме того, полезные замечания по отдельным главам сделали кол-

---

[1] Буквально: компьютерная наука (*англ.*) — *Прим. перев.*

[2] GAMM — общество прикладной математики и механики; NTG — общество техники связи. — *Прим. перев.*

[3] Обучение сверху вниз (*англ.*). — *Прим. перев.*

леги И. Айкель, П. Дойсен, В. Хан и г-жа У. Хилл. Первую главу просмотрели также коллеги В. Кайдель (Эрланген) и Х. Цеманек (Вена). Мы от души благодарим всех, кто сотрудничал с нами, у нас просто нет возможности назвать всех поимённо. Это относится не только к секретаршам и сотрудникам, помогавшим нам при чтении корректур.

Особо должны мы упомянуть Х. Вёснера, который проверил алгольные примеры на синтаксическую правильность. За период, когда старший из авторов[1] был председателем терминологической комиссии специального комитета по стандартизации обработки информации, он многое вынес из проводившихся там дискуссий, и ему хочется выразить свою благодарность участникам этих дискуссий.

В тяжёлые послевоенные годы у нас, к счастью, нашлись три человека, обладавшие мужеством верить в возрождение научной работы в новой и до 1955 года весьма ограниченно развивавшейся области вычислительного дела. Это — Альвин Вальтер, Ганс Пилоти и Роберт Зауэр. Им, подготовившим появление информатики в нашей стране, посвящена первая часть этой книги.

Мюнхен, лето 1970 г. *Ф. Л. Бауэр, Г. Гооз*

Из предисловия к первому изданию части 2

Эта вторая часть „Информатики“ завершает наш вводный курс. Она посвящена памяти Хайнца Рутисхаузера (30.01.1918—10.11.1970). Рутисхаузер был одним из первых, кто осознал и использовал возможности цифровых вычислительных машин, выходящие за рамки числовых вычислений, и тем способствовал становлению информатики.

Мюнхен и Карлсруэ, весна 1971 г. *Ф. Л. Бауэр, Г. Гооз*

Предисловие ко второму изданию части 1

В настоящем издании улучшены многие детали изложения. Мы получили многочисленные замечания относительно первого издания и хотим выразить признательность всем их авторам. Особая наша благодарность — г-же д-ру Х. Фогг за неутомимую помощь при переработке текста.

Мюнхен и Карлсруэ, весна 1973 г. *Ф. Л. Бауэр, Г. Гооз*

---

[1] Ф. Л. Бауэр. — *Прим. перев.*

Предисловие ко второму изданию части 2

Во втором издании удалось улучшить многие детали изложения. В связи с этим мы благодарим всех, высказавших нам свои отзывы о первом издании. Раздел 6.2 об управлении данными изложен полностью заново, с включением в него материала прежнего раздела 6.1.3. За многочисленные критические замечания, касающиеся этого нового варианта изложения, мы признательны целому ряду коллег, особенно П. Локкеману. Наконец, добавлены раздел о языке программирования паскаль и приложение об устройствах ввода/вывода данных.

Мюнхен и Карлсруэ, весна 1974 г. *Ф. Л. Бауэр, Г. Гоос*

Предисловие к третьему изданию части 1

Ввиду того взлёта достижений, который наблюдался в информатике за двенадцать лет, прошедшие с момента выхода первого издания этой книги, при подготовке настоящего, третьего её издания (31-я — 34-я тысячи экземпляров) потребовалась полная переработка текста.

Чисто внешне читателю, по-видимому, больше всего бросится в глаза то обстоятельство, что примеры программ записаны теперь в два столбца, на алголе-68 и на паскале. В пользу привлечения языка паскаль говорило не только его широкое распространение начиная с 1975 года, но и прежде всего то, что он удачно компенсирует некоторые недостатки алгола-68, скажем в части, касающейся структур данных; с другой стороны, спроектированный не зависящим от процесса перевода алгол-68 в некоторых местах имеет более прозрачную нотацию — хотя из-за своей явной вычурности в ряде областей он просто не смог утвердиться. Как уже было в первом издании с алголом-68, так и здесь с паскалем — правда, в гораздо более скромных масштабах — пришлось обреза́ть дикие заросли языковых украшательств. Добавлять же пришлось, по существу, только некоторые конструкции для совместного исполнения.

В мирном сосуществовании алгола-68 и паскаля отражается, главным образом, перемена, происшедшая в семидесятых годах и состоящая в формировании общего и абстрактного, не зависящего от используемой нотации базиса понятий для управляющих структур и (с некоторым отставанием) для структур данных. С семантической точки зрения, исходя из этого базиса, легко получить как алгол-68, так и паскаль (и не только их), причём при таком подходе для алгола-68 неизбежными оказываются некоторые нововведения (скажем, вычёркивание описа-

ния тождества из списка основных понятий); некоторая «узкотрудность» паскаля (скажем, отсутствие описаний промежуточных результатов и условных формул) также требует определённых мер, которые, собственно, и вырисовываются в некоторых версиях паскаля (MESA и др.).

Это значит, что за всем этим стоит абстрактная, не зависящая от машины операционная семантика, основанная на записи и упрощении формул, с надстройкой из абстрактных типов данных и подстановок термов (см. F. L. Bauer „Algorithms and Algebra", *in:* Algorithms in Modern Mathematics and Computer Science (Urgench Symposium 1979), LNCS 122[1]). Однако в книге для начинающих нельзя начинать с такой теории. Вместо неё используется неформальная, интуитивно ясная модель «машины обработки формуляров». Она представляет собой, собственно говоря, прототип машины так называемого «аппликативного стиля», модель, на базе которой можно строить самые разнообразные модели машин, ориентированные как на организацию потока управления, так и на организацию потока данных.

Тем самым книга призвана внести вклад в необходимую корректировку содержания курса по информатике. Здесь надо постоянно взвешивать, что́ в курсе должно быть опущено как несущественные для начинающего детали или же как чисто теоретические рассмотрения.

Читатель, которому мешает соседство алгольной (в книге всегда слева) и паскалевской (справа) редакций программ, может принимать к сведению только какую-нибудь одну из двух колонок; если же он обратит свой взгляд на обе, то иногда сможет кое-что извлечь также из их сравнения, пусть это будет даже лишь тот факт, что разница просто несущественна. В частности, преподаватель, который рекомендует эту книгу студентам в качестве учебного пособия, может на лекциях, скажем по соображениям, связанным с практикой обучения, сосредоточить внимание на одной из двух редакций, сохраняя вторую в качестве побочной или же оставляя её студентам для самостоятельного изучения.

Точно так же, исходя из потребностей семинарских занятий, преподаватель может изменить порядок расположения материала в курсе. Некоторые указания в этом отношении даны в подстрочных примечаниях к заголовкам соответствующих параграфов.

В этой книге намеренно проводится некая средняя линия между радикально современным и «консервативным» построе-

---

[1] Имеется перевод: Бауэр Ф. Л. Алгоритмы и алгебра. — В сб.: Труды международного симпозиума „Алгоритмы в современной математике и её приложениях", ч. II. — Новосибирск, 1982, с. 230—239. — *Прим. перев*

нием информатики. Это значит, что мы не начинаем с современной алгебры и теории неподвижных точек. Однако если по дидактическим, методическим или теоретическим соображениям какой-нибудь преподаватель захочет начать даже не с рекурсии общего вида, а всего лишь с команд повторения (и перехода), т. е. с элементов так называемого «процедурного стиля», то, с незначительными видоизменениями в аргументации, он может произвести соответствующую перекройку материала и от раздела 2.3.1 сразу перейти к главе 3, а в случае когда новая трактовка базируется на паскале, даже к разделу 3.2. Пропущенную часть второй главы можно тогда разобрать после разделов 3.4 и 3.5. Разделы 3.6 и 3.7 касаются вопросов, зависящих от машинной реализации, и их изучение тоже можно отложить на более позднее время.

В приложениях даётся дополнительный материал, не лежащий в основном русле изложения, в том числе и введение в шенноновскую теорию информации, которое раньше, будучи помещено в первой главе, замедляло целенаправленное движение к понятию алгоритма.

Здесь уместно также сделать одно предупреждение: мы не придерживаемся по-рабски ортодоксальных нотаций. Например, для паскаля мы считаем так называемые предописания (forward-конструкции) для перекрёстных систем промашкой, обусловленной техникой построения трансляторов. Мы не заботимся также о том, что употреблять идентификатор *mod* в качестве обозначения функции нельзя, ибо слово **mod** уже резервировано. Почему мы должны быть более скрупулёзными, чем сам Вирт в [35], где он обозначает через & отсутствующую в паскале конкатенацию? Правда, ван Вейнгаарден как создатель алгола-68 не оставил нам в этом плане никаких лазеек.

И ещё одно предупреждение: иногда текст содержит весьма сжатые формулировки, которые должны побуждать к основательному размышлению.

Новый текст книги также возник из вводного курса лекций, который, кстати, теперь не ограничивается двумя семестрами и читается уже не на отделении математики, а на факультете математики и информатики Мюнхенского технического университета.

Что касается благодарностей, то прежде всего я должен назвать моего покойного друга Клауса Замельзона, беседы с которым помогли мне глубже понять многое.

Далее, мне хочется выразить признательность моим коллегам И. Айкелю, В. Хану, М. Паулю, А. Яммелю и Р. Байеру за полезные конкретные советы, и многим моим коллегам и ученикам, которых я не имею здесь возможности назвать поимённо, за плодотворную критику. Особенно помогли мне в преодоле-

нии многочисленных трудностей, с которыми связан мучительный процесс появления книги на свет,— включая перепроверку примеров программ — мои сотрудники из рабочей группы CIP; среди них мне хотелось бы назвать в первую очередь заместителя директора полной средней школы г-на В. Доша, который позаботился ещё о подготовке конспекта указанных выше лекций и вычитал корректуру книги с достойными уважения аккуратностью и терпением. Наконец, моя особая благодарность — г-же д-ру Х. Бауэр-Фогг за то понимание, с которым она относилась к своему постоянно занятому мужу.

Мюнхен, Пасха 1982 г. *Ф. Л. Бауэр*

Предисловие к третьему изданию части 2

Переработанный текст второй части «Информатики» сохраняет то же деление на главы, что и во втором издании, однако больше подчёркнута главная идея — *структуры,* дополняющая основную тему первой части — *алгоритмы.*

Пятая глава была ужата и переориентирована в сторону структурных характеристик программ, прежде всего блочной структуры. Шестая глава начинается исследованием структуры (внешней) памяти и подводит к изучению структур данных общего вида. В седьмой главе рассматриваются формальные системы; эта глава строится на фундаменте теории отношений. Завершает вторую часть восьмая глава, посвящённая синтаксису и семантике алгоритмических языков. Она написана с учётом современного состояния теории и уточняет многие неформальные пояснения из первой части.

Как и в части 1, я устоял перед искушением поставить абстрактные типы во главу угла и строить алгоритмы исходя из понятия структуры терма. В соответствии с этим абстрактные типы рассматриваются лишь в заключительном разделе, причём ради простоты мы ограничиваемся, по существу, всюду определёнными операциями. Как и раньше, книга остаётся вводным курсом, к которому должны примыкать специальные курсы и который должен облегчать переход к изучению монографий. Ввиду неодинаковой подготовки приступающих к изучению курса информатики нельзя было отказаться от некоторого минимума пропедевтики.

Впрочем, лектор может поменять местами относительно независимые главы с пятой по седьмую; особенно легко сделать это с пятой и шестой главами.

Как из первой части был вытеснен короткий раздел о строении вычислительной машины и её системе команд, точно так же из второй выпал раздел об операционных системах. Эти темы сегодня лучше изучать в курсе системного программирования как его фундамент[1] (см. G. Seegmüller, Einführung in die Systemprogrammierung [162]).

Вместе с гипотетической третьей частью, главная тема которой — *машины*, настоящая книга покрывает содержание широко распространённого сегодня четырёхсеместрового вводного курса по информатике и предоставляет теоретический базис для занятий программистской практикой, столь важной уже на самых первых порах.

Приложение об истории информатики оставлено в основном без изменений. Неизменной остаётся и справедливость сказанного в предисловии к первому изданию, а именно — для того, чтобы достичь полного понимания даже такой молодой науки, как информатика, её нужно рассмотреть в историческом разрезе.

В конце книги помещены синтаксические диаграммы для использованных нами версий алгола-68 и паскаля. В их основе лежит язык CIP-L, для которого А. Лаут, Т. А. Матцнер и Р. Обермайер в рамках проекта CIP разработали базирующийся на паскале компилятор.

За предложения и критические замечания я вновь должен поблагодарить друзей и коллег, уже упоминавшихся в предисловии к третьему изданию первой части, а также Г. Гооза, П. Дойсена, В. Брауэра, М. Броя, М. Вирзинга, Г. Шмидта, Р. Герольда, Ф. Пайшля, Х. Кусса, Т. Штрёляйна и сотрудников рабочей группы CIP — Р. Бергхаммера, К. Дельгадо-Клооз, Ф. Эрхарда, Р. Гнаца, У. Хилл-Замельзон, А. Хорша, А. Лаута, Т. А. Матцнера, В. Майкснера, Б. Мёллера, Х. Парча, П. Пеппера, Р. Обермайера, Р. Штайнбрюггена и особенно Х. Вёснера. Г-ну заместителю директора полной средней школы В. Дошу я снова хочу выразить признательность за неоценимую помощь в подготовке рукописи и правке корректур.

Помимо Хайнца Рутисхаузера этот том посвящён моему незабвенному другу Клаусу Замельзону (21.12.18 — 25.5.80). Замельзон был одним из первых, кто наполнил содержанием понятие алгоритмического языка и тем самым способствовал выделению информатики в самостоятельную дисциплину.

Мюнхен, Пасха 1984 г. *Ф. Л. Бауэр*

---

[1] Пожалуй, только программирование параллельных процессов не удастся втиснуть в узкие рамки машинной ориентации. Нужно надеяться, что бурное развитие исследований в области архитектуры ЭВМ прояснит дело в этом отношении.

Посвящается памяти

Альвина Вальтера
(1898—1967)

Ганса Пилоти
(1894—1969)

Роберта Зауэра
(1898—1970)

Вступление

L'INFORMATIQUE:
Science de traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications, dans les domaines techniques, économiques et socials

(Académie Française) [1]

Прошло неполных 50 лет с тех пор, как началось развитие современных *компьютеров* — в Германии оно было начато работами Конрада Цузе в 1934 г. Успехи компьютерной техники стали внушительны уже в пятидесятых годах, когда появились первые вычислительные машины, изготовленные для продажи. Постоянное увеличение скорости работы и объёма памяти машин сочеталось с сокращением расходов на их приобретение и эксплуатацию. Кто бы мог поверить в 1956 г., когда по финансовым соображениям в ФРГ для научных целей был приобретён и введён в эксплуатацию всего лишь один экземпляр самой большой в то время (американской) установки — для чего потребовалось сооружение целого вычислительного центра,— что для той же самой вычислительной мощности сегодня нужна лишь сумма порядка десяти тысяч марок, а само устройство разместится на столе в рабочем кабинете.

Прорыв произошел благодаря микроминиатюризации, которая требует крупных капиталовложений, но после их амортизации позволяет изготавливать интегральные микросхемы («чипы»), стóящие не больше хорошей лампочки накаливания. Причина, конечно же, в огромном количестве сбываемых экземпляров, которое в свою очередь обусловлено низкими розничными ценами.

В отличие от ламп накаливания компьютеры можно, более того — нужно, программировать. Это ведёт к появлению новых рабочих мест. В нашем мире зреют необычайные перемены, масштаб которых не вполне ещё осознан с политической и экономической точек зрения. Достаточно сказать, что в промышленно развитых странах уже сегодня более 1 % населения кормится прямо или косвенно благодаря компьютеру.

---

[1] Информатика: Наука об осуществляемой преимущественно с помощью автоматических средств целесообразной обработке информации, рассматриваемой как представление знаний и сообщений в технических, экономических и социальных областях (Французская Академия) (*франц.*). Перевод А. Ф. Рара. — *Прим. перев.*

Компьютер стал символом положения дел. Общественное мнение связывает с ним такие понятия, как „современный", „прогрессивный", „необходимый". Это отражается и в том, что так называемый шрифт E13B — „магнитно-читаемый" шрифт для устройств считывания с документов — после соответствующей графической стилизации используется в рекламе (см. рис. 1).

С компьютером многие связывают, однако, и такие представления, как „непонятный", „опасный", „вредный". Хотя неприкосновенность личной жизни с давних пор охраняется законом, сейчас, когда с ничтожными затратами можно записать и хранить в течение месяца в машинной памяти информацию о том, кто, когда, с кем говорил по телефону, страх закрадывается в сердце даже тем, кому нечего скрывать. Не получится ли так, что так называемый прогресс в очередной раз подомнёт под себя человека и сделает его лишь более несчастным?

Читателю, в особенности молодому студенту, стоит задуматься над тем, насколько важно в этой ситуации иметь представление о возможностях современных компьютеров и пределах этих возможностей.

CANADIAN
COMPUTER
CONFERENCE

SESSION'72

Queen Elizabeth Hotel,
Montreal,
June 1-2-3, 1972

Рис. 1. „Компьютерный шрифт" в рекламе, 1972 и 1982 гг.

Глава 1

Информация и сообщение

Мы начнем с обсуждения центральных понятий «сообщение» и «информация», а затем, рассмотрев примеры передачи сообщений между людьми, изучим некоторые из основных понятий, важных с точки зрения обработки дискретных сообщений, прежде всего понятие алгоритма.

1.1. Сообщение и информация

„Сообщения — это символы для информации, смысл которых нужно выучить."

Мелис[1]

«Сообщение» и «информация» — это основные понятия информатики, техническое значение которых не вполне соответствует употреблению этих двух слов в обиходной речи. Необходимое в связи с этим уточнение содержания указанных понятий не может быть достигнуто с помощью определения, так как последнее лишь сводило бы эти понятия к другим не определённым основным понятиям. Поэтому мы вводим ***сообщение*** и ***информацию*** как неопределяемые основные понятия и разъясняем их использование на ряде примеров. В дальнейшем нам представится возможность проверить правильность получаемых при этом представлений.

При разграничении понятий сообщения и информации мы исходим из распространённых оборотов речи типа

«это сообщение не даёт мне никакой информации»,

что приводит к следующему отношению между этими понятиями:

(абстрактная) информация передаётся

посредством (конкретного) сообщения.

---

[1] Arthur Mehlis (род. 1903), немецкий специалист по технике связи. — *Прим перев.*

Соответствие между сообщением и информацией не является взаимно-однозначным. Для одной и той же информации могут существовать различные передающие её сообщения, например сообщения на разных языках или сообщения, которые получаются добавлением *неважного* сообщения, не несущего никакой дополнительной информации. Сообщения, передающие одну и ту же информацию, образуют класс эквивалентных сообщений. Обратно, одно и то же сообщение может передавать совершенно различную информацию: сообщение о падении самолета для близких родственников погибшего имеет совсем иной смысл, нежели для авиакомпании; разные читатели из одной и той же газетной статьи черпают совершенно различную информацию, соответствующую кругу их интересов.

Таким образом, одно и то же сообщение, по-разному *интерпретированное*, может передавать разную информацию. Абстрагируя, мы можем сказать, что решающим для связи между сообщением N и информацией I является некое отображение α, представляющее собой результат договорённости между отправителем и получателем сообщения или предписанное им обоим и называемое *правилом интерпретации*. Символически мы будем записывать правило интерпретации в следующей форме:

$$N \overset{\alpha}{\longmapsto} I.$$

Правило интерпретации α для данного сообщения часто получается как частный случай некоторого общего правила, применимого к целому множеству $\mathfrak{N}$ сообщений, которые построены по одинаковым законам. Если мы формулируем сообщения на некотором *языке* (см. 1.1.1), то высказывание

„X понимает язык $\mathfrak{N}$“

выражает тот факт, что лицо X знает правило интерпретации α для всех (или по крайней мере для большинства) сообщений, формулируемых на данном языке.

Иногда правило интерпретации известно лишь ограниченному кругу лиц; сюда относятся правила интерпретации для специальных языков, в частности для различных профессиональных и научных языков (жаргонов). Сленг, арго, блатной жаргон служат для формирования и отгораживания определённой социальной группы посредством преднамеренного уменьшения понятности используемого языка; они возникли из подлинных тайных языков уголовного мира.

Связь между сообщением и информацией особенно отчётливо видна в криптографии: здесь никто посторонний не должен суметь извлечь информацию из передаваемого сообщения, иначе это означало бы, что он располагает „ключом“.

Часто встречаются и такие сообщения, которые могут интерпретироваться по-разному, причём различные интерпретации основываются одна на другой. Так, сообщение „идёт дождь" может нести дополнительную информацию »нужно взять с собой зонтик«. В этом случае говорят об информациях различной ***степени отвлечённости***.

Другие возможные связи между сообщением и информацией иллюстрируются с помощью примеров табл. 1.

Таблица 1

Языковые сообщения

a) „до завтра"/„see you tomorrow"
b) „Ла2 — с2"/„дезоксирибонуклеиновая кислота"
c) „видел морских львов"
d) „Momepr"/„JDOOLD HVW RPQLV GLYLVD
e) „ьлерпа"
f) „ТЕТЯ АНЯ ГРОБУ ТЧК БУДЕМ УСТРАИВАТЬ 13 НОЯБРЯ ДНЕМ ТЧК ЕВГЕНИЯ САМОЙЛОВА"
g) „Поздно, но приду"/„Поздно, не приду"
h) „Ротшильд обращался с ним очень фамильонно"/„Матч века машин"
i) die gefangenen fliegen

В примере а) приведены русское[1] и английское сообщения, каждое из которых, в обычном их понимании, передаёт одну и ту же информацию. В примере b) мы имеем дело не с сообщениями на тайных языках, а с сообщениями на специальных языках шахмат и химии соответственно. В с) речь может идти о некотором заранее условленном сообщении для выполнения определенного действия, т. е. о ***секретном сообщении, замаскированном*** под открытое. В случае d) мы имеем зашифрованные сообщения: коммерческий шифр для указания цены в варианте, который в течение многих лет использовался[2] для обозначения даты упаковки масла (ключевое слово MILCHPROBE), и метод шифровки, который применялся ещё Юлием Цезарем, когда вместо нужной буквы пишется третья следующая за ней в алфавите[3]. Если в обоих примерах d) речь идёт о кодировании с помощью ***подстановки***, то в е) мы имеем простейший случай кодирования посредством ***обращения***[4]. Читая в обратном порядке, справа налево, получим закодированное в е) сообщение „апрель". В f) дело может идти об открытом сообщении, в котором ***скрыто секретное сообщение***: читая первые буквы по кругу,

[1] В оригинале — немецкое. — *Прим. перев.*

[2] В Германии. — *Прим. перев*

[3] Левая часть даёт (после расшифровки по ключу MILCHPROBE) цену 181067, а правая часть — изречение Юлия Цезаря „Gallia est omnis divisa" („Галлия вся разделена"). — *Прим. перев.*

[4] Которое часто используется и в полифонической музыке (Krebskanon).

начиная с четвёртой, получим слово БУНДЕСТАГ. Пример g) показывает, как незначительное изменение текста может существенно изменить передаваемую информацию. В h) мы имеем крайний случай, когда сообщение является само по себе непонятным, в первом примере из-за орфографически неправильного слова „фамильонно“ (принадлежащего Генриху Гейне), а во втором — из-за двусмысленности; игра слов и многие остроты обязаны своим существованием кажущейся бессмысленности. Второй из этих примеров становится при устной передаче однозначным благодаря ударению и фразировке. Наконец, пример i) выявляет недостатки написания всего текста строчными буквами[1].

Сообщения имеют и коммерческую ценность — существуют информационные агентства (Agence Havas (1832 г.), Reuter (1851 г.)). Службы информации занимаются сбором информации (бывает, и незаконным способом), а иногда распространяют дезинформацию.

1.1.1. Языковые сообщения

"If there is one thing on which all linguists are fully agreed, it is that the problem of the origin of human speech is still unsolved." [2]

Марио Пей

Для сообщений, которыми обмениваются люди, в большинстве случаев имеются соглашения относительно их формы. О таких сообщениях мы говорим, что они передаются в ***языковой форме***, что они составлены на некотором ***языке***. При этом слово „язык“ используется в существенно более широком смысле, чем в случае связанного с ним понятия „говорить“[3]. Мы знаем разговорный и письменный языки, язык глухонемых, построенный на жестах и мимике (рис. 2), печать для слепых, воспринимаемую осязанием (рис. 22). Два последних примера показывают, что высокоразвитое языковое общение не ограничивается устной и письменной речью.

Хотя многое говорит в пользу того, что именно разговорный язык знаменует начало истории человека, всё же и в современ-

---

[1] В немецком языке существительные пишутся с прописных букв. В данном случае допустимы два толкования сообщения i): „пойманные мухи“ („die gefangenen Fliegen“) и „пленные летают“ („die Gefangenen fliegen). — *Прим. перев.*

[2] „Если в чём-то и сходятся мнения всех лингвистов, так это в том, что проблема происхождения человеческой речи всё ещё не решена“ (*англ.*). — *Прим. перев.*

[3] В немецком языке эти понятия связаны особенно тесно — они одного корня: говорить — sprechen, язык — Sprache. — *Прим. перев.*

Рис. 2. Некоторые знаки языка глухонемых

ном обществе остаётся язык жестов, дополненный специфическими звуками, такими как шипение, мычание, свист и щелчки,— хоть и примитивное, но иногда решающее вспомогательное средство, обеспечивающее взаимопонимание.

Примеры в конце предыдущего раздела даны печатным (типографским) шрифтом; в рукописи они были сначала написаны от руки, на лекции они могут быть сформулированы устно. Разумеется, их можно выразить и на языке глухонемых или языке слепых (печать для слепых); наконец, глухой может понять произносимое слово по движению губ. Когда мы говорим о ***языковых сообщениях***, мы имеем в виду то общее, что присуще каждому из этих случаев; способ передачи — письменно, устно, посредством осязания или ещё как-то — не имеет здесь никакого значения. При этом, однако, следует иметь в виду, что, например, информация, содержащаяся в устном сообщении, не всегда полностью воспроизводится соответствующим письменным сообщением. Такие настроения, как гнев, радость, горечь, искренность, находят своё полное выражение только в устной речи. Выше уже отмечалось, что ударения и паузы также несут информацию. Что информацию такого рода бывает невозможно или непросто восстановить, исходя из контекста, показывают некоторые мучительные оговорки дикторов.

В немецком языке[1] слово язык (Sprache) используется не только в смысле французского langage[2], но и в смысле французского langue[3]. В этом последнем случае, т. е. когда мы говорим о немецком языке, английском языке и т. д., лучше было бы использовать термин ***язык-речь***[4]. Для различия между языком и языком-речью характерно, что внутри данного языка-

---

[1] И в русском тоже. — *Прим. перев.*

[2] Язык как речь, стиль (*langage imagé* — образный язык, *langage articulé* — членораздельная речь, *langage fleuri* — цветистый язык); крик животных, пение птиц; также: *langage de fleurs* — язык цветов, *langage de signes* — язык знаков. — *Прим. ред.*

[3] Язык как лингвистическая система (*langue russe* — русский язык, *langue vivant* — живой язык, *langue étrangère* — иностранный язык). — *Прим. ред*

[4] В оригинале *Zunge* (близко по значению к французскому *langue*, в частности в отличие от *Sprache*, имеет, как и *langue*, значение „язык" в анатомическом смысле; на русский в большинстве случаев переводится словом „язык"). — *Прим. ред.*

речи можно говорить на высоком языке, на разговорном языке, на блатном языке (сленге) или на профессиональном языке (жаргоне). Существуют также языки, которые не принадлежат и вообще не могут быть причислены ни к какому языку-речи, например искусственные языки вроде эсперанто (Заменхоф, 1887 г.) или некоторые специальные языки, в том числе язык формул математики и языки программирования.

Наконец, понятие языка не ограничивается случаем общения между людьми, оно используется и в случае сравнительно высоко развитых форм общения между другими живыми существами. Примером может служить открытый К. Фришем язык ориентации пчел.

1.1.2. Письмо

'Verba volent, scripta manent'[1]

Для нас особенно важны языки, в которых для передачи сообщений используются ***долговременные носители информации***.

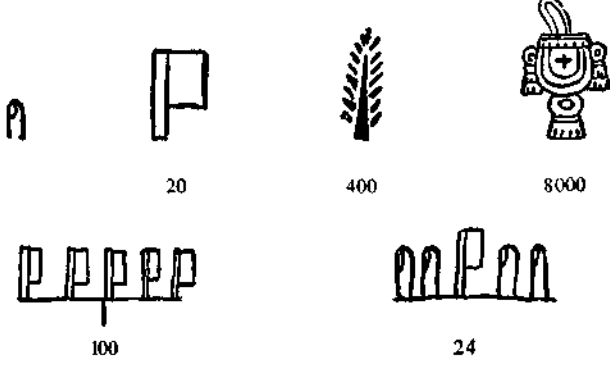


Рис. 3. Числовые знаки ацтеков.

При этом передача освобождается от гнёта реального времени, и становятся даже возможными сообщения человека самому себе — ***заметки*** на память — и тем самым уменьшение нагрузки на человеческую память за счёт использования „инструмента".

Представление сообщений на долговременных носителях мы называем ***письмом***[2], а сам долговременный носитель — ***носителем письма***.

[1] *Слова улетают, написанное остаётся (лат.). — Прим. перев.*

[2] Истоки возникновения письма (а оно впервые появляется лишь в высокоразвитых культурах), ра́вно как и истоки возникновения языка, покрыты мраком неизвестности; по-видимому, важную роль сыграло иероглифическое письмо (см., например, числовые знаки на рис. 3).

Прежде всего следует упомянуть ***зрительно воспринимаемое письмо***, которое создается вручную (рукопись) или механически (машинопись, типографская печать). ***Письмо, воспринимаемое на слух,*** долгое время было тщетной мечтой изобретателей [1], пока Эдисон не изобрел фонограф. ***Письмом, воспринимаемым осязанием***, является письмо слепых, которое пишется вручную (посредством наколов иголкой), а также механически. Фиксация изображений (например, в кино) также представляет собой письмо.

1.2. Органы чувств

В предыдущих разделах были упомянуты органы чувств, которые могут служить для передачи языковых сообщений. Эти передающие и воспринимающие органы чувств человека и высших животных перечислены в табл. 2. Некоторые из воспринимающих органов чувств служат также и для односторонней неязыковой связи с окружающим миром. У высших живых существ только слуховые, зрительные и тактильные восприятия достататочно дифференцированы, чтобы естественным образом служить для передачи языковых сообщений. Наряду с языками непосредственного общения: разговорной речью, призывающими и предостерегающими звуками (слуховое восприятие), языком глухонемых (зрительное восприятие) и воспринимаемым рукой языком слепых (тактильное восприятие) выступают языки, в которых используются инструменты: барабанный бой, стук, свистки, звуки рожка, трубы, сирена — сигнал тревоги (слуховое восприятие), световые сигналы и сигналы флажками (зрительное восприятие).

Определенные знания о свойствах и работе органов чувств человека оказываются существенными, когда мы хотим рационально включить человека в цепочку обработки и передачи информации (в её начало или конец). (Некоторые из количественных аспектов интересующих нас в этой связи вопросов обсуждаются в приложении В.)

1.2.1. Работа органов чувств, проведение возбуждения

Функциональная способность органов чувств лежит в определенных пределах. Здесь прежде всего следует назвать ***время реакции*** (латентное, или скрытое, время). Для акустических (звуковой импульс) и оптических (загорание лампочки) сигна-

---

[1] Замёрзшие звуки почтового рожка Мюнхаузена решают эту проблему неудовлетворительно.

Таблица 2

Передающие и воспринимающие органы человека и высших животных

Передающий орган (*эффектор*)	Физический носитель сообщений	Воспринимающий орган (*рецептор*)	Способ передачи
голосовой аппарат гортани (дар речи)	звуковые волны (от 16 до 16000 Гц)	слух (улитка уха, волосковые клетки базилярной мембраны)	слуховой
мышцы лица и другие (мимика и жестикуляция)	световые волны (около 10^{15} Гц)	зрение (сетчатая оболочка глаза, палочки и колбочки)	оптический (зрительный)
мышцы рук и кистей (навыки)	давление	осязание (рецепторы прикосновения и давления)	тактильный
—	температура (от −20 до +50 °С)	терморецепторы	
—	концентрация молекул в жидком растворе	вкус (язык, вкусовые почки слизистой оболочки рта)	
запаховые железы	концентрация молекул газа	обоняние (полость носа, обонятельные рецепторы слизистой оболочки носа)	
—	ускорение	чувство равновесия (вестибулярный аппарат)	
—	механические и другие повреждения тканей	болевая чувствительность (свободные нервные окончания)	

лов оно составляет для человека 140—250 мс до ответа, состоящего в том, что испытуемый нажимает кнопку. Для более сложных заданий время реакции заметно увеличивается (прочитать указанное слово: 350—550 мс, назвать указанный предмет домашнего обихода: 600—800 мс). Это уже говорит о том, что процесс восприятия — не только функция рецепторов. Сюда примыкают проведение раздражения по нервным путям, переработка его в мозге, а также проведение ответа к эффектору. При этом на глаз как на воспринимающий орган приходится около 40 мс, а на срабатывание мышц руки как передающего органа — около 50 мс.

Скорость проведения возбуждения по нервным путям составляет для нерва ноги виноградной улитки 0.4 м/с, для седалищного нерва лягушки при 18 °С — 28 м/с, для двигательных нервных волокон человека — 120 м/с.

При этом по нервным путям пробегают импульсы электрохимической природы с максимальной амплитудой 80 мВ и длительностью порядка 1 мс. Интенсивность раздражения определяет частоту таких импульсов, а именно: частота импульсов

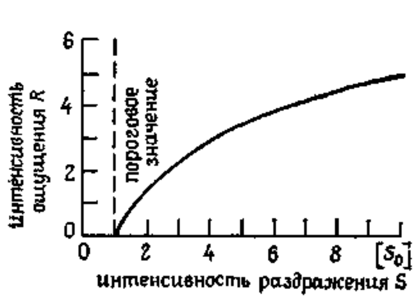


Рис. 4. Закон Фехнера.

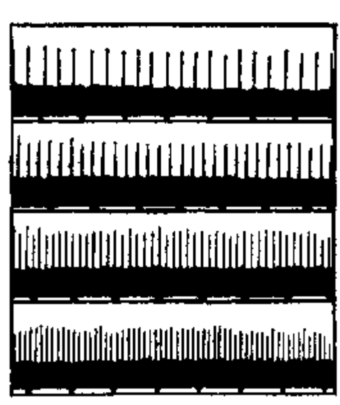


Рис. 5. Потенциалы действия нервного волокна при различной интенсивности раздражения.

в общем случае пропорциональна логарифму интенсивности раздражения. Этот результат согласуется с постулированным в 1850 г. Г. Т. Фехнером и подтверждённым психологическими экспериментами законом, утверждающим, что интенсивность ощущения пропорциональна логарифму интенсивности раздражения (рис. 4):

$$R = C \lg(S/S_0).$$

В терминах электротехники речь здесь идет об импульсно-частотной модуляции (см. 1.3.2). Такой способ проведения возбуждения (рис. 5) объясняет, кстати, почему при низкой интенсивности раздражения, а тем самым при низкой частоте импульсов увеличивается время реакции. Для того чтобы орган вообще смог что-либо воспринять, интенсивность раздражения должна превосходить определённое ***пороговое значение*** S_0. Для слуха пороговое значение составляет около $2 \cdot 10^{-7}$ мбар [1].

---

[1] Чтобы вызвать слуховое ощущение, достаточно энергии в 10^{-12} эрг, для появления зрительного ощущения — 10^{-10} эрг.

Кроме того, при длительности импульса в 1 мс теоретически возможна передача импульсов с частотой не более 1000 Гц, а практически — не более 250 Гц, откуда следует верхняя граница для интенсивности раздражения. Болевой порог для слуха лежит где-то около $2 \cdot 10^{-1}$ мбар.

Наряду с законом Фехнера сюда относится закон Вебера, который гласит, что разрешающая способность, т. е. способность воспринимать раздельно два различных раздражения, пропорциональна интенсивности раздражения. Э. Х. Вебер сформулировал его впервые в 1834 г. для чувства осязания.

Если через S обозначить интенсивность раздражения, а через δS — разрешающую способность, т. е. минимальное изменение раздражения, которое приводит к еще улавливаемому различию ощущений, то закон Вебера утверждает, что

$$\delta S = kS, \quad \text{или} \quad \delta(\ln S) = k.$$

Таким образом, в логарифмической шкале, которую в соответствии с законом Фехнера нужно взять для интенсивности раздражения, разрешающая способность оказывается постоянной, как только раздражение превзойдёт пороговое значение.

Значения безразмерной величины k распределены в широкой области и зависят от рассматриваемого ощущения и индивидуальности испытуемого. Ниже приведены минимальные значения для особенно „тонко чувствующих" испытуемых:

яркость света	$k \approx 0.015$,	вес	$k \approx 0.019$,
длина отрезков	$k \approx 0.025$,	вкус	$k \approx 0.25$,
громкость звука	$k \approx 0.03$,	запах	$k \approx 0.35$.
высота звука	$k \approx 0.003$,		

Воспринимаемая интенсивность раздражения от порогового значения до границы болевого ощущения для большинства видов раздражений лежит в весьма широких пределах, для описания которых приходится привлекать 10 в большой степени, например:

для яркости	$1:10^{14}$,
для громкости звука	$1:10^{6}$,
для высоты звука	$1:10^{3}$.

Так как для двух максимально близких ещё отличимых друг от друга интенсивностей раздражения S и S', согласно закону Вебера, справедливо соотношение

$$S = (1 + k) S',$$

то отсюда вытекают следующие верхние оценки для общего количества различимых между собой интенсивностей

раздражения:

$$14/\lg 1.015 = 14/0.0064 \approx 2^{11} \text{ степеней яркости,}$$
$$6/\lg 1.03 = 6/0.013 \approx 2^{9} \text{ степеней громкости,}$$
$$3/\lg 1.003 = 3/0.0013 \approx 2^{11} \text{ высот звука.}$$

Совершенно иная картина наблюдается, когда речь идет о числе интенсивностей раздражения, которые способен одновременно уловить испытуемый. Если говорить о слухе — то всего лишь от 5 до 7 различных высот звука.

1.2.2. Обработка раздражения в мозге

Функции эффекторов и рецепторов более углублённо изучаются психологией органов чувств, а проведение возбуждения по нервным путям — нейрофизиологией и нейроанатомией.

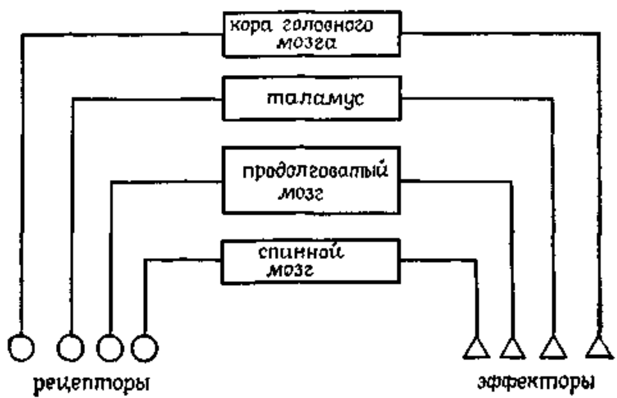


Рис. 6. Уровни ответа на раздражение.

Остаётся вопрос, где и каким образом происходит собственно обработка сенсорных сигналов и где возникает возможный ответ на них.

На рис. 6 показано, что это может происходить на четырёх различных уровнях нервной системы: в качестве ***рефлекторного***[1] ***центра*** могут выступать спинной мозг, продолговатый мозг (medulla oblongata), часть мозжечка, называемая таламусом, и поверхностный слой головного мозга, называемый корой (cortex). Удар по коленной чашечке (точнее, по коленному

[1] Своим происхождением слово „рефлекс" обязано представлению Декарта о том, что раздражение, исходящее от рецептора, как бы посредством зеркала „рефлектируется" (отражается) к эффектору.

сухожилию) вызывает непроизвольное подскакивание голени. При этом время реакции составляет около 30 мс. Центр „коленного рефлекса" находится в спинном мозге. Значительно большее время реакции (порядка 200—400 мс) имеют условные рефлексы, которые вырабатываются сначала посредством приучения, а затем становятся непроизвольными. В этих случаях

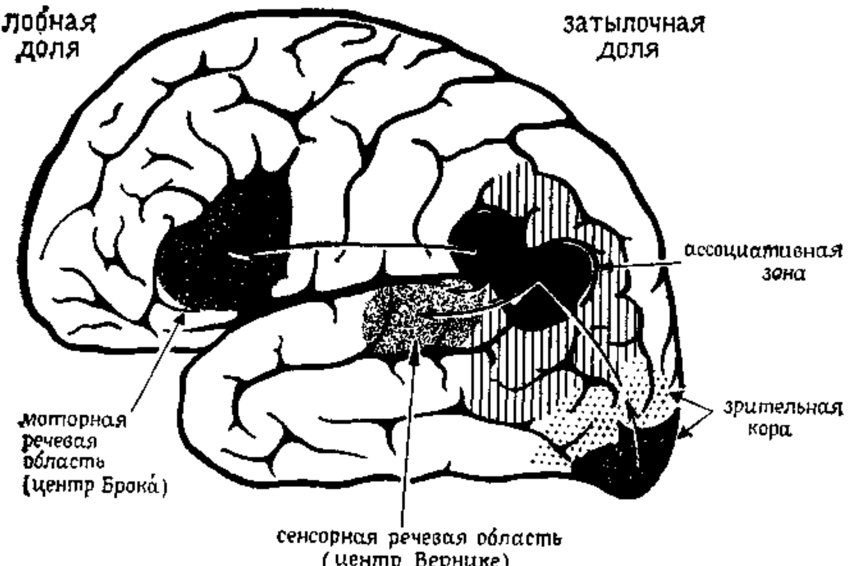


Рис. 7. Доминирующая (левая) половина головного мозга человека (по О.-И. Грюссеру)

рефлекторный центр расположен на более высоком уровне, однако всё же не в головном мозге. О продолговатом мозге известно, что он является центром вегетативных функций и наряду с другими рефлекторными центрами содержит дыхательный центр.

Для наиболее важных рецепторов исходные и конечные точки передачи раздражения расположены в коре головного мозга. Проблема локализации (рис. 7), которой занимаются начиная с Альберта Магнуса, в настоящее время частично решена. О том, каким образом происходит переработка раздражения в головном мозге, до сих пор точно ничего не известно. Однако то, что такая переработка происходит и что речь идёт об очень сложных процессах, в которых частично участвует и память, подтверждено экспериментально. В частности, можно утверждать, что между раздражением и ощущением нет однозначной и неизменной связи.

Для того чтобы пояснить последнее утверждение, рассмотрим сферу зрительного восприятия. Прежде всего следует отметить, что ощущение света и цвета возникает не только при действии на глаз света подходящей длины волны, но и в результате механического или химического воздействия. Образ, отвечающий ощущению, определяется не самим раздражением сетчатки глаза, а соответствующим центром в головном мозге

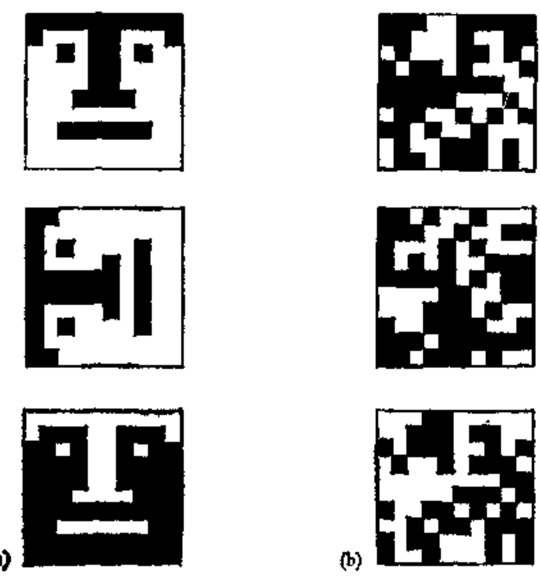


Рис. 8. Феномен образа: (a) положительный, (b) отрицательный.

(в таламусе или коре). Это согласуется с тем фактом, что состояние полной слепоты вызывается не только повреждением глаз, сетчатки или нервных путей, но и выпадением функций определённого участка коры головного мозга. Повреждения в двух других соседних участках коры приводят к нарушению обработки зрительных ощущений, придающей им смысл (зрительная агнозия — расстройство зрительного восприятия).

Дальнейшее представление о рассматриваемом предмете даёт явление ***аккомодации***. Несомненно, что глаз передает на сетчатку оптическое изображение, причём настройка на расстояние до объекта наблюдения достигается изменением фокусного расстояния хрусталика. Если к наблюдателю приближается человек, то его изображение на сетчатке глаза наблюдателя будет увеличиваться Несмотря на это, смотрящему не кажется, что размеры приближающегося человека стали больше. Гельмгольц назвал это явление (1866) „бессознательным заключе-

нием", коррекцией, предпринимаемой центром по обработке зрительных впечатлений в коре головного мозга автоматически. Так что она выполняется без всяких размышлений, непроизвольно. Экспериментально установлено, что измерение расстояния, необходимое для коррекции, осуществляется статически за счёт искривления хрусталика, смещения оптических осей глаз друг относительно друга и стереоскопического эффекта, а также динамически за счёт параллакса при движениях головы.

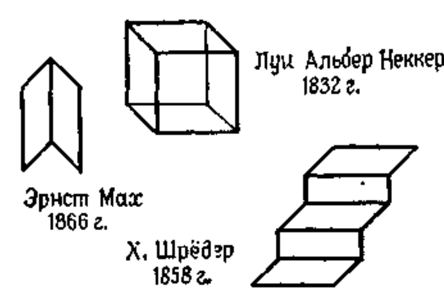


Рис. 9. Книга, куб и лестница.

Далее следует упомянуть феномен ***образа*** (В. Кёлер, М. Вертхаймер, около 1920 г.). На достаточно однородном „фоне" зри-

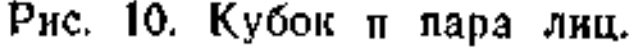


Рис. 10. Кубок и пара лиц.

Рис. 11. Картинка с секретом.

тельным восприятием выделяются некоторые обособленные цельные замкнутые области с определенными очертаниями („фигуры"). На рис. 8а „с первого взгляда" видно, что всё это изображения одной и той же фигуры. Совершенно аналогично

построен и рис. 8b: здесь также мы имеем дело с поворотом изображения и заменой чёрных полей на белые и наоборот. Однако для того чтобы обнаружить это, нужно сравнивать друг с другом отдельные поля, клетка за клеткой. Здесь нельзя распознать какую-либо фигуру, поворот которой можно было бы заметить. За формирование зрительного образа отвечают

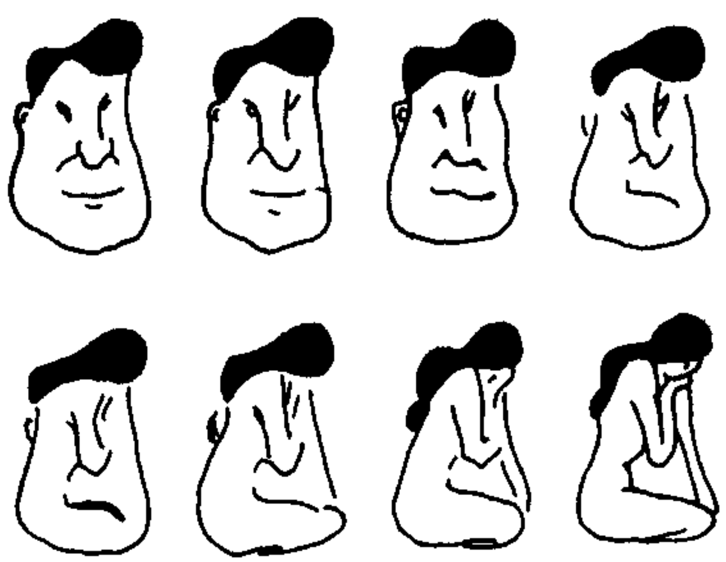


Рис. 12. Гистерезис при распознавании образов.

не сами глаза. Это следует хотя бы из того, что в некоторых изображениях можно распознавать несколько различных фигур. Так например, на рис. 9 книга, куб и лестница стереоскопически двусмысленны; на рис. 10 (по Эдгару Рубину (датский психолог), 1915 г.) мы видим кубок или пару лиц в соответствии с тем, берём мы фон чёрным или белым; на рис. 11 (карикатуриста У. Э. Хилла, 1915 г.) представлена картинка с секретом: молодая и старая женщина. При разглядывании этих картинок часто происходит периодическое „опрокидывание", перестройка структуры зрительного образа.

С этим явлением тесно связан феномен ***гистерезиса***[1]: если последовательно, одну за другой, рассматривать картинки на рис. 12 слева направа и сверху вниз, то сначала мы видим мужское лицо, а потом женскую фигуру; „перескок" происходит обычно на пятом шаге (на шестой картинке). Если же

[1] От греческого *hysteresis* — отставание, запаздывание. — *Прим. перев.*

идти по цепочке картинок в противоположном направлении, то опрокидывание зрительного образа происходит обычно тоже лишь на пятом шаге (на третьей картинке).

Феномен образа проявляется также в случае, когда при обработке изображения в мозге оно дополняется деталями, отсутствующими в рассматриваемой картинке (***дополнение образа***), как, скажем, на рис. 13, где буква ,A' воспринимается

Рис. 13. Дополнение (достраивание) образа (по О.-Й. Грюссеру).

Рис. 14. Марка (фирменный знак) баварского радио (дополнение образа как изобразительный приём).

как пространственная, хотя это ощущение вызывается всего лишь тремя чёрными пятнами, или в фирменном знаке баварского радио (рис. 14) — пример использования феномена дополнения в качестве художественного выразительного средства.

Весьма широко известно явление ***оптических обманов***, или ***иллюзий***. На рис. 15 приведён ряд примеров, среди них иллюзия стрелок Мюллер-Люэра (1889 г.) (левый отрезок кажется более длинным, чем правый), параллелограмм Зандера (диагональ AB кажется длиннее диагонали BC), иллюзия параллельных Херинга (1861 г.) и иллюзия окружностей Эббингхауза (круг, который окружён большими кругами, кажется меньше круга, окружённого малыми).

Менее известен, но очень эффектен феномен ***последействия***. Если некоторое время, зафиксировав взгляд, рассматривать изображенную на рис. 16 фигуру, а затем быстро перевести взгляд на черную точку справа, то кажется, будто видно множество движущихся линий, расходящихся перпендикулярно имеющимся на рисунке. Напрашивается мысль о фильтрации раздражения сетчатки по отношению к направлению очертаний, тем более что ряд других эффектов также говорит о существовании фильтрации. Это, например, известный с древних времён эффект водопада: если долго смотреть на водопад, то кажется, будто видно какое-то движение в противоположном направлении, что также указывает на фильтрацию (в данном случае скорости).

Существуют также цветовые иллюзии, интерес к которым проявляется в изобразительном искусстве — от пуантилизма импрессионистов до поп-арта [1].

О слуховом восприятии можно было бы высказать замечания, аналогичные сделанным выше для зрительного. Вычлене-

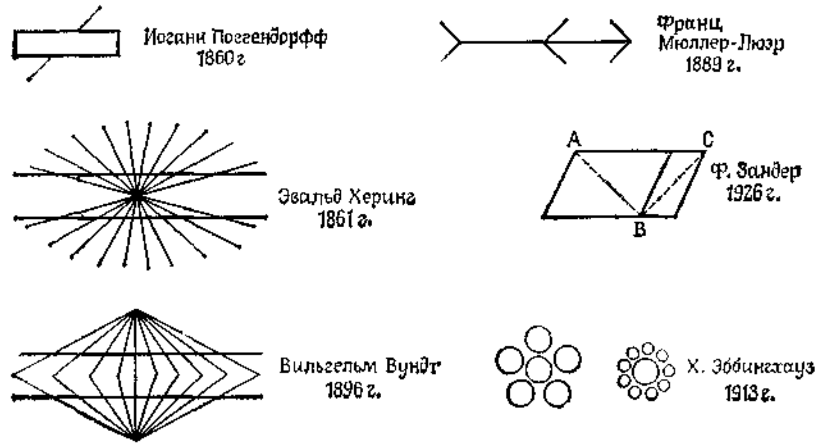


Рис. 15. Оптические иллюзии.

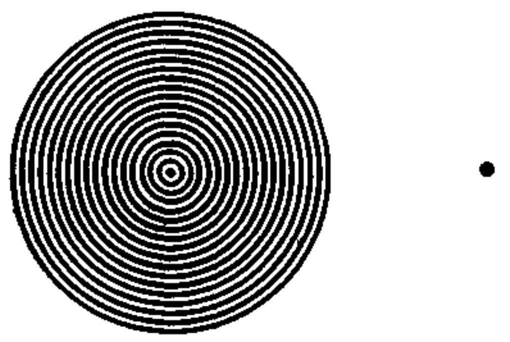


Рис. 16. Эффект последействия.

ние определённого голоса из хаоса звуков или определённой музыкальной линии из полифонического произведения даёт представление об акустическом феномене образа; и здесь наряду со способностью воспринимать образы приходится считаться с существованием иллюзий и последействий.

---

[1] Поп-арт (от английского popular art) — направление в современном зарубежном изобразительном авангардистском искусстве, использующее реальные предметы, изображения, рекламу и т. д. и претендующее на общедоступность и демократизм. — *Прим. перев.*

1.2.3. Значение информационных представлений

Знакомство с физиологией и психологией чувств учит нас многому, и прежде всего тому, что обработка информации (а не только её передача) есть обязательная составная часть нашего чувственного восприятия. Когда мы ниже говорим об искусственной, т. е. изобретённой людьми и выполняемой машинами обработке информации, это должно напоминать нам, что обработка информации не является чем-то принципиально новым. Кроме того, это должно призывать нас к скромности: наше понимание истинного, действительного протекания процессов обработки информации в мозге до сих пор неудовлетворительно, и мы не знаем, каких пределов этого понимания по существу мы можем достичь. Следует отклонить как чересчур поспешные выводы, которые наивно делаются посредством (не-

допустимой) редукции („человека можно объяснить с помощью законов физики“, „умственная деятельность — не что иное, как приём, обработка, накопление и выдача информации“): уже сфера сенсорного восприятия говорит о столь сложных процессах в мозге, что такие утверждения не могут служить даже рабочей гипотезой. Другие стороны „умственной деятельности“, не относящиеся к непосредственному сенсорному восприятию, включая феномен сознания, можно связывать с машинной обработкой информации лишь чисто спекулятивным образом.

Во все времена человек пытался понять себя в свете последних естественнонаучных достижений и при этом в избытке воодушевления впадал в рискованную крайность претендовать на всеобщность своих взглядов. Так было в древности; так случилось с Декартом и Локком; материализм[1] хотел объяснить всё, исходя из понятия энергии. На рубеже этого века, когда появились первые автоматические телефонные коммутаторы, уже им пришлось стать на время козлом отпущения в попытках объяснить функции мозга. Сторонники наивного „информационизма“ считают, что посредством привлечения понятия информации они смогут или должны дать объяснение всего на свете. Это снова может оказаться неверным.

1.3. Устройства связи

Письмо и газета относятся к самым старым и до сих пор не устаревшим примерам (случайной и регулярной) передачи сообщений посредством записи на ***долговременном*** носителе сообщений. В случае передачи информации с помощью ***недолговременного*** носителя сообщений человек также использует различные физические устройства в соответствии с уровнем развития техники на данный момент. Примерами таких устройств связи служат телефон, радио и телевидение, предназначенные как для случайной, так и для регулярной передачи сообщений.

1.3.1. Типы устройств связи

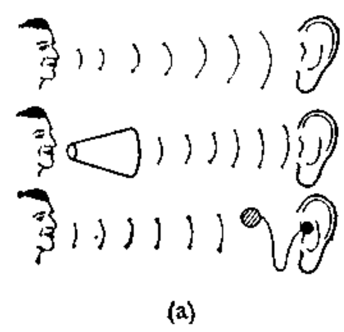
Внешне ***устройство связи***, или, точнее, „приёмо-передающее устройство“, состоит из ***приёмника*** (получателя) и ***передатчика*** (отправителя). О внутреннем строении устройств связи никаких общих утверждений сделать нельзя, разве что при более внимательном рассмотрении многие из них оказываются составленными из нескольких более мелких устройств связи.

Может случиться, что для сообщений на входе и на выходе используется один и тот же носитель. Такие устройства служат

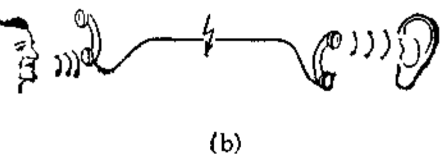
---

[1] Авторы имеют в виду механистический материализм. — *Прим. ред.*

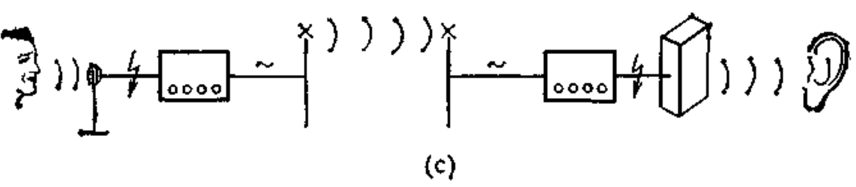
лишь для возможного ***усиления*** или ***регенерации*** сообщения, связанной с устранением помех, и называются ***релейными линиями***[1]. Примерами таких устройств являются рупор, слуховая трубка, а также их современные электронные варианты — мегафон и слуховой аппарат (рис. 17а). Если для сообщений на



(а)



(b)



(c)

Рис. 17. Простые и составные устройства связи: (а) устройства связи для усиления, (b) электрический ток как носитель сообщений, (с) передача с многократным преобразованием.

входе и выходе устройства используются различные физические носители, то устройство связи называют ***преобразователем***.

Если устройство предназначено для связи между людьми, то сообщения на входе или выходе должны быть производимы или соответственно воспринимаемы людьми, т. е. носители

[1] От французского relayer — менять лошадей.

должны соответствовать человеческим эффекторам и рецепторам. В качестве примеров назовём музыкальный инструмент, на котором играют, нажимая на клавиши (физический носитель на входе — давление, на выходе — звуковые волны), и осциллограф, управляемый через микрофон (на входе — звуковые волны, на выходе — световые волны).

Как уже упоминалось, два или более устройств связи можно соединить друг с другом таким образом, что результирующее

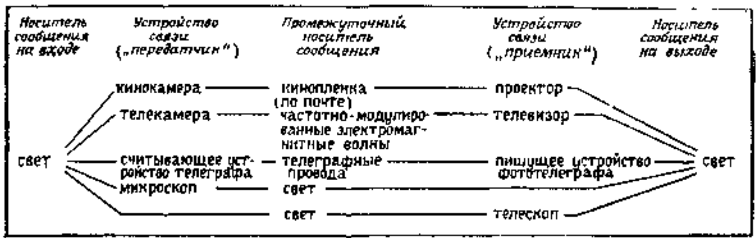


Рис. 18. Передача оптических сообщений.

устройство снова будет устройством связи. Приёмником составного устройства является приёмник первого устройства, участвующего в соединении, а передатчиком — передатчик последнего устройства. В этом случае между передатчиком одного устройства связи и приёмником другого могут использоваться и такие носители, которые недоступны человеческим эффекторам и рецепторам. Примером могут служить телефонная связь по проводам или радио (рис. 17b); на рис. 17c изображено устройство связи для передачи акустических сообщений, составленное из нескольких преобразователей. На рис. 18 приведены составные устройства связи для передачи оптических сообщений.

Исходя из подобных примеров, протяжённую в пространстве среду, „через" которую носитель сообщения передаётся от передатчика к приёмнику, называют ***каналом (связи)***.

Современная техника в качестве носителей при передаче сообщений по каналам связи чаще всего использует

— механическое движение,
— механическое давление жидкостей и газов (гидравлика, пневматика),
— волны давления в жидкостях и газах (до 1 МГц, включая звуковые волны),
— электрические напряжения и токи,
— свободные электромагнитные волны (от 10^2 кГц до 10^6 МГц), в том числе световые волны,
— пучки электромагнитных волн (светосигнальные аппараты, лазеры).

Помимо бумаги, используемой в письме и чтении человеком, в качестве долговременных носителей текстовых сообщений в современной технике чаще всего используются намагничиваемые и светочувствительные плёнки, а также перфорируемая бумага (перфокарты, перфоленты).

Аналогии между рецепторами и эффекторами живых существ и техническими приёмо-передающими устройствами служат предметом исследований в ***кибернетике***. Кибернетика занимается главным образом аспектами, общими для человека и технических устройств с точки зрения передачи и переработки сообщений.

1.3.2. Сигналы и параметры сигналов

Принципиальным является то, что передача сообщений происходит во времени. Поэтому в качестве носителей заслуживают внимания только такие физические величины, которые могут из-

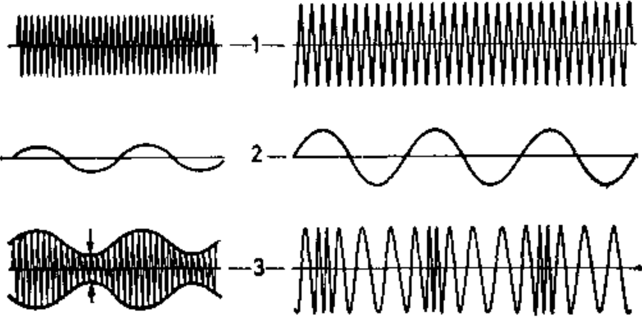


Рис. 19. Амплитудная (слева) и частотная (справа) модуляции: 1) колебания носителя, 2) переносимые колебания, 3) модулированные колебания с амплитудой (слева) и частотой (справа) в качестве параметра сигнала.

меняться во времени. Изменение некоторой физической величины во времени, обеспечивающее передачу сообщения (а тем самым и информации), называется ***сигналом***. При этом для воспроизведения сообщения могут использоваться различные свойства сигнала. Та характеристика сигнала, которая служит для представления сообщения, называется ***параметром сигнала***.

В качестве примера рассмотрим радио. Сигналами здесь являются электромагнитные колебания. В диапазоне средних волн сообщение воспроизводится амплитудой колебаний, а в диапазоне ультракоротких волн — частотой колебаний (***амплитудная*** и ***частотная модуляции*** соответственно). Таким образом,

в первом случае параметром сигнала является амплитуда, а во втором — частота колебаний (рис. 19).

Если для передачи сообщений используются импульсы, то параметром сигнала может быть либо амплитуда импульса, либо интервал между импульсами (***амплитудно-импульсная*** и ***частотно-импульсная модуляции*** соответственно). С последней из названных мы уже познакомились, когда рассматривали вопрос о проведении нервного раздражения (см. 1.2.1, рис. 5).

1.4. Дискретные сообщения

Сигнал называется ***дискретным***, если параметр сигнала может принимать лишь конечное число значений, и существен лишь в конечном числе моментов времени (возможно, периодически повторяющихся).

Дискретными сообщениями называются такие сообщения, которые могут быть переданы с помощью дискретных сигналов.

1.4.1. Знаки, наборы знаков, алфавиты

„Das ABC ist äußerst wichtig,
im Telefonbuch steht es richtig." [1]

Иоахим Рингельнац [2]

Языковые сообщения в письменной форме строят обычно, записывая знаки письма (***графемы***) друг за другом. Хотя длинные сообщения могут размещаться на многих строчках и страницах, это разбиение не имеет, вообще говоря, никакого значения; оно не несёт важной информации. По существу такие сообщения являются последовательностями знаков. Это оказывается справедливым и для устных языковых сообщений, если разложить устный текст на элементарные составные части, так называемые ***фонемы***, и под знаками понимать фонемы. Чтобы можно было воспроизводить фонемы и письменно, принято соглашение о международных письменных знаках для отдельных фонем. В табл. 3 перечислены фонемы, встречающиеся в немецком языке (исключая заимствованные слова).

Точка зрения, что сообщение есть последовательность знаков, не ограничивается, разумеется, тем случаем, когда знаки — это фонемы или графемы (например, знаки букв и цифр, знаки препинания). Знаки планет или знаки зодиака и даже кивок и покачивание головой также могут пониматься как зна-

---

[1] „Знать алфавит ужасно важно,
он в телефонной книге каждой."
(*нем.*). Перевод Л. Макаровой. — *Прим. изд. ред.*

[2] Joachim Ringelnatz (1883—1934), немецкий поэт-сатирик. — *Прим. перев.*

Фонемы немецкого языка

Знаки международной фонетической транскрипции	Пример записи в фонетической транскрипции	Пример записи по правилам немецкой орфографии
[:]	[ra:tə]	Rate
[a]	[ratə]	Ratte
[ɛ]	[fɛt]	Fett
[e]	[lebɛn]	Leben
[ə]	[maxə]	mache
[ɪ]	[mɪç]	mich
[i]	[filaiçt]	vielleicht
[œ]	[hœlə]	Hölle
[ø]	[hølə]	Höhle
[ʏ]	[fʏnf]	fünf
[y]	[kyn]	kühn
[ɔ]	[fɔl]	voll
[o]	[zon]	Sohn
[ʊ]	[kʊrts]	kurz
[u]	[mut]	Mut
[p]	[par]	Paar
[b]	[bal]	Ball
[t]	[takt]	Takt
[d]	[dan]	dann
[k]	[kalt]	kalt
[g]	[gast]	Gast
[m]	[markt]	Markt
[n]	[næn]	nein
[ŋ]	[laŋ]	lang
[l]	[lant]	Land
[r]	[redə]	Rede
[f]	[fal]	Fall
[v]	[vant]	Wand
[s]	[lasən]	lassen
[z]	[razən]	Rasen
[ʃ]	[ʃandə]	Schande
[ʒ]	[ʒenɪ]	Genie
[ç]	[ɪç]	ich
[j]	[ja]	ja
[x]	[bax]	Bach
[h]	[hant]	Hand
[ae]	[baen]	Bein
[ao]	[haot]	Haut
[ˈ]	[be'obaxtuŋ]	Beobachtung

ки[1]. Поэтому мы определим понятие знака существенно шире, чем это можно извлечь из примеров в начале раздела.

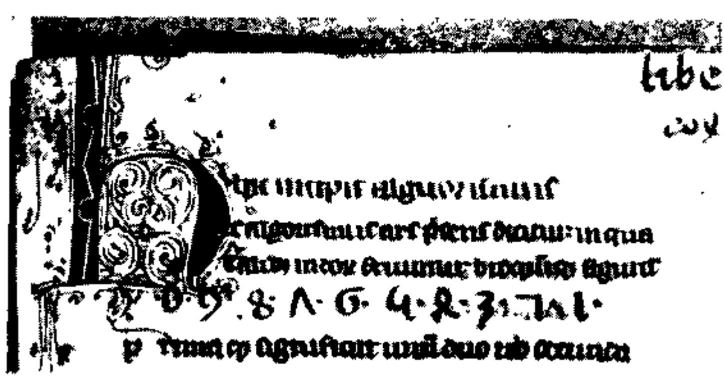


Рис. 20. „Ода алгоритму“ Александра де Вилла Дея обучает новым правилам вычислений в стихах. Рукопись 13-го века. Гесская земельная библиотека Дармштадт.

Знак [2] *— это элемент некоторого конечного множества* [3] *отличимых друг от друга* „вещей“, ***набора знаков.***

Набор знаков, в котором определён (линейный) порядок знаков, называется ***алфавитом*** [4].

Вот некоторые примеры алфавитов (порядок в них — это порядок перечисления):

а) алфавит десятичных цифр

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

---

[1] В Германии [как и во многих других странах. — *Ред.*] кивок головой обычно передаёт информацию ›да‹, или ›верно‹, а покачивание головой — ›нет‹, или ›неверно‹. В Греции же кивок головой вверх означает ›нет‹, а кивок головой вниз — ›да‹. Таким образом, и самые элементарные знаки могут по-разному интерпретироваться и передавать разную информацию.

[2] Процитируем DIN 44300 [*западногерманский промышленный стандарт.— Перев.*]: „**знак**: *элемент определенного конечного множества (различных) элементов. Это множество элементов называют* **набором знаков** (*character set* [английский термин. — *Перев.*]).“

[3] Теоретически можно было бы рассматривать и счётные наборы знаков. В формальной логике иногда так и поступают. Практически же этот вопрос не имеет никакого значения, поскольку за конечное время всегда можно передать только сообщения, построенные из конечного числа знаков.

[4] Снова процитируем DIN 44300: „**алфавит**: *набор знаков, упорядоченный в определённой последовательности.*“ Различие между алфавитом и набором знаков часто упускают из виду.

Как и на телефонном диске, ноль уже в древней арабской арифметике располагали в конце цифрового ряда; в Carmen de Algorismo[1] (рис. 20) мы обнаруживаем цифры 0 9 8 7 6 5 4 3 2 1 (читаемые справа налево);

b) алфавит заглавных латинских букв

$$\{A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z\};$$

c) алфавит строчных греческих букв

$$\{\alpha,\beta,\gamma,\delta,\varepsilon,\zeta,\eta,\theta,\iota,\varkappa,\lambda,\mu,\nu,\xi,o,\pi,\rho,\sigma,\tau,\nu,\varphi,\chi,\psi,\omega\};$$

d) алфавит заглавных кириллических букв

{А,Б,В,Г,Д,Е,Ж,З,И,Й,К,Л,М,Н,О,П,Р,С,Т,
У,Ф,Х,Ц,Ч,Ш,Щ,Ъ,Ы,Ь,Э,Ю,Я};

e) алфавит 12 знаков зодиака

{♈ ♉ ♊ ♋ ♌ ♍ ♎ ♏ ♐ ♑ ♒ ♓}.

f) алфавит японской катаканы[2] (рис. 21).

Вот некоторые наборы знаков, для которых нет какого-либо общепринятого порядка знаков:

g) набор знаков международной фонетической транскрипции (Свит (1877 г.), Пасси (1887 г.)), т. е. набор фонем, встречающихся в естественных языках (см. табл. 3);

h) набор знаков клавиатуры пишущей машинки;

i) набор знаков азбуки Брайля (для слепых; рис. 22);

j) набор китайских идеограмм (несколько тысяч знаков);

k) набор знаков планет

{☿ ♀ ♁ ♂ ♃ ♄ ♅ ♆ ♇}.

l) набор знаков фаз луны

{● ◐ ○ ◑}

[1] Поэма об алгоритмах (*лат.*). — *Прим. перев.*

[2] Один из двух шрифтов японской фонетической (слоговой) азбуки — каны. — *Прим. ред.*

Рис. 21. Катакана. [Латинскими буквами указана международная, русскими — принятая у нас транскрипция слогов катаканы. — *Перев.*]

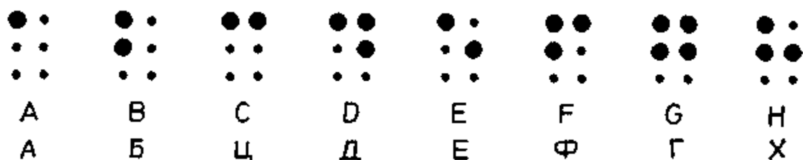
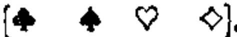


Рис. 22. Некоторые знаки рельефного шрифта Брайля. [„Русские значения" знаков добавлены при переводе. — *Перев.*]

m) набор знаков мастей игральных карт



n) набор шерлок-холмсовских „пляшущих человечков" (рис. 23);

Рис. 23. Секретное сообщение ("Am here Abe Slaney" — „Я здесь Эйб Слейни") из „Пляшущих человечков" Конан Дойля.

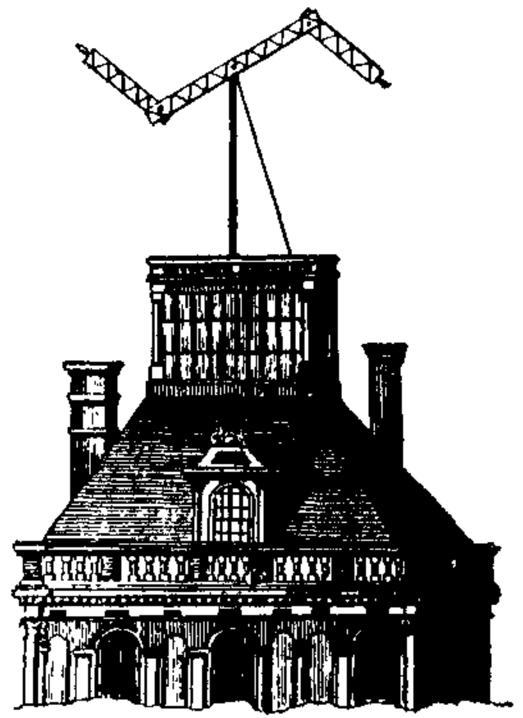


Рис. 24. Телеграф Шаппа (с гравюры того времени).

о) набор знаков „генетического кода", состоящий из четырёх букв А, Ц, Г и Т, обозначающих соответственно химические соединения: аденин, цитозин, гуанин и тимин.

Богатую историю имеет оптическая передача сообщений, начинающаяся факелами Полибия и впервые приобретающая серьёзное военно-политическое значение с появлением оптического телеграфа Клода Шаппа[1] (1791 г.), см. рис. 24. И в наши дни на флоте используются:

[1] Claude Chappe (1763—1805), французский механик, изобретатель оптического телеграфа. — *Прим. перев.*

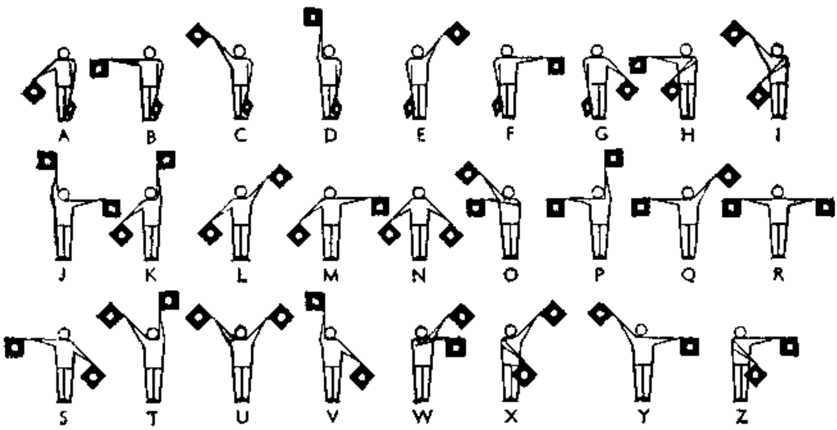


Рис 25 Международный семафорный код

A B C D E
F G H I J
K L M N O
P Q R S T
U V W X Y
Z

красный жёлтый черный
зеленый синий белый

Рис 26 Международный флажковый код.

p) алфавит международного кода семафорной сигнализации (рис. 25);

q) алфавит международного кода флажковой сигнализации (рис 26).

В черно-белом изображении флажкового кода на рис. 26 использован применяемый в геральдике

r) международный штриховой код для цветов (красок).

С появлением электрического телеграфа, а позднее — технологии обработки данных возникли важные технические коды (подробности в 1.4 2):

s) набор знаков азбуки Морзе (рис. 27);

Буквы			Цифры
А · —	К — · —	Ф · · — ·	1 · — — — —
Б — · · ·	Л · — · ·	Х · · · ·	2 · · — — —
В · — —	М — —	Ц — · — ·	3 · · · — —
Г — — ·	Н — ·	Ч — — — ·	4 · · · · —
Д — · ·	О — — —	Ш — — — —	5 · · · · ·
Е ·	П · — — ·	Щ — — · —	6 — · · · ·
Ж · · · —	Р · — ·	Ь, Ъ — · · —	7 — — · · ·
З — — · ·	С · · ·	Ы — · — —	8 — — — · ·
И ·	Т —	Э · · — · ·	9 — — — — ·
Й · — — —	У · · —	Ю · · — —	0 — — — — —
		Я · — · —	

Рис 27 Азбука Морзе [В оригинале приведен немецкий варнант азбуки — *Перев*]

t) набор знаков 2-го международного телеграфного кода CCIT-2 [1] (рис 28);

u) набор знаков кода ЕС ЭВМ для пробивки перфокарт (рис. 29);

v) набор знаков 7-разрядного двоичного кода ISO [2] (рис. 30).

Особенно важное значение имеют наборы, состоящие всего из двух знаков. Такие наборы называют ***двоичными наборами знаков*** [3], а сами знаки — ***двоичными знаками*** [4]. Вместо термина

---

[1] CCIT — сокращение для Comité consultatif international de telegraphie (Международный консультативный комитет телеграфии)

[2] ISO — сокращение для International Standards Organisation (Международная организация по стандартизации)

[3] Процитируем DIN 44300 „**бинарный** (двоичный) — могущий принимать два значения"

[4] Согласно DIN 44300 „**двоичный знак** каждый знак из набора знаков состоящего из двух знаков"

номер разряда (T – разряд такта)		5	4	3	T	2	1
пусто	пусто				•		
3	E				•		●
перевод строки	перевод строки				•	●	
пропуск	пропуск			●	•		
возврат каретки	возврат каретки		●		•		
5	T	●			•		
–	A				•	●	●
8	I			●	•	●	
,	N		●	●	•		
9	O	●	●		•		
'	S			●	•		●
4	R		●		•	●	
10	H	●		●	•		
кто там	D		●		•		●
)	L	●			•	●	
+	Z	●			•		●
7	U			●	•	●	●
:	C		●	●	•	●	
.	M	●	●	●	•		
[	F		●	●	•		●
]	G	●	●		•	●	
;	J		●		•	●	●
0	P	●		●	•	●	
2	W	●			•	●	●
×	B	●	●		•		●
6	Y	●		●	•		●
(	K		●	●	•	●	●
=	V	●	●	●	•	●	
/	X	●	●	●	•		●
1...	1...	●	●		•	●	●
1	Q	●		●	•	●	●
A...	A...	●	●	●	•	●	●
A... буквы (аннулирование при работе с перфолентой)							
1... цифры и знаки							

Рис. 28. Набор знаков 2-го международного телеграфного кода (распределение знаков выполнено группой ALCOR).

„двоичный знак“ часто употребляют сокращение ***бит***[1] (от английского binary digit). Примерами двоичных наборов являются:

[1] Слово „бит“ имеет ещё и другое значение, см. подстрочное примечание на стр. 621. — *Прим. перев.*

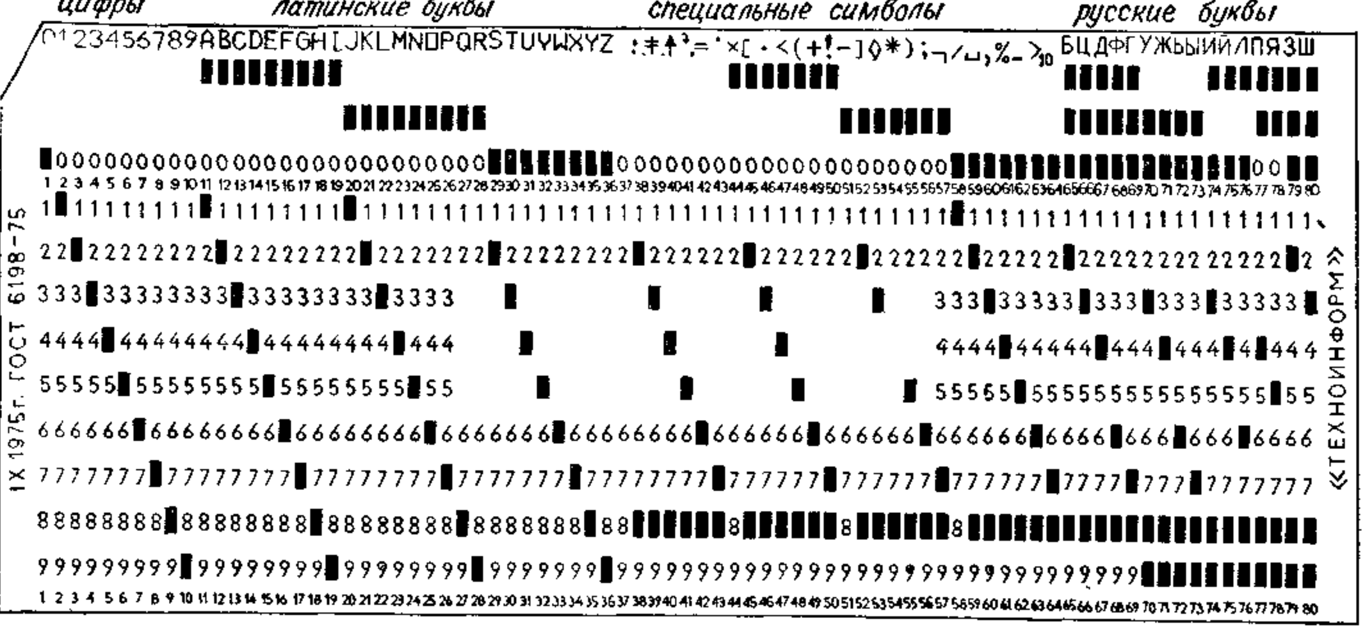


Рис 29. Набор знаков кода ЕС ЭВМ для пробивки перфокарт. [В оригинале приведён набор знаков кода IBM. — *Перев.*]

	OOO	OOL	OLO	OLL	LOO	LOL	LLO	LLL
OOOO	NUL	DLE		0	@	P		p
OOOL	SOH	DC1	!	1	A	Q	a	q
OOLO	STX	DC2	"	2	B	R	b	r
OOLL	ETX	DC3	#	3	C	S	c	s
OLOO	EOT	DC4	$	4	D	T	d	t
OLOL	ENQ	NAK	%	5	E	U	e	u
OLLO	ACK	SYN	&	6	F	V	f	v
OLLL	BEL	ETB	'	7	G	W	g	w
LOOO	BS	CAN	(	8	H	X	h	x
LOOL	HT	EM	)	9	I	Y	i	y
LOLO	LF	SUB	*	:	J	Z	j	z
LOLL	VT	ESC	+	;	K	[	k	{
LLOO	FF	FS	,	<	L	\	l	|
LLOL	CR	GS	–	=	M	]	m	}
LLLO	SO	RS	.	>	N	^	n	~
LLLL	SI	US	/	?	O	_	o	DEL

Рис. 30. Набор знаков 7-разрядного двоичного кода ISO в американской редакции (ASCII). Первые два столбца содержат так называемые управляющие символы. Сплошным треугольничком в правом верхнем углу помечены позиции, которые, согласно принятым в данной стране нормам, могут заполняться иначе. Комбинация OOOOOLO означает »пробел«.

aa) пара цветов {»красный«, »зелёный«},
bb) пара яркостей {»светлый«, »тёмный«},
cc) пара цифр {0, 1},
dd) пара состояний {»пробивка«, »нет пробивки«},
ee) пара жестов {»кивок головой«, »покачивание головой«},
ff) пара значений истинности {»истина«, »ложь«},
gg) пара ответов {»да«, »нет«},

hh) пара знаков {↶, ↷} (»повернуть налево«, »повернуть направо«),

ii) пара знаков {♂, ♀} (»мужской«, »женский«),

jj) пара знаков {⊙, ◎} (»включено«, »выключено«),

kk) пара напряжений {12 В, 2 В},

ll) пара импульсов {⊓, ⊔},

mm) пара знаков {·, —} (»точка«, »тире«),
nn) пара знаков {+, —},
oo) пара знаков {→, ←} (»направо«, »налево«),
pp) пара знаков {○, ●},
qq) пара знаков {<, >} (»меньше«, »больше«).

В качестве абстрактных знаков для двоичных наборов мы будем использовать знаки {L, O}. При этом никакого постоянного соответствия между этим набором и другими двоичными наборами не устанавливается; L может отождествляться с I, »да«, »истина«, »светлый«, но не обязательно.

1.4.2. Коды и кодирования

Если N — предложение некоторого естественного языка, то N можно рассматривать как последовательность знаков по крайней мере тремя разными способами.

Прежде всего N представляет собой последовательность букв, цифр, знаков препинания и т. д.; далее, N — это последовательность слов, которые в другом контексте могут сами рассматриваться как знаки; наконец, и всё предложение целиком можно рассматривать как один знак.

Первое понимание используется, например, когда имеется правило для нанесения сообщения N на перфокарты; второе понимание лежит в основе стенографических сокращений; крайнее третье понимание бывает уместным при переводе на другой естественный язык, когда пословица одного языка переводится соответствующей по смыслу пословицей другого языка.

Дискретные сообщения представляют собой (конечные или бесконечные) ***последовательности знаков***[1]. При этом, исходя из соображений, связанных с физиологией органов чувств, или из чисто технических соображений, их обычно разбивают на конечные последовательности знаков[2], называемые ***словами***[3]. На более высоком уровне каждое слово можно снова рассматривать как знак[4], при этом соответствующий набор знаков будет, вообще говоря, шире первоначального. Обратно, данный набор знаков можно получить с помощью составления слов, исходя из некоторого набора с меньшим числом знаков, в частности из двоичного набора знаков. Некоторые из перечисленных

---

[1] Синоним ***цепочки знаков*** (Туэ, Сколем).

[2] Само сообщение, напротив, с теоретической точки зрения не обязано быть конечной последовательностью знаков.

[3] Из DIN 44300: „слово: *последовательность знаков, которая в определённом контексте рассматривается как нечто целое.* Сюда включается и пустое слово."

[4] Если к этому слову добавляется ещё и его значение, или смысл (см. 1.4.4), то говорят о ***словарном символе***

выше наборов получены с помощью словообразования „над" конкретными двоичными наборами знаков или, абстрактно, над набором {L,O}.

Слова над двоичным набором знаков называются ***двоичными словами***. Они не обязаны иметь постоянную длину (см. азбуку Морзе), если это всё же так, то говорят об n-разрядных двоичных знаках и n-разрядных двоичных кодах (см., например, 2-й международный телеграфный код (5-разрядные двоичные знаки), 7-разрядный двоичный код ISO).

Дадим теперь точное определение:

Кодом[1, 2] называется правило, описывающее отображение одного набора знаков в другой набор знаков (или слов); также называют и множество образов при этом отображении

Помимо основного значения слова „code" — «кодекс», «свод законов» (гражданский кодекс, кодекс Наполеона) — начиная с середины 19-го в. оно означало книгу, в которой словам естественного языка сопоставлены группы цифр или букв. Употребление таких кодов приобрело значение скорее в связи со стремлением сэкономить на стоимости телеграмм, чем в связи с соображениями конспиративности (ABC-код В. Клаузен-Туэ, 1874).

Если каждый образ при кодировании является отдельным знаком, то такое отображение мы называем ***шифровкой***, а образы — ***шифрами*** (англ. cipher). Поскольку здесь имеется криптографический аспект, обращение этого отображения — когда оно однозначно — называется ***декодированием или дешифровкой***.

В коммерческих и криптографических кодах слова, фразы и понятия естественных языков кодируются в большинстве случаев словами над некоторым буквенным или цифровым алфавитом, обычно пятерками (рис. 31). В технических кодах буквы, цифры и другие знаки почти всегда кодируются двоичными словами. Примеры можно найти на приведённых выше рисунках.

У большинства используемых в технике кодов все слова имеют одинаковую длину. Самый старый из них — это восходящий к И. М. Э. Бодо 2-й международный телеграфный код CCIT-2 (рис. 28), пятиразрядный двоичный код, который до сих пор используется во внутренней и международной открытой телеграфной связи (телекс)[3]. Несколько моложе семиразряд-

---

[1] Из DIN 44300: „**код**: *1. Правило, описывающее однозначное соответствие* (**кодирование**) *знаков одного набора знакам другого набора (множества образов). 2. Набор знаков, который выступает в качестве множества образов при кодировании.*"

[2] Часто знаки исходного набора или множества образов являются последовательностями над некоторым другим набором знаков

[3] По проводам L передается посредством „такта тока", а O — посредством „такта паузы".

ный двоичный ISO (рис. 30), имеющий международный стандарт[1]. Код ЕС ЭВМ (рис. 29) для пробивки перфокарт является двенадцатиразрядным двоичным кодом Он восходит к Х. Холлериту (1860—1929).

M. N. O. P. R. S. T. U. Y. Z.
0 1 2 3 4 5 6 7 8 9

19140	UVVIM	slackness.	Schlaffheit, Geschäftsstille.	вялость, затишье.
19141	UVVON	**Slag (s).**	**Schlacke(n).**	шлак.
19142	UVWEO	**Slander(s).**	**Verleumden (-e, -t).** Verleumdung(en).	клевета.
19143	UVWUP	slandered.	verleumdet.	оклеветанный.
19144	UVWYR	slandering.	verleumdend.	дискредитирующий.
19145	UVYBS	slanderous.	verleumderisch.	клеветнический.
19146	UVYCT	**Slate(s).**	**Schiefer**, Schiefertafel(n).	шифер, сланец.
19147	UVYDU	**Sleeper(s).**	**(Bahn-) Schwelle(n).**	шпала.
19148	UVYFY	**Sleeve-valve.**	**Muffenventil.**	вентиль (с муфтой).
19149	UVYMZ	**Slide(s).**	**Gleiten (-e, -et)**, Gleitbahn(en), Gleitführung(en).	скольжение.
19150	UVYUM	slide-valve	Schieberventil, Ventilschieber.	вентиль с заслонкой.
19151	UVYVN	sliding.	gleitend.	скользящий.
19152	UVYWO	sliding scale.	Gleitskala.	плавающая шкала.
19153	UVYZP	**Slight**	**Gering**, von geringer Wichtigkeit.	ничтожный, незначительный.
19154	UVZUR	slightest.	geringst.	минимальный, самый ничтожный.
19155	UVZYS	not the slightest.	nicht das (der, die) geringste.	немаловажный.
19156	UWAFT	slightly.	in geringem Masse, leicht.	легкий, ничтожной массы.
19157	UWAGU	**Slime(s).**	**Schleim(e)**, Schlamm, (Schlämme).	грязь, ил, слизь.
19158	UWAHY	slimy.	schleimig, schlammig.	клейкий, вязкий.
19159	UWALZ	**Slip(s).**	**Schlüpfen (-e, -t)**, (aus) gleiten (-e, -t); Fehltritt(e).	выскальзывать, спотыкаться.

Рис. 31. Многоязычный коммерческий код Джеймса К. Х. Макбета (Marconi Wireless Telegraph Co. Ltd.). [В оригинале крайний правый столбец — на голландском, а не на русском языке. — *Перев.*]

В криптографических целях двоичный код — тоже пятиразрядный — использовался Фрэнсисом Бэконом[2] уже в 1580 г. На рис. 32 приведена его таблица кодов для 24 знаков (обращаем внимание, что в соответствии с правилами того времени *u* и *v* не различаются). Этот же код был использован в оптиче-

[1] Западногерманская редакция стандарта зафиксирована в DIN 66003.
[2] Fransis Bacon (1561—1621), английский философ, современник Шекспира.

ском ламповом телеграфе Шюди (1787). Слова кода упорядочены лексикографически (***прямой код***).

Кажущееся случайным расположение слов кода Бодо определяется тем, что с учетом частоты знаков количество тактов тока и тем самым расход энергии оказываются минимальными.

a AAAAA	e AABAA	i ABAAA	n ABBAA	r BAAAA	w BABAA
b AAAAB	f AABAB	k ABAAB	o ABBAB	s BAAAB	x BABAB
c AAABA	g AABBA	l ABABA	p ABBBA	t BAABA	y BABBA
d AAABB	h AABBB	m ABABB	q ABBBB	v BAABB	z BABBB

Рис. 32. Двоичный код Фрэнсиса Бэкона (около 1580 г.).

В английском буквы по частоте употребления располагаются так:

ETAONIRSHDLU...

Этот порядок отражён также в клавиатуре линотипа, изобретенного Оттмаром Мергенталером в 1886 г.

Особым случаем является двоичное кодирование цифр. Оно осуществляется методами, учитывающими наряду с прочим действия над числами, в частности их сложение. На рис. 33 приведены некоторые из наиболее распространённых двоичных кодов постоянной длины для десятичных цифр.

Значение этих кодов в последние годы заметно снизилось, так как машинные вычисления над числами осуществляются преимущественно в системе счисления с основанием 2, т. е. они производятся с ***двоичными цифрами*** 0 и 1. Числа, записанные в двоичной системе счисления, т. е. ***двоичные числа***, — это слова, составленные из двоичных цифр, и eo ipso[1] двоичные слова. Часто устанавливают соответствие

$$0 \triangleq \mathrm{O},$$

$$1 \triangleq \mathrm{L},$$

хотя и против противоположного соответствия

$$0 \triangleq \mathrm{L},$$

$$1 \triangleq \mathrm{O}$$

также возразить нечего.

Более подробные сведения о системах счисления приведены в приложении А.

---

[1] Тем самым (*лат.*). — *Прим. перев.*

символ цифры	двоичный код прямой	Грея	плюс-3 (Штибица)	Грея–Штибица	Айкена	биквинарный	1-из-10	2-из-5	CCIT-2
0	OOOO	OOOO	OOLL	OOLO	OOOO	OOOOOL	OOOOOOOOOL	LLOOO	OLLOL
1	OOOL	OOOL	OLOO	OLLO	OOOL	OOOOLO	OOOOOOOOLO	OOOLL	LLLOL
2	OOLO	OOLL	OLOL	OLLL	OOLO	OOOLOO	OOOOOOOLOO	OOLOL	LLOOL
3	OOLL	OOLO	OLLO	OLOL	OOLL	OOLOOO	OOOOOOLOOO	OOLLO	LOOOO
4	OLOO	OLLO	OLLL	OLOO	OLOO	OLOOOO	OOOOOLOOOO	OLOOL	OLOLO
5	OLOL	OLLL	LOOO	LLOO	LOLL	LOOOOL	OOOOLOOOOO	OLOLO	OOOOL
6	OLLO	OLOL	LOOL	LLOL	LLOO	LOOOLO	OOOLOOOOOO	OLLOO	LOLOL
7	OLLL	OLOO	LOLO	LLLL	LLOL	LOOLOO	OOLOOOOOOO	LOOOL	LLLOO
8	LOOO	LLOO	LOLL	LLLO	LLLO	LOLOOO	OLOOOOOOOO	LOOLO	OLLOO
9	LOOL	LLOL	LLOO	LOLO	LLLL	LLOOOO	LOOOOOOOOO	LOLOO	OOOLL
веса позиций	8421	15731			2421	543210	9876543210	74210	

Рис. 33. Наиболее распространенные двоичные коды десятичных цифр.

Цифры естественным образом образуют алфавит: порядок цифр — это порядок счёта. В алфавитах может оказаться полезным, чтобы соседние знаки при двоичном кодировании переходили в минимально отличающиеся друг от друга слова, т. е. в слова, различающиеся лишь в одном бите. Существуют коды, которые удовлетворяют этому условию; они называются

Рис. 34. „Извилина" кода Грея (см. рис. 33) и её вариант.

кодами Грея или ***одношаговыми кодами***. На рис. 34а изображена „извилина", соответствующая той последовательности, в которую могут быть выстроены расположенные в 4 строках 16 четырёхразрядных комбинаций, чтобы дать код Грея для десятичных цифр. На рис. 34b приведён вариант, являющийся одношаговым и для перехода от 9 к 0, т. е. являющийся ***циклически одношаговым***. Одношаговый пятиразрядный двоичный код впервые использовал Бодо в 1874 г. для своего печатающего телеграфа.

Интересно, что n-разрядный двоичный код для алфавита часто можно получить, передвигая ***смотровое*** (считывающее) ***окошко*** вдоль подходящего циклического расположения не более 2^n битов так, чтобы оно в каждый момент времени выхватывало n следующих друг за другом битов. Для $n = 3$ это можно сделать, например, расположив $2^3 = 8$ битов в следующем порядке:

→OOOLLLOL

или 7 битов в порядке

→OOLLLOL

Коды с таким порядком алфавита называются ***циклическими кодами***. Первый циклический код (для $n = 5$) также был открыт Бодо (1882 г.). ***Код 1-из-n*** тоже может рассматриваться как циклический, а именно как n-разрядный двоичный код с циклическим расположением n битов, которые все, за исключе-

нием одного, совпадают друг с другом (пример кода 1-из-10 приведён на рис. 33).

Если двоичные слова, поставленные в соответствие цифрам, идут начиная с OO ... O в лексикографическом порядке, как в коде Бэкона (рис. 32), то и здесь говорят о ***прямом коде***. При этом число, которое определяется цифрой, называется ***прямым целочисленным эквивалентом*** двоичного слова. Прямой и некоторые другие из приведённых на рис. 33 кодов десятичных цифр являются двоичными ***кодами с весами позиций***[1]: прямой код имеет поразрядные веса позиций 8-4-2-1, код Айкена — веса 2-4-2-1. Очевидно, что коды с весами позиций упрощают сложение — для таких кодов оно осуществляется поразрядным сложением с переносом. Код плюс-3 — это сдвинутый прямой код: чтобы получить представляемую данным двоичным словом цифру, нужно из прямого целочисленного эквивалента вычесть 3. В этом коде взаимно дополнительные пары цифр, т. е. 0 и 9, 1 и 8 и т. д., переходят друг в друга в результате взаимной замены O на L, тем самым код плюс-3 упрощает получение противоположных (т. е. взятых с обратным знаком) чисел (см. также A.2).

4×7 5×7 4×7

18 бит 35 бит 9 бит

Рис. 35. Коды для матричной индикации.

Троичное кодирование, несмотря на ряд технических попыток, успеха не имело. Четвертичное кодирование в криптографических целях использовал Альберти ещё в 1466 г. Оно представлено также „генетическим кодом“ (см. 1.4.1), в котором используются слова длины 3. Из 64 комбинаций 61 поставлена

[1] В случае кодов с весами позиций для получения значения десятичной цифры по её коду веса двоичных разрядов (взятые со знаком плюс или минус) умножаются на значения разрядов и складываются — *Прим. ред.*

в соответствие двадцати встречающимся в природе аминокислотам. Разумеется, это соответствие неоднозначно, так например, глутаминовая кислота представляется двумя кодовыми словами ЦТТ и ЦТЦ, аланин — четырьмя кодовыми словами ЦГА, ЦГГ, ЦГТ и ЦГЦ, а аргинин — шестью кодовыми словами ГЦА, ГЦГ, ГЦТ, ГЦЦ, ТЦТ и ТЦЦ. Точный смысл остальных трёх кодовых слов ещё не выяснен. Предполагается, что

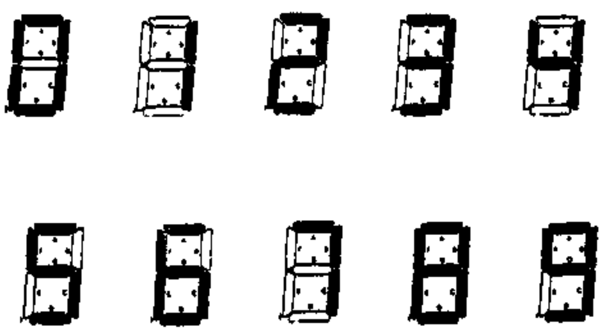


Рис. 36. Семисегментный код для световой индикации карманного калькулятора.

они служат для маркировки начала и конца цепочки аминокислот.

Для световой индикации цифр (и букв) используются специальные коды. На рис. 35 показаны коды для лампочной индикации, а на рис. 36 — широко распространённый код со светящимися сегментами.

Коды со словами разной длины встречаются в технике довольно редко. Исключением является код Морзе (рис. 27). Грубо говоря, это двоичный код с набором знаков {»точка«, »тире«} и словами длины не более 5 для кодирования букв и цифр. Более точно, следует ещё добавить в качестве третьего знака знак »пропуск«, который помечает стыки между кодовыми словами[1] (слова нельзя отделить друг от друга по их длине). Установив соответствие

$$\begin{aligned} \cdot &\triangleq \mathrm{OL} \\ - &\triangleq \mathrm{OLLL} \\ \text{»пропуск«} &\triangleq \mathrm{OOO} \end{aligned}$$

код Морзе можно рассматривать как двоичный код. Это двоичное кодирование, соответствующее общепринятому правилу ра-

[1] Подобным образом был устроен, по-видимому, и код, использованный Гауссом и Вебером ещё в 1833 г.

дистов „продолжительность точки равна продолжительности паузы, продолжительность тире равна трём продолжительностям точки, продолжительность пропуска равна трём продол-

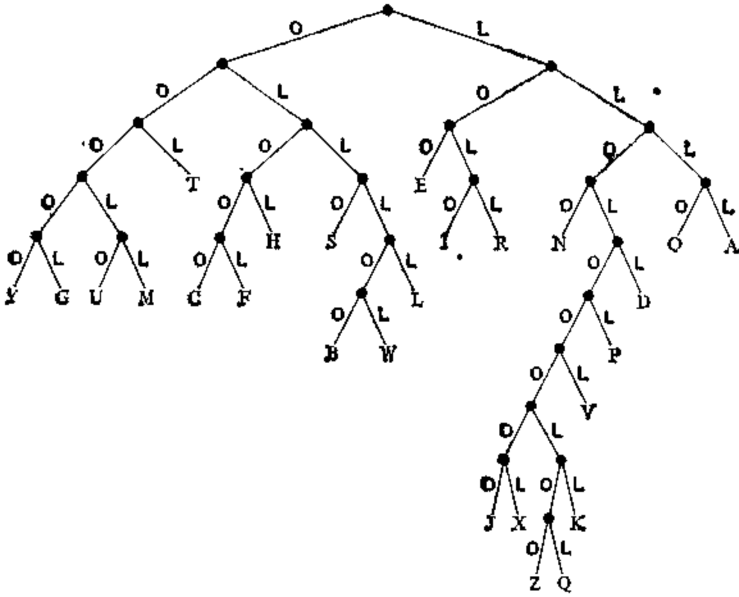


Рис. 37. Кодовое дерево для кода, удовлетворяющего условию Фано.

жительностям паузы“, лежит в основе большинства технических реализаций (ток — нет тока, звук — нет звука). Например:

A ≙ · — ≙ OLOLLL,
Ы ≙ — · — — ≙ OLLLOLLLOLLL.

Другой пример — это код счёта, который лежит в основе системы телефонной связи:

1	LO	6	LLLLLLO
2	LLO	7	LLLLLLLO
3	LLLO	8	LLLLLLLLO
4	LLLLO	9	LLLLLLLLLO
5	LLLLLO	0	LLLLLLLLLLO

Ещё один пример кода со словами разной длины — это кодирование натуральных чисел в двоичной системе счисления

словами без начальных нулей. Число нуль представляется здесь пустым словом.

Конечные двоичные коды можно описывать также с помощью ***кодового дерева*** (рис. 37). На рис. 38 изображено кодовое дерево для азбуки Морзе. С кодовыми деревьями мы ещё

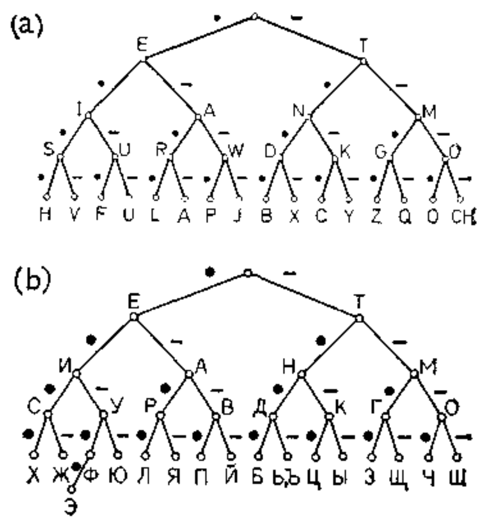


Рис. 38. Кодовое дерево для азбуки Морзе [Дерево (b) для русского варианта азбуки добавлено при переводе. — *Перев.*]

встретимся при рассмотрении диаграмм переходов в теории автоматов (гл. 7).

1.4.3. Последовательная и параллельная передача

В двоичных кодах с постоянной длиной кодовых слов слова могут следовать друг за другом непосредственно (***последовательная передача***), так что получается единая последовательность двоичных знаков. Расположение стыков и тем самым исходная группировка кодовых слов устанавливаются с помощью отсчёта, и, таким образом, сообщения, составленные из кодовых слов, однозначно декодируемы. Правда, при отсчёте кодовой длины нельзя просчитаться, нельзя „сбиться с ритма“, а это ведёт к усложнению с технической точки зрения (параллельность, синхронизация).

Напротив, для кодов с переменной длиной кодовых слов расположение стыков, вообще говоря, восстановить нельзя. При определённых условиях сообщение, состоящее из несколь-

ких кодовых слов, либо вовсе не декодируется, либо декодируется неоднозначно. Однако, декодируемость будет обеспечена, если соблюдается следующее

Условие Фано. Никакое кодовое слово не является началом другого кодового слова („свойство префиксности").

Тогда, очевидно, стык между кодовыми словами определяется тем моментом, когда „дальше не читается". Очевидно также, что код удовлетворяет условию Фано тогда и только тогда, когда кодовое дерево не содержит ни одного языка во „внутренних" вершинах (***дерево с размеченными листьями***).

Условие Фано является достаточным, но не необходимым условием однозначной декодируемости, как это показывает следующий контрпример:

$$A \triangleq \mathsf{L},$$

$$B \triangleq \mathsf{LOL}.$$

Тривиальная возможность обеспечить выполнение условия Фано состоит в том, чтобы каждое кодовое слово начинать специальным знаком (или группой знаков), называемым ***разделителем***. Это, очевидно, имеет место в случае кода Морзе, а именно пропуск является разделителем для последовательности точек и тире, а группа знаков OOO — разделителем при двойном кодировании кода Морзе. С технической точки зрения при передаче по телеграфу также передается разделитель (синхронизирующий »такт разбивки«).

Пример кода, удовлетворяющего условию Фано без употребления разделителя, представлен на рис. 37.

Проблема декодирования кодов с переменной длиной кодовых слов впервые была осознана как таковая итальянскими учёными из семьи Ардженти, жившими в 16-м в. при папском дворе.

При ***параллельной передаче*** мы в отличие от последовательной ограничены кодами со словами постоянной длины: для n-разрядного двоичного кода используется n параллельных двоичных каналов передачи. В случае оптического, электростатического, электролитического и электромагнитного телеграфа путь технического прогресса шёл прежде всего от параллельной к последовательной передаче.

Вопрос о том, какие коды являются оптимальными с точки зрения передачи, изучается в теории информации (см. приложение В).

1.4.4. Символы

„Знак — это чувственно воспринимаемое в символе."

Людвиг Витгенштейн[1]

Следует различать собственно знак и его смысл. Флажковый знак, приведённый на рис. 26 во второй строке вторым слева, означает букву G; однако при другом использовании этого знака он означает также «Мне нужен лоцман». Знак вместе с его смыслом называется ***символом***. Упомянутый флажковый знак является, таким образом, символом для вполне определённого запроса, причём он не только интернационально понятен, его смысл вообще не зависит ни от какого языка речи. Атанасиус Кирхер ещё в 1663 г. привёл список международных понятий, содержащий 1048 символов. Родственная идея заключена в многоязычном коде Маркони (см. рис. 31).

В соответствии с целью употребления один и тот же знак часто имеет разный смысл. Знак ♀ применяется в астрономии как символ планеты Венера, а в биологии — как символ женской особи. К несчастью, часто бывает также, что разные знаки имеют одинаковый смысл; например, знаки $\cdot$ и $\times$, а в последнее время и $*$ все понимаются как символы умножения.

Обычно всякое сообщение имеет смысл, т. е. уже является символом. Очевидно, что этот символ получается в результате присоединения к сообщению той информации, которая им передается.

„Весь наблюдаемый мир — это просто склад образов и знаков."

Бодлер

1.5. Обработка сообщений и обработка информации

1.5.1 Обработка сообщений как кодирование

Всякое правило обработки сообщений можно понимать как отображение (функцию) ν

$$\mathfrak{N} \xrightarrow{\nu} \mathfrak{N}',$$

которое сообщениям N из некоторого множества сообщений $\mathfrak{N}$ ставит в соответствие новые сообщения N' из множества сооб-

---

[1] Ludwig Wittgenstein (1889—1951), австрийский философ и логик, представитель так называемой аналитической философии.— *Прим. ред.*

щений $\mathfrak{N}'$[1]. Каждое из сообщений N и N' — это последовательность знаков (см. 1.4.2).

Большая свобода в понимании сообщения как последовательности знаков, просматриваемая в обсуждавшихся выше примерах, вместе с рассуждениями раздела 1.4.2 позволяет констатировать: *всякую обработку сообщений можно рассматривать как кодирование*[2]. Конечно, это соображение является важным и для изучения процессов обработки сообщений у живых существ, но прежде всего оно лежит в основе всякой машинной обработки дискретных сообщений.

Чтобы правило обработки $\mathfrak{N} \xrightarrow{\nu} \mathfrak{N}'$ могло служить основой для обработки сообщений, недостаточно, чтобы правило ν неким аксиоматическим образом задавало те условия, которым должно удовлетворять сообщение $N' = \nu(N) \in \mathfrak{N}'$. Правило ν должно задавать некоторый с п о с о б п о с т р о е н и я сообщения $\nu(N) \in \mathfrak{N}'$ исходя из сообщения $N \in \mathfrak{N}$. Конечно, если $\mathfrak{N}$ — конечное множество, то это можно сделать посредством перечисления единичных соответствий. Если же $\mathfrak{N}$ бесконечно или, хотя и конечно, но так велико, что перечисление оказывается непрактичным, то нужно задать конечное множество операций (***элементарных шагов*** (или ***тактов***) ***обработки***) таким образом, чтобы каждый переход от N к N' можно было осуществить за конечное число таких элементарных тактов. Кроме того, нужно задать *операционное* ***правило обработки*** — к этому мы ещё вернемся после введения понятия алгоритма в разделе 1.6.4.1.

Так как обработку дискретных сообщений можно рассматривать как кодирование, то те операции, которые следует задать, должны иметь вид преобразований последовательностей знаков („чистая игра со знаками").

Кодирование технически всегда связано с передачей сообщений и поэтому осуществляется во времени (см. 1.3.2). Кодирование, а значит и обработка сообщений, никогда не осуществляется „мгновенно", а всегда требует определённого времени, которым зачастую нельзя пренебречь. При рассмотрении поня-

---

[1] На самом деле это не столько констатация факта, сколько допущение. Авторы уточняют понятие „обработка", сводя его по существу к понятию отображения, или соответствия. Допущение состоит в том, что результатом обработки всегда является именно сообщение, а не что-нибудь другое. — *Прим. ред.*

[2] Строго говоря, такая трактовка является расширительной по отношению к понятию кодирования. Во-первых, она по существу требует допущения рассматривать счётные наборы (см. подстрочное примечание на стр. 42); во-вторых, при обработке сколь угодно длинного сообщения для каждого знака z результирующего сообщения можно было бы указать такой конечный кусок p исходного сообщения, что z является функцией исключительно p. — *Прим. ред.*

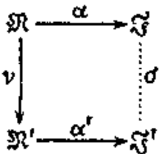
тия обработки сообщений этот факт служит существенным дополнением к понятию отображения в чистой математике. Его стараются не замечать главным образом потому, что правило обработки ν — но не фактическое выполнение отображения — часто можно задать как отображение в математическом смысле. Зависимость же от времени приводит к понятию ***эффективности правила обработки*** сообщений, определяемой объёмом и длительностью процесса обработки, в сравнении с другими процессами, дающими тот же результат. В последующих главах мы ещё вернемся к этому.

1.5.2. Интерпретация обработки сообщений

Множество $\mathfrak{N}$ сообщений N представляет интерес только тогда, когда ему посредством некоторого правила соответствия α сопоставлено (по крайней мере одно) множество $\mathfrak{J}$ сведений[1] J:

$$\mathfrak{N} \xrightarrow{\alpha} \mathfrak{J}.$$

Так как множеству сообщений $\mathfrak{N}'$ также соответствует некоторое множество сведений $\mathfrak{J}'$, то правило обработки $\mathfrak{N} \xrightarrow{\nu} \mathfrak{N}'$ даёт нам следующую диаграмму:



В каком отношении между собой находятся $\mathfrak{J}$ и $\mathfrak{J}'$? Очевидно, что каждому сообщению $N \in \mathfrak{N}$ сопоставлена пара (J, J'), $J = \alpha(N)$, $J' = \alpha'(\nu(N))$; тем самым определено соответствие σ между $\mathfrak{J}$ и $\mathfrak{J}'$. Если α — необратимое[2] отображение, т. е. существуют два сообщения N_1 и N_2, которые передают одну и ту же информацию J, то соответствие σ может и не быть отображением, поскольку обработанные сообщения $\nu(N_1)$ и $\nu(N_2)$ могут нести различные информации $J'_1 = \alpha'(\nu(N_1))$, $J'_2 = \alpha'(\nu(N_2))$. *Говорят, что правило обработки* ν ***сохраняет***

[1] Когда нам будет нужно слово „информация" во множественном числе, мы часто будем писать „сведения". — *Прим. перев.*

[2] Авторы используют слово „обратимый" (в оригинале umkehrbar) в смысле „инъективный", а не „биективный". (Инъективное отображение (инъекция) — это взаимно однозначное отображение *в*, т. е., быть может, не *на всё* множество образов, а биективное отображение (биекция) — это взаимно однозначное отображение *на*). — *Прим. ред.*

информацию, если соответствие σ является отображением. Тогда мы имеем диаграмму

$$(\ast)\qquad \begin{array}{ccc} \mathfrak{N} & \xrightarrow{\alpha} & \mathfrak{J} \\ \nu\downarrow & & \downarrow\sigma \\ \mathfrak{N}' & \xrightarrow{\alpha'} & \mathfrak{J}' \end{array}$$

где композиция отображений α и σ совпадает с композицией отображений ν и α':

$$\sigma\alpha = \alpha'\nu.$$

В таком случае диаграмма (∗) называется ***коммутативной***, а отображение σ называют правилом ***обработки информации.***

Обычно сообщения обрабатывают именно для того, чтобы обработать информацию. Фактически всегда исходят из определённого правила σ и пытаются определить ν, α и α' таким образом, чтобы получилась ситуация, представленная на диаграмме (∗). Поэтому мы можем предполагать в дальнейшем, что ν сохраняет информацию, так что отображения ν, α и α' определяют некоторое правило σ обработки информации.

В соответствии с тем, является σ обратимым отображением или нет, мы различаем следующие случаи:

1. Если σ — обратимое отображение, т. е. информация при обработке не теряется, то соответствующую обработку сообщений называют ***перешифровкой.***

1.1. Если и ν обратимо, то мы имеем простой случай перекодировки: по сообщению $N' = \alpha(N)$ можно восстановить не только исходную информацию, но и само исходное сообщение N. Особенно часто встречается тот частный случай, когда $\mathfrak{J} = \mathfrak{J}'$, а σ — тождественное отображение. В идеале всякая передача сообщений должна иметь именно такой вид.

1.2. Если σ обратимо, а ν — нет, то несколько сообщений $N \in \mathfrak{N}$ будут кодироваться одним и тем же сообщением $N' \in \mathfrak{N}'$. Но так как при этом никакой информации не теряется, то это означает, что исходное множество сообщений $\mathfrak{N}$ было избыточным: в $\mathfrak{N}$ имеется несколько сообщений, которые несут одну и ту же информацию. Во всяком случае, количество сообщений с таким свойством в $\mathfrak{N}'$ меньше, чем в $\mathfrak{N}$. Перешифровку ν такого рода мы называем ***сжимающей.*** Если к тому же α' обратимо, то мы называем ν ***вполне сжимающей.***

2. Если σ — необратимое отображение, т. е. разные сведения $I \in \mathfrak{J}$ отображаются в одну и ту же информацию $I' \in \mathfrak{J}'$, то соответствующую обработку сообщений ν называют ***избирательной.*** Особенно часто встречается случай, когда $\mathfrak{J}'$ — подмно-

жество $\mathfrak{J}$, и σ для сведений из $\mathfrak{J}'$ является тождественным отображением. В этом случае σ по существу производит выбор из заданного множества сведений. Выбор может быть уже предопределён тем, что несколько различных сообщений $N \in \mathfrak{N}$ отображаются в одно сообщение $N' \in \mathfrak{N}'$. Однако обработка сообщений ν вполне могла быть обратимой. В этом случае выбор осуществляется „односторонней" интерпретацией α'.

Проиллюстрируем это на нескольких примерах:

a) Обычный способ чтения газеты избирателен. Изучение разных газетных статей, описывающих одно и то же событие, является сжимающим.

b) Переход от избыточного кода к менее избыточному или вообще к коду без избыточности тем не менее, как правило, однозначно обратим. Таким образом, речь идет о несжимающей перешифровке: уменьшается не количество сообщений, а их длина (см. рис. 33).

c) Пусть сообщение (a, b), составленное из пары двоично закодированных целых чисел (где $b > 0$), передает информацию „рациональное число r, представляемое дробью a/b". Отображение

$$\alpha\colon (a, b) \longmapsto r$$

не является обратимым. Пусть теперь множество пар чисел $\mathfrak{N}$ отображается в подмножество $\mathfrak{N}'$ пар взаимно простых чисел, причем ν: $(np, nq) \longmapsto (p, q)$. Тогда ν — сжимающее отображение; получающееся при этом отбражение α' будет обратимым.

Для вводимых в следующем разделе правил („алгоритмов") обработки дискретных сообщений („объектов") важна общая интерпретация объектов и алгоритмов.

1.6. Алгоритмы

Почти во всех сферах жизни нам приходится иметь дело с инструкциями (предписаниями, рецептами, правилами), в соответствии с которыми можно или нужно что-то сделать. Вот несколько простых примеров:

(a) Инструкция по пользованию телефоном-автоматом, представленная на рис. 39.

(b) „1. Опустить монету.
2. Установить указатель товара на желаемый товар.
3. Потянуть за ручку и держать её, пока товар не выпадет.
При отказе нажать на кнопку возврата денег".

(c) „Если врач не прописал иначе, то 3—4 раза в день по 15—20 капель, лучше всего в горячей сладкой воде. Детям только половину этой дозы".

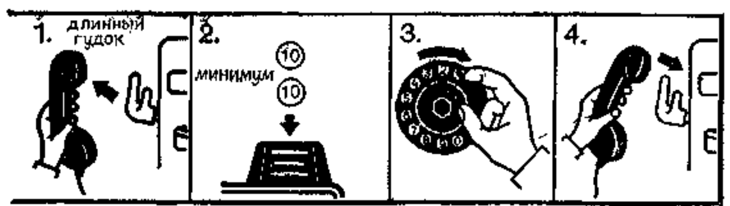


Рис. 39. Инструкция по пользованию телефоном-автоматом.

Сходным образом звучат инструкции по эксплуатации пылесосов, швейных машин, автомобилей, по установке палатки, сборке байдарки, изготовлению модели корабля или самолета.

Когда такие инструкции удовлетворяют определённым минимальным требованиям, говорят об алгоритме[1].

1.6.1. Характеристические свойства алгоритмов

***Алгоритм** — это точное, т. е. сформулированное на определённом языке, конечное описание того или иного общего метода, основанного на применении исполнимых элементарных тактов обработки.*

Описание должно быть составлено настолько точно, чтобы было возможным его однозначное понимание. Такому пониманию не мешают, вообще говоря, ни орфографические ошибки, ни безобидные опечатки; один и тот же алгоритм может быть записан на немецком или английском языке, на каком-либо искусственном языке или даже на языке пиктограмм, как в примере (а). При этом неважно, описаны единичные такты обработки словесно или в виде формул. *Правда, речь действительно должна идти о м е т о д е*[2]; одного представления о желаемой цели здесь недостаточно.

Элементарные такты обработки чаще выполняются один за другим (***последовательно***), но иногда и одновременно (***параллельно***). Когда разрешено как последовательное, так и параллельное выполнение, говорят о ***совместной*** (***коллатеральной***) ситуации. Должны быть учтены и исключительные ситуации („При отказе нажать на кнопку возврата денег"). Наконец, описание должно быть конечным, иначе его передача исполнителю (человеку или машине) длилась бы бесконечно долго. Метод является общим, если его можно применить более чем в одном случае.

---

[1] Слово „алгоритм" произошло от имени средневекового математика аль-Хорезми (9-й в.) родом из Хорезма (расположенного сегодня на территории советской республики Узбекистан); аль-Хорезми жил во „дворце мудрости" багдадского калифа.

[2] Здесь под „методом" подразумевается „способ действий". — *Прим. ред.*

Приведенные выше примеры не вполне удовлетворяют всем требованиям, предъявляемым к алгоритму: в примере (а) мы имеем недостаточно общий метод, в примере (с) нет достаточной точности. (Когда следует прекратить принимать капли — когда пройдёт кашель или когда склянка опустеет?)

Другие примеры алгоритмов, предназначенных для человека, можно найти в руководствах по вязанию (для полных и худых) и в рецептах приготовления блюд (для маленьких и больших семей); между прочим, здесь возможны и одновременные такты обработки; примеры алгоритмов, предназначенных для машины, мы находим в случае стиральных и посудомоечных автоматов с „выбором программ". Далее, монтажные схемы понимаются радиолюбителями как алгоритмы для изготовления соответствующих приборов; наконец, имеются даже алгоритмы для взлома замков. В разговорной речи часто не делают различий между собственно алгоритмом и его исполнением.

Часто от алгоритма требуют, чтобы он обязательно заканчивался, т. е. содержал конечное число элементарных тактов. Такой алгоритм называется ***завершающимся***. Далее, исполнение алгоритма может не быть однозначно определённым. В таком случае говорят о ***недетерминистическом*** алгоритме. Однако многие интересные недетерминистические алгоритмы приводят тем не менее к однозначно определенному результату; тогда они называются ***детерминированными***[1]. Конечно, детерминистические алгоритмы все детерминированы.

Таким образом, детерминированные алгоритмы определяют отображения (функции) — каждому конкретному набору исходных данных соответствует вполне определённый результат. Разумеется, различные алгоритмы могут задавать одно и то же отображение, причём каждый из них может достигать результата своим собственным конструктивным путем. Недетерминированные алгоритмы определяют только соответствия („многозначные функции"), а их исполнение доставляет какой-нибудь результат (осуществляет ***выбор***) из некоторого множества возможных результатов.

1.6.2. Примеры алгоритмов

Несколько более „изысканными" с технической точки зрения, чем приведенные выше „бытовые" алгоритмы, являются следующие примеры, сформулированные весьма неформально (на „разговорном языке").

---

[1] Терминология, введённая здесь, не является общепринятой; обычно вместо „детерминистический" говорят „детерминированный", а вместо „детерминированный" — „однозначный". Мы решили следовать здесь авторской терминологии. — *Прим. перев.*

(a) Алгоритм сложения двух положительных десятичных чисел.

Этот алгоритм запечатлён в наших мозгах с начальной школы; обычно мы исполняем его наполовину бессознательно. Сложность алгоритма мы замечаем только тогда, когда пытаемся явно описать эту хорошо знакомую нам процедуру.

(b) Алгоритмы разложения натурального числа на простые множители.

Если в нашем распоряжении имеется достаточно длинная таблица простых чисел, то можно пытаться последовательно делить заданное число на 2, 3, 5, 7, ... — не разделится ли без остатка,— пока, наконец, не придём к 1.

Если же таблицы простых чисел в распоряжении нет, то можно также последовательно пытаться делить заданное число на натуральные числа 2, 3, 4, 5, 6, 7... до тех пор, пока не останется 1; при этом для каждого составного числа как делителя выполняемая попытка деления будет бесполезной.

Последовательность получающихся в конечном счёте делителей даст требуемое разложение на простые множители.

(c) Алгоритм вставки карточки в (упорядоченную) картотеку. (Предполагается, что в картотеке нет рейтера[1].)

В случае пустой картотеки (пустой ящик картотеки) вставка карточки тривиальна. В противном случае раскроем картотеку в произвольном месте и сравним открывшуюся карточку с вставляемой по рассматриваемому признаку („сортировка"). В соответствии с результатом этого сравнения будем действовать тем же самым способом, вставляя карточку соответственно в переднюю или заднюю часть картотеки. Процесс заканчивается, когда карточку нужно вставлять в пустое множество карт.

(d) Алгоритм сортировки (несортированной) картотеки.

Сортировка пустой или одноэлементной картотеки тривиальна. В противном случае стопка карт *произвольным* образом разбивается на две непустые части, каждая из частей независимо сортируется, а затем обе сортированные стопки „смешиваются" в одну сортированную картотеку.

Разумеется, для такого смешивания нужно в свою очередь задать алгоритм. А именно, если одна из двух стопок пуста, то нужно взять вторую. В противном случае сравнивают первые карточки стопок по признаку сортировки. Ту из карточек, которая должна идти перед другой или одного с ней ранга, вынимают, остаток стопки смешивают с другой стопкой и перед по-

---

[1] Зажимы для удобства отыскания картотечных карточек. — *Прим. изд. ред.*

лучившейся в результате смешивания стопкой кладется вынутая карточка.

Этот пример демонстрирует ***иерархическую*** структуру: алгоритм сортировки основан на алгоритме смешивания.

(e) Алгоритм вычисления значения дроби $(a+b)/(a-b)$.

Сначала вычисляем (используя алгоритмы сложения и вычитания) значения выражений $a+b$ и $a-b$ (всё равно, последовательно или одновременно, поскольку ситуация здесь совместная), а потом образуем частное от деления полученных результатов (используя алгоритм деления).

В случае общих формул обнаруживается как иерархическое строение, так и совместность.

(f) Алгоритм вычисления числа e (т. е. вычисления последовательности дробей — приближений для e).

Основание натуральных логарифмов e иррационально, поэтому его можно определить только с помощью бесконечной последовательности рациональных чисел, всё лучше приближающих e. По Ламберту (1766 г.) такую последовательность можно получить следующим образом.

Начиная с

$$A_0 = 1, \qquad A_1 = 2,$$
$$B_0 = 0, \qquad B_1 = 1,$$

последовательно вычислять

$$A_{i+1} = (4 \times i + 2) \times A_i + A_{i-1},$$
$$B_{i+1} = (4 \times i + 2) \times B_i + B_{i-1}$$

и образовать последовательность рациональных чисел $(A_i + B_i)/(A_i - B_i)$.

Здесь речь идёт о незавершающемся алгоритме для вычисления вычислимого вещественного числа, опирающемся на алгоритм из пункта (e). (Майхилл показал в 1953 г., что существуют и невычислимые вещественные числа.)

(g) Алгоритм, распознающий, можно ли получить последовательность знаков a из последовательности знаков b посредством вычёркивания некоторых знаков.

Если a — пустая последовательность знаков, то ответом будет »да«. В противном случае нужно посмотреть, не пуста ли последовательность b. Если это так, то ответом будет »нет«. Иначе нужно сравнить первый знак последовательности a с первым знаком последовательности b. Если они совпадают, то надо снова применить тот же алгоритм к остатку последовательности a и остатку последовательности b. В противном слу-

чае нужно снова применить тот же алгоритм к исходной последовательности a и остатку последовательности b.

Этот алгоритм выдаёт двузначный результат, »да« или »нет«, т. е. он является ***алгоритмом распознавания*** свойства „быть частью данной последовательности знаков". Заметим, что распознавание того, является ли a (связным) *подсловом* b, — вещь более сложная.

В случаях (c) и (d) речь идет о недетерминистических и, вообще говоря, недетерминированных алгоритмах. Все другие являются примерами детерминистических алгоритмов, причем все, кроме (f), — завершающиеся.

Читателю настоятельно рекомендуется самому „прокрутить" некоторые из этих алгоритмов для подходящих объектов.

1.6.3. Рекурсия и итерация

В примере (e) мы имеем наиболее простой случай: количество элементарных тактов обработки постоянно и не зависит от чисел a, b. Иначе обстоит дело в других примерах: в случае (a) это количество зависит от разрядности большего слагаемого, в случае (b) — от величины разлагаемого числа, в случаях (c) и (d) — от размера картотеки, а в случае (f) оно даже бесконечно. Хотя описание алгоритма конечно и постоянно, количество фактически выполняемых тактов — величина переменная; это оказывается возможным благодаря использованию приёма, сводящего общую задачу к „более простой" задаче того же класса. Этот трюк называют ***рекурсией***. В примерах (c) и (d) наличие рекурсии очевидно из самого словесного описания алгоритма. В примерах (a), (b) и (g) мы имеем специальный случай рекурсии — чисто ***повторительную*** рекурсию. При словесном описании её часто записывают, как например в случае (b), в форме ***итерации***: «Пока выполнено определённое условие, повторяй...». В примере (f) речь идёт о безусловном (не зависящем от выполнения какого-либо условия) повторении.

Рекурсия — это широко распространённый метод, понятный и без математической формализации, интуитивно близкий любому „человеку с улицы" [1]. Для рекурсии в её наиболее общей, неитеративной форме типична необходимость отсрочки некоторых действий; см., в частности, пример (d). По этой причине она вряд ли показана для забывчивых людей, но вполне пригодна для подходящим образом оборудованных машин. Повторение же — это в значительной степени наш повседневный опыт.

---

[1] Рекурсия кроется в идее „картины в картине" (рис. 40), а также рисующих друг друга рук (рис. 41); она же обнаруживается в стихах Рингельнаца „...а этот глист страдал глистами, что мучились глистами сами" [перевод Л. Макаровой. — *Ред.*]

Наряду с рекурсией и повторением в алгоритмах встречается также разбор возможных случаев; см. (g). Без разбора от-

Рис. 40. Рекурсия: картина в картине.

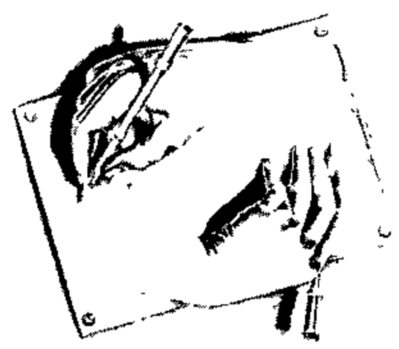


Рис. 41. Рекурсия: рисующие руки, литография М. Эшера, 1948 г. (© BEELDRECHT, Amsterdam/BILD-KUNST, Bonn 1982).

дельных случаев, в частности, было бы невозможно окончание рекурсивных алгоритмов; см. (c), (d), (g).

1.6.4. Специальные формы описания алгоритмов

Алгоритмы обрабатывают определённые объекты („входные") и выдают объекты в качестве результатов. Объекты могут быть конкретными, как, например, десятичное число в случае (a), или абстрактными, как, скажем, натуральные числа (для которых могут использоваться разнообразные эквивалентные системы представления) в случае (b) — для описанного там алгоритма совершенно несущественно, записываются числа в десятичной или двоичной системе счисления или даже римскими цифрами, свойства делимости от этого не меняются. В теоретических исследованиях предпочитают опираться на алгоритмы, которые работают, например, только с натуральными числами (Гёдель) либо только с цепочками знаков (Марков). С практической точки зрения нет никакой пользы или нужды в таких ограничениях, допустимы какие угодно множества объектов,

если только можно аккуратно определить их свойства. Разве лишь, поскольку приходится привлекать разбор отдельных случаев, необходимо включить в множество объектов по крайней мере значения истинности »истина« и »ложь«.

В зависимости от того, какие допускаются классы объектов (и соответствующих операций), приходят к различным классам алгоритмов.

1.6.4.1. Пример: алгоритмы Маркова

Исходя из сказанного в 1.5.1, мы можем констатировать, что операционные правила обработки дискретных сообщений являются алгоритмами над последовательностями знаков. Приступим теперь к уточнению понимавшегося до сих пор интуитивно понятия алгоритма. В частности, нам предстоит выяснить, что можно считать „элементарными тактами обработки".

Без сомнения, элементарной операцией над последовательностями знаков может считаться замена подслова на некоторое слово (***текстовая замена***). Будем исходить из множеств $\mathfrak{N}$ и $\mathfrak{N}'$ слов над общим набором знаков $\mathbb{V}$ (это можно делать без ограничения общности). Отдельную операцию замены (***продукцию***) будем записывать в виде $a \to b$ и понимать её так:

(*) „Если a является подсловом заданного слова x, то заменить это подслово на b. В случае если подслово a встречается в x несколько раз, словом b заменяется то из них, которое стоит в самой левой позиции."[1]

Далее, если дано (конечное) множество таких продукций, перечисленных в определённом порядке, то текстовая замена должна производиться посредством применения самой первой (относительно этого порядка) из применимых продукций. Всё это повторяется до тех пор, покуда возможно, или же до применения особым образом отмеченной продукции („останавливающей").

Такого рода алгоритмы называют ***алгоритмами Маркова*** по имени советского математика А. А. Маркова, который впервые описал их (в 1951 г.); сам Марков называл их „нормальными алгорифмами" Алгоритмы Маркова можно считать уточнением понятия алгоритма, достигаемым за счёт использования специальной формы описания.

В качестве примера приведем алгоритм Маркова, который по заданному двоичному слову строит производное двоичное

---

[1] Если учитывать при этом и пустое слово (см. 1.4.2), то текстовая замена включает в себя вставку и присоединение знаков, а также вычёркивание знаков. Поэтому мы не упоминаем специально эти особые случаи.

слово. Этот алгоритм использует вспомогательные знаки α, β („челноки") и содержит следующие продукции:

$$
\begin{array}{l}
\alpha O \to O\alpha \\
\alpha L \to L\beta \\
\beta O \to L\alpha \\
\beta L \to O\beta \\
\alpha \to \cdot \\
\beta \to \cdot \\
\to \alpha
\end{array}
$$

Останавливающие продукции отмечены точкой. На рис. 42 показано применение алгоритма к слову OOLLOLLLOLOL.

Благодаря компактным правилам замены алгоритмы Маркова представляют собой мощное средство описания: на сегодня нет таких алгоритмов над последовательностями знаков, для

OOLLOLLLOLOL
αOOLLOLLLOLOL
OαOLLOLLLOLOL
OOαLLOLLLOLOL
OOLβLOLLLOLOL
OOLOβOLLLOLOL
OOLOLαLLLOLOL
OOLOLLβLLOLOL
OOLOLLOβLOLOL
OOLOLLOOβOLOL
OOLOLLOOLαLOL
OOLOLLOOLLβOL
OOLOLLOOLLLαL
OOLOLLOOLLLLβ
OOLOLLOOLLLL

Рис. 42. „Челнок".

которых не существовало бы алгоритма Маркова, выполняющего ту же самую обработку сообщений. Описания, сформулированные в виде алгоритмов Маркова, часто оказываются весьма короткими по сравнению с другими, менее „рафинированными" формами описания[1]. Заметим, что элементарную для алгоритмов Маркова операцию замены (*) можно также рас-

[1] Намного более детальными оказываются, как правило, описания алгоритмов, которые должны выполняться на так называемых *машинах Тьюринга* (Тьюринг, 1936 г.). За подробностями по этим вопросам теоретического характера читатель может обратиться к литературе по теории алгоритмов (см., например, [18]).

сматривать как сложную, рекурсивно определяемую операцию над последовательностями знаков; подробнее об этом см. 3.5.7. Поэтому понятие элементарности всегда относительно.

В зависимости *от информации*, содержащейся в обрабатываемых сообщениях, часто, наоборот, рассматривают как элементарные такие такты обработки, которые, собственно говоря, являются сложными. Основанием для этого может служить, например, тот факт, что такие макротакты фактически состоят из конечного числа описанных выше текстовых замен, причём их внутренняя структура несущественна для предпринимаемой обработки. Так обстоит дело, скажем, для сложения, вычитания, умножения и деления с остатком, в случае когда сообщения представлены целыми числами в некоторой цифровой форме записи. В остальном же несущественно, представлена последовательность инструкций в виде диаграммы или как-нибудь более схематично, равно как несущественно и то, описаны отдельные такты обработки словесно или заданы посредством формул.

Собственно действия с числами в цифровой записи относятся, таким образом, к классу алгоритмов над цепочками знаков. Здесь в арифметике мы обнаруживаем исторические корни слова algorismo; еще Лейбниц говорил об „алгоритме умножения".

1.6.4.2. Рекурсивные алгоритмы по Маккарти

И всё-таки при описании алгоритмов, вообще говоря, нецелесообразно составлять правило обработки из простых шагов текстовой замены, как это делается в алгоритмах Маркова.

Мозаичность описания посредством обособленных операций текстовой замены без нужды затрудняет составление алгоритма, а также проверку того, реализует ли записанный алгоритм поставленную цель. При этом громоздким становится описание даже „школьного" выполнения арифметических действий над числами в цифровой форме записи. Использование вычислительной машины в качестве инструмента наводит пользователя на мысль рассматривать некоторые изначально сложные такты обработки как элементарные; хотя, впрочем, сведéние этих тактов обработки к текстовым заменам может происходить поразному на разных вычислительных машинах.

Другую форму описания алгоритмов, обладающую вдобавок тем преимуществом, что её принципиальные элементы не ограничиваются явными действиями над последовательностями знаков, мы положим в основу в следующей главе. Эти алгоритмы — мы будем называть их ***подпрограммами*** — определяют отображения посредством формальных композиций (последо-

вательное выполнение отображений) и разбора случаев; свою выразительную силу они получают благодаря употреблению рекурсии [1].

В качестве базы для построения будут использованы объекты („данные") и (вычислительные) операции над объектами, которые могут быть описаны и ***абстрактно***, т. е. *исключительно посредством указания их свойств* („аксиоматически"). Такие совокупности, состоящие из множеств объектов и соответствующих им (вычислительных) операций, будут называться ***примитивными (вычислительными) структурами***. Образцом здесь могут служить целые числа с основными арифметическими операциями над ними. В следующей главе мы приведём ряд таких вычислительных структур, от которых мы будем отправляться в первую очередь. Среди них имеется и вычислительная структура последовательностей знаков; класс формулируемых над нею рекурсивных подпрограмм обеспечивает столько же возможностей, сколько и класс алгоритмов Маркова.

---

[1] Этот класс, изученный впервые Маккарти в 1962 г., родствен классу μ-рекурсивных функций логики, представляющему собой классическое уточнение понятия алгоритма.

Глава 2

Основные понятия программирования

Из упомянутых в конце предыдущей главы форм описания алгоритмов ближе всего подходят к условиям реально используемых языков программирования правила обработки („программы", „подпрограммы"), которые построены на разборе случаев, композиции и рекурсии, выполняемых над некоторыми простыми, заранее заданными вычислительными структурами.[1]

Если при обработке так называемой числовой информации к сравнительно простым объектам типа чисел применяются относительно сложные операции, то при обработке так называемой нечисловой информации, напротив, обычно имеют дело с относительно сложно структурированными объектами, над которыми выполняются более простые операции. Таким образом, мы сталкиваемся как с проблематикой структуры операций, так и с проблематикой структуры объектов.

При конструировании ***алгоритмического языка*** следует исходить из некоторого набора операций, которые считаются элементарными и типичными для объектов определенного „вида". Кроме того, нужно позаботиться о том, чтобы можно было вводить как составные операции, так и составные (структурированные) объекты (см. гл. 6). При этом важно выбрать форму, удобную как для человека, который составляет правило обработки, так и для человека, который должен будет читать и понимать это правило, — форму, соответствующую кругу человеческих понятий и представлений.

В соответствии с принципом экономии мышления и учебно-методическими принципами следует использовать возможно более универсальные структуры и операционные понятия

---

[1] Используемые в этой книге основные понятия и обозначения в значительной степени опираются на алгоритмические языки *алгол-68* и *паскаль*; учтены также некоторые черты языка лисп. [По поводу названия второго из этих языков см. следующее подстрочное примечание; ALGOL — сокращение от ALGOrithmic Language (*англ.*: algorithmic — *алгоритмический*, language — *язык*); LISP — сокращение от LISt Processing language (list — список, process — обрабатывать). — *Изд. ред.*]

в возможно меньшем числе. В своём взаимодействии они составляют смысловую часть, ***семантику*** алгоритмического языка.

Кроме того, алгоритмический язык должен удовлетворять определённым внешним законам формы. В совокупности эти законы составляют ***синтаксис*** алгоритмического языка.

В дальнейшем в этой главе мы будем считать, что первична семантика, а синтаксис лишь вторичен. В частности, не следует обращать особое внимание на различия между обозначениями алгола и паскаля[1]. Поскольку мы занимаемся алгоритмическим языком, семантика всегда будет операционной: даже при описании структурированных объектов, с которыми производятся манипуляции, на первом плане стоят порождение и изменение таких структур, а не какие-либо „статические" свойства объектов, как в чистой математике. Проблема сведения операций, рассматриваемых в данном языке как элементарные, к отдельным шагам текстовой замены — и тем самым указание смысла заданных элементарных вычислительных структур — не относится к семантике языка. Этой проблемой занимается ***прагматика***. Мы ещё вернемся к этому в гл. 8.

Этот вводный раздел следует закончить замечанием, полное значение которого станет понятным лишь позже. Всякая алгоритмическая формулировка операционной обработки сообщений, записывается при помощи конечного числа знаков, так что сама является сообщением. (Информация, содержащаяся в этом сообщении, — это алгоритм; отображение интерпретации α (см. 1.1 и 1.5.2) определяется семантическим и прагматическим содержанием синтаксических формулировок.) И такое сообщение снова может быть объектом некоторой обработки сообщений. Это отражается прежде всего в текстовых преобразованиях, которым подвергается описание алгоритма в ходе его выполнения.

Существование сообщений, означающих обработку сообщений, — корни этого факта глубоко прослеживаются в философии, — наиболее замечательный феномен информатики. Он был обнаружен в 1951 г. Х. Рутисхаузером (при разработке способов трансляции алгоритмических языков на вычислительные устройства коммерческого типа), который исходил из основополагающих идей фон Неймана; Маккарти в 1960 г. впервые описал обработку некоторого языка программирования (лиспа) с помощью того же самого языка программирования.

---

[1] Этот язык назван по имени французского философа, математика и конструктора первого арифмометра Блеза Паскаля (1623—1662).

2.1. Основные вычислительные структуры

2.1.1. Объекты

„Die Gegenstände kann ich nur nennen. Zeichen vertreten sie. Ich kann nur von ihnen sprechen, sie aussprechen kann ich nicht.“ [1]

Людвиг Витгенштейн

Сообщение N вместе с сопоставленной ему информацией I в дальнейшем будет называться ***объектом*** (используется также термин „све́дение“ (Angabe)[2], а во множественном числе — ***данные***). Примером могут служить сообщения (записываемые арабскими цифрами в позиционной системе счисления) и связанная с ними информация, которую называют „натуральными числами“, а также символы (см. 1.4.4).

Итак, объект есть пара (N, I) с $N \overset{\alpha}{\longmapsto} I$, при этом информацию I называют ***значением*** объекта, а сообщение N — ***обозначением*** объекта. Говорят, что обозначение N *обладает* значением I при интерпретации α.

Например, обозначение 7 обладает значением »семь«, обозначение 007 — значением »семь«, обозначение 3.14 — значением »три целых и четырнадцать сотых«. При этом обозначение определяет значение, которым оно обладает, однозначно. Поэтому для краткости говорят просто „объект x“ вместо „объект с обозначением x“.

Различные обозначения могут обладать одним и тем же значением — отображение α обычно не является обратимым.

2.1.1.1. Сорта объектов

Объекты в алгоритмах играют роль предметов, над которыми производятся определённые операции. На практике классы объектов часто выделяются благодаря тому, что на них определен некоторый естественный процесс обработки сообщений и информации. Над объектами „натуральные числа“ как операндами[3] определены одноместная операция „переход к следующему натуральному числу“ и двуместные операции „сложение“ и „умножение“. В качестве результата эти операции

---

[1] „Предметы я могу лишь называть. Их представляют знаки. Я могу лишь говорить *о них*, выговорить их я не могу“. (*нем.*). — *Прим. перев.*

[2] К. Цузе (1944 г.), в одной работе, написанной в порядке подготовки к построению „исчисления планов“.

[3] То есть тем, над чем производятся операции. — *Прим. перев.*

вырабатывают натуральное число. Обратной к операции сложения натуральных чисел служит определённая лишь частично двуместная операция „вычитание". Кроме того, имеется двуместная операция „деление с остатком", которая в качестве результата вырабатывает два натуральных числа — частное и остаток.

Не всякий объект годится как операнд для той или иной операции. Множество объектов, для которых естественным образом определено некоторое количество операций, называется множеством объектов определённого ***сорта*** (или „вида" (алгол), „типа" (паскаль)). Таким образом, сорт объектов характеризуется операциями, которые могут над ними выполняться. Прежде всего следует сказать о самых распространённых — числовых объектах, множествах целых, рациональных, вещественных (машинных) и комплексных (машинных) чисел. Существуют и более сложные математические объекты — пространства и многообразия в геометрии, выражения в алгебре, клеточные комплексы в топологии, — на которых определены сложные операции. Если не ограничиваться областью чистой математики, то можно указать такой класс объектов, как символы (см. 1.4.4), обозначаемые словами над некоторым алфавитом или набором знаков.

Далее, объектами являются значения истинности, а именно »истина« и »ложь«, обозначаемые через T и F соответственно[1].

Универсально определены (двуместные) операции сравнения на „равенство" с результатом »истина« или »ложь«. Для чисел, равно как и для слов над некоторым алфавитом, имеется (двуместная) операция сравнения на „предшествование" в смысле естественного или лексикографического порядка, также с результатом »истина« или »ложь«. Таким образом, значения истинности (»истина«, »ложь«) суть ***универсальные объекты.***

Количество объектов данного сорта может быть бесконечным, однако рассматриваемое множество объектов должно быть счётным. Более того: чтобы объект мог служить операндом в некотором алгоритме, он должен быть представим конечным числом знаков из какого-нибудь исходного набора знаков. А именно требуется, чтобы каждый объект порождался за конечное число шагов („принцип порождения") из конечного числа базовых объектов („выделенные элементы"). Множество объектов должно быть „перечислимым" посредством некоторого алгоритма.

Для наиболее часто употребляемых в дальнейшем сортов (табл. 4) мы используем в качестве стандартных сокращений

---

[1] От английских true (истинный) и false (ложный). — *Прим. изд. ред.*

Таблица 4

Часто используемые сорта и их индикаторы

Запись на алголе-68	Запись на паскале	Множество объектов
int (**цел**)[a]	*integer*	алфавит целых чисел (см. 2.1.3.1)
real (**вещ**)	*real*	алфавит машинных вещественных чисел[b] (см. 2.1.3.3)
bool (**лог**)	*Boolean*	алфавит значений истинности »истина«, »ложь« (см. 2.1.3.6)
char (**лит**)	*char*	алфавит символов, которые представляются отдельными знаками („литерами")[c] (см. 2.1.3.4)
string (**строк**)	*string*[d]	алфавит символов, которые представляются конечными последовательностями знаков (словами) над некоторым набором знаков (см. 2.1.3.4)
bit[e] (**призн**)	*bit*[d]	алфавит символов, задаваемых с помощью двоичных знаков O L (см. 2.1.3.4)
bits (**бит**)	*bitstring*[d]	алфавит символов, определяемых с помощью двоичных слов[f] (см. 2.1.3.4)
lisp[g] (**лисп**)	*lisp*[g]	множество символов, задаваемых с помощью двоичных списков (см. 2.1.3.5), т. е. с помощью бинарных деревьев с размеченными листьями (см. 1.4.3)

[a] В скобках приводится русскоязычный вариант. — *Прим. изд. ред.*

[b] Приближённое представление вещественных чисел конечным числом разрядов; см. также А 4.

[c] В *стандартном* (или *ортодоксальном*) [т. е. описанном в соответствующем официальном документе. — *Перев.*] паскале это набор знаков 7-разрядного кода *ISO* (рис. 30).

[d] В паскале, собственно говоря, не задан, однако определим средствами языка.

[e] В стандартном алголе-68 не предусмотрен, заменяется на **bits** с длиной слова 1.

[f] В стандартном алголе-68 ограничиваются словами (какой-либо) фиксированной длины.

[g] В лиспе это типичный сорт, отсюда наше обозначение. В алголе и паскале стандартным образом не предусмотрен, однако определяется средствами языка.

словарные символы (см. табл. 4)[1]. Они называются ***индикаторами***.

Объекты сорта **string**, **bits**, **lisp** (соотв. string, bitstring, lisp) представляют первые примеры ***составных*** объектов. Подробнее об этом — в гл. 6. Прочие же объекты, введенные в настоящем

---

[1] В общеупотребительных языках программирования, в частности в алголе и в паскале, все словарные обозначения образованы от слов английского языка. В частности, в табл. 4 они образованы от слов: integer (целое число), real (вещественный), character (здесь: буква, знак), string (струна, ряд); по поводу bit и lisp см. соответственно конец раздела 1.4.1 и введение к этой главе; наконец, Boolean — это прилагательное, образованное от фамилии Boole (см. 2.1.3.6).

Во всех дальнейших пояснениях такого рода речь идёт о словах английского языка. — *Прим. изд. ред.*

разделе, являются ***простыми***. Но пока мы не будем обращать внимание на это различие.

Следует подчеркнуть, что, скажем, целые числа нельзя рассматривать как подмножество машинных вещественных чисел, а можно трактовать лишь как образ соответствующего подмножества при некотором взаимно-однозначном отображении. Относительно этого отображения см. 2.1.3.7.

2.1.1.2. Стандартные обозначения объектов

Для записи объектов вышеперечисленных сортов имеются устоявшиеся, так называемые ***стандартные обозначения***, или ***изображения***.

Изображения

34 1000 2 0 00123 123

обладают сопоставляемыми им обычным образом значениями целых чисел. Последние два изображения обладают одним и тем же значением.

Точно так же изображения[1]

$0.000123 \quad 1.23_{10}{-4} \quad 1_{10}{-4} \quad 1.23 \quad 0.123 \quad .123$

(см. также А.4) обладают сопоставляемыми им обычным образом значениями машинных вещественных чисел. Первые два изображения обладают равными значениями.

О позиционной системе счисления и представлении чисел, в частности о десятичной системе, см. приложение А.

Изображения

T, F

— соответственно в алголе[2]

true (истина), **false (ложь)**,

в паскале

true, *false*

— обладают значениями истинности »истина« и »ложь« (вида **bool (лог)**, соответственно типа Boolean).

---

[1] Использование десятичной точки (вместо запятой) является общепринятой международной нормой; $_{10}{-4}$ означает в обычном написании 10^{-4}, а $1.23_{10}{-4}$ означает 1.23×10^{-4}. Опускание десятичного основания в нижний индекс позволяет писать показатель на строке. В паскале (версия 1972 г.) вместо $_{10}$ пишут *E*, а запись .123 недопустима.

[2] Далее в скобках приводятся обозначения, используемые в русскоязычной версии алгола. — *Прим. перев.*

Изображения[1]

'a' 'f' 'α' 'ω' '♂' '♀' '␣'

— соответственно в алгольной записи

"a" "f" "α" "ω" "♂" "♀" "␣"

и в паскалевской

'a' 'f' 'α' 'ω' '♂' '♀' '␣'

— изображают определённые символы, которые обладают значениями (вида **char** (**лит**), соответственно типа *char*) »строчная буква А«, ..., »строчная буква омега«, »Марс«, »Венера«, »пробел«. Во избежание ошибок пробел должен иметь своё собственное изображение.

Изображения

"лихтенштейн" "FAZ" "ханс␣закс" "вторник"

— соответственно

'лихштенштейн' 'FAZ' 'ханс␣закс' 'вторник'

— также изображают символы, которые обладают значениями (вида **string** (**строк**), соответственно типа *string*), определяемыми при помощи последовательности значений отдельных знаков.

Для изображений

O, L

— а в алголе-68

"0", "1"

— значениями (вида **bit** (**призн**), соответственно типа *bit*) служат значения тех символов (над некоторым двоичным набором знаков), в качестве абстрактных представителей которых они выступают.

Изображения

OOLOL, LOLLO, LLOOL

— а в алголе-68

"2r00101", "2r10110"

---

[1] Стирание различия между ' и ' ухудшает читабельность.

— обладают в качестве значений (сорта **bits** (**бит**), соответственно *bitstring*) значениями тех символов или тех последовательностей символов, кодами которых служат соответствующие двоичные слова.

2.1.2. Операции

Операции, с которыми мы познакомимся в первую очередь, — это весьма простые и (из прагматических соображений) неразложимые в алгоритмическом языке правила обработки, которые изображаются ***символами операций*** (или ***операционными символами***). В соответствии с количеством операндов такие операции называются ***одноместными (унарными***[1]***)*** или ***двуместными (бинарными***[2]***)***[3].

При применении операций используют различные формы записи. Наиболее распространена ***функциональная запись***, при которой операция обозначается некоторым символом-буквой (или же символом-словом); для обозначения результата применения операции следом за её символом указываются операнды, заключенные в скобки и разделенные запятыми:

операция *fac* („факториал") с результатом применения *fac* (17);

операция *gcd*, или н. о. д. (наибольший общий делитель)[4] с результатом применения *gcd* (72, 30).

Однако к базовым операциям функциональную запись применяют неохотно, потому что появляются немыслимые горы скобок. Для двуместных операций в основном применяется ***инфиксная запись***. При этом символом операции служит, как правило, один знак. При записи результата применения операции он ставится между операндами. Мы выражаем это точками, располагаемыми до и после знака операции:

операция $.+.$ с результатом применения $17 + 4$;
операция .**mod**. с результатом применения 88 **mod** 17;
операции $\leqslant.$ с результатом применения $13 \leqslant 22$.

В последнем случае значением результата является значение истинности.

Для одноместных операций имеется соответствующая ***префиксная запись***, при которой знак операции стоит перед операндом. Мы выражаем это точкой, поставленной после знака операции:

операция —. с результатом применения —273.

---

[1] В оригинале monadisch. — *Прим. перев.*

[2] В оригинале dyadisch. — *Прим. перев.*

[3] Операции большей местности встречаются редко.

[4] По-английски greatest common divisor, отсюда сокращение gcd. — *Прим. изд. ред.*

Префиксная запись употребляется для многих элементарных функций математики:

sin., cos., ln. .

Аналогично применяется иногда ***постфиксная запись***, в частности для одноместных операций удвоения и деления пополам, счёта „вперёд“, возведения в квадрат и т. п.:[1]

операция .**div** 2 с результатом применения 12870 **div** 2;
операция .—1 с результатом применения 25 — 1;
операция .↑2 с результатом применения 5 ↑ 2 (алгольная запись).

Для многоместных операций префиксная и постфиксная записи представляют скорее теоретический интерес (*бесскобочная запись*, в частности *польская запись*, см. 3.7.3).

Некоторая разновидность функциональной записи возникает, когда символ операции опускается, а вид операции определяется формой скобок. В геометрии и физике такой способ используется для записи скалярного и векторного произведений:

$$(a, b), \ [f, g];$$

здесь символами операции служат скобки (и запятые). Еще пример:

операция образования пар ⟨. .⟩ с результатом применения ⟨f, g⟩.

Последняя операция встретится нам в 2.1.3.5 и в гл. 6.

Операции задают отображения; характеристики соответствующего отображения — его местность и сорта областей изменения аргументов и области значений — называют ***типом отображения*** или ***(функциональным) типом операции***[2]. Можно принять, что в приведённых выше примерах мы имеем следующие типы операций

fac: (**int**) **int**, соотв. (*integer*) : *integer*,

gcd, .+., .**mod**. } : (**int**, **int**) **int**:, соотв. (*integer*, *integer*) : *integer*,

.⩽. : (**int**, **int**) **bool**, соотв. (*integer*, *integer*) : *boolean*,

—., .**div** 2, .—1, .↑2 } : (**int**) **int**, соотв. (*integer*) : *integer*.

[1] Ниже **div** от английского division (деление). — *Прим. изд. ред.*
[2] В оригинале Funktionalität. - *Прим. изд ред.*

Как предельный случай можно рассматривать нульместные операции[1], выдающие постоянный результат, например 0 или 1, с функциональным типом

$$\left.\begin{matrix}0\\1\end{matrix}\right\}: \mathbf{int}:, \text{ соотв. } :integer.$$

Иногда встречаются также операции с „многомерным" результатом; примером могут служить частное и остаток для операции целочисленного деления, с функциональным типом

$$divmod:(\mathbf{int},\mathbf{int})\,\mathbf{int},\mathbf{int}:, \text{соотв.} (integer,\ integer):integer, integer.$$

Из-за сложностей с записью таких операций в большинстве языков программирования (в частности, и в алголе, и в паскале) этой возможности не предусматривается и приходится прибегать к окольным описаниям.

Функции (операции), которые имеют областью значений множество значений истинности »истина«, »ложь«, называются ***предикатами***.

2.1.3. Вычислительные структуры

"The introduction of suitable abstractions is our only mental aid to organize and master complexity." [2]

Э. Дейкстра

Вычислительная структура состоит из одного или нескольких множеств объектов, называемых сортами, и некоторых основных (элементарных, базовых) операций над этими сортами, каждая с результатом одного из этих сортов. Сюда же относят зачастую и предикаты, для которых результатом служит значение инстинности. При этом встречающиеся в вычислительной структуре ***выделенные элементы***, такие как нуль в множестве целых чисел, трактуются и включаются в структуру как нульместные операции. Совокупность сортов и операций называется ***сигнатурой*** вычислительной структуры. Операции, у которых операнд и результат — одного сорта, называют ***внутренними*** операциями (над этим сортом), прочие — ***смешанными***.

---

[1] Хотя простоты ради между ними часто не делают различия, объекты сорта **int** (*integer*) следует отличать от нульместных операций с функциональным типом **int**: (*:integer*).

[2] „Введение подходящих абстракций — это для нашей мысли единственный способ организовать сложное и управлять им." (*англ.*) — *Прим. перев.*

Рассматриваемые операции в большинстве случаев подчиняются определенным законам; например, многие двуместные операции ассоциативны. Это имеет то важное следствие, что в инфиксной записи можно опускать скобки. Многие двуместные операции коммутативны, т. е. не зависят от порядка операндов. Некоторые из одноместных операций инволютивны — повторное применение операции возвращает к исходному операнду, другие идемпотентны — повторное применение операции не даёт ничего нового [1].

С помощью достаточного числа законов можно однозначно охарактеризовать вычислительную структуру; её можно понимать как „чёрный ящик" (*англ.*: black box). Этим мы хотим сказать, что о „внутреннем устройстве" объектов и операций данной вычислительной структуры не надо ничего знать, что, более того, возможны даже различные реализации структуры, неразличимые по их отношению к „внешнему миру",— напомним, например, о различных возможностях (двоичного) кодирования натуральных или целых чисел и о многообразии возможных схем для выполнения арифметических операций.

Позже мы дадим примеры основных вычислительных структур для сортов, приведенных в табл. 4 (что одни из них могут быть сведены к другим, не должно нас сейчас интересовать). Мы будем пока обходиться этими ***примитивными*** (или ***базовыми***) вычислительными структурами при иллюстрации построения алгоритмов; общий вопрос о введении новых вычислительных структур будет обсуждён лишь в гл. 6.

2.1.3.1. Вычислительная структура Z целых чисел

Пожалуй, самая ходовая — она то и дело используется в повседневной жизни — это вычислительная структура Z ***целых чисел***, состоящая из сорта целых чисел и некоторого ряда производимых над ними ***арифметических*** операций. Что при этом относить к числу базовых операций, в большой мере произвольно; добавлять ли к четырём „основным арифметическим действиям" — сложению, вычитанию, умножению и делению (последняя операция является лишь частично определённой, см. ниже) — операции нахождения частного и остатка при целочисленном делении [2], минимума и максимума, наибольшего общего делителя и наименьшего общего кратного и включать ли в число базовых одноместные операции взятия абсолютного значения, возведения в квадрат, взятия знака или операцию изменения

---

[1] Сохраняется результат первого применения. — *Прим. перев.*

[2] *Частное* a **div** b и *остаток* a **mod** b [здесь **div** — от английского division (деление), **mod** — от латинского modulo (по модулю). — *Изд. ред.*] опреде-

знака на противоположный — это вопрос вкуса и целесообразности. (Даже от основных арифметических операций можно отказаться и опираться лишь на две одноместные операции перехода к „преемнику" и „предшественнику" (к следующему и предшествующему числам), а также на выделенный объект „нуль". Как отсюда могут быть выведены другие упомянутые выше операции, будет показано в ходе дальнейшего изложения на ряде примеров.) Наряду с внутренними операциями для целых чисел определены операции сравнения: предикаты *равно*, *не равно*, *меньше или равно*, *больше*, *меньше*, *больше или равно*, а также одноместные предикаты вроде *чётно*, *нечётно*.

Основные законы, такие как коммутативность и ассоциативность сложения и умножения (а также операций взятия минимума и максимума, н. о. д. и н. о. к.) и закон дистрибутивности, знакомы всем со школьной скамьи. Целые числа с операциями сложения и умножения удовлетворяют законам ***кольца***[1] („кольцо $\mathbb{Z}$ целых чисел").

Для вычитания и деления (при условии что последнее определено) выполняется закон ***правой коммутативности***:

$$(a-b)-c=(a-c)-b, \quad (a/b)/c=(a/c)/b.$$

Операция обращения знака инволютивна. Операция взятия абсолютного значения идемпотентна.

Обзор операций, рассматриваемых в дальнейшем как базовые, дан в табл. 5.

Стандартное обозначение целых чисел получается из десятичной записи с возможным применением операционного сим-

---

лены лишь для $b > 0$. А именно, в теории чисел они определяются соотношениями

$$(1)\ (a\,\mathbf{div}\,b)\times b + b + a\,\mathbf{mod}\,b = a \text{ и } (2)\ 0\leqslant a\,\mathbf{mod}\,b < b.$$

Из этих соотношений следует, что

$$(-a)\,\mathbf{mod}\,b + a\,\mathbf{mod}\,b = \begin{cases} 0, & \text{если } b\,|\,a, \\ b & \text{в противном случае} \end{cases}$$

и

$$-1 \leqslant (-a)\,\mathbf{div}\,b + (a\,\mathbf{div}\,b) \leqslant 0.$$

В паскале в качестве определяющего тоже берется соотношение (1), но на этот раз вместе с соотношениями

$$\begin{array}{ll} 0\leqslant a\,\mathbf{mod}\,b < b & \text{при } a > 0, \\ -b < a\,\mathbf{mod}\,b \leqslant 0 & \text{при } a < 0, \end{array}$$

в случае $a < 0$ отклоняющимися от (2). Принятое в стандартном [т. е., в данном случае, описанном в [05]. — *Перев.*] алголе определение, для которого даже (1) не имеет места, является непригодным.

[1] См., например, G. Birkhoff, T. Bartee, Angewandte Algebra, Oldenbourg, 1973 [имеется перевод с оригинального английского издания 1970 г.: Биркгоф Г., Барти Т. Прикладная алгебра. — М.: Мир, 1976. Впрочем, годится и любой другой учебник по „высшей" алгебре. — *Изд. ред.*]

Таблица 5

Обзор: вычислительная структура $\mathbb{Z}$ целых чисел

	Запись на алголе-68 [a]	Запись на паскале
Множество объектов	**int**	*integer*
Двуместные внутренние операции:		
тип операции	**(int, int) int:**	*(integer, integer) : integer*
сложение	.+.	.+.
вычитание	.−.	.−.
умножение	.×.	.*.
частное [b]	**.div.** .÷. [c]	.div.
остаток [b]	**.mod.**	.mod.
минимум $min(.,.)$	«стандартная форма не предусмотрена»	«стандартная форма не предусмотрена»
максимум $max(.,.)$		
н. о. д. $gcd(.,.)$		
н. о. к. $lcm(.,.)$		
степень	.↑.	«стандартная форма не предусмотрена»
Одноместные внутренние операции:		
тип операции	**(int) int:**	*(integer) : integer*
преемник $succ(.)$	.+1	*succ* (.) .+1
предшественник $pred(.)$	.−1	*pred* (.) .−1
удвоение	.×2	.*2
деление пополам	**.div** 2 .÷2	.div 2
абсолютное значение	**abs.**	*abs* (.)
знак	**sign.**	«стандартная форма не предусмотрена»
квадрат	.↑2	*sqr* (.)
перемена знака	−.	−.
Выделенные элементы:		
тип операции	**int:**	*:integer*
нуль	0	0
единица	1	1
двойка	2	2
Предикаты		
тип операции	**(int, int) bool:**	*(integer, integer) : Boolean*
равно	.=.	.=.
не равно	.≠.	.≠. .<>.
меньше или равно	.≦. .⩽.	.≦. .⩽. .<=.
больше	.>.	.>.
меньше	.<.	.<.
больше или равно	.≧. .⩾.	.≧. .⩾. .>=.

Таблица 5 (продолжение)

тип операции	(int) bool:	(*integer*) : *Boolean*
свойство быть равным нулю	.=0	.=0
свойство быть неравным нулю	.≠0	.≠0. < > 0
свойство быть чётным	«стандартная форма не предусмотрена»	«стандартная форма не предусмотрена»
свойство быть нечётным	odd.	*odd* (.)

[a] В русскоязычной версии алгола, div заменяется на дел, mod — на ост, abs — на абс, sign — на знак, odd — на нчт. — *Прим. перев.*

[b] Частично определенная операция.

[c] В алголе-68 .over. или .÷..

вола —. для отрицательных чисел. Разрешены „незначащие" нули в старших разрядах [1].

По техническим соображениям почти во все языки программирования вводятся ограничения на рассматриваемую область целых чисел. Однако на первых порах на это можно не обращать внимания.

Операции нахождения частного и остатка являются лишь частично определёнными — второй операнд должен быть отличным от нуля.

2.1.3.2. Вычислительная структура $\mathbb{N}$ натуральных чисел

'There was an old man who said "Do
Tell me how I'm to add two and two?
Arithmetical lore
claims they add up to four
But I fear that is almost too few."' [2]

Вычислительная структура $\mathbb{N}$ натуральных чисел может быть определена без ссылок на $\mathbb{Z}$, что и соответствует историческому развитию понятия числа. При этом можно положить в основу одну-единственную одноместную операцию перехода

[1] Если отказаться от представлений с начальными незначащими нулями, то целые положительные числа представимы однозначно.

[2] „Один старик сказал когда-то:
— Что будет — к двум прибавишь два ты?
Нам арифметика твердит:
— четыре, но душа болит —
боюсь, немного маловато!"

(*англ.*) Перевод Л. Макаровой. — *Прим. изд. ред.*

к следующему числу, а также выделенный объект „нуль“ (Дедекинд (1887 г.), Пеано (1889 г.))[1].

В языках программирования, напротив, принято вводить натуральные числа как подмножество целых, а именно как множество неотрицательных целых. Поэтому ни в алголе, ни в паскале для них как для сорта не предусмотрены стандартизованные изображения; соответствующее ограничение должно быть указано с помощью условий-предохранителей (см. 2.3.1.3). У частично определённых операций нахождения частного и остатка второй операнд по-прежнему должен быть отличным от нуля. Ввиду основного ограничения и некоторые другие операции из $\mathbb{Z}$ определены в $\mathbb{N}$ лишь частично, а именно вычитание и переход к предшествующему числу; операция обращения знака вообще не определена (кроме как для 0).

2.1.3.3. Вычислительные структуры для вычислений с рациональными, вещественными и комплексными числами

Для рациональных чисел наряду со сложением, вычитанием и умножением выполнимо и деление; оно совпадает со взятием частного, остаток же всегда тривиальным образом равен нулю. Понятия н. о. д. и н. о. к. теряют смысл, равно как и понятия последующего и предыдущего числа, а также свойства быть чётным или нечётным. Рациональные числа образуют ***поле***[2] („поле $\mathbb{Q}$ рациональных чисел“).

Рациональные числа можно использовать для приближения вещественных. На практике ограничиваются десятичными или двоичными дробями с некоторым заранее заданным максимальным числом значащих разрядов, т. е. без начальных нулей. Вследствие этого при выполнении арифметических операций, как правило, требуется проводить ***округление***[3]. Проблемами точности вычислений, которые встают при выполнении действий над машинными вещественными числами, занимается численная математика. Подробнее об этом см., например, (A1) (особенно гл. 1).

Часть введённых для целых чисел операций: двуместные операции $.+.$, $.-.$, $.\times.$ и одноместные операции **abs**, **sing**, $-.$, а также операции сравнения $.=.$, $.\neq.$, $.\leqslant.$, $.>.$, $.<.$, $.\geqslant.$ — с соответственно изменёнными типами операций, используется в

---

[1] В противоположность традиционному определению (см., например, van der Waerden, Algebra, 7. Aufl. Springer, 1966 [имеется перевод: ван дер Варден Б. Л. Алгебра. — М.: Наука, 1979. — *Изд. ред.*]), мы включаем нуль в множество натуральных чисел.

[2] См. предпоследнее подстрочное примечание в разделе 2.1.3.1.

[3] Поэтому в строгом алгебраическом смысле слова машинные рациональные числа не образуют поля.

дальнейшем и для машинных рациональных чисел. Операции **div.** и **mod.** вырождаются, их место заступает деление ./. . Эта операция по-прежнему является частично определённой, так как деление на нуль, как и ранее, не определено. Для машинных рациональных чисел, „близких к нулю", выполнение деления „ненадёжно": речь идёт о связанной с механизмом округления неустойчивости.

Таблица 6

Обзор: элементарные функции

	Запись на алголе-68	Запись на паскале	Примечания
тип функции	(**real**)**real**:	(*real*) : *real*	
квадрат	.↑2	*sqr*(.)	
абсолютное значение	**abs**.	*abs*(.)	
квадратный корень[a]	*sqrt*(.)	*sqrt*(.)	для отрицательных аргументов не определён
экспонента	*exp*(.)	*exp*(.)	
натуральный логарифм	*ln*(.)	*ln*(.)	для неположительных аргументов не определён
синус	*sin*(.)	*sin*(.)	
косинус	*cos*(.)	*cos*(.)	
тангенс	*tan*(.)	«стандартный вид не предусмотрен»	
арксинус	*arcsin*(.)	«стандартный вид не предусмотрен»	главное значение (из интервала [π/2, π/2])
арккосинус	*arccos*(.)	«стандартный вид не предусмотрен»	главное значение (из интервала [0, π])
арктангенс	*arctan*(.)	*arctan*	главное значение (из интервала [—π/2, π/2])

[a] По-английски square root. Отсюда обозначение. — *Прим. изд. ред.*

Так называемые элементарные функции чистой математики так часто используются в численной математике, что их чаще всего включают в качестве ***стандартных функций*** в вычислительную структуру машинных вещественных чисел. Сказанное выше резюмирует табл. 6[1].

Алгол-68 располагает также особой вычислительной структурой для проведения вычислений над машинными комплексными числами.

---

[1] В отличие от обычной математической практики в алголе-68 и паскале применяется функциональная запись *ln*(*a*). *sin*(*a*) вместо ln *a*, sin *a*.

2.1.3 4*. Вычислительные структуры для нечисловых вычислений: последовательности знаков

Вычислительные структуры для нечисловых вычислений должны разрешить в первую очередь работу со словами (т. е. с последовательностями знаков конечной длины) над заданным алфавитом V. Типичными операциями являются операции добавления нового знака к данному слову с того или другого конца, а также (частично определённые) обратные к ним операции. Выделенным элементом служит пустое слово.

К этим операциям может быть сведена операция „сцепления“ двух последовательностей знаков (***конкатенация***). Обратно, используя операцию ***обобщения*** — переход от знака к одноэлементному слову, которое состоит из этого знака,— в сочетании с конкатенацией, получаем операцию прибавления знака. В алголе-68 именно так и делается, причём операция обобщения остаётся никак не обозначенной. Однако обратных операций в алголе-68 нет; они доступны лишь через описания. В паскале стандартизованных операций для работы с последовательностями знаков нет, но они могут быть введены в специальных реализациях.

В дальнейшем мы будем проводить все рассмотрения для слов сорта **string** (соотв. *string*) над алфавитом **char** (соотв. *char*) (см. табл. 4). Для слов сорта **bits** (соотв. *bitstring*) над двоичным алфавитом **bit** (соотв. *bit*), состоящим из элементов O, L, всё обстоит аналогично (см. также табл. 11).

Обзор операций, которые в дальнейшем рассматриваются как базовые для вычислительной структуры (V^*, V) слов над заданным алфавитом V, дан в табл. 7.

Стандартные обозначения слов и знаков получаются заключением их в двойные кавычки-штрихи (алгол-68), соответственно в одинарные кавычки-штрихи (паскаль).

По техническим причинам почти во всех языках программирования вводится ограничение на максимальную длину слова. На это ограничение пока можно не обращать внимания.

Операции *rest*(.), *lead*(.), *first*(.), *last*(.) являются лишь частично определёнными; а именно, они определены лишь для непустого аргумента[1]. На пустоту „проверяет“ предикат $. = \diamond$ (соотв.[2] *isemty*(.))

Конкатенация является ассоциативной, но вообще говоря не коммутативной операцией. Множество слов с операцией кон-

---

* Изучение этого раздела можно отложить до 2.3.2.

[1] Для бесконечных последовательностей знаков (см. ниже) операции *lead* и *last* тоже не определены.

[2] Ниже *isempty* — от is empty (is — является, empty — пустой). Такой способ образования обозначений часто будет встречаться и в дальнейшем. — *Прим. изд. ред.*

Таблица 7

Обзор: вычислительная структура (𝕍*, 𝕍) знаковых последовательностей над алфавитом 𝕍

	Запись на алголе-68 [a, b]	Запись на паскале [c]
Сорта:		
знаки (𝕍)	**char**	*char*
знаковые последовательности (𝕍*)	**string**	*string*
Двуместные внутренние операции:		
тип операции	**(string,string)string**:	(*string*, *string*) : *string*
конкатенация	.+.	*conc*(. , .)
Одноместные внутренние операции:		
тип операции	**(string)string**:	(*string*) : *string*
„все, кроме первого“[d]	*rest*(.)	*rest*(.)
„все, кроме последнего“[d]	*lead*(.)	*lead*(.)
Двуместные смешанные операции:		
тип операции	**(char, string) string**;	(*char*, *string*): *string*
приставить впереди	⟨.⟩+. или .+.	*prefix*(. , .)
тип операции	**(string, char)string**;	(*string*, *char*): *string*
приставить сзади	.+⟨.⟩ или .+.	*postfix* (.,.)
Одноместные смешанные операции:		
тип операции	**(string)char:**	(*string*) : *char*
„первый“[d]	*first*(.)	*first*(.)
„последний“[d]	*last*(.)	*last*(.)
тип операции	**(char)string**:	(*char*) : *string*
обобщение	⟨.⟩ или .	*prefix* (. , *empty*) или ⟨.⟩
Выделенный элемент:		
тип операции	**string:**	: *string*
пустая последовательность знаков	◇ или ""	*empty* или ''
Предикаты:		
тип операции	**(string, string)bool**;	(*string*, *string*): *Boolean*
равно	.=.	
не равно	.≠.	
лексикографически: меньше или равно	.≦. .⩽.	
лексикографически: больше	.>.	
лексикографически: меньше	.<.	
лексикографически: больше или равно	.≧. .⩾.	
тип операции	**(string)bool**;	(*string*) : *Boolean*
свойство быть пустым	=⟨⟩ или .=""	*isempty* (.)

[a] Примечание. Операция обобщения ⟨.⟩ в стандартном алголе-68 остаётся никак не обозначаемой. Для пустой последовательности знаков необходимо использовать стандартное изображение "". Операции *rest*, *lead*, *first*, *last* стандартным образом не предусмотрены. Но могут быть введены средствами алгола-68, например

first (*a*) как *a* [1], *rest* (*a*) как *a* [2: **upb** *a*],

last (*a*) как *a* [**upb** *a*], *lead* (*a*) как *a* [1: **upb** *a* − 1].

[b] В русскоязычной версии алгола-68 *rest* заменяется на *хвост*, *lead* — на *вед*, *first* — на *перв*, *last* — на *посл*, **upb** — на **вегр**. — *Прим. перев.*

[c] Примечание. Тип *string* в паскале стандартным образом не предусмотрен.

[d] Частично определённая операция.

катенации и пустым словом в качестве выделенного элемента удовлетворяет аксиомам полугруппы с единицей („полугруппа V^* слов").

Другие операции, определенные на словах, характеризуются следующими законами (на паскале):

$$first(prefix(x, a)) = x,$$
$$rest(prefix(x, a)) = a,$$

а также (для непустых a)

$$prefix(first(a), rest(a)) = a$$

и их аналогами для случая другого конца слова:

$$last(postfix(a, x)) = x,$$
$$lead(postfix(a, x)) = a$$

и (для непустых a)

$$postfix(lead(a), last(a)) = a.$$

При инфиксной записи конкатенации для алгола-68 мы имеем также (в случае непустых a)

$$\langle first(a)\rangle + rest(a) = a,$$
$$lead(a) + \langle last(a)\rangle = a.$$

Дальнейшие связи этих операций с конкатенацией состоят в том, что при $a \neq \Diamond$ или $b \neq \Diamond$

$$first(a + b) = \begin{cases} first(a) & \text{при} \quad a \neq \Diamond, \\ first(b) & \text{при} \quad a = \Diamond \end{cases}$$

и

$$rest(a + b) = \begin{cases} rest(a) + b & \text{при} \quad a \neq \Diamond, \\ rest(b) & \text{при} \quad a = \Diamond \end{cases}$$

и справедливы соответствующие соотношения для случая другого конца слова.

Обратим внимание, что множество слов (т. е. конечных последовательностей знаков) над данным конечным алфавитом является счётным (и даже перечислимым с помощью некоторого алгоритма). Это, впрочем, справедливо и для множества всех слов над счётным алфавитом, например для множества всех конечных последовательностей натуральных чисел (Гёдель, 1928 г.). Напротив, множество всех бесконечных последовательностей знаков — даже над *бинарным* алфавитом — уже не является счётным. Действительно, названное множество можно взаимно-однозначно сопоставить множеству всех правильных бесконечных десятичных дробей, это последнее соответствует множеству всех вещественных чисел из замкнутого интервала [0..1], а Кантор ещё в 1873 г. показал, что множество точек интервала [0..1] не является счётным. Доказательство Кантора („диагональный метод") прямо переносится на случай общих бесконечных последовательностей знаков.

Поэтому информатика должна ограничиваться рассмотрением подмножества вычислимых бесконечных последовательностей знаков (соответственно вычислимых вещественных чисел, к каковым относятся, скажем, $\sqrt{2}$, e, π). Сначала мы будем обходиться даже конечными последовательностями знаков.

2.1.3.5 *. Вычислительные структуры для нечисловых вычислений: размеченные бинарные деревья

В слове знаки расположены последовательно („линейно"). В кодовом же дереве (см. рис. 37, 38) мы находим знаки как листья на дереве. Если разветвление всегда происходит на две ветви и к тому же важен порядок ветвей, то речь идёт о ***бинарном*** (***упорядоченном***) ***дереве***[1]. Вычислительной структурой, представляющей значительный исторический, теоретический и практический интерес, является структура бинарных упорядоченных ***деревьев с размеченными*** (знаками из $\mathbb{V}$) ***листьями***. Такие деревья называют ещё ***облиственными*** или ***деревьями с помеченными листьями***. В качестве предельного случая сюда относят также состоящие из одного знака ***атомарные деревья***. Основные операции — это ***соединение***[2,3] двух деревьев в одно (обозначаемое через *cons* (., .)[4,5]), а также (частично определённые) обращения этой операции — взятие *левого поддерева* и *правого поддерева* (обозначаемые через *car*(.) и *cdr*(.)).

Операция ***обобщения***, т. е. перехода от знака к атомарному дереву, чаще всего остаётся необозначенной. Наконец, нужен ещё предикат *isatom*(.), чтобы различать атомарные и „настоящие" деревья.

Бинарные деревья с помеченными листьями служат графическими представлениями элементов абстрактной вычислительной структуры ***двоичных*** (или ***бинарных***) ***списков***. Они рекурсивно определяются следующим образом:

Двоичный список — это либо ***атомарный*** двоичный список, т. е. знак, либо ***неатомарный*** двоичный список, т. е. (упорядоченная) пара двоичных списков.

---

* Изучение этого раздела можно отложить до 2.4.1.6.

[1] Точнее было бы сказать — о *диадическом* дереве (или *диадическом* списке). В английском одно и то же слово binary употребляется там, где немцы используют три разных: binär, dual и dyadisch. [В русском более или менее на равных употребляются два слова — *бинарный* и *двоичный*. — *Изд. ред.*] Под ***деревом*** мы всюду понимаем *ориентированное дерево с корнем*.

[2] В оригинале Kombination, что и объясняет следующее подстрочное примечание. — *Прим. изд. ред.*

[3] От латинского combinare — соединять по два, сдваивать.

[4] От английского constructor (конструктор).

[5] Это обозначение сложилось исторически, равно как и появляющиеся далее обозначения car и cdr (Маккарти, 1959 г.).

Если использовать для операции соединения списков обозначение $\langle..\rangle$, то мы получим стандартное представление двоичных списков в виде

$$a, \langle ab\rangle, \langle\langle ab\rangle c\rangle, \langle a\langle bc\rangle\rangle,$$

где a, b, c — либо знаки, либо бинарные списки. В частности, мы имеем такие двоичные списки, как

$$'A', \langle 'A'\ 'B'\rangle, \langle\langle 'A'\ 'B'\rangle\ 'C'\rangle, \langle 'A'\ \langle 'B'\ 'C'\rangle\rangle,$$
$$\langle\langle 'A'\ 'B'\rangle\langle 'B'\ 'C'\rangle\rangle.$$

Списки с „сосредоточенными" слева или справа скобками вроде

$$\langle\langle 'A'\ 'B'\rangle\ 'C'\rangle$$

или

$$\langle 'A'\langle 'B'\langle 'C'\langle 'D'\langle 'E'\rangle\rangle\rangle\rangle\rangle$$

называют (лево- или право-) ***линейными*** списками.

Таблица 8

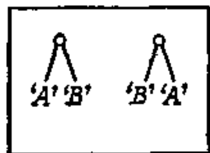
Обзор: Вычислительная структура (V^λ, V) двоичных (знаковых) списков (упорядоченных бинарных деревьев с размеченными листьями) над алфавитом V [a]

	Запись на алголе-68	Запись на паскале
Сорта:		
знаки (V)	**char**	*char*
двоичные списки (V^λ)	**lisp**	*lisp*
Двуместные внутренние операции:		
тип операции	**(lisp, lisp)lisp :**	(*lisp*, *lisp*) : *lisp*
конструктор $\langle..\rangle$	*cons* (. , .)	*cons* (.,.)
Одноместные внутренние операции:		
тип операции	**(lisp)lisp:**	(*lisp*) : *lisp*
„левое поддерево" [b]	*car* (.)	*car* (.)
„правое поддерево" [b]	*cdr* (.)	*cdr* (.)
Одноместные смешанные операции:		
тип операции	**(char)lisp:**	(*char*) : *lisp*
обобщение	*mkatom* (.)	*mkatom* (.)
тип операции	**(lisp)char:**	(*lisp*) : *char*
низведение [b]	*val* (.)	*val* (.)
Предикаты:		
тип операции	**(lisp, lisp)bool:**	(*lisp*, *lisp*) : *Boolean*
равно	.=.	
не равно	.≠.	
тип операции	**(lisp)bool:**	(*lisp*) : *Boolean*
свойство быть атомарным	*isatom* (.)	*isatom* (.)

[a] Приводимые ниже операции не предусмотрены стандартным образом ни в стандартном алголе-68, ни в паскале, однако определимы средствами этих языков.

[b] Частично определённая операция.

Двоичные размеченные деревья являются типичными объектами языка программирования лисп (Маккарти, 1959 г.). Ни в алголе, ни в паскале они стандартным образом не предусмотрены, однако могут быть определены с помощью имеющихся языковых средств. В табл. 8 дан обзор операций, которые в



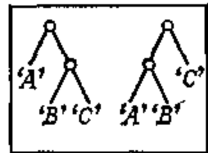


Рис. 43. Бинарные деревья.

дальнейшем используются в качестве базовых операций вычислительной структуры ($\mathbb{V}^{\lambda}$, $\mathbb{V}$) двоичных размеченных деревьев $\mathbb{V}^{\lambda}$ над алфавитом $\mathbb{V}$.

В алголе и паскале стандартных обозначений для списков нет, их надо строить явно с помощью операции *cons*.

Операции *car*(.) и *cdr*(.) определены лишь частично, а именно лишь для неатомарного аргумента.

На атомарность проверяет предикат *isatom*(.). Операция *val*(.) определена лишь на атомарных деревьях.

Операция соединения *cons* ни коммутативна, ни ассоциативна. Действительно, как спискам

$$\langle 'A'\ 'B' \rangle \text{ и } \langle 'B'\ 'A' \rangle,$$

так и спискам

$$\langle 'A' \langle 'B'\ 'C' \rangle \rangle \text{ и } \langle \langle 'A'\ 'B' \rangle 'C' \rangle$$

отвечают два различных дерева (рис. 43). Вообще числу различных способов расстановки парных скобок для данной последовательности знаков соответствует такое же число различных деревьев.

Размеченные бинарные деревья с операцией соединения образуют лишь ***группоид***[1] („группоид размеченных деревьев“).

Упомянутые выше операции над двоичными списками (соотв. над бинарными размеченными деревьями) характеризуются следующими свойствами:

$$car\,(cons(a, b)) = a,$$
$$cdr(cons(a, b)) = b,$$

---

[1] Группоид — это алгебра с одной бинарной операцией, без каких-либо законов (см., например, указанную выше книгу Биркгофа и Барти).

а также (для неатомарных a)

$$cons(car(a), cdr(a)) = a.$$

Кроме того,

$$val(mkatom(x)) = x$$

и (для атомарных a)

$$mkatom(val(a)) = a,$$

а также

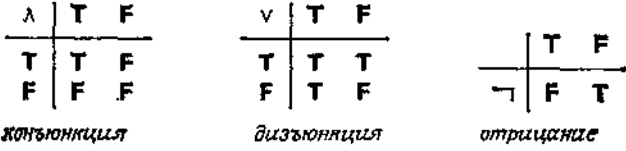
$$isatom(mkatom(x))$$

и

$$\neg\, isatom(cons(a, b)).$$

2.1.3.6 *. Вычислительная структура $\mathbb{B}_2$ значений истинности

При построении алгоритмического языка по Маккарти существен *разбор случаев*, и потому необходимо иметь возможность формально выражать выполнение или невыполнение тех или



∧	T	F
T	T	F
F	F	F

конъюнкция

∨	T	F
T	T	T
F	T	F

дизъюнкция

	T	F
¬	F	T

отрицание

Рис. 44. Таблицы значений для базовых операций в $\mathbb{B}_2$

иных условий. Таким образом, должен быть универсально введена некоторая вычислительная структура значений истинности {T, F}.

В качестве базовых операций используют обычно две двуместные операции ***конъюнкции*** и ***дизъюнкции***, которые в обычном языке выражаются союзами „и" и „или" (лат.: *vel*), и одноместную операцию ***отрицания***, которая в обычном языке выражается частицей „не". Так как множество объектов здесь конечно, то эти операции можно определить, задав для них таблицы значений (рис. 44).

И здесь можно обойтись меньшим числом базовых операций, что, однако, представляет лишь теоретический интерес (см. всё же операции NAND и NOR из 4.1.1.3).

Конъюнкция и дизъюнкция ассоциативны и коммутативны, отрицание инволютивно. Эти и другие законы сведены в табл. 9. Это законы так называемой ***булевой алгебры***[1]; вычислительная

---

* Этот раздел можно изучать, не прочитав 2.1.3.1.

[1] По имени Джорджа Буля (1815—1864), английского математика и логика.

Таблица 9

Законы булевой алгебры

$\neg(\neg f)=f$		(закон инволюции)
$f\wedge g=g\wedge f$	$f\vee g=g\vee f$	(законы коммутативности)
$(f\wedge g)\wedge h=f\wedge(g\wedge h)$	$(f\vee g)\vee h=f\vee(g\vee h)$	(закон ассоциативности)
$f\wedge f=f$	$f\vee f=f$	(законы идемпотентности)
$f\wedge(f\vee g)=f$	$f\vee(f\wedge g)=f$	(законы поглощения)
$f\wedge(g\vee h)=(f\wedge g)\vee(f\wedge h)$	$f\vee(g\wedge h)=(f\vee g)\wedge(f\vee h)$	(законы дистрибутивности)
$\neg(f\wedge g)=\neg f\vee\neg g$	$\neg(f\vee g)=\neg f\wedge\neg g$	(законы де Моргана)
$f\wedge(g\vee\neg g)=f$	$f\vee(g\wedge\neg g)=f$	(законы нейтральности)

структура $\mathbb{B}_2$ является двухэлементной (и с точностью до изоморфизма единственной двухэлементной) моделью булевой алгебры[1].

С помощью соответствия

$$\mathsf{F} \mathrel{\hat{=}} \mathsf{O},\ \mathsf{T} \mathrel{\hat{=}} \mathsf{L} \qquad (1)$$

Таблица 10

Обзор: вычислительная структура $\mathbb{B}_2$ значений истинности

	Запись на алголе-68[a]	Запись на паскале
Множество объектов	**bool**	*Boolean*
Двуместные внутренние операции:		
тип операции	(**bool**, **bool**)**bool** :	(*Boolean*, *Boolean*): *Boolean*
конъюнкция	$.\wedge.$	**.and.** $.\wedge.$
дизъюнкция	$.\vee.$	**.or.** $.\vee.$
Одноместные внутренние операции:		
тип операции	(**bool**)**bool**:	(*Boolean*) : *Boolean*
отрицание	$\neg.$	**not.** $\neg.$
Выделенные элементы:		
тип операции	**bool**:	: *Boolean*
T («истина»)	**true**[a]	*true*
F («ложь»)	**false**[a]	*false*
Предикаты:		
тип операции	(**bool**, **bool**)**bool**:	(*Boolean*, *Boolean*): *Boolean*
равно	$.=.$	$.=.$
не равно	$.\neq.$	$.\neq.$

[a] В русскоязычном варианте алгола-68 **true** заменяется на **истина**, **false** — на **ложь**. — *Прим. перев.*

---

[1] См. „Прикладную алгебру" Биркгофа и Барти.

операции отрицания, конъюнкции и дизъюнкции переносятся также на сорт **bit**; при этом мы получаем изоморфную вычислительную структуру BIT.

В табл. 10 дан обзор операций, рассматриваемых в дальнейшем как универсальные („булевы") операции логики высказываний.

2.1.3.7. Переходы между сортами

Наряду с общим применением некоторых символов операций для внутренних операций над целыми и вещественными числами алгол и паскаль допускают также и смешанные операнды Это достигается отображением кольца целых чисел в изоморфное подкольцо вещественных чисел, которое (отображение) остаётся н е о б о з н а ч а е м ы м. Такую вспомогательную операцию, которая переводит объект из одного сорта в другой, называют ***неявным обобщением***[1].

Далее, вводятся операции с функциональным типом

(real) int:, соотв. (*real*):*integer*,

которые переводят вещественные числа в целые, например (определения см. ниже в табл. 11):[2]

в алголе-68

sign.	взятие знака,
entier.	взятие ближайшего снизу целого числа,
round.	округление,

в паскале

trunc(.)	взятие ближайшего целого числа, заключённого между нулём и данным вещественным числом,
round(.)	округление, задаваемое формулой

$$round(x) = \begin{cases} trunc(x+0.5) & \text{при} \quad x \geqslant 0, \\ trunc(x-0.5) & \text{при} \quad x \leqslant 0. \end{cases}$$

В алголе-68 имеется также смешанная операция возведения в степень с целочисленным показателем

.↑. с функциональным типом **(real, int)real** : .

---

[1] О неявном обобщении (с функциональным типом **(char)string**:) идёт речь (и тогда, когда в вычислительной структуре (V*, V) мы заменяем ⟨.⟩ на . (алгол-68); см. табл. 7.

[2] Ещё одна (частично определённая) операция того же функционального типа, переводящая целые вещественные числа в соответствующие целые числа, остаётся обычно необозначаемой.

Над V (и даже над V^*) определено[1] отображение в натуральные числа, называемое ***перечислением, нумерацией*** или ***счётом.*** Оно характеризуется тем, что не имеет „пробелов" и сохраняет порядок знаков (в случае V^* — лексикографический порядок слов), а потому однозначно обратимо. Это отображение с типом (**char**)**int**., соотв. (*char*): *integer*, обозначается в алголе-68 через **abs**, в паскале через *ord*(.).

Дальнейшие подробности об этих операциях, в частности описание обратных к ним операций **repr**., соотв. *chr*(.), см. в табл. 11

Другой важной операцией на V^* со значениями в $\mathbb{Z}$ является операция *length*(.) нахождения длины слова (в алголе-68 обозначаемая через **upb**.). Нет никакой необходимости включать эту операцию в число базовых, потому что она, как это будет показано в дальнейшем, может быть сведена к другим базовым операциям.

Это верно и в отношении таких смешанных операций, как

«взятие i-го элемента слова a»
с функциональным типом
(**string, int**)**char**:
(в алголе-68 эта операция обозначается через $a[i]$)[2],

или

«переход от данного слова a к слову, которое получается из a путём i-кратного применения операции *rest* (если i неотрицательно) или $(-i)$-кратного применения операции *lead* (если i неположительно)»
с функциональным типом
(**string, int**)**string**:
(в алголе-68 эта операция обозначается через $a[i+1:\mathbf{upb}\ a]$ в случае положительных i и через $a[1:(\mathbf{upb}\ a)+i]$ в случае отрицательных)[2],

или

«переход от данного слова a к слову, совпадающему с a во всех элементах, за исключением i-го, который равен x»
с функциональным типом
(**string, int, char**)**string**:
(Грис ввёл для этой операции обозначение $(a;\ i:x)$).

---

[1] Поскольку V является конечным или в крайнем случае счётным набором знаков.

[2] В алголе-68 эта операция непосредственно имеется в распоряжении, в паскале же она должна быть надлежащим образом введена средствами языка.

Вообще можно сказать, что выбор операций в качестве примитивных — это до известной степени дело вкуса. Этот вопрос надо в каждом отдельном случае решать, исходя из соображений конкретной целесообразности, а также и из соображений машинной реализации (см. гл. 4 и 6).

2.1.3.8*. Составные объекты

Слова (последовательности знаков) и бинарные списки являются примерами ***составных объектов***: они составлены из отдельных знаков. С таким же успехом можно рассматривать последовательности или бинарные списки, которые состоят из объектов другого сорта. Опережая общее изложение этого вопроса в гл. 6, мы уже сейчас ради содержательности примеров будем рассматривать очевидные конкретные варианты — последовательности целых чисел, последовательности вещественных чисел и последовательности значений истинности, соответственно битовые последовательности; при этом в табл. 7 надо лишь заменить **char** на **int**, **real**, **bool**, **bit**. Индикаторы этих новых множеств объектов образуются добавлением **sequ** перед индикатором составных элементов:

sequ int, **sequ real**, **sequ bool**, **sequ bit**.

Таким образом, **string** можно трактовать как сокращение для **sequ char**, а **bits** — как сокращение для **sequ bit**.

2.2. Формулы

2.2.1. Обозначения параметров

> *„Der Name ist durch keine Definition weiter zu zergliedern:*
> *Er ist ein Urzeichen."* [1]
>
> Людвиг Витгенштейн

Алгоритмы — это описания общих методов; хотя фиксированные стандартные обозначения объектов и могут в них встречаться, их, однако, недостаточно для такого описания: алгоритмы должны выполняться над меняющимися объектами — ***параметрами***. В связи с этим требуются ***обозначения параметров***. Такие обозначения можно выбирать произвольно (***свободно выбираемые обозначения, идентификаторы*** (*англ.*: *identifier*)); обычно это буква, комбинация букв или комбинация букв и цифр, которая начинается буквой. (Чтобы обеспечить себе пол-

---

* Изучение этого раздела можно отложить до 3.3.3.2.

[1] „Имя нельзя расчленить дальше никаким определением: это первичный знак." (*нем.*) — *Прим. перев.*

ную свободу, стандартные обозначения (изображения) знаков и последовательностей знаков заключают в кавычки.)

Примеры: *a*, *x*, *mvar*, *anton*, *a*1, *x*0.

Обозначения параметров — это некий суррогат стандартных обозначений объектов, они по своей природе суть *имена*, которые можно присваивать объектам[1]. Изменяющаяся „загрузка" имён объектами (*интерпретация*) находит выражение также в слове *переменная*. (Мы будем избегать такого словоупотребления, чтобы не возникало путаницы, с одной стороны с базовыми именами алгола-68, с другой — с (программными) переменными из гл. 3).

"The name of the song is called 'Haddocks Eyes'" "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
"No, you don't understand", the Knight said, looking a little vexed. "That's what the name is called. The name really is 'The Aged Aged Man',"
"Then I ought to have said 'That's what the song is called'?" Alice corrected herself.
"No, you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called, you know!"
"Well, what is the song, then?" said Alice, who was by this time completely bewildered.
"I was comming to that", the Knight said. "The song really is 'A sitting On A Gate': and the tune's my own invention."

Льюис Кэррол (Чарлз Лютвидж Доджсон (1832—1898), английский математик)
'Through the Looking-Glass', Ch. VIII.[2]

---

[1] Здесь уместно сказать несколько слов по поводу различия между изображениями и обозначениями. Формально изображение также является обозначением некоторого значения. Разница состоит в том, что обозначение несет в себе элемент произвола, временного и локального соглашения, устанавливаемого, в частности, описанием тождества, тогда как изображение исчерпывающе и постоянно характеризует конкретное значение. Например, мы можем обозначать отношение длины окружности к диаметру как π, *пи* или *pi*, но иметь только одно изображение для него с помощью десятичных чисел: 3.1415926... . Поясним теперь, почему для имён не так важны изображения. Пусть читатель предположит себя в роли сотрудника дирекции по эксплуатации зданий, устанавливающего внизу подъезда почтовые ящики для жильцов. Как физические экземпляры ящики могут, например, отличаться заводским номером или положением на стенке. Однако такое „изображение" ящика, т. е. обозначение, исчерпывающе характеризующее данный физический экземпляр, никому не интересно — ни сотруднику дэза, ни жильцу, ни почтальону. Повесив их на стенке, сотрудник напишет краской на них номера квартир (обозначения-идентификаторы), которые и будут фактически „работать" при доставке корреспонденции. — *Прим. ред.*

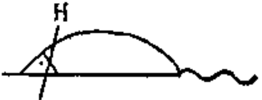
[2] — Заглавие этой песни называется „Пуговки для сюртуков".

— Вы хотите сказать — песня так называется? — спросила Алиса, стараясь показать, что песня её очень интересует.

— Нет, ты не понимаешь, — ответил нетерпеливо Рыцарь. — Это загла-

2.2.2. Формулы и формуляры

2.2.2.1. Построение формул



Формулы получаются при подстановке операций в операции (композиция операций), например:

$gcd(gcd(60,24),15)$, $(4+1)\times(4-1)$,

$first(rest(rerst(''ABCD'')))$, $\langle''H''\rangle + rest(''MAUS'')$;

в них могут входить и обозначения параметров, скажем:

$$gcd(gcd(a,b),c),\ (u+1)\times(u-1),$$

$$first(rest(rest(a))),\ \langle x\rangle + rest(a).$$

Само собой разумеется, при подстановке соответствующие типы операций должны быть согласованы между собой: ***вырабатываемый результат*** подставляемой формулы должен быть как раз того сорта, который требуется на месте замены в формуле, куда она подставляется. Итак:

Формула — это
либо обозначение параметра,
либо стандартное обозначение для фиксированного объекта (***постоянная, константа***)[1],
либо символ операции с формулами в качестве операндов (обозначаемая этим символом операция называется ***доминирующей*** *(или* ***определяющей****) операцией формулы) — это и есть собственно формула („настоящая" формула).*

При инфиксной записи подставляемые формулы заключают в скобки:

$$((a+b)+c)+d,\qquad ((t\times u)\times v)\times w,$$

$$a+(b+(c+d)),\qquad t\times(u\times(v\times w)).$$

вие так называется. А песня называется „Древний старичок".

— Это у песни такое заглавие? — переспросила Алиса.

— Да нет! Заглавие совсем другое. „С горем пополам"! Но это она только так называется!

— А песня это какая? — спросила Алиса в полной растерянности.

— Я как раз собирался тебе об этом сказать. „Сидящий на столбе"! Вот какая это песня! Музыка собственного изобретения!"

(Перевод Н. Демуровой („Сквозь зеркало и что там увидела Алиса", Изд-во литературы на иностранных языках, София, 1967).) ... „Алиса в Зазеркальи", гл. VIII. (*англ.*) — *Прим. перев.*

[1] В паскале с целью сокращения записи можно вводить константы (с произвольно выбираемыми обозначениями) и для объектов, обладающих стандартными обозначениями.

Таблица 11

Приоритеты в алголе-68

Одноместные операции

Группа 10

Символ операции	Операнд a	Результат c	Примечания
— —	**int** **real**	**int** **real**	Перемена знака
sign	**int** **real**	**int** **int**	$c = \begin{cases} -1 \text{ для отрицательных } a \\ 1 \text{ для положительных } a \\ 0 \text{ для } a = 0 \end{cases}$
entier	**real**	**int**	даёт ближайшее снизу целое число (**entier** $(-0.5) = -1$)
round	**real**	**int**	округляет до ближайшего целого числа (**round** 0.5 не определено!)
abs	**int** **real** **char** **bool**	**int** **real** **int** **int**	} даёт абсолютную величину подряд по порядку **false** $\mapsto$ 0, **true** $\mapsto$ 1
repr	**int**	**char**	обращение операции **abs**, определённой на **char** (для отрицательных a не определено)
odd	**int**	**bool**	$c = \begin{cases} \textbf{true} \text{ для нечётных } a \\ \textbf{false} \text{ для чётных } a \end{cases}$
¬	**bool** **bit** **bits**	**bool** **bit** **bits**	} отрицание поэлементное отрицание

Двуместные операции

Группа	Символ операции	Левый операнд a	Правый операнд b	Результат c	Примечание
8	↑	**int** **real**	**int** **int**	**int** **real**	$c = a^b$ для $b \geqslant 0$ $c = a^b$
	↑	**bits**	**int**	**bits**	сдвиг a на b позиций влево при $b > 0$, на $-b$ позиции вправо при $b < 0$ с заполнением O

Таблица 11 (продолжение)

Группа	Символ операции	Левый операнд a	Правый операнд b	Результат c	Примечание
7	×	**int**	**int**	**int**	умножение
	× /	**real**	**real**	**real**	умножение, де ение (деление на 0 исключено)
	div **mod**	**int** **int**	**int** **int**	**int** **int**	} результат и остаток при целочисленном делении (деление на 0 исключено)
6	−	**int** **real**	**int** **real**	**int** **real**	} вычитание
	+	**int** **real**	**int** **real**	**int** **real**	} сложение
	+	**string**	**string**	**string**	конкатенация
5	< >	**int** **real** **char** **string**	**int** **real** **char** **string**	**bool** **bool** **bool** **bool**	} в обычном смысле в смысле лексикографического порядка над алфавитом объектов вида **char**
	⩽ ⩾	**int** **real** **char** **string** **bool** **bit** **bits**	**int** **real** **char** **string** **bool** **bit** **bits**	**bool** **bool** **bool** **bool** **bool** **bool** **bool**	} в обычном смысле в смысле лексикографического порядка над алфавитом об ектов вида **char** } $\leqslant : c =$ **true**, если $(a \vee b) = b$
4	= ≠	**int** **real** **char** **string** **bool** **bit** **bits**	**int** **real** **char** **string** **bool** **bit** **bits**	**bool** **bool** **bool** **bool** **bool** **bool** **bool**	в обычном смысле
3	∧	**bool** **bit** **bits**	**bool** **bit** **bits**	**bool** **bit** **bits**	} конъюнкция поэлементная конъюнкция
2	∨	**bool** **bit** **bits**	**bool** **bit** **bits**	**bool** **bit** **bits**	} дизъюнкция поэлементная дизъюнкция

В случае ассоциативных операций эти скобки могут быть опущены:

$$a+b+c+d, \qquad t \times u \times v \times w.$$

Строго говоря, речь идёт в этом случае о многоместных операциях

$$.+.+.+., \qquad .\times.\times.\times.,$$

которые по желанию можно «читать» слева направо либо справа налево, т. е. представлять тем или иным образом с помощью последовательности двуместных операций.

Для неассоциативных операций, разумеется, опускать скобки нельзя:

$$a-(b-c), \qquad u/(v/w).$$

В арифметике издавна для борьбы со скобками применяют неявное ***правило старшинства***: инфиксные символы для умножения и деления „связывают" сильнее, чем инфиксные символы для сложения и вычитания.

Таблица 12

Приоритеты в паскале

Порядковый номер	Операции
1	**not**
2	* / **div mod and**
3	+ − **or**
4	$= \neq < \leqslant \geqslant >$

Развивая эту идею, символы двуместных арифметических операций, операций сравнения и булевых операций разбивают на несколько групп по их „рангу", скажем в алголе-68 — на десять групп (табл. 11), в паскале — на три группы (табл. 12). Эти определения имеют много общего — и там и тут $(a-b) \leqslant x$ можно сократить до $a-b \leqslant x$, — однако в целом они между собой несравнимы; в других языках программирования применяются свои определения. Международного стандарта на этот счёт до сих пор нет, да, пожалуй, он и не нужен.

Между префиксными символами для одноместных операций и инфиксными символами также определяется отношение предпочтения; в алголе-68 все префиксные символы (группа 10 в табл. 11) имеют приоритет перед всеми инфиксными, в паскале знак отрицания $\neg$ имеет приоритет перед знаком конъюнкции $\wedge$, знак же — (обращение знака) приоритета перед ним не

имеет[1]. Несмотря на наличие правил старшинства, стоит (и не только новичкам!) при помощи „лишних" скобок чётко выражать свои „намерения"; формулы

$$(7 \times 3) < (4 \times 5) \wedge (2 \times 2 = 5), \quad (-3) \uparrow 2 + (-4) \uparrow 2,$$
$$3 \times (-1)$$

явно легче читаются, чем соответственно (алгол-68)

$$7 \times 3 < 4 \times 5 \wedge 2 \times 2 = 5, \ -3 \uparrow 2 + -4 \uparrow 2, \ 3 \times -1.$$

В паскале же писать, например,

$$(7 * 3 < 4 * 5) \,\mathbf{and}\, (2 \times 2 = 5) \text{ или } (m \geqslant 0) \,\mathbf{and}\, (n \geqslant 0)$$

просто требуется!

В большинстве употребительных языков программирования можно не только опускать скобки, но и добавлять сколько угодно „лишних" скобок, как, скажем, в формуле $(3 \times ((-1)))$ (в алголе-68 и паскале это, во всяком случае, допустимо).

Формулы, которые „ведут" в $\mathbb{B}_2$ и, стало быть, результатом имеют значения истинности (*булевы формулы*), называются ***высказываниями***; в них используются и предикаты.

Высказываниями (с параметрами подходящего сорта) являются, например

$a \leqslant b$,
$a \leqslant b \wedge b \leqslant c$ (часто сокращается до $a \leqslant b \leqslant c$).

(Всегда истинные высказывания вроде

$$a \vee \neg a \quad \text{или} \quad \neg p \vee (p \wedge q) \vee \neg q$$

называются ***тавтологиями***.)

2.2.2.2. Формуляры

Формулам соответствуют в качестве самого простого и естественного вспомогательного средства для проведения вычислений (***вычислительные***) ***формуляры*** („расчётные бланки"), на которых и проводятся соответствующие ***вычисления*** („обработка", „отработка" формулы). Типичным примером из повседневной жизни служат формуляры, которые применяются в учреждениях, например формуляр для расчёта подоходного налога (рис. 45). Более сложные примеры можно найти в инженерных науках. На рис. 46 приведён заимствованный у Цузе пример из статики строительных сооружений (рис. 46).

---

[1] В соответствии с математическим обычаем писать — (—1) [а не — —1, — *Изд. ред.*], *понимая —. как сокращённую запись для 0—.*.

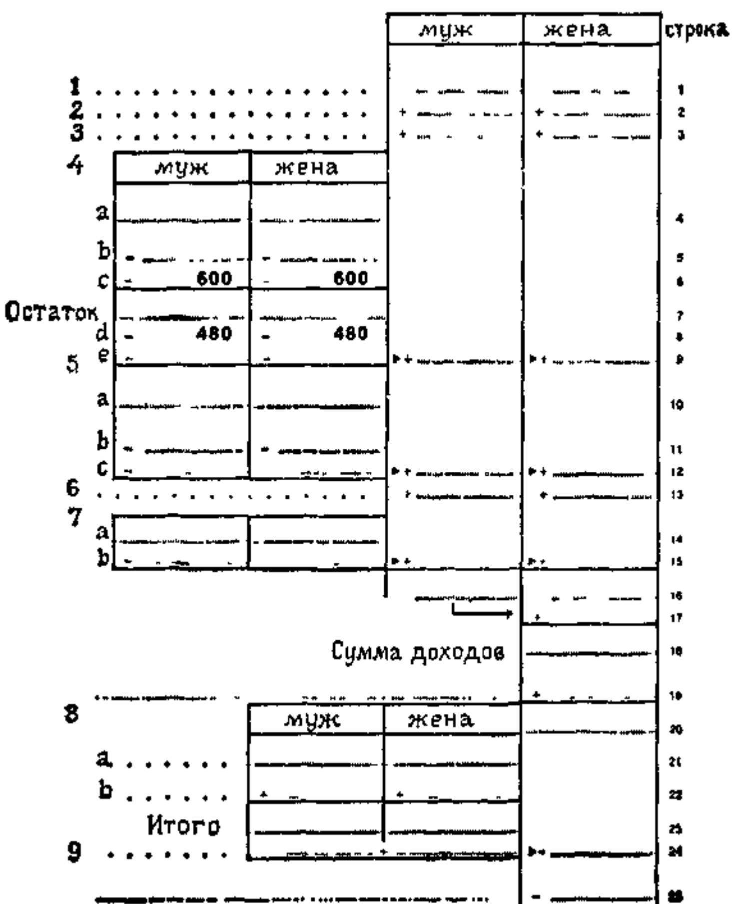


Рис. 45 Фрагмент формуляра для расчёта подоходного налога.

1. Доходы от занятий сельским хозяйством и лесным хозяйством
2. Доходы от занятий ремеслом
3. Доходы от независимой работы
4. Доходы от работы на предприятии
   - a Полная заработная плата
   - b Необлагаемая налогом часть доходов от социального обеспечения (40 %, но не больше 4800 марок на каждого из супругов)
   - c Необлагаемая налогом часть денежного подарка от предпринимателя по случаю рождества
   - d Необлагаемая налогом часть заработной платы
   - e Расходы на рекламу (максимум на общую сумму 564 марки)
5. Доходы от капитала
   - a Сумма доходов
   - b Расходы на рекламу (максимум на общую сумму 100 марок, на супружескую пару 200 марок)

c Необлагаемая налогом часть дохода (300 марок, на супружескую пару 600 марок)
6 Доходы от сдачи внаём и в аренду
7. Прочие доходы
a Сумма доходов (при пожизненной ренте лишь соответствующая часть)
b Расходы на рекламу (максимум на общую сумму 200 марок)
8. Сумма, не облагаемая возрастным налогом, для родившихся до 2.1.1917
a Полная заработная плата, не считая доходов от социального обеспечения
b Сумма положительных величин под номерами с 1 по 3 и с 5 по 7 (но не считая дохода от пожизненной ренты)
9. 40 % итоговой суммы, но не больше 3000 марок на каждого из супругов

[Всё — в марках ФРГ. В оригинальном формуляре все пояснения даны прямо на бланке в соответствующих строках. — *Изд. ред*.]

Для рассматриваемой в этом примере формулы

$$(*) \qquad (b\times 2+a)\times d+(a\times 2+b)\times c$$

с параметрами a, b, c и d можно использовать компактный формуляр, изображённый в левой части рис. 47 (умножения вы-

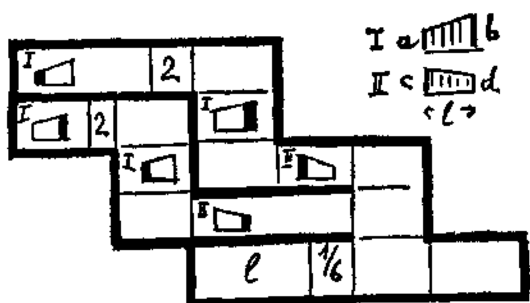


Рис 46 Вычислительный формуляр „для проведения расчёта с двумя моментами, распределенными по линейному закону“, в том виде, как он был у К. Цузе

полняются по горизонтали, сложения — по вертикали); выбранное здесь графическое решение формуляра является одним из многих возможных, экономящих место. В дальнейшем мы будем обычно изображать формуляры схематически — как на рис. 47 справа; в „ромбики“ заносятся значения параметров и (промежуточных либо окончательных) результатов.

Для теоретических целей уместно продолжить процесс абстрагирования; взаимосвязь между ромбиками для операндов и ромбиками для результатов, взаимосвязь выполняемых при „от-

работке“ формулы подстановок выражается топологически[1, 2] ***деревом Канторовича***[3, 4] или ***схемой потока данных***[5], отвечающими данной формуле (обычно они изображаются так, как показано на рис. 48).

Формуляры являются графическими представлениями дерева Канторовича, они устанавливают взаимосвязь шагов вычисления

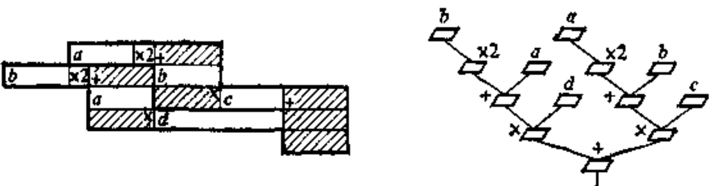


Рис. 47. Практический и схематический вычислительные формуляры для формулы $(b \times 2 + a) \times d + (a \times 2 + b) \times c$.

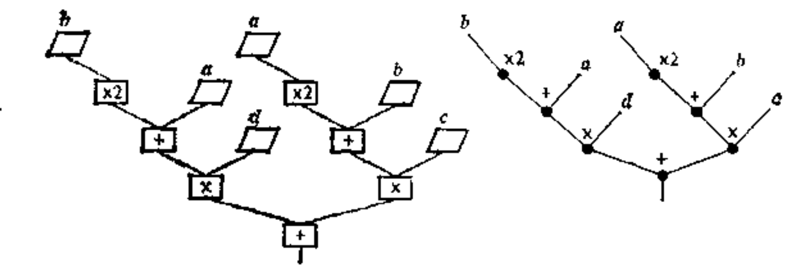


Рис. 48. Дерево Канторовича (справа) и схема потока данных (слева) для формуляра на рис. 47.

и, кроме того, предлагают место для проведения вычисления. Обработка может вестись в тех узлах, для которых уже определены все операнды.

---

[1] Здесь „топологически“ означает „с точки зрения теории графов“. — *Прим. перев.*

[2] Для фигурирующих в формуле операций можно рассмотреть отношение „вычислить раньше, чем“, которое является отношением (частичного) порядка (см. „Прикладную алгебру“ Биркгофа и Барти). Дерево Канторовича и схема потока данных представляют порождающий граф этого отношения порядка.

[3] По имени русского математика Л. В. Канторовича, который в 1955 г. ввел это понятие для случая арифметических выражений.

[4] В [действующей в ФРГ. — *Изд. ред.*] школьной программе по математике (для 5-го года обучения) дерево Канторовича называется „вычислительным деревом“, см., например, Schmitt-Wohlfarth, Mathematik Buch 5, Bayerischer Schulbuch-Verlag, München, 1976.

[5] Схемы потоков данных в узком смысле, широко используемые при коммерческой обработке данных, иллюстрируют формулы для обработки последовательностей знаков. Символическое изображение таких схем стандартизовано (см. DIN 66001).

Отметим, что (в противоположность обычному в школьной математике вычислению значений формул) вычислительный формуляр типа указанного выше позволяет организовать проведение вычислений (***вычислительный процесс***) многими различными способами (***свобода вычислений***); скажем, можно сначала

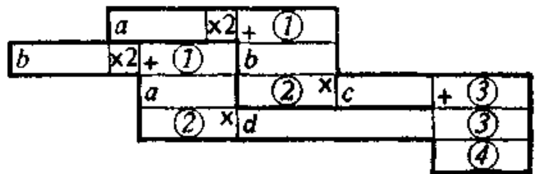


Рис. 49. Вычислительный формуляр для формулы $(b\times2+a)\times d+(a\times2+b)\times c$ с указанием номера шага при параллельной обработке

найти $a\times2$, затем $b\times2$, затем $a\times2+b$, затем $b\times2+a$, $(a\times é+b)\times c$ и т. д. („вычисление снизу вверх"), а можно сначала $b\times2$, затем $b\times2+a$, затем $(b\times2+a)\times d$, затем $a\times2$

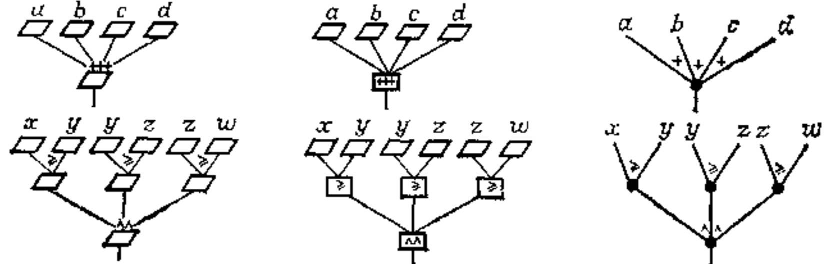


Рис. 50. Вычислительные формуляры, схемы потока данных и деревья Канторовича для формулы с многоместными ассоциативными операциями.

и т. д. („вычисление слева направо"). Вообще сразу много лиц могут параллельно выполнять определённые части вычисления [1]. Правда, *некоторые* части формул должны при этом вычисляться *перед другими*; для примера, приведенного выше, на рис. 49 в клеточках для промежуточных результатов цифрами в кружках указано, на каком такте (шаге) вычисления они могут быть вычислены. *Лишь* в случае формул специального вида, таких как

$$(((a1+b1)\times a2+b2)\times a3+b3)\times a4+b4,$$

для процесса вычисления нет выбора, здесь мы вынужденным образом имеем ***последовательный вычислительный процесс***.

---

[1] Во время *второй мировой войны* некотоые баллистические расчёты выполнялись таким способом одновременно десятками (людей-) вычислителей (сообщение Альвина Вальтера).

В общем же случае формулы обычно допускают совместное вычисление (см. 1.6.1); в частности, так обстоит дело, когда в них встречаются многоместные ассоциативные операции (рис. 50). Части вычислительного процесса, которые могут выполняться параллельно, называются ***подпроцессами***.

2.2.2.3. Строгость операций

Прежде всего резюмируем сказанное выше относительно вычисления значений формул:

результат формулы, которая не сводится к обозначению параметра или стандартному обозначению, определяется рекурсивно: он получается применением доминирующей операции формулы к результатам формул-операндов.

Поэтому результат формулы останется тем же самым, если некоторую подформулу заменить на её результат.

При этом та или иная операция вполне может обладать тем свойством, что (в определённых случаях) её результат можно

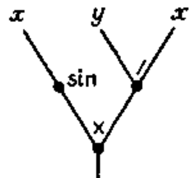


Рис. 51. Дерево Канторовича для формулы $sin(x) \times (y/x)$.

указать ещё до того, как станут известны все операнды. Так, например, результат умножения

›левая формула‹ $\times$ ›правая формула‹

уже известен, если вычислен результат ›левой формулы‹ и оказалось, что он равен нулю; в этом случае результат ›правой формулы‹ можно вообще не вычислять. Аналогично обстоит дело с конъюнкцией и дизъюнкцией высказываний.

Особая обработка такого рода ситуаций должна проводиться не „тайком", а явно, посредством надлежащего разбора случаев. Соответствующие средства будут введены в 2.2.3. В связи с этим сформулируем следующий принцип:

Все базовые операции используемых вычислительных структур должны быть ***строгими***, т. е. при их применении в вычислительных формулярах при всех обстоятельствах должны вычисляться в с е их операнды.

При этом не только, скажем, формула

$$1/0$$

не имеет определённого результата (операция ./. является частично определённой, см. 2.1.3.3), но и результат формулы

$$0 \times (1/0)$$

не определён; вычислительный формуляр для формулы $sin(x) \times (y/x)$ с деревом Канторовича, представленным на рис. 51, при $x = 0$ не даёт никакого результата.

2.2.2.4. Преобразование формул

Если применить к формуле, которая определена над некоторой вычислительной структурой, какие-либо законы этой структуры, то получится *эквивалентная* формула, которая определяет то же самое отображение[1]. Обычно преобразованной формуле (несмотря на её эквивалентность исходной) отвечает другой вычислительный формуляр, а с ним и другое дерево Канторовича. Иногда эти изменения незначительны, как будет, например, в случае, если приведённую выше формулу (*) над $\mathbb{Z}$ с помощью закона коммутативности преобразовать к виду

$$(a \times 2 + b) \times c + (b \times 2 + a) \times d.$$

Формуле (*) эквивалентны также (получаемые при использовании закона дистрибутивности в $\mathbb{Z}$) формула

$$(c \times 2 + d) \times a + (d \times 2 + c) \times b$$

(отличающаяся от исходной лишь взаимной заменой b на c и a на d) и формула

$$(a \times c + b \times d) \times 2 + (a \times d + b \times c)$$

с одним дополнительным умножением, вычислительный формуляр для которой изображён на рис. 52.

С точки зрения возможностей параллельной работы представляет интерес нахождение для заданной формулы не только эквивалентной ей формулы с наименьшим числом операций, но и формулы, которая допускает вычисление за наименьшее число вычислительных шагов (тактов). Для вычисления значения полинома

$$a_0x^4 + a_1x^3 + a_2x^2 + a_3x + a_4$$

имеется формула

$$(((a0 \times x + a1) \times x + a2) \times x + a3) \times x + a4,$$

[1] Эквивалентность формул включает в себя также и совпадение областей определения соответствующих отображений. Формула $x \times (y/x)$ над $\mathbb{Z} \setminus \{0\}$ и формула y над $\mathbb{Z}$ не эквивалентны!

которая требует четырёх умножений, четырёх сложений и восьми вычислительных тактов (соответствующий формуляр,

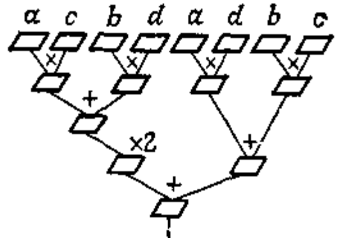


Рис. 52. Другой вычислительный формуляр для „проведения расчёта с двумя моментами".

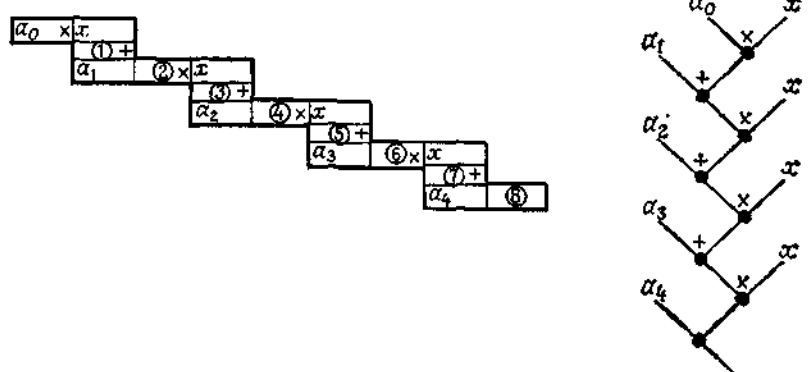


Рис. 53. Схема Горнера для вычисления значения полинома $a_0x^4 + a_1x^3 + a_2x^2 + a_3x + a_4$ и соответствующее дерево Канторовича.

	x × ×	x		
	a_3	① × ×	x	
a_4 +	①	a_2	② × ×	x
	② +	②	a_1	③ ×
		③ +	③	a_0
			④ +	④
				⑤

Рис. 54. Формуляр для параллельного вычисления значения того полинома, что и на рис. 53.

называемый *схемой Горнера*, и дерево Канторовича для схемы Горнера показаны на рис. 53).

Однако имеется и формуляр (рис. 54) для вычисления значений вышеупомянутого полинома, который требует семи умно-

жений и четырёх сложений, но зато даёт значение полинома всего за пять вычислительных тактов (на полях промежуточных результатов указано цифрами в кружках, на каком наименьшем такте этот результат может быть вычислен). Обобщение на полиномы произвольной степени является очевидным [1].

Способ формального описания таких формуляров с многократным применением промежуточных результатов мы изучим в 3.1.1.

2.2.3. Условные формулы

Если помимо композиции операций ввести в игру ещё и разбор случаев, то мы получим ***условные формулы***.

2.2.3.1. Альтернатива и последовательный разбор случаев

Простейший разбор случаев — это ***альтернатива*** [2], которая состоит из двух подформул — ***да-формулы*** и ***нет-формулы***, одна из которых и подлежит обработке (говорят также о двух ***ветвях*** альтернативы).

Выбор выполняется на основе результата, вырабатываемого некоторой другой формулой — некоторым высказыванием, которое называется ***условием***. Вырабатываемым результатом условия является значение истинности, т. е. либо T (и тогда „выбери да-формулу"), либо F („выбери нет-формулу"). Таким образом, условные формулы имеют следующий вид:

if ›условие‹ **then** ›да-формула‹ **else** ›нет-формула‹ **fi**
(алгол-68) [3]

или

(›условие‹ → ›да-формула›; T → ›нет-формула‹) (лисп) [4].

Фигурирующие здесь символы **if**, **then**, **else** и **fi** служат дальнейшими примерами использования словесных символов.

---

[1] Имеются схемы каскадного типа, для которых число тактов пропорционально лишь логарифму степени полинома.

[2] Называемая также ***бинарным разбором случаев.*** (Правильнее было бы говорить о *диадическом* разборе случаев [см. первое подстрочное примечание в разделе 2.1.3.5. — *Изд. ред.*].)

[3] В русскоязычной версии алгола эта формула выглядит так:

если ›условие‹ **то** ›формула-ТО‹ **иначе** ›формула-ИНАЧЕ‹ **всё**.

Здесь **если**, **то**, **иначе** — дословные переводы соответствующих английских слов, а вместо **fi** (**if** „наоборот") поставлено не **илсе**, а **всё**. — *Прим. изд. ред.*

[4] T стоит вместо »истина«; последовательная редакция этой формулы дана ниже.

В качестве да- или нет-формул (как и в качестве условия) могут стоять как (простые) формулы, так и условные формулы: *Условные формулы сами являются формулами.*

Примеры (с параметрами подходящего сорта):

(1) **if** $x > 0$ **then** x **else** $-x$ **fi**,

(2) **if** a **then** 1 **else** 0 **fi**,

(3) **if** $m > n$ **then** m **else** n **fi**,

(4) **if** $x > 0$ **then** 1 **else** (**if** $x \geqslant 0$ **then** 0 **else** -1 **fi**) **fi**.

Стало быть, с помощью разбора случаев операции взятия абсолютного значения, максимума (минимума) и знака могут быть сведены к другим операциям над целыми, соответственно над вещественными числами. При опускании скобок (они не нужны) последний пример принимает вид

if $x > 0$
then 1
else if $x \geqslant 0$
  then 0
  else -1 **fi fi**.

Таким образом, разбор случаев с более чем двумя ветвями можно реализовать с помощью вложенных альтернатив.

Для того частного случая вложения, когда, как в последнем примере, каждый раз (если не считать са́мого конца) в нет-случае ставится новое условие и потому все завершающие **fi** сосредоточиваются у правого края (***последовательный разбор случаев***)[1], вводится сокращенная форма записи (**else if** „стягиваются" до **elsf**), скажем для вышеприведённого примера

if $x > 0$ **then** 1 **elsf** $x \geqslant 0$ **then** 0 **else** -1 **fi** (алгол-68)

(отметим, что при этом все внутренние **fi** опускаются) или

$(x > 0 \rightarrow 1;\ x \geqslant 0 \rightarrow 0;\ \mathbf{T} \rightarrow -1)$ (лисп).

В паскале завершающее **fi** опускается, но есть возможность „расставить скобки" с помощью *словарных скобок* **begin** и **end**[2]:

begin if ›условие‹ **then** ›да-формула‹ **else** ›нет-формула‹ **end**.

---

[1] Обычные правила применения продукций в алгоритмах Маркова также приводят к последовательному разбору случаев.

[2] Begin — начало, end — конец. — *Прим. изд. ред.*

Это верно и для последовательного разбора случаев, при котором **else if** не стянуты в **elsf**. Примеры:

begin if $x > 0$ **then** x **else** $-x$ **end** (паскаль),

begin if $x > 0$ **then** 1 **else if** $x \geqslant 0$ **then** 0 **else** -1 **end** (паскаль).

В стандартном паскале, однако, условные формулы, строго говоря, не предусмотрены, в частности их нельзя подставлять в другие формулы. Однако условные формулы запрятаны за такими конструкциями, как

begin if ›условие‹ **then** ›Res‹ ⇐ ›да-формула‹
else ›Res‹ ⇐ ›нет-формула‹ **end**,

где ›Res‹ — это обозначение *результата*, см. 2.3.1.1.

Само собой разумеется, что обе ветви альтернативы, да-формула и нет-формула, должны иметь результат одного и того же сорта (вида, типа), и тогда условная формула приводит к тому же сорту. Далее, очевидно, что формуле

if ›условие‹ **then** ›да-формула‹ **else** ›нет-формула‹ **fi**

эквивалентна формула

if ¬›условие‹ **then** ›нет-формула‹ **else** ›да-формула‹ **fi**

(***перестановка ветвей*** альтернативы при отрицании условия).

Пример

$x +$ **if** $x > 0$ **then** x **else** $-x$ **fi**

показывает, что условные формулы могут выступать как составная часть (простой) формулы — правда не в паскале. Впрочем, такие формулы можно преобразовать к „обычному" виду, перенося соответствующую операцию (это возможно, поскольку все операции у нас строгие) в да-формулу и в нет-формулу:

if $x > 0$ **then** $x + x$ **else** $x - x$ **fi**.

Применение законов из вычислительной структуры Z даёт окончательно

if $x > 0$ **then** $x \times 2$ **else** 0 **fi**.

При работе с объёмистым, обширным разбором случаев — особенно вложенного типа — весьма полезен так называемый „метод таблиц решений". Этому подходу близок (а функционально тождествен) подход с использованием двузначных схем („алгебра схем"), см. гл. 4.

2.2.3.2. Охраняемый разбор случаев

Альтернатива является частным (двучленным) случаем ***охраняемого (контролируемого, защищаемого) разбора случаев с n ветвями***:

if ›условие 1‹ **then** ›формула 1‹

□ ›условие 2‹ **then**› формула 2‹

.
.
.

□ ›условие n‹ **then** ›формула n‹ **fi**,

при котором перед к а ж д о й формулой ставится некоторое условие, называемое ***охраняющим***, или ***защищающим***, или коротко — ***стражем***. Альтернативе

if ›условие‹ **then** ›да-формула‹

else ›нет-формула‹ **fi**,

эквивалентен охраняемый разбор случаев с двумя ветвями

if ›условие‹ **then** ›да-формула‹

□ ¬›условие‹ **then** ›нет-формула‹ **fi**.

Примеру (1) из 2.2.3.1 соответствует охраняемый разбор случаев

if $x > 0$ **then** x

□ $x \leqslant 0$ **then** $-x$ **fi**.

Трёхчленный пример доставляет операция **sign** (см. 2.1.3.7 и табл. 11):

if $x > 0$ **then** 1

□ $x = 0$ **then** 0

□ $x < 0$ **then** -1 **fi**.

Порядок, в котором записываются отдельные ветви охраняемого разбора случаев, неважен. Не говоря уже о том, что охраняемый разбор случаев обладает поэтому совершенно симметричной структурой и в явном виде предъявляет условия для в с е х ветвей, в частности и для ветви **else** в случае альтернативы, в охраняемом разборе случаев заключены принципиально новые возможности. А именно, может случиться, что сразу несколько стражей дадут значение »истина«. Тогда открыт выбор между соответствующими формулами, и этот выбор может быть осуществлён п р о и з в о л ь н о. Такой недетерминистический (см. 1.6.1) разбор случаев может привести к формулам,

которые не дают однозначного результата и поэтому задают не отображения, а соответствия [1], т. е. „многозначные отображения", с „открытым" выбором значений. Что это не обязательно должно быть так, показывает пример формулы

$$\textbf{if } x \geqslant 0 \textbf{ then } x$$
$$\square\ x \leqslant 0 \textbf{ then } -x \textbf{ fi}$$

— полностью симметричного варианта вышеприведенного примера (1) (определение абсолютного значения).

Напротив, формула [2]

$$\textbf{if } x \geqslant 0 \textbf{ then } sqrt(x)$$
$$\square\ x \geqslant 0 \textbf{ then } -sqrt(x) \textbf{ fi}$$

не задаёт однозначно определённого отображения. Для $x < 0$ эта формула не определена, а для $x \geqslant 0$ выдаёт „обращение" функции возведения в квадрат. Какое из двух значений квадратного корня получится в результате, неизвестно — формула недетерминированна.

А вот следующая формула будет детерминированной, несмотря на то что стражи частично „перекрываются":

$$\begin{array}{lll}\textbf{if } x \leqslant -y & \textbf{then } 0 & \\ \square\ -y \leqslant x \wedge x \leqslant 0 & \textbf{then } -sqrt(-x \times (y+x)) & \\ \square\ 0 \leqslant x \wedge x \leqslant y & \textbf{then } sqrt(x \times (y-x)) & \\ \square\ y \leqslant x & \textbf{then } 0 & \textbf{fi}.\end{array}$$

Детерминированной является также формула (ср. с 2.2.2.3)

$$\begin{array}{ll}\textbf{if } x = 0 & \textbf{then } 0 \\ \square\ y = 0 & \textbf{then } 0 \\ \square\ x \neq 0 \wedge y \neq 0 & \textbf{then } x \times y \textbf{ fi}\end{array}$$

и даже формула

$$\begin{array}{ll}\textbf{if } x = 0 & \textbf{then } 0 \\ \square\ y = 0 & \textbf{then } 0 \\ \square\ \textbf{true} & \textbf{then } x \times y \textbf{ fi}.\end{array}$$

Недетерминистический охраняемый разбор случаев в алголе-68, как и в паскале, не предусмотрен.

В паскале завершающего **fi** не ставится.

Идея охраняемого разбора случаев восходит к Дейкстре (1975 г., '*guarded commands*' [3]).

---

[1] См. приложение С. — *Прим. изд. ред.*

[2] Ниже *sqrt* — операция извлечения квадратного корня. — *Прим. изд. ред.*

[3] „Охраняемые команды" (*англ.*). — *Прим. изд. ред.*

2.2.3.3. Проведение вычислений на формулярах с разбором случаев

Условным формулам отвечают вычислительные формуляры с разбором случаев. Альтернатива выступает в соответствующих деревьях Канторовича и схемах потока данных в виде

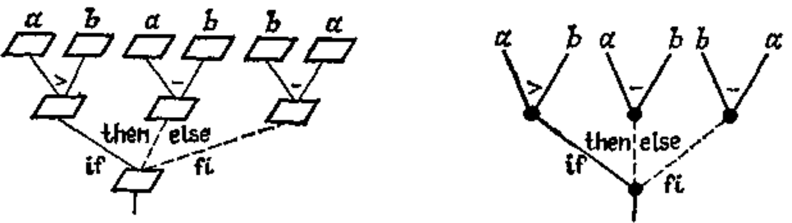


Рис. 55. Формуляр и дерево Канторовича для приведённой в тексте альтернативы.

трёхместной операции, как, например, на рис. 55 для условной формулы

$$\textbf{if } a > b \textbf{ then } a - b \textbf{ else } b - a \textbf{ fi}.$$

Если действовать „наивно“ (т. е. понимать альтернативу как строгую трёхместную операцию), то надо вычислять $a > b$, $a - b$ и $b - a$. Правда, не слишком много толка в том, чтобы вычислять обе последующие формулы, а затем одну из них (какую именно, зависит от результата сравнения $a > b$) отбрасывать.

Для экономии работы примем следующий принцип обработки альтернативы:

Сначала надо обработать условие альтернативы, а затем в зависимости от полученного результата исключить ненужную ветвь; обрабатывается лишь оставшаяся ветвь.

Чтобы это подчеркнуть, будем в формулярах и деревьях Канторовича, равно как и в схемах потоков данных, линии, ведущие к да- и нет-формулам, делать пунктирными (см. рис. 55). При фактической обработке альтернативы пунктирная линия, которая отвечает выбранной ветви, станет „сплошной“.

Аналогично поступаем мы и при охраняемом разборе случаев. На рис. 56 изображены вычислительный формуляр и дерево Канторовича для вышеприведенного примера с функцией **sign**.

Итак, для обработки охраняемого разбора случаев мы принимаем следующий принцип:

Те ветви, стражи которых выдали F, *безоговорочно исключаются; между всеми ветвями, стражи которых выдали* T, *производится свободный выбор.*

Наименование „страж" получает тем самым своё оправдание: страж оберегает „свою" ветвь от ненужной обработки. Также и условие альтернативы выполняет „охранительные функции". Если ровно один страж выдаёт **T**, то, конечно, не

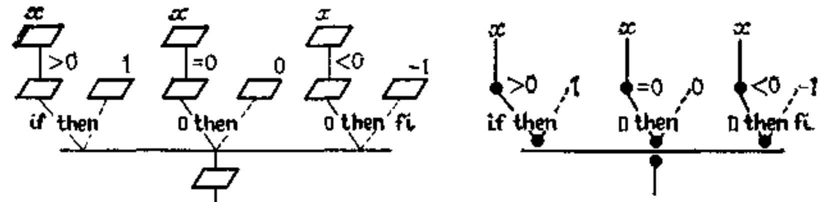


Рис. 56. Формуляр и дерево Канторовича для охраняемого разбора случаев, определяющего функцию sign.

остаётся никакого выбора, это классическая *детерминированная* ситуация. Если же ни один страж не выдаёт **T**, то результат вычисления формулы является неопределённым.

Но и при охраняемом разборе случаев всё ещё имеется значительная свобода вычислений: порядок, в котором вычисляются стражи и ветви, ограничен лишь тем, что ветвь нельзя выполнять, пока не будет вычислен её страж и результатом этого вычисления не окажется **T**.

Простейшая возможность состоит в том, чтобы сначала вычислить все стражи (друг за другом или разом), а потом выбрать допустимую ветвь. При другом, более экономичном способе работы произвольно выбирают один страж за другим и вычисляют их значения; как только появится **T**, процесс заканчивается вычислением соответствующей ветви. В любом случае все условия должны иметь определённый результат.

Стражи не только оберегают „свою" ветвь от ненужной обработки — они защищают её от случайной обработки (так что их роль не сводится исключительно к экономии работы). Это верно и в отношении условия альтернативы: вышеприведенный принцип позволяет рассматривать условную формулу

$$\textbf{if } a \geqslant b \textbf{ then } a - b \textbf{ else } b - a \textbf{ fi}$$

и над $\mathbb{N}$, потому что частично определённую (в $\mathbb{N}$) операцию вычитания нужно будет выполнять только тогда, когда она выполнима.

В принципе выполнимость всех частично определённых операций должна быть обеспечена с помощью подходящих стражей, как, скажем, в формуле

$$\textbf{if } x \neq 0 \textbf{ then } sin(x)/x \textbf{ else } 1 \textbf{ fi}.$$

Такая страховка необходима и для такого охраняемого разбора случаев, у которого результат может быть не определён, например для формулы

$$\textbf{if } a > b \textbf{ then } a - b$$
$$\square\ a < b \textbf{ then } b - a \textbf{ fi}.$$

Эта формула должна быть поставлена под охраняющее условие $a \neq b$ (или даже под какое-нибудь более сильное условие), скажем в такой форме:

$$\textbf{if } a = b \textbf{ then } 0$$
$$\square\ a \neq b \textbf{ then if } a > b \textbf{ then } a - b$$
$$\square\ a < b \textbf{ then } b - a \textbf{ fi fi}.$$

То же относится и к охраняемому разбору случаев типа

$$\textbf{if true then } x \times (y/x)$$
$$\square \textbf{ true then } y \qquad \textbf{fi},$$

где имеется ветвь, результат которой не всегда определён; такой разбор случаев должен быть по меньшей мере поставлен под охрану стража $x = 0$ или лучше приведём к форме, в которой охраняется критическая ветвь:

$$\textbf{if } x \neq 0 \textbf{ then } x \times (y/x)$$
$$\square \textbf{ true then } y \qquad \textbf{fi}.$$

2.2.3.4. Нестрогий характер разбора случаев

Разбор случаев не мог бы исполнять своё второе назначение — охрану ветвей,— если бы он был введён как строгая операция.

Нестрогий характер разбора случаев ясно виден на следующих примерах.

Альтернативы

if ›условие 1‹ **then** ›условие 2‹ **else false fi**,

if ›условие 1‹ **then true else** ›условие 2‹ **fi**

имеют в качестве да- и нет-формул высказывания и являются логическими операциями. На рис. 57 показаны соответствующие деревья Канторовича. Функционально эти конструкции совпадают соответственно с конъюнкцией и дизъюнкцией, однако с точки зрения обработки существенно отличаются от них: формула ›условие 2‹ в некоторых ситуациях не обрабатывается. Для экономии работы конъюнкцию или дизъюнкцию всегда можно заменить на соответствующую альтернативу. Обратная

же замена в общем случае недопустима, так как утрачивается „охранительная функция".

Для указанных конструкций иногда вводят сокращенную запись

$$›\text{условие } 1‹ \mathbin{\underset{\cdot}{\wedge}} ›\text{условие } 2‹,$$

$$›\text{условие } 1‹ \mathbin{\dot{\vee}} ›\text{условие } 1‹.$$

Точки напоминают, что речь здесь идёт о нестрогих операциях конъюнкции и дизъюнкции, которые для отличия от „настоя-

Рис 57. Деревья Канторовича для последовательных конъюнкции и дизъюнкции.

щих" называют соответственно ***последовательной конъюнкцией*** и ***последовательной дизъюнкцией***.

К примеру, предикат

$$length(a) \leqslant 1$$

можно выразить так:

$$a = \lozenge \mathbin{\dot{\vee}} rest(a) = \lozenge \quad | \quad isempty \mathbin{\dot{\vee}} isempty(rest(a)).$$

За исключением разбора случаев и связанных с ним последовательных конъюнкции и дизъюнкции, в формулы, как и раньше, могут входить только строгие операции.

2.3. Подпрограммы

2.3.1. Описание подпрограмм

2.3.1.1. Нотация

Прежде чем ввести вслед за композицией операций и разбором случаев третий основной принцип, используемый при построении подпрограмм, — рекурсию, мы должны научиться записывать определения подпрограмм.

Для задания подпрограммы до сих пор служила формула, которая подлежит обработке. Однако по самой формуле не всегда видно, какого с о р т а её параметры. Вот почему необходимо указывать её функциональный тип, т. е. сорта (виды, типы) входных параметров и сорт результата. В математике для этого

обычно используют запись в две строчки, например:

$$f\colon \begin{matrix} \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z} \\ (u,v)\longmapsto(u+v)\,\mathbf{div}\,(u-v) \end{matrix} \quad \text{или} \quad \begin{matrix} \mathbb{Z}\times\mathbb{Z}\xrightarrow{f}\mathbb{Z} \\ f(u,v)\Leftarrow(u+v)\,\mathbf{div}\,(u-v). \end{matrix}$$

В распространённых языках программирования для ***описания подпрограмм*** используют несколько иную, исторически сложившуюся[1] форму записи, которая, грубо говоря получается, если вышеприведенные строчки „прочитать по столбцам". В алголе-68 пишут[2]

$$\mathbf{funct}\ f\equiv(\mathbf{int}\ u,\mathbf{int}\ v)\,\mathbf{int}:(u+v)\,\mathbf{div}\,(u-v)$$

или, короче,

$$\mathbf{funct}\ f\equiv(\mathbf{int}\ u,v)\,\mathbf{int}:(u+v)\,\mathbf{div}\,(u-v),$$

а в паскале[3] —

$$\mathbf{function}\ f(u:integer;\ v:integer):integer;$$
$$\mathbf{begin}\ f\Leftarrow(u+v)\,\mathbf{div}\,(u-v)\,\mathbf{end}$$

или, короче,

$$\mathbf{function}\ f(u,v:integer):integer;$$
$$\mathbf{begin}\ f\Leftarrow(u+v)\,\mathbf{div}\,(u-v)\,\mathbf{end},$$

с явным указанием *результата* посредством ***произвольно выбираемого обозначения*** подпрограммы.

Таким образом, в алголе и паскале перед самой формулой, называемой ***телом*** подпрограммы, ставится ***заголовок***[4], задающий обозначения параметров и функциональный тип подпрограммы. Алгольные и паскалевские заголовки различаются лишь порядком написания: в алголе сначала идёт сорт, а потом обозначение, в паскале — наоборот.

Разумеется, *никакие два обозначения параметров в заголовке подпрограммы не должны совпадать.*

Заголовок должен быть поставлен и на формуляре. При этом описание подпрограммы задаётся путём сопоставления дерева Канторовича фигурирующей в подпрограмме формулы с (тривиальным) деревом Канторовича определяемой операции, см. рис. 58.

При ***вызове*** подпрограммы к её обозначению добавляются фактические значения параметров, заключенные в скобки и

---

[1] Нотация алгола-68 — это по существу нотация так называемого *лямбда-исчисления* Чёрча.

[2] В стандартном алголе-68 надо писать **proc** вместо **funct**, а также $=$ вместо $\equiv$.

[3] В стандартном паскале необходимо писать $:=$ вместо $\Leftarrow$.

[4] В оригинале Kopfleiste (буквально: заставка). — *Прим. изд. ред.*

разделённые запятыми (ср. с 2.1.2), например $f(3, 1)$. В качестве фактических значений могут выступать также формулы,

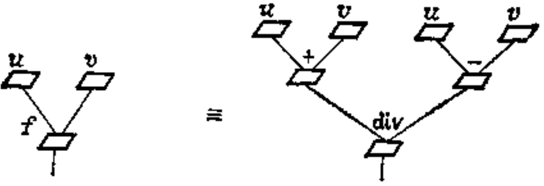


Рис 58. Формуляр с описанием подпрограммы.

точнее результаты их обработки (композиция операций, см. 2.2.2.1).

2.3.1.2. Системы подпрограмм

Формулы можно „структурировать“ — разлагать на составные части — посредством введения вспомогательных подпрограмм. Так, формула (∗) из 2.2.2.2 может быть преобразована к виду

$$h(b, a, d) + h(a, b, c),$$

где

funct $h \equiv (real\ a, b, c)$ **real**: $(a \times 2 + b) \times c$	**function** $h(a, b, c : real) : real$; **begin** $h \Leftarrow (a * 2 + b) * c$ **end**.

Если теперь ввести для (∗) подпрограмму

funct $f \equiv$ (**real** a, b, c, d) **real**: $h(b, a, d) + h(a, b, c)$	**function** $f(a, b, c, d : real)$; **begin** $f \Leftarrow h(b, a, d) + h(a, b, c)$ **end**,

то получится ***иерархическая система*** подпрограмм (f, h), в которой f ***непосредственно опирается*** на h. На рис. 59 изображён формуляр для этой системы. Для проведения вычисления f на этом формуляре потребуются два экземпляра формуляра для h.

Возможна и дальнейшая структуризация, скажем можно описание h заменить системой

funct $h \equiv$ (**real** a, b, c) **real**: $g(a, b) \times c$	**function** $h(a, b, c : real) : real$; **begin** $h \Leftarrow g(a, b) * c$ **end**
funct $g \equiv (real\ u, v)$ **real**: $u \times 2 + v$	**function** $g(u, v : real) : real$; **begin** $g \Leftarrow u * 2 + v$ **end**

В этом случае говорят, что f опирается на g ***косвенно***.

Рис. 59. Формуляр со вспомогательными формулярами.

2.3.1.3. Предохранители

Многие подпрограммы имеют смысл только для определённых значений параметров, т. е. они задают лишь частично определённую функцию. В частности, может случиться, что рассматриваемая формула содержит операции, которые определены лишь частично. Так обстоит дело, скажем для подпрограммы f из 2.3.1.1 или для следующей подпрограммы:

funct $z \equiv$ (*real* a, b) **real**: $a/b + b/a$	**function** $z(a, b : real) : real$; **begin** $z \Leftarrow a/b + b/a$ **end**.

Если надлежащим образом ограничить область определения — в указанных примерах с помощью дополнительных условий $u \neq v$ и $a \neq 0 \wedge b \neq 0$ соответственно, то мы опять получим *всюду определённую* функцию. Ограничительное условие может быть задано в форме ***комментария***, т. е. произвольного текста,

который в алголе-68 заключается в „словарные скобки" **co co**[1], а в паскале — в фигурные скобки { }:

funct $f \equiv$ (**int** u, v	**function** $f(u, v : integer$
co $u \neq v$ **co**) **int**:	$\{u \neq v\}) : integer;$
$(u + v)$ **div** $(u - v)$	**begin** $f \Leftarrow (u + v)$ **div** $(u - v)$ **end**

и

funct $z \equiv$ (**real** a, b	**function** $z(a, b : real$
co $a \neq 0 \wedge b \neq 0$ **co**) **real**:	$\{(a \neq 0) \wedge (b \neq 0)\}) : real;$
$a/b + b/a$	**begin** $z \Leftarrow a/b + b/a$ **end.**

Такого рода условия, ограничивающие области определения параметров, называются ***(условиями)-предохранителями***[2]. При вызове подпрограммы надо проверять, выполнены ли они.

2.3.1.4. Связанные обозначения

Обозначение данного параметра подпрограммы можно заменить на любое другое обозначение, если только эта замена непротиворечива, потому что „вовне программы" обозначения параметров никакого значения не имеют. И

$$(\mathbf{int}\ a, b)\,\mathbf{int}\colon a - b,$$

и

$$(\mathbf{int}\ b, a)\,\mathbf{int}\colon b - a$$

— это записи одной и той же подпрограммы. Говорят, что обозначения параметров являются ***связанными*** („привязанными" к данной подпрограмме) ***обозначениями***[3] (*англ.*: *bound identifier*[4]), их ***областью связывания*** служит вся подпрограмма.

2.3.2. Рекурсия

Третий и решающий принцип построения подпрограмм — ***рекурсия***. Она возникает, когда при определении подпрограммы мы (прямо или косвенно) ссылаемся на саму определяемую подпрограмму, т. е. когда подпрограмма прямо или косвенно опирается на саму себя.

---

[1] От английского commentary (комментарий). — *Прим. изд. ред.*

[2] В стандартных трансляторах для алгола-68 и паскаля их истинность, конечно же, не проверяется дополнительно, они прочитываются как голый комментарий.

[3] Применение связанных обозначений в формальных системах подробно исследовал А. Чёрч в своих работах по лямбда-исчислению (начиная с 1936 г.).

[4] Буквально связанный идентификатор. — *Прим. изд. ред.*

funct fac = (**int** n **co** $n \geq 0$ **co**) **int**:	**function** $fac(n: integer[n \geq 0]): integer$;
	begin
if $n = 0$	**if** $n = 0$
then 1	**then** $fac \Leftarrow 1$
else $n \times fac(n-1)$ **fi**	**else** $fac \Leftarrow n * fac(n-1)$
	end

Рис. 60.

Понятие рекурсии совсем не трудно для понимания, и это понимание не связано со знанием какого-то определённого формализма (ни даже со знанием какой-то определённой нотации). Все излагаемые далее примеры можно было бы описать и устно. Любой „человек с улицы" вполне в состоянии использовать рекурсию, если он столкнётся с ней в практической жизни, и уж во всяком случае в состоянии понять, о чём идёт речь, когда ему объясняют рекурсию на самом обычном языке. Примеры этому были даны в 1.6.2 ((c) и (d)).

Впрочем, здесь уместно одно предостережение. До сих пор мы могли быть уверены, что при обработке формулы, фигурирующей в теле подпрограммы, если все выполняемые операции являются всюду определёнными, никаких проблем не возникнет, в частности могли быть уверены, что число операций не превзойдёт некоторого фиксированного максимального числа. Теперь это уже не так: чтобы быть уверенным, что мы со своим „самовызовом" не устроим *circulus vitiosus*[1], мы должны для рекурсивных программ всегда проводить ***доказательство окончания***[2] (см. 2.4.3).

Для начала приведем два элементарных примера прямой рекурсии:

(a) Подпрограмма для вычисления факториала неотрицательного числа (см. рис. 60 и 61). Этот стандартный пример никак нельзя обойти. Данная подпрограмма — это просто переписанное на алголе-68 и на паскале обычное определение факториала

$$n! = \begin{cases} n \times (n-1)! & \text{при} \quad n > 0, \\ 1 & \text{при} \quad n = 0, \end{cases}$$

представляющее собой не что иное, как два (условных) равенства, характеризующих рассматриваемую функцию.

---

[1] Порочный круг (*лат.*). — *Прим. изд. ред*

[2] Известная шутка: „Как поймать стаю львов в пустыне? Надо поймать одного льва и тем самым свести задачу к более простому случаю." Доказательство окончания здесь очевидно, если каждый лев может быть пойман за конечное время.

Рис. 61. Формуляр для *fac*.

```
funct gcd ≡ (int m, int n                 | function gcd (m, n : integer
           co m > 0 ∧ n > 0 co) int :     |          {(m > 0) ∧ (n > 0)}) : integer ;
                                          | begin
  if m = n                                | if m = n
  then m                                  | then gcd ⇐ m
  else if m < n                           | else if m < n
       then gcd (n, m)                    |      then gcd ⇐ gcd (n, m)
       else gcd (m − n, n) fi fi          |      else gcd ⇐ gcd (m − n, n)
                                          | end
```

Рис. 62.

(b) Подпрограмма для вычисления наибольшего общего делителя двух положительных чисел (см. рис. 62 и 63). Это тоже часто используемый пример. Подпрограмма представляет собой формальную запись алгоритма, описанного ещё Эвклидом (Эвклид, Начала, книга 7, теорема 2).

Косвенная рекурсия может быть задана в системе подпрограмм, которые определяются „бок о бок" и *взаимно опираются друг на друга*:

(c) Пара подпрограмм[1] (*iseven*, *isodd*) для определения того, чётно или нечётно количество знаков в данной последовательности знаков (рис. 64 и 65). Подставляя, скажем, *isodd* в *iseven*, можно получить непосредственную рекурсию для *iseven* (рис. 66). Исключению *isodd* соответствует подстановка формуляра для *isodd* в формуляр для *iseven*.

(d) Примером *иерархической* рекурсивной системы подпрограмм может служить представленная на рис. 67 и 68 пара (*gcd1*, *mod*), где *mod* — подпрограмма для операции **.mod.**

[1] Ниже *iseven* и *isodd* — от even (чётный) и odd (нечётный). — *Прим. изд. ред.*

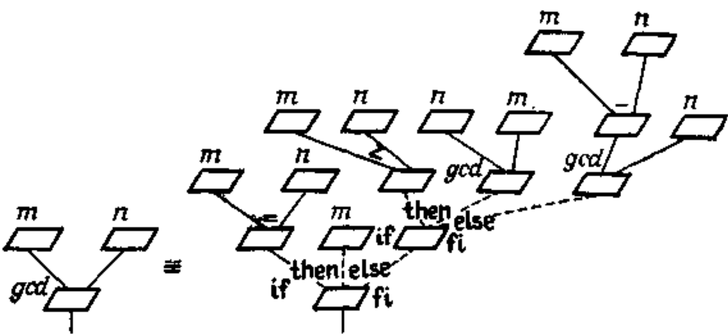


Рис. 63. Формуляр для *gcd*.

(с положительным делителем), а *gcd1* — подпрограмма для вычисления наибольшего общего делителя двух неотрицательных целых чисел. Простое исключение *mod*, с тем чтобы, скажем, получить подпрограмму *gcd* из (b), здесь уже невозможно.

(e) Следующая подпрограмма вычисляет длину последовательности знаков (см. 2.1.3.7)

funct *fac* ≡ (**int** *n* **co** $n \geq 0$ **co**) **int** ;	**function** *fac* (*n* : *integer*{$n \geq 0$}) : *integer* ;
	begin
if $n = 0$	**if** $n = 0$
then 1	**then** *fac* ⇐ 1
else $n \times fac(n-1)$ **fi**	**else** *fac* ⇐ $n * fac(n-1)$
	end

Очевидно, что с помощью этой подпрограммы функцию *iseven* из (c) можно вычислить просто как

funct *iseven* ≡ (**string** *m*) **bool** ;	**function** *iseven* (*m* : *string*) : *Boolean* ;
	begin
¬ **odd** *length* (*m*)	*iseven* ⇐ **not** *odd* (*length* (*m*))
	end

(f) Мы можем теперь сформулировать также некоторые алгоритмы из раздела 1.6.2. Для примера (d) из этого раздела будем сначала опираться как на элементарную операцию «разбей (непустую и неодноэлементную) последовательность знаков *a* в произвольном месте на две составные части *lpart* (*a*) и

```
funct iseven ≡ (string m) bool :

   if m = ◇
   then true
   else isodd (rest (m)) fi

funct isodd ≡ (string m) bool :

   if m = ◇
   then false
   else iseven (rest (m)) fi
```

```
function iseven (m: string): Boolean;
begin
if isempty (m)
then iseven ⇐ true
else iseven ⇐ isodd (rest (m))
end

function isodd (m : string) : Boolean ;
begin
if isempty (m)
then isodd ⇐ false
else isodd ⇐ iseven (rest (m))
end
```

Рис. 64.

Рис. 65. Формуляр для системы (*iseven*, *isodd*).

rpart (*a*)[1]». Тогда указанный там алгоритм можно написать (если для простоты принять, что каждая карточка в картотеке содержит один-единственный знак) в виде подпрограммы *sort*[2], представленной на рис. 69 и 70. Фигурирующая в ней подпро-

[1] От left part (левая часть) и right part (правая часть) соответственно. — *Прим. изд. ред.*

[2] От sorting (сортировка.) — *Прим. изд. ред.*

funct *iseven* = (**string** *m*) **bool**:	**function** *iseven*(*m*:*string*): *Boolean*;
	begin
if *m* = ◇	**if** *isempty*(*m*)
then true	**then** *iseven* ⇐ *true*
elsf *rest*(*m*) = ◇	**else if** *isempty*(*rest*(*m*))
then false	**then** *iseven* ⇐ *false*
else *iseven*(*rest*(*rest*(*m*))) **fi**	**else** *iseven* ⇐ *iseven*(*rest*(*rest*(*m*)))
	end

Рис. 66.

funct *mod* = (**int** *m*, **int** *n* **co** $m \geqq 0 \wedge n > 0$ **co**) **int**:	**function** *mod*(*m*, *n* : *integer* {$(m \geqq 0) \wedge (n > 0)$}) : *integer*;
	begin
if $m < n$	**if** $m < n$
then *m*	**then** *mod* ⇐ *m*
else *mod*($m - n$, *n*) **fi**	**else** *mod* ⇐ *mod*($m - n$, *n*)
	end
funct *gcd1* = (**int** *m*, **int** *n* **co** $m \geqq 0 \wedge n \geqq 0$ **co**) **int**:	**function** *gcd1*(*m*, *n* : *integer* {$(m \geqq 0)$ *and* $(n \geqq 0)$}) : *integer*;
	begin
if $n = 0$	**if** $n = 0$
then *m*	**then** *gcd1* ⇐ *m*
else *gcd1*(*n*, *mod*(*m*, *n*)) **fi**	**else** *gcd1* ⇐ *gcd1*(*n*, *mod*(*m*, *n*))
	end

Рис. 67.

грамма *merge*[1], служащая для слияния двух сортированных последовательностей в одну, представлена на рис. 71.

Отметим, что недетерминистический разбор случаев здесь отвечает характеру задачи. Так как в стандартном алголе-68 и паскале недетерминистический разбор случаев не предусмотрен, то там приходится, нарушая естественную симметрию, использовать детерминистический разбор случаев.

Остается ещё разобраться с операцией «разбей...». Это можно сделать многими разными способами, соответствующий алгоритм недетерминистичен. Разумеется, никто не запрещает производить разложение каждый раз одним и тем же определенным способом (детерминированно), например на две части $lpart(a)$ и $rpart(a)$ с $0 \leqslant length(lpart(a)) - length(rpart(a)) \leqslant 1$ („двоичная сортировка").

Можно также взять $\langle first(a)\rangle$ в качестве $lpart(a)$ в $rest(a)$ в качестве $rpart(a)$. Тогда $sort(lpart(a)) = sort(\langle first(a)\rangle)$ вырождается в $\langle first(a)\rangle$, и мы получаем „линейный" алгоритм

[1] От merge (сливать, соединять). — *Прим. изд. ред.*

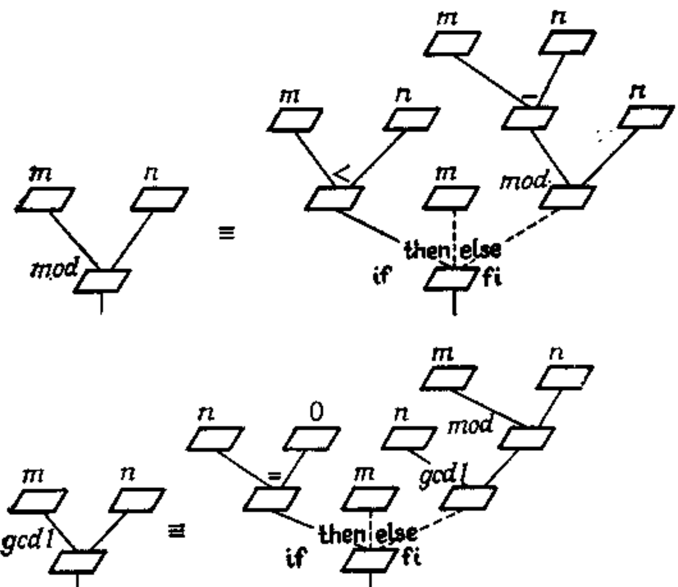


Рис. 68. Формуляр для системы (*gcd1*, *mod*).

```
funct sort ≡ (string a) string :

  if a = ◊ ∨ rest(a) = ◊
  then a
  else merge(sort(lpart(a)),
             sort(rpart(a)))
  co lpart(a) + rpart(a) = a co fi
```

```
function sort(a : string) : string ;
begin
if isempty(a) ∨ isempty(rest(a))
then sort ⇐ a
else sort ⇐ merge(sort(lpart(a)),
                  sort(rpart(a)))
{conc(lpart(a), rpart(a)) = a}
end
```

Рис. 69

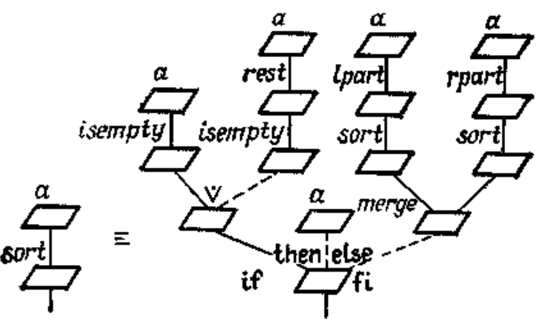


Рис. 70. Формуляр для *sort*.

```
funct merge ≡ (string u, v) string :      | function merge(u, v : string) : string ;
                                          | begin
  if u = ◇                                | if isempty(u)
  then v                                  | then merge ⇐ v
  ▯ v = ◇                                 | ▯ isempty(v)
  then u                                  | then merge ⇐ u
  else if first(u) ≤ first(v)             | else if first(u) ≤ first(v)
       then ⟨first(u)⟩ +                  |      then merge ⇐ prefix(first(u),
            merge(rest(u), v)             |                 merge(rest(u), v))
       ▯ first(u) ≥ first(v)              |      ▯ first(u) ≥ first(v)
       then ⟨first(v)⟩ +                  |      then merge ⇐ prefix(first(v),
            merge(rest(v), u)             |                 merge(rest(v), u))
                                   fi fi  | end
```

Рис. 71.

сортировки: [1]

```
funct linsort ≡ (string a) string :       | function linsort(a : string) : string ;
                                          | begin
  if a = ◇                                | if isempty(a)
  then a                                  | then linsort ⇐ a
  else insort(first(a),                   | else linsort ⇐ insort(first(a),
              linsort(rest(a))) fi        |                 linsort(rest(a)))
                                          | end
```

Алгоритм *merge* для слияния двух сортированных последовательностей содержит как частный случай алгоритм для вставки одного элемента в сортированную последовательность; он получается, если, скажем, последовательность u одноэлементна, т. е. $u = \langle x \rangle$. В этом случае подпрограмма *merge* превращается в следующую подпрограмму *insort* [2] (отметим, что $(\langle x \rangle = \Diamond) =$ **false** и $first(\langle x \rangle) = x$!):

```
funct insort ≡ (char x, string v)         | function insort(x : char; v : string) :
                              string :    |                          string ;
                                          | begin
  if v = ◇                                | if isempty(v)
  then ⟨x⟩                                | then insort ⇐ prefix(x, empty)
  elsf x ≤ first(v)                       | else if x ≤ first(v)
  then ⟨x⟩ + v                            |      then insort ⇐ prefix(x, v)
  else ⟨first(v)⟩ +                       |      else insort ⇐ prefix(first(v),
          insort(x, rest(v)) fi           |                 insort(x, rest(v))),
                                          | end
```

[1] Ниже *lin* — от linear (линейный). — *Прим. изд. ред.*

[2] От in (в). — *Прим. изд. ред.*

Теперь в *linsort* можно использовать *insort* вместо *merge*, что даёт

funct *linsort* ≡ (**string** *a*) **string** ;	**function** *linsort*(*a* : *string*) : *string* ;
	begin
if *a* = ◇	**if** *isempty*(*a*)
then *a*	**then** *linsort* ⇐ *a*
else *merge*(⟨*first*(*a*)⟩, *linsort*(*rest*(*a*))) **fi**	**else** *linsort* ⇐ *merge*(*prefix*(*first*(*a*), *empty*), *linsort*(*rest*(*a*)))
	end

Приведённый в примере (е) раздела 1.6.2 недетерминистический алгоритм вставки является более общим. Он приводит к следующей подпрограмме:

funct *insort* ≡ (**char** *x*, **string** *v*) **string** ;	**function** *insort*(*x* : *char*; *v* : *string*) : *string* ;
	begin
if *v* = ◇	**if** *isempty*(*v*)
then ⟨*x*⟩	**then** *insort* ⇐ *prefix*(*x*, *empty*)
elsf *x* < *first*(*rpart*(*v*))	**else if** *x* < *first*(*rpart*(*v*))
then *insort*(*x*, *lpart*(*v*)) + *rpart*(*v*)	**then** *insort* ⇐ *conc*(*insort*(*x*, *lpart*(*v*)), *rpart*(*v*))
else *lpart*(*v*) + *insort*(*x*, *rpart*(*v*))	**else** *insort* ⇐ *conc*(*lpart*(*v*), *insort*(*x*, *rpart*(*v*)))
fi	**end**

Снова можно получить „линейную" редакцию алгоритма, взяв ⟨*first*(*v*)⟩ в качестве *lpart*(*v*) и *rest*(*v*) в качестве *rpart* (*v*); она аналогична вышеприведенной линейной редакции.

(g) Некоторые смешанные операции, о которых говорилось в 2.1.3.7, могут быть теперь введены рекурсивным образом; например, операцию выбора *i*-го знака из данной последовательности можно определить так: [1]

funct *sel* ≡ (**string** *a*, **int** *i* **co** $1 \leq i \wedge i \leq length(a)$ **co**) **char** :	**function** *sel*(*a* : *string* ; *i* : *integer* {(1 ≤ *i*) **and** (*i* ≤ *length*(*a*))}) : *char* ;
	begin
if *i* = 1	**if** *i* = 1
then *first*(*a*)	**then** *sel* ⇐ *first*(*a*)
▯ *i* > 1	▯ *i* > 1
then *sel*(*rest*(*a*), *i* − 1) **fi**	**then** *sel* ⇐ *sel*(*rest*(*a*), *i* − 1)
	end

[1] Ниже *sel* — от select (выбирать). — *Прим. изд. ред.*

Отметим, что в алголе-68 вместо *sel*(*a*, *i*) используется обозначение *a*[*i*].

2.3.3. Рекурсивная машина обработки формуляров

Ход исполнения подпрограммы определён отвечающим ей формуляром с точностью до *совместности*. Если в самом формуляре снова в качестве операции встречается подпрограмма[1], то необходимо создать формуляр этой подпрограммы (***вызов подпрограммы***, или ***обращение к подпрограмме***), а затем результат её исполнения передать обратно. Это относится и к рекурсивно определённым подпрограммам — с той лишь особенностью, что при таких рекурсивных обращениях требуются новые экземпляры формуляра самой исходной подпрограммы.

При каждом вызове в новый экземпляр формуляра прежде всего заносятся слева соответствующие значения аргументов (*call by value*[2]). Каждый такой формуляр называют ***воплощением*** подпрограммы; для обозримости можно в ходе вычисления нумеровать все воплощения и соответствующие вызовы.

В случае рекурсии особенно важно, чтобы в результате разбора случаев реализовался выбор, который сокращает работу; после того как обозначения параметров заменены значениями, проставленными слева на формуляре, как можно раньше вычисляются и в основном формуляре, и во всех его последующих воплощениях значения условий и затем исключаются недопустимые ветви. Рекурсия заканчивается на воплощениях, в которых нет больше ветвей, содержащих рекурсивные вызовы. Процесс вычисления ***завершается*** (для заданного набора значений параметров), если он требует лишь конечного числа воплощений.

Деятельность человека, который работает с формулярами указанным образом, может быть очевидным путём механизирована. Так мы приходим к понятию (мысленной) рекурсивной машины — ***машины обработки формуляров***, в которой попрежнему сохраняется полная свобода вычислений. Отметим, что новый экземпляр формуляра создаётся и тогда, когда те же самые значения уже ранее встречались,— машина обработки формуляров не учитывает (на описанном здесь уровне) возможность многократного использования однажды полученного результата.

В частности, упомянутое выше исключение недопустимых ветвей заведомо возможно, если в условиях нет рекурсивных

---

[1] Для базовых операций, т. е. для элементарных операций рассматриваемых вычислительных структур, никакого формуляра не требуется.

[2] Вызов значением (*англ.*). — *Прим. изд. ред.*

вызовов. Ещё нагляднее случай ***линейной рекурсии***, при которой, кроме того, в отдельных ветвях разбора случаев рекурсивный вызов встречается не более одного раза; тогда при каждом воплощении порождается не более одного нового воплощения. Впрочем, почти все ранее изложенные примеры попадают в этот класс.

Для подпрограммы *fac* из 2.3.2 машина обработки формуляров работает, как показано на рис. 72. Типичным для рекурсии является „задерживание" вычислений, откладывание их „на потом",— лишь когда на воплощении $fac^{(3)}$ заканчивается рекурсия, становятся выполнимыми и выполняются вычисления, отложенные на потом [1] в $fac^{(2)}$, $fac^{(1)}$ и $fac^{(0)}$; окончательный результат поставляет основной формуляр $fac^{(0)}$. В частном случае выполнение отложенных операций может свестись к простой передаче результатов отдельных воплощений, как это показано на рис. 73 для примера вызова *gcd1*(15, 9), см. 2.3.2. Такие вызовы называются ***гладкими*** (или ***регулярными***). Если в линейной рекурсии имеются только гладкие вызовы, то говорят о ***повторительной рекурсии***.

В линейно-рекурсивных программах — если отвлечься от уже упоминавшейся совместности для формуляра — порядок, в котором порождаются требуемые воплощения, определён однозначно. В общем случае это не всегда так: если в какой-то ветви имеется несколько вызовов, то при определённых условиях возможны различные порядки порождения и даже параллельная работа.

Это отражено на рис. 74 для примера программы *sort* из 2.3.2, в предположении что выполняется двоичная сортировка. Здесь для последовательности знаков длины 2^n получается ровно $2^{n+1}-1$ воплощений, но всего лишь $n+1$ тактов, при линейной же сортировке требуется 2^n воплощений, но зато 2^n тактов. Также и в отношении расхода времени на сравнение знаков, которое необходимо проводить при исполнении подпрограммы *merge*, двоичная сортировка выгодней: она требует максимум $(n-1)\times 2+1$ сравнений, в то время как линейная сортировка — до $2^{n-1}\times(2^n-1)$ сравнений.

Во всех предыдущих примерах молчаливо принималось очевидное правило: начинать новое воплощение, а значит и создавать новый формуляр только тогда, когда все аргументы подготовлены. Это всегда будет предполагаться и в дальнейшем, т. е. мы принимаем следующий принцип:

Определённая посредством описания подпрограммы операция всегда рассматривается как строгая.

---

[1] На рис. 59 ромбики, отвечающие задержанным операциям, заштрихованы.

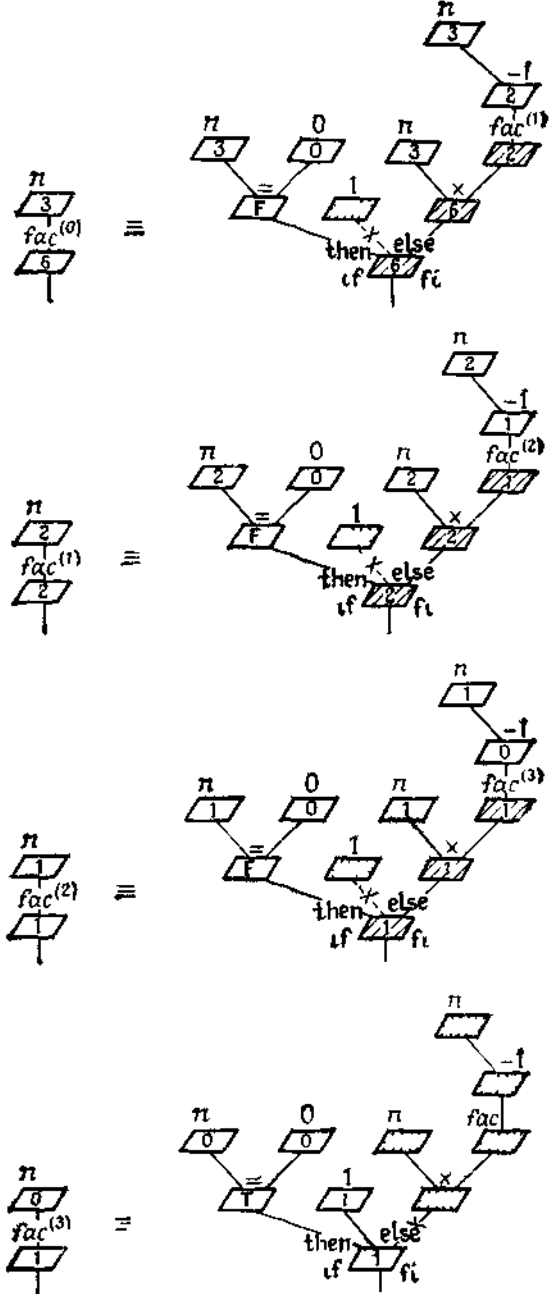


Рис 72 Работа машины обработки формуляров при вызове $fac(3)$.

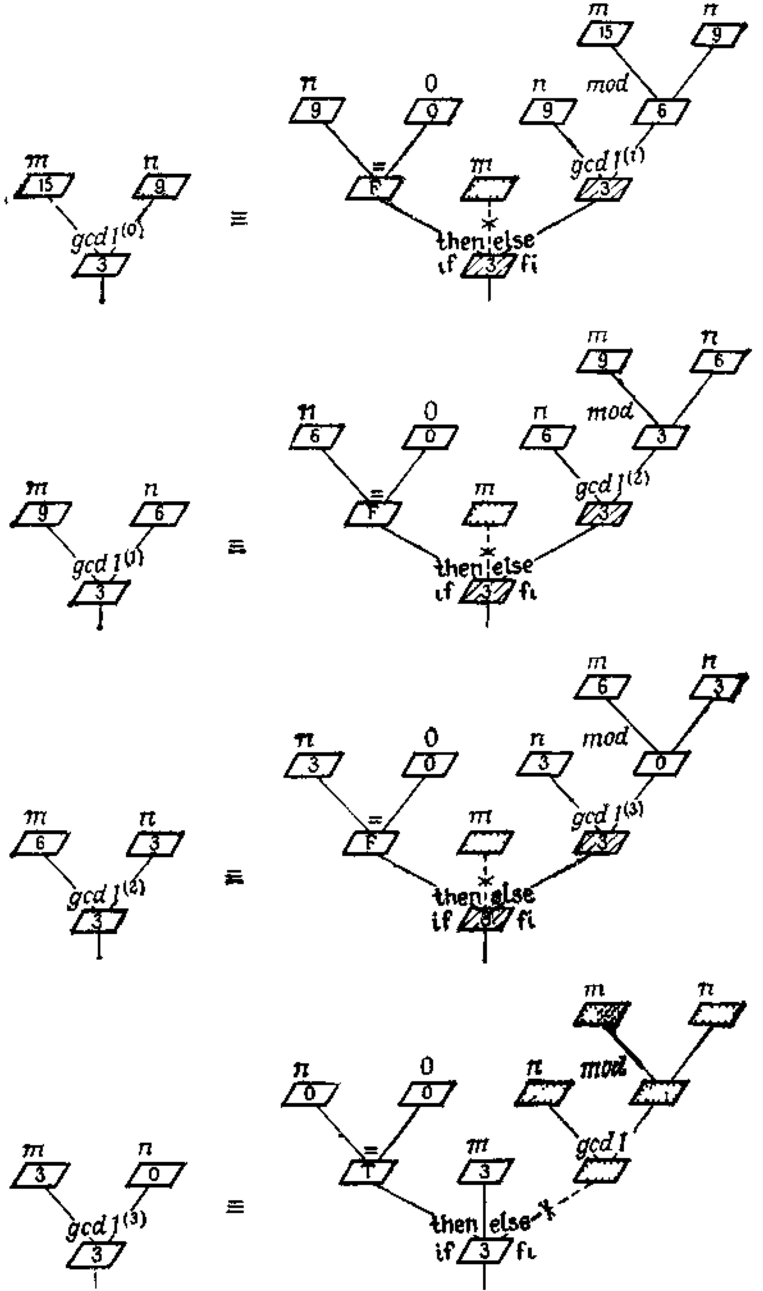


Рис 73 Работа машины обработки формуляров при вызове *gcd 1* (15, 9).

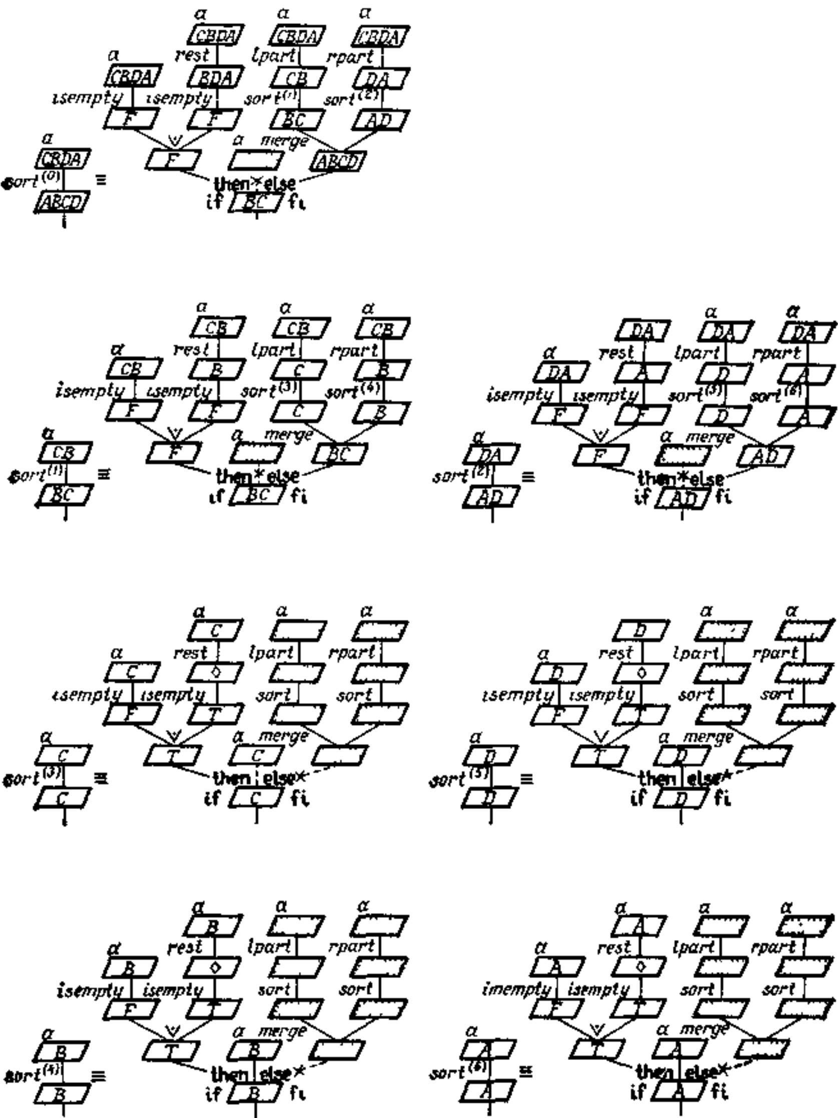
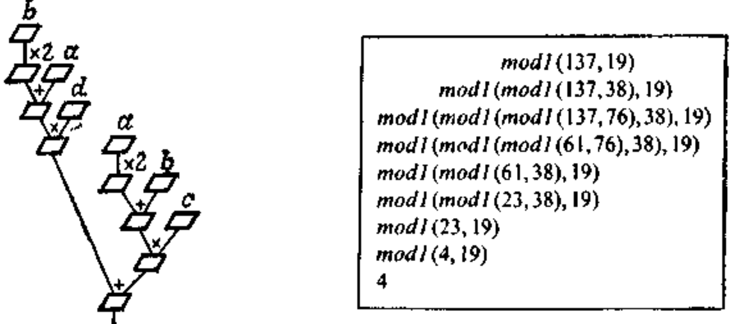


Рис. 74. Работа машины обработки формуляров при вызове *sort*('CBDA') (обработка слева направо).

В случае иерархических вызовов, как например в приводимой ниже подпрограмме *mod1* („быстром“ варианте подпро-



mod1(137, 19)
mod1(*mod1*(137, 38), 19)
mod1(*mod1*(*mod1*(137, 76), 38), 19)
mod1(*mod1*(*mod1*(61, 76), 38), 19)
mod1(*mod1*(61, 38), 19)
mod1(*mod1*(23, 38), 19)
mod1(23, 19)
mod1(4, 19)
4

Рис. 75. Вычислительный формуляр и принцип магазина.

Рис. 76. Ход вычислений при вызове *mod1* (137, 19).

граммы *mod* из 2.3.2) для вычисления остатка при целочисленном делении, это означает обработку „изнутри наружу“:

```
funct mod1 ≡ (int m, int n co m ≧ 0 ∧ n > 0 co) int :
        if m ≧ 2×n then mod1(mod1(m, 2×n), n)
      elsf m ≧ n   then mod1(m−n, n)
                   else m                    fi.
```

В совместных ситуациях всё ещё сохраняется свобода вычислений. В конкретных версиях машины обработки формуляров от неё часто отказываются. Если, скажем, дополнительно потребовать, чтобы в случае совместности обработка производилась последовательно „слева направо“, то мы получаем „принцип магазина“[1] Бауэра и Замельзона (1957 г.)[2] для последовательного вычислительного процесса:

В ходе вычисления считываемые слева направо операции (как примитивные, так и описанные) откладываются, если их аргументы еще не (полностью) подготовлены, и выполняются, как только их аргументы полностью подготовлены,— „откладываются по необходимости, наполняются по возможности“. [3]

---

[1] В оригинале „Kellerprinzip“ (Keller — погреб, подвал). Имеется в виду магазин винтовки (патрон, помещённый туда последним, выстреливается первым). См. также 3.7.2. — *Прим. изд. ред.*

[2] Патенты за номерами 3047228 (США) и 1094019 (ФРГ).

[3] Это правило обработки эквивалентно leftmost-innermost computation rule [правилу лево-внутреннего вычисления; см. 8.2.6. — *Изд. ред.*] Морриса (1968 г.).

Получаемый таким образом порядок выполнения формулы (∗) из 2.2.2 показан на рис. 75. Процесс вычисления при вызове *mod1* (137, 19) представлен на рис. 76.

Проведённые ранее рассмотрения без труда переносятся на системы (рекурсивных) подпрограмм, например на систему (*iseven*, *isodd*) из 2.3.2, в которой мы имеем дело с косвенной рекурсией. Для рекурсивных систем вроде (*gcd1*, *mod*) речь идёт, очевидно, о рекурсии внутри рекурсии.

Об особой роли частного случая повторительной рекурсии мы ещё поговорим в гл. 3.

2.4. О технике рекурсивного программирования

2.4.1. Как приходят к рекурсивным подпрограммам?*

Нет готовых рецептов, как для данной произвольной проблемы получить (рекурсивный) алгоритм её решения. Однако некоторые рекомендации имеются.

2.4.1.1. Органически рекурсивные определения

Многие рекурсивные подпрограммы являются точным „слепком“ с соответствующего определения. Это верно, скажем, в отношении подпрограммы *fac*. Иначе обстоит дело с классическим определением возведения в степень. Оно попадает под общую схему n-кратно итерированной операции ρ, где $.\rho.$ — любая внутренняя двуместная ассоциативная операция над произвольным сортом λ, соотв. λ (область значений параметра n ограничивается положительными натуральными числами):

funct *iter* ≡ (λ *a*, **int** *n* **co** *n* > 0 **co**) λ :
  if *n* = 1
  then *a*
  else *a* ρ *iter*(*a*, *n* − 1) **fi**
	function *iter*(*a* : λ; *n* : *integer* {*n* > 0}) : λ ;
	  begin if *n* = 1
	    then *iter* ⇐ *a*
	    else *iter* ⇐ *a* ρ *iter*(*a*, *n* − 1)
	  end

2.4.1.2. Извлечение рекурсии из постановки задачи

В большинстве случаев, однако, рассматриваемая задача не является алгоритмически сформулированной. Поэтому пытаются прийти к (рекурсивному) алгоритму, сводя общую задачу к „более простым“ задачам того же рода, см 1.6.3 и 2.3.2.

---

* Этот раздел можно пропустить.

Точнее говоря, речь идёт о получении достаточного числа (условных) уравнений, определяющих искомую функцию.

Иногда непосредственно ясно, как это сделать, но чаще всего требуется интуиция. Например, для решения задачи нахождения наибольшего общего делителя двух (положительных) чисел можно привлечь следующий математический результат: если a — большее из двух чисел a и b, то пары a, b и $a - b$, b имеют одни и те же делители, а значит, один и тот же наибольший общий делитель. Отсюда непосредственно следует алгоритм *gcd* из 2.3.2(b).

Оправдывая одно замечание, сделанное в 2.1.3.1, покажем здесь также, как можно определить сложение целых чисел, опираясь на операции перехода к следующему и предыдущему числам. При этом одновременно будет охвачено и вычитание, ибо прибавление отрицательного числа есть вычитание соответствующего положительного.

В силу законов ассоциативности и коммутативности имеем $(a+1)+(b-1) = a+b$. Поэтому операцию сложения можно ввести так:

funct *plus* ≡ (**int** *a*, *b*) **int**:	**function** *plus*(*a*, *b* : *integer*) : *integer*;
	begin
if $b = 0$	**if** $b = 0$
then *a*	**then** *plus* ⇐ *a*
▯ $b > 0$	▯ $b > 0$
then *plus*$(a+1, b-1)$	**then** *plus* ⇐ *plus*(*succ*(*a*), *pred*(*b*))
▯ $b < 0$	▯ $b < 0$
then *plus*$(a-1, b+1)$	**then** *plus* ⇐ *plus*(*pred*(*a*), *succ*(*b*))
fi	**end**

Если ограничиться сложением натуральных чисел, то третья ветвь отпадает.

2.4.1.3. Вложение

Если не удаётся извлечь рекурсию из самой постановки задачи, то часто оказывается полезным обобщить задачу (например, введя дополнительные параметры); подходящее обобщение позволяет усмотреть возможность рекурсии, а возвращаясь к частному случаю, мы получаем алгоритм для первоначальной задачи.

Классический пример применения такого приёма ***вложения*** дал Маккарти в 1962 г. Исходная задача была[1]

isprim(*n*): «Установи, является ли заданное положительное натуральное число *n* простым.»

---

[1] Ниже *prim* — от prime number (простое число). — *Прим. изд. ред.*

При этом предполагается, что известно определение простого числа:

> «Натуральное число n называется простым, если оно больше 1 и не делится ни на одно число, большее или равное 2 и меньшее n.»

Удачным обобщением будет такая задача (в которой вводится один дополнительный параметр):

> *ispr* (n, m): «Установи, верно ли, что заданное натуральное число n не делится ни на одно число, большее или равное m и меньшее n.»

(Не уменьшая общности, можно считать, что $2 \leqslant m \leqslant n$.)

Но для этой новой задачи ясно, что *ispr* (n, m) истинно, во-первых, если $m = n$, и, во-вторых, если истинно *ispr* $(n, m + 1)$ и n не делится на m, и мы приходим к подпрограмме

funct *ispr* ≡ (**int** *n*, *m* **co** 2 ≦ *m* ≦ *n* **co**) **bool** : **if** *m* = *n* **then true** **else** (*n* **mod** *m* ≠ 0) ∧ *ispr*(*n*, *m* + 1) **fi**	**function** *ispr*(*n*, *m* : *integer* {2 ≦ *m* ≦ *n*}) : *Boolean* ; **begin** **if** *m* = *n* **then** *ispr* ⇐ *true* **else** *ispr* ⇐ (*n* **mod** *m* ≠ 0) **and** *ispr*(*n*, *m* + 1) **end**

В этой подпрограмме содержится еще заключительное „захлопывание" всех „задержанных"конъюнкций. От него можно избавиться и обеспечить за счёт этого более раннее окончание, заменив конъюнкцию на альтернативу (2.2.3.4) и перейдя тем самым к повторительной рекурсии:

funct *ispr* ≡ (**int** *n*, *m* **co** 2 ≦ *m* ≦ *n* **co**) **bool** : **if** *m* = *n* **then true** **else if** *n* **mod** *m* = 0 **then false** **else** *ispr*(*n*, *m* + 1) **fi fi**	**function** *ispr*(*n*, *m* : *integer* {2 ≦ *m* ≦ *n*}) : *Boolean* ; **begin** **if** *m* = *n* **then** *ispr* ⇐ *true* **else if** *n* **mod** *m* = 0 **then** *ispr* ⇐ *false* **else** *ispr* ⇐ *ispr*(*n*, *m* + 1) **end**

В качестве частного случая получаем интересующую нас подпрограмму:

funct *isprim* ≡ (**int** *n* **co** $n \geqq 1$ **co**) **bool**:	**function** *isprim* (*n* : *integer* {$n \geqq 1$}) : *Boolean*;
	begin
if $n = 1$	**if** $n = 1$
then false	**then** *isprim* ⇐ *false*
else *ispr*(*n*, 2) **fi**	**else** *isprim* ⇐ *ispr*(*n*, 2)
	end

Впрочем, подпрограмму *ispr* можно ещё усовершенствовать, заменив условие $m = n$ на $m \uparrow 2 > n$. Эту подпрограмму обозначим *ispr'*. Если верно *ispr'* (*n*, 2), то у *n* нет делителя *t*, удовлетворяющего условию $2 \leqslant t < m$, где *m* — наименьшее число,

funct *fac* ≡ (**int** *n* **co** $n \geqq 0$ **co**) **int**:	**function** *fac* (*n* : *integer* {$n \geqq 0$}) : *integer*;
	begin
facr(1, *n*)	*fac* ⇐ *facr*(1, *n*)
	end
funct *facr* ≡ (**int** *y*, *n* **co** $n \geqq 0$ **co**) **int**:	**function** *facr* (*y*, *n* : *integer* {$n \geqq 0$}) : *integer*;
	begin
if $n = 0$	**if** $n = 0$
then *y*	**then** *facr* ⇐ *y*
else *facr*($y \times n$, $n - 1$) **fi**	**else** *facr* ⇐ *facr*(*y***n*, $n - 1$)
	end

Рис. 77.

для которого $m^2 > n$,— в противном случае мы имели бы окончание с выдачей **false**. Далее, если бы у *n* был делитель t_1, $m \leqslant t_1 < n$, то был бы и делитель t_2, $1 < t_2 < m$. Следовательно, у *n* нет делителя *t*, $2 \leqslant t < n$.

С помощью вложения иногда удаётся для уже имеющейся рекурсии получить п о в т о р и т е л ь н у ю редакцию. Если определить двуместную функцию[1] *facr* как $facr(y, n) = y \times fac(n)$, т. е., грубо говоря (при $n > 0$), как $y \times n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$, то, ввиду ассоциативности умножения, для $n > 0$

$$facr(y, n) = facr(y \times n, n-1), \quad facr(y, 0) = y,$$

и мы приходим к повторительной системе, представленной на рис. 77.

[1] Ниже *r* (в *facr*) — от repetitive (повторительный). — *Прим. изд. ред.*

2.4.1.4. Метод родственных задач

Как, однако, прийти к паре (*iseven*, *isodd*) рекурсивных подпрограмм из 2.3.2? Вместо того чтобы вкладывать задачу в более общую, при использовании ***метода родственных задач*** к ней „приставляют" одну или несколько задач таким образом, чтобы подпрограммы из возникающей системы опирались друг на друга.

Прежде всего задачу (c) из 2.3.2 можно сформулировать словами так:

> *iseven*(*m*): «Установи, содержит ли данная последовательность знаков *m* чётное число знаков.»

При этом справедливо очевидное утверждение:

> «Число знаков в последовательности чётно тогда и только тогда, когда все знаки могут быть сгруппированы в пары.»

Если теперь ввести родственную задачу

> *isodd*(*m*): «Установи, содержит ли данная последовательность знаков *m* нечётное число знаков.»,

то обнаруживается взаимосвязь двух задач: непустая последовательность знаков содержит нечётное число знаков в том и только том случае, если после удаления одного знака она содержит чётное число знаков,— и чётное, если после удаления одного знака остаётся нечётное число знаков. Далее, пустая последовательность знаков тоже может сгруппироваться в (пустое) множество пар и, следовательно, содержит чётное, а стало быть, не нечётное, число пар. Мы приходим к таким формулировкам:

> *iseven*(*m*): «Если последовательность *m* пуста, то: да. В противном случае: Установи, содержит ли укороченная на один знак последовательность нечётное число знаков.»
> *isodd*(*m*): «Если последовательность *m* пуста, то: нет. В противном случае: Установи, содержит ли укороченная на один знак последовательность чётное число знаков.»

Из этих словесных формулировок вытекают основанные на базовой операции *rest* из 2.1.3.4 формулировки, данные в 2.3.2(c).

Рассмотрим сходную задачу[1]

> *ispos*(*m*): «Установи, чётно ли число минусов в заданной последовательности *m* знаков плюс и минус».

---

[1] Ниже *pos* — от positive (положительный), *neg* — от negative (отрицательный). — *Прим. изд. ред.*

Она возникает при определении знака произведения по знакам отдельных множителей. Разумеется, имеет значение лишь число

funct *ispos* ≡ (**string** *m*) **bool** :	**function** *ispos* (*m* : *string*) : *Boolean* ;
	begin
if *m* = ◇	**if** *isempty* (*m*)
then true	**then** *ispos* ⇐ *true*
else if *first* (*m*) = "−"	**else if** *first* (*m*) = '−'
then *isneg* (*rest* (*m*))	**then** *ispos* ⇐ *isneg* (*rest* (*m*))
▯ *first* (*m*) = "+"	▯ *first* (*m*) = '+'
then *ispos* (*rest* (*m*)) **fi fi**	**then** *ispos* ⇐ *ispos* (*rest* (*m*))
	end
funct *isneg* ≡ (**string** *m*) **bool** :	**function** *isneg* (*m* : *string*) : *Boolean* ;
	begin
if *m* = ◇	**if** *isempty* (*m*)
then false	**then** *isneg* ⇐ *false*
else if *first* (*m*) = "−"	**else if** *first* (*m*) = '−'
then *ispos* (*rest* (*m*))	**then** *isneg* ⇐ *ispos* (*rest* (*m*))
▯ *first* (*m*) = "+"	▯ *first* (*m*) = '+'
then *isneg* (*rest* (*m*)) **fi fi**	**then** *isneg* ⇐ *isneg* (*rest* (*m*))
	end

Рис. 78.

знаков минус. Поэтому мы приходим к следующей формулировке:

> *ispos* (*m*): «Если последовательность *m* пуста, то ответ — „да". В противном случае, если на первом месте стоит знак минус: Установи, содержит ли укороченная на первый знак последовательность нечётное число минусов; если же нет, то: Установи, содержит ли укороченный на первый знак ряд чётное число минусов.»

И аналогично для функции *isneg*. В результате получается система подпрограмм, представленная на рис. 78. Здесь уже нельзя исключить одну из подпрограмм путём подстановки; речь идёт о паре с (неустранимо) ***перекрёстной*** рекурсией.

2.4.1.5. Использование характеристических свойств

Часто бывает полезно сначала выявить какие-нибудь характеристические свойства поставленной задачи. Например, для

задачи о том, является ли a подсловом слова b (см. 1.6.2(g)), характеристическим предикатом может служить

«Существуют последовательности u, v, такие что $u + a + v = b$.»

Разбирая отдельно случаи, когда последовательность u пуста и когда она непуста, приходим к рекурсивному алгоритму, опирающемуся на вспомогательную задачу

«Установи, является ли последовательность a началом последовательности b.»

Иногда бывает целесообразно переформулировать характеристическое свойство задачи так, чтобы из него можно было извлечь какой-либо (другой) алгоритм.

Так, в задаче: для заданного натурального числа $n \geqslant 1$ определить (единственное) натуральное число[1] $ld(n)$, такое что

$$2^{ld(n)-1} \leqslant n < 2^{ld(n)},$$

необходимо выполнение предиката $P(ld(n), n)$, где P задаётся следующим образом:

$$P(a, n) \equiv 2^{a-1} \leqslant n < 2^{a}.$$

Но для этого характеристического предиката $P(a, n)$ имеет место очевидная рекурсия

$$P(a,n) = \begin{cases} a = 1 & \text{при} \quad n = 1, \\ P(a-1, n \textbf{ div } 2) & \text{при} \quad n > 1, \end{cases}$$

а значит, и $ld(n)$ допускает рекурсивное определение

$$ld(n) = \begin{cases} 1 & \text{при} \quad n = 1, \\ ld(n \textbf{ div } 2) + 1 & \text{при} \quad n > 1, \end{cases}$$

из которого получаем алгоритм:

funct $ld \equiv$ (**int** n **co** $n \geq 1$ **co**) **int**:	**function** $ld(n: integer\{n \geq 1\}): integer$;
	begin
if $n = 1$	**if** $n = 1$
then 1	**then** $ld \Leftarrow 1$
else $ld(n$ **div** $2) + 1$ **fi**	**else** $ld \Leftarrow ld(n$ **div** $2) + 1$
	end

Обычно характеристические свойства определяют алгоритм не однозначно. Это особенно ярко видно на примере задачи

[1] Ниже ld — от латинского logarithmus dualis (двоичный логарифм). — *Прим. изд. ред.*

о сортировке из 1.6.2, где имеется большая свобода в выборе *lpart* и *rpart*. Характеристическое свойство здесь таково: новая последовательность знаков должна быть **перестановкой** исходной и удовлетворять предикату *issorted*[1], который можно задать, например, следующим способом:

```
funct issorted ≡ (string a) bool :

  if length(a) ≤ 1
  then true
  else if first(a) > first(rest(a))
       then false
       else issorted(rest(a)) fi fi
```

```
function issorted(a:string):Boolean;
begin
if length(a) ≤ 1
then issorted ⇐ true
else if first(a) > first(rest(a))
     then issorted ⇐ false
     else issorted ⇐ issorted(rest(a))
end
```

С использованием *lpart* и *rpart* это может быть записано в эквивалентном, но более общем виде:

```
funct issorted1 ≡ (string a) bool :


  if length(a) ≤ 1
  then true
  else if last(lpart(a)) > first(rpart(a))
       then false
       else issorted1(lpart(a)) ∧
                issorted1(rpart(a)) fi fi
```

```
function issorted1(a:string):
                           Boolean;
begin
if length(a) ≤ 1
then issorted1 ⇐ true
else if last(lpart(a)) > first(rpart(a))
     then issorted1 ⇐ false
     else issorted1 ⇐
              issorted1(lpart(a)) and
                  issorted1(rpart(a))
end
```

Отсюда уже легче прийти к решению, приведённому в 2.3.2(f).

2.4.1.6. Обращение

Один из самых распространённых типов задачи — задачи на обращение функций. Мы обсудим в заключение (на примере с двоичными списками, см. 2.1.3.5) вопрос об обращении функции, заданной посредством алгоритма.

Размеченные бинарные деревья можно понимать как кодовые деревья двоичных кодирований, удовлетворяющих условию Фано. Для декодирования последовательности знаков O, L

---

[1] От sorted (сортированный). — *Прим. изд. ред.*

имеем алгоритм [1]

```
funct decod ≡ (lisp s, bits a
  co a – элемент кода, опре-
  деляемого списком s co) char:

  if isatom(s) ∨ a = ◇
  then val(s)
  else if first(a) = O
    then decod(car(s), rest(a))
    ▯ first(a) = L
    then decod(cdr(s), rest(a))  fi fi
```

```
function decod(s: lisp; a: bitstring
      {a – элемент кода, опре-
      деляемого списком s}): char;
begin
  if isatom(s) or isempty(a)
  then decod ⇐ val(s)
  else if first(a) = O
    then decod ⇐ decod(car(s), rest(a))
    ▯ first(a) = L
    then decod ⇐ decod(cdr(s), rest(a))
end
```

Его обращение определяется так:

```
funct cod ≡ (lisp s, char x) bits:
  «любое двоичное слово a,
   такое что
  decod(s, a) = x».
```

```
function cod(s: lisp; x: char): bitstring;
  «любое двоичное слово a,
   такое что   decod(s, a) = x».
```

Решением служит

```
funct cod ≡ (lisp s, char x
      co contains(s, x) co) bits:

  if isatom(s)
  then ◇
  else if contains(car(s), x)
    then ⟨O⟩ + cod(car(s), x)
    ▯ contains(cdr(s), x)
    then ⟨L⟩ + cod(cdr(s), x) fi fi
```

```
function cod(s: lisp; x: char
          {contains(s, x)}): bitstring;
begin
  if isatom(s)
  then cod ⇐ empty
  else if contains(car(s), x)
    then cod ⇐ prefix(O, cod(car(s), x))
    ▯ contains(cdr(s), x)
    then cod ⇐ prefix(L, cod(cdr(s), x))
end
```

где

```
funct contains ≡ (lisp s, char x)
                          bool:

  if isatom(s)
  then x = val(s)
  else contains(car(s), x) ∨
       contains(cdr(s), x) fi
```

```
function contains(s: lisp; x: char):
                          Boolean;
begin
  if isatom(s)
  then contains ⇐ x = val(s)
  else contains ⇐ contains(car(s), x) or
                  contains(cdr(s), x)
end
```

[1] Ниже *decod*, *cod* и *contains* — соответственно от decode (декодировать), code (кодировать) и contains (содержит). — *Прим. изд. ред.*

Если „проиграть" этот алгоритм на машине обработки формуляров, то мы увидим, что в (каскадной) рекурсии для *contains* повторяются вызовы вычислений при одних и тех же значениях аргумента. Чтобы избежать этого, надо построить полную кодовую таблицу. Это особенно выгодно, если имеется много вызовов функции *cod* с одним и тем же значением аргумента *s* (*частичное вычисление*, см. 2.6.3).

2.4.2. Как доказывать свойства алгоритмов?

Утверждения о свойствах алгоритмов весьма важны для понимания их сути. Так, подпрограммы *merge*, *gcd* и .+. обладают тем свойством, что они допускают перестановку аргументов, а значит задают коммутативные двуместные операции. Доказать этот факт, исходя из рекурсивного определения, стоит значительно больших усилий.

В случае подпрограммы *merge* из 2.3.2(f) указанное свойство устанавливается просто. Запись этой подпрограммы симметрична относительно *u* и *v*, при перестановке *u* и *v* она переходит сама в себя, поскольку порядок отдельных ветвей при охраняемом разборе случаев не имеет значения. Здесь интересующее нас свойство видно с одного взгляда на подпрограмму.

В случае подпрограммы *gcd1* из 2.3.2(d) необходимо сначала преобразовать подпрограмму. Нам достаточно показать, что для $m < n$ справедливо равенство $gcd1(m, n) = gcd1(n, m)$. Но из данного в 2.3.2(d) определения подпрограммы *mod* следует, что

$$m < n \Rightarrow mod(m,n) = m,$$

поэтому

$$gcd1(m,n) = (\textbf{if } n = 0 \textbf{ then } m \textbf{ else } gcd1(n,m) \textbf{ fi}).$$

Тем самым для $n > 0$ доказано, что $gcd1(m, n) = gcd1(n, m)$. Чтобы установить равенство $gcd1(m, 0) = gcd1(0, m)$, достаточно рассмотреть случай $m \neq 0$. В этом случае, однако, по определению, $gcd1(0, m) = gcd1(m, mod(0, m))$, а $mod(0, m) = 0$.

Наконец, в случае подпрограммы *plus* из 2.4.1.2 доказать коммутативность такими преобразованиями нельзя, здесь необходимо прибегнуть к доказательству по индукции (Сколем, 1923 г.).

Часто при решении задач исходят из характеристических свойств искомого алгоритмического решения (см. 2.4.1.5). Если надо убедиться, что данная (рекурсивная) подпрограмма является решением, то это делают, выводя из неё все требуемые свойства. Например, подпрограмма *mod* из 2.3.2(c) обладает

(характеристическими) свойствами

$$0 \leqslant mod(a,b) < b,$$
$$\exists \ \text{int}\ q : a = q \times b + mod(a,b)^1.$$

Однако доказать это не проще, чем прямо вывести из этих свойств, что

$$mod(a,b) = \begin{cases} mod(a-b,b) & \text{при} \quad a \geqslant b, \\ a & \text{при} \quad a < b. \end{cases}$$

Отсюда уже сразу можно извлечь рекурсию, применяемую в подпрограмме *mod*,— остаётся только провести доказательство окончания рекурсии.

Утверждения о некотором алгоритме иногда легче вывести из его характеристических свойств, чем из самого алгоритма.

Часто прямо из характеристического предиката можно получить свойства, которые имеют место для всякого (корректного) алгоритма, отвечающего этому предикату. При этом детали характеристического предиката могут быть совершенно неважны. Проиллюстрируем это на следующем ярком примере.

Пусть для произвольного сорта λ отображение f задано так:

funct $f = (\lambda a)\lambda$: «то x сорта λ, для которого $x\rho a = b$»,

где b — произвольный *фиксированный элемент* из λ и $.\rho.$ — двуместная операция над λ, которая ассоциативна и однозначно разрешима[2]. В таком случае

$$f(f(a))\rho b = b\rho a.$$

Следовательно, f инволютивно, если b коммутирует со всеми элементами из λ

Аналогичным образом, достаточно знать (доказательство этого факта, правда, нетривиально), что подпрограмма *sort* (см. 2.3.2(f)) отображает все перестановки некоторой последовательности знаков в одну и ту же последовательность, чтобы доказать, что

$$sort(sort(a)) = sort(a),$$

т. е. что операция сортировки является идемпотентной. (Если сортировка проводится лишь по некоторому признаку, то могут встретиться различные карточки с одинаковым признаком; при недетерминистическом выполнении подпрограммы *sort* окончательный порядок этих карточек не регламентирован, так что в этом случае идемпотентности нет.)

---

[1] Читается: „существует целое число q, такое что. .“.

[2] В том смысле, что однозначно разрешимо относительно x уравнение $x\rho a = b$. — *Прим. изд. ред.*

2.4.3. *Некоторые замечания об окончании и о роли предохранителей и стражей*

„Во всех делах твоих помни о конце твоём."

Книга премудростей Иисуса сына Сирахова, 7, 36

2.4.3.1

Приведённые выше примеры показывают, что нельзя рассчитывать на окончание рекурсивного алгоритма как на что-то само собой разумеющееся. Впрочем, никто и не ожидает, чтобы закончились алгоритм вычисления факториала для отрицательного числа или алгоритм нахождения остатка при целочисленном делении на нуль. В том что алгоритм здесь не заканчивается, гораздо больше смысла, чем если бы он выдал какое-нибудь значение, не имеющее никакого отношения к поставленной задаче.

Чтобы указать область окончания алгоритма, в заголовок подпрограммы вводят соответствующий предохранитель (см. 2.3.1.3) и таким образом приходят к всюду определённой функции. Если при этом следить, чтобы соблюдение условия-предохранителя всегда обеспечивалось соответствующим стражем, то отпадает необходимость в „сообщениях об ошибке".

Однако область определения рекурсивной подпрограммы нельзя, конечно, ограничивать посредством предохранителей произвольным образом. При этом надо учитывать и все воплощения. Например, ограничение, которое использовано в следующей подпрограмме:

```
funct fac ≡ (int n co n ≧ 1 co) int:   | function fac(n : integer
                                        |                 {n ≧ 1}) : integer ;
                                        | begin
  if n = 0                              |   if n = 0
  then 1                                |   then fac ⇐ 1
  else n × fac(n − 1) fi                |   else fac ⇐ n * fac(n − 1)
                                        | end
```

недопустимо — последнее, завершающее воплощение *fac* не удовлетворяет условию-предохранителю.

2.4.3.2

Часто условия в альтернативах и в последовательных разборах случаев либо стражи в охраняемых разборах случаев автоматически гарантируют, что в соответствующих ветвях тре-

буемые для рекурсивного вызова условия-предохранители будут соблюдены.

Например, в подпрограмме *gcd1* из 2.3.2(d) подпрограмма *mod* вызывается только в ветви со стражем $n > 0$, поэтому выполнение условия-предохранителя $n > 0$ всегда оканчивающегося алгоритма

funct *mod* ≡ (**int** *m*, **int** *n* **co** $m \geq 0 \wedge n > 0$ **co**) **int** :	**function** *mod*(*m*, *n* : *integer* {($m \geq 0$) ∧ ($n > 0$)}) : *integer* ;
	begin
if $m < n$	**if** $m < n$
then *m*	**then** *mod* ⇐ *m*
else *mod*(*m* − *n*, *n*) **fi**	**else** *mod* ⇐ *mod*(*m* − *n*, *n*)
	end

гарантировано. (Условие $m \geqslant 0$ „унаследовано" от *gcd1*.)

2.4.3.3

К этому стоит добавить, что замена последовательной конъюнкции или дизъюнкции на строгую не только ведет к большей работе, но и может привести просто к нарушению „охранительной функции", необходимой как раз в рекурсивной ситуации.

Если, например, тело оканчивающейся подпрограммы *issorted* из 2.4.1.5 преобразовать к виду

```
funct issorted ≡ (string a) bool :
  length(a) ≦ 1 ∨
    if first(a) > first(rest(a))
    then false
    else issorted(rest(a)) fi
```

то алгоритм заедет в „не определено".

2.4.3.4

Строя подпрограмму с помощью рекуррентных равенств, надо внимательно следить за тем, чтобы подпрограмма действительно заканчивалась. Так, для подпрограммы *gcd1* из 2.3.2(d), справедливо равенство

$$gcd1(m, n) = gcd1(m \ \mathbf{mod}\ n, n) \quad (n > 0).$$

Однако алгоритм

funct *gcd2* ≡ (**int** *m*, *n* **co** $m \geqq 0 \wedge n \geqq 0$ **co**) **int**:	**function** *gcd2* (*m*, *n* : *integer* {$m \geqq 0 \wedge n \geqq 0$}) : *integer*;
	begin
if $n = 0$	**if** $n = 0$
then *m*	**then** *gcd2* ⇐ *m*
else *gcd2* (*m* **mod** *n*, *n*) **fi**	**else** *gcd2* ⇐ *gcd2* (*m* **mod** *n*, *n*)
	end

для $n > 0$ не заканчивается!

2.4.3.5

Доказательство окончания подпрограммы может потребовать сложных логических или математических рассуждений. Дадим образчик такого доказательства на примере приведённой выше подпрограммы *mod*.

Если $n > 0$, то и $m - n < m$; с каждым шагом рекурсии значение первого аргумента убывает. Через некоторое конечное число шагов оно станет меньше *n* и алгоритм закончит свою работу ввиду наличия предохранителя $m \geqslant 0$.

2.5. Подчинение подпрограмм

2.5.1. Подчинённые подпрограммы

Часто „снаружи" интересуются лишь какой-нибудь одной из подпрограмм данной системы. Так бывает не только с иерархически построенными системами вроде (*isprim*, *ispr*) из 2.4.1.3, но и с перекрёстными рекурсивными системами, такими как (*ispos*, *isneg*) из 2.4.1.4. В таких случаях локального употребления подпрограмм, введённых как вспомогательные по отношению к некоторой „основной", можно через ***подчинение*** их этой основной подпрограмме выразить запрет на их использование „извне". Это подчинение указывается при помощи соответствующих скобок (рис. 79).

Аналогично для *fac* и *facr* (см. 2.4.1.3), *gcd1* и *mod* (см. 2.3.2). Для *ispos* и *isneg* подчинение второй подпрограммы первой выражается, как показано на рис. 80.[1]

В алголе формула, которой предшествует подчинённая подпрограмма, называется ***сегментом*** или ***предложением.*** Сегмент заключают в скобки-уголки[2] ⌈ ⌋. В качестве разделительного

[1] В стандартном паскале транслятор вообще требует, чтобы вызову некоторой подпрограммы предшествовало её описание (или по крайней мере „предварительное оповещение" [08]).

[2] В стандартном алголе-68 используются словарные скобки **begin** **end**.

знака между подпрограммой и формулой используют точку с запятой.

Сегмент сам является формулой (в обобщённом смысле) и может выступать как в качестве операнда операции (и в разборах случаев[1]*), так и в качестве тела подпрограммы.*

```
funct isprim ≡ (int n
                  co n ≧ 1 co) bool :
⌈funct ispr ≡ (int n, m
             co 2 ≦ m ≦ n co) bool :

    if m = n
    then true
    else if n mod m = 0
         then false
         else ispr(n, m+1) fi fi
                                  ;

  if n = 1
  then false
  else ispr(n, 2) fi               ⌋
```

```
function isprim (n : integer
                  {n ≧ 1}) : Boolean ;
  function ispr(n, m : integer
                  {2 ≦ m ≦ n}) : Boolean ;
  begin
  if m = n
  then ispr ⇐ true
  else if n mod m = 0
       then ispr ⇐ false
       else ispr ⇐ ispr(n, m+1)
  end ;
begin
if n = 1
then isprim ⇐ false
else isprim ⇐ ispr(n, 2)
end
```

Рис. 79.

В паскале конец подчинённой подпрограммы указывается с помощью словарной скобки **end**, которая соотносится со стоящей в начале тела скобкой **begin**; отвечающая сегменту конструкция (вместе с фиксацией результата под некоторым обозначением) будет названа в 3.2.4 *оператором*.

В соответствии с произведённым подчинением связываются и обозначения подчинённых подпрограмм (а не только их параметры), а именно связываются в пределах данного сегмента, а если он служит телом подпрограммы — то в пределах этой подпрограммы (***область связывания***).

Областью связывания для подпрограммы *isneg* (в этой области обозначение *isneg* можно заменить произвольным, лишь бы взаимосогласованным образом) является тело „главной“ подпрограммы *ispos*.

Подчинённые подпрограммы вводят, в частности, тогда, когда имеется несколько подвыражений, одинаково устроенных, но с различными обозначениями. Это позволяет сократить запись:

Пример. Формула (*) из 2.2.2.2

$$(b \times 2 + a) \times d + (a \times 2 + b) \times c$$

---

[1] **if.then, elsf.then, ⌈.then, then.elsf, then.⌋, then.fi, then.else, else.fi** действуют как скобки-уголки.

```
funct ispos ≡ (string m) bool :          | function ispos (m : string) : Boolean ;
⌈funct isneg ≡ (string m) bool :         |   function isneg (m : string) : Boolean ;
                                         |   begin
     if m = ◊                            |   if isempty(m)
     then false                          |   then isneg ⇐ false
     else if first(m) = "−"              |   else if first(m) = '−'
          then ispos(rest(m))            |        then isneg ⇐ ispos(rest(m))
          ▯ first(m) = "+"               |        ▯ first(m) = '+'
          then isneg(first(m)) fi fi     |        then isneg ⇐ isneg(rest(m))
                                       ; |   end ;
                                         | begin
   if m = ◊                              | if isempty(m)
   then true                             | then ispos ⇐ true
   else if first(m) = "−"                | else if first(m) = '−'
        then isneg(rest(m))              |      then ispos ⇐ isneg(rest(m))
        ▯ first(m) = "+"                 |      ▯ first(m) = '+'
        then ispos (rest(m)) fi fi       |      then ispos ⇐ ispos(rest(m))
                                       ⌋ | end
```

Рис. 80.

может быть записана в виде следующего сегмента, соответственно оператора (›Res‹ снова обозначает „результат", см. 2.3.1.2):

```
⌈funct h ≡ (real u, v, w) real :         | function h (u, v, w : real) : real ;
        (u × 2 + v) × w ;                |   begin h ⇐ (u*2 + v)*w end ;
                                         | begin
  h(b, a, d) + h(a, b, c)           ⌋    |   ›Res‹ ⇐ h(b, a, d) + h(a, b, c)
                                         | end
```

Этот сегмент (соотв. оператор) можно было использовать в теле подпрограммы f из 2.3.1.2.

Хотя такое введение вспомогательной подпрограммы и приводит к сокращению записи, никакой экономии вычислений при этом не достигается.

2.5.2. Подавленные параметры

2.5.2.1. Глобальные и нелокальные параметры

> *"Dog! That ain't no fittin' name for a dog."* [1]
>
> Ричард Нэш [2]
> 'The rainmaker' [3]

Иногда бывает, что какой-то параметр подчинённой подпрограммы при всех её возможных вызовах остаётся неизменным,—

---

[1] „Пёс! Неподходящее это имя для пса." — *Прим. перев*

[2] Н. Ричард Нэш (род. 1916), современный американский драматург. — *Прим. ред.*

[3] Буквально: „Делающий дождь" (*англ.*). Эта пьеса шла у нас под названием „Продавец дождя". — *Прим. перев.*

если пользоваться карточной терминологией, „пасует“. Именно так обстоит дело в сегменте, который после подчинения подпрограммы *ispr* даёт тело подпрограммы *isprim* из 2.5.1: первый параметр *n* подпрограммы *ispr* „пасует“ в вызове *ispr*(*n*, 2) и во всех рекурсивных вызовах *ispr*(*n*, *m* + 1). Поскольку подпрограмма *ispr* — подчинённая, можно отказаться от задания этого параметра (и тем самым сделать его неизменяемым); новое тело

```
⌈funct ispr1 ≡ (int m                  | function ispr1(m : integer
            co 2≦m≦n co) bool:         |                 {2≦m≦n}) : Boolean ;
                                       |   begin
   if m = n                            |   if m = n
   then true                           |   then ispr1 ⇐ true
   else if n mod m = 0                 |   else if n mod m = 0
     then false                        |        then ispr1 ⇐ false
     else ispr1(m+1) fi fi ;           |        else ispr1 ⇐ ispr1(m+1)
                                       |   end ;
 if n = 1                              | begin if n = 1
 then false                            | then isprim ⇐ false
 else ispr1(2) fi                      | else isprim ⇐ ispr1(2)
                                       | end
```

Рис. 81.

isprim примет в таком случае вид, показанный на рис. 81. В этом сегменте (соотв. операторе) *n* обозначает ***подавленный параметр***. Если сегмент (соотв. оператор), как только что выше, стоит — или рассматривается — изолированно, то подавленный параметр называют также ***глобальным*** параметром сегмента (соотв. оператора).

В общем же случае подавленный параметр обычно является связанным в некоторой „главной“ подпрограмме, в нашем примере глобальный параметр *n* связан в подпрограмме *isprim*, имеющей заголовок

```
funct isprim ≡ (int n                  | function isprim(n : integer
               co n≧1 co) bool:        |                 {n≧1}) : Boolean ;
```

В такой ситуации подавленный параметр называют ***нелокальным*** — в отличие от неподавленных параметров, которые называют ***локальными***. В нашем примере *n* — нелокальный, а *m* — локальный параметры подпрограммы *ispr1* в теле подпрограммы *isprim*.

Также и в системе (*gcd1*, *mod*) из 2.3.2(d) второй параметр подпрограммы *mod* „пасует“. После подчинения *mod* этот параметр может быть подавлен; тело подпрограммы *gcd1* примет

тогда вид, показанный на рис. 82 (для перестраховки оставшийся неподавленным параметр мы обозначили через *u* вместо

funct *gcd1* ≡ (**int** *m*, **int** *n*	**function** *gcd1* (*m*, *n* : *integer*
co $m \geqq 0 \wedge n \geqq 0$ **co**) **int**:	{$(m \geqq 0) \wedge (n \geqq 0)$}) : *integer* ;
⌈**funct** *mod1* ≡ (**int** *u*	**function** *mod1* (*u* : *integer*
co $u \geqq 0$ **co**) **int**:	{$u \geqq 0$}) : *integer* ;
	begin
if $u < n$	**if** $u < n$
then *u*	**then** *mod1* ⇐ *u*
else *mod1* (*u* − *n*) **fi** ;	**else** *mod1* ⇐ *mod1* (*u* − *n*)
	end ;
	begin
if $n = 0$	**if** $n = 0$
then *m*	**then** *gcd1* ⇐ *m*
else *gcd1* (*n*, *mod1* (*m*)) **fi** ⌋	**else** *gcd1* ⇐ *gcd1* (*n*, *mod1* (*m*))
	end

Рис. 82.

m). Параметр *n*, глобальный в сегменте (соотв. операторе), представляющем собой тело подпрограммы *gcd1*, в самой этой подпрограмме связан.

2.5.2.2. Экранирование

„Name ist Schall und Rauch“ [1]

Гёте

„Фауст“, часть I

Так как связанное обозначение может быть заменено (согласованным образом) на любое другое, то в приведённом выше примере параметр в *mod1* в дальнейшем может быть обозначен через *m* (рис. 83). Теперь обозначение *m* используется в двух разных значениях. Тем не менее путаницы произойти не может: *m* внутри тела *mod1* означает параметр из *mod1*, а в остальной части тела подпрограммы *gcd1* — первый параметр этой подпрограммы. Говорят, что ***область действия*** (*англ.: scope* [2]) параметра *m* в *gcd1* имеет „дыру“; область действия параметра — это его область связывания за вычетом области связывания одинакового, но используемого на „более внутреннем“ уровне обозначения, которое „загораживает“ (***экранирует***) обозначение, используемое на „более внешнем“ уровне.

---

[1] „...имя — только дым и звук“ (*нем.*). Перевод Н. Холодковского (М.: Детская литература, 1973). — *Прим. перев.*

[2] Здесь: сфера, поле (деятельности). — *Прим. изд. ред.*

Чтобы не возникало никаких забот, связанных с учётом экранизации, проще всего последовательно использовать различные обозначения или, если они уже заданы, выполнить соответствующие переименования — как, скажем, в приведённом выше примере, где мы заменили параметр *m* из *mod1* на *u*.

```
funct gcd1 ≡ (int m, n
              co m ≧ 0 ∧ n ≧ 0 co) int:
 ⌈funct mod1 ≡ (int m
               co m ≧ 0 co) int:

   if m < n
   then m
   else mod1(m − n) fi ;


  if n = 0
  then m
  else gcd1(n, mod1(m)) fi          ⌋
```

```
function gcd1(m, n : integer
          {(m ≧ 0) ∧ (n ≧ 0)}) : integer ;
  function mod1(m : integer
                 {m ≧ 0}) : integer ;
    begin
      if m < n
      then mod1 ⇐ m
      else mod1 ⇐ mod1(m − n)
    end ;
begin
  if n = 0
  then gcd1 ⇐ m
  else gcd1 ⇐ gcd1(n, mod1(m))
end
```

Рис. 83.

При „сосуществовании“ подпрограмм *mod* и *gcd1* в системе (*gcd1*, *mod*) из 2.3.2(d) или подпрограмм *ispos* и *isneg* в системе (*ispos*, *isneg*) из 2.4.1.4, очевидно, вопрос об экранировании не стоит. Но если, как в 2.5.1, подчинить подпрограмму *isneg* подпрограмме *ispos* без переобозначений, то возникнет экранирование!

2.5.2.3. Неподвижные параметры

Встаёт вопрос: какие параметры подавляемы? В нерекурсивной подпрограмме все параметры подавляемы, и подавление параметра просто означает, что мы отказываемся от возможности изменять его значение. В рекурсивных подпрограммах дело обстоит не так. Здесь подавляемы лишь ***неподвижные параметры***[1]; мы называем параметр данной подпрограммы неподвижным, если его значение не изменяется ни при каком рекурсивном вызове этой подпрограммы, являющемся прямым или косвенным следствием её первоначального вызова.

Неподвижны, например, первый параметр подпрограммы *ispr* (см. 2.4.1.3 и 2.5.1) и второй параметр подпрограммы *mod*

[1] В оригинале konstant bezetzte Parameter (параметры с постоянной загрузкой). — *Прим. изд. ред.*

(см. 2.3.2(d)). Неподвижен также первый параметр подпрограммы *iter* из 2.4.1.1. Очевидно, что каждый параметр нерекурсивной подпрограммы неподвижен.

2.5.3. Подпрограммы, свободные от параметров

Подпрограммы, которые совсем не имеют параметров (***подпрограммы без параметров***), мало кому нужны — если они детерминированны, то они вырабатывают один постоянный результат. Единственное исключение — это базовые подпрограммы, представляющие стохастические источники, например «случайное вещественное число из полузамкнутого интервала [0, 1)» (стандартная функция в алголе-68) или «**true** или **false**, по случайному выбору».

Иначе обстоит дело с (нерекурсивными) подпрограммами, у которых все параметры подавлены (***подпрограммы, свободные от параметров***). Для отличия от объектов в алголе-68 формула, которая даёт подпрограмму, свободную от параметров, предваряется видом результата, отделённым от нее двоеточием.

Примеры (справа в скобках указаны глобальные параметры):

int: $i+k-1$	(i,k),
bool: $(a \wedge b \wedge c) \vee (\neg a \wedge \neg b \wedge \neg c)$	(a,b,c),
string: $s +$ "ческая"	(s),
real: $(((a0 \times x + a1) \times x + a2) \times x + a3) \times x + a4$	$(a0, a1, a2, a3, a4, x)$,
real: **if** $x > 0$ **then** x **else** $-x$ **fi**	(x).

При описании для такой подпрограммы можно использовать произвольно выбранное обозначение, например:

funct *absx* $\equiv$ *real*: **if** $x > 0$ **then** x **else** $-x$ **fi**	**function** *absx* : *real*; **begin** *absx* $\Leftarrow$ **if** $x > 0$ **then** x **else** $-x$ **end**.

Здесь *absx* — обозначение подпрограммы, вычисляющей модуль глобального параметра x, внешне неотличимое от обозначения параметра.

Если нужно выполнить такую подпрограмму, то это можно указать, поместив перед её обозначением специальный операционный символ **exec** [1] („исполни"). Такой символ опускается в большинстве алгоритмических языков, в том числе и в алголе-68 и в паскале. Если написано

$$x + absx,$$

[1] От execute (исполнять). — *Прим. изд. ред.*

то это означает

$$x + \textbf{if } x > 0 \textbf{ then } x \textbf{ else } -x \textbf{ fi}.$$

Говорят, что подпрограмма, свободная от параметров, вызывается уже заданием своего обозначения.

2.6. Подпрограммы в качестве параметров и в качестве результатов

2.6.1. Подпрограммы в качестве параметров

Иногда в подпрограммах заменяемы не только объекты, которые она обрабатывает, но и операции. В таком случае соответствующая подпрограмма должна быть введена как параметр (***функциональный параметр***, или ***параметрическая функция***). Классическим примером служит формула Симпсона для приближённого вычисления интеграла $\int_a^b f(x)\,dx$

$$(a-b)\times(f(a)+4\times f((a+b)/2)+f(b))/6$$

с функциональным параметром f для обозначения интегрируемой функции. Для подпрограммы, используемой в качестве параметра некоторой подпрограммы, в заголовке последней надлежащим способом указывается функциональный тип первой, например [1]

funct *simpson* ≡ (**funct** (**real**) **real** *f*, **real** *a*, *b*) **real**:	**function** *simpson* (**function** *f*(*real*) : *real*; *a*, *b* : *real*) : *real*;
	begin
$(a-b)\times(f(a)+4\times f((a+b)/2)+f(b))/6$	*simpson* ⇐ $(a-b)*(f(a)+4*f((a+b)/2)+f(b))/6$
	end

При вызове подпрограммы *simpson* необходимо фактически задать подпрограмму вычисления интегрируемой функции, например

$$simpson(g, 2.05, 2.06),$$

[1] В некоторых реализациях паскаля тип параметров подпрограммы, выступающей в роли параметра, задаётся не полностью, а именно задаётся лишь тип результата; от группы параметров он отделяется, как и прежде, точкой с запятой. Кроме того, параметры параметрической функции не могут быть в свою очередь функциональными параметрами.

где подпрограмма g определена, скажем, так:

funct $g \equiv$ (**real** x) **real**: $exp(x)/x$	**function** $g(x : real) : real$; **begin** $g \Leftarrow exp(x)/x$ **end**

В алголе-68 подставляемая вместо интегрируемой функции подпрограмма может быть подставлена и прямо, без введения специального обозначения:

$$simpson\ ((\mathbf{real}\ x)\mathbf{real}:\ exp(x)/x,\ 2.05,\ 2.06),$$

что ближе к математической записи

$$\int_{2.05}^{2.06} dx\, exp(x)/x, \quad \text{или, привычнее,} \quad \int_{2.05}^{2.06} (exp(x)/x)\, dx;$$

в обеих записях x является связанным обозначением!

Что касается соответствующей машины обработки формуляров, то в неё теперь вводят дополнительно формуляр-образец для (**real** x)**real** : $exp(x)/x$; в теле нерекурсивной подпрограммы *simpson* будут затребованы три его копии — подставляемая подпрограмма появляется в трёх воплощениях.

С функциональным параметром можно реализовать и рекурсивную схему *iter* из 2.4.1.1, например интерпретируя λ (соотв. λ) как сорт **int** (соотв. *integer*)[1]:

```
funct iter ≡                              | function iter(function rho(integer, integer):integer;
     (funct (int,int) int rho.            |              a, n:integer [n > 0]) : integer;
     int a, int n co n>0 co) int:         |
                                          |   begin
  if n = 1                                |   if n = 1
  then a                                  |   then iter ⇐ a
  else rho(a, iter(rho, a, n−1)) fi       |   else iter ⇐ rho(a, iter(rho, a, n−1))
                                          |   end
```

Обратите внимание, что здесь и *rho*, и *a* являются неподвижными параметрами. Сколько копий формуляра *rho* потребуется, зависит от n.

Аналогичным образом можно осуществить и введение сортирующего предиката в данный алгоритм сортировки. Другой типичный пример — это обращение подпрограммы с функцио-

[1] Ниже *rho* — от Rho (немецкое название греческой буквы ρ). — *Прим. изд. ред.*

нальным типом (int)int: (соотв. (*integer*): *integer*):[1]

```
funct invert ≡                          | function invert
    (funct(int)int f, int a) int:       |    (function f(integer):integer; a:integer):integer;
                                        |  begin invert ⇐
    «irgendein x ≧ 0 derart,            |     «irgendein x ≧ 0 derart, daß f(x) = a gilt»
              daß f(x) = a gilt»        |
                                        |  end
```

Если вложить эту задачу в более общую: «Найти какое-нибудь $x \geqslant i$, такое что $f(x) = a$», то в качестве одного из решений получим

```
funct invert ≡                          | function invert
    (funct(int)int f, int a) int:       |    (function f(integer):integer; a:integer):integer;
  ⌈funct inv ≡ (funct(int)int f,        |  function inv(function f(integer):integer;
                  int a, int i) int:    |           a, i:integer):integer;
                                        |  begin
    if f(i) = a                         |  if f(i) = a
    then i                              |  then inv ⇐ i
    else inv(f, a, i+1) fi;             |  else inv ⇐ inv(f, a, i+1)
                                        |  end;
                                        | begin
  inv(f, a, 0)                          | invert ⇐ inv(f, a, 0)
                                      ⌋ | end
```

Подпрограмма *invert* заканчивается, если обращение возможно. Фактически *inv* определяет лишь наименьшее $x \geqslant i$, для которого $f(x) = a$. В подпрограмме *inv* как f, так и a неподвижны; подавляя их, получаем

```
funct invert ≡                          | function invert
    (funct(int)int f, int a) int:       |    (function f(integer): integer; a:integer):integer;
                                        |
  ⌈funct inv ≡ (int i) int:             |  function inv(i:integer):integer;
                                        |  begin
    if f(i) = a                         |  if f(i) = a
    then i                              |  then inv ⇐ i
    else inv(i+1) fi;                   |  else inv ⇐ inv(i+1)
                                        |  end;
                                        | begin
  inv(0)                                |  invert ⇐ inv(0)
                                      ⌋ | end
```

2.6.2. Задержанные вычисления, осуществляемые посредством использования подпрограмм, свободных от параметров, в качестве параметра

Если в подпрограмме вместо некоторого объектного параметра сорта **λ** (соотв. λ) задать свободную от параметров подпро-

[1] Ниже *invert* и *inv* — от inversion (обращение), а выражение в кавычках (на немецком) означает «любое $x \geqslant 0$, такое что $f(x) = a$».

грамму с функциональным типом λ: (алгол), соотв.: λ (паскаль), то вместо формулы, обработка которой даёт значение, присваиваемое обычному параметру, надо будет использовать обобщённую формулу, представляющую собой подпрограмму, свободную от параметров. Это существенно изменяет процесс вычисления на машине обработки формуляров — вводимые формулы обрабатываются не один раз (***вызов значением***, англ.: call by value), а столько раз, сколько они будут вызываться в теле (***вызов выражением***, *англ.*: call by expression)[1]. Это может привести как к увеличению, так и к уменьшению затрат на обработку. Таким образом „смягчают" строгость подпрограмм с объектными параметрами.

Пример. Даны две (по существу идентичные) подпрограммы: одна с объектным параметром x:

funct $d \equiv$ (**real** x, **bool** a) **real**:
if a **then** $x \times ln(x)$ **else** 0 **fi**,

другая — со свободной от параметров подпрограммой e в качестве параметра:

funct $dd \equiv$ (**funct real** e, **bool** a) **real**:
if a **then** $e \times ln(e)$ **else** 0 **fi**.

Каждый из вызовов

$d(tan(18.325)$, **true**), $d(tan(18.325)$, **false**)

требует ровно одного вычисления $tan(18.325)$. В противоположность этому вызов

dd(**real** : $tan(18.325)$, **true**)

влечёт за собой два вычисления $tan(18.325)$, вызов же

dd(**real** : $tan(18.325)$, **false**)

не инициирует вообще ни одного вычисления $tan(18.325)$.

Однако наиболее эффективны (Хендерсон, Моррис, 1976 г.) ***задержанные вычисления***, когда каждое вычисление проводится самое большее один раз:

funct $dd \equiv$ (**funct real** e; **bool** a) **real**:
⌈**funct** $h \equiv$ (**real** y) **real** : $y \times ln(y)$;
if a **then** $h(e)$ **else** 0 **fi** ⌋.

В паскале такое прямое использование подпрограмм, свободных от параметров, в качестве аргументов невозможно.

---

[1] Ср. с 8.2.5. — *Прим. изд. ред.*

Однако в ряде языков программирования приняты меры для возможности трактовки вводимых формул как подставляемых выражений, а не значений (алгол-60, вызов именем, *англ.*: *call by name*, — в противоположность обычному вызову значением). В лиспе даже принципиально вызов всех параметров выполняется как вызов выражением. С этим, однако, как правило, связана потеря эффективности.

2.6.3. Подпрограммы в качестве результата

Если подпрограммы допускаются в качестве параметра, почему бы не допустить их и в качестве результата? Однако вычислять по подпрограмме всё-таки значительно проще, чем вычислять подпрограмму. Поэтому здесь мы рассмотрим лишь простейший случай, когда подпрограмма образуется из некоторой другой подпрограммы путём фиксации одного или нескольких параметров. Нам уже приходилось встречаться с этим в некоторых случаях вложения — в последний раз в 2.6.1 (вложение *invert* в *inv*), а раньше, скажем, в 2.4.1.3 (при вычислении *isprim*). Из подпрограммы *iter* (см. 2.6.1) фиксацией *rho* получается подпрограмма с функциональным типом (**int**, **int**)**int**. В общем случае мы должны были бы написать[1, 2]

funct *genitor* ≡ (**funct** (**int**, **int**) **int** *rho*) **funct** (**int**, **int**) **int**:
(**int** *a*, **int** *n* **co** $n \geqslant 1$ **co**) **int**: *iter* (*rho*, *a*, *n*).

Посредством вызова

genitor (*plus*)

получаем обозначаемый через *mult*[3] алгоритм обычного умножения:

funct *mult* ≡ (**int** *a*, **int** *n* **co** $n \geqslant 1$ **co**) **int**:
if $n = 1$ **then** *a* **else** *plus*(*a*, *mult* (*a*, $n - 1$)) **fi**,

а двойной вызов

genitor(*genitor*(*plus*)),

т. е. вызов *genitor*(*mult*), даёт обозначаемый через *pow*[4] алгоритм обычного возведения в степень

funct *pow* ≡ (**int** *a*, **int** *n* **co** $n \geqslant 1$ **co**) **int**:
if $n = 1$ **then** *a* **else** *mult*(*a*, *pow*(*a*, $n - 1$) **fi**.

---

[1] Конструкции подобного рода в паскале не предусмотрены, а в симуле и алголе-68 допустимы лишь с оговорками; они имеют законную силу в лямбда-исчислении Чёрча.

[2] Ниже *genitor* и *gen* — от английского (или, точнее, латинского) genitor (родитель, отец). — *Прим. изд. ред.*

[3] От multiplication (умножение). — *Прим. изд. ред.*

[4] От power (степень). — *Прим. изд. ред.*

Если, далее, в *mult* или *pow* фиксировать второй аргумент, то получим алгоритмы с функциональным типом (**int**) **int** для умножения на фиксированное число и возведения в фиксированную степень. Скажем, в последнем случае можно написать

$$\textbf{funct } genpow \equiv (\textbf{int } n \textbf{ co } n \geqslant 1 \textbf{ co}) \textbf{ funct } (\textbf{int}) \textbf{ int}: (\textbf{int } a) \textbf{ int} : pow(a, n),$$

и тогда вызов

$$genpow(4)$$

даёт операцию возведения в четвёртую степень (***двухступенчатая параметризация***).

Заметим, что в этом случае фиксируется единственный „подвижный" параметр подпрограммы *pow*, что позволяет последовательными подстановками выполнить рекурсию заранее раз и навсегда (***частичное вычисление***). Поэтому вместо наивного исполнения приведённого выше алгоритма *genpow*, который для каждого *n* выдаёт рекурсивную подпрограмму, следовало бы иметь более эффективный алгоритм, например выдающий по вызову *genpow* (4) „конкретную" подпрограмму

$$(\textbf{int } a) \textbf{ int} : mult(a, mult(a, mult(a, a))).$$

В общем, здесь перед нами открывается обширное поле деятельности — манипуляции с формулами и алгоритмами.

Необходимо ясно представлять себе, что функциональные параметры подпрограмм заведомо неподвижны лишь в том случае, если подпрограммы не допускаются в качестве результатов подпрограмм. В противном случае работа машины обработки формуляров уже не описывается больше таким простым образом (введение экземпляров-образцов для действительных формуляров). Но тут мы вступаем в область собственно „функционального программирования", практическое освоение которой ещё далеко от завершения.

Глава 3

Машинно-ориентированные алгоритмические языки

Основные понятия программирования были развиты в гл. 2 на основе понятий, связанных с машиной обработки формуляров. Это машина, которая действует очень „по-человечески": легко представить себе, как организовать весь ход вычислений для некоторого одиночного вычислителя (скажем, по принципу магазина[1]) или для некоторой группы вычислителей, причём свобода вычислений в зависимости от обстоятельств либо ограничивается весьма незначительно, либо не ограничивается вовсе.

Однако в реальной машине обработки формуляров организация вычислений должна быть полностью механизирована. Реальные вычислительные машины доминирующей (до сих пор) архитектуры („фоннеймановского типа") не столь рафинированы, как машина обработки формуляров; в частности, они совершенно беспомощны перед лицом рекурсии в её полной общности. Причина этого — технологические тиски, господствовавшие в середине нашего века. Они не позволили машине обработки формуляров из-за её „широкого характера" найти широкое применение: для каждого воплощения нужно заводить новый формуляр, что легко делается только на бумаге (или, лучше сказать, только когда не жаль бумаги). В то время не было дешёвой техники, позволяющей автоматизировать чтение и запись в рабочие поля формуляра.

Поэтому Конрад Цузе в 1934 г. перешёл к многократно используемым формулярам, в которых можно переписывать заново записи в результативных полях[2]; для всей рекурсии многократно используется один-единственный формуляр. Прежде всего это исключает рекурсию с „последействием" (см. 2.3.3). Тем самым допускается *лишь* повторительная рекурсия. В простом случае непосредственной рекурсии это приводит к понятию *повторения*, а в других случаях — к понятию *перехода*. Кроме того, возможность повторного использования результатных полей приводит к понятию *переменных памяти* и к понятию *памяти*

[1] Об этом принципе см. ниже — *Прим. ред.*

[2] Подробности см. в статье [23].

как множества переменных памяти. Родственное понятие *программной переменной* наряду с понятиями повторения и перехода будет играть ключевую роль в этой главе. Естественное употребление программных переменных в качестве параметров ведёт, далее, от строго функционального понимания подпрограммы к понятию *процедуры*.

Машинам фоннеймановского типа свойственны линейный порядок на всём множестве переменных памяти (*линейная память*) и строго последовательная организация хода вычислений, которая при условии исключения какой бы то ни было параллельности подразумевает обработку в соответствии с магазинным принципом.

«La plus belle ruse du Diable est de nous persuader qu'il n'existe pas.»[1]

Бодлер

3.1. Предложения общего вида

В разделе 2.5.1 формулы, которым предшествуют подчинённые им подпрограммы, были названы предложениями. Теперь мы обобщим это понятие, включив возможность описания промежуточных результатов, с помощью чего нам уже удалось однажды устранить один изъян, обнаруженный в конце раздела 2.2.2.4.

3.1.1. Описания промежуточных результатов

Для трёх сторон a, b, c треугольника наряду с неравенствами $a \geqslant 0$, $b \geqslant 0$, $c \geqslant 0$ должно выполняться условие (неравенство треугольника)

$$b + c \geqslant a \wedge c + a \geqslant b \wedge a + b \geqslant c.$$ [2]

Проверка выполнения этого условия требует трёх сложений (и трёх сравнений). Если же преобразовать его к виду

$$(a + b + c)/2 \geqslant a \wedge (a + b + c)/2 \geqslant b \wedge (a + b + c)/2 \geqslant c,$$

то можно обойтись двумя сложениями и одним делением пополам, ибо достаточно *один раз* вычислить многократно встречающееся подвыражение $(a + b + c)/2$. В формулах, которые мы рассматривали до сих пор, нет средств для реализации этой идеи. Строго потоковый характер, отражающийся в древесном

[1] „Самая коварная уловка дьявола состоит в том, чтобы убедить нас, будто его не существует". — *Прим. перев.*

[2] Фактически неравенства $a \geqslant 0$, $b \geqslant 0$ и $c \geqslant 0$ вытекают из неравенства треугольника — *Прим. перев.*

строении формуляра, не допускает многократного использования промежуточных результатов. Наоборот, многократное использование промежуточных результатов означает наличие разветвлений в потоке данных, что невозможно в строго потоковых системах, т. е. означает отказ от древесной структуры и

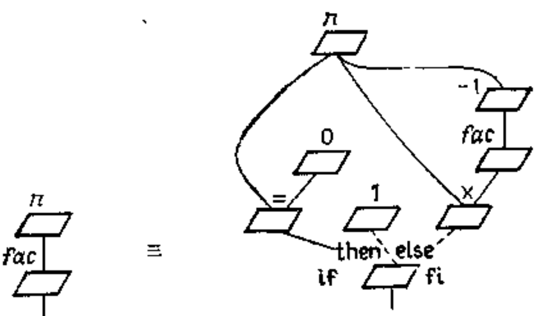


Рис. 84. Формуляр с сетевой структурой для функции *fac*.

переход к сетевой структуре. Однако многократное использование, как правило, имеет место для параметров подпрограмм: посмотрите, например, на приведённые в 2.3.2 формуляры для функции *fac* и других функций и сравните их с рис. 84. В нашем примере желаемого эффекта можно достичь, если ввести подпрограмму

funct $p \equiv$ (**real** a, b, c, s) **bool**:	**function** $p(a, b, c, s: real)$: *Boolean*;
	begin
$s \geq a \wedge s \geq b \wedge s \geq c$	$p \Leftarrow (s \geq a)$ **and** $(s \geq b)$ **and** $(s \geq c)$
	end

а затем вызывать её со значением последнего аргумента, равным $(a + b + c)/2$:[1]

funct *istriangle* $\equiv$ (**real** a, b, c) **bool**:	**function** *istriangle*$(a, b, c, : real)$: *Boolean*;
	begin
$p(a, b, c, (a+b+c)/2)$	*istriangle* $\Leftarrow p(a, b, c, (a+b+c)/2)$
	end

поскольку в соответствии с определением машины обработки формуляров все аргументы должны быть вычислены (ровно один раз) до исполнения (строгой!) подпрограммы.

[1] Ниже *istriangle* — от triangle (треугольник). — *Прим. перев.*

На рис. 85 слева изображен формуляр, соответствующий исходной формуле, а справа — его вариант с вспомогательной подпрограммой p.

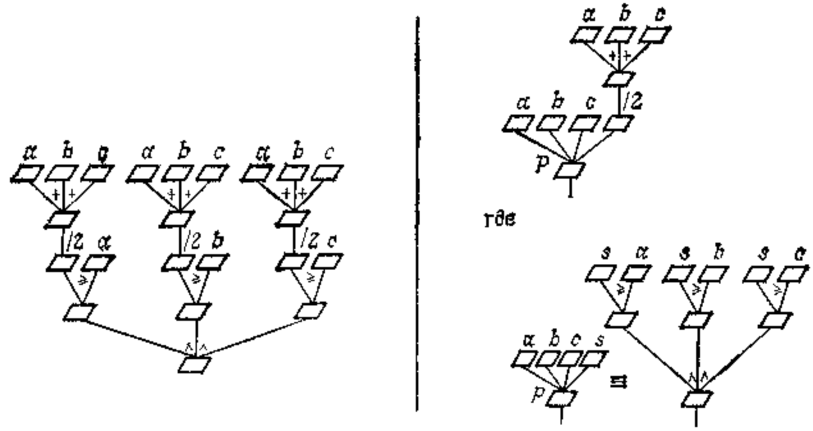


Рис. 85. Формуляр для формулы с общими подвыражениями и выделение вспомогательной подпрограммы для его структурирования.

Используя подчинение подпрограмм и удаляя неизменные параметры, подпрограмму *istriangle* можно записать короче:

```
funct istriangle ≡ (real a, b, c) bool :

  ⌈funct p ≡ (real s) bool :

      s≥a ∧ s≥b ∧ s≥c;


   p((a+b+c)/2)                      ⌋
```

```
function istriangle(a, b, c : real) :
                              Boolean;
  function p(s : real) : Boolean;
  begin
    p ⇐ (s≥a) and (s≥b) and (s≥c)
  end;
begin
  istriangle ⇐ p((a+b+c)/2)
end
```

3.1.1.1. Упрощённая нотация

Другой способ действий, более предпочтительный с точки зрения простоты нотации (хотя для машины обработки формуляров равносильный в принципе прежнему), состоит в том, чтобы ввести s как обозначение промежуточного результата. При этом нужно будет также указать тот тип, к которому относится s. В алголе-68 для этого имеется следующая нотация[1] (где снова

[1] В стандартном алголе-68 вместо символа ≡ нужно писать знак равенства. Мы используем здесь ≡, чтобы исключить возможность путаницы с универсальной операцией проверки равенства.

используются скобки-уголки, а точку с запятой следует считать как „внутри“ или „в“):

$$\lceil \mathbf{real}\ s \equiv (a+b+c)/2;\quad s \geqslant a \wedge s \geqslant b \wedge s \geqslant c \rfloor.$$

Такое введение обозначения для промежуточного результата будем называть ***описанием промежуточного (вспомогательного) результата*** или ***описанием объекта***.

На рис. 86 показан переход от формуляра с древесной структурой для исходной формулы (с внесённым в него обозна-

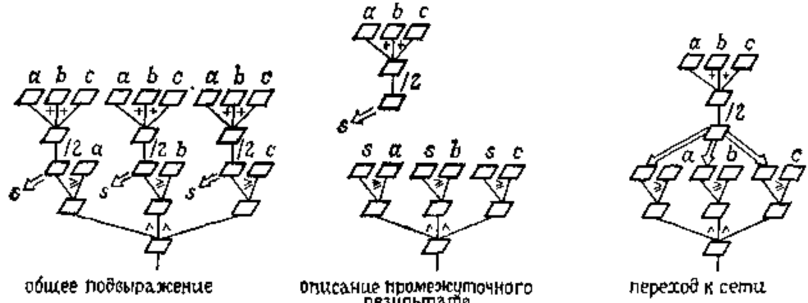


Рис. 86. Структурирование формуляра посредством описания промежуточного результата (описания объекта).

чением s для общего подвыражения) к формуляру с разветвлённым потоком данных. В результате мы получаем уже не дерево, а всего лишь ***сеть***, т. е. ацикличный ориентированный граф.

Полученная описанным образом „формула“ также будет считаться предложением; тем самым она будет формулой в некотором более общем смысле. Благодаря этому оказывается возможным записать наше условие (неравенство треугольника) для *невырожденного* треугольника как обобщённую формулу

$$a > 0 \wedge b > 0 \wedge c > 0 \wedge$$
$$\lceil \mathbf{real}\ s \equiv (a+b+c)/2;\ s > a \wedge s > b \wedge s > c \rfloor.$$ [1]

Итак, в совокупности мы получим следующие подпрограммы:

$$\mathbf{funct}\ istriangle \equiv (\mathbf{real}\ a, b, c)\ \mathbf{bool}:$$
$$\lceil \mathbf{real}\ s \equiv (a+b+c)/2;\ s \geqslant a \wedge s \geqslant b \wedge s \geqslant c \rfloor$$

---

[1] Условие $a > 0 \wedge b > 0 \wedge c > 0$ является следствием условия $\lceil \mathbf{real}\ s \equiv (a+b+c)/2;\ s > a \wedge s > b \wedge s > c \rfloor$ (см. подстрочное примечание в начале раздела 3.1.1). — *Прим. перев.*

и

$$\begin{aligned}&\mathbf{funct}\ istriangle1 \equiv (\mathbf{real}\ a,b,c)\ \mathbf{bool}:\\ &\quad a>0 \land b>0 \land c>0 \land\\ &\quad \lceil \mathbf{real}\ s \equiv (a+b+c)/2;\ s>a \land s>b \land s>c \rfloor.\end{aligned}$$

Другой пример — вычисление площади треугольника по формуле Герона с помощью подпрограммы

$$\begin{aligned}&\mathbf{funct}\ heron \equiv (\mathbf{real}\ a,b,c\ \mathbf{co}\ istriangle(a,b,c)\ \mathbf{co})\ \mathbf{real}:\\ &\quad \lceil \mathbf{real}\ s \equiv (a+b+c)/2;\ sqrt(s \times (s-a) \times (s-b) \times (s-c)) \rfloor.\end{aligned}$$

Описания промежуточных результатов можно вводить постепенно. Так, формула

$$\begin{aligned}&\lceil \mathbf{real}\ s1 \equiv a0 \times x + a1;\\ &\quad \lceil \mathbf{real}\ s2 \equiv s1 \times x + a2;\\ &\quad\quad \lceil \mathbf{real}\ s3 \equiv s2 \times x + a3;\ s3 \times x + a4 \rfloor\rfloor\rfloor\end{aligned}$$

даёт тот же результат, что и формула четырёхчленной схемы Горнера (см. 2.2.2.4), а ход вычислений у этих двух формул совпадает. Как и в случае последовательного разбора случаев, здесь также можно сэкономить на скапливающихся с правого края скобках и писать короче

$$\begin{aligned}&\lceil \mathbf{real}\ s1 \equiv a0 \times x + a1;\\ &\quad \mathbf{real}\ s2 \equiv s1 \times x + a2;\\ &\quad\quad \mathbf{real}\ s3 \equiv s2 \times x + a3;\ s3 \times x + a4 \rfloor.\end{aligned}$$

Разумеется, обозначения, введённые в любом предложении посредством описаний промежуточных результатов, должны быть попарно различными. Они снова оказываются связанными: если их согласованно заменить другими обозначениями, то получится эквивалентный алгоритм. Областью связывания обозначения промежуточного результата является предложение, в начале которого находится соответствующее описание. Это в точности соответствует области связывания параметра той подпрограммы (как, например, p в *istriangle*), которую можно было бы ввести взамен[1]. Промежуточные результаты называются ***локальными*** в том предложении, в начале которого они описаны. Область действия всякого обозначения совпадает с областью связывания этого обозначения, уменьшенной, если надо, на область связывания того же самого обозначения, описанного на вложенном уровне и „экранирующего" используемое на внешнем уровне. Примером может служить предложение

$$\lceil \mathbf{real}\ h \equiv \lceil \mathbf{real}\ h \equiv a \times a;\ h \times h \rfloor;\ h \times h \rfloor,$$

---

[1] Эту взаимосвязь обнаружил Ландин (1966 г.).

представляющее собой запись упрощённого вычисления выражения

$$((a \times a) \times (a \times a)) \times ((a \times a) \times (a \times a)).$$

Польза от обозначений промежуточных результатов, позволяющих избавиться от многократного вычисления одинаковых подвыражений, очевидна[1]. Часто удаётся, как выше, подходящим образом преобразовать формулу. Например, для вычисления логарифмической производной $p'(x)/p(x)$ полинома $p(x)$ можно получить „двухрядную схему Горнера"; для нашего примера это будет

$$\begin{array}{ll} \lceil \mathbf{real}\ s1 \equiv a0 \times x + a1; & \mathbf{real}\ t2 \equiv a0 \times x + s1; \\ \mathbf{real}\ s2 \equiv s1 \times x + a2; & \mathbf{real}\ t3 \equiv t2 \times x + s2; \\ \mathbf{real}\ s3 \equiv s2 \times x + a3; & \mathbf{real}\ t4 \equiv t3 \times x + s3; \\ & t4/(s3 \times x + a4) \rfloor. \end{array}$$

3.1.1.2. Обозначения промежуточных результатов в рекурсивных подпрограммах

Обозначения промежуточных результатов иногда могут служить для сведе́ния рекурсии общего вида к существенно более простой линейной рекурсии. Проиллюстрируем это на примере ***ханойских башен***. Это название древней восточной игры используют также как название определённого класса проблем с той же структурой.

Пусть имеется n игральных дисков $A, B, C, D, \ldots$ возрастающего диаметра. И пусть диски в соответствии с их размерами сложены в форме башни, так что диск наибольшего диаметра лежит в самом низу. Тогда задача состоит в том, чтобы переложить башню с данного места (место 1) на другое (место 2). В качестве вспомогательного средства для решения задачи в нашем распоряжении ещё одно место (место 3), куда можно складывать диски во время игры. Диски перекладываются по одному, причём в каждый момент игры можно брать лишь верхний диск одной из башен. В любой момент времени на каждом из трёх мест диск большего диаметра должен лежать ниже диска меньшего диаметра.

Задача кажется весьма трудной, однако рекурсивное решение очевидно. Обозначим самый нижний диск нашей башни буквой i. Переместим остальную часть башни, нижний диск которой обозначим предшественником буквы i, с исходного места 1 на свободное место 3; далее переместим обозначенный буквой i диск на место 2 и затем перемещённую ранее часть

---

[1] При этом в случае недетерминистических выражений множество возможных результатов иногда сужается.

башни переместим с места 3 на тот диск, что уже находится на месте 2. Итак, подпрограмма, результатом которой будет последовательность обозначений перемещаемых друг за другом дисков, запишется следующим образом [1, 2]:

```
funct toh ≡ (char i) string:

  if i > "A"
  then toh(pred(i)) + ⟨i⟩ + toh(pred(i))

  ▯ i = "A"
  then ⟨i⟩                              fi
```

```
function toh(i : char) : string;
begin
  if i > 'A'
  then toh ⇐ conc(toh(pred(i)),
                  prefix(i, toh(pred(i))))
  ▯ i = 'A'
  then toh ⇐ prefix(i, empty)
end
```

Вызов, скажем, *toh*(*'D'*) приведёт сначала к

$$toh('C') + \langle 'D' \rangle + toh('C'),$$

затем к

$$toh('B') + \langle 'C' \rangle + toh('B') + \langle 'D' \rangle + toh('B') + \langle 'C' \rangle + toh('B')$$

и, наконец, к

$$'A\,B\,A\,C\,A\,B\,A\,D\,A\,B\,A\,C\,A\,B\,A'.$$

Порождаемая цепочка всегда будет палиндромом [3] вида $h + \langle i \rangle + h$. (Она задаёт также построение циклического одно-

Рис. 87. Построение одношагового двоичного кода.

шагового n-разрядного двоичного кода, содержащего 2^n элементов, см. рис. 87.)

С помощью описания промежуточного результата из вышеприведенной формулировки подпрограммы *toh* можно получить

---

[1] Ниже *toh* — от английского названия ханойских башен: towers of Hanoi — *Прим. перев.*

[2] Используемая ниже функция *pred* определяется так:

```
funct pred ≡ (char n) char:
   repr(abs n − 1)
```

```
function pred(n : char) : char;
begin
pred ⇐ chr(ord(n) − 1)
end
```

[3] Палиндром — слово, читающееся одинаково слева направо и справа налево. — *Прим. перев.*

существенно более эффективную формулировку

```
funct toh ≡ (char i) string:

  if i > "A"
  then
        string h ≡ toh(pred(i));
        h + ⟨i⟩ + h

  ▯ i = "A"
  then ⟨i⟩                fi
```

```
function toh(i : char) : string;
begin
  if i > 'A'
  then begin
         h : string = toh(pred(i));
         toh ⇐ conc(h, prefix(i, h))
       end
  ▯ i = 'A'
  then toh ⇐ prefix(i, empty)
end
```

в которой каскадная рекурсия замещена линейной [1].

Заметим, что i-е воплощение $toh^{(i)}$ функции toh содержит, конечно же, свой собственный промежуточный результат $h^{(i)}$. При работе с машиной обработки формуляров настоящие вычисления начнутся лишь после того, как будет готов последний формуляр и для значения функции $toh^{(n)}$ (при аргументе 'A') получится результат 'A'; теперь продолжается выполнение „повисшей" операции **string** $h \equiv toh(pred(i))$, т. е. вводится обозначение $h^{(n-1)}$ для 'A' и вычисляется результат 'ABA' применения функции $toh^{(n-1)}$ к аргументу 'B'; затем вводится обозначение $h^{(n-2)}$ для 'ABA' и вычисляется результат '$ABACABA$' применения функции $toh^{(n-2)}$ к аргументу 'C', и т. д.

Наконец, отсюда уже нетрудно вывести соответствующий итеративный алгоритм, используя преобразование вложения (ниже $succ$ — обращение функции $pred$):

```
funct toh ≡ (char i) string:
⌈funct to ≡ (char i, string h) string:

  if i < t
  then to(succ(i), h + ⟨i⟩ + h)
  else h + ⟨i⟩ + h               fi;


to("A", ◇)                            ⌋
```

```
function toh(i : char) : string;
function to(i : char; h : string) : string;
begin
  if i < t
  then to ⇐ to(succ(i), conc(h, prefix(i, h)))
  else to ⇐ conc(h, prefix(i, h))
end;
begin
  toh ⇐ to('A', empty)
end
```

Использование обозначений промежуточных результатов для целей структурирования особенно бросается в глаза в примерах вроде следующей подпрограммы для вычисления восьмой

---

[1] Относительно паскалевской версии см. также 3.1.1.4.

степени числа:

funct $pow8 \equiv$ (**real** a) **real**:
⌈**real** $h1 \equiv a \uparrow 2$; **real** $h2 \equiv h1 \uparrow 2$; $h2 \uparrow 2$⌋.

Отметим, что эта „детализованная" форма является сокращенной записью для системы

funct $pow8 \equiv$ (**real** a) **real** : $p1(a \uparrow 2)$
funct $p1 \equiv$ (**real** $h1$) **real** : $p2(h1 \uparrow 2)$
funct $p2 \equiv$ (**real** $h2$) **real** : $h2 \uparrow 2$,

или, с использованием подчинения функций,

funct $pow8 \equiv$ (**real** a) **real**:
⌈**funct** $p1 \equiv$ (**real** $h1$) **real**:
⌈**funct** $p2 \equiv$ (**real** $h2$) **real**: $h2 \uparrow 2$;
$p2(h1 \uparrow 2)$ ⌋;
$p1(a \uparrow 2)$ ⌋.

Подстановка даст, разумеется,

funct $pow8 \equiv$ (**real** a) **real** : $((a \uparrow 2) \uparrow 2) \uparrow 2$.

3.1.1.3. Линеаризация хода вычислений

Для совместных формул (параллельно-последовательных) свободу в вычислениях можно ограничивать с помощью разбие-

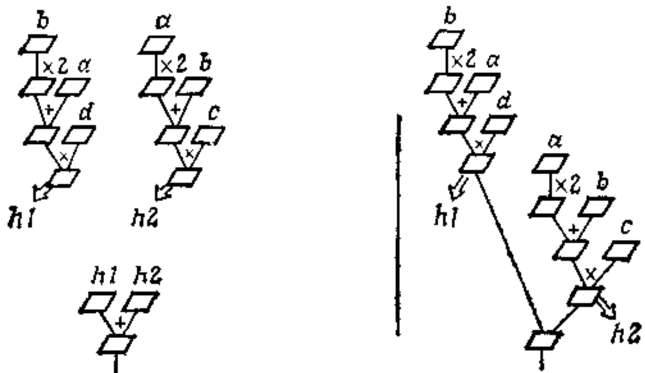


Рис. 88. Формуляры, получающиеся в результате декомпозиции и линеаризации.

ния на предложения. Так, например, из формулы

$$(b \times 2 + a) \times d + (a \times 2 + b) \times c$$

разбиением на предложения можно получить формулу

$$\lceil \mathbf{real}\ h1 \equiv (b \times 2 + a) \times d;\ \mathbf{real}\ h2 \equiv (a \times 2 + b) \times c;\ h1 + h2 \rfloor,$$

которая однозначно задаёт теперь последовательное вычисление „слева направо" (см. рис. 75) и уже не допускает вычисления „изнутри наружу". Точка с запятой становится знаком, задающим порядок вычислений.

На рис. 88 изображены соответствующие записи вычислительных формуляров.

3.1.1.4. Замечание относительно паскаля

Описания промежуточных результатов называются в алголе-68 ***описаниями тождества***. Строго говоря, в паскале они отсутствуют; в стандартном паскале нет ни предложений, ни условных выражений. Однако некоторые версии паскаля, как, скажем MESA, допускают описания промежуточных результатов, например:

$$\begin{array}{l} \mathbf{function}\ \mathit{istriangle}(a, b, c : \mathit{real}) : \mathit{Boolean}; \\ \mathbf{begin}\ s : \mathit{real} = (a + b + c)/2; \\ \qquad \mathit{istriangle} \Leftarrow (s \geqslant a)\ \mathbf{and}\ (s \geqslant b)\ \mathbf{and}\ (s \geqslant c) \\ \mathbf{end.} \end{array}$$

Мы воспользовались этим в разделе 3.1.1.2 для описания функции *toh*.

3.1.2. Групповые описания промежуточных результатов

Подобно тому как это делалось в 3.1.1, с помощью одной вспомогательной подпрограммы можно добиться однократного вычисления и каждого из нескольких подвыражений, многократно встречающихся в некоторой формуле. Например, для формулы

$$(a \uparrow 2 + b \uparrow 2) \times \mathit{ln}((a + b)/(a - b)) + (a + b) \times (a - b)$$

можно ввести вспомогательную подпрограмму

$$\mathbf{funct}\ p \equiv (\mathbf{real}\ a, b, s, d)\ \mathbf{real} : (a \uparrow 2 + b \uparrow 2) \times \mathit{ln}(s/d) + s \times d$$

и вызов

$$p(a, b, a + b, a - b).$$

Более коротко это записывается так:

$$\lceil (\mathbf{real}\ s,\ \mathbf{real}\ d) \equiv (a + b, a - b);\ (a \uparrow 2 + b \uparrow 2) \times \mathit{ln}(s/d) + s \times d \rfloor,$$

или, ещё короче,

$$\lceil (\mathbf{real}\ s, d) \equiv (a + b, a - b);\ (a \uparrow 2 + b \uparrow 2) \times \mathit{ln}(s/d) + s \times d \rfloor$$

— нет никаких оснований вводить промежуточные результаты последовательно, один за другим. Если в дальнейшем мы хотим допустить параллельную обработку, то сверх обычного, строго последовательного постепенного введения вспомогательных

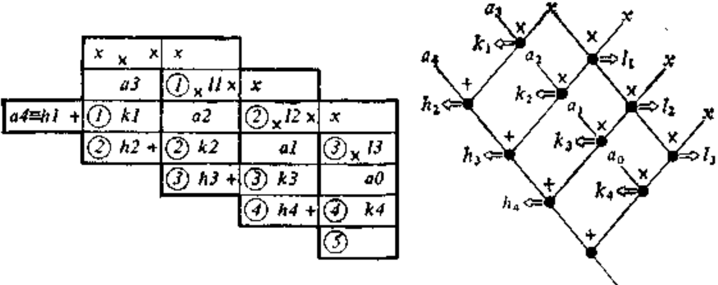


Рис. 89. Формуляр для параллельного вычисления значения полинома с занесёнными в него обозначениями промежуточных результатов и отвечающая ему вычислительная сеть.

обозначений следует разрешить также ***групповые описания промежуточных результатов.***

Например, вместо формулы

$$(b \times 2 + a) \times d + (a \times 2 + b) \times c$$

мы могли бы записать

$$\ulcorner(\mathbf{real}\ h1, h2) \equiv ((b \times 2 + a) \times d, (a \times 2 + b) \times c);\ h1 + h2 \lrcorner,$$

сохраняя коллатеральный характер подформул.

Теперь можно сформулировать и подпрограмму для параллельного вычисления значения полинома, которое до сих пор определялось формуляром (рис. 54):

```
funct poly4 ≡ (real a4, a3, a2, a1, a0, x)real:
  ⌈(real h1, k1, l1) ≡ (a4,    x×a3, x×x);
   (real h2, k2, l2) ≡ (h1+k1, l1×a2, l1×x);
   (real h3, k3, l3) ≡ (h2+k2, l2×a1, l2×x);
   (real h4, k4)     ≡ (h3+k3, l3×a0);
   h4+k4                                  ⌋
```

Пять нижних строчек соответствуют пяти тактам вычисления, см. рис. 89.

В алголе-68 (как и в паскале) такая возможность записи отсутствует: если мы не хотим линеаризовать ход вычислений, то приходится использовать систему вспомогательных под-

программ:

```
funct poly4 ≡ (real a4, a3, a2, a1, a0, x) real : p1(a4, x×a3, x×x, a2, a1, a0, x)
funct p1    ≡ (real h1, k1, l1, a2, a1, a0, x) real : p2(h1+k1, l1×a2, l1×x, a1, a0, x)
funct p2    ≡ (real h2, k2, l2, a1, a0, x) real : p3(h2+k2, l2×a1, l2×x, a0)
funct p3    ≡ (real h3, k3, l3, a0) real : p4(h3+k3, l3×a0)
funct p4    ≡ (real h4, k4) real : h4+k4
```

3.2. Программирование с переменными

Наиболее характерный аксессуар стиля программирования, ориентированного на машину фон-неймановского типа,— это программные переменные, для краткости называемые просто переменными.

3.2.1. Повторно используемые обозначения промежуточных результатов

По техническим причинам часто бывает удобно использовать „отслужившее" обозначение снова, после того как оно „освободилось". Такая возможность имеется, например, в случае предложения

$$\begin{aligned}&\lceil \mathbf{real}\ s1 \equiv a0 \times x + a1;\\ &\ \mathbf{real}\ s2 \equiv s1 \times x + a2;\\ &\ \mathbf{real}\ s3 \equiv s2 \times x + a3;\\ &\qquad s3 \times x + a4 \qquad \rfloor.\end{aligned}$$

Здесь обозначение $s1$ используется лишь при вычислении $s2$, $s2$ — лишь при вычислении $s3$ и т. д. Аналогичное справедливо для предложения

$$\lceil \mathbf{real}\ h1 \equiv a \uparrow 2;\ \mathbf{real}\ h2 \equiv h1 \uparrow 2;\ h2 \uparrow 2 \rfloor.$$

Имея в виду реализовать такую возможность, мы откажемся от требования, чтобы каждому обозначению соответствовал ровно один объект; наоборот, обозначению промежуточного результата нужно будет поочередно сопоставлять целую кучу объектов, условившись, что за данным обозначением всякий раз стоит последний из сопоставленных ему объектов.

В принципе можно вернуться к ситуации, когда каждый отдельный объект обладает своим собственным обозначением, снабдив невидимым индексом обозначение, повторно используемое в ходе вычислений, т. е. заменяя $s1$ на $s^{(1)}$, $s2$ на $s^{(2)}$, $s3$ на $s^{(3)}$, но считая видимым только s и аналогично заменяя $h1$ на $h^{(1)}$, $h2$ на $h^{(2)}$ и считая видимым только h. Обозначения s и h называют тогда ***программными переменными***. Переход от одного

заключенного в скобки индекса к следующему называется ***изменением состояния*** соответствующей переменной, а $s^{(i)}$ обозначает текущее ***значение*** переменной s.

3.2.2. Описания и присваивания

Подобно обозначениям промежуточных результатов, обозначения для переменных также вводятся с помощью описаний, которые определяют область связывания и область действия. Однако, чтобы подчеркнуть сменный характер соответствия значений обозначениям, для этой цели используют особые символы.

На алголе-68 упомянутые выше примеры запишутся так[1, 2]:

$$\lceil \mathbf{var\ real}\ s := a0 \times x + a1;\quad s := s \times x + a2;\quad s := s \times x + a3;\quad s \times x + a4 \rfloor$$

и

$$\lceil \mathbf{var\ real}\ h := a \uparrow 2;\ h := h \uparrow 2;\ h \uparrow 2 \rfloor.$$

Конструкции вида

$$\mathbf{var\ real}\ s := a0 \times x + a1$$

или

$$\mathbf{var\ real}\ h := a \uparrow 2$$

называются ***описаниями переменных*** (с ***инициализацией***), а конструкции вида

$$s := s \times x + a2$$

или

$$h := h \uparrow 2$$

— ***присваиваниями***. Формула, которой предшествуют описания переменных и, возможно, следующие за ними присваивания, впредь будет называться ***предложением*** (или ***сегментом***).

Предложения являются формулами. Например, для детерминированных a, b предложение

$$1 + \lceil \mathbf{var\ real}\ x := a + b;\ x := x \times x - a \times b;\ x \rfloor$$

эквивалентно формуле

$$1 + ((a + b) \times (a + b) - a \times b).$$

Конечно, сэкономить на обозначениях промежуточных результатов за счёт использования переменных удаётся далеко не всегда.

---

[1] В стандартном алголе-68 var нужно опустить.

[2] Ниже var — от variable (переменная). — *Прим. перев.*

Скажем, предложение (см. 3.1.1.3)

$$\lceil \mathbf{real}\ h1 \equiv (b \times 2 + a) \times d;\ \mathbf{real}\ h2 \equiv (a \times 2 + b) \times c;\ h1 + h2 \rfloor$$

можно переписать лишь в виде

$$\lceil \mathbf{var\ real}\ h := (b \times 2 + a) \times d;\ \mathbf{var\ real}\ k := (a \times 2 + b) \times c; h + k \rfloor,$$

поскольку обозначение h нельзя применять в новом смысле прежде[1], чем оно будет использовано в формуле $h + k$.

Целесообразно ни в каком предложении не производить присваиваний переменным, описанным в объемлющем предложении. Выполнение этого требования означает, что все переменные, которым производятся присваивания в теле некоторой подпрограммы, описаны в ней как *локальные*. Благодаря этому никакой вызов подпрограммы внутри формулы не может привести к „побочному эффекту", т. е. изменению глобальных переменных.

Как и описания промежуточных результатов, описания переменных могут встречаться в телах рекурсивных подпрограмм. Здесь, напротив, для каждого воплощения появляется своя собственная переменная; эти переменные различаются индексом воплощения.

В паскале присваивания записываются в точности так же, как в алголе; однако в паскале нет описаний переменных с инициализацией. В описании указывается (см. ниже) лишь обозначение переменной, а инициализация переменной записывается как обычное присваивание. Открывающая скобка предложения $\lceil$, записываемая как **begin**, сдвигается вправо, а именно ставится после описания переменной (и всех других следующих за ним описаний), а область связывания обозначения упомянутой переменной продолжается вплоть до соответствующего **end**. Аналогичным образом можно записать друг за другом сразу несколько описаний переменных.

3.2.3. Неизменные переменные

Обозначения промежуточных результатов в их прежнем смысле, т. е. когда они неизменно соответствуют одному и тому же значению, для отличия от программных переменных часто называют ***константами***. После введения понятия переменной можно обойтись и без констант, вместо них выступают тогда ***неизменные переменные***, т. е. переменные, присваивание кото-

---

[1] Слово „прежде" относится здесь к порядку исполнения. Чтобы получить порядок исполнения, нужно обратить ставший привычным за счёт распространения фортрана порядок записи переменной и выражения в описаниях с инициализацией и присваиваниях. Цузе (в исчислении планов, 1946 г.) и Рутисхаузер (1951 г.) использовали „правильный" порядок.

рым производится всего один раз (*'single assignment variable'*[1]). Такова точка зрения, представленная в языке паскаль.

Некоторые из примеров раздела 3.1.1.1 запишутся теперь на паскале следующим образом:

```
function istriangle(a,b,c : real) : Boolean ;
  var s : real ;
  begin s:=(a+b+c)/2 ;
        istriangle ⇐ (s≧a) ∧ (s≧b) ∧ (s≧c) end

function heron(a,b,c : real {istriangle(a,b,c)}) : real ;
  var s : real ;
  begin s:=(a+b+c)/2 ;
        heron ⇐ sqrt(s×(s−a)×(s−b)×(s−c)) end
```

3.2.4. Операторы

В алголе-68 и паскале принимаются следующие определения:

Оператор — это

либо присваивание некоторой формулы (причем в алголе-68 это может быть также условная формула или предложение) некоторой переменной,

либо заключенная в скобки-уголки[2] *непустая последовательность операторов, отделённых друг от друга точками с запятой (**составной оператор**).*

Блок *— это*

в алголе-68: *заключенная в скобки-уголки*[2] *и разделённая точкой с запятой пара, первым элементом которой служит непустая последовательность описаний, отделённых друг от друга точками с запятой или запятыми, а вторым — непустая последовательность операторов, разделённых точками с запятой;*

в паскале: *последовательность групп описаний, каждая из которых завершается точкой с запятой, и записанный вслед за этой последовательностью составной оператор.*

Выступающему в качестве тела подпрограммы предложению языка алгол-68 соответствует в паскале блок; при этом определение результата подпрограммы записывается и обрабатывается в стандартном паскале в точности как простое присваивание.

В алголе-68 блок является частным случаем оператора.

---

[1] „Переменная с одним-единственным присваиванием" (*англ.*). — *Прим. перев.*

[2] В стандартном алголе-68 и в паскале вместо скобок-уголков используются **begin** и **end**. См. также последнее подстрочное примечание в разделе 2.5.1.

Это означает, что блоки в алголе-68 могут быть вложены друг в друга (о „блочной структуре“ речь пойдёт в гл. 5). В паскале же блок не может служить оператором и описания могут встречаться лишь в начале тела подпрограммы.

3.2.5. *Примеры*

На рис. 90 наши прежние примеры записаны в форме подпрограмм, содержащих операторы, в двух параллельных столбцах на алголе-68 и паскале. В каждой из подпрограмм *pow8* и *horn4* многократно повторяется некоторый образец присваиваний; это становится ещё более явным, если в начале и конце предложения (соответственно блока) вставить „избыточные“ присваивания, как это сделано на рис. 91. В последнем из двух примеров, представленных на этом рисунке, присваивания различаются исключительно величинами *a*0, *a*1, *a*2, *a*3, *a*4. Если отвлечься от этих различий,то наблюдается следующий порядок расположения: вначале идёт описание переменной с инициализацией, за ним следует некий повторяющийся оператор, и заканчивается всё определением результата.

```
funct horn4 = (real a0, a1, a2, a3, a4, x)       | function horn4(a0,a1,a2,a3,a4,x:real)
                                      real :     |                                  :real;
                                                 |   var s : real;
  ⌈var real s := a0×x+a1 ;                       | begin s := a0*x+a1 ;
            s := s×x+a2 ;                        |       s := s*x+a2 ;
            s := s×x+a3 ;                        |       s := s*x+a3 ;
                 s×x+a4                          |       horn4 ⇐ s*x+a4
                                                 | end

funct pow8 = (real a) real :                     | function pow8(a : real) : real ;
                                                 |   var h : real ;
  ⌈var real h := a↑2 ;                           | begin h := sqr(a) ;
            h := h↑2 ;                           |       h := sqr(h) ;
                 h↑2            ⌋                |       pow8 ⇐ sqr(h)
                                                 | end

funct z = (real a, b, c, d) real :               | function z(a, b, c, d : real) : real ;
                                                 |   var h, k : real ;
  ⌈var real h := (b×2+a)×d ;                     | begin h := (b*2+a)*d ;
   var real k := (a×2+b)×c ;                     |       k := (a*2+b)*c ;
                 h+k            ⌋                |       z ⇐ h+k
                                                 | end
```

Рис. 90.

```
⌈var real h := a ;                    |   var h : real ;
      h := h↑2 ;                      | begin h := a ;
      h := h↑2 ;                      |       h := sqr(h) ;
      h := h↑2 ;                      |       h := sqr(h) ;
      h                          ⌋    |       h := sqr(h) ;
                                      |       pow8 ⇐ h
                                      | end
                                      |   var s : real ;
⌈var real s := 0 ;                    | begin s := 0 ;
      s := s×x+a0 ;                   |       s := s*x+a0 ;
      s := s×x+a1 ;                   |       s := s*x+a1 ;
      s := s×x+a2 ;                   |       s := s*x+a2 ;
      s := s×x+a3 ;                   |       s := s*x+a3 ;
      s := s×x+a4 ;                   |       s := s*x+a4 ;
      s                          ⌋    |       horn4 ⇐ s
                                      | end
```

Рис. 91.

3.2.6. Групповые описания переменных и групповые присваивания

В случае когда хотят повторно использовать целые наборы обозначений промежуточных результатов, групповые описания промежуточных результатов (см. 3.1.2) превращают в групповые описания переменных и групповые присваивания. Таким образом, для подпрограммы *poly4* из 3.1.2 мы получим

```
funct poly4 ≡ (real a4, a3, a2, a1, a0, x) real :
⌈ (var real h, k, l) := (a4, x×a3, x×x) ;
   (h, k, l) := (h+k, l×a2, l×x) ;
   (h, k, l) := (h+k, l×a1, l×x) ;
   (h, k) := (h+k, l×a0) ;
   h+k                                        ⌋
```

или, с избыточными присваиваниями в начале и конце, добавленными для большей регулярности (см. также рис. 92),

```
funct poly4 ≡ (real a4, a3, a2, a1, a0, x) real :
⌈ (var real h, k, l) := (0, 0, 1) ;
   (h, k, l) := (h+k, l×a4, l×x) ;
   (h, k, l) := (h+k, l×a3, l×x) ;
   (h, k, l) := (h+k, l×a2, l×x) ;
   (h, k, l) := (h+k, l×a1, l×x) ;
   (h, k, l) := (h+k, l×a0, l×x) ;
   h+k                                        ⌋
```

И двухрядную схему Горнера из 3.1.1.1 тоже можно записать с использованием групповых описаний и групповых присваива-

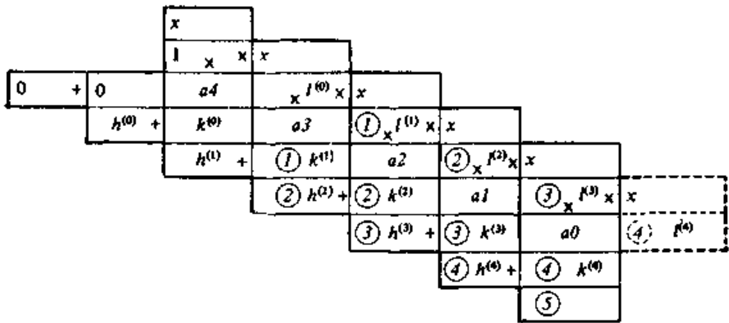


Рис 92 Расширенный формуляр для параллельного вычисления значения полинома с занесенными в него обозначениями промежуточных результатов.

ний, сохранив совместный (параллельно-последовательный) характер вычислений:

```
⌈ (var real t,s) = (0,0);
          (t,s) = (t×x+s, s×x+a0);
          (t,s) = (t×x+s, s×x+a1);
          (t,s) = (t×x+s, s×x+a2);
          (t,s) = (t×x+s, s×x+a3);
          (t,s) := (t×x+s, s×x+a4);
          t/s                          ⌋
```

Поскольку групповые присваивания получаются из групповых описаний промежуточных результатов за счёт экономии обозначений, само собой разумеется, что *сначала вычисляются все формулы в правой части, после чего переменные в левой части начинают обозначать полученные таким образом значения* („*одновременное присваивание*").

В алголе-68, как и в паскале, групповые описания переменных и групповые присваивания отсутствуют [1]. Приходится с помощью линеаризации записывать их в виде последовательности простых присваиваний; см. 3.3.4.

3.3. Итеративное программирование

3.3.1. Повторительные программы с точки зрения итерации

Свое главное обоснование переменные находят в линейной рекурсии Машина обработки формуляторов заводит новый фор-

[1] Предусмотренная в алголе-68 возможность совместной (параллельно-последовательной) записи присваиваний не годится для этой цели оператор ⌈$u := v,\ v := u$⌋ просто недопустим.

муляр для каждого воплощения. В точности также как эти формуляры можно пронумеровать по порядку, можно перенумеровать и перекрывающие друг друга графы результатов в этих

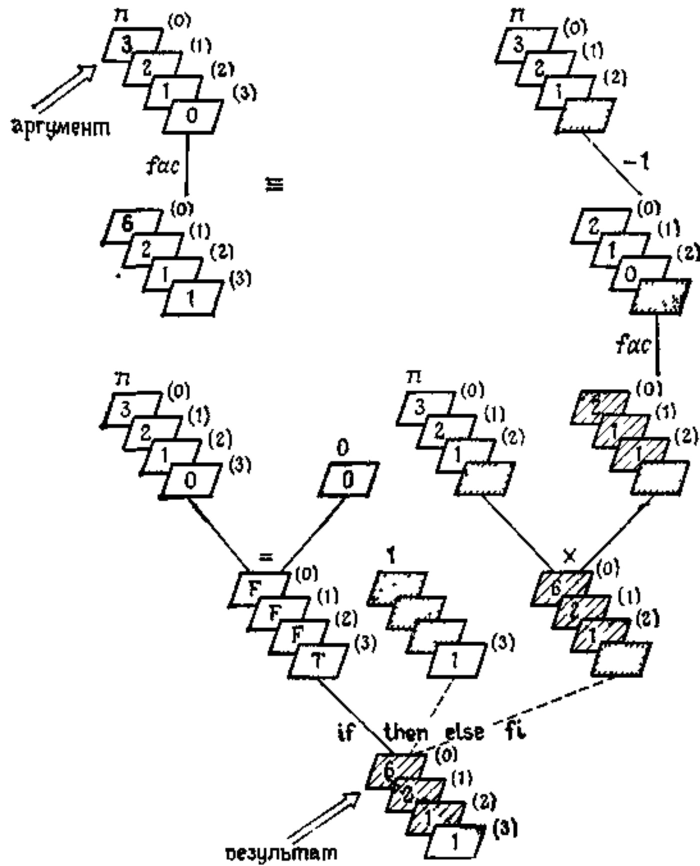


Рис. 93. Формуляр со стопками граф результатов для вычисления *fac*(3).

формулярах. После этого последовательность формуляров заменяется **одним** формуляром, содержащим последовательности граф результатов.

Невидимые (заключенные в скобки) индексы будут тогда обозначать то воплощение, к которому относится промежуточный результат. На рис. 93 изображён такой формуляр для вычисления функции *fac* (см. также рис. 72). Затемнённые графы не используются в соответствующем воплощении. Заштрихованные графы заполняются в обратной последовательности, при

выполнении „задержанных" операций Результат *fac*(3) оказывается в конце концов в 0-м воплощении. На рис 94 рассмотрен пример повторительной рекурсии — вычисление *gcd* (см.

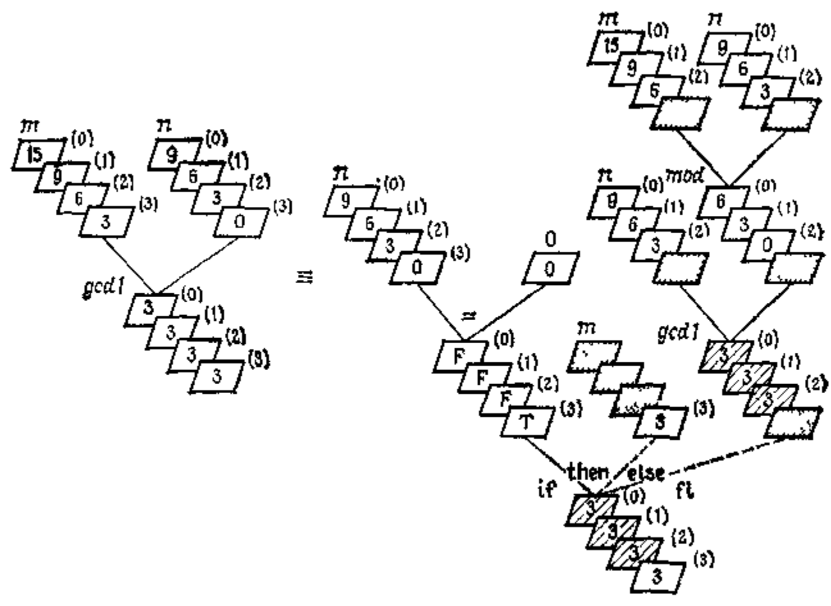


Рис 94 Формуляр со стопками граф результатов для вычисления *gcd1* (15, 9).

рис. 73). Так как здесь „задержанных" операций нет, одно и то же значение результата распространяется в конце концов во все воплощения.

Далее, ни на каком шаге нам не надо возвращаться к более „глубоко" расположенным промежуточным результатам Поэтому можно отказаться от нумерации, с помощью которой различаются воплощения, и работать со стопкой граф, в которой важна только последняя графа, т. е вместо последовательности промежуточных результатов использовать переменную (см 3 2). Эта особая ситуация относится ко всем графам промежуточных результатов, если имеет место повторительная рекурсия Таким образом, рекурсия вырождается в простое *повторение*, управляемое некоторым условием.

Однако доведём наш пример до конца. Напрашивается такое упрощение формуляра на рис 94 — зачерненные графы опустить, заштрихованные - объединить Можно также отождествить между собой графы аргументов для *m* и *n* (рис. 95). Для

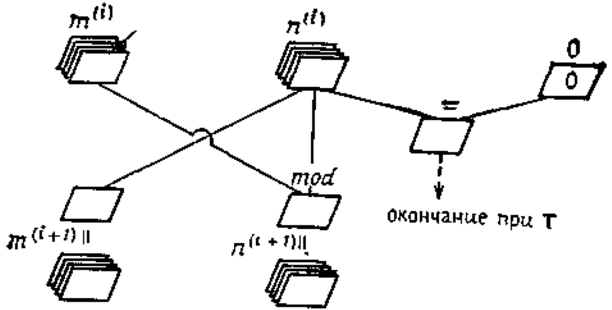


Рис 95 Основная форма итеративного программного формуляра для вычисления функции *gcd1*

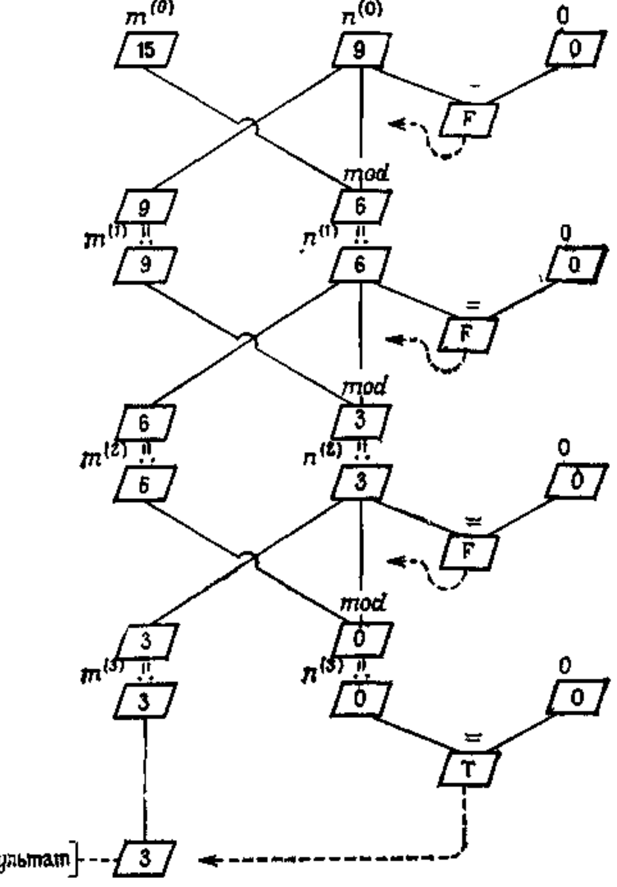


Рис 96 „Итеративный формуляр, полученный „разверткой" программного формуляра на рис 95

„развёртки" повторительной рекурсии в плоскую таблицу используют (см. рис. 95) итерируемый ***программный формуляр***, содержащий правило итераций

$$(m^{(i+1)},\ n^{(i+1)}) := (n^{(i)},\ mod\,(m^{(i)},\ n^{(i)})).$$

В результате получается представленный на рис. 96 „развёрнутый" формуляр для вычисления $gcd(15, 9)$. Аналогичным образом устроено ньютоново правило итераций для вычисления последовательных приближений к квадратному корню из заданного (машинного) вещественного числа a:

$$x^{(i+1)} := (x^{(i)} + a/x^{(i)})/2.$$

Такие итерационные правила следует повторять под управлением соответствующего условия (на рис. 95, 96 это управление отмечено пунктирными стрелками).

3.3.2. Повторение

Таким образом, вырождение повторительной рекурсии выражается введением ***оператора цикла с условием продолжения***: некоторый программный формуляр, т. е. определённый оператор, повторяется до тех пор, пока не перестанет выполняться определённое условие.

И здесь алгол-68 и паскаль используют похожие, хотя и отличающиеся в деталях способы записи [1, 2]:

while ⟩условие⟨	**while** ⟩условие⟨ **do**
do ⟩оператор⟨ **od**	⟩оператор⟨

На машине обработки формуляров повторение характеризуется повторным использованием одного и того же формуляра со сложенными стопкой графами, т. е. итеративным применением некоторого программного формуляра.

Оператор цикла является ***отвергающим*** в том смысле, что если условие с самого начала не выполнено, то подлежащий повторению оператор не исполняется ни разу [3].

Примем, что *оператор цикла — это тоже оператор.*

---

[1] В стандартном паскале нет нужды заключать в скобки **begin end** одно-единственное присваивание или один-единственный оператор цикла. В алголе-68 скобки **do.od** действуют как скобки-уголки.

[2] Ниже **while** и **do** означают соответственно „пока" и „делай"; **od** — это „перевёрнутое" **do**. — *Прим. перев.*

[3] В паскале наряду с этим имеется также неотвергающий оператор цикла, используемый как сомнительной ценности сокращение.

```
funct gcd1 ≡ (int m, n                   | function gcd1 (m, n : integer
          co m ≧ 0 ∧ n ≧ 0 co) int :     |        {(m ≧ 0) ∧ (n ≧ 0)}) integer;
                                         |   var x, y, z : integer;
 ⌈var int x := m ;                       | begin x := m;
  var int y := n ;                       |       y := n;
  var int z ;                            |
  while y > 0                            |   while y > 0 do
  do z := x ;                            |     begin z := x;
    while z ≧ y                          |       while z ≧ y do
    do z := z − y od ;                   |       z := z − y;
    x := y ;                             |       x := y;
    y := z                               |       y := z
                    od ;                 |     end;
  x                                   ⌋  |   gcd1 ⇐ x
                                         | end
```

Рис. 97

Тогда повторяемый оператор сам может быть оператором цикла или содержать его, как в следующем примере[1]:

```
funct gcd1 ≡ (int m, n co m ≧ 0 ∧ n ≧ 0 co) int :
 ⌈var int x := m ;
  var int y := n ;
  while y > 0
  do var int z := x ;
     while z ≧ y
     do z := z − y od ;
     x := y ;
     y := z           od ;
  x                           ⌋
```

В повторяемом операторе может встречаться и описание переменной, как, скажем, **var int** *z* в приведённом выше примере. Если внешнее повторение выразить посредством соответствующей повторительной рекурсии, то для каждого воплощения появится своя собственная переменная. Таким образом, машина обработки формуляров работает со стопкой переменных $z^{(0)}, z^{(1)}, z^{(2)}, \ldots, z^{(n)}$ (каждая из которых в свою очередь обозначает стопку промежуточных результатов). Для *gcd1* (15, 9) получим

$z^{(0)}$ [15, 6] $z^{(1)}$ [9, 3] $z^{(2)}$ [6, 3, 0]

[1] Систематическое рассмотрение этого вопроса будет проведено в разделах 3.3.2.2 и 3.3.4.

В паскале описание переменных внутри оператора цикла не допускается. Такие описания должны быть вынесены „наружу" — в начало ближайшего охватывающего блока. Хотя в алголе-68 это тоже допускается, однако в этом мало смысла, поскольку теряется результат инициализации (см. рис. 97).

3.3.2.1. Итеративные программы

Вообще, справедливо такое утверждение: любую рекурсивную повторительную подпрограмму, тело которой состоит из одной-единственной альтернативы, можно сразу же записать с использованием переменных и оператора цикла. Действительно, работа машины обработки формуляров в данном случае состоит исключительно в том, что вычисляются новые фактические значения для вызова следующего воплощения, причём эти вычисления всякий раз производятся по одним и тем же формулам. Для этой цели необходимо иметь одновременно столько программных переменных, сколько неконстантных параметров у подпрограммы. Условием оператора цикла будет то самое условие, которое охраняет ветвь с рекурсивным вызовом. В связи с этим рассмотрим еще раз ход реализации рекурсии для *gcd1* (15, 9), представленный на рис. 94.

В общем и целом ход вычислений отражается соответствующими последними значениями $m^{(i)}$, $n^{(i)}$ двух переменных *mvar* и *nvar*:

mvar	*nvar*
15	9
9	6
6	3
3	0

В конце вычислений *mvar* обозначает результат.

Предполагая примитивность операции *mod*, получим версию, основанную на повторении группового присваивания паре переменных (*mvar*, *nvar*):

```
funct gcd1 ≡ (int m, n
        co m ≧ 0 ∧ n ≧ 0 co) int:


⌈(var int mvar, nvar) := (m, n);
  while nvar > 0
  do (mvar, nvar) :=
        (nvar, mod (mvar, nvar)) od;
  mvar                          ⌋
```

```
function gcd1 (m, n : integer
        {m ≧ 0 ∧ n ≧ 0}) : integer
  var mvar, nvar : integer;
begin
  (mvar, nvar) := (m, n);
  while nvar > 0 do
  (mvar, nvar) :=
        (nvar, mod (mvar, nvar));
  gcd1 ⇐ mvar
end
```

В этом примере обнаруживается типичное деление тела подпрограммы на три части:

(a) (Групповое) описание и инициализация переменных. Инициализация осуществляется параметрами (или их фактическими значениями при вызове внутри некоторой объемлющей подпрограммы).

(b) Повторение (группового) присваивания, соответствующего замещению параметров, при выполнении условия продолжения рекурсии.

(c) Фиксация (получение, вывод) результата.

Это проявляется и при рассмотрении подпрограммы *sel* из примера (g) раздела 2.3.2. Здесь мы получаем

```
funct sel ≡ (string a, int i
        co 1 ≤ i ∧ i ≤ length(a) co) char:

 ⌈(var string avar, var int ivar) := (a, i);
  while ivar > 1
  do (avar, ivar) := (rest(avar), ivar − 1) od;
  first(avar)                                  ⌋
```

```
function sel (a : string; i : integer
        {(1 ≤ i) and (i ≤ length(a))}) : char;
  var avar : string; ivar : integer;
begin
  (avar, ivar) := (a, i);
  while ivar > 1 do
  (avar, ivar) := (rest(avar), ivar − 1);
  sel ⇐ first (avar)
end
```

Точно так же можно преобразовать в итеративную форму подпрограммы типа *invert* (см. 2.6.1).

Аналогично можно поступать и с такими подпрограммами, как *issorted* (см. 2.4.1.5) или *isprim* (см. 2.4.1.3), только в случае последней подпрограммы, основанной на функции *ispr* (см. 2.4.1.3) надо предварительно преобразовать фигурирующее в *ispr* условие:

```
funct ispr ≡ (int n, m
        co 2 ≤ m ≤ n co) bool:

  if n mod m ≠ 0
  then ispr(n, m + 1)
  else m = n          fi
```

```
function ispr(n, m : integer
        {2 ≤ m ≤ n}) : Boolean;
begin
  if n mod m ≠ 0
  then ispr ⇐ ispr(n, m + 1)
  else ispr ⇐ m = n
end
```

(Отметим справедливость импликации $n \bmod m \neq 0 \Rightarrow m \neq n$.) В результате вставки (параметр *n* подпрограммы *ispr* оказывается неизменным[1]) получим подпрограмму, представленную на рис. 98.

---

[1] Параметр *n* подпрограммы *ispr* можно было бы опустить ещё раньше, после подчинения этой подпрограммы подпрограмме *isprim* (см. подпрограмму *ispr* в разделе 2.5.2.1).

```
funct isprim ≡ (int n
              co n ≧ 1 co) bool:



  if n = 1
  then false
  else var int mvar := 2;
     while n mod mvar ≠ 0
     do mvar := mvar + 1 od;
     mvar = n               fi
```

```
function isprim (n : integer
              {n ≧ 1}) : Boolean;
  var mvar : integer;
begin
  if n = 1
  then isprim ⇐ false
  else begin mvar := 2;
        while n mod mvar ≠ 0 do
        mvar := mvar + 1;
        isprim ⇐ mvar = n
        end
end
```

Рис. 98.

Повторительные подпрограммы, в телах которых встречается несколько (гладких) вызовов, можно преобразовать в рассмотренную выше форму с одним-единственным (гладким) вызовом: для этого нужно только распространить действие стражей, под охраной которых находятся отдельные вызовы, на аргументы вызовов. Как это сделать, проиллюстрировано на примере в 3.3.5.

Итак, если ограничиться исключительно повторительными подпрограммами, то можно обойтись использованием одних лишь переменных и операторов цикла. При этом мы по-прежнему можем считать их способом сокращённой записи для особенно простого случая рекурсии. С другой стороны, мы можем также, следуя историческому развитию, подвести под всё это внешне нерекурсивную семантику; для этого нужно не только отождествить разные воплощения формуляра, но и вынести наружу стопки промежуточных результатов, т. е. переменные.

Тем самым выполнение вычисления („исполнение") сводится к ***движению*** по программному формуляру, а ***состояние*** вычисления определяется последними текущими значениями используемых программных переменных и позицией в программном формуляре.

Разумеется, групповые присваивания могут выполняться параллельно, если для этого имеется необходимое оборудование.

Подпрограммы, которые, подобно рассмотренным в этом разделе, содержат повторения, но не обнаруживают рекурсии, также называют ***итеративными***.

Формальная семантика итеративных формулировок будет рассмотрена в гл. 8.

3.3.2.2. Иерархические повторительные системы и вложенные циклы

Наконец, мы можем рассмотреть и иерархическую систему (*gcd1*, *mod*) из примера (d) раздела 2.3.2. Здесь лучше всего избавиться от неизменного второго параметра *n* подпрограммы *mod* (после подчинения этой подпрограммы подпрограмме *gcd1* (см. 2.5.2.1)). Получим

```
funct mod1 ≡ (int u                     | function mod1(u : integer
              co u ≥ 0 co) int :        |                  {u ≥ 0}) : integer ;
                                        |   var uvar : integer ;
 ⌈var int uvar := u ;                   | begin uvar := u ;
  while uvar ≥ n                        |     while uvar ≥ n
  do uvar := uvar − n   od ;            |     do uvar := uvar − n ;
  uvar                              ⌋   |     mod1 ⇐ uvar
                                        | end
```

Теперь надо описание подпрограммы *mod* записать внутри подпрограммы *gcd1*. При этом глобальный параметр *n* подпрограммы *mod1* должен быть замещён значением нелокальной переменной *nvar*. Это даёт подпрограмму, представленную на рис. 99.

```
funct gcd1 ≡ (int m, n                    | function gcd1(m, n : integer
              co m ≥ 0 ∧ n ≥ 0 co) int :  |            {m ≥ 0 ∧ n ≥ 0}) : integer ;
                                          |   var mvar, nvar : integer ;
⌈funct mod1 ≡ (int u co u ≥ 0 co) int :   |   function mod1(u : integer {u ≥ 0}) : integer ;
                                          |     var uvar : integer ;
  ⌈var int uvar := u ;                    |   begin uvar := u ;
   while uvar ≥ nvar                      |     while uvar ≥ nvar
   do uvar := uvar − nvar od ,            |     do uvar := uvar − nvar ;
   uvar                    ⌋ ;            |     mod1 ⇐ uvar
                                          |   end ;
 (var int mvar, nvar) := (m, n) ;         | begin (mvar, nvar) := (m, n) ;
 while nvar > 0                           |   while nvar > 0
   do (mvar, nvar) :=                     |     do (mvar, nvar) :=
         (nvar, mod1(mvar)) od ;          |           (nvar, mod1(mvar)) ;
 mvar                                 ⌋   |   gcd1 ⇐ mvar
                                          | end
```

Рис. 99.

Если мы захотим подставить тело подпрограммы *mod1* в *gcd1*, то инициализацию нужно видоизменить так: **var int** *uvar* := *mvar*, а в заключительном групповом присваивании

$$(mvar,\ nvar) := (nvar,\ mod1\,(mvar))$$

вместо *mod1*(*mvar*) использовать полученный в *uvar* результат. Это приводит к подпрограмме, представленной на рис. 100. Здесь

```
funct gcd1 ≡ (int m, n                        | function gcd1 (m, n : integer
        co m ≧ 0 ∧ n ≧ 0 co) int:             |          {(m ≧ 0) ∧ (n ≧ 0)}) : integer;
                                              |   var mvar, nvar, uvar : integer;
                                              | begin
 ⌈(var int mvar, nvar) := (m, n);             |   (mvar, nvar) := (m, n);
  while nvar > 0                              |   while nvar > 0
  do var int uvar := mvar;                    |   do begin uvar := mvar;
      while uvar ≧ nvar                       |         while uvar ≧ nvar
      do uvar := uvar − nvar od;              |         do uvar := uvar − nvar;
      (mvar, nvar) := (nvar, uvar) od;        |         (mvar, nvar) := (nvar, uvar)
                                              |      end;
  mvar                                    ⌋   |   gcd1 ⇐ mvar
                                              | end
```

Рис. 100.

появляются вложенные операторы цикла, поскольку в теле подпрограммы *gcd1*, описанной в примере (d) раздела 2.3.2, вызов подпрограммы *mod* вложен в вызов подпрограммы *gcd1*. Иначе обстоит дело в случае следующей системы (*ldfac*, *ldr*), (ср. с подпрограммами *facr* из 2.4.1.3 и *ld* из 2.4.1.5):

```
funct ldfac ≡ (int y, n co y ≧ 1 ∧ n ≧ 0 co) int:
   if n = 0
   then ldr(0, y)
   else ldfac(y × n, n − 1) fi

funct ldr ≡ (int x, m co m ≧ 1 co) int:
   if m = 1
   then x
   else ldr(x + 1, m div 2) fi
```

Для *ldfac* получается итеративная формулировка с последовательными операторами цикла:

```
funct ldfac ≡ (int y, n co y ≧ 1 ∧ n ≧ 0 co) int:
 ⌈(var int yvar, nvar) := (y, n);
  while nvar > 0
  do (yvar, nvar) := (yvar × nvar, nvar − 1) od;
  (var int xvar, mvar) := (0, yvar);
  while mvar > 1
  do (xvar, mvar) := (xvar + 1, mvar div 2) od;
  xvar                                          ⌋
```

При вызове *ldfac*(1, *n*) вычисляется округленное значение двоичного логарифма от *n*!

Что касается паскалевской нотации для *ldfac*, см. рис. 104. Между прочим, переменные *mvar* и *yvar* можно отождествить, равно как и переменные *xvar* и *nvar*, поскольку подпрограммы *ldfac* и *ldr* должны выполняться строго друг за другом, а типы переменных согласованы. Заметим, что после окончания первого оператора цикла переменная *nvar* имеет значение 0.

3.3.2.3. Диаграммы Насси — Шнейдермана

Компактное графическое представление хода исполнения программного формуляра, отражающее структуру вложенности

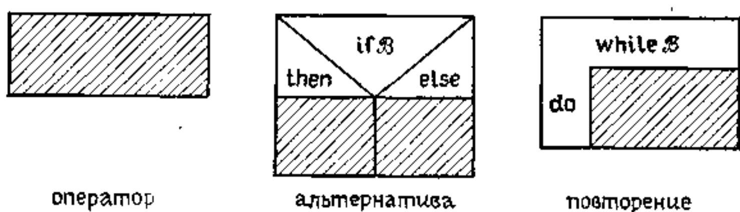


Рис. 101. Символы, из которых строятся диаграммы Насси—Шнейдермана.

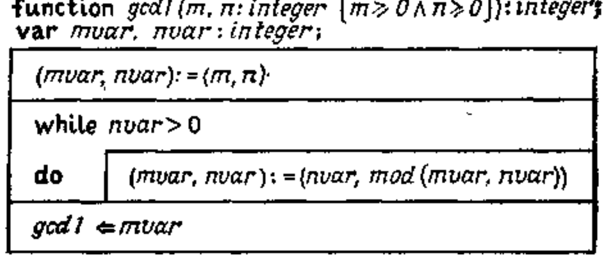


Рис. 102. Диаграмма Насси — Шнейдермана для подпрограммы *gcd1*, опирающейся на подпрограмму *mod*.

составляющих конструкций, даёт ***диаграмма Насси — Шнейдермана***. Базовыми элементами, из которых строятся такие диаграммы, служат изображённые на рис. 101 символы оператора, альтернативы и цикла, внутри которых может записываться соответствующий текст на принятом языке программирования; эти символы могут быть произвольным образом вложены друг в друга. Тем самым мы получаем способ наглядно представить структуру даже весьма сложных итеративных программ. Простой пример для подпрограммы *gcd1* из 3.3.2.1 показан на рис. 102.

По поводу употребления альтернативы см. 3.3.5.1. Пример вложенной структуры представлен на рис. 103 (для подпрограммы *gcd1* из 3.3.2.2).

function *gcd1* (*m*, *n*: *integer* [$(m \geqslant 0) \wedge (n \geqslant 0)$]): *integer*;
var *mvar*, *nvar*, *uvar*: *integer*;

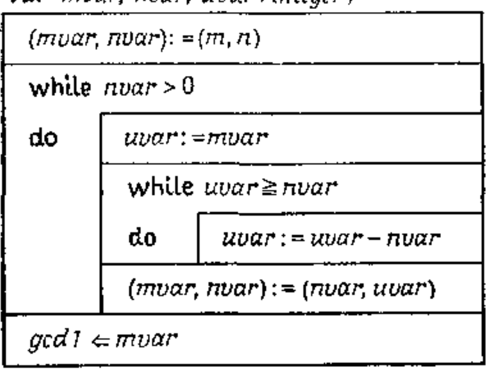


Рис. 103. Диаграмма Насси — Шнейдермана для *gcd1*.

function *ldfac* (*y*, *n*: *integer* [$y \geqslant 1 \wedge n \geqslant 0$]): *integer*;
var *yvar*, *nvar*, *xvar*, *mvar*: *integer*;

```
(yvar, nvar) := (y, n)
while nvar > 0
do    (yvar, nvar) := (yvar * nvar, nvar − 1)
(xvar, mvar) := (0, yvar)
while mvar > 1
do    (xvar, mvar) := (xvar + 1, mvar div 2)
ldfac ⇐ xvar
```

Рис. 104. Диаграмма Насси — Шнейдермана для *ldfac*.

Для сравнения на рис. 104 изображена диаграмма с двумя циклами, которые не вложены друг в друга; эта диаграмма соответствует подпрограмме *ldfac* из 3.2.2.2.

3.3.3. Решение задач с помощью итеративных форм

Один из надёжных способов решения задач состоит в следующем: сначала формируем повторительную рекурсию, а потом переходим к итеративной формулировке.

3.3.3.1

Пусть надо вычислить a^{2^n}. Используя соотношения

$$a^{2^0} = a \text{ и } a^{2^n} = a^{2 \cdot 2^{n-1}} = (a \times a)^{2^{n-1}} \quad (n \geqslant 1),$$

приходим к повторительной рекурсии

funct *potz* ≡ (**real** *a*, **int** *n* **co** *n* ≧ 0 **co**) **real** : **if** *n* = 0 **then** *a* ▯ *n* > 0 **then** *potz*(*a*↑2, *n* − 1) **fi**	**function** *potz*(*a* : *real*; *n* : *integer* {*n* ≧ 0}) : *real*; **begin** **if** *n* = 0 **then** *potz* ⇐ *a* ▯ *n* > 0 **then** *potz* ⇐ *potz*(*sqr*(*a*), *n* − 1) **end**

Отсюда получается итеративная формулировка:

funct *potz* ≡ (**real** *a*, **int** *n* **co** *n* ≧ 0 **co**) **real** : ⌈(**var real** *avar*, **var int** *nvar*) := (*a*, *n*); **while** *nvar* > 0 **do** (*avar*, *nvar*) := (*avar*↑2, *nvar* − 1) **od** ; *avar* ⌋	**function** *potz*(*a* : *real*; *n* : *integer* {*n* ≧ 0}) : *real*; **var** *avar* : *real*; *nvar* : *integer*; **begin** (*avar*, *nvar*) := (*a*, *n*); **while** *nvar* > 0 **do** (*avar*, *nvar*) := (*sqr*(*avar*), *nvar* − 1); *potz* ⇐ *avar* **end**

Рассмотренная ранее в 3.2.5 подпрограмма *pow8* содержится здесь как частный случай — „развёртка" вызова *potz*(*a*, 3) даст как раз *pow8*(*a*) (частичное вычисление, см. 2.6.3).

3.3.3.2

Для получения алгоритма вычисления значения полинома произвольной степени n также нет нужды искать „обобщение" подпрограммы *horn4*. Последовательность коэффициентов *a0*, *a1*, *a2*, ... полинома будет теперь (непустым) объектом a сорта **sequ real** и длины $n + 1$: $a0 = a[1]$, $a1 = a[2]$ и т. д. Если для $i \leqslant length(a) - 1$ обозначить

$$a[1]\,x^i + a[2]\,x^{i-1} + \ldots + a[i+1]$$

через $p(a, i, x)$, то мы имеем

для $i > 0$: $p(a, i, x) = p(a, i-1, x) \times x + a[i+1]$,

а также $p(a, 0, x) = a[1]$.

Получающаяся при этом рекурсивная подпрограмма

```
funct p ≡ (sequ real a, int n, real x
            co a ≠ ◇ ∧ 0 ≦ n ∧ n ≦ length(a) − 1 co) real:
  if n = 0 then a[1]
  ▯ n > 0 then p(a, n−1, x) × x + a[n+1] fi
```

(с параметром n вместо i) не является, к сожалению, повторительной и не ведёт непосредственно к итеративной формулировке. (Относительно операции .[.] см. пример (g) из 2.3.2.)

К повторительной рекурсии можно прийти, если взяться за дело „с другого конца". Обозначим через $f(a, n, i, s, x)$ выражение

$$(\ldots((s \times x + a[i]) \times x + a[i+1]) \times x + \ldots) \times x + a[n+1],$$

$1 \leqslant i \leqslant n+1$. Тогда $f(a, n, 1, 0, x) = p(a, n, x)$ (вложение!). Далее, имеем

для $i \leqslant n$: $f(a, n, i, s, x) = f(a, n, i+1, s \times x + a[i], x)$,

а также $f(a, n, n+1, s, x) = a[n+1]$.

Тем самым (избавляясь от параметров a, n, x подпрограммы f после её подчинения объемлющей подпрограмме) мы получаем

```
funct horn ≡ (sequ real a, int n, real x
                 co a ≠ ◇ ∧ n = length(a) − 1 co) real:
  ⌈funct f ≡ (int i, real s co 1 ≦ i ∧ i ≦ n+1 co) real:
     if i = n+1 then a[n+1]
     ▯ i ≦ n then f(i+1, s × x + a[i]) fi;
   f(1, 0)                                              ⌋
```

Отсюда уже непосредственно следует итеративная формулировка:

```
funct horn ≡ (sequ real a, int n, real x
                 co a ≠ ◇ ∧ n = length(a) − 1 co) real:
  ⌈(var int i, var real s) := (1, 0);
   while i ≦ n do
   (i, s) := (i+1, s × x + a[i]) od;
   s                                                    ⌋
```

Сравните с этой подпрограммой подпрограмму *horn4* из 3.2.5.

3.3.3.3

Аналогично можно рассмотреть и задачу параллельного вычисления значения полинома. Если через $g(a, n, i, h, k, l, x)$ обозначить

$$h + k + a[n+1-i] \times l + a[n-i] \times l \times x + \ldots + a[1] \times l \times x^{n-i},$$

$1 \leqslant i \leqslant n+1$, то $g(a, n, 0, 0, 0, 1, x)$ будет задавать значение полинома. Далее, для $i \leqslant n$ справедливо рекурсивное соотношение

$$g(a,n,i,h,k,l,x) = g(a,n,i+1,h+k,a[n+1-i] \times l, l \times x, x),$$

а для $i = n+1$ также

$$g(a,n,n+1,h,k,l,x) = h+k.$$

Параметры a, n и x являются неизменными. Пожалуй, мы можем на этот раз пропустить „промежуточную стоянку" — рекурсивную попрограмму — и сразу записать итеративную формулировку (сравните её с подпрограммой *poly4* из 3.2.6):

```
funct poly ≡ (sequ real a, int n, real x
                          co a ≠ ◊ ∧ n = length(a) − 1 co) real :
 ⌈(var int i, var real h, k, l) := (0,0,0,1) ;
  while i ≤ n
  do (i, h, k, l) := (i+1, h+k, a[n+1−i]×l, l×x) od ;
  h+k                                                   ⌋
```

Такое итеративное вычисление значения полинома выгодно, когда имеются устройства для параллельного выполнения группового присваивания.

3.3.3.4

Бывают и такие случаи, когда задача с самого начала формулируется так, что ничего не стоит сразу же записать итеративное решение. Примером может служить задача

funct $q \equiv$ (**string** a, b, **int** n **co** $n \geqslant 0$) **string**:

«цепочка, получающаяся конкатенацией повторённого n раз знака a, повторенного n раз знака b и повторённого ещё n раз знака a,— коротко $a^n b^n a^n$»

с решением

```
funct q ≡ (string a, b, int n co n ≥ 0 co) string :
  ⌈(var int i, var string s, t) := (n, ◊, ◊) ;
   while i > 0
   do (i, s, t) := (i−1, s+b, t+a) od ;
   t+s+t                                        ⌋
```

Опыт показывает, что при решении задач такого типа нередко случаются ошибки — здесь, например, легко просчитаться на $+1$ или на -1 — и что необходимо проверять правильность да-

же таких простых программ в отношении их соответствия постановке задачи.

Такую выполняемую задним числом проверку, или ***верификацию*** итеративной программы, так сказать „свалившейся с потолка", производят с помощью какого-либо ***инварианта итерации***. В нашем случае для этой цели подходит выражение

$$Q(a, b, i, s, t) \equiv t + a^i + s + b^i + t + a^i.$$

В самом деле, после очередного шага итерации мы получим

$$\begin{aligned} & Q(a, b, i-1, s+b, t+a) \\ & = (t+a) + a^{i-1} + (s+b) + b^{i-1} + (t+a) + a^{i-1} \\ & = t + a^i + s + b^i + t + a^i = Q(a, b, i, s, t). \end{aligned}$$

В начале итерации Q имеет значение

$$Q(a, b, n, \lozenge, \lozenge) = a^n + b^n + a^n.$$

Таким образом, это же значение будет иметь Q и в конце итерации, когда $i = 0$. Итак, выражение $Q(a, b, 0, s, t) = t + s + t$ действительно даёт требуемый результат.

Но нахождение такого инварианта равносильно соответствующему рекурсивному определению. С тем же успехом мы могли бы с самого начала искать такое определение. В нашем случае это было бы вложение

```
funct q ≡ (string a, b, int n co n≧0 co) string:
  ⌈funct Q ≡ (int i, string s, t co i≧0 co) string:
     if i>0 then Q(i−1, s+b, t+a)
            else t+s+t                    fi ;
   Q(n, ◇, ◇)                                ⌋
```

Подобно тому как это делалось в 3.3.3.1 и последующих разделах, построение решения начинается здесь выводом рекурсии из приведенного выше выражения Q.

3.3.4. Линеаризация

Групповые описания и присваивания возникают естественным образом, когда повторительная рекурсия записана посредством операторов цикла. Однако в стандартном алголе-68 и паскале приходится по соображениям машинной ориентации переходить к последовательности простых описаний и присваиваний (***линеаризация***). Как показывает уже пример группового присваивания

$$(mvar, nvar) := (nvar, uvar)$$

(см. подпрограмму *gcd1* из 3.3.2.2), для присваиваний линеаризацию нельзя выполнять в произвольном порядке; например, вариант

$$mvar := nvar;\ nvar := uvar$$

вполне корректен, вариант же

$$nvar := uvar;\ mvar := nvar$$

„переписывает" переменную, старое значение которой ещё потребуется. Кроме того, пример

$$(a, b) := (b, a)$$

показывает, что линеаризация не всегда возможна без введения обозначений промежуточных результатов.

С другой стороны, на основе указанной в конце раздела 3.2.6 семантики группового присваивания всегда возможна линеаризация, реализуемая в соответствии с принципом "*master-slave*"[1]. А именно, для линеаризации n-членного группового присваивания $(v_1, v_2, \ldots, v_n) := (E_1, E_2, \ldots, E_n)$ вводят n вспомогательных обозначений промежуточных результатов, получающих одновременно значения $E_1, E_2, \ldots, E_n$. Затем эти промежуточные результаты одновременно присваиваются переменным $v_1, v_2, \ldots, v_n$. Как получающееся таким образом групповое описание промежуточных результатов, так и последующее групповое присваивание могут быть линеаризованы теперь в любой последовательности.

Пример. Групповое присваивание

$$(a, b) := (b, a),$$

где a, b имеют произвольный вид **var** λ, преобразуется сначала в

$$(\lambda\ aconst, bconst) \equiv (b, a);$$
$$(a, b) := (aconst, bconst),$$

откуда линеаризацией получаем

$$\lambda\ aconst \equiv b;\ \lambda\ bconst \equiv a;\ b := bconst;\ a := aconst.$$

Иногда можно ещё сэкономить на некоторых из вспомогательных обозначений, скажем в нашем примере обойтись без *bconst*, поскольку пару операторов

$$\lambda\ bconst \equiv a;\ b := bconst$$

можно сократить до

$$b := a.$$

---

[1] „Хозяин — раб" (*англ.*). — *Прим. перев.*

Часто за счёт использования подходящей последовательности удаётся обойтись вообще без вспомогательных обозначений, как это нам удалось выше для присваивания

$$(mvar, nvar) := (nvar, uvar).$$ [1]

В некоторых примерах, скажем в случае присваивания

$$(avar, nvar) := (sqr(avar), nvar - 1)$$

(см. подпрограмму *potz* из 3.3.3.1), линеаризацию можно выполнять в любой последовательности — годится как

$$avar := sqr(avar);\ nvar := nvar - 1,$$

так и

$$nvar := nvar - 1;\ avar := sqr(avar).$$

Если надлежащим образом линеаризовать групповые описания и присваивания итеративной подпрограммы *gcd1* из 3.3.3.2, то после подходящего переименования (связанных) переменных получим записанную в начале раздела 3.3.2 строго последовательную формулировку для *gcd1*.

3.3.5. Условные операторы

В 3.3.2 мы ограничились рассмотрением лишь таких повторительных подпрограмм, тела которых содержат одну-единственную альтернативу. Это ограничение „двухстрочной" рекурсией является излишним.

Рассмотрим в качестве примера повторительную подпрограмму[2]

funct *mc* ≡ (**int** *n* **co** *n* > 0 **co**) **int**:
  if *n* = 1
  then 1
  else if odd *n*
    then *mc*(3 × *n* + 1)
    else *mc*(*n* ÷ 2) **fi fi**
	function *mc*(*n* : *integer* {*n* > 0}) : *integer*;
	begin if *n* = 1
	  then *mc* ⇐ 1
	  else if *odd*(*n*)
	    then *mc* ⇐ *mc*(3**n* + 1)
	    else *mc* ⇐ *mc*(*n* **div** 2)
	end

Её также можно заменить оператором цикла, в повторяемом операторе которого некоторой переменной *nvar* альтернативно

[1] Отметим, что это не проходит в случае

$$(mvar, nvar) := (nvar, mod1(mvar)),$$

в частности потому, что подпрограмма *mod*1 использует значение *nvar* как глобальный параметр.

[2] Если эта подпрограмма завершается, то выдаёт 1. Однако доказательство того, что она завершается для любого положительного натурального числа *n*, пока не найдено.

присваивается либо *nvar* ÷ 2 (соотв. *nvar* **div** 2), либо 3 × *nvar* (в стандартном паскале такое присваивание не допускается):

```
funct mc ≡ (int n co n > 0 co) int:       | function mc(n : integer {n > 0}) : integer;
                                          |    var nvar : integer;
 ⌈var int nvar := n;                      | begin nvar := n;
   while nvar > 1                         |    while nvar > 1
   do nvar := if odd nvar                 |    do nvar := if odd(nvar)
              then 3 × nvar + 1           |               then 3*nvar + 1
              else nvar ÷ 2   fi od;      |               else nvar div 2;
   1                                  ⌋   |    mc ⇐ 1
                                          | end
```

3.3.5.1. Операторы альтернативы

Введем теперь ***оператор альтернативы***, оператор, выбирающий между двумя операторами; выбор производится в соответствии с результатом проверки некоторого условия. В алголе-68 оператор альтернативы записывается так:

if ›условие‹ **then** ›да-оператор‹ **else** ›нет оператор‹ **fi**,

в паскале —

if ›условие‹ **then** ›да-оператор‹ **else** ›нет-оператор.

Используя оператор альтернативы, получим следующую подпрограмму:

```
funct mc ≡ (int n co n > 0 co) int:       | function mc(n : integer {n > 0}) : integer;
                                          |    var nvar : integer;
 ⌈var int nvar := n;                      | begin nvar := n;
   while nvar > 1                         |    while nvar > 1
   do if odd nvar                         |    do if odd(nvar)
      then nvar := 3 × nvar + 1           |       then nvar := 3*nvar + 1
      else nvar := nvar ÷ 2   fi od;      |       else nvar := nvar div 2;
   1                                  ⌋   |    mc ⇐ 1
                                          | end
```

Для алгоритма Эвклида (пример (b) из 2.3.2) имеется следующая эквивалентная формулировка:

```
funct gcd ≡ (int m, int n                 | function gcd(m, n : integer
             co m > 0 ∧ n > 0 co) int:    |              {(m > 0) ∧ (n > 0)}) : integer;
                                          | begin
   if m ≠ n                               |    if m ≠ n
   then if m < n then gcd(n, m)           |    then if m < n then gcd ⇐ gcd(n, m)
                 else gcd(m − n, n) fi    |                  else gcd ⇐ gcd(m − n, n)
   else m                            fi   |    else gcd ⇐ m
                                          | end
```

Её также можно заменить оператором цикла, в повторяемом операторе которого переменным (*mvar*, *nvar*) альтернативно

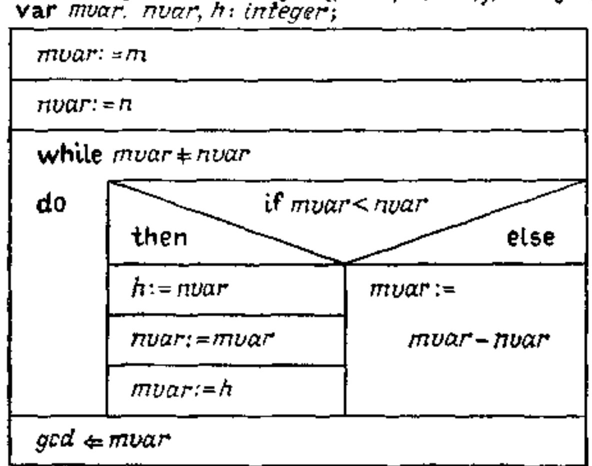


Рис. 105. Диаграмма Насси — Шнейдермана для *gcd*.

присваивается либо (*nvar*, *mvar*), либо (*mvar* — *nvar*, *nvar*):

```
funct gcd ≡ (int m, n co m > 0 ∧ n > 0 co) int:
 ⌈(var int mvar, nvar) := (m, n);
  while mvar ≠ nvar
    do (mvar, nvar) := if mvar < nvar then (nvar, mvar)
                                     else (mvar − nvar, nvar) fi od;
  mvar                                                         ⌋
```

Перейдем здесь к оператору альтернативы, ветви которого являются групповыми присваиваниями:

```
do if mvar < nvar then (mvar, nvar) := (nvar, mvar)
                  else (mvar, nvar) := (mvar − nvar, nvar) fi od
```

При линеаризации первой ветви потребуется некоторая вспомогательная величина, а во второй ветви переменная *nvar* остаётся

```
funct rom ≡ (int n                  | function rom(n : integer
   co 0≤n<10000 co) string:          |          {0≤n<10000}) : string :
                                     | begin
if n≥1000                            |    if n≥1000
then ("M") + rom(n − 1000)           |    then rom ⇐ prefix('M', rom(n − 1000))
elsf n≥ 500                          |    else if n≥ 500
then ("D") + rom(n −  500)           |    then rom ⇐ prefix('D', rom(n −  500))
elsf n≥ 100                          |    else if n≥ 100
then ("C") + rom(n −  100)           |    then rom ⇐ prefix('C', rom(n −  100))
elsf n≥  50                          |    else if n≥  50
then ("L") + rom(n −   50)           |    then rom ⇐ prefix('L', rom(n −   50))
elsf n≥  10                          |    else if n≥  10
then ("X") + rom(n −   10)           |    then rom ⇐ prefix('X', rom(n −   10))
elsf n≥   5                          |    else if n≥   5
then ("V") + rom(n −    5)           |    then rom ⇐ prefix('V', rom(n −    5))
elsf n≥   1                          |    else if n≥   1
then ("I") + rom(n −    1)           |    then rom ⇐ prefix('I', rom(n −    1))
else ◇                          fi   |    else rom ⇐ empty
                                     | end
```

Рис. 106.

без изменения, так что мы получаем

```
funct gcd ≡ (int m, n co m>0 ∧ n>0 co) int:
⌈var int mvar := m; var int nvar := n;
 while mvar ≠ nvar
 do if mvar < nvar then int h ≡ nvar;
                        nvar := mvar;
                        mvar := h
                   else mvar := mvar − nvar      fi od;
mvar                                               ⌋
```

На рис. 105 показана соответствующая диаграмма Насси — Шнейдермана с символом альтернативы.

3.3.5.2. Последовательные условные операторы

Прежде всего примем, что *оператор альтернативы сам является оператором.*

В частности, операторы альтернативы могут быть вложенными, а для последовательных условных операторов можно, как в 2.2.3, ввести сокращённую запись, например писать

if $x > 0$ **then** $x := 1$ **elsf** $x \geqslant 0$ **then** $x := 0$ **else** $x := -1$ **fi**

```
funct rom ≡ (int n
   co 0 ≦ n < 10000 co) string:
   ⌈funct ro ≡ (int n, string z)
                          string:

   if n ≧ 1000
   then ro(n − 1000, z + ("M"))
   elsf n ≧  500
   then ro(n −  500, z + ("D"))
   elsf n ≧  100
   then ro(n −  100, z + ("C"))
   elsf n ≧   50
   then ro(n −   50, z + ("L"))
   elsf n ≧   10
   then ro(n −   10, z + ("X"))
   elsf n ≧    5
   then ro(n −    5, z + ("V"))
   elsf n ≧    1
   then ro(n −    1, z + ("I"))
   else z                      fi;

   ro(n, ◇)                     ⌋
```

```
function rom(n : integer
             [0 ≦ n < 10000]) : string;
function ro(n : integer; z : string) : string;

begin
if n ≧ 1000
then ro ⇐ ro(n − 1000, postfix(z, 'M'))
else if n ≧  500
then ro ⇐ ro(n −  500, postfix(z, 'D'))
else if n ≧  100
then ro ⇐ ro(n −  100, postfix(z, 'C'))
else if n ≧   50
then ro ⇐ ro(n −   50, postfix(z, 'L'))
else if n ≧   10
then ro ⇐ ro(n −   10, postfix(z, 'X'))
else if n ≧    5
then ro ⇐ ro(n −    5, postfix(z, 'V'))
else if n ≧    1
then ro ⇐ ro(n −    1, postfix(z, 'I'))
else ro ⇐ z
end;
begin
  rom ⇐ ro(n, empty)
end
```

Рис. 107.

```
funct rom ≡ (int n co 0 ≦ n < 10000 co) string:

 ⌈(var int nvar, var string zvar) := (n, ◇);
  while nvar ≧ 1
  do
  if   nvar ≧ 1000
  then (nvar, zvar) := (nvar − 1000, zvar + ("M"))
  elsf nvar ≧  500
  then (nvar, zvar) := (nvar −  500, zvar + ("D"))
  elsf nvar ≧  100
  then (nvar, zvar) := (nvar −  100, zvar + ("C"))
  elsf nvar ≦   50
  then (nvar, zvar) := (nvar −   50, zvar + ("L"))
  elsf nvar ≧   10
  then (nvar, zvar) := (nvar −   10, zvar + ("X"))
  elsf nvar ≧    5
  then (nvar, zvar) := (nvar −    5, zvar + ("V"))
  else (nvar, zvar) := (nvar −    1, zvar + ("I"))
                                            fi od;
  zvar                                          ⌋
```

```
function rom(n : integer [0 ≦ n < 10000]) : string;
   var nvar : integer; zvar : string;
begin (nvar, zvar) := (n, empty);
   while nvar ≧ 1
   do
   if   nvar ≧ 1000
   then (nvar, zvar) := (nvar − 1000, postfix(zvar, 'M'))
   else if nvar ≧  500
   then (nvar, zvar) := (nvar −  500, postfix(zvar, 'D'))
   else if nvar ≧  100
   then (nvar, zvar) := (nvar −  100, postfix(zvar, 'C'))
   else if nvar ≧   50
   then (nvar, zvar) := (nvar −   50, postfix(zvar, 'L'))
   else if nvar ≧   10
   then (nvar, zvar) := (nvar −   10, postfix(zvar, 'X'))
   else if nvar ≧    5
   then (nvar, zvar) := (nvar −    5, postfix(zvar, 'V'))
   else (nvar, zvar) := (nvar −    1, postfix(zvar, 'I'));

   rom ⇐ zvar
end
```

Рис. 108.

```
funct decod ≡ (lisp s, bits a
        co a ist Element aus dem durch
           s definierten Code co) char:

 ⌈(var lisp svar, var bits avar):=(s, a);
  while ¬isatom(svar) ∧ avar ≠ ◊

    do if first(avar) = O
        then (svar, avar):=
              (car(svar), rest(avar))
        ▯ first(avar) = L
        then (svar, avar):=
              (cdr(svar), rest(avar)) fi od;
  co isatom(svar) ∧ avar = ◊ co
  val(svar)                                ⌋
```

```
function decod(s : lisp; a : bitstring
              {a ist Element aus dem durch
               s definierten Code}) : char;
  var svar : lisp;
      avar : bitstring;
begin
  (svar, avar) := (s, a);
  while not isatom(svar) and not
                              isempty(avar)
    do if first(avar) = O
        then (svar, avar) :=
                    (car(svar), rest(avar))
        ▯ first(avar) = L
        then (svar, avar) :=
                    (cdr(svar), rest(avar));
   {isatom(svar) and isempty(avar)}
  decod ← val(svar)
end
```

Рис. 109.

вместо

$$\textbf{if } x > 0 \textbf{ then } x := 1 \textbf{ else if } x \geqslant 0 \textbf{ then } x := 0 \textbf{ else } x := -1 \textbf{ fi fi}$$

(запись присваивания $x := \mathbf{sign}\, x$).

В качестве примера рассмотрим представленную на рис. 106 подпрограмму перевода натурального числа в „римскую запись“ [1]. Используя вложение, приходим к очевидному повторительному варианту, приведённому на рис. 107. В результате перехода к итеративной форме (рис. 108) получаем всего один оператор цикла, содержащий в своём теле соответствующий последовательный разбор случаев.

3.3.5.3. Охраняемые операторы

Представляется целесообразным по аналогии с охраняемым разбором случаев ввести также ***охраняемые операторы*** как наиболее общую форму условных операторов. Охраняемые операторы были предложены Дейкстрой в 1975 г., поэтому в алголе-68 и паскале их ещё нет. Способ их записи ясен из аналогии с 2.3.3.2. Примером употребления охраняемых операторов может служить итеративная формулировка подпрограммы *decod* (см. 2.4.1.6), представленная на рис. 109. Линеаризация здесь тривиальна, при этом присваивание $avar := rest(avar)$ можно

---

[1] Эта подпрограмма реализует первоначальную римскую систему счисления. Такие сокращённые способы записи, как, например, IV вместо IIII, вошли в употребление лишь в начале 16-го века.

вывести из обеих ветвей:

$$\begin{aligned}&\textbf{do if } \mathit{first}(avar) = \mathsf{O} \textbf{ then } svar := car(svar)\\ &\quad \square\ \mathit{first}(avar) = \mathsf{L} \textbf{ then } svar := cdr(svar) \textbf{ fi};\\ &\quad avar := rest(avar) \qquad\qquad\qquad \textbf{od}.\end{aligned}$$

Как и охраняемые формулы, охраняемые операторы являются, в принципе, недетерминированными: если несколько стражей дают свободный путь, то можно выполнять любой из разрешённых тем самым операторов. Выбор будет, конечно, детерминирован, если все условия-стражи взаимно исключают друг друга. Если быть последовательным, то в случае, когда все стражи блокируют пути, следовало бы считать, что продолжение работы невозможно. Однако Дейкстра в этом случае принимает, что получающаяся ситуация равносильна пустому оператору. Мы не будем следовать этому определению. Напротив, мы хотим сохранить в алгольной нотации отличие конструкции

$$\textbf{if } x > 0 \textbf{ then } x := 1 \ \square\ x < 0 \textbf{ then } x := -1 \ \square\ x = 0 \textbf{ then } x := 0 \textbf{ fi}$$

от оператора

$$\textbf{if } x > 0 \textbf{ then } x := 1 \ \square\ x < 0 \textbf{ then } x := -1 \textbf{ fi},$$

который лишь частично определён и для которог оследовало бы поэтому ввести условие-предохранитель **co** $x \neq 0$ **co**.

3.3.6. Пустой оператор

Иногда оказывается, что в одной из ветвей условного оператора нечего делать, например когда $x := \textbf{abs}\, x$ записывают в виде

$$\textbf{if } x \geqslant 0 \textbf{ then } x := x \textbf{ else } x := -x \textbf{ fi}$$

или, как выше, $x := \textbf{sign}\, x$ — в виде

$$\textbf{if } x > 0 \textbf{ then } x := 1 \textbf{ elsf } x \geqslant 0 \textbf{ then } x := 0 \textbf{ else } x := -1 \textbf{ fi},$$

где присваивание $x := 0$ избыточно. Для такого пустого оператора хорошо бы иметь свой собственный символ, скажем **skip**[1] (алгол-68):

$$\textbf{if } x \geqslant 0 \textbf{ then skip else } x := -x \textbf{ fi},$$
$$\textbf{if } x > 0 \textbf{ then } x := 1 \textbf{ elsf } x \geqslant 0 \textbf{ then skip else } x := -1 \textbf{ fi}.$$

---

[1] Буквально: прыжок, скачок, пропуск. — *Прим. перев.*

```
funct rom ≡ (int n co 0≤n<10000 co) string :


 ⌈(var int nvar, var string zvar) := (n, ◊);
   while nvar ≥ 1000
   do (nvar, zvar)
         := (nvar − 1000, zvar + ("M")) od;
   if nvar ≥ 500
   then (nvar, zvar)
         := (nvar − 500, zvar + ("D")) else skip fi;
   while nvar ≥ 100
   do (nvar, zvar)
         := (nvar − 100, zvar + ("C")) od;
   if nvar ≥ 50
   then (nvar, zvar)
         := (nvar − 50, zvar + ("L")) else skip fi;
   while nvar ≥ 10
   do (nvar, zvar)
         := (nvar − 10, zvar + ("X")) od;
   if nvar ≥ 5
   then (nvar, zvar)
         := (nvar − 5, zvar + ("V")) else skip fi;
   while nvar ≥ 1
   do (nvar, zvar)
         := (nvar − 1, zvar + ("I")) od;
   zvar                                            ⌋
```

```
function rom(n : integer {0≤n<10000}) : string
  var nvar : integer;
      zvar : string;
begin (nvar, zvar) := (n, empty);
  while nvar ≥ 1000
  do (nvar, zvar)
        := (nvar − 1000, postfix(zvar, 'M'));
  if nvar ≥ 500
  then (nvar, zvar)
        := (nvar − 500, postfix(zvar, 'D'));
  while nvar ≥ 100
  do (nvar, zvar)
        := (nvar − 100, postfix(zvar, 'C'));
  if nvar ≥ 50
  then (nvar, zvar)
        := (nvar − 50, postfix(zvar, 'L'));
  while nvar ≥ 10
  do (nvar, zvar)
        := (nvar − 10, postfix(zvar, 'X'));
  if nvar ≥ 5
  then (nvar, zvar)
        := (nvar − 5, postfix(zvar, 'V'));
  while nvar ≥ 1
  do (nvar, zvar)
        := (nvar − 1, postfix(zvar, 'I'));
  rom ⇐ zvar
end
```

Рис. 110.

Переставляя ветви, получаем также

if $x < 0$ **then** $x := -x$ **else skip fi**,
if $x > 0$ **then** $x := 1$ **elsf** $x < 0$ **then** $x := -1$ **else skip fi**.

В стандартном алголе-68 **else skip fi** р а з р е ш а е т с я сокращать до **fi**. Однако для большей ясности, а также имея в виду ситуацию с охраняемыми операторами, мы не будем использовать эту возможность.

В паскале пустой оператор „записывается" пустой цепочкой знаков: вырожденному („однорукому") разбору случаев

if B **then** S **else skip fi**

соответствует в паскале оператор

if B **then** S **else**; ,

который также разрешается сокращать до

if B **then** S; .

Этого тоже не стоит делать. Именно за счёт такой возможности сокращения в паскале возникает известная ещё из алгола-60

многозначность конструкций типа

$$\textbf{if } x \neq 0 \textbf{ then if } x > 0 \textbf{ then } x := 1 \textbf{ else } x := -1.$$

В паскале эта многозначность искусственно разрешается принятием соглашения, что такого рода конструкцию следует понимать как сокращение для

$$\textbf{if } x \neq 0 \textbf{ then begin if } x > 0 \textbf{ then } x := 1 \textbf{ else } x := -1 \textbf{ end}.$$

Повторение последовательного разбора случаев, получившееся у нас в примере *rom* из 3.3.5.2, можно заменить более эффективным последовательным разбором случаев, каждый из которых представляет собой повторение (оператор цикла). При этом те из появляющихся операторов цикла, которые повторяют своё тело не более одного раза, можно преобразовать в вырожденный разбор случаев. См. рис. 110.

3.4. Операторы перехода

3.4.1. Гладкие вызовы и операторы перехода

В разделах 3.3.2.1 и 3.3.5 было показано, как из прямых повторительных подпрограмм получить итеративные формулировки, записанные с помощью операторов цикла. Теперь мы рассмотрим более общий метод, применяемый, в частности, к системам повторительных подпрограмм. Прежде всего переформулируем систему (*ispos*, *isneg*) из 2.4.1.4. Если „снаружи" важна лишь подпрограмма *ispos*, то мы можем подчинить ей подпрограмму *isneg* (см. 2.5.1) или, лучше (для сохранения симметрии), пару подпрограмм (*isp*, *isn*) — см. рис. 111.

Если проследить работу машины обработки формуляров для этой повторительной системы после начального вызова *isp*(*m*), то можно обнаружить, что в ходе вычислений мы скачем туда-сюда между подпрограммами *isp* и *isn*. В соответствии с нашим прежним образом действий введём переменную *pvar* вместо параметра *p* подпрограммы *isp* переменную *nvar* вместо параметра *n* подпрограммы *isn*. Тогда *гладкий* вызов (см. 2.3.3) подпрограммы *isp* (соотв. *isn*) можно заменить присваиванием новых значений параметров переменной *pvar* (соотв. *nvar*) с последующим продолжением исполнения тела вызванной подпрограммы (впрочем, переменные *pvar* и *nvar* здесь можно было бы и отождествить).

Чтобы выразить это, нам понадобится новый языковый элемент, называемый ***оператором перехода***[1, 2]:

goto ›метка‹,

[1] В оригинале Sprung (прыжок, скачок). — *Прим. перев.*

[2] Ниже **goto** — слитно написанное **go to** (иди к). — *Прим. перев.*

```
funct ispos ≡ (string m) bool :          function ispos(m : string) : Boolean ;
 ⌈funct isp ≡ (string p) bool :            function isp(p : string) : Boolean ;
                                           begin
   if p ≠ ◇                                  if not isempty(p)

   then if first(p) = "−"                    then if first(p) = '−'
        then isn(rest(p))                         then isp ⇐ isn(rest(p))
        ▯ first(p) = "+"                          ▯ first(p) = '+'
        then isp(rest(p)) fi                      then isp ⇐ isp(rest(p))
   else true                 fi ,            else isp ⇐ true
                                           end ;
  funct isn ≡ (string n) bool :            function isn(n : string) : Boolean ;
                                           begin
   if n ≠ ◇                                  if not isempty(n)
   then if first(n) = "−"                    then if first(n) = '−'
        then isp(rest(n))                         then isn ⇐ isp(rest(n))
        ▯ first(n) = "+"                          ▯ first(n) = '+'
        then isn(rest(n)) fi                      then isn ⇐ isn(rest(n))
   else false                fi ;            else isn ⇐ false
                                           end;
  isp(m)                     ⌋           begin ispos ⇐ isp(m)
                                         end
```

Рис. 111.

где ›метка‹ представляет собой обозначение пометки, помечающей начало того оператора, которым должно продолжаться исполнение. Пометками служат:

в алголе-68 — идентификаторы (см. 2.2.1),

в паскале — изображения целых без знака,

вслед за которыми ставится двоеточие:

```
goto mp                                  goto 1
  mp : if pvar ≠ ◇ then ⋯                  1 : if not isempty(pvar) then ⋯
```

Помеченный оператор сам является оператором[1]. *Оператор перехода — это также оператор.*

Итак, используя переменные *pvar* и *nvar* вместо соответствующих значений параметров подпрограмм *isp* и *isn*, а также переменную *result*, в которой „спасается" результат после завершения рекурсии в *isp* или *isn*, мы получаем для нашей системы подпрограмм алгольную версию, представленную в левом столбце на рис. 112[2]. Для паскалевской версии (правый

[1] В паскале запрещается многократно помечать один и тот же оператор; правда, в этом и нет необходимости. В алголе-68 можно помечать и формулы.

[2] Оператор перехода отличается от вызова подпрограммы (и от гладкого вызова) тем, что машина, выполняющая оператор перехода, может

```
funct ispos ≡ (string m) bool:

 ⌈var string pvar, nvar, var bool result;
        pvar := m;
        goto mp;
   mp: if pvar ≠ ◇
        then if first(pvar) = "−"
                then nvar := rest(pvar);
                        goto mn

                ▯ first(pvar) = "+"
                then pvar := rest(pvar);
                        goto mp        fi


      else result := true;
            goto end                  fi;


  mn: if nvar ≠ ◇
      then if first(nvar) = "−"
              then pvar := rest(nvar);
                    goto mp

              ▯ first(nvar) = "+"
              then nvar := rest(nvar);
                    goto mn           fi


      else result := false;
            goto end                  fi;

  end: skip; result                   ⌋
```

```
function ispos(m: string): Boolean;
  label 1, 2, 3;
  var pvar, nvar: string;
begin pvar := m;
  goto 1;
  1 if not isempty(pvar)
    then begin if first(pvar) = '−'
                  then begin nvar := rest(pvar);
                                goto 2
                          end
                  ▯ first(pvar) = '+'
                  then begin pvar := rest(pvar);
                                goto 1
                          end
              end
    else begin ispos ⇐ true;
                goto 3
          end;

  2: if not isempty(nvar)
    then begin if first(nvar) = '−'
                  then begin pvar := rest(nvar);
                                goto 1
                          end
                  ▯ first(nvar) = '+'
                  then begin nvar := rest(nvar);
                                goto 2
                          end
              end
    else begin ispos ⇐ false;
                goto 3
          end;
  3:
end
```

Рис. 112.

столбец) особая логическая переменная для хранения вырабатываемого результата не нужна, поскольку установка результата *ispos* ⇐ *true* или *ispos* ⇐ *false* здесь может осуществляться непосредственно [1]. Конечно, здесь можно ещё кое на чём сэкономить. Например, переход на оператор, непосредственно следующий за оператором перехода, является излишним. Из программных участков

```
~~ goto mp;                    | ~~ goto 1;
     mp: if pvar ≠ ◇           |      1: if not isempty(pvar)
           then ~~             |            then ~~
```

„позабыть", откуда произошёл переход. Поэтому перед каждым из операторов **goto** *end* нужно обеспечить получение (вывод) соответствующего результата.

[1] В паскале обозначение *ispos* ради простоты трактуется так, будто это программная переменная для результата подпрограммы *ispos*.

и

```
~~ goto end fi;          | ~~ goto 3 end;
   end: result ⌋         |    3: end
```

операторы перехода можно удалить, поскольку для каждого из них точкой продолжения является следующий исполняемый оператор.

В паскале метки должны быть описаны в начале соответствующего блока перед описанием локальных переменных; тем самым этот блок является их областью связывания и определяет также соответствующую область действия меток. Поэтому переход „извне" внутрь оператора оказывается невозможным.

В алголе-68 вхождение обозначения *m* в *m*: считается описанием метки *m*, для которого справедливы обычные правила связывания обозначений. Благодаря этому здесь также запрещён переход „извне" внутрь предложения или (составного) оператора.

Например,

```
~~ goto m; ~~ ⌈real x = ~~; m: x × ln(x) ⌋ ~~
~~ goto m; ~~ ⌈var real x := ~~; m: x := x × x; x × ln(x) ⌋ ~~
~~ goto m; ~~ ⌈var real x := ~~; x := m: x × x; x × ln(x) ⌋ ~~
```

— это бессмысленные и уже хотя бы поэтому недопустимые конструкции. Ради единообразия в алголе-68 запрещается переход внутрь любого предложения, в том числе и предложения без описаний. В этом правиле содержится, разумеется, и запрет на переход извне в тело подпрограммы.

Заметим, кстати, что и пары **if**. **then**, **then**. **else**, **else**. **fi** и т. д. (см. последнее подстрочное примечание в разделе 2.5.1), а также пара **do**. **od**[1] действуют как скобки предложения (или как операторские скобки). Это значит, что не разрешаются и конструкции вроде

```
if x > 0 then x := x + 1; m: y := y + 1
         else x := x + 2; goto m        fi.
```

Вместо этого можно написать и проще, и яснее:

```
if x > 0 then x := x + 1
         else x := x + 2 fi;
y := y + 1.
```

В противоположность этому переход „изнутри" в объемлющее предложение всегда разрешается.

---

[1] См. первое подстрочное примечание в разделе 3.3.2.

Переходы могут, как в приведенном выше примере, в точности воспроизводить структуру алгоритма. Однако ясности программы они не способствуют, а бездумное их употребление часто выдаёт плохое понимание структуры алгоритма. Умышленное употребление переходов для связывания программных участков друг с другом — это плохой, сомнительный стиль, чреватый ошибками.

3.4.2. Реализация повторения при помощи переходов

Поскольку прямые повторительные подпрограммы являются частным случаем повторительных систем, итеративную формулировку повторительной подпрограммы можно также получить, используя вместо операторов цикла операторы перехода.

3.4.2.1

Так, для подпрограммы *potz* из 3.3.3.1 мы получим формулировку, представленную на рис. 113. На операторах **goto** *end*

```
funct potz ≡ (real a, int n                |  function potz (a : real; n : integer
                    co n ≧ 0 co) real :     |                        {n ≧ 0}) : real;
                                            |    label 1, 2;
                                            |    var avar : real; nvar : integer;
⌈(var real avar, var int nvar) := (a,n);    |  begin (avar, nvar) := (a,n);
 rep : if nvar > 0                          |    1 : if nvar > 0
      then (avar, nvar) :=                  |       then begin (avar, nvar) :=
                  (avar↑2, nvar−1);         |                 (sqr(avar), nvar − 1);
           goto rep                         |                 goto 1
                                            |            end
      else goto end               fi;       |       else goto 2;
 end : skip ; avar                        ⌋ |    2: potz ⇐ avar
                                            |  end
```

Рис. 113.

(соотв. **goto** *2*) снова можно сэкономить. Переменной *result* для фиксации результата здесь не понадобилось.

3.4.2.2

Вообще, каждый оператор цикла можно записать с помощью операторов перехода: оператор

```
while ›условие‹          |   while ›условие‹
  do оператор od         |   do ›оператор‹
```

эквивалентен ***петле***

```
rep: if ›условие‹
       then ›оператор‹;
            goto rep

     else skip    fi
```

```
1:
if ›условие‹
then begin ›оператор‹;
           goto 1
     end
else skip;
```

3.4.2.3

Подпрограмма *ispr 1* из 2.5.2.1 не допускает такого упрощения, какое мы провели в разделе 3.3.2.1 для подпрограммы *ispr*. Однако здесь и нет нужды в таком преобразовании, поскольку за счёт непосредственного использования оператора перехода получается подпрограмма, показанная на рис. 114 (гд n — неизменный параметр подпрограммы *ispr 1*).

```
funct isprim ≡ (int n
            co n ≧ 1 co) bool :


  if n = 1
  then false
  else var int mvar := 2 ,
       var bool result ;
       rep : if mvar↑2 > n
             then result := true
             elsf n mod mvar = 0
             then result := false

             else mvar := mvar + 1;
                  goto rep
                                    fi ;
       result
                                    fi
```

```
function isprim (n : integer
                 {n ≧ 1}) : Boolean ;
  label 1; var mvar : integer ;
begin
  if n = 1
  then isprim ⇐ false
  else begin mvar := 2 ;

    1 : if sqr(mvar) > n
        then isprim ⇐ true
        else if n mod mvar = 0
          then isprim ⇐ false
          else begin
                 mvar := mvar + 1;
                 goto 1
               end
        end
end
```

Рис. 114.

С использованием оператора цикла получим совсем другую формулировку, представленную на рис. 115, в которой, несмотря на компактность записи, условие $mvar \uparrow 2 \leqslant n$ (соотв. $sqr(mvar) \leqslant n$) приходится выписывать дважды.

Таким образом, у нотации с переходами могут быть и преимущества, когда в рекурсивной (повторительной) формулировке имеется несколько ветвей окончания.

<table>
<tr>
<td>

```
funct isprim ≡ (int n
             co n≧1 co) bool :



  if n = 1
  then false
  else var int mvar := 2 ;
       while mvar↑2 ≦ n ∧
               n mod mvar ≠ 0
       do mvar := mvar + 1 od ;
       ¬ (mvar↑2 ≦ n)
                              fi
```

</td>
<td>

```
function isprim (n : integer
               {n ≧ 1}) : Boolean ;
  var mvar : integer ;
begin
  if n = 1
  then isprim := false
  else begin mvar := 2 ;
       while (sqr(mvar) ≦ n) and
              (n mod mvar ≠ 0)
       do mvar := mvar + 1 ;
       isprim := not (sqr(mvar) ≦ n)
       end
end
```

</td>
</tr>
</table>

Рис. 115.

3.4.3. Блок-схемы программ

На уровне машины обработки формуляров введение переходов означает обобщение, состоящее в появлении дополнительной специальной обработки в случае наиболее простого расположения гладкого вызова без „задержанных" операций. Для машины обработки формуляров операторы перехода и повторения (оператор цикла можно воспринимать как петлю, замкнутую переходом) остаются всего лишь способами сокращённой записи рекурсии, и тем не менее мы имеем здесь обобщение семантики, отражаемое и самим выбором слова „скачок" [1]: порождаемый гладким вызовом переход к началу другого или (например, в случае повторения) того же самого формуляра происходит без обязательства возврата, т. е. без заведения нового воплощения, и без какого-либо ущерба для уже имеющихся к этому моменту обязательств возврата.

Если ограничиться, как в предшествующих примерах, повторительными системами, то после введения переходов вообще не будет нужды в заведении воплощений; за счёт использования переменных исходные вычислительные формуляры системы превращаются в ***программные формуляры*** общего вида, содержащие стопки граф.

3.4.3.1

Выполнение вычисления превращается тем самым в ***движение*** внутри совокупного набора программных формуляров, соот-

---

[1] См. первое подстрочное примечание в разделе 3.4.1.— *Прим. перев.*

ветствующих некоторой системе подпрограмм; в предельном случае — в движение по программному формуляру, соответствующему некоторой непосредственно рекурсивной подпрограмме.

Если графически изобразить процесс движения (называемый также ***исполнением***), то из совокупности отдельных программных формуляров возникнет „схема движения" — ***блок-схема***

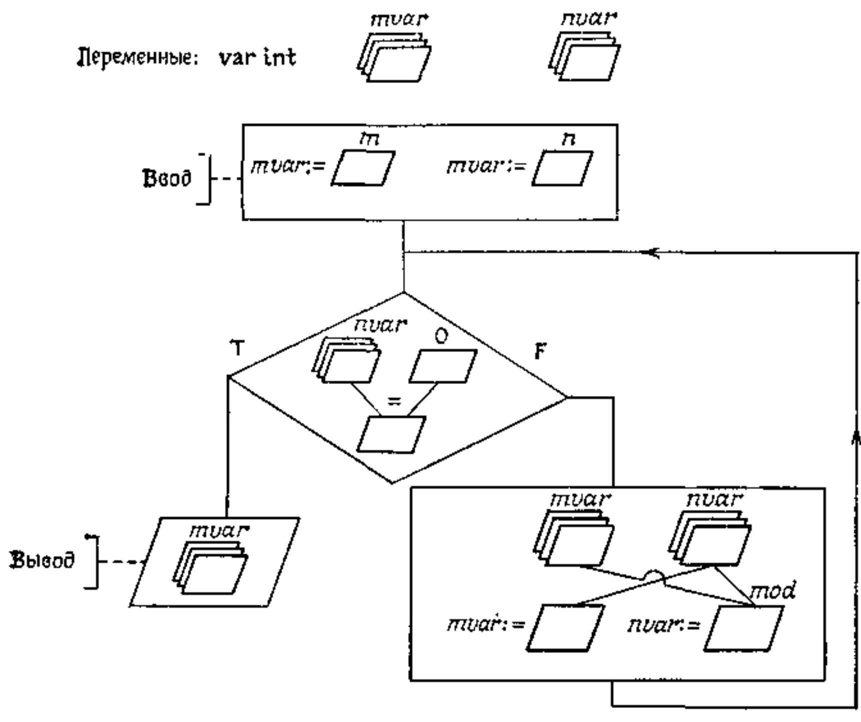


Рис. 116. Блок-схема, полученная из программного формуляра на рис. 95.

программы. На рис. 116 дан пример такой блок-схемы, полученной из программного формуляра на рис. 95, соответствующего подпрограмме *gcd1*, „обрамлением" (заключением в рамочки) исполняемых действий и проведением ***дуг***, или ***линий***, (***передачи***) ***управления***. Использованные здесь графические символы взяты из стандарта DIN 66001 (рис. 117).

Более компактной является другая форма блок-схем, получающаяся возведением графической „надстройки" над программным текстом. Так, в соответствии со сказанным в 3.4.2 подпро-

Символическое изображение	Объяснение
	Операция общего вида, в особенности если она не подпадает ни под один из следующих четырёх типов.
	Разветвление. Такие точки в блок-схеме, в которых на основе проверки некоторого условия нужно выбирать одно из нескольких возможных продолжений.
	Вызов подпрограммы; обычно эта подпрограмма описывается в другой блок-схеме.
	Модификация программы, т. е., например, места́ расположения запрограммированных переключателей, изменение индексных регистров или модификация самого́ программного текста (не рекомендуется!).
	Ручная операция, связанная, вообще говоря, с ожиданием.
	Ввод, вывод.
	Линия (дуга) передачи управления, предпочтительные направления — сверху вниз и слева направо; возможны — а при отклонении от предпочтительных направлений даже необходимы — стрелки, направленные к следующему символическому изображению.
	Схождение, одинаковое продолжение после различных точек в блок-схеме.
	Точка перехода, пара одинаковых обозначений, используемая для „связывания" двух концов разорванной на блок-схеме линии.
	Пограничные точки, т. е. начало и конец исполнения, а также программные прерывания.
	Синхронизация при параллельном выполнении, особенно в следующих трёх случаях:
	Расщепление в режиме параллельного выполнения.
	„Сращивание" ветвей в режиме параллельного выполнения.
	Синхронизационное сечение.
	Примечание, может быть присоединено к любому другому символу.

Рис. 117. Символы для блок-схем (стандарт DIN 66001).

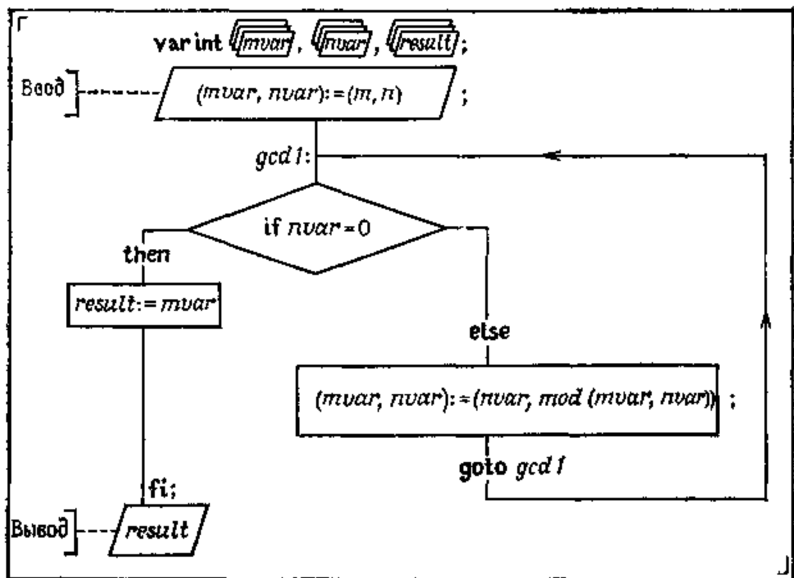


Рис. 118. Блок-схема, полученная возведением графической „надстройки" над программным текстом.

грамма *gcd1* (см. 3.3.2.1) допускает формулировку

```
funct gcd1 ≡ (int m, n co m≧0 ∧ n≧0 co) int:
⌈          var int mvar, nvar, result;
    (mvar, nvar):=(m, n);
    gcd1: if nvar = 0
          then result:= mvar
          else (mvar, nvar):= (nvar, mod(mvar, nvar));
               goto gcd1                           fi;
          result
```

которая даёт блок-схему, изображенную на рис. 118.

3.4.3.2

Повторительной системе подпрограмм отвечает набор из соответствующего числа формуляров, который становится блок-схемой благодаря связующим переходам. Рис. 119 демонстрирует это для рассмотренного в 2.4.1.4 и 3.4.1 примера системы подпрограмм (*ispos*, *isneg*).

Чтобы не превращать блок-схему в „лабиринт", управляющие дуги можно разрывать, отмечая соответствующие „вися-

щие" концы с помощью ***точек перехода*** (см. рис. 117). На уровне языка программирования этим точкам соответствуют оператор перехода и помеченное место, куда осуществляется переход. Переходы сами по себе — вещь вполне естественная, если они обеспечивают лучшее структурирование и происходят, скажем, от гладких вызовов в некоторой повторительной системе.

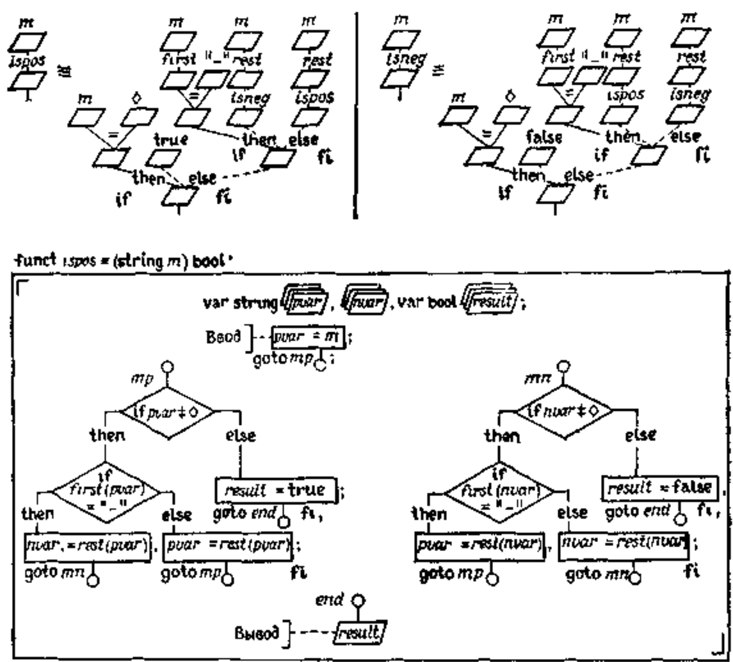


Рис. 119 Вычислительные формуляры для системы подпрограммы (*ispos*, *isneg*) из 2.4.1.4 и блок-схема, построенная в соответствии со сказанным в 3.4.1.

Иначе обстоит дело с „дикими" переходами, которые вводят по причине недостаточного понимания структуры алгоритма. Дейкстра с полным основанием поднял свой голос, предостерегая против употребления такого не наглядного и плохо прослеживаемого „дальнодействия"[1]. Некоторые чересчур рьяные сто-

[1] Слегка утрируя, Дейкстра пишет: "...the quality of programmers is a decreasing function of the density of **go to**'s in the programs they produce... ...the **go to** statement should be abolished from all 'higher level' programming languages (i. e. everything except, perhaps, plain machine code)" [„...квалификация программиста является убывающей функцией от плотности операторов **go to** в создаваемых им программах... ...оператор **go to** следовало бы упразднить во всех языках программирования „высокого уровня" (т. е. всюду, кроме, разве, *простого машинного кода*) (англ.). — *Перев.*

ронники не только вывели отсюда, что в „хорошо структурированных программах“ лучше всего вообще не использовать переходов, но и что всякая программа, записанная в форме диаграммы Насси — Шнейдермана, будет „хорошо структурированной“.

Без переходов можно обойтись в том случае, когда отдельные повторительные подпрограммы данной системы (которая

funct *isposneg* ≡ (**string** *m*, **bool** *v*) **bool**:

Рис. 120. Блок-схема для *isposneg*.

сама по себе не обязана быть повторительной) могут быть вложены друг в друга как повторения или выстроены последовательно друг за другом, см. 3.3.2.3. Часто, однако, требование избегать переходов приводит к дублированию программного текста. Иногда (но далеко не всегда) это может даже оказаться полезным для понимания, см. в связи с этим подпрограмму *isprim* из 3.4.2.3.

Для системы подпрограмм (*ispos*, *isneg*) сведéние к одному-единственному оператору цикла требует с л и я н и я обеих подпрограмм в одну с добавлением нового логического параметра. Такую подпрограмму *isposneg*, определяемую следующим образом:

funct *isposneg* ≡ (**string** *m*, **bool** *v*) **bool**:
if *v* **then** *ispos*(*m*) **else** *isneg*(*m*) **fi**,

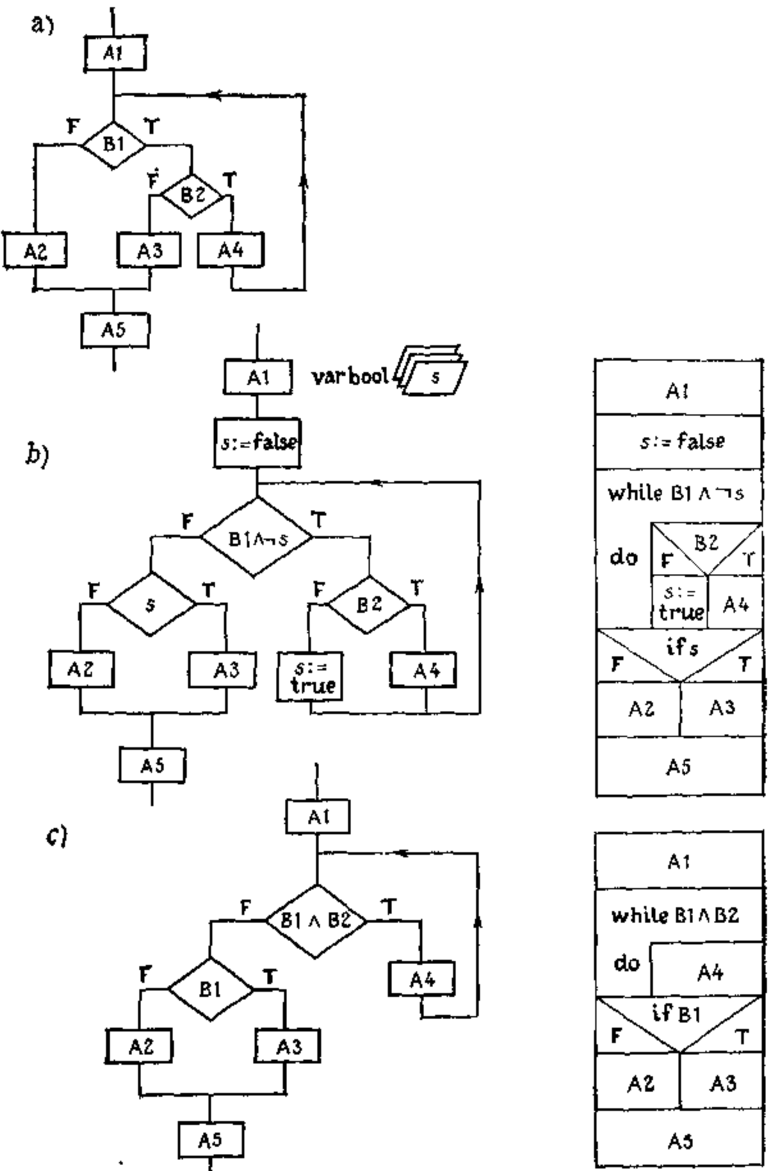


Рис. 121. Блок-схемы с переходами и дистанционными переключателями.

можно записать как повторительную:

```
funct isposneg ≡ (string m, bool v) bool:
  if m ≠ ◇
  then if first(m) = "−"
         then isposneg(rest(m), ¬v)
         else isposneg(rest(m), v) fi
  else v                              fi .
```

Соответствующая итеративная форма подпрограммы изображена на рис. 120 в виде блок-схемы, допускающей дальнейшие упрощения.

3.4.3.3

В общем случае было доказано (Бём, Джакопини (1966 г.), Ашкрофт, Манна (1971 г.)), что за счёт введения дополнительных логических переменных от переходов в блок-схемах программ всегда можно избавиться в том смысле, что можно обойтись одними операторами цикла и соответственно использовать для записи программ только диаграммы Насси — Шнейдермана. Однако такие дополнительные логические переменные являются „дистанционными переключателями" и как таковые приводят по меньшей мере к столь же скверному „дальнодействию", что и переходы.

Довольно сомнительно, чтобы в блок-схеме (b) на рис. 121, содержащей лишь одну петлю и представимой в виде диаграммы Насси — Шнейдермана, достигалась бо́льшая ясность, чем в блок-схеме (a). Пожалуй, здесь лучше было бы подчинить повторение оператора A4 условию $B1 \wedge B2$, после чего ещё раз смотреть, выполнено ли условие B1 (для выбора между A2 и A3), как это реализовано в блок-схеме (c).

3.5. Процедуры

(Вспомогательные) подпрограммы предназначены для сокращения записи часто встречающихся формул и алгоритмов, текстуально совпадающих или имеющих одну и ту же структуру. После того как в этой главе были введены операторы, у нас появились еще и операторы с одинаковой структурой; например, мы уже многократно встречали оператор присваивания вида $v := rest(v)$. Возникает потребность параметризовать также и операторы и благодаря этому иметь возможность употреблять для них сокращённые обозначения. Однако прежде нужно разрешить использовать переменные в качестве параметров. Из-за этого теряется, правда, важный отличительный признак подпрограмм — их чисто функциональный характер. В таком случае мы говорим о „процедурах".

3.5.1. Параметры-переменные

Рассмотрим три просто устроенных оператора

$$m := rest(m),$$
$$(x, y) := (1, 0),$$
$$(a, b) := (b, a).$$

Здесь m, x, y и a, b должны быть переменными, поскольку им что-то присваивается. Так как переменной m присваивается результат применения операции *rest*, m может быть переменной для объектов вида **string** (соотв. типа *string*); можно коротко сказать, что m имеет вид **var string**. Точно также x и y должны быть переменными для объектов, имеющих изображения 0 и 1, т. е. для объектов целого или вещественного сорта. Лишь в специальных случаях вроде „оператора взаимного обмена" $(a, b) := (b, a)$ совершенно не важно, какого сорта те объекты, для которых a и b являются переменными.

Итак, переменные, используемые в качестве параметров оператора, представляют собой, вообще говоря, переменные вполне определённого сорта. Один из основных принципов алгола-68 и паскаля состоит в требовании явного указания сорта параметров-переменных.

Операторы, содержащие присваивания переменным, которые не описаны внутри этих операторов и тем самым нелокальны для них, имеют ***побочные эффекты***; упомянутые переменные являются параметрами. Насколько важно запрещение побочных эффектов для предложений, настолько вредно было бы не допускать их и для операторов; процедура без побочных эффектов была бы безрезультатной. ***Параметры-переменные*** замещаются при вызове переменными. Можно сказать, что *вызванная процедура использует переменные из окружения вызова, она носит паразитический характер по отношению к параметрам-переменным.*

Параметры-переменные обладают чертами, с которыми мы ещё не сталкивались при рассмотрении обычных параметров. Обычные параметры заменяются при вызове фактическими значениями, так что вызовы вроде

$$gcd(18, 18) \quad \text{или} \quad gcd(a, a)$$

вполне корректны.

Однако разные обозначения для переменных, например x и y в $(x, y) := (1, 0)$ или a и b в $(a, b) := (b, a)$, означают, что речь идёт о существенно различных объектах. Отсюда вытекает следующий

Запрет отождествлений для параметров-переменных. *Никакие два параметра-переменных с разными обозначениями не должны при вызове замещаться одной и той же переменной.*

Бессмысленным будет, разумеется, и вызов процедуры с (нетривиальной) формулой в позиции параметра-переменной.

3.5.2. Описания процедур

Параметризованный оператор называется ***процедурой.*** В отличие от подпрограммы процедура не вырабатывает никакого результата, т. е. не может встречаться непосредственно в выражениях. В соответствии с этим в описании не нужно указывать сорт результата. Чтобы процедура давала некоторый эффект, она, как правило, должна содержать по крайней мере один параметр-переменную, хотя наряду с этим могут присутствовать все обсуждавшиеся до сих пор параметры — как обыкновенные, так и параметры-подпрограммы. Параметры процедуры могут быть перечислены в явном виде или опущены. Это относится и к параметрам-переменным. Телом описания процедуры является оператор, а сама процедура выглядит как программный формуляр (см. 3.3.1). Способ записи ***описаний процедур*** похож на запись описаний подпрограмм; единственное отличие, кроме отсутствия сорта результата, состоит в том, что вместо **funct** или **function** пишут соответственно **proc** или **procedure**.

Примеры[1]. (а) Оператор взаимной замены[2]:

```
proc exch ≡ (var int a, b) :          | procedure exch (var a, b : integer) ;
                                      |   var h : integer ;
⌈int h ≡ a ;                          | begin h := a ;
  a := b ;                            |       a := b ;
  b := h   ⌋                          |       b := h
                                      | end
```

(b) „Укорачивание“[3] переменной-цепочки:

```
proc pop ≡ (var string m              | procedure pop (var m : string
                  co m ≠ ◇ co) :      |                {not isempty(m)}) ;
                                      | begin
  m := rest(m)                        |   m := rest(m)
                                      | end
```

---

[1] В стандартном алголе вместо символа **var** в заголовке пишут **ref** (от reference (буквально: упоминание, отсылка). — *Перев.*)

[2] Ниже *exch* — от exchange (обмен, замена). — *Прим. перев.*

[3] В оригинале ‚Poppen‘ — немецкий неологизм, произведённый от английского „pop“ (хлопать, выбивать) и означающий „выталкивание“; отсюда и используемое ниже обозначение *pop*. — *Прим. перев.*

Здесь предохранитель означает не ограничение на *обозначение* фактической переменной для процедуры *pop*, а то, что эта переменная при вызове процедуры должна иметь своим *значением* непустую цепочку.

(с) Система процедур с иерархической структурой: рис. 122[1]. В вызове *exch*(*u*, *v*), фигурирующем в *ord*, процедура *exch* ис-

```
proc ord ≡ (var int u,v):                 | procedure ord(var u,v : integer);
                                          | begin
  if u<v then exch(u,v)                   |   if u<v then exch(u,v)
  else skip                               |          else skip
                                      fi  | end

proc exch ≡ (var int a,b):                | procedure exch(var a,b : integer);
                                          |   var h : integer;
 ⌈int h ≡ a ;                             | begin h := a ;
     a := b ;                             |        a := b ;
     b := h  ⌋                            |        b := h
                                          | end
```

Рис. 122.

пользует параметры *u*, *v* процедуры *ord*; **skip** (соотв. *skip*) — это (пустая) процедура без параметров.

Вложение описаний процедур друг в друга, как и подчинение подпрограмм друг другу,— важное средство структурирования. В паскале в качестве средства структурирования допускаются только процедуры, что значительно затрудняет преобразование программ. (В противоположность этому в алголе-68 вложенными могут быть даже блоки.)

Описания процедур могут также быть рекурсивными:

```
proc pot ≡ (var real avar, var int nvar): | procedure pot(var avar : real; var nvar : integer);
  if nvar>0                               | begin if nvar>0
  then (avar, nvar)                       |       then begin (avar, nvar)
          := (avar↑2, nvar−1);            |                 := (sqr(avar), nvar−1);
      pot(avar, nvar)                     |            pot(avar, nvar)
                                          |          end
  else skip                           fi  |       else skip
                                          | end
```

Отметим, что оператор

$$pot(u, v)$$

означает в точности то же самое, что и $(u, v) := (potz(u, v), 0)$ (см. 3.3.3.1), поскольку *pot*(*avar*, *nvar*) является просто (рекур-

[1] Обозначение *ord* для первой из двух процедур на этом рисунке — от order (порядок). — *Прим. перев.*

сивно сформулированным) сокращением для того оператора цикла, что стоит в теле подпрограммы *polz*.

В подпрограмме *gcd1* из 3.3.2.1 повторяемый оператор можно привести к виду

$$mvar := mod(mvar, nvar);\ (mvar, nvar) := (nvar, mvar),$$

и, значит, его можно реализовать двумя процедурными вызовами

$$mo(mvar, nvar);\ exch(mvar, nvar),$$

где процедура *exch* была описана выше, а *mo* описывается так:

proc *mo* ≡ (**var int** *uvar, vvar*) :
  if *uvar* ≧ *vvar*
  then *uvar* := *uvar* − *vvar* ;
    mo (*uvar, vvar*)
 	
  else skip **fi**
	procedure *mo* (**var** *uvar, vvar* : *integer*) ;
	begin if *uvar* ≧ *vvar*
	    then begin *uvar* := *uvar* − *vvar* ;
	        mo (*uvar, vvar*)
	      end
	    else *skip*
	end

3.5.3. Вызовы процедур

Как видно уже из приведённых выше примеров, вызовы процедур, подобно вызовам подпрограмм, записываются посредством подключения фактических значений параметров процедуры. При этом фактическими значениями обыкновенного параметра являются объекты, получающиеся в результате исполнения формул, а фактические значения параметра-переменной — это просто заменяющие его описанные переменные.

3.5.3.1

Разумеется, в качестве фактических значений можно использовать только такие переменные, область действия которых содержит соответствующий вызов. При этом речь может идти о локально описанной переменной или же снова о параметре-переменной. Первый случай имеет место в приведённом на рис. 123 примере вызова процедуры *ord* в ещё одном варианте подпрограммы *gcd* (см. 3.3.5.1), со вторым мы сталкиваемся при вызове процедуры *exch* в теле процедуры *ord*, а также при рекурсивных вызовах внутри процедур *pol* и *mo*. В обоих случаях речь идёт о простой з а м е н е параметров-переменных неизменными или в свою очередь заменёнными обозначениями, т. е. теми переменными из окружения вызова, с которыми должна вестись работа.

Вызов процедуры — это тоже оператор. В нерекурсивном случае он равносилен программному формуляру, получающе-

```
funct gcd ≡ (int n, int m
             co n > 0 ∧ m > 0 co) int :


 ⌈(var int mvar, nvar) := (m, n) ;
  while mvar ≠ nvar
  do ord (mvar, nvar) ;
     mvar := mvar − nvar od ;

  mvar
                                    ⌋
```

```
function gcd (n, m : integer
        {(n > 0) ∧ (m > 0)}) : integer ;
  var mvar, nvar : integer ;
begin
  (mvar, nvar) := (m, n) ;
  while mvar ≠ nvar
  do begin ord (mvar, nvar) ;
           mvar := mvar − nvar
     end ;
  gcd := mvar
end
```

Рис. 125.

муся из тела процедуры в результате последовательной замены параметров фактическими значениями и переменными. Вложенность процедур порождает при этом некоторую блочную структуру.

3.5.3.2

Рекурсивные же процедуры, как и в случае машины обработки формуляров, ведут прежде всего к появлению различных воплощений программных формуляров, причём в каждом из воплощений реализуется своя собственная замена параметров фактическими значениями и переменными. Лишь в частном случае повторительной рекурсии, проявляющей себя наличием операторов цикла и перехода, можно обойтись без различных воплощений; организация последовательного исполнения описывается в этом случае блок-схемой программы.

При вызове процедур должен соблюдаться запрет отождествлений для параметров-переменных. Его смысл в полной мере проявляется в связи с групповыми присваиваниями. Например, для процедуры

```
proc hp ≡ (var real x, y):
  (x, y) := (1, 0)
```

```
procedure hp (var x, y : real);
begin (x, y) := (1, 0)
end
```

вызов $hp(a, a)$, конечно же, запрещён[1].

В языках программирования, не допускающих записи групповых присваиваний, такого рода явное предостережение отно-

[1] Запрет отождествлений целесообразно сохранять даже в таких специальных случаях (вроде оператора взаимного обмена $(a, b) := (b, a)$), когда нарушение этого запрета не ведёт непосредственно к противоречиям.

сительно отождествлений отсутствует[1]. Однако и здесь пренебрежение запретом отождествлений приводит, вообще говоря, к осложнениям. А именно, в таких языках вместо процедуры *hp* приходится прибегать к её линеаризациям, например к

proc *hp1* ≡ (**var real** *x*, *y*): ⌈*x* := 1; *y* := 0⌋	**procedure** *hp1* (**var** *x*, *y* : *real*); **begin** *x* := 1; *y* := 0 **end**

или же

proc *hp2* ≡ (**var real** *x*, *y*): ⌈*y* := 0; *x* := 1⌋	**procedure** *hp2* (**var** *x*, *y*: *real*): **begin** *y* := 0; *x* := 1 **end**

Обе процедуры *hp1* и *hp2* являются корректными линеаризациями (идеальной) процедуры *hp* и эквивалентны между собой при соблюдении запрета отождествлений; тем не менее вызовы

$$hp1(a, a) \quad \text{и} \quad hp2(a, a)$$

— если их допустить — приводят к разным результатам, а именно $a := 0$ и $a := 1$.

3.5.4. Транзитные параметры и параметры-результаты

В результате выполнения вызова процедуры значение фактической переменной, вообще говоря, изменяется. Такие ***транзитные*** параметры-переменные (***параметры доступа***) представляют собой типичный случай, они уже встречались нам в примерах *exch*, *pop* и *ord*, а также в *pot*.

Однако может случиться и так, что имеющееся в месте вызова значение переменной, являющейся значением параметра, не оказывает никакого влияния на эффект процедуры. В этом случае говорят о ***результатном параметре*** или о ***параметре-результате***. В примере процедуры *hp* мы обнаруживаем самые настоящие параметры-результаты. За счёт использования нескольких параметров-результатов процедуры становятся пригодными также для описания алгоритмов, имеющих пару, тройку, *n*-ку результатов. Для подпрограмм алгол-68 и паскаль такой возможности не допускают (без явного введения составных объектов).

С параметров-результатов можно „урвать клок шерсти", используя их в теле процедуры как вспомогательные переменные.

Пример. Из тела подпрограммы *mod1* (см. 3.3.2.2) получается процедура вычисления остатка от деления целых чисел,

---

[1] В сообщении об алголе-68 важность запрета отождествлений была недооценена, по-видимому, из-за отсутствия в алголе групповых присваиваний. Руководство по паскалю гласит: "...schould be distinct variables." [„...*должны быть различными переменными.*" (*англ*). — *Перев.*]

которую можно достроить до процедуры, находящей одновременно и частное:

```
proc moddiv1 ≡ (var int r,q, int m,n        | procedure moddiv1 (var r,q : integer ; m,n : integer
               co m≥0 ∧ n>0 co):            |                           {(m≥0) and (n>0)}) ;
                                            | begin
⌈q:=0 ; r:=m ;                              |   q:=0 ; r:=m ;
 while r≥n                                  |   while r≥n
 do q:=q+1 ;                                |   do begin q:=q+1 ;
    r:=r−n od⌋                              |            r:=r−n
                                            |      end
                                            | end
```

В этой процедуре m, n — обыкновенные параметры, а добавленные параметры-переменные r, q — параметры-результаты. Для r, q характерно, что они не обязаны иметь каких-либо значений при входе в процедуру, а коли даже они имеют какие-то значения, то эти значения несущественны. (Именно по этой причине почти все языки программирования разрешают описывать переменные и без их инициализации.) Если теперь, скажем, обыкновенный параметр m во всех вызовах процедуры *moddiv1* представлен значением некоторой программной переменной *mvar*, которая после этих вызовов больше не используется, то переменную *mvar* можно отождествить с одним из параметров-результатов, лучше с r, и получить

```
proc moddiv2 ≡ (var int r, q, int n         | procedure moddiv2 (var r, q:
               co r ⩾ 0 ∧ n > 0 co):        | integer; n:integer {(r⩾0) and (n>0)});
                                            | begin
⌈q:=0; r:=r;                                |   q := 0;  r := r;
 while r ⩾ n                                |   while r ⩾ n
 do q := q + 1;                             |   do begin q := q + 1;
    r := r − n od⌋                          |            r := r − n
                                            |      end
                                            | end
```

Отождествление *mvar* с r даёт вырожденное присваивание $r := r$ и тем самым позволяет повысить эффективность процедуры.

Хотя ни в алгольном, ни в паскалевском варианте никаких различий в записи параметров-переменных r и q не делается, однако природа их весьма различна: если q — это по-прежнему параметр-результат, то переменная r при входе в процедуру должна обладать значением делимого, а после окончания указывает остаток. Таким образом, параметр-переменная r изменяется процедурой, т. е. r — транзитный параметр.

Чтобы в нашем примере целочисленного деления обойтись без обыкновенных параметров, т. е. всего лишь двумя (транзитными) параметрами-переменными, значениями которых являются вначале делимое и делитель, а в конце остаток и част-

ное, нужно использовать локальное для процедуры вспомогательное обозначение:

```
proc moddiv3 ≡ (var int r, q                  | procedure moddiv3(var r, q : integer
                co r ≥ 0 ∧ q > 0 co):          |                 {(r ≥ 0) and (q > 0)});
                                               |   var n : integer;
                                               | begin
 ⌈int n ≡ q , q := 0;                          |   n := q ; q := 0;
  while r ≥ n                                  |   while r ≥ n do
  do r := r − n ; q := q + 1 od ⌋.             |   begin r := r − n ; q := q + 1 end
                                               | end
```

Разумеется, подпрограммы с простым результатом также можно преобразовать в процедуры за счёт введения параметров-результатов. По этой причине в некоторых (устаревших) языках программирования подпрограммы отсутствуют вовсе — в них предусмотрены только процедуры.

Транзитный параметр всегда можно расщепить на один обыкновенный и один параметр-результат. Это продемонстрировано на приводимом ниже примере процедуры с транзитным параметром для „счёта", выполняемого „в некоторой заданной переменной":[1]

```
proc increm ≡ (var int x):    | procedure increm (var x : integer);
  ⌈x := x + 1⌋                | begin x := x + 1 end
```

Выходное значение параметра-переменной „портится" вызовом процедуры *increm*. Чтобы избежать этого, нужно „расщепить" транзитный параметр:

```
proc incr ≡ (int n, var int x):   | procedure incr (n : integer;
                                  |    var x : integer);
  ⌈x := n + 1⌋                    | begin x := n + 1 end
```

Теперь эффект вызова *increm*(a) достигается специальным вызовом *incr*(a, a). Отметим, что в последнем вызове нет никакого нарушения запрета отождествлений!

3.5.5. Входные параметры

Конечно, в случае процедур вместо обычных параметров всегда можно было бы вводить параметры-переменные. Например, исходя из процедуры *moddiv1*, получим

```
proc moddiv4 ≡ (var int m, n, r, q              | procedure moddiv4(var m, n, r, q : integer
                co m ≥ 0 ∧ n > 0 co):            |                 {(m ≥ 0) and (n > 0)});
                ~~~                              |                 ~~~
```

[1] Ниже *increm* — от increment (приращение). — *Прим. перев.*

причём никаких изменений в теле процедуры по сравнению с *moddiv1* не требуется. В этом теле нет ни одного присваивания параметрам-переменным m, n, т. е. их значения остаются неизменными. Такие параметры-переменные называются ***входными параметрами***. Например, параметр *vvar* процедуры *mo* из 3.5.2 является входным.

Употребление входных параметров вместо обыкновенных связано с тем неудобством, что в качестве фактического параметра вызова можно использовать только переменную, но не формулу. Понятие входного параметра берёт начало от машинно-зависимого представления о „доступе только по чтению“ (*'read-only'*[1]).

3.5.6. Подавляемые параметры

Параметры-переменные можно опускать („подавлять“), если их замена не предусматривается.

Примером полного подавления параметров служит следующая процедура *счёт*, которую можно использовать, например, для подсчёта того, сколько раз употребляется определённая операция в данном алгоритме:

proc *счёт* ≡ :⌈$x := x + 1$⌋	**procedure** *счёт*; **begin** $x := x + 1$ **end**

Предупреждение о том, что x — это (глобальная) переменная, а не обозначение обыкновенного объекта, снимается в алголе-68 и паскале вместе с упразднением списка параметров.

```
funct potz ≡ (real a, int n                function potz (a : real ; n : integer
                      co n ≧ 0 co) real :                          {n ≧ 0}) : real ;
                                              var avar : real ; nvar : integer ;
⌈proc po ≡                                    procedure po ;
    : if nvar > 0                               begin if nvar > 0
    then (avar, nvar) :=                          then begin (avar, nvar) :=
                    (avar ↑2, nvar − 1) ;                (sqr (avar), nvar − 1) ;
        po                                              po                  end
    else skip                    fi ;             else skip
                                                end ;
(var real avar, var int nvar) := (a, n) ;   begin (avar, nvar) := (a, n) ;
po ;                                              po ;
avar                                  ⌋          potz ⇐ avar
                                            end
```

Рис. 124

[1] „Только читать“ (*англ.*). — *Прим. перев*

Для того чтобы отличить процедуры, в которых все параметры опущены, от обыкновенных операторов, в алголе-68 перед ними ставят двоеточие. При вызове обозначение такой процедуры записывается прямо между точками с запятой или скобками-уголками.

Опуская неизменные параметры *avar* и *nvar* рекурсивной процедуры *pot* из 3.5.2, мы получим рекурсивную процедуру *po* без параметров. Её можно использовать в подпрограмме *potz* в формулировке, представленной на рис. 124. И в рекурсивной процедуре *mo* из 3.5.2 также можно опустить оба неизменных параметра *uvar* и *vvar*.

Эффект процедуры с опущенными параметрами-переменными всецело зависит от предыстории. Поскольку подавление параметров-переменных ведёт к побочным эффектам, которые легко упустить из виду, не стоит применять его бездумно. В частности, при *частичном* подавлении параметров-переменных может случиться так, что останется незамеченным нарушение запрета отождествлений, как в следующем примере процедуры „удвоения" с глобальным параметром-переменной n:

proc *удв* ≡ (**var int** *u*): ⌈$u := u \times 2$; $n := n + 1$⌋	**procedure** *удв* (**var** *u* : *integer*); **begin** $u := u * 2$; $n := n + 1$ **end**

В последовательности вызовов

удв(a); *удв*(b); *удв*(a); *удв*(n)

последний нарушает запрет отождествлений (и даёт $n := 2 \times n + 1$, что явно не совпадает с тем, что мы ожидаем).

3.5.7. Процедуры как средство структурирования

Процедуры представляют собой полезное средство членения программного текста. Программистская методика, делающая упор на работу с программными переменными (например, по соображениям машинной аргументации), склонна к выдвижению процедур на передний план как самостоятельных конструкций. (Это, в частности, относится к случаю, когда желательно постоянно печатать результаты в ходе исполнения некоторой процедуры. Здесь было бы лучше выразить в явном виде, что в качестве результата желательно иметь последовательность объектов.) Процедуры как средство организации иерархической структуры особенно важны в паскале, где они являются единственным пригодным для этой цели инструментом. В качестве

примера рассмотрим операцию замены (*) в алгоритмах Маркова (см. 1.6.4.1). Вспомогательной процедурой будет служить поэлементно сравнивающая „разборка“ (слева) двух строк (цепочек символов) a, v: [1]

proc *subtr* ≡ (**var string** *a*, *b*) :	**procedure** *subtr*(**var** *a*, *b* : *string*) ;
	begin
while ($a \neq \diamond \ \wedge \ b \neq \diamond$) $\wedge$	**while not** (*isempty* (*a*) **or** *isempty* (*b*))
first(*a*) = *first*(*b*)	$\wedge$ *first*(*a*) = *first*(*b*)
do (*a*, *b*) := (*rest*(*a*), *rest*(*b*)) **od**	**do** (*a*, *b*) := (*rest*(*a*), *rest*(*b*))
	end

Нам нужно заменить первое слева вхождение строки A в строку B на строку C. После введения двух переменных a, b с их инициализацией значениями A и B соответственно и вызова $subtr(a, b)$ ситуация такова. Если $a = \diamond$, то A является началом B, т. е. $B = A + b$, и A в B можно заменить на C. Далее, если $a \neq \diamond$, но $b = \diamond$, то A длиннее B, и интересующая нас замена вообще невозможна. В оставшемся случае нужно попробовать предпринять замену A на C в строке $rest(B)$, а перед полученным результатом поставить знак $first(B)$. Таким образом, используя вспомогательную процедуру *subtr*, мы получаем рекурсивную формулировку, приведённую на рис. 125. Можно перевести ее в итеративную форму, вводя две новые переменные — одну для не являющегося неизменным параметра B и другую для построения результата.

funct *ersetze* ≡ (**string** *A*, *B*, *C*) **string**:	**function** *ersetze*(*A*, *B*, *C* : *string*) : *string* ;
	var *a*, *b* : *string* ;
	begin
⌈(**var string** *a*, *b*) := (*A*, *B*) ;	(*a*, *b*) := (*A*, *B*) ;
subtr(*a*, *b*) ;	*subtr*(*a*, *b*) ;
if $a = \diamond$	**if** *isempty* (*a*)
then $C + b$	**then** *ersetze* := *conc* (*C*, *b*)
elsf $b = \diamond$	**else if** *isempty* (*b*)
then B	**then** *ersetze* := *B*
else ⟨*first*(*B*)⟩ +	**else** *ersetze* := *prefix*(*first*(*B*),
ersetze(*A*, *rest*(*B*), *C*) **fi** ⌋	*ersetze*(*A*, *rest*(*B*), *C*))
	end

Рис. 125.

[1] Ниже *subtr* — от subtraction (вычитание), а ersetze (*нем.*) означает „замени“. — *Прим. перев.*

3.5.8. Рекурсивное определение повторения

3.5.8.1. Переход как гладкий вызов процедуры без параметров

В подпрограмме *potz* из раздела 3.5.6 нам встретилась процедура (без параметров) такого вида:

```
⌈proc ℛ ≡

   : if ›условие‹
       then ›оператор‹;
            ℛ
       else skip        fi;

 ℛ                        ⌋
```

```
procedure ℛ;
  begin
  if ›условие‹
    then begin ›оператор‹;
            ℛ             end
    else skip
  end;
begin
ℛ
end
```

Если сравнить эту формулировку с соответствующей формулировкой, в которой используются операторы перехода, а именно со следующей петлей (см. 3.4.2.2):

```
⌈
 rep: if ›условие‹
       then ›оператор‹;
              goto rep
       else skip        fi
                         ⌋
```

```
   begin
l: if ›условие‹
   then begin ›оператор‹;
          goto l       end
   else skip
end
```

то становится понятно, что операторы перехода — это просто вырожденные **регулярные** вызовы (рекурсивных) процедур без параметров.

Таким образом, кто хочет запретить операторы перехода, должен запретить и рекурсивные процедуры.

3.5.8.2. Рекурсивное определение условного повторения

После сказанного выше ясно, что оператор цикла (см. 3.3.2)

```
while ›условие‹
  do ›оператор‹ od
```

```
while ›условие‹
do ›оператор‹
```

может быть определен как процедура (*) из 3.5.8.1.

При помощи подстановки получается явное рекурсивное определение:

```
if ›условие‹                    | if ›условие‹
  then ›оператор‹;              |   then begin ›оператор‹;
       while ›условие‹          |              while ›условие‹
       do ›оператор‹ od         |              do ›оператор‹ end
  else skip            fi       |   else skip
```

Таким образом, тело процедуры *pot* из 3.5.2 и тело процедуры без параметров *po* из 3.5.6 запишутся совершенно одинаково в форме с оператором цикла (см. также *polz* в 3.3.3.1):

```
                                  | begin
while nvar > 0                    |   while nvar > 0
do (avar, nvar) :=                |   do (avar, nvar) :=
       (avar ↑ 2, nvar − 1) od    |          (sqr(avar), nvar − 1)
                                  | end
```

3.5.8.3. Рекурсивное определение повторения с пересчётом

Повторение часто связано со счётом, как это имеет место, например, в *potz*, где счёт ведётся в переменной *avar*, начиная с *a*, „вниз“ до 0. Для этой цели вводят, следуя Гутисхаузеру (1951 г.), специальное сокращение — ***оператор цикла, работающий по счётчику*** (или ***с пересчётом***), где счётчик — это параметр цикла, который пробегает целые значения от ***начального*** ›start‹ до ***конечного*** ›goal‹ [1] (включительно) с ***шагом*** ›step‹ [2] (причем ›step‹ $\neq 0$):

for *счётчик* **from** ›**start**‹ **by** ›**step**‹ **to** ›**goal**‹ **do** ›оператор‹ **od**
(алгол-68)[3].

Повторяемый оператор ›оператор‹ может при этом содержать переменную *счётчик*.

Повторение с пересчётом можно определить как применение некоторой рекурсивно определённой процедуры *G*. А именно, оператор цикла, работающий по счётчику:

for *счётчик* **from** ›start‹ **by** ›step‹ **to** ›goal‹ **do** ›оператор‹ **od**

---

[1] Цель (*англ.*). — *Прим. перев.*
[2] Шаг (*англ.*). — *Прим. перев.*
[3] Соответствующий русскоязычный вариант выглядит так:

для *счётчик* **от** ›старт‹ **через** ›шаг‹ **до** ›финиш‹ **цк** ›оператор‹ **кц**.

равносилен (при условии, что значение, получающееся при вычислении ⟩step⟨, положительно) следующему оператору:

```
⌈(int start, step, goal) ≡ (⟩start⟨, ⟩step⟨, ⟩goal⟨);
 proc G ≡ (int счётчик):
   if счётчик ⩽ goal
     then ⟩оператор⟨; G(счётчик + step)
     else skip                               fi
       G(start)                               ⌋
```

и заканчивается тогда и только тогда, когда заканчивается G. По сути своей *счётчик* — это обозначение параметра, связанное в повторении, порождаемом процедурой G. Поэтому вне

```
funct potz ≡ (real a, int n                  | function potz (a : real; n : integer
               co n ≧ 0 co) real:            |                     {n ≧ 0}) : real;
                                             |   var avar : real; i : integer;
                                             | begin
 ⌈var real avar := a;                        |   avar := a;
  if n ≧ 1                                   |   if n ≧ 1
  then for i from n by −1 to 1               |   then for i := n downto 1
    do avar := avar↑2 od                     |          do avar := sqr(avar)
  else skip                        fi;       |   else skip;
  avar                              ⌋        |   potz ⇐ avar
                                             | end
```

Рис. 126.

повторения обозначение *счётчик* смысла не имеет. Программной переменной *счётчик* не является; присваивание *счётчику* бессмысленно. В стандартном алголе-68 это экранирование параметра цикла явно описано — хоть и неуклюже, но корректно. В паскале же, который в этом отношении следует ещё алголу-60, приходится описывать *счётчик* как программную переменную вне повторения с пересчётом, для чего приходится, далее, вводить взятые „с потолка" абсурдные ограничения вроде „⟩оператор⟨ не должен содержать присваиваний *счётчику*" и „после окончания повторения значение *счётчика* не определено".

Если ⟩step⟨ < 0, то проверять нужно условие *счётчик* $\geqslant$ $\geqslant$ *goal*.

Впрочем, в паскале допускаются лишь случаи $step = 1$ и $step = -1$; в этих случаях пишут соответственно

for *счётчик* := ⟩start⟨ **to** ⟩goal⟨ **do** ⟩оператор⟨ (паскаль)

и

for *счётчик* := ⟩start⟨ **downto** ⟩goal⟨ **do** ⟩оператор⟨ (паскаль).

В алголе-68 тоже разрещается опускать **by**⟩шаг⟨, если $step = 1$:

for *счётчик* **from** ⟩start⟨ **to** ⟩goal⟨ **do** ⟩оператор⟨ **od** (алгол-68).

Примеры. Подпрограмму *potz* из 3.3.3 и 3.4.2 можно теперь записать, как показано на рис. 126. Здесь повторяемый

funct *potz* ≡ (**real** *a*, **int** *n* **co** $n \geq 0$ **co**) **real**:	**function** *potz* (*a* : *real*; *n* : *integer* {$n \geq 0$}) : *real*;
	var *avar* : *real*; *i* : *integer*;
	begin
⌈**var real** *avar* := *a*;	*avar* := *a*;
if $n \geq 1$	**if** $n \geq 1$
then for *i* **from** 1 **to** *n*	**then for** *i* := 1 **to** *n*
do *avar* := *avar* ↑ 2 **od**	**do** *avar* := *sqr* (*avar*)
else skip **fi**;	**else** *skip*;
avar ⌋	*potz* ⇐ *avar*
	end

Рис. 127

оператор *avar* := *avar* ↑ 2 (соотв. *avar* := *sqr* (*avar*)) не содержит вхождений счётчика *i*. Поэтому счётчик может пробегать значения и в обратном порядке (рис. 127).

Начальный 12-членный отрезок бесконечного ряда $\sum_{i=1}^{\infty} \frac{1}{i^4}$ для $\pi^4/90$ можно получить так:

	var *s* : *real*; *i*: *integer*;
	begin
⌈**var real**⟩ *s* := 0;	*s* := 0;
for *i* **from** 1 **to** 12	**for** *i* := 1 **to** 12
do *s* := *s* + ((1/*i*) ↑ 2) ↑ 2 **od**;	**do** *s* := *s* + *sqr*(*sqr*(1/*i*));
s ⌋	⟩Res⟨ ⇐ *s*
	end

Таким образом, в стандартном алголе-68 не требуют, чтобы повторение с пересчётом заканчивалось при *счётчик* = *goal*; повторение обрывается, если

при $step > 0$ *счётчик* > *goal*

или

при $step < 0$ *счётчик* < *goal*.

3.6. Массивы

По технологическим причинам архитектура большинства вычислительных машин строится так, что (программные) пере-

менные располагают в линейном („одномерном“) порядке. Это имеет далеко идущие последствия, касающиеся распространенных программистских „привычек“.

3.6.1. Индексированные переменные

Если вам понадобилось большое число переменных для объектов одного и того же сорта — неважно, в качестве параметров или локальных переменных,— то рекомендуется не описывать их по отдельности, а ввести ***массив***, т. е. набор ***индексированных переменных*** (алгол-60).

В качестве ***индексов*** берут, как правило, целые числа, получая то преимущество, что с такими индексами можно выполнять вычисления. Обычно индексы в массиве образуют, кроме того, (целочисленную) последовательность без пропусков. Тогда для задания ***диапазона индексов*** достаточно указать нижнюю и верхнюю границы.

Для описания массива используют следующие способы записи[1]:

[1..5] **var int** *a*	**var** *a* : **array** [1..5] **of** *integer*
[0..*n*] **var real** *q*	**var** *q* : **array** [0..*n*] **of** *real*

Предостережение. Если для задания диапазона индексов использованы обозначения, отличные от изображений, то это значит, что речь идёт о глобальных параметрах, которые должны быть описаны раньше, на объемлющем уровне и значения которых (в случае переменных) должны быть заданы[2]; формулы в месте описания вычисляются раз и навсегда. В паскале наряду с изображениями здесь разрешаются только особым образом описанные константы (см. первое подстрочное примечание в разделе 2.2.2.1).

В алголе[3] разрешена также инициализация вида

[1..5] **var int** $a := (1, 2, 4, 5, 16)$.

Одиночная переменная выбирается из массива переменных при помощи указания индекса:

$$a[2] \quad a[i+1] \quad q[0] \quad q[r \times s],$$

причём индекс должен находиться внутри границ диапазона, указанного в соответствующем описании. На индексных пози-

---

[1] В стандартном алголе-68 символ **var** снова опускается.

[2] Таким образом, речь идет не о сменном параметре, который можно было бы задавать, скажем, в том же самом заголовке, здесь требуется двухступенчатая параметризация (см. 2.6.3).

[3] В стандартном алголе-68 — лишь в том случае, когда индексация начинается с 1.

циях могут стоять и формулы с целочисленным результатом, но, конечно, не с вещественным, так что в случае, когда, скажем, используется деление ./., нужно явно указывать округление.

Такая индексированная переменная для объектов определённого сорта может употребляться повсюду, где до того могли находиться переменные (***простые переменные***) для объектов этого сорта, например в выражениях:

$2 \times a[2]$	$2 * a[2]$
$a[i+1] \uparrow 2$	$sqr(a[i+1])$
$sin(q[r \times s])$	$sin(q[r * s])$
$a[i \times (i-1) \div 2]$	$a[i * (i-1)$ **div** $2]$
a [**round** $(i \times (i-1)/2]$	$a[trunc(i * (i-1)/2]$,

в левой части присваиваний:

$$a[i-1] := 3$$
$$q[0] := 8.34/5.62$$

и в качестве (фактических) параметров процедур с параметрами-переменными:

$$ord(a[i], a[k])$$
$$hp(q[r], q[s]).$$

Удобной возможности в ы ч и с л е н и я индексированной переменной противостоит тот недостаток, что это вычисление требует затрат — даже в тривиальных случаях вроде $a[i]$ или $q[r]$. Поэтому одноэлементный массив индексированных переменных лучше заменить простой переменной.

Другую проблему доставляет запрет отождествлений; в его соблюдении нельзя теперь убедиться „с одного взгляда“. Так, для вызова

$$hp(q[r], q[s])$$

только при выполнении условия предохранителя $r \neq s$ гарантировано, что запрет отождествлений не нарушен.

3.6.1.1

Параметром может служить также целый массив индексированных переменных, причём сорты параметров записываются по существу так же, как в описании.[1, 2]

---

[1] В стандартном алголе [. .. .]var λ заменяется на ref[. .. .]λ.

[2] В стандартном паскале для типа array[0 .. 20] of *integer* сначала нужно ввести сокращение — некоторый идентификатор, после чего это сокращение используется в заголовке.

Пример. Нахождение минимума и максимума значений для 21-элементного массива переменных:[1]

```
proc minmax ≡ ([0..20] var int a,              | procedure minmax (var a : array[0..20] of integer;
                       var int min, max);       |                       var min, max : integer);
                                                |   var i : integer;
⌈(min, max) := (a[0], a[0]);                    | begin (min, max) := (a[0], a[0]);
 for i from 1 to 20                             |   for i := 1 to 20
 do                                             |   do begin
    if a[i] < min                               |      if a[i] < min
    then min := a[i] else skip        fi;       |      then min := a[i] else skip;
    if a[i] > max                               |      if a[i] > max
    then max := a[i] else skip                  |      then max := a[i] else skip
                                  fi od ⌋       |      end
                                                | end
```

В этом примере *min* и *max* — чистые параметры-результаты; кроме того, ни одной из 21 индексированных переменных $a[0], \ldots, a[20]$ нет ни одного присваивания, они являются чистыми входными параметрами.

3.6.1.2

Транзитные параметры имеются в приведённом ниже примере ***сортировки разделением*** для массива индексированных переменных. Идея сортировки разделением подобна идее сортировки слиянием (см. пример (d) в разделе 1.6.2 и систему подпрограмм (*sort*, *merge*) из примера (f) раздела 2.3.2): если данное слово (содержащее по крайней мере два элемента) разложить не на два произвольных подслова, а на два непустых подслова (каждое из которых короче исходного) *lcut*(*a*) и *rcut*(*a*), образующих ***сечение***, т. е. таких, что для каждого знака *u* из *lcut*(*a*) и каждого *v* из *rcut*(*a*) справедливо неравенство $u < v$, то после сортировки *lcut*(*a*) и *rcut*(*a*) никакого специального слияния проводить уже не нужно — можно просто соединить отсортированные подслова. Тем самым получаем подпрограмму[2]

```
funct cutsort ≡ (string a) string:            | function cutsort(a : string) : string;
                                              | begin
if length(a) ≤ 1                              |   if length(a) ≤ 1
  then a                                      |   then cutsort ⇐ a
  else cutsort(lcut(a)) +                     |   else cutsort ⇐ conc(cutsort(lcut(a)),
                   cutsort(rcut(a)) fi        |                     cutsort(rcut(a)))
                                              | end
```

[1] Ниже *l* и *r* в *lcut* и *rcut* — соответственно от left (левый) и right (правый), а cut означает „разрез". — *Прим. перев.*

[2] Ниже *cutsort* — от cut и sorting (сортировка). — *Прим. перев.*

Собственно работа перекладывается на операции *lcut* и *rcut*. Выбор *lcut*, *rcut* не детерминирован: взяв любой ***секущий знак*** x сорта **char**, можно получить *lcut* и *rcut* как последовательности таких знаков u из a, для которых выполняется условие $u \leqslant x$ или соответственно $u > x$ (в предположении, что выбор x произведён так, что ни одна из этих подпоследовательностей не пуста. Если в качестве x взять какой-нибудь знак

```
funct cutsort1 ≡ (string a) string:

if a = ◇
then a
else cutsort1 (lcut1 (rest (a), first (a))) + first (a))
     + cutsort1 (rcut1 (rest (a), first (a)))
                                                     fi
```

```
function cutsort1 (a : string) : string;
begin
if isempty(a)
then cutsort1 ⇐ a
else cutsort1 ⇐ conc(cutsort1 (lcut1 (rest (a), first (a))),
     prefix (first (a), cutsort1 (rcut1 (rest (a), first (a)))))
end
```

Рис. 128.

слова a, то $lcut(a)$ наверняка будет содержать его, т. е. будет непустым словом; однако $rcut(a)$ может оказаться пустым; так будет, например, в случае, когда все знаки в слове a одинаковы, а значит, $lcut(a) = a$. Итак, для обеспечения завершения будет лучше, если при условии $a \neq \diamondsuit$ мы удалим из a один знак — скажем, $first(a)$ — и используем его в качестве секущего знака x. Если обозначить теперь через $lcut1(a, x)$ и $rcut1\ (a, x)$ такие (возможно, пустые) подслова слова a, образующие сечение, что $lcut1(a, x) + x + rcut1(a, x)$ представляет собой перестановку знаков слова a, то получится алгоритм, представленный на рис. 128.

Пусть теперь нужно так переставить значения $a[m]$, $a[m+1]$, ..., $a[n]$ заданного массива индексированных переменных $[s\,..\,t]$ **var char** a (соотв. **var** a : **array** $[s\,..\,t]$ **of** *char*), чтобы они стали упорядочены. Идея сортировки разделением наводит на мысль создать сначала такое сечение, чтобы все переменные с индексами, не превосходящими некоторого определённого индекса, содержали значения из левого множества сечения, а все последующие переменные массива — значения из первого множества сечения. В качестве секущего возьмём знак $a[f]$, где f — произвольно выбранный индекс, лежащий в рассматриваемом диапазоне индексов. Просмотр массива начинается с $i = m$ и $j = n$. Если для некоторого i выполнено неравенство $a[i] \leqslant a[f]$, или для некоторого j — неравенство $a[j] \geqslant a[f]$, то такой знак стоит уже „правильно“, так что можно увеличить i или соответственно уменьшить j. Если же не справедливо ни одно из этих неравенств, то $a[j] < a[i]$. После обмена переменных $a[j]$ и $a[i]$ значениями оба знака будут стоять „правильно“, так что можно увеличить i и уменьшить j. Во время выполнения описанного условного повторения

постоянно выполнено следующее условие:

$$\forall k \in \mathbb{N}: \ (j < k \leqslant n \Rightarrow a[f] \leqslant a[k]) \land (m \leqslant k < i \Rightarrow a[k] \leqslant a[f]).$$

По окончании условного повторения это условие по-прежнему выполнено, а кроме того, $i = j + 1$. Теперь в зависимости от позиции секущего элемента он ещё переставляется, если надо, в

```
proc quicksort = ([s..t] var char a, int m, int n
                                co s ≤ m ∧ n ≤ t co)

  if m < n
  then (var int i, j) := (m, n);
    int f = «irgendeine ganze Zahl f: m ≤ f ≤ n»;
    while i ≤ j
    do if a[i] ≤ a[f]
       then i := i + 1
       ▯ a[f] ≤ a[j]
        then j := j − 1
       ▯ a[j] < a[f] ∧ a[f] < a[i]
       then exchc(a[i], a[j]) fi od,

co i = j + 1 ∧ ∀k . (j < k ≤ n ⇒ a[k] ≥ a[f])
                   ∧ (m ≤ k < i ⇒ a[k] ≤ a[f]) co
    if f = i
    then i := i + 1
    ▯ f = j
    then j := j − 1
    ▯ f < j
    then exchc(a[f], a[j]), j := j − 1
    ▯ i < f
    then exchc(a[f], a[i]); i := i + 1      fi;
    co i = j + 2 co
    quicksort (a, m, j);
    quicksort (a, i, n)
  else skip                                 fi
```

```
procedure quicksort (var a: array [s..t]
            of char; m, n: integer
            {(s ≤ m) and (n ≤ t)});
  var i, j, f: integer;
begin
if m < n
then begin (i, j) := (m, n);
  f := «irgendeine ganze Zahl f: m ≤ f ≤ n»;
    while i ≤ j
    do if a[i] ≤ a[f]
       then i := i + 1
       ▯ a[f] ≤ a[j]
       then j := j − 1
       ▯ (a[j] < a[f]) and (a[f] < a[i])
       then exchc(a[i], a[j]);

         {i = j + 1 and ∀k: (j < k ≤ n ⇒ a[k] ≥ a[f])
         and (m ≤ k < i ⇒ a[k] ≤ a[f])}
       if f = i
       then i := i + 1
       ▯ f = j
       then j := j − 1
       ▯ f < j
       then begin exchc(a[f], a[j]); j := j − 1 end
       ▯ i < f
       then begin exchc(a[f], a[i]); i := i + 1 end;
       {i = j + 2}
       quicksort (a, m, j);
       quicksort (a, i, n)
       end else skip
 end
```

Рис. 129. [Здесь и на рис. 130 слова „Irgendeine ganze Zahl" (*нем.*) означают „какое-нибудь целое число". — *Перев.*]

„точку сечения" и либо i увеличивается, либо соответственно j уменьшается. Тем самым справедливы условия $j - m < n - m$ и $n - i < n - m$, гарантирующие завершение. В совокупности получается представленная на рис. 129 рекурсивная процедура *quicksort*[1] для „сортировки на месте" (опирающаяся на некоторую процедуру exchc замены знаков), принцип которой описал Ч. А. Р. Хоар в 1961 г.

Этот алгоритм можно ещё на разные лады линеаризовать и усовершенствовать, например так, как показано на рис. 130 (Хоар, 1971 г.). То что алгоритм завершается, вытекает из того

---

[1] От quick (быстрый). — *Прим. перев.*

факта, что после окончания повторения выполняются (наряду с неравенством $i > j$ также и) неравенства $i > m$ и $j < n$, поскольку i по крайней мере один раз увеличивалось на 1, j по крайней мере один раз уменьшалось на 1. В противном случае мы имели бы $a[i] < a[f]$ для всех i, таких что $m \leqslant i \leqslant n$, т. е. и для $i = f$; $a[f] < a[f]$, а это невозможно. Таким образом, как

```
proc quicksort =
        ([s..t] var char a, int m, int n
        co s≤m ∧ n≤t co):



  if m<n
  then
    var int i:=m;
    var int j:=n;
    int f = «irgendeine ganze Zahl f: m≤f≤n»;
    char x = a[f];
    while i≤j

    do while a[i]<x do i:=i+1 od;
       while x<a[j] do j:=j-1 od;
       if i≤j
       then
            char w = a[i];
            a[i]:=a[j]; a[j]:=w;
            i:=i+1; j:=j-1

       else skip                    fi
                                            od;
    quicksort (a, m, j);
    quicksort (a, i, n)
  else skip                                 fi
```

```
procedure quicksort
      (var a: array [s..t] of char; m, n: integer
                         {(s≤m) and (n≤t)});
  var i, j, f: integer; x, w: char;
begin
if m<n
then begin
     i:=m;
     j:=n;
     f:=«irgendeine ganze Zahl f: m≤f≤n»;
     x:=a[f];
     while i≤j
     do begin
        while a[i]<x do i:=i+1;
        while x<a[j] do j:=j-1;
        if i≤j
        then begin
             w:=a[i];
             a[i]:=a[j]; a[j]:=w;
             i:=i+1; j:=j-1
             end
        else skip
        end;
     quicksort (a, m, j);
     quicksort (a, i, n)
     end else skip
end
```

Рис. 130.

$quicksort(a, m, j)$, так и $quicksort(a, i, n)$ „обрабатывают" собственные подмножества множества индексов, обрабатываемого $quicksort(a, m, n)$.

Заметим, что два вызова процедуры *quicksort* в третьей и четвёртой снизу строчках на рис. 130 имеют дело с непересекающимися подмножествами индексированных переменных, так что эти вызовы можно было бы выполнять и параллельно.

Эффективность алгоритма зависит от выбора индекса f. Выбор $f = m$ или $f = n$ приводит к тому, что даже для уже упорядоченного массива требуется целых $(t - s + 1)^2$ тактов. Хоар первоначально советовал выбрать f из заданного диапазона случайно, а позднее рекомендовал брать просто $f = (m + n)$ div 2 — этот выбор практически столь же хорош.

3.6.2. Многомерные массивы

Элементы массива могут быть переменными для объектов произвольного (но всегда одного и того же) сорта. В частности, элементы некоторого массива сами могут быть массивами, т. е. переменные можно индексировать многократно („многоступенчатые", или ***многомерные массивы***). Вот пример описания двумерного массива:

[1..2][1..5] **var int** *kappa*	**var** *kappa*: **array** [1..2] **of array** [1..5] **of** *integer*;

или, короче,

[1..2,1..5] **var int** *kappa*	**var** *kappa*: **array** [1..2,1..5] **of** *integer*;

трёхмерного массива:

[0..*r*,0..*s*,0..*t*] **var bool** *f*	**var** *f*: **array** [0..*r*,0..*s*,0..*t*] **of** *Boolean*;

Для массива *kappa*

kappa [2]

означает второй элемент этого массива, т. е. массив из пяти элементов, а

kappa [2] [5]

или, короче,

kappa [2, 5]

— последний элемент только что упомянутого пятиэлементного массива.

3.6.2.1

Двумерные массивы можно рассматривать как матрицы, выделяя в них обычным образом столбцы и строки. Тогда $a[i, j]$ понимается как j-й элемент i-й компоненты массива a, т. е. j-й элемент i-й строки матрицы a, или элемент, находящийся на пересечении i-й строки и j-го столбца.

Пример. Умножение двух матриц: рис. 131[1]. Здесь a, b — чистые входные параметры, c — чистый параметр-результат. Переменную для суммирования *sum* в алголе можно было бы описать и в са́мом внутреннем цикле; можно и вовсе обойтись без неё, если сумму формировать сразу в $c[i, k]$. Однако, поскольку для всякой индексированной переменной её положение в массиве должно вычисляться, указанная форма является,

[1] *mult* в названии процедуры — от multiplication (умножение). — *Прим. перев*

```
proc matrixmult =
          ([1..r, 1..s] var real a,
           [1..s, 1..t] var real b,
           [1..r, 1..t] var real c):
 ⌈var real sum;
  for i from 1 to r
  do for k from 1 to t
     do sum := 0;
        for j from 1 to s
        do sum := sum + a[i, j] × b[j, k]   od;
        c[i, k] := sum                  od od ⌋
```

```
procedure matrixmult
        (var a : array [1..r, 1..s] of real;
             b : array [1..s, 1..t] of real;
             c : array [1..r, 1..t] of real);
  var i, j, k : integer; sum : real;
begin for i := 1 to r
      do for k := 1 to t
         do begin sum := 0;
              for j := 1 to s
              do sum := sum + a[i, j] * b[j, k];
              c[i, k] := sum
            end
end
```

Рис. 131

вообще говоря, более выгодной. Отметим также, что два внешних заголовка циклов можно переставить — тогда c будет заполняться по столбцам, а не по строкам.

3.6.2.2

В частном случае $t = 1$ матрицы b и c становятся матрицами, состоящими из одного столбца, поэтому их лучше пред-

```
proc matrixappl =
          ([1..n, 1..n] var real a,
           [1..n] var real b):
 ⌈[1..n] var real c;
  var real sum;


  for i from 1 to n
  do sum := 0;
     for j from 1 to n
     do sum := sum + a[i, j] × b[j] od;
     c[i] := sum                    od;

  for i from 1 to n
  do b[i] := c[i]        od          ⌋
```

```
procedure matrixappl
        (var a : array [1..n, 1..n] of real;
             b : array [1..n] of real);
  var c : array [1..n] of real;
      sum : real;
      i, j : integer;
begin
  for i := 1 to n
  do begin sum := 0;
           for j := 1 to n
           do sum := sum + a[i, j] * b[j];
           c[i] := sum
     end;
  for i := 1 to n
  do b[i] := c[i]
end
```

Рис. 132.

ставлять одномерными массивами. Если нужно вычислить произведение квадратной ($r = s$) матрицы a на вектор-столбец b, а результат сохранить снова в b, то b будет т р а н з и т н ы м параметром. Тогда в теле процедуры необходимо описать вспомогательный массив, в котором будет строиться результат (рис. 132[1]).

---

[1] *appl* в названии процедуры — от application (приложение) — *Прим. перев.*

3.6.2.3

Многомерные массивы легко приводят к большим затратам памяти. Квадратная матрица с $n = 2^9 = 512$ строками содер-

```
proc ft = ([1 : n] var real x) :
 ⌈[1 : n] var real c ;
   var real sum ;



   for i from 1 to n
   do sum := 0 ;
     for j from 1 to n
     do sum := sum +
              sin(pi × i × j/(n + 1)) × x[j] od ;
     c[i] := sum/sqrt((n+1)/2)              od ;

   for i from 1 to n
   do x[i] := c[i] od                         ⌋
```

```
procedure ft(var x : array [1..n] of real);
   var c : array [1..n] of real;
       sum : real;
       i,j : integer;
begin
   for i := 1 to n
   do begin sum := 0;
            for j := 1 to n
            do sum := sum +
                  sin(pi*i*j/(n+1))*x[j];
            c[i] := sum/sqrt((n+1)/2)
      end;
   for i := 1 to n
   do x[i] := c[i]
end
```

Рис. 133.

жит $2^{18} \approx 262000$ элементов. Если эти элементы достаточно просто вычисляются, как, скажем, в случае

$$a[i, j] = sin(\pi \times i \times j/(n + 1)/sqrt((n + 1)/2)$$

(„численное преобразование Фурье"), то рекомендуется вообще не заводить для них массива, а непосредственно вводить соответствующие значения в вычисления (рис. 133[1]).

3.6.3. Сведение многомерных массивов к одномерным

Многомерные массивы индексированных переменных требуют больших затрат на организацию доступа к элементам. В этом разделе мы покажем, как „избавляются" от многомерных массивов, сводя их к одномерным. Число индексированных переменных r-мерного ($r \geqslant 2$) массива a сорта

$$[\rangle m_1\langle .. \rangle n_1\langle , \rangle m_2\langle .. \rangle n_2\langle , \ldots , \rangle m_r\langle .. \rangle n_r\langle] \ \mathbf{var}\ \lambda,$$

соответственно

$$\mathbf{var\ array}\ [\rangle m_1\langle .. \rangle n_1\langle , \rangle m_2\langle .. \rangle n_2\langle , \ldots , \rangle m_r\langle .. \rangle n_r\langle]\ \mathbf{of}\ \lambda$$

равно

$$K = k_1 \times k_2 \times \ldots \times k_r,$$

где

$$k_i = n_i - m_i + 1$$

[1] *ft* — от Fourier transformation (преобразование Фурье). — *Прим. перев.*

— ***длина*** массива по i-му измерению. Столько же элементов имеет одномерный массив $\hat{a}$ сорта

$$[\,\rangle\hat{m}\langle\,..\,\rangle\hat{m}+K-1\langle\,]\ \mathbf{var}\ \lambda,$$

с произвольно выбранным первым индексом $\rangle\hat{m}\langle$, получающийся путём последовательного „вытягивания" исходного массива, а именно последовательным выстраиванием друг за другом k_1 $(r-1)$-мерных массивов сорта

$$[\,\rangle m_2\langle\,..\,\rangle n_2\langle,\ \ldots,\ \rangle m_r\langle\,..\,\rangle n_r\langle\,]\ \mathbf{var}\ \lambda,$$

каждый из которых в свою очередь получается последовательным выстраиванием k_2 $(r-2)$-мерных массивов сорта

$$[\,\rangle m_3\langle\,..\,\rangle n_3\langle,\ \ldots,\ \rangle m_r\langle\,..\,\rangle n_r\langle\,]\ \mathbf{var}\ \lambda,$$

и т. д.

При этом между отдельными элементами наших массивов устанавливается следующее соответствие:

$$a[i_1,\ i_2,\ \ldots,\ i_r] \Leftrightarrow \hat{a}[j],$$

где

$$j = k_2k_3\ \ldots\ k_r(i_1-m_1)+k_3\ \ldots\ k_r(i_2-m_2)+\ \ldots$$
$$\ldots + k_r(i_{r-1}-m_{r-1})+(i_r-m_r)+\hat{m}.$$

Если ввести вспомогательную линейную функцию f по формуле

$$f(x_1,\ x_2,\ \ldots,\ x_r) = x_1k_2k_3\ \ldots\ k_r + x_2k_3\ \ldots\ k_r + \ldots$$
$$\ldots + x_{r-1}k_r + x_r,$$

или (в виде, отвечающем схеме Горнера)

$$f(x_1,\ x_2,\ \ldots,\ x_r) = (\ldots((x_1k_2+x_2)k_3+x_3)k_4+\ \ldots$$
$$\ldots + x_{r-1})k_r + x_r,$$

то предыдущее равенство можно переписать так:

$$j = f(i_1-m_1,\ i_2-m_2,\ \ldots,\ i_r-m_r)+\hat{m}$$
$$= f(i_1,\ i_2,\ \ldots,\ i_r)+\mathrm{const},$$

где

$$\mathrm{const} = \hat{m} - f(m_1,\ m_2,\ \ldots,\ m_r).$$

Функцию f называют ***функцией выборки*** или ***функцией размещения в памяти.***

Пример. Для

[1..5,1..5] **var real** *a*, соотв. **var** *a*: **array** [1..5,1..5] **of** *real*

имеем

$$k_1 = 5,\ k_2 = 5,\ K = 25.$$

Если описание массива a заменить на

[6..30] **var real** $\hat{a}$, соотв. **var** $\hat{a}$: **array** [6..30] **of** *real*,

то формулы

$$a[i+k, i-k],\ a[i,i],\ a[i,k]+a[k,i]$$

следует заменить на

$$\hat{a}[6\times i+4\times k],\ \hat{a}[6\times i],\ \hat{a}[5\times i+k]+\hat{a}[5\times k+i].$$

Таким образом, описание многомерного массива заменяется на соответствующее описание одномерного массива, а каждая (многократно) индексированная переменная заменяется на соответствующую переменную получаемого таким образом одномерного массива, для чего, в частности, и необходимо знать длины многомерного массива по измерениям.

При этом в индексных позициях могут стоять, вообще говоря, и более сложные формулы. Однако линейные выражения в индексных позициях переходят при рассмотренной замене снова в линейные выражения.

3.6.4. Статическое распределение памяти

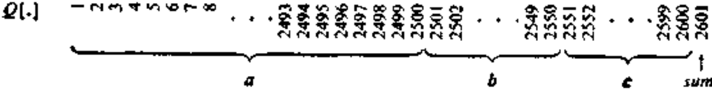
3.6.4.1

После сведéния многомерных массивов к одномерным можно сделать ещё один шаг в направлении линеаризации памяти и объединить все параметры-переменные некоторой процедуры (для простых переменных и одномерных массивов индексированных переменных) в один-единственный массив индексированных переменных, который в этом случае называют ***памятью*** процедуры. Сюда же целесообразно включить и описанные в процедуре локальные переменные и массивы — в паскале на это намекает уже и способ записи, поскольку все эти описания следуют непосредственно за заголовком процедуры.

Кроме того, обыкновенные параметры, а также локально описанные обозначения промежуточных результатов можно заменить на неизменные переменные и тоже включить в эту память.

Пример. Для процедуры *matrixappl* из 3.6.2.2 при $n = 50$ возможен следующий вариант распределения памяти с использованием массива [1..2601] **var real** Q (если не учитывать i, j):

[1..50, 1..50] **var real** *a*	↔	[1..2500] **var real** *Q*	параметры
[1..50] **var real** *b*	↔	[2501..2550] **var real** *Q*	
[1..50] **var real** *c*	↔	[2551..2600] **var real** *Q*	локальные параметры
var real *sum*	↔	[2601..2601] **var real** *Q*	



Тем самым каждой процедуре соответствует некоторый подмассив, элементы которого (если только они не отвечают чистым параметрам-результатам или локальным величинам) заполняются должным образом перед вызовом процедуры (и в ходе исполнения замещаются новыми значениями, если только это не входные или обыкновенные параметры). Правда, когда взаимодействует сразу несколько процедур, при таком простом образе действий потребуются изрядные затраты, связанные с удалением и возвратом значений в память.

Отдельные индексы введенной таким образом памяти называются (***относительными***) ***адресами***, и соответственно мы говорим об ***адресуемой памяти***.

В случае рекурсивных процедур вроде *quicksort* для каждого воплощения потребуется своё собственное место в памяти — для локально описанных промежуточных результатов, переменных, массивов, а также обыкновенных параметров. Необходимое для этого „динамическое распределение памяти", включающее в себя некоторую специальную обработку (*'call by reference'*[1]) упоминавшихся в 3.6.1 вычисляемых переменных, мы обсудим лишь в гл. 5.

Применение же описанного выше ***статического распределения памяти*** ограничивается случаем итеративных программ и массивов с постоянным диапазоном индексов. Оно часто используется для малых машин.

3.6.4.2

Более эффективный способ распределения памяти, который не только сокращает издержки на переброски в памяти, но и экономит сами элементы памяти, основан на совместном пользовании памятью из окружения вызова — он принимает в расчёт паразитический характер параметров-переменных (см. 3.5). Соответствующая машинная реализация организует при вызове процедуры передачу фактических значений параметров особым переменным процедуры. А именно, в случае обыкновенных параметров значения в форме объектов передаются обыкновенным переменным, а в случае параметров-переменных значения в форме ***ссылок***, т. е. адресов, передаются специальным переменным, называемым ***переменными связи***. Предусматриваются особые меры для того, чтобы возможным было присваивание и такой индексированной переменной, индекс которой вычисляется, т. е. переменной, которая является значением некоторой переменной связи (***косвенная адресация***, X. Шехер, 1955 г.).

---

[1] „Вызов ссылкой" (*англ.*). — *Прим. перев.*

Аналогично рассматриваются также (одномерные и многомерные) массивы с постоянным диапазоном индексов; передаваемой ссылкой служит при этом некоторый адрес-ссылка, как например $\hat{m}$ в разделе 3.6.3. Подробнее обо всём этом будет сказано в гл. 5 при обсуждении динамического распределения памяти.

' 6 4.3

При технической реализации памяти применяют преимущественно носители с возможностью обновления содержимого. С точки зрения работы с вычислительными формулярами это означает использование карандаша *и резинки* с целью сэкономить место для размещения результатов. В конце вычислений такой формуляр уже не отражает полностью весь ход проделанных вычислений, он „забывает предысторию". На ход вычислений это никак не влияет, а вот на возможности перепроверки выполненных вычислений, на возможности обозреть их с помощью формуляра, конечно, сказывается.

Отдельные программные переменные реализованной описанным способом памяти называются ***переменными памяти.***

3.7. Декомпозиция формул[1]

Формулы машинно-ориентированного языка элементарны, т. е. содержат не более одного знака операции. Таким образом, помимо обозначений вроде

17
true, соотв. *true*
"ледокол", соотв. 'ледокол'
a
xo
холост

допускаются только одноместные элементарные формулы, такие как

— 17
¬ *холост*, соотв. **not** *холост*
abs *a*, соотв. *abs*(*a*)

---

[1] Изучение этого раздела можно отложить до гл. 5.

и двуместные элементарные формулы типа

$17+4$
$a \times b$, соотв. $a * b$
"ледо" + "кол", соотв. *conc* ('ледо','кол')
a **div** 2

(в предположении что все основные операции не более чем двуместны). Более сложные формулы нужно разлагать на элементарные, вводя вспомогательные обозначения для промежуточных результатов. Запись формул при этом становится длиннее — с точки зрения читателя-человека информация становится чересчур „разжиженной". В результате получается весьма близкая к машине нотация, которая и характеризует в первую очередь облик „языков ассемблера".

3.7.1. Декомпозиция по принципу магазина

Пример. Пусть

a, b — обозначения констант вида **int**, соотв. *integer*,
x, z — переменные вида **var real**, соотв. **var** *real*.

Формулу

abs $(12 \times x - z/a) < b \times 1_{10} - 8$, соотв.
abs $(12 * x - z/a) < b * 1E - 8$,

можно разложить в последовательность иерархически опирающихся друг на друга (групповых или линеаризованных) описаний промежуточных результатов, завершающуюся искомым результатом:

⌈(**real** *h1*, **real** *h2*) ≡ (12 × *x*, *z*/*a*);
real *h3* ≡ *h1* − *h2*;
(**real** *h4*, **real** *h5*) ≡ (**abs** *h3*, *b* × 1_{10} − 8);
bool *h6* ≡ *h4* < *h5*;
h6 ⌋,

которая отражает структуру вычислительного формуляра, а также порождаемого им дерева Канторовича для этой формулы (рис. 134).

Теперь в максимальной мере может быть реализована возможность параллельного исполнения. Результат, вырабатываемый формулой, доставляется последней строчкой последовательности; таким образом, из формулы получилось предложение.

Аналогичным образом описание промежуточного результата или присваивание с формулой в правой части преобразуется в

иерархию описаний промежуточных результатов, соответственно в (иерархически построенный) оператор.

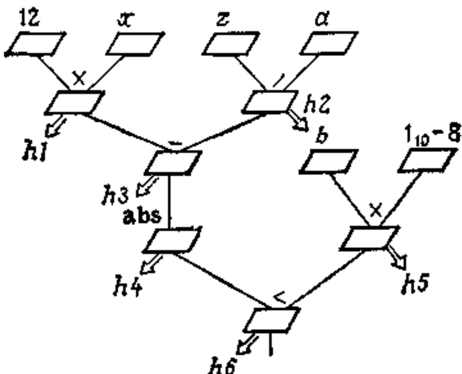


Рис. 134. Дерево Канторовича для формулы $\mathbf{abs}(12 \times x - z/a) < b \times 1_{10} - 8$.

При ориентации на машины последовательного действия получающееся предложение придётся линеаризовать, например так:

```
⌈real h1 ≡ 12×x;
 real h2 ≡ z/a;
 real h3 ≡ h1 − h2;
 real h4 ≡ abs h3;
 real h5 ≡ b×1₁₀−8;
 bool h6 ≡ h4<h5;
 h6                 ⌋
```

```
begin
h1: real = 12*x;
h2: real = z/a;
h3: real = h1−h2;
h4: real = abs(h3);
h5: real = b*1E−8;
h6: Boolean = h4<h5;
›Res‹ ⇐ h6
end
```

Эта специальная последовательность соответствует правилу вычислений „по принципу магазина", описанному в 2.3.3 для работающей последовательно машины обработки формуляров: *при движении слева направо выполнение операций откладывается лишь в случае необходимости, а отложенные операции выполняются сразу, как только это становится возможным.*

При этом приоритет „отложенных" операций выражается с помощью явной расстановки скобок или с помощью правил старшинства операций (см. 2.2.2.1).

Одну из „линейных" форм вычисления для данной формулы можно получить, записывая в одну строчку ее дерево Канторовича с сохранением его топологической структуры.

На рис. 135 изображено дерево Канторовича для приведённой выше формулы, „линейно упорядоченное“ в соответствии с принципом магазина.

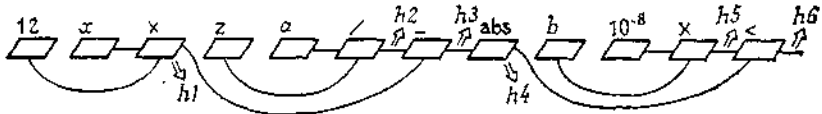


Рис. 135. Топологически упорядоченное дерево Канторовича в линейной записи.

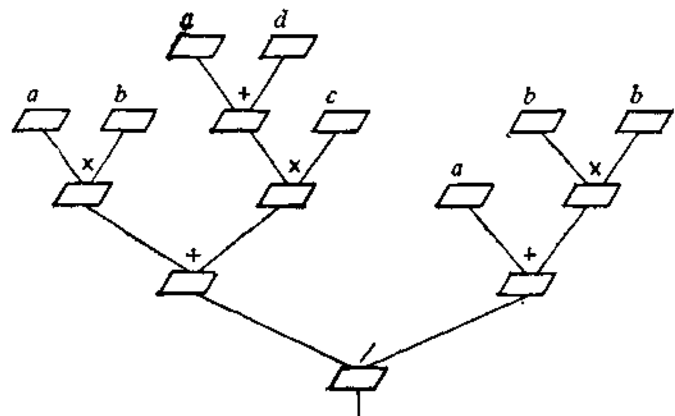


Рис. 136. Дерево Канторовича для формулы $(a \times b + (a + d) \times c)/(a + b \times b)$.

Другим примером может служить формула

$$(a \times b + (a + d) \times c)/(a + b \times b),$$

(её дерево Канторовича изображено на рис. 136), для которой по принципу магазина получается следующая линеаризация:

```
⌈real h1 ≡ a×b;
 real h2 ≡ a+d;
 real h3 ≡ h2×c;
 real h4 ≡ h1+h3;
 real h5 ≡ b×b;
 real h6 ≡ a+h5;
 real h7 ≡ h4/h6;
 h7                ⌋
```

```
begin
h1 : real = a*b;
h2 : real = a+d;
h3 : real = h2*c;
h4 : real = h1+h3;
h5 : real = b*b;
h6 : real = a+h5;
h7 : real = h4/h6;
›Res‹ ⇐ h7
end
```

Промежуточные результаты, возникающие при декомпозиции собственно формул, характеризуются тем, что они используются всего один раз [1].

3.7.2. *Использование магазина промежуточных результатов*

„Многие же будут первые последними и последние первыми."
Евангелие от Матфея, 19, 30

Использование принципа магазина для линеаризации особенно выгодно, когда экономно используются обозначения промежуточных результатов, т. е. вместо них употребляются (вспомогательные) переменные.

Такую переменную, употребленную вместо обозначения промежуточного результата, можно задействовать повторно после того, как она однажды уже была использована. Максимальное число требуемых при этом переменных в общем случае намного меньше количества имеющихся промежуточных результатов.

Для последнего из приведённых в предыдущем разделе примеров при использовании массива индексированных переменных мы получаем, что переменная $h[1]$ последовательно игра т роль $h1$, $h4$, $h7$, а переменная $h[2]$ — роль $h2$, $h3$, $h5$ и $h6$:

⌈$h[1]:=a\times b$;	**begin**
$h[2]:=a+d$;	$h[1]:=a*b$;
$h[2]:=h[2]\times c$;	$h[2]:=a+d$;
$h[1]:=h[1]+h[2]$;	$h[2]:=h[2]*c$;
$h[2]:=b\times b$;	$h[1]:=h[1]+h[2]$;
$h[2]:=a+h[2]$;	$h[2]:=b*b$;
$h[1]:=h[1]/h[2]$;	$h[2]:=a+h[2]$;
$h[1]$ ⌋	$h[1]:=h[1]/h[2]$;
	›Res‹ ⇐ $h[1]$
	end

Этот пример показывает, что при работе по принципу магазина первой доступной для употребления *всегда* становится та переменная, куда значение было помещено последним (*англ.*: 'last in — first out' [2]).

---

[1] Это справедливо лишь до тех пор, пока мы не задаёмся целью (исходя из соображений эффективности и с использованием совместности) вычислять каждую много раз встречающуюся подформулу только один раз и тем самым выходим за рамки собственно формул.

[2] Дословно: „последний в — первый из". — *Прим. перев.*

Память, организованная по такому принципу, называется ***магазинной памятью*** или просто ***магазином*** [2, 3] (*англ.*: *pushdown store*, *pushdown* [3]). Массив, который работает в таком режиме, называется ***пульсирующей памятью***. Индекс последней использованной магазинной переменной, или ***указатель верхушки*** пульсирующей памяти („конец занятой памяти"), изменяется в зависимости от того, сколько (вспомогательных) переменных встречается в правой части присваивания; а именно, он

— *увеличивается на 1, если там нет магазинных переменных;*
— *остаётся неизменным, если там ровно одна магазинная переменная;*
— *уменьшается на 1, если там две магазинные переменные* [4].

Тем самым наш последний пример можно теперь записать в следующей форме с использованием глобального указателя верхушки i:

```
⌈co i=0 co h[i+1]:=a×b;              i:=i+1;
           h[i+1]:=a+d;              i:=i+1;
           h[i]  :=h[i]×c;
           h[i-1]:=h[i-1]+h[i];      i:=i-1;
           h[i+1]:=b×b;              i:=i+1;
           h[i]  :=a+h[i];
           h[i-1]:=h[i-1]/h[i];      i:=i-1;
 co i=1 co h[i]
```

После исполнения формулы вырабатываемый ею результат находится всегда в переменной $h[1]$, т. е. справедливо равенство $i = 1$ (контроль!). В результате выполнения последующего присваивания или описания промежуточного результата магазин полностью освобождается, т. е., как и в начале, мы имеем $i = 0$.

3.7.3. Перевод в трёхадресную форму

Если уж мы используем магазин или пульсирующую память, то естественно возникает мысль не применять для выполнения арифметических операций никаких других переменных, кроме

---

[1] В оригинале Keller (см. подстрочное примечание к „принципу магазина" в разделе 2.3.3). — *Прим. изд. ред.*

[2] Термин „Keller" был введён Бауэром и Замельзоном в [западногерманской. — *Перев.*] патентной заявке от 30 марта 1957 г.

[3] Push down буквально означает „проталкивать вниз", store — склад, магазин. — *Прим. перев.*

[4] Однако если бы мы захотели сохранить совместность, то одного магазина оказалось бы недостаточно. Для каждой „ветви" совместного исполнения понадобился бы свой собственный магазин. См. также 3.7.5.

магазинных. Такое незначительное видоизменение потребует „загрузки" в магазин всех величин, входящих в формулу,— в порядке их появления слева направо. В результате получим формулировку, представленную на рис. 137 (справа — вариант

```
⌈h[1]:=a;
 h[2]:=b;
 h[1]:=h[1]×h[2];
 h[2]:=a;
 h[3]:=d;
 h[2]:=h[2]+h[3];
 h[3]:=c;
 h[2]:=h[2]×h[3];
 h[1]:=h[1]+h[2];
 h[2]:=a;
 h[3]:=b;
 h[4]:=b;
 h[3]:=h[3]×h[4];
 h[2]:=h[2]+h[3];
 h[1]:=h[1]/h[2];
 h[1]                ⌋
```

```
⌈co i=0 co h[i+1]:=a;               i:=i+1;
           h[i+1]:=b;               i:=i+1;
           h[i-1]:=h[i-1]×h[i];     i:=i-1;
           h[i+1]:=a;               i:=i+1;
           h[i+1]:=d;               i:=i+1;
           h[i-1]:=h[i-1]+h[i];     i:=i-1;
           h[i+1]:=c;               i:=i+1;
           h[i-1]:=h[i-1]×h[i];     i:=i-1;
           h[i-1]:=h[i-1]+h[i];     i:=i-1;
           h[i+1]:=a;               i:=i+1;
           h[i+1]:=b;               i:=i+1;
           h[i+1]:=b;               i:=i+1;
           h[i-1]:=h[i-1]×h[i];     i:=i-1;
           h[i-1]:=h[i-1]+h[i];     i:=i-1;
           h[i-1]:=h[i-1]/h[i];     i:=i-1;
 co i=1 co h[i]                             ⌋
```

Рис. 137.

с изменением указателя верхушки, слева — без изменения этого указателя).

Встречающиеся здесь присваивания вида

$$h[i] := h[j] \ \gamma \ h[k],$$

где γ — какая-то из двуместных основных операций, называются ***трёхадресными командами***; (относительные) адреса j, k — это ***адреса операндов***, i — ***адрес результата***.

Вместе с ***двухадресными командами*** вида

$$h[i] := \sigma \ h[j],$$

где σ — одноместная основная операция, а также ***командами засылки*** вида

$$h[i] := \rangle\text{значение}\langle$$

трёхадресные команды образуют костяк ассортимента основных команд многих машин и появляются поэтому в соответствующих языках ассемблера. Однако при работе по принципу магазина двух- и трёхадресные команды встречаются только в форме

$$h[i] := \sigma \ h[i] \quad \text{и} \quad h[i-1] := h[i-1] \ \gamma \ h[i]$$

соответственно, в которой один из адресов операндов совпадает с адресом результата, а адрес другого операнда получается как следующий адрес.

Грёхадресную форму можно получить и непосредственно из дерева Канторовича (см. рис. 136); проще всего это сделать,

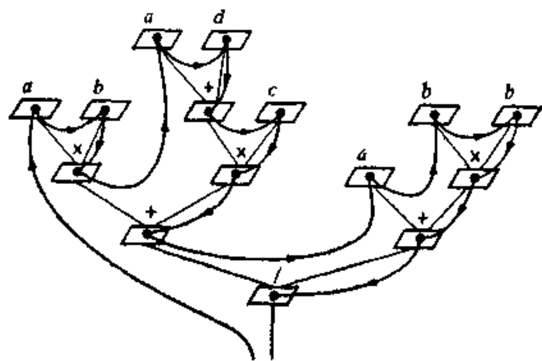


Рис. 138. Дерево Канторовича для формулы $(a \times b + (a + d) \times c)/(a + b \times b)$ с указанием порядка обхода.

записав рассматриваемую формулу в постфиксной форме (обращённая функциональная запись):

$$(((a,b)f_\times,\ ((a,d)f_+,c)f_\times)f_+,\ (a,(b,b)f_\times)f_+)f_\backslash,$$

или, без скобок [1],

$$(*)\qquad ab \times ad + c \times + abb \times + \backslash\,.$$

Если каждому знаку операнда сопоставить соответствующую команду засылки, а каждому знаку операции — соответствующую трёхадресную команду, то (бесскобочная) польская инверсная запись перейдёт в ту приведённую выше последовательность команд, которая была получена при декомпозиции формулы в соответствии с принципом магазина. Перевод в (бесскобочную) польскую инверсную запись и декомпозиция формулы в соответствии с принципом магазина оказываются эквивалентными [2].

Таким образом, алгоритмический метод перевода дерева Канторовича в польскую инверсную запись основан на следую-

---

[1] Бесскобочная префиксная запись, называемая также „варшавской нормальной формой“ или „польской записью“ (*англ.*: polish notation), была введена в школе польского логика Лукасевича около 1925 г. Приведённая здесь бесскобочная постфиксная запись называется ***польской инверсной записью*** (***ПОЛИЗ***).

[2] Ангстль и Бауэр (1950 г.). Этот принцип был реализован в вычислителе формул Stanislaus.

шем рекурсивном способе ***обхода дерева*** в так называемом ***концевом порядке*** (*англ.*: post-order):

Назовём дерево Канторовича ***атомарным***, если оно состоит из одного-единственного знака операнда. Польская инверсная запись неатомарного дерева Канторовича получается последовательным выписыванием слева направо польских инверсных записей всех его поддеревьев с последующим выписыванием знака операции, указанного в корне дерева. Польской инверсной записью атомарного дерева Канторовича является знак операнда.

В нашем примере все операции двуместны, так что дерево Канторовича — бинарное; концевой порядок отвечает такой последовательности прохождения:

левое поддерево — правое поддерево — корень.

На рис. 138 показан ход исполнения алгоритма для нашего примера.

3.7.4. Перевод в одноадресную форму

В большинстве машин для основных операций предусмотрено не три адреса, а всего лишь один; для операций, в которых один из адресов операндов совпадает с адресом результата, используется специальная переменная **AC** (называемая ***сумматором***[1]), которая и выступает в качестве соответствующего операнда, например

$$\mathbf{AC} := \mathbf{AC} + a.$$

Стандартное обозначение **AC**, как указывает уже сам выбор шрифта для него, только для этих целей и будет использоваться. При этом мы будем далее считать, что речь идёт о переменных какого-то вполне определенного сорта; строго говоря, следует различать сумматоры $\mathbf{AC}_{\mathrm{real}}$ для вычислений над вещественными числами, $\mathbf{AC}_{\mathrm{int}}$ для целочисленных вычислений и т. д.

Таким образом, ***одноадресные команды*** для двуместных операций имеют такой вид:

$$\mathbf{AC} := \mathbf{AC}\, \gamma\, h[i];$$

сюда же относятся команды для одноместных операций

$$\mathbf{AC} := \sigma\, \mathbf{AC},$$

команды засылки в сумматор

$$\mathbf{AC} := \rangle\text{значение}\langle^{2} \text{ и } \mathbf{AC} := h[i],$$

[1] Соответствующий английский термин — accumulator. Отсюда обозначение **AC**. — *Прим. перев.*

[2] В некоторых машинах нет команд типа $\mathbf{AC} := \rangle$значение$\langle$ и $\mathbf{AC} := \mathbf{AC}\,\gamma\,\rangle$значение$\langle$ с явным указанием объекта в команде.

а также специальная команда обращения к памяти, называемая „присваиванием из сумматора“,

$$h[i] := \mathbf{AC}.$$

Результат, вырабатываемый формулой, попадает теперь в **AC**. Перевод декомпозированной формы (см. 3.7.2) в одноадресную

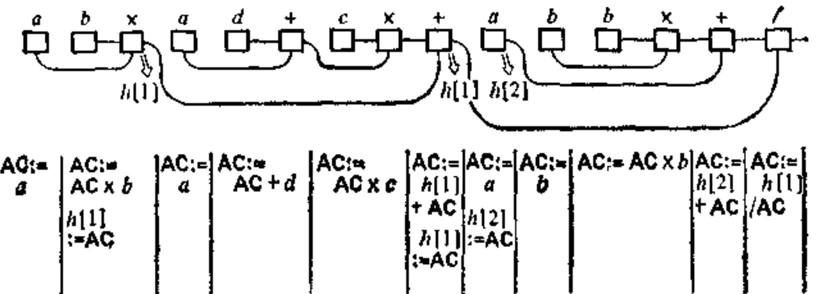


Рис. 139. Получение одноадресной формы исходя из дерева Канторовича.

можно осуществить чисто механически, заменяя каждое одиночное присваивание $a := b \gamma c$ на последовательность одноадресных команд

$$\mathbf{AC} := b;\ \ \mathbf{AC} := \mathbf{AC}\,\gamma\, c;\ \ a := \mathbf{AC}.$$

Правда, иногда при этом появляются излишние команды, которые можно вычеркнуть. Скажем, в случае примера, рассматриваемого в 3.7.2, мы получаем

(*)

⌈AC:=a;	AC:=AC×b;	h[1]:=AC;
AC:=a;	AC:=AC+d;	~~h[2]:=AC~~;
~~AC:=h[2]~~;	AC:=AC×c;	h[2]:=AC;
AC:=h[1];	AC:=AC+h[2];	h[1]:=AC;
AC:=b;	AC:=AC×b;	h[2]:=AC;
AC:=a;	AC:=AC+h[2];	h[2]:=AC;
AC:=h[1];	AC:=AC/h[2];	h[1]:=AC;
h[1]		⌋

Последовательность $n[2]$:= **AC**; **AC** := $h[2]$ является излишней, и её можно вычеркнуть. Можно также заменить на **AC** заключительную последовательность $h[1]$:= **AC**; $h[1]$.

Однако этот способ получения одноадресной формы — всё же окольный путь. Более простой способ состоит в том, чтобы записать дерево Канторовича для заданной формулы (см. рис. 136 и 137) в одну строку (рис. 139) в соответствии с принципом магазина („в топологическом порядке“), а затем „прочитать“ одноадресные операции.

В результате получим

```
        ⌈AC:=a;  AC:=AC×b;        h[1]:=AC;
         AC:=a;  AC:=AC+d;
                 AC:=AC×c;
                 AC:=h[1]+AC;  h[1]:=AC;
(**)    ┌ - - - - - - - - - - - - - - - - - -┐
        ¦AC:=a;                h[2]:=AC;    ¦
        ¦AC:=b;  AC:=AC×b;                  ¦
        ¦        AC:=h[2]+AC;               ¦
        └ - - - - - - - - - - - - - - - - - -┘
                 AC:=h[1]/AC;
         AC                              ⌋
```

Некоторое упрощение по сравнению с (*) объясняется тем, что здесь наряду с командами

$$\mathbf{AC} := \mathbf{AC}\ \gamma\ h[i]$$

употребляются также ***обращённые*** команды

$$\mathbf{AC} := h[i]\ \gamma\ \mathbf{AC}.$$

Так, фигурирующая в конце (*) последовательность команд

$$h[2] := \mathbf{AC};\ \ \mathbf{AC} := h[1];\ \ \mathbf{AC} := \mathbf{AC}/h[2]$$

заменена на

$$\mathbf{AC} := h[1]/\mathbf{AC}.$$[1]

Конечно, для коммутативных двуместных операций обращённая форма команды не нужна. Поскольку сложение коммутативно (даже для вычислений над машинными вещественными числами), последовательность команд

$$h[2] := \mathbf{AC};\ \ \mathbf{AC} := h[1];\ \ \mathbf{AC} := \mathbf{AC} + h[2]$$

в (*) можно сократить не только до

$$\mathbf{AC} := h[1] + \mathbf{AC},$$

но и до

$$\mathbf{AC} := \mathbf{AC} + h[1].$$

Обратите также внимание на заключённые в рамку пять команд в (**) для реализации вычисления $\mathbf{AC} := a + b \times b$. Им отвечает в (*) другая версия, также с пятью командами; разница связана с тем, что в одном случае употребляется $\mathbf{AC} := h[2] + \mathbf{AC}$, а в другом $\mathbf{AC} := \mathbf{AC} + h[2]$.

---

[1] Если отказаться от жёсткого порядка „слева направо", диктуемого *принципом магазина, то можно обойтись и без обращённых команд*, обращая порядок подготовки операндов для совместных операций.

3.7.5. Границы применимости принципа магазина

Принцип магазина благодаря его прозрачности и простоте технической реализации получил широкое распространение, прежде всего при построении трансляторов. Обычно он обеспечивает весьма экономное распределение памяти, хотя в отдельных случаях вспомогательные переменные используются чересчур расточительно. К числу недостатков принципа магазина следует отнести его несимметричность. Так, если обрабатывать формулу

$$f/(a+b\times b)$$

слева направо в соответствии с принципом магазина, то получится одноадресная форма

```
⌈AC:=f;                          h[1]:=AC;
 AC:=a;                          h[2]:=AC;
 AC:=b;  AC:=AC×b;
         AC:=h[2]+AC;
         AC:=h[1]/AC;
 AC                                      ⌋
```

при обработке же справа налево мы получили бы

```
⌈ AC:=b;  AC:=b×AC;
          AC:=a+AC;
          AC:=f/AC;
  AC                 ⌋
```

В современных больших вычислительных машинах не только действительно применяют специализированные по сортам сумматоры, но и стараются иметь в распоряжении побольше сумматоров вместе с независимо работающими устройствами обработки для выполнения основных операций. Мы ещё поговорим ниже о появляющихся за счёт этого возможностях параллельной работы и о трудностях их оптимального использования.

3.7.6. Обработка разбора случаев

В настоящем разделе мы рассмотрим условные формулы. Если такая формула образует правую часть некоторого присваивания, то можно перейти к условному оператору, в остальных случаях нужно ввести вспомогательную переменную; присваивание

»переменная« := **if** »условие« **then** »да-формула«
else »нет-формула« **fi**

перейдёт в оператор

if ›условие‹ **then** ›переменная‹ := ›да-формула‹
else ›переменная‹ := ›нет-формула‹ **fi**.

В паскале и без того приходится обходиться одними условными операторами [1].

Однако условный оператор можно свести к ***условным переходам***

if ›условие‹ **then goto** ›метка‹ **else skip fi**.

Для этого из оператора альтернативы вида

if ›условие‹ **then** ›да-оператор‹
else ›нет-оператор‹ **fi**; ~~

сначала вычленяются ⟩да-оператор⟨ и ⟩нет-оператор⟨: [2]

if ›условие‹ **then goto** *mj* **else goto** *mn* **fi**;
mn: ⌈›нет-оператор‹⌋; **goto** *m*;
mj: ⌈›да-оператор‹⌋; **goto** *m*;
m: ~~.

Это можно сократить до формулировки с „обходом" нет-оператора:

if ›условие‹ **then goto** *mj* **else skip fi**;
⌈›нет-оператор‹⌋; **goto** *m*;
mj: ⌈›да-оператор‹⌋;
m: ~~

Если ⟩да-оператор⟨ пуст, т. е.

if ›условие‹ **then skip else** ›нет-оператор‹ **fi**,

то метки *m* и *mj* отождествляются:

if ›условие‹ **then goto** *m* **else skip fi**;
⌈›нет-оператор‹⌋;
m: ~~

Если же пуст ⟩нет-оператор⟨, т. е.

if ›условие‹ **then** ›да-оператор‹ **else skip fi**; ~~,

---

[1] Поскольку паскалевские формулировки в этом разделе лишь тривиальным образом отличаются от алгольных, мы не стали их здесь явно приводить.

[2] Ниже *j* и *n* — от немецких ja (да) и nein (нет). — *Прим. перев*

то лучше поменять ⟩да-оператор⟨ и ⟩нет-оператор⟨ ролями; в результате получится формулировка без „перепрыгивания" через ⟩да-оператор⟨:

if ¬ ⟩условие⟨ **then goto** *m* **else skip fi**;
⌈ ⟩да-оператор⟨ ⌋;
m: ~

Итак, в результате получаются условные переходы, в которых ⟩условие⟨ представляет собой формулу, вырабатывающую некоторое значение истинности. Если результат попадает в некоторый сумматор $\mathbf{AC}_{\mathbf{bool}}$, то нужно проверить каждый из случаев $\mathbf{AC}_{\mathbf{bool}} = \mathsf{T}$ и $\mathbf{AC} = \mathsf{F}$. Перестановка ветвей в операторе альтернативы позволяет сводить один из этих случаев к другому. Таким образом, можно обойтись ***командой условного перехода***

if $\mathbf{AC}_{\mathbf{bool}}$ **then goto** ⟩метка⟨ **else skip fi.**

Если же последняя операция при вычислении условия — это одна из операций сравнения

$$.=.,\ .\neq.,\ .\leqslant.,\ .>.,\ .<.,\ .\geqslant.$$

для целых или вещественных чисел, то при помощи вычитания её можно свести к одному из шести случаев:

$$\begin{array}{ll} \mathbf{AC} = 0, & \mathbf{AC} \neq 0, \\ \mathbf{AC} \geqslant 0, & \mathbf{AC} < 0, \\ \mathbf{AC} > 0, & \mathbf{AC} \leqslant 0. \end{array}$$

Возможность перестановки операндов вычитания делает операции из нижней строчки ненужными.

Перестановка ветвей в операторе альтернативы сводится к отрицанию и делает излишними операции из правого столбика; в принципе для целых и вещественных чисел можно обойтись следующими двумя формами условного перехода по содержимому сумматора:

if $\mathbf{AC} = 0$ **then goto** ⟩метка⟨ **else skip fi**

и

if $\mathbf{AC} \geqslant 0$ **then goto** ⟩метка⟨ **else skip fi.**

Пример (ср. с примером (с) из 3.5.2).

```
proc ord ≡ (var int u, v):
  if u ≧ v then skip
  ▯ u ≦ v then exch(u, v) fi
```

Представляется разумным сузить второе условие-страж до $u < v$; при помощи вычитания получим

```
proc ord ≡ (var int u, v):
 ⌈AC:= u;
  AC:= AC − v;
  if AC ≧ 0 then skip
          else exch(u, v) fi ⌋
```

а отсюда

```
proc ord ≡ (var int u, v):
 ⌈AC:= u;
  AC:= AC − v;
  if AC ≧ 0 then goto m else skip fi;
  exch(u, v);
  m: skip                           ⌋
```

Аналогичную постепенную обработку допускает и последовательный разбор случаев. То же самое справедливо для охраняемого разбора случаев. Скажем,

if $x > 0$ **then** $x := 1$ ▯ $x = 0$ **then skip** ▯ $x < 0$ **then** $x := -1$ **fi**,

применяя линеаризацию, можно преобразовать в

if $x \geqslant 0$ **then if** $x = 0$ **then skip else** $x := 1$ **fi else** $x := -1$ **fi**

откуда получаем

```
⌈    AC:= x;
     if AC ≧ 0 then goto mj else skip fi;
     x:= −1; goto m;
mj:  if AC = 0 then goto m else skip fi;
     x:= 1;
 m : skip
```

Здесь неявно подразумевается, что (условный или безусловный) переход никогда не меняет содержимого сумматора.

3.7.7. Исключение логических операций

В 2.2.3.4 уже говорилось о том, что конъюнкцию $.\wedge.$ (соотв. **.and.**) и дизъюнкцию $.\vee.$ (соотв. **.or.**) можно заменять подходящим разбором случаев, причём иногда это приводит к упрощениям, поскольку при определённых условиях вычисления второго операнда проводить не обязательно (последовательная конъюнкция, последовательная дизъюнкция). Предпосылкой для этого является, разумеется, отсутствие побочных эффектов, на что в стандартном алголе-68 следует обращать особое внимание.

Получающиеся таким образом операторы с разбором случаев можно затем снова свести к условным переходам.

Пример.

$x := $ **if** $x \geqslant 0 \wedge x < 1$ **then** $sqrt((1-x) \times x)$ **else** 0 **fi**; ~

Прежде всего, условие $x \geqslant 0 \wedge x < 1$ можно заменить на

if $x < 1$ **then** $x \geqslant 0$ **else false fi**.

аёт

i (**if** $x < 1$ **then** $x \geqslant 0$ **else false fi**) **then** $x := sqrt((1-x) \times x)$
else $x := 0$ **fi**; ~,

что можно записать подробно так:

```
if (if x<1
   then ┌if x≥0─────────────┐
        │      then true    │
        └──────else false fi┘
   else false               fi) then x:=sqrt((1-x)×x)
                                  else x:=0 fi ; ~~,
```

Отсюда получается следующее упрощение:

```
if x<1
   then if x≥0 then x:=sqrt((1-x)×x)
               else x:=0              fi
   else x:=0                            fi ; ~~ .
```

От двойного написания присваивания $x := 0$ мы избавимся за счёт выделения обоих вхождений этого присваивания:

```
if x<1 then if x≥0 then goto mj
                   else goto mn fi
       else goto mn                fi ;
mn : x:=0 ; goto m ;
mj : x:=sqrt((1-x)×x) ; goto m ;
m  : ~~
```

Тем самым получим окончательно

```
AC:=x ; AC:=AC-1 ;
if AC≥0 then goto mn else skip  fi ;
AC:=x ;
if AC≥0 then goto mj else skip  fi ; goto mn ;
mn : AC:=0 ; x:=AC ; goto m ;
mj : AC:=x ;
     AC:=1-AC ;
     AC:=AC×x ;
     AC:=sqrt(AC) ; x:=AC ; goto m ;
m  : ~~
```

Глава 4

Двоичные комбинационные и переключательные схемы

В предыдущей главе были введены машинно-ориентированные понятия и методы, которые важны для исследования автоматизированных процессов вычисления. Лежащие в их основе вычислительные структуры не принимались пока во внимание. В этой главе мы учтём ещё одну характерную черту современных машин — сведение всех вычислительных структур к двоичным словам и алгоритмам их обработки. При этом будет дано представление о технической реализации процессов вычисления. Предварительно мы изложим некоторые основные факты, касающиеся абстрактной булевой алгебры.

4.1 *. Булева алгебра

4.1.1. Абстрактное определение булевой алгебры

Для введенной в 2.1.3.6 и кратко обсуждавшейся там вычислительной структуры $\mathbb{B}_2$ значений истинности выполняются перечисленные в табл. 9 законы, однако эти законы никоим образом не определяют однозначно (с точностью до изоморфизма) некоторую модель.

Множество элементов с заданными на нём двуместными операциями $.\wedge.$ и $.\vee.$ (***конъюнкцией*** и ***дизъюнкцией***), удовлетворяющими законам коммутативности, ассоциативности, идемпотентности и поглощения, называется ***структурой***, а если выполняется ещё и закон дистрибутивности, то ***дистрибутивной структурой***. В случае когда к указанным выше операциям добавляется ещё одна одноместная инволютивная операция $\neg.$ (***отрицание***), причём удовлетворяются законы де Моргана и законы нейтральности, говорят о ***булевой структуре*** или ***булевой алгебре***.

4.1.1.1

Из перечисленных законов можно вывести, что для произвольных f, g справедливы равенства $f \wedge \neg f = g \wedge \neg g$ и

* Изучение этого раздела можно начать сразу же вслед за 2.2.

$f \vee \neg f = g \vee \neg g$. Например, в силу законов нейтральности и коммутативности имеем

$$f \wedge \neg f = (f \wedge \neg f) \vee (g \wedge \neg g) = (g \wedge \neg g) \vee (f \wedge \neg f) = g \wedge \neg g.$$

Если обозначить $f \wedge \neg f$ через O и $g \vee \neg g$ через L, то выполняются равенства

$$(*) \qquad \begin{array}{ll} \neg \mathsf{L} = \mathsf{O}, & \neg \mathsf{O} = \mathsf{L}, \\ f \wedge \mathsf{L} = f, & f \wedge \mathsf{O} = \mathsf{O}, \\ f \vee \mathsf{L} = \mathsf{L}, & f \vee \mathsf{O} = f. \end{array}$$

Булева алгебра называется ***вырожденной***, если O и L совпадают; в таком случае ввиду равенств $f = f \wedge \mathsf{L} = f \wedge \mathsf{O} = \mathsf{O}$ она не содержит никаких других элементов, а значит состоит ровно из одного элемента. Всякая невырожденная булева алгебра — а только такие и будут рассматриваться в дальнейшем — содержит два ***нейтральных элемента***: O (***нулевой элемент***) и L (***единичный элемент***).

Из $f \wedge g = \mathsf{L}$ следует, что $f = g = \mathsf{L}$. Действительно,

$$f = f \vee (f \wedge g) = f \vee \mathsf{L} = \mathsf{L}.$$

Этот факт называют ***неразложимостью*** нейтрального элемента L.

4.1.1.2

Таким образом, всякая невырожденная двухэлементная булева алгебра состоит из нулевого и единичного элементов.

∧	L	O
L	L	O
O	O	O

∨	L	O
L	L	L
O	L	O

	L	O
¬	O	L

Рис. 140. Таблицы значений для базовых операций в $\mathbb{BIT}$.

Одной из её моделей служит ***двоичная модель*** $\mathbb{BIT}$ с двумя „выделенными элементами“ $\{\mathsf{O}, \mathsf{L}\}$ (см. 2.1.3.6) и таблицами значений, которые получаются из (*) (рис. 140).

Изоморфной двухэлементной моделью является ***модель исчисления высказываний*** — вычислительная структура $\mathbb{B}_2$ значений истинности (см. рис. 44), изоморфизм осуществляется соответствием

$$(1) \qquad \mathsf{F} \triangleq \mathsf{O}, \quad \mathsf{T} \triangleq \mathsf{L}.$$

Множество всех подмножеств произвольного (непустого) множества G с операциями пересечения, объединения и дополнения в роли $.\wedge.$, $.\vee.$ и $\neg$. также образует (невырожденную) модель $\mathfrak{P}(G)$, в которой единичным элементом служит само множество G, а нулевым — пустое множество. Модель $\mathbb{B}_2$ изоморфна модели $\mathfrak{P}(G)$, где G — одноэлементное множество. Согласно знаменитой теореме Стоуна (1934 г.), каждая конечная модель булевой алгебры изоморфна модели $\mathfrak{P}(G)$ для некоторого конечного множества G и, следовательно, содержит 2^n $(n \in \mathbb{N})$ элементов; конечные модели изоморфны, если они имеют одинаковое количество элементов.

4.1.1.3

Кроме конъюнкции и дизъюнкции особенно важны с точки зрения технической реализации переключательных функций следующие коммутативные операции (приведённые в инфиксной записи):

$a_1 \overline{\vee} a_2 =_{\text{def}} \neg a_1 \wedge \neg a_2$ (***функция Пирса***, или (см. ниже) „операция NOR“ [1]),

$a_1 \overline{\wedge} a_2 =_{\text{def}} (\neg a_1 \wedge a_2) \vee (a_1 \wedge \neg a_2) \vee (\neg a_1 \wedge \neg a_2)$ (***штрих Шеффера***[2], или (см. ниже) „операция NAND“[3]).

Ещё чаще встречаются некоммутативные операции (снова в инфиксной записи)

$a_1 \rightarrow a_2 =_{\text{def}} \neg a_1 \vee a_2$ (***субъюнкция***, или „импликация“)[4],

$a_1 \setminus a_2 =_{\text{def}} a_1 \wedge \neg a_2$ (***разность***[5]).

В частности, для произвольного f

$$O \rightarrow f = L, \qquad L \rightarrow f = f,$$
$$f \rightarrow L = L, \qquad f \rightarrow O = \neg f.$$

---

[1] От "not or" [„не или“ (*англ.*). — *Изд. ред.*].

[2] Название объясняется тем, что для обозначения этой операции обычно используется вертикальная черта. — *Прим. изд. ред.*

[3] От "not and" [„не и“ (*англ.*). — *Изд. ред.*].

[4] Смысл этого названия станет понятен в 4.1.3.

[5] В случае модели множества всех подмножеств говорят о „теоретико-множественной разности“.

Коммутативны и ассоциативны следующие две операции (в инфиксной записи):

$a_1 \leftrightarrow a_2 =_{\text{def}} (a_1 \wedge a_2) \vee (\neg a_1 \wedge \neg a_2)$ (***бисубъюнкция***, или „функция эквивалентности“ [1]),

$a_1 \leftarrow\!\mid\!\rightarrow a_2 =_{\text{def}} (a_1 \wedge \neg a_2) \vee (\neg a_1 \wedge a_2)$ (***симметрическая разность***, или „функция неэквивалентности“ [1,2]).

В частности, для любого f

$$O \leftrightarrow f = \neg f, \quad L \leftrightarrow f = f,$$

а также

$$O \leftarrow\!\mid\!\rightarrow O = O, \qquad O \leftarrow\!\mid\!\rightarrow L = L,$$
$$L \leftarrow\!\mid\!\rightarrow O = L, \qquad L \leftarrow\!\mid\!\rightarrow L = O.$$

Если O отождествить с 0, а L с 1, то $.\leftarrow\!\mid\!\rightarrow.$ обозначает операцию *сложения по модулю* 2.

4.1.1.4

На основе законов булевой алгебры можно доказывать утверждения о тождественности (эквивалентности) выражений с булевыми операциями. Так, используя законы де Моргана и закон инволюции, получим

$$a_1 \overline{\vee} a_2 = \neg (a_1 \vee a_2),$$
$$a_1 \setminus a_2 = \neg (a_1 \rightarrow a_2),$$
$$a_1 \leftarrow\!\mid\!\rightarrow a_2 = \neg (a_1 \leftrightarrow a_2).$$

Несколько труднее доказывается следующее соотношение для эквивалентности:

$$(\neg a_1 \vee a_2) \wedge (a_1 \vee \neg a_2) = (a_1 \wedge a_2) \vee (\neg a_1 \wedge \neg a_2).$$

Более общим образом, с помощью законов дистрибутивности и идемпотентности можно показать, что

$$(\neg a_1 \vee a_2) \wedge (a_1 \vee a_3) = (a_1 \wedge a_2) \vee (\neg a_1 \wedge a_3).$$

---

[1] Смысл этого названия станет понятен в 4.1.3.

[2] Эту операцию называют также *исключающим ИЛИ* (*лат.*: *aut*) в противоположность дизъюнкции, называемой *неисключающим ИЛИ* (*лат.*: *vel*); в патентной литературе неисключающее ИЛИ записывают как „и/или“.

Выражения с булевыми операциями (***булевы выражения***) часто удается значительно упростить. Пример:

$$\begin{aligned}(a \wedge b) \vee (\neg a \wedge b) \vee (a \wedge \neg b) &= \\ ((a \wedge b) \vee (\neg a \wedge b)) \vee (a \wedge \neg b) &= \\ ((a \vee \neg a) \wedge b) \vee (a \wedge \neg b) &= \\ (\mathsf{L} \wedge b) \vee (a \wedge \neg b) &= \\ b \vee (a \wedge \neg b) &= \\ (b \vee a) \wedge (b \vee \neg b) &= \\ (b \vee a) \wedge \mathsf{L} &= \\ b \vee a&.\end{aligned}$$

Отсюда, в частности, следует, что

$$a_1 \,\overline{\wedge}\, a_2 = \neg a_2 \vee \neg a_1,$$

или

$$a_1 \,\overline{\wedge}\, a_2 = \neg (a_1 \wedge a_2).$$

Наконец, закон идемпотентности даёт

$$a \,\overline{\vee}\, a = \neg a,$$

так что

$$(a \,\overline{\vee}\, a) \,\overline{\vee}\, (b \,\overline{\vee}\, b) = a \wedge b,$$

$$(a \,\overline{\vee}\, b) \,\overline{\vee}\, (a \,\overline{\vee}\, b) = a \vee b.$$

Таким образом, все булевы выражения можно записать с помощью одной только операции NOR (или аналогично с помощью одной операции NAND); ср. с 4.1.6.

4.1.1.5

Вместе с каждой моделью $\mathscr{M}$ булевой алгебры такую модель образуют и n-местные ***булевы функции*** $\mathscr{M} \times \mathscr{M} \times \ldots \times \mathscr{M} \to \mathscr{M}$ *с поточечно определёнными операциями* $.\wedge.$, $.\vee.$, $.\neg.$. А именно, пусть

$$f_\mu : (x_1, x_2, \ldots, x_n) \longmapsto f_\mu(x_1, x_2, \ldots, x_n), \quad \mu = 1, 2.$$

Тогда по определению

$$\begin{aligned}f_1 \wedge f_2 &: (x_1, x_2, \ldots, x_n) \longmapsto f_1(x_1, x_2, \ldots, x_n) \wedge f_2(x_1, x_2, \ldots, x_n), \\ f_1 \vee f_2 &: (x_1, x_2, \ldots, x_n) \longmapsto f_1(x_1, x_2, \ldots, x_n) \vee f_2(x_1, x_2, \ldots, x_n), \\ \neg f_1 &: (x_1, x_2, \ldots, x_n) \longmapsto \neg f_1(x_1, x_2, \ldots, x_n).\end{aligned}$$

Две *постоянные* функции

$$\mathrm{o} : (x_1, x_2, \ldots, x_n) \longmapsto \mathsf{O},$$

$$\mathrm{i} : (x_1, x_2, \ldots, x_n) \longmapsto \mathsf{L}$$

являются соответственно нулевым и единичным элементами в этой модели.

Если модель $\mathcal{M}$ двухэлементна (как, например, $\mathbb{B}$IT), то описанные выше функции называются ***двоичными функциями***. Имея в виду естественную интерпретацию L $\widehat{=}$ »включено«, O $\widehat{=}$ »выключено«, говорят также о (двоичных) ***переключательных функциях***, а в случае модели $\mathbb{B}_2$ — о ***функциях истинности*** или о ***функциях*** (операциях) ***логики высказываний***.

4.1.2. Теорема о булевой нормальной форме

В модели (двоичных) переключательных функций одной переменной, т. е. одноместных двоичных функций, лишь четыре различные функции, а именно ***тождественная*** функция

$$\mathrm{id}: x_1 \longmapsto x_1,$$

инволютивная функция ***отрицания***

$$\mathrm{neg}: x_1 \longmapsto \neg x_1$$

и две постоянные функции $o: x_1 \longmapsto \mathrm{O}$ и $\mathrm{i}: x_1 \longmapsto \mathrm{L}$, которые являются нулевым и единичным элементами этой четырёхэлементной модели. Заметим, что, например,

$$\neg\,\mathrm{id} = \mathrm{neg}, \quad \mathrm{id} \wedge \mathrm{neg} = o.$$

В модели (двоичных) переключательных функций от двух переменных, т. е. двуместных двоичных функций, уже шестнадцать различных функций. В четыре поля таблицы значений записываются в произвольной комбинации O или L, а таких комбинаций 2^4. В общем случае имеется $2^{(2^n)}$ различных n-местных двоичных функций, которые образуют модель с $2^{(2^n)}$ элементами.

Естественно возникает вопрос, нельзя ли свести все переключательные функции к какому-нибудь меньшему числу „базисных" переключательных функций. Это действительно возможно сделать, например, можно свести всё к (одноместной) функции отрицания и двум двуместным переключательным функциям, а именно ***конъюнкции*** („операции И")

$$\mathrm{conjunct}: (x_1, x_2) \rightarrow x_1 \wedge x_2$$

и ***дизъюнкции*** („операции ИЛИ")

$$\mathrm{disjunct}: (x_1, x_2) \rightarrow x_1 \vee x_2.$$

Сначала докажем следующую лемму:

Лемма. Для всякой n-местной переключательной функции f выполняется соотношение

$$f(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1} \ldots, a_n) = \\ (a_i \wedge f(a_1, a_2, \ldots, a_{i-1}, \mathsf{L}, a_{i+1}, \ldots, a_n)) \\ \vee (\neg a_i \wedge f(a_1, a_2, \ldots, a_{i-1}, \mathsf{O}, a_{i+1}, \ldots, a_n)).$$

Доказательство. Рассмотрим отдельно два случая.

1. Пусть $a_i = \mathsf{L}$. Тогда $\neg a_i = \mathsf{O}$. Правая часть доказываемого соотношения равна

$$(\mathsf{L} \wedge f(a_1, a_2, \ldots, a_{i-1}, \mathsf{L}, a_{i+1}, \ldots a_n)) \\ \vee (\mathsf{O} \wedge f(a_1, a_2, \ldots, a_{i-1}, \mathsf{O}, a_{i+1}, \ldots, a_n)).$$

В соответствии с таблицей значений конъюнкции первый член имеет значение $f(a_1, a_2, \ldots, a_{i-1}, \mathsf{L}, a_{i+1}, \ldots, a_n)$ а второй — значение O. Следовательно, согласно таблице значений дизъюнкции, правая часть равна $f(a_1, a_2, \ldots, a_{i-1}, \mathsf{L}, a_{i+1}, \ldots, a_n)$. Но то же самое значение имеет и левая часть.

2. Пусть $a_i = \mathsf{O}$. Совершенно аналогично получаем, что правая часть равна

$$f(a_1, a_2, \ldots, \mathsf{O}, a_{i+1}, \ldots, a_n).$$

Эта лемма позволяет „выносить" переменную a_i за знак переключательной функции. Последовательным применением леммы к $a_1, a_2, \ldots, a_n$ устанавливается

Теорема о булевой нормальной форме. Каждую переключательную функцию можно однозначно представить в следующей ***(дизъюнктивной) нормальной форме***:

$$f(a_1, a_2, \ldots a_n) = (a_1 \wedge a_2 \wedge \ldots \wedge a_{n-1} \wedge a_n \wedge f(\mathsf{L}, \mathsf{L}, \ldots \mathsf{L}, \mathsf{L})) \\ \vee (\neg a_1 \wedge a_2 \wedge \ldots \wedge a_{n-1} \wedge a_n \wedge f(\mathsf{O}, \mathsf{L}, \ldots \mathsf{L}, \mathsf{L})) \\ \vee (a_1 \wedge \neg a_2 \wedge \ldots \wedge a_{n-1} \wedge a_n \wedge f(\mathsf{L}, \mathsf{O}, \ldots \mathsf{L}, \mathsf{L})) \\ \vdots \\ \vee (\neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_{n-1} \wedge a_n \wedge f(\mathsf{O}, \mathsf{O}, \ldots \mathsf{O}, \mathsf{L})) \\ \vee (\neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_{n-1} \wedge \neg a_n \wedge f(\mathsf{O}, \mathsf{O}, \ldots \mathsf{O}, \mathsf{O})).$$

Если $f(a_1, a_2, \ldots, a_n) = \mathsf{O}$, то соответствующий член, разумеется, выпадает из представления. Таким образом, всякая переключательная функция представима в виде дизъюнкции $\varkappa$, $0 \leqslant \varkappa \leqslant 2^n$, членов — так называемых ***совершенных конъюнкций***, каждая совершенная конъюнкция — это n-местная конъюнкция, у которой все аргументы — либо сами переменные, либо их отрицания.

Пример. Переключательную функцию f с таблицей значений

a_1	L	O	L	O	L	O	L	O
a_2	L	L	O	O	L	L	O	O
a_3	L	L	L	L	O	O	O	O
$f(a_1, a_2, a_3)$	L	O	O	L	O	L	L	O

можно представить в виде

$$f(a_1, a_2, a_3) = (a_1 \wedge a_2 \wedge a_3) \vee (\neg a_1 \wedge \neg a_2 \wedge a_3) \vee (\neg a_1 \wedge a_2 \wedge \neg a_3) \vee (a_1 \wedge \neg a_2 \wedge \neg a_3).$$

В частности, все шестнадцать различных двуместных двоичных функций сводятся к отрицанию, конъюнкции и дизъюнкции. При этом мы получаем:

a) одну двоичную функцию без совершенных конъюнкций и одну со всеми четырьмя совершенными конъюнкциями; это постоянные функции O и L соответственно;

b) четыре двоичные функции, содержащие по одной совершенной конъюнкции, и четыре, содержащие по три;

c) шесть двоичных функций, содержащих по паре совершенных конъюнкций.

Из функций, попадающих в разряд (b), имеют по одной совершенной конъюнкции:

конъюнкция, функция Пирса и две функции взятия разности (с прямым и обратным порядком аргументов);

по три совершенные конъюнкции:

дизъюнкция, штрих Шеффера и две импликации (с прямым и обратным порядком аргументов).

В разряде c) нетривиальны лишь две функции:

эквивалентность и симметрическая разность.

(Дизъюнктивная) нормальная форма может рассматриваться как представитель всех эквивалентных двоичных функций.

То что для двоичных функций имеется нормальная форма, означает, что вопрос о тождественности двух двоичных выражений можно решить, приведя их обоих к нормальной форме. В конечном счёте дело сводится к перебору всех комбинаций значений аргументов. Некоторые из доказанных в 4.1.1 общих тождеств в булевой алгебре для случая двоичных функций могут быть непосредственно доказаны таким образом.

Это, однако, ни в коей мере не означает, что все доказательства равенства переключательных функций надо проводить путем такого перебора. Для переключательных функций многих переменных этот путь был бы и практически неприемлем. Напротив, многие доказательства лучше проводить, „манипулируя" последовательностями символов.

4.1.3. Отношение порядка в булевой алгебре. Импликация

4.1.3.1. Отношение „сильнее"

В булевой алгебре можно ввести двухместное отношение $\geqslant$, определяемое следующим образом:

$$f \geqslant g \quad (\text{„}f \text{ сильнее, чем } g\text{"})^{1},$$

если $f = f \wedge g$.

Из закона поглощения следует, что

$$f = f \wedge g \text{ эквивалентно } f \vee g = g^{2}.$$

Теорема 1. Отношение $\geqslant$ рефлексивно, транзитивно и антисимметрично (т. е. является ***отношением порядка***):

$$f \geqslant f,$$

$$\text{если } f \geqslant g \text{ и } g \geqslant h, \text{ то } f \geqslant h,$$

$$\text{если } f \geqslant g \text{ и } g \geqslant f, \text{ то } f = g.$$

Доказательство. Согласно закону идемпотентности, $f \wedge f = f$, поэтому $f \geqslant f$. Из $f \wedge g = f$ и $g \wedge h = g$ следует, что

$$f \wedge h = (f \wedge g) \wedge h = f \wedge (g \wedge h) = f \wedge g = f.$$

Наконец, из $f \wedge g = f$ и $g \wedge f = g$ следует, что $f = g$.

Теорема 2. Для каждого элемента f

$$\mathsf{O} \geqslant f \geqslant \mathsf{L}.$$

Доказательство. Имеем $\mathsf{O} = \mathsf{O} \wedge f$ и $f = f \wedge \mathsf{L}$.

Теорема 3. Соотношение $f \geqslant g$ справедливо тогда и только тогда, когда

$$f \rightarrow g = \mathsf{L}.$$

Доказательство. Пусть $f \geqslant g$, т. е. $f = f \wedge g$. Тогда $f \wedge \wedge \neg g = f \wedge (g \wedge \neg g) = \mathsf{O}$, а значит $\neg f \vee g = \mathsf{L}$, т. е. $f \rightarrow$

[1] Для модели с множеством всех подмножеств $a \geqslant b$ означает „a является подмножеством b". Выбор слова „сильнее" получит оправдание при последующем применении к высказываниям.

[2] Так что $a \geqslant b$ соответствует $a \leqslant b = \text{true}$, в смысле табл. 11!

$\to g = \mathsf{L}$. Обратно, пусть $\neg f \vee g = \mathsf{L}$. Тогда

$$g = (f \wedge \neg f) \vee g = (f \vee g) \wedge (\neg f \vee g) = (f \vee g) \wedge \mathsf{L} = f \vee g.$$

Теорема 4. Соотношение $f = g$ справедливо тогда и только тогда, когда

$$f \leftrightarrow g = \mathsf{L}.$$

Доказательство. Имеем

$$f \leftrightarrow f = (\neg f \vee f) \wedge (\neg f \vee f) = \mathsf{L} \wedge \mathsf{L} = \mathsf{L}.$$

Обратно, пусть $(\neg f \vee g) \wedge (f \vee \neg g) = \mathsf{L}$. Ввиду неразложимости L, $\neg f \vee g = \mathsf{L}$ и $f \vee \neg g = \mathsf{L}$, а значит $f \geqslant g$ и $g \geqslant f$.

4.1.3.2. Отношение „сильнее" на булевых функциях

Так как булевы функции

$$\mathscr{M} \times \mathscr{M} \times \ldots \times \mathscr{M} \to \mathscr{M}$$

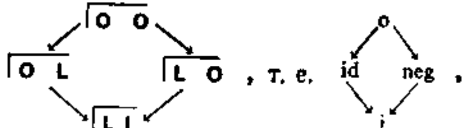
над моделью $\mathscr{M}$ сами образуют булеву алгебру, то и для них тоже определено отношение порядка „сильнее". В смысле этого отношения порядка все они лежат между двумя постоянными функциями i и o. Для булевых функций f, g

$$f \geqslant g \text{ (соотв. } f = g)$$

означает просто, что для любой комбинации значений аргументов $(x_1, x_2, \ldots, x_n)$

$$f(x_1, x_2, \ldots, x_n) \geqslant g(x_1, x_2, \ldots, x_n)$$
$$(\text{соотв. } f(x_1, x_2, \ldots, x_n) = g(x_1, x_2, \ldots, x_n)).$$

В частности, сравнение двоичных функций сводится к поэлементному сравнению соответствующих таблиц значений. Для четырёх одноместных двоичных функций диаграмма сравнения выглядит следующим образом:



Функции id и neg между собой несравнимы.

Если для всех комбинаций $(x_1, x_2, \ldots, x_n)$

$$f(x_1, x_2, \ldots, x_n) = \mathsf{L},$$

то $f = \mathrm{i}$. В таком случае выражение, определяющее f, называется ***тождественно истинным*** (или ***общезначимым***, или ***тавтологией***).

4.1.3.3. Отношение порядка между булевыми выражениями

Сравнивать между собой булевы выражения особенно удобно при помощи теоремы 3. Так,

$$f \wedge g \geqslant f,$$

потому что $f \wedge g \to f = (\neg f \vee \neg g) \vee f = (\neg f \vee f) \vee \neg g = \mathsf{L} \vee \neg g = \mathsf{L}$.
Аналогично

$$f \geqslant f \vee g.$$

Далее,

$$(f \wedge (f \to g)) \geqslant g,$$

так как

$$(f \wedge (f \to g)) \to g = \neg (f \wedge (\neg f \vee g)) \vee g = \neg ((f \wedge \neg f) \vee (f \wedge g)) \vee g$$
$$= \neg (f \wedge g) \vee g = \neg f \vee \neg g \vee g = \neg f \vee \mathsf{L} = \mathsf{L}.$$

Часто из этих отношений можно простым способом вывести дальнейшие заключения. Допустим, что $f \wedge (f \to g) = \mathsf{L}$. Тогда $\mathsf{L} \geqslant g$. Вместе с $g \geqslant \mathsf{L}$ (теорема 2) это даёт $g = \mathsf{L}$. Тем самым доказана

Теорема 5. Если $f \wedge (f \to g) = \mathsf{L}$, то $g = \mathsf{L}$.

Пусть теперь $f \wedge g = \mathsf{L}$. Тогда в силу соотношений $\mathsf{L} \geqslant f \wedge g$ и $f \wedge g \geqslant f$ имеем $\mathsf{L} \geqslant f$. Аналогично заключаем, что $\mathsf{L} \geqslant g$, и тем самым получена

Теорема 6. Если $f \wedge g = \mathsf{L}$, то $f = \mathsf{L}$ и $g = \mathsf{L}$ („неразложимость нейтрального элемента операции .$\wedge$.“ (ср. с 4.1.1.1)).

Далее, из эквивалентности (равнозначности) двух импликаций (субъюнкций) следует эквивалентность двух отвечающих им соотношений порядка Эквивалентны, например, выражения $p \to (q \to r)$ и $(p \wedge q) \to r$, которые оба можно преобразовать к $\neg (p \wedge q \wedge \neg r)$. Следовательно, справедлива

Теорема 7. Соотношение $p \geqslant (q \to r)$ выполняется тогда и только тогда, когда $p \wedge q \geqslant r$.

4.1.3.4. Приложения к высказываниям и предикатам

Для $\mathscr{M} = \mathbb{B}_2$ (модель исчисления высказываний) вместо «высказывание f сильнее, чем высказывание g» говорят также «f влечёт (за собой) g». Поскольку в этом случае $f \to g = \mathsf{i}$, операцию .$\longrightarrow$. тоже называют *импликацией*. Точно также операцию .$\longleftrightarrow$. называют *эквивалентностью*, так как отношение равенства есть отношение эквивалентности, а $f = g$ равносильно $f \leftrightarrow g = \mathsf{i}$.

В табл. 10 мы понимали .=. как булеву операцию; в этом

разделе соответствующая булева операция будет обозначаться как $.\leftrightarrow.$, знак же равенства будет использоваться исключительно как знак отношения (эквивалентности) (причём в речи обычно рядом фигурирует слово типа „справедливо"). Для большей ясности будем в случае $\mathcal{A} = \mathrm{i}$ писать[1] ag(A), а стало быть, будем писать, $ag(f \leftrightarrow f)$ в случае эквивалентности $f = f$ и $ag(f \wedge g \rightarrow f)$ в случае импликации $f \wedge g \geqslant f$.

Соотношение $f \wedge g \geqslant f$ читается сейчас так:

„высказывание $f \wedge g$ $\left\{\begin{matrix}\text{влечёт} \\ \text{сильнее, чем}\end{matrix}\right\}$ высказывание f",

а соотношение $f \wedge (f \rightarrow g) \geqslant g$—как

„высказывание $f \wedge (f \rightarrow g)$ $\left\{\begin{matrix}\text{влечёт} \\ \text{сильнее, чем}\end{matrix}\right\}$ высказывание g".

По внешней форме схожи, но доказываются независимо следующие две теоремы:

Теорема 8. $ag(f \wedge g)$ тогда и только тогда, когда $ag(f)$ и $ag(g)$.

Доказательство. $ag(f \wedge g)$ означает, что $f \wedge g = \mathrm{i}$. Значит, ввиду неразложимости нейтрального элемента, $f = \mathrm{i}$ и $g = \mathrm{i}$, т. е. $ag(f)$ и $ag(g)$. Обратно, если $ag(f)$ и $ag(g)$, то $f = \mathrm{i}$ и $g = \mathrm{i}$, а потому $f \wedge g = \mathrm{i}$, т. е. $ag(f \wedge g)$.

Теорема 9 [*modus ponens*[2] традиционной логики). Если $ag(f)$ и

$$„f \left\{\begin{matrix}\text{влечёт} \\ \text{сильнее, чем}\end{matrix}\right\} g",$$

то $ag(g)$.

Доказательство. По условию $f = \mathrm{i}$ и $f \geqslant g$; значит, $\mathrm{i} \geqslant g$. Вместе с $g \geqslant \mathrm{i}$ (4.1.3.1, теорема 2) это даёт $g = \mathrm{i}$, т. е. $ag(g)$.

Предикаты возникают, когда строятся высказывания относительно общих операций, которые приводят к значениям истинности. В частности, предикаты выступают в качестве условий при разборе случаев. Преобразование предикатов к эквивалентному виду и (допустимое) сужение условий-стражей посредством перехода к более сильному предикату — это задачи, с которыми постоянно приходится иметь дело при разработке программ. Знакомство с аппаратом булевой алгебры позволяет уверенно решать такие задачи.

---

[1] Ниже ag — сокращение от allgemeingültig [общезначимый (*нем.*). — *Перев.*].

[2] Модус поненс, или правило отделения, — самое знаменитое правило вывода в логике. — *Прим. изд. ред.*

Рассмотрим, например, предикат $x \leqslant 0$ в

$$\begin{aligned}&\textbf{if } x \geqslant 0 \textbf{ then } x\\ &\square\ x \leqslant 0 \textbf{ then } -x \textbf{ fi}.\end{aligned}$$

По определению $x \leqslant 0$ означает ни больше ни меньше как $x < 0 \vee x = 0$. Поскольку $f \geqslant f \vee g$, то $x < 0$ сильнее, чем $x \leqslant 0$, поэтому условие-страж во второй строке можно сузить:

$$\begin{aligned}&\textbf{if } x \geqslant 0 \textbf{ then } x\\ &\square\ x < 0 \textbf{ then } -x \textbf{ fi}.\end{aligned}$$

Так как всё ещё $x \geqslant 0 \vee x < 0 = \textbf{true}$, неопределённой ситуации возникнуть не может.

Далее, рассмотрим разбор случаев

$$\begin{aligned}&\textbf{if } x \geqslant 1 \textbf{ then if } x \geqslant 0 \textbf{ then } 3 \times x\\ &\qquad\qquad\qquad\ \square\ x < 0 \textbf{ then } -x \textbf{ fi}\\ &\square\ x < 1 \textbf{ then } x \qquad\qquad\qquad\qquad \textbf{fi}.\end{aligned}$$

Предикаты $x \geqslant 1$ и $x \geqslant 0$ сравнимы; из арифметики (вычислительная структура $\mathbb{Z}$) мы знаем, что $1 \geqslant 0$, а значит,

$$(x \geqslant 1) \geqslant (x \geqslant 0).$$

Следовательно, если $x \geqslant 1 = \textbf{true}$, то и $x \geqslant 0 = \textbf{true}$ (и $x < 0 = \neg (x \geqslant 0) = \textbf{false}$). Таким образом, первую из двух внутренних ветвей можно освободить от стража, а вторую вообще отбросить, и мы получаем

$$\begin{aligned}&\textbf{if } x \geqslant 1 \textbf{ then } 3 \times x\\ &\square\ x < 1 \textbf{ then } x \textbf{ fi}.\end{aligned}$$

В заключение рассмотрим алгоритм *contains*, который определяет, содержится ли в данном слове a знак x:

```
funct contains = (string a, char x) bool :
  if a = ◇ then false
  elsf first(a) = x then true
                    else contains(rest(a), x) fi .
```

Его можно преобразовать в „двустрочную“ рекурсию:

```
funct contains = (string a, char x) bool :
      if a = ◇ ∨ first(a) = x then if a = ◇ then false
                                   □ a ≠ ◇ ∧ first(a) = x then true fi
                              else contains(rest(a), x) fi
```

Можно ли упростить первую ветвь, когда и как? Интуитивно ясно, что, поскольку стражи

$$a = \Diamond \text{ и } a \neq \Diamond \mathbin{\dot\wedge} first(a) = x$$

исключают друг друга:

$$a = \Diamond \wedge (a \neq \Diamond \mathbin{\dot\wedge} first(a) = x) = \mathbf{false},$$

достаточно проверить, выполнено ли условие $a = \Diamond$. Более формально: при выполнении условия

$$a = \Diamond \mathbin{\dot\vee} first(a) = x$$

мы имеем $a \neq \Diamond \geqslant first(a) = x$, а значит (см. 4.1.3.1), по определению,

$$(a \neq \Diamond \mathbin{\dot\wedge} first(a) = x) = (a \neq \Diamond).$$

В результате получаем

```
~~ then if a = ◇ then false
        ▯ a ≠ ◇ then true  fi
   else ~~                    .
```

Дальнейшее упрощение даёт

```
funct contains ≡ (string a, char x) bool :
  if a = ◇ ∨̇ first(a) = x then a ≠ ◇
                         else contains (rest(a), x) fi  .
```

4.1.4. Таблицы решений

4.1.4.1. Совместные таблицы решений

В качестве вспомогательного графического средства представления разбора случаев для булевых функций многих переменных служат так называемые *таблицы решений*. Пример такой таблицы представлен на рис. 141. Каждой переменной отвечает своя строка, а по столбцам указывается, сама ли переменная или её отрицание входит в совершенную конъюнкцию. Перевод таблицы в *охраняемый* детерминированный[1] разбор случаев не составляет проблемы (ниже a, b, c, d — вида **bool**,

[1] Недетерминированные примеры в литературе не встречаются.

а u, v, w, x — вида **int**):

(*)

$$
\begin{array}{l}
\textbf{co}\ \neg(a \wedge \neg b)\ \textbf{co} \\
\textbf{if}\quad a \wedge b \wedge c \wedge d\ \textbf{then}\ u \\
\square\quad a \wedge b \wedge c \wedge \neg d\ \textbf{then}\ u \\
\square\quad a \wedge b \wedge \neg c \wedge d\ \textbf{then}\ v \\
\square\quad a \wedge b \wedge \neg c \wedge \neg d\ \textbf{then}\ v \\
\square\ \neg a \wedge b \wedge c \wedge d\ \textbf{then}\ u \\
\square\ \neg a \wedge b \wedge c \wedge \neg d\ \textbf{then}\ v \\
\square\ \neg a \wedge b \wedge \neg c \wedge d\ \textbf{then}\ v \\
\square\ \neg a \wedge b \wedge \neg c \wedge \neg d\ \textbf{then}\ w \\
\square\ \neg a \wedge \neg b \wedge c \wedge d\ \textbf{then}\ u \\
\square\ \neg a \wedge \neg b \wedge c \wedge \neg d\ \textbf{then}\ w \\
\square\ \neg a \wedge \neg b \wedge \neg c \wedge d\ \textbf{then}\ v \\
\square\ \neg a \wedge \neg b \wedge \neg c \wedge \neg d\ \textbf{then}\ x\quad \textbf{fi}
\end{array}
$$

Для столбцов, обозначенных „ошибка", ветви не предусматриваются; результат в таком случае не определён. Отсутствие соответствующих совершенных конъюнкций неизбежно в нашем примере ввиду зависимости между переменными. Так как „моложе 14 лет" влечёт „до 18 лет включительно", то комбинация $a \wedge \neg b$, если никаких „сбоев" нет, возникнуть не может; разбор случаев „ставится на предохранитель" $\neg(a \wedge \neg b)$, т. е. $a \rightarrow b$. Получающаяся упрощенная таблица показана на рис. 142, причём крестики заменены явным указанием цены билета.

Далее, порядок расположения столбцов не имеет значения. Поэтому столбцы с равными значениями результата можно

		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
a	моложе 14 лет	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
b	до 18 лет вкл.	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
c	группа	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
d	учащиеся	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
u	0,50 марок	×	×							×				×			
v	1.— марок			×	×						×	×				×	
w	1,50 марок												×		×		
x	2.— марок																×
	ошибка					×	×	×	×								

Рис. 141. Таблица решений с крестиками. [Пример заимствован из книги W. Ebben, Entscheidungstabellentechnik, de Gruyter, Berlin, 1973. Текстуально он звучит так:

Пример. Расценки на входные билеты в плавательный бассейн. Дети моложе 14 лет — 1 марка, подростки до 18 лет включительно — 1,5 марки, взрослые — 2 марки. Для групп не менее 10 человек установлен следующий тариф: *дети — 0,5 марки, подростки — 1 марка, взрослые — 1,5 марки*. Школьники, студенты и ученики на производстве независимо от возраста платят по детскому тарифу.]

		R1	R2	R3	R4	R9	R10	R11	R12	R13	R14	R15	R16
a	до 14 лет	T	T	T	T	F	F	F	F	F	F	F	F
b	до 18 лет вкл.	T	T	T	T	T	T	T	T	F	F	F	F
c	группа	T	T	F	F	T	T	F	F	T	T	F	F
d	учащиеся	T	F	T	F	T	F	T	F	T	F	T	F
		0.50	0.50	1.—	1.—	0.50	1.—	1.—	1.50	0.50	1.50	1.—	2.—

Рис. 142. Упрощённая таблица решений.

		R1	R2	R9	R13	R3	R4	R10	R11	R15	R12	R14	R16
a	до 14 лет	T	T	F	F	T	T	F	F	F	F	F	F
b	до 18 лет вкл.	T	T	T	F	T	T	T	T	F	T	F	F
c	группа	T	T	T	T	F	F	T	F	F	F	T	F
d	учащиеся	T	F	T	T	T	F	F	T	T	F	F	F
		0.50				1.—					1.50		2.—

Рис. 143. Таблица решений с объединенными по результатам столбцами.

объединить (рис. 143). Этому отвечает дизъюнкция (дизъюнктивное объединение) соответствующих совершенных конъюнкций:

$$
\begin{aligned}
&\textbf{if}\ (a \wedge b \wedge c \wedge d) \vee (a \wedge b \wedge c \wedge \neg d) \vee (\neg a \wedge b \wedge c \wedge d)\\
&\qquad \vee (\neg a \wedge \neg b \wedge c \wedge d)\ \textbf{then}\ u\\
&\square\ (a \wedge b \wedge \neg c \wedge d) \vee (a \wedge b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge c \wedge \neg d)\\
&\qquad \vee (\neg a \wedge b \wedge \neg c \wedge d) \vee (\neg a \wedge \neg b \wedge \neg c \wedge d)\\
&\qquad \textbf{then}\ v\\
&\square\ (\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge c \wedge \neg d)\ \textbf{then}\ w\\
&\square\ \neg a \wedge \neg b \wedge \neg c \wedge \neg d\ \textbf{then}\ x\quad \textbf{fi}
\end{aligned}
$$

В качестве стражей стоят четыре высказывания в дизъюнктивной нормальной форме. Закон дистрибутивности позволяет упростить это, например, до

$$
\begin{aligned}
&\textbf{if}\ c \wedge (a \wedge b \vee \neg a \wedge d) && \textbf{then}\ u\\
&\square\ \neg c \wedge (a \wedge b \vee \neg a \wedge d) \vee \neg a \wedge b \wedge c \wedge \neg d && \textbf{then}\ v\\
&\square\ \neg a \wedge \neg d \wedge (b \wedge \neg c \vee \neg b \wedge c) && \textbf{then}\ w\\
&\square\ \neg a \wedge \neg d \wedge (\neg b \wedge \neg c) && \textbf{then}\ x\quad \textbf{fi}
\end{aligned}
$$

С таблицами решений так далеко уйти нельзя, и уже одно это показывает границы их применения. Единственный способ упростить таблицу — это объединить столбцы, если результат не зави-

сит от одной (или нескольких) переменных; соответствующие „пустые места“ отмечаются в таблице прочерком (рис. 144).

		R1/2	R9/13	R3/4	R10	R11/15	R12	R14	R16
a	до 14 лет	T	F	T	F	F	F	F	F
b	до 18 лет вкл.	T	—	T	T	—	T	F	F
c	группа	T	T	F	T	F	F	T	F
d	учащиеся	—	T	—	F	T	F	F	F
		0.50	0.50	1.—	1.—	1.—	1.50	1.50	2.—

Рис. 144. Таблица решений с прочерками.

Зависимость между переменными позволяет сократить число „задаваемых вопросов“; в нашем примере ввиду импликации

$\{a \to b\}$		R1/2	R9/13	R3/4	R10	R11/15	R12	R14	R16
a	до 14 лет	T	F	T	F	F	F	—	—
b	до 18 лет вкл.	—	—	—	T	—	T	F	F
c	группа	T	T	F	T	F	F	T	F
d	учащиеся	—	T	—	F	T	F	F	F
		0,50	0,50	1,—	1,—	1.—	1.50	1.50	2.—

Рис. 145. Таблица решений с предохранителем $a \to b$.

$a \to b$ можно отказаться от b в R1/2 и R3/4, а ввиду $\neg b \to \neg a$ — от a в R14 и R16 (рис. 145). Получается охраняемый разбор случаев

```
co a→b co    if  a       ∧ c        then u
             ▯ ¬a ∧   c ∧  d        then u
             ▯  a        ∧¬c        then v
             ▯ ¬a ∧  b ∧  c ∧¬d     then v
             ▯ ¬a ∧¬c ∧   d         then v
             ▯ ¬a ∧  b ∧¬c ∧¬d      then w
             ▯      ¬b ∧  c ∧¬d     then w
             ▯      ¬b ∧¬c ∧¬d      then x   fi
```

с восемью, но всё-таки не с четырьмя ветвями.

4.1.4.2. Последовательные таблицы решений

Таблицы решений используются обычно не описанным выше „совместным“ способом, а последовательным способом, осуществляя проверку условий слева направо. Поэтому объедине-

ние произвольных столбцов теперь не разрешается — всё зависит от порядка рассмотрения столбцов. Такие таблицы решений соответствуют *последовательному* разбору случаев.

	{a→b}.	R1/2	R3/4	R9/13	R10	R11/15	R12	R14	R16
							последовательная		
a	до 14 лет	T	T	—	—	—	—	—	—
b	до 18 лет вкл.	—	—	—	T	—	T	F	F
c	группа	T	—	T	T	F	F	T	F
d	учащиеся	—	—	T	F	T	F	F	F
		0.50	1.—	0.50	1.—	1.—	1.50	1.50	2.—

Рис. 146. Последовательная таблица решений, промежуточная форма.

Совместную („строго независимую") таблицу решений всегда можно сделать последовательной, задав произвольным образом порядок следования столбцов и „читая" столбцы в этом порядке.

За счёт расположения столбцов в последовательность можно иногда сократить число „опросов", используя следующий факт:

	{a→b}	R1/2	R3/4	R9/13	R10	R11/15	R12	R14	R16
							последовательная		
a	до 14 лет	T	T	—	—	—	—	—	—
b	до 18 лет вкл.	—	—	—	T	—	T	—	—
c	группа	T	—	T	T	—	—	T	—
d	учащиеся	—	—	T	—	T	—	—	—
		0.50	1.—	0.50	1.—	1.—	1.50	1.50	2.—

Рис. 147. Последовательная таблица решений, окончательная форма без отрицательных опросов.

в таблице решений, читаемой слева направо, можно (*не изменяя значение результата*) дизъюнктивно объединить каждую комбинацию условий с любой другой, расположенной *левее*.

Так, можно, например, на рис. 145 комбинацию T — F — в столбце R3/4 объединить с комбинацией T — T — из столбца R1/2, что даст комбинацию T — — —. Если ещё поставить столбец R3/4 на второе место, то во всех последующих столбцах можно опустить F в первой строке (рис. 146). Дальнейшие упрощения получаются: в столбце R10 путем объединения с комбинацией — — TT, стоящей в R9/13; в столбце R11/15 — путём объединения с комбинацией из R9/13; в столбцах R12 и R14 — путём объединения с новыми комбинациями — — — T

из R11/15 и —TT— из R10. Наконец, таким же способом устраняются все F в столбце R16. В результате получаем таблицу (рис. 147), в которой „отрицательных опросов" нет. Всё-таки в получающемся последовательном разборе случаев

```
co a→b co          if    a ∧ c then u
                   elsf a      then v
                   elsf c ∧ d then u
                   elsf b ∧ c then v
                   elsf d      then x
                   elsf b      then w
                   elsf c      then w
                               else x  fi
```

глубины 7 остаётся ещё 10 опросов. Таблицы решений как инструмент для упрощения разбора случаев оптимальны не во всех отношениях. Из приведенной формы (*) можно, например, получить древовидный разбор случаев

```
if a ∧ b ∨ ¬a ∧ d then if c then u
                           else v  fi
elsf   b ∧  c              then v
elsf ¬b ∧ ¬c               then x
                           else w  fi
```

с всего 9 опросами и глубины 3. А из исходной таблицы считывается разбор случаев

```
if ¬d then if ¬c then   if a then v
                      elsf b then w
                             else x  fi
              else      if a then u
                      elsf b then v
                             else w  fi  fi
else if ¬c then v
           else u                    fi  fi
```

с всего 7 опросами и глубины 4 — вариант, особенно выгодный в случае, если цены билетов будут часто изменяться.

4.1.5. Переключательные функции

В этом разделе (двоичные) переключательные функции, т. е. булевы функции над моделью BIT (см. 4.1.1.5), исследуются с точки зрения их технической реализации.

4.1.5.1. Символические изображения переключательных функций

Реализацию переключательной функции f от n переменных представляют себе как „чёрный ящик" F с входами $a_1, a_2, \ldots$ $\ldots, a_n$ и выходом $f(a_1, a_2, \ldots, a_n)$. При этом совсем не обязательно ограничиваться одним выходом; в общем случае имеется m переключательных функций $j_i = f_i(a_1, a_2, \ldots$ $\ldots, a_n)$, $i = 1, 2, \ldots, m$, для выходов $j_1, \ldots, j_m$ (рис. 148). Такое „функциональное образование", называемое ***комбинационной схемой***[1] (для $m = 1$ — ***переключательным элементом***), можно рассматривать как кодовый преобразователь („перекодировщик") с n-разрядными двоичными словами на входе и m-разрядными двоичными словами на выходе.

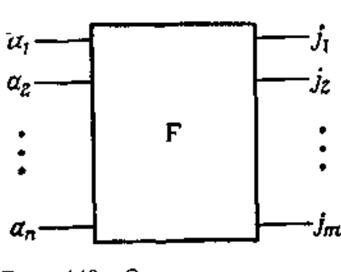


Рис. 148. Символическое изображение переключательного элемента.

Для трёх переключательных функций: отрицания, конъюнкции и дизъюнкции — вместо прямоугольника используются три

Символическое изображение	a, b — r	a, b — r	a — r или a — r
Формула	$r = a \wedge b$	$r = a \vee b$	$r = \neg a$
Название	И-элемент	ИЛИ-элемент	НЕ-элемент

Рис. 149. Символические изображения для конъюнкции, дизъюнкции и отрицания.

особых символических изображения (рис. 149). Соответствующие переключательные элементы называются ***НЕ-элементом***, ***И-элементом*** и ***ИЛИ-элементом***; о них говорят как о ***логических элементах***.

---

[1] В оригинале Schaltnetz (дословно: переключательная сеть). В отечественной литературе обычно используются термины „комбинационная схема" или „переключательная схема без памяти". — *Прим. перев.*

4.1.5.2. Соединение символических изображений

Применение основных операций к переключательным функциям реализуется посредством соответствующего соединения

Рис. 150. Символическое изображение NOR-элемента, закон де Моргана.

символических изображений. При этом получается схема потока данных специального вида.

Таким образом возникают комбинационные схемы (в том числе и с многими выходами), построенные лишь из конъюнкции, дизъюнкции и отрицания. Вместо отрицания на входе конъюнкции или дизъюнкции ставят просто жирную „точку отрицания"; это даёт такие картинки, как, например, на рис. 150 (где левая часть в соответствии со сказанным в разделе 4.1.1.3 изображает NOR-элемент).

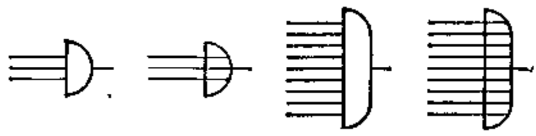


Рис. 151. Символические изображения с более чем двумя входами.

Законы, перечисленные в 4.1.1, представляются с помощью символических изображений весьма наглядно. Прежде всего это

Рис. 152. Закон поглощения.

относится к законам де Моргана (рис. 150) и закону инволюции („закону отрицания отрицания"), который означает, что две точки отрицания взаимно уничтожаются.

Закон коммутативности проявляется в симметрии символических изображений относительно входов. Закон ассоциативности отражает возможность снабжать символические изображения более чем двумя входами (рис. 151).

Законы поглощения и дистрибутивности представлены в виде картинок на рис. 152 и рис. 153. В обоих случаях мы огра-

ничились одним из двойственных законов, другой получается перестановкой конъюнкции и дизъюнкции. Таким образом,

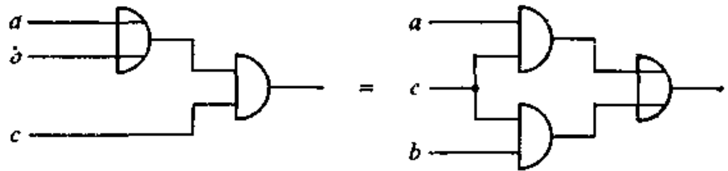


Рис. 153. Закон дистрибутивности.

закон дистрибутивности можно рассматривать как „протаскивание символического изображения с удвоением[1]".

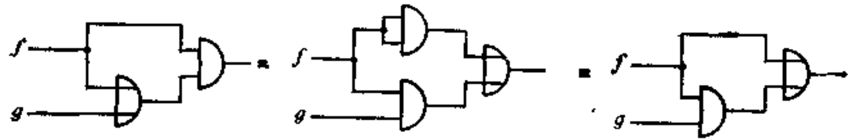


Рис. 154. $f \wedge (f \vee g) = f \vee (f \wedge g)$.

Между прочим, закон поглощения не вытекает из законов дистрибутивности и идемпотентности, однако из них, во всяком случае, следует, что

$$f \wedge (f \vee g) = (f \wedge f) \vee (f \wedge g) = f \vee (f \wedge g)$$

(см. рис. 154). Из закона же поглощения следует закон идемпотентности:

$$f = f \wedge (f \vee (f \wedge g)) = f \wedge f.$$

4.1.5.3. Пример: полусумматор, полный сумматор

Для поразрядного сложения двух двоичных закодированных чисел применяется комбинационная схема, называемая ***полусумматором***[2] (или ***квазисумматором***), с двумя перестановочными входами a, c и двумя выходами[3]

$$s = a \leftarrow\!|\!\rightarrow c,$$
$$\ddot{u} = a \wedge c.$$

---

[1] Того изображения, через которое мы протаскиваем. — *Прим. перев.*

[2] В оригинале Halbaddierer, отсюда используемое далее обозначение **HA**. От того же Addierer обозначение **A**, которое появится на рис. 156. — *Прим. изд. ред.*

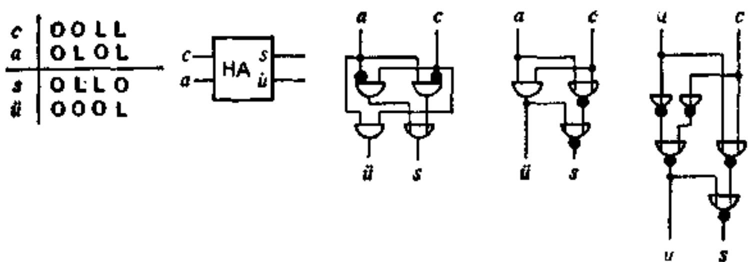
[3] Ниже s — от Summe (сумма), а $\ddot{u}$ — от Übertragsausgang (выход переноса); см. 4.2.3. — *Прим. изд. ред.*

На рис. 155 представлены три реализации полусумматора. В случае центральной реализации использовано преобразование

$$(\neg a \wedge b) \vee (a \wedge \neg b) = ((\neg a \wedge b) \vee a) \wedge ((\neg a \wedge b) \vee \neg b)$$
$$= ((a \vee b) \wedge (\neg a \vee \neg b)) = (a \vee b) \wedge \neg (a \wedge b) = (a \mathbin{\overline{\vee}} b) \mathbin{\overline{\vee}} (a \wedge b).$$

Справа показана реализация с пятью NOR-элементами (заметим, что $a \mathbin{\overline{\vee}} a = \neg a$).

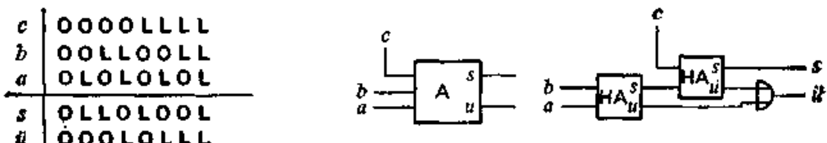
Полусумматоры можно применять для сложения двух чисел, закодированных в прямом коде (см. 1.4.2). При этом наряду



c	O	O	L	L
a	O	L	O	L
s	O	L	L	O
ü	O	O	O	L

Рис. 155. Полусумматор.

с поразрядными значениями обоих чисел всегда надо учитывать и перенос из ближайшего „младшего“ разряда. Символическое



c	O	O	O	O	L	L	L	L
b	O	O	L	L	O	O	L	L
a	O	L	O	L	O	L	O	L
s	O	L	L	O	L	O	O	L
ü	O	O	O	L	O	L	L	L

Рис. 156. Полный сумматор.

обозначение и таблица значений такого ***полного сумматора*** с тремя перестановочными входами a, b, c и двумя выходами s, $ü$ представлены на рис. 156.

4.1.5.4. Пример: перекодировщики

В принципе всякий перекодировщик может быть построен в два этапа: на первом этапе входной код, содержащий n знаков, преобразуется в код 1-из-n, а на втором — код 1-из-n преобразуется в выходной код. Если предположить, что вместе со всякой переключательной переменной на входе мы располагаем и её отрицанием и что на выходе вместе с каждой переключатель-

ной переменной используется и её отрицание, то, основываясь на булевой нормальной форме, реализацию первого этапа можно строить из одних конъюнкций, а второго — из одних дизъюнкций.

Пример преобразования кода плюс-3 в прямой код представлен на рис. 157. Если применить ко входам и выходам от-

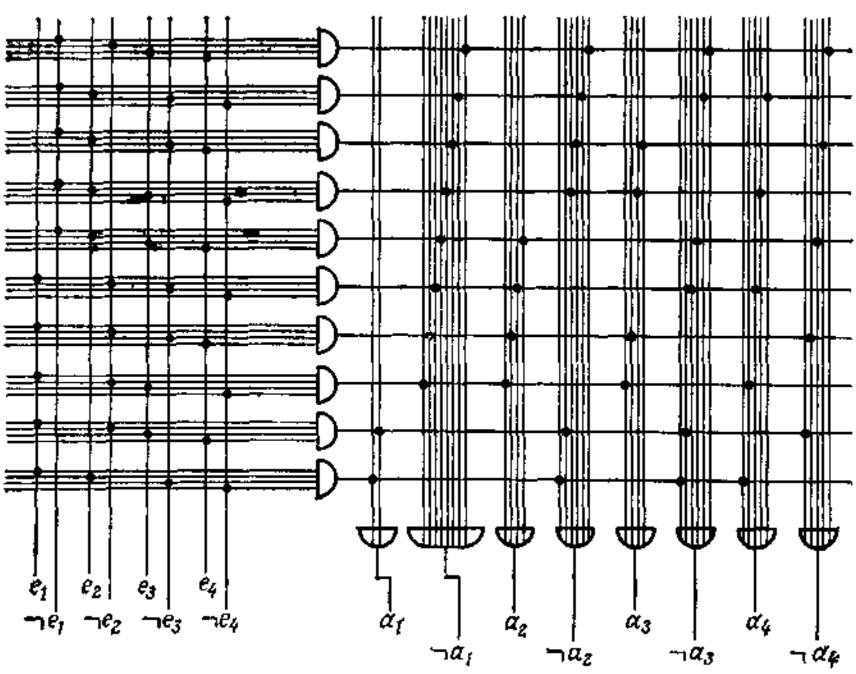


Рис. 157. Преобразование кода плюс-3 в прямой код.

рицание (заменить e_i на $\neg e_i$ и a_i на $\neg a_i$), то получится двухступенчатый перекодировщик, основанный на NOR-элементах.

4.1.6. Техническая реализация комбинационных схем

Катодные лампы накаливания, более 40 лет назад впервые использованные для построения комбинационных схем, на сегодня так же устарели, как и известные свыше 100 лет электромеханические реле. Уже более 25 лет для реализации комбинационных схем используются преимущественно транзисторы. Для примера на рис. 158 изображена принципиальная схема

реализации NOR-элемента с помощью двух *n-p-n*-транзисторов [1]. Как было упомянуто в 4.1.1.3, любая комбинационная схема может быть построена из одних NOR-элементов [2].

Недавно разработанная технология ***интегральных схем*** позволяет получать тысячи логических элементов на пластинке

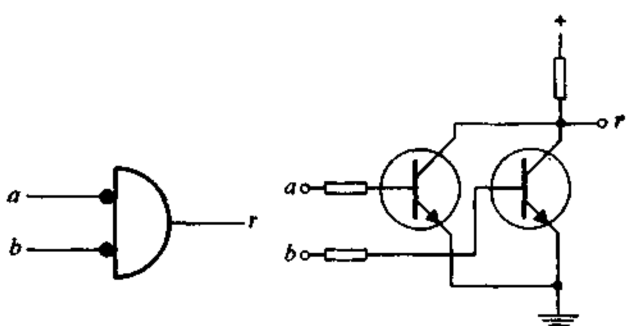


Рис. 158. Реализация NOR-элемента с помощью двух *n-p-n*-транзисторов.

площадью всего несколько квадратных миллиметров; процесс их изготовления включает в себя напыление и диффузионное внедрение полупроводниковых материалов.

4.2*. Двоичное кодирование

„Но да будет слово ваше: да, да; нет, нет; а что сверх этого, то от лукавого."

Евангелие от Матфея 5, 37

Двоичное кодирование (см. 1.4.2) означает представление объектов произвольного сорта двоичными словами — объектами сорта *bits*, соотв. *bitstring* (см. 2.1.3.4). Причём вначале не предполагается, что длина этих слов фиксирована или ограничена какой-то фиксированной константой. В 1.4.2 нам уже встречалось двоичное кодирование (десятичных) цифр (рис. 33); двоичное кодирование натуральных чисел аналогично. Особенно важное кодирование натуральных чисел получается из их двоичной записи посредством сопоставления

(2) $$0 \triangleq \mathsf{O}, \quad 1 \triangleq \mathsf{L};$$

[1] При этом предполагается положительный знак сигнала: O ≙ »низкое напряжение«, L ≙ »высокое напряжение«. При обратном соответствии получим реализацию NAND-элемента.

[2] Подробности см. в [41].

* Этот раздел можно читать сразу же после 2.4.

при этом *прямом коде* обычному порядку натуральных чисел соответствует лексикографический (без пробелов) порядок двоичных слов. Начальные O в двоичных словах опускаются; в частности, натуральное число 0 кодируется пустым двоичным словом.

Представление целых чисел требует дополнительного бита для указания знака числа. Подробнее об этом и о представлении вещественных (машинных) чисел говорится в приложении А.

Очевидное двоичное кодирование сорта **bool**, соотв. *Boolean*, с помощью соответствия (1) из 2.1.3.6 и 4.1.1.2 подразумевается всюду в дальнейшем.

Двоичное кодирование сорта **char**, соотв. *char*, можно выполнять различными способами, исходя из различных технических точек зрения; см. 1.4.2. В последующем мы всегда предполагаем, что кодировка осуществляется в соответствии с семиразрядным кодом ISO в редакции ASCII (рис. 30). Для получения двоичной кодировки объектов сорта **string**, соотв. *string*, можно в таком случае брать конкатенацию кодирующих двоичных слов отдельных знаков; стыки слов однозначно определяются.

Иначе обстоит дело при двоичном кодировании размеченных деревьев, т. е. объектов сорта **lisp**, соотв. *lisp*. Здесь не должна быть потеряна содержащаяся в скобках информация о структуре размеченного дерева (см. 2.1.3.5). Простая возможность состоит в том, чтобы включить в код два специальных знака ⟨и⟩ и с их помощью двоично закодировать, скажем,

$$\langle 'A' \langle 'B' \, 'C' \rangle \rangle \quad \text{как} \quad A\langle BC \rangle,$$
$$\langle \langle 'A' \, 'B' \rangle \, 'C' \rangle \quad \text{как} \quad \langle AB \rangle C.$$

4.2.1. Двоичное сравнение

Операция сравнения $.=.$, соотв. $.\neq.$, имеет важное практическое значение для всех сортов. Для однозначно обратимых (двоичных) кодировок — а только такие и рассматриваются в дальнейшем — два объекта равны тогда и только тогда, когда кодирующие их (двоичные) слова поразрядно совпадают.

Например, для **char**, соотв. *char*, это приводит при кодировании с помощью семиразрядного кода ISO к задаче построения комбинационной схемы для сравнения двух семиразрядных слов $a_7a_6a_5a_4a_3a_2a_1$ и $b_7b_6b_5b_4b_3b_2b_1$. Переключательная функция имеет вид

$$(a_1 \longleftrightarrow b_1) \wedge (a_2 \longleftrightarrow b_2) \wedge (a_3 \longleftrightarrow b_3) \wedge (a_4 \longleftrightarrow b_4) \wedge (a_5 \longleftrightarrow b_5) \wedge (a_6 \longleftrightarrow b_6) \wedge (a_7 \longleftrightarrow b_7).$$

На рис. 159 изображена эквивалентная комбинационная схема с полусумматорами и одним многовходовым NOR-элементом —

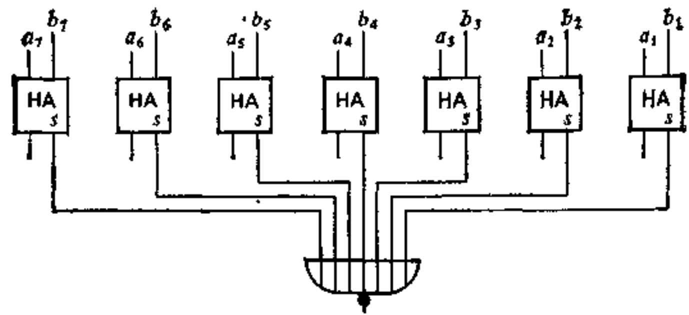


Рис. 159. Схема сравнения.

так называемая ***схема сравнения***. По поводу общего решения задачи сравнения, использующего вычитание, см. 4.2.2.2.

4.2.2. Двоичная арифметика

При двоичном кодировании целых чисел необходимо ещё перенести операции над объектами сорта **int** (*integer*) на соответствующие двоичные слова. О том, как это сделать систематически, и пойдёт речь дальше.

4.2.2.1. Двоичный счётчик

Операция перехода к следующему числу для натуральных чисел соответствует операции перехода к следующему слову в смысле лексикографического порядка для двоичных слов. Мы получим нужный алгоритм из следующих соображений.

Пусть $c(a) = \hat{a}$ — двоичное слово, кодирующее натуральное число a (без начальных „нулей" O). В таком случае $a = 0$ эквивалентно $\hat{a} = \lozenge$, а для $a \neq 0$

$$\neg\, \mathbf{odd}\ a \text{ эквивалентно } last(\hat{a}) = \mathsf{O},$$
$$\mathbf{odd}\ a \text{ эквивалентно } last(\hat{a}) = \mathsf{L}.$$

Далее, для $a \neq 0$

$$c(a \times 2) = \hat{a} + \langle \mathsf{O} \rangle,$$
$$c(a \times 2 + 1) = \hat{a} + \langle \mathsf{L} \rangle,$$
$$\left.\begin{array}{ll} c(a/2) & \text{(определено для чётных } a) \\ c((a-1)/2) & \text{(определено для нечётных } a) \end{array}\right\} = lead(\hat{a}).$$

Определим теперь[1] *binsucc* равенствами

$$binsucc(c(a)) = c(succ(a)) \text{ и } binsucc(\hat{0}) = \langle \mathsf{L} \rangle.$$

Для чётных $a \neq 0$ имеем (ввиду тождества $a + 1 = (a/2) \times \times 2 + 1$)

$$c(succ(a)) = binsucc(\hat{a}) = lead(\hat{a}) + \langle \mathsf{L} \rangle.$$

Для нечётных $a \neq 0$

$$c(succ(a)) = binsucc(\hat{a}) = binsucc(lead(\hat{a})) + \langle \mathsf{O} \rangle$$

(ввиду тождества $a + 1 = ((a - 1)/2 + 1) \times 2$). Отсюда получаем следующий рекурсивный алгоритм двоичного „счёта“ (то что этот алгоритм оканчивается, очевидно):

```
funct binsucc ≡ (bits u
                  co isc(u) co) bits:

  if u = ◇
  then ⟨L⟩
  elsf last(u) = O
  then lead(u) + ⟨L⟩
  else binsucc(lead(u)) + ⟨O⟩ fi
```

```
function binsucc(u : bitstring
                 {isc(u)}) : bitstring;
begin
  if isempty(u)
  then binsucc ⇐ postfix(empty, L)
  else if last(u) = O
    then binsucc ⇐ postfix(lead(u), L)
    else binsucc ⇐
           postfix(binsucc(lead(u)), O)
end
```

причём двоичные слова с начальными O исключаются :[2]

```
funct isc ≡ (bits u) bool:
        u = ◇ ∨ first(u) = L
```

```
function isc (u : bitstring) : Boolean;
begin isc ⇐ isempty(u) ∨ (first(u) = L)
end
```

Очевидно, что за подформулой *binsucc*(*lead*(*u*)) последней строки подпрограммы *binsucc* стоит действие, которое при обычном выполнении сложения в десятичной системе называют „переносом единички в старший разряд“.

Аналогично рассматривается функция перехода к предшествующему числу. На основе тождества

$$a - 1 = (a/2 - 1) \times 2 + 1 \text{ для чётных } a \neq 0$$

---

[1] Ниже *bin*, конечно, от binary (бинарный). — *Прим. изд. ред.*

[2] Буква *c* в *isc*, по-видимому, от code (код). — *Прим. изд. ред.*

получаем алгоритм

```
funct binpred ≡ (bits u                          | function binpred(u: bitstring
     co u ≠ ◇ ∧ first(u) = L co) bits:           |   {(not isempty(u)) ∧ (first(u) = L)}): bitstring;
                                                 | begin
  if lead(u) = ◇                                 |   if isempty(lead(u))
  then ◇                                         |   then binpred ⇐ empty
  elsf last(u) = L                               |   else if last(u) = L
    then lead(u) + ⟨O⟩                           |     then binpred ⇐ postfix(lead(u), O)
    else binpred(lead(u)) + ⟨L⟩          fi      |     else binpred ⇐
                                                 |          postfix(binpred(lead(u)), L)
                                                 | end
```

4.2.2.2. Сложение и вычитание

Если вышеприведенные алгоритмы *binsucc* и *binpred* использовать вместо *succ* и *pred* в соответствующих определениях, например в подпрограмме *plus* из 2.4.1.2, то мы получим алгоритм для сложения и вычитания двоично представленных натуральных чисел. При этом вычисление суммы $a + b$ требует b вызовов *binsucc* и *binpred*. Однако двоичное представление позволяет строить значительно более эффективные алгоритмы.

```
funct add ≡ (int a, b co                         | function add(a, b: integer
               a ≧ 0 ∧ b ≧ 0 co) int:            |        {(a ≧ 0) and (b ≧ 0)}): integer;
                                                 | begin
  if b = 0                                       |   if b = 0
  then a                                         |   then add ⇐ a
  ▯ a = 0                                        |   ▯ a = 0
  then b                                         |   then add ⇐ b

  ▯ a ≠ 0 ∧ b ≠ 0 then                           |   ▯ a ≠ 0 ∧ b ≠ 0 then
     if odd a ∧ odd b                            |      if odd(a) and odd(b)
     then (add((a − 1) ÷ 2,                      |      then add ⇐ (add((a − 1) div 2,
               (b − 1) ÷ 2) + 1) × 2             |                 (b − 1) div 2) + 1)*2
     ▯ ¬odd a ∧ odd b                            |      ▯ not odd(a) and odd(b)
     then add(a ÷ 2, (b − 1) ÷ 2) × 2 + 1        |      then add ⇐ add(a div 2, (b − 1) div 2)*2 + 1
     ▯ odd a ∧ ¬odd b                            |      ▯ odd(a) and not odd(b)
     then add((a − 1) ÷ 2, b ÷ 2) × 2 + 1        |      then add ⇐ add((a − 1) div 2, b div 2)*2 + 1
     ▯ ¬odd a ∧ ¬odd b                           |      ▯ not odd(a) and not odd(b)
     then add(a ÷ 2, b ÷ 2) × 2          fi fi   |      then add ⇐ add(a div 2, b div 2)*2
                                                 | end
```

Рис. 160.

Сначала улучшим алгоритм *plus* до „быстрого" варианта[1] *add* (рис. 160), основанного на том, что (для чётных a, b) справедливо (в силу закона дистрибутивности) тождество

$$add(a, b) = add(a \div 2, b \div 2) \times 2.$$

[1] Ниже *add* — от add (складывать). — *Прим. изд. ред.*

Решающим моментом является то, что в двоичной системе операции удвоения и деления пополам осуществляются очень

```
funct binadd ≡ (bits a, bits b
                co isc(a) ∧ isc(b) co) bits:

if b = ◇
then a
▯ a = ◇
then b
▯ a ≠ ◇ ∧ b ≠ ◇ then
    if last(a) = L ∧ last(b) = L
    then
      binsucc(binadd(lead(a),
                     lead(b))) + ⟨O⟩
    ▯ (last(a) = O ∧ last(b) = L) ∨
      (last(a) = L ∧ last(b) = O)
    then
      binadd(lead(a), lead(b)) + ⟨L⟩
    ▯ last(a) = O ∧ last(b) = O
    then
      binadd(lead(a), lead(b)) + ⟨O⟩   fi fi
```

```
function binadd(a, b: bitstring
        {isc(a) and isc(b)}): bitstring;
begin
  if isempty(b)
  then binadd ⇐ a
  ▯ isempty(a)
  then binadd ⇐ b
  ▯ a ≠ ◇ ∧ b ≠ ◇ then
    if (last(a) = L) and (last(b) = L)
    then binadd ⇐
      postfix(binsucc(binadd(lead(a),
                             lead(b))), O)
    ▯ ((last(a) = O) and (last(b) = L)) or
      ((last(a) = L) and (last(b) = O))
    then binadd ⇐
      postfix(binadd(lead(a), lead(b)), L)
    ▯ (last(a) = O) and (last(b) = O)
    then binadd ⇐
      postfix(binadd(lead(a), lead(b)), O)
end
```

Рис. 161.

просто. В результате получаем подпрограмму *binadd*, представленную на рис. 161.

```
funct sub = (int a, b
             co a ≧ b ∧ b ≧ 0 co) int:

  if b = 0
  then a
  else
    if odd a ∧ odd b
    then sub((a − 1) ÷ 2, (b − 1) ÷ 2) × 2
    ▯ ¬ odd a ∧ odd b
    then sub(a ÷ 2, (b − 1) ÷ 2) × 2 − 1

    ▯ odd a ∧ ¬ odd b
    then sub((a − 1) ÷ 2, b ÷ 2) × 2 + 1
    ▯ ¬ odd a ∧ ¬ odd b
    then sub(a ÷ 2, b ÷ 2) × 2          fi fi
```

```
function sub(a, b: integer
        {(a ≧ b) and (b ≧ 0)}): intege
begin
  if b = 0
  then sub ⇐ a
  else
    if odd(a) and odd(b)
    then sub ⇐ sub((a − 1) div 2, (b − 1) div 2)*2
    ▯ not odd(a) and odd(b)
    then sub ⇐ sub(a div 2, (b − 1) div 2)*2 − 1

    ▯ odd(a) and not odd(b)
    then sub ⇐ sub((a − 1) div 2, b div 2)*2 + 1
    ▯ not odd(a) and not odd(b)
    then sub ⇐ sub(a div 2, b div 2)*2
end
```

Рис. 162.

Аналогично пишется „быстрый" вариант *sub* для вычитания (рис. 162), который будет оканчивающимся при условии-предохранителе $a \geqslant b$. Переход к двоичному представлению даёт подпрограмму *binsub*, приведённую на рис. 163. Второй случай

во внутреннем разборе случаев отражает то, что в начальной школе (для десятичной системы) выражают словами „занимаем

```
funct binsub ≡ (bits a, bits b
        co a ≥ b ∧ isc(a) ∧ isc(b) co) bits :

if b = ◇
then a
else
  if last(a) = L ∧ last(b) = L
  then
    binsub(lead(a), lead(b)) + ⟨O⟩
  ▯ last(a) = O ∧ last(b) = L
  then
    binpred(binsub(lead(a), lead(b)) + ⟨O⟩)
  ▯ last(a) = L ∧ last(b) = O
  then
    binsub(lead(a), lead(b)) + ⟨L⟩
  ▯ last(a) = O ∧ last(b) = O
  then
    binsub(lead(a), lead(b)) + ⟨O⟩          fi fi
```

```
function binsub(a, b : bitstring
    {(a ≥ b) and isc(a) and isc(b)}) : bitstring;
begin
  if isempty(b)
  then binsub ⇐ a
  else
    if (last(a) = L) and (last(b) = L)
    then binsub ⇐
      postfix(binsub(lead(a), lead(b)), O)
    ▯ (last(a) = O) and (last(b) = L)
    then binsub ⇐
        binpred(postfix(binsub(lead(a), lead(b)), O))
    ▯ (last(a) = L) and (last(b) = O)
    then binsub ⇐
      postfix(binsub(lead(a), lead(b)), L)
    ▯ (last(a) = O) and (last(b) = O)
    then binsub ⇐
      postfix(binsub(lead(a), lead(b)), O)
end
```

Рис. 163.

единичку". О том, как быть со знаками при сложении и вычитании целых чисел, см. приложение А.

Сравнение на равенство $.=.$, соотв. $.\neq.$, для натуральных и целых чисел с помощью вычитания может быть сведено к сравнению с 0 в смысле раздела 4.2.1, стало быть к многократному простому сравнению. Сравнения $.\leqslant.$, $.<.$, $.>.$, $.\geqslant.$ таким же образом могут быть сведены к проверке знака.

Сравнение двоично кодированных слов на взаимное расположение в смысле лексикографического порядка тоже может быть сведено к вычитанию и проверке знака.

```
 23  ×  43
 46     21
 92     10   (зачёркнуто: 92)
184      5
368      2   (зачёркнуто: 368)
736      1
----
989
```

Рис. 164. „Древнеегипетское умножение", называемое также „русским", „эфиопским", „крестьянским" методом.

4.2.2.3. Умножение

Осталось ещё объяснить, как выполняется в двоичной арифметике умножение. „Быстрый" вариант обыкновенного умножения (задаваемого подпрограммой *genitor* из 2.6.3 со сложением в качестве исходной функции) был известен уже в Древнем Египте (рис. 164). Он даёт алгоритм

funct *mult* ≡ (**int** *a*, *b* **co** *a* ≥ 0 **co**) **int**:	**function** *mult* (*a*, *b* : *integer* {*a* ≥ 0}) : *integer*;
	begin
if *a* = 0	**if** *a* = 0
then 0	**then** *mult* ← 0
else	**else**
if odd *a*	**if** *odd* (*a*)
then *mult* ((*a* − 1) ÷ 2, *b*) × 2 + *b*	**then** *mult* ← *mult* ((*a* − 1) **div** 2, *b*)*2 + *b*
▯ ¬**odd** *a*	▯ **not** *odd* (*a*)
then *mult* (*a* ÷ 2, *b*) × 2 **fi fi**.	**then** *mult* ← *mult* (*a* **div** 2, *b*)*2
	end

После перехода к двоичному представлению числа *a* получаем [1]

funct *bmult* ≡ (**bits** *a*, **int** *b* **co** *isc*(*a*) **co**) **int**:	**function** *bmult* (*a* : *bitstring*; *b* : *integer* {*isc*(*a*)}) : *integer*;
	begin
if *a* = ◇	**if** *isempty* (*a*)
then 0	**then** *bmult* ← 0
else	**else**
if *last* (*a*) = **L**	**if** *last* (*a*) = **L**
then *bmult* (*lead* (*a*), *b*) × 2 + *b*	**then** *bmult* ← *bmult* (*lead* (*a*), *b*)*2 + *b*
▯ *last* (*a*) = **O**	▯ **last** (*a*) = **O**
then *bmult* (*lead* (*a*), *b*) × 2 **fi fi**.	**then** *bmult* ← *bmult* (*lead* (*a*), *b*)*2
	end

Наконец, представляя и второй множитель в двоичном виде, приходим к подпрограмме

funct *binmult* ≡ (**bits** *a*, **bits** *b* **co** *isc*(*a*) ∧ *isc*(*b*) **co**) **bits**:	**function** *binmult* (*a*, *b* : *bitstring* {(*isc* (*a*)) **and** (*isc* (*b*))}) : *bitstring*;
	begin
if *a* = ◇	**if** *isempty* (*a*)
then ◇	**then** *binmult* ← *empty*
else	**else**
if *last* (*a*) = **L**	**if** *last* (*a*) = **L**
then	**then** *binmult* ←
binadd (*binmult* (*lead* (*a*), *b*) + ⟨**O**⟩, *b*)	*binadd* (*postfix* (*binmult* (*lead* (*a*), *b*), **O**), *b*)
▯ *last* (*a*) = **O**	▯ *last* (*a*) = **O**
then	**then** *binmult* ←
binmult (*lead* (*a*), *b*) + ⟨**O**⟩ **fi fi**.	*postfix* (*binmult* (*lead* (*a*), *b*), **O**)
	end

4.2.2.4. Операции с двоично представленными последовательностями знаков и размеченными деревьями

Введённые в 2.1.3.4 операции для сортов **string** (*string*) и **lisp** (*lisp*) никак не затрагиваются двоичным кодированием, по-

[1] Ниже *b*, как и *bin*, — от binary. — *Прим. изд. ред.*

скольку единственное, что требуется,— это группировать биты соответствующим образом (в случае принятого нами семиразрядного кода ISO — в семёрки).

4.2.3. Арифметика с ограниченным числом разрядов

Наложим теперь ограничение, что все рассматриваемые нами натуральные числа не превосходят некоторого определённого числа, скажем представимы с помощью не более чем k двоичных разрядов и, стало быть, лежат в интервале от 0 до $2^k - 1$.

В таком случае рекурсивный алгоритм *binsucc* из 4.2.2.1 завершается самое позднее после k воплощений; при условии-предохранителе $length(u) \leqslant k$ его можно также задать явной формулой, которая получается последовательными подстановками. Если мы теперь перейдём от использовавшегося нами до сих пор двоичного кодирования, отвечающего обычному представлению чисел без начальных нулей, к кодированию двоичными словами, которые дополнены „нулями" до полного k-разрядного слова (см. рис. 33, прямой код), то это будет в точности соответствовать определению, принятому в стандартном алголе-68 (см. примечание к **bits** в табл. 4).

Поскольку переход к числу, следующему за $\overbrace{\mathsf{LLL}\ \ldots\ \mathsf{L}}^{k} \mathrel{\hat{=}} 2^k -$ $- 1$, выводит за рамки k разрядов, можно, чтобы результат оставался в той же области, вычеркнуть „лишний" первый бит (бит „переполнения"). Это приводит к k-разрядному ***циклическому двоичному счётчику;*** например, для $k = 4$ мы получаем подпрограмму, представленную на рис. 165[1]. Чтобы теперь перейти к общепринятой комбинационной схеме, рассмотрим компоненты

$$\begin{aligned} u_1 &= last(u), \\ u_2 &= last(lead(u)), \\ u_3 &= last(lead(lead(u))), \\ u_4 &= last(lead(lead(lead(u)))) \end{aligned}$$

параметра u и компоненты s_1, s_2, s_3, s_4 результата как самостоятельные переменные.

Из исходной (нециклической) рекурсии ясно, что для первого u_i, отличного от L, мы имеем $s_{i-1} = s_{i-2} = \ldots = s_1 = \mathsf{O}$, значением же s_i будет L, а $(s_{i+1}, s_{i+2}, \ldots)$ совпадает с $(u_{i+1}, u_{i+2}, \ldots)$. Таким образом, нам нужно просто в каждом

[1] Где *cyclesucc* — от *cycle* (цикл). — *Прим. изд. ред.*

разряде реализовать функцию полусумматора. Для $k = 4$ получается комбинационная схема на рис. 166, где для единообра-

```
funct cyclesucc ≡ (bits u
          co length(u) = 4 co) bits:

  if last(u) = O
    then lead(u) + ⟨L⟩
  elsf last(lead(u)) = O
    then
      lead(lead(u)) + ⟨L⟩ + ⟨O⟩
  elsf last(lead(lead(u))) = O
    then
      lead(lead (lead(u))) + ⟨L⟩ + ⟨O⟩ + ⟨O⟩

  elsf last(lead(lead(lead(u)))) = O
    then
      ⟨L⟩ + ⟨O⟩ + ⟨O⟩ + ⟨O⟩
    else
      ⟨O⟩ + ⟨O⟩ + ⟨O⟩ + ⟨O⟩          fi
```

```
function cyclesucc(u : bitstring
                   {length(u) = 4}) : bitstring;
begin
  if last(u) = O
    then cyclesucc ← postfix(lead(u), L)
  else if last(lead(u)) = O
    then cyclesucc ←
      postfix(postfix(lead(lead(u)), L), O)
  else if last(lead(lead(u))) = O
    then cyclesucc ←
      postfix(postfix(postfix(lead(lead
                  (lead(u))), L), O), O)
  else if last(lead(lead(lead(u)))) = O
    then cyclesucc ← postfix(postfix(postfix(
                  postfix(empty, L), O), O), O
    else cyclesucc ← postfix(postfix(postfix(
                  postfix(empty, O), O), O), O)
end
```

Рис. 165. Комбинационная схема четырёхразрядного (циклического) двоичного счётчика.

зия последний разряд также представлен полусумматором, в котором второй вход c постоянно установлен на L.

Отметим, что в случае, когда на ***счётном входе*** c подаётся O вместо L, мы получаем тождественное отображение. ***Выход переноса*** $\ddot{u}$ при использовании данной схемы как циклического двоичного счётчика является лишним; однако если объединить две та-

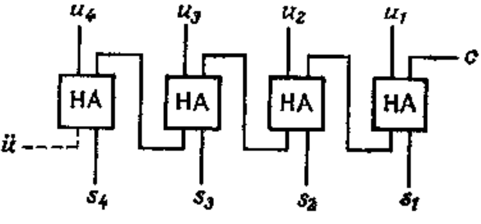


Рис. 166.

кие схемы вместе так, чтобы выход переноса одной был связан со счётным входом другой, то снова получится схема циклического двоичного счётчика. Таким образом можно сложить схему циклического двоичного счётчика с большим числом разрядов из „элементарных кирпичиков“, например из четырёх- или восьми-

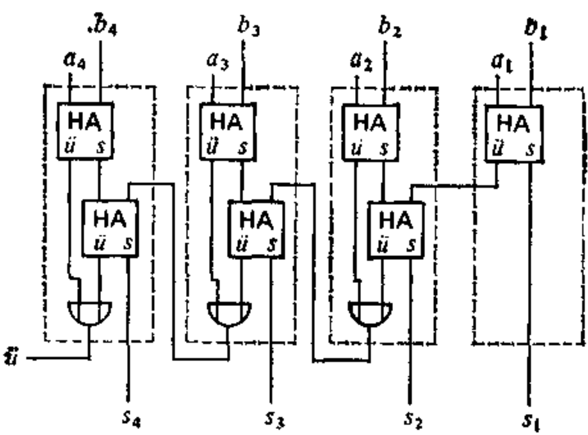


Рис 167. Четырёхразрядная комбинационная схема сложения.

разрядных счётчиков. Полусумматор — это предельный случай одноразрядного двоичного счётчика.

Аналогичным способом может быть получена k-разрядная комбинационная схема для операции перехода к предшествующему числу, равно как и для сложения; так, из алгоритма *binadd* (см. 4.2.2.2) для $k = 4$ получается комбинационная схема сложения, показанная на рис. 167. Если дополнить её ещё одним полусумматором в разряде единиц и снабдить дополнительным счётным входом c (рис. 168), то мы снова получим „кирпичики", из которых можно складывать многоразрядные

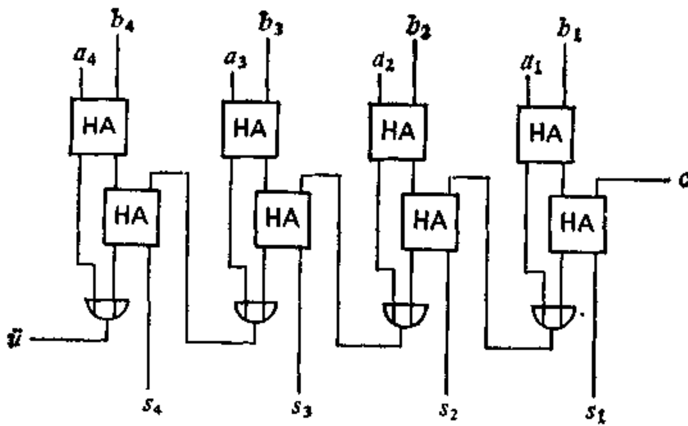


Рис. 168. Четырёхразрядный „кирпичик" комбинационной схемы сложения.

комбинационные схемы суммирования. Полный сумматор — это предельный случай одноразрядной комбинационной схемы суммирования.

В случае вычитания можно исходить из алгоритма *binsub* (см. 4.2.2.2), однако k-разрядную комбинационную схему вычитания можно получить и вычислив сначала обратный код (двоичное дополнение со сдвигом) для вычитаемого (см. приложение А), а затем произведя сложение. Для компенсации необходимо добавить единицу в разряде единиц, стало быть, вход счёта надо установить на **L** („возврат" единицы, *англ.*: *end-around carry*[1]).

Во всех этих примерах комбинационная схема получалась как результат „развёртывания" рекурсии в пространстве. Они показывают, что комбинационные схемы можно „вычислять" — наблюдение, весьма важное с точки зрения получения корректных схем на основе „разумных" спецификаций.

Если, опираясь на комбинационные схемы k-разрядного сложения и вычитания, аналогичным образом реализовать операции умножения или деления, то схемы получаются довольно громоздкие. Впрочем, комбинационные схемы для умножения, соответственно деления, двух восьмиразрядных или шестнадцатиразрядных двоичных чисел умещаются сегодня — в рамках технологии сверхбольших интегральных схем — на крохотном кристалле („чипе").

4.3. Переключательные схемы

Для переключательных функций и комбинационных схем существенно то, что переменные могут принимать различные значения. Изучая вопрос о том, как реализовать их с помощью физических устройств, мы до сих пор рассматривали комбинационные схемы лишь в статике, т. е. в состоянии покоя или „успокоения". Однако изменение значений переменных, которое в обычных реализациях проявляется как изменение во времени, не происходит мгновенно; представление о том, что переключательная переменная мгновенно перепрыгивает от значения **O** к значению **L** или обратно, является сугубо идеализированным. На самом же деле значение параметра соответствующего сигнала изменяется не скачком. Это означает, что лишь по истечении некоторого определённого времени δt достоверно могут быть определены значения на выходе комбинационной схемы, а в „переходный" период эти значения вообще не определены. В частности, один из аспектов технической реализации переключательных переменных — это задание так на-

---

[1] Буквально: переносить из конца обратно (в начало). — *Прим. изд. ред.*

зываемых ***моментов опроса***, или ***точек отсчёта времени*** (см. ниже).

Разумеется, ***время срабатывания*** δt зависит от уровня развития техники; если в пятидесятые годы для отдельного И-элемента, реализованного с помощью релейных устройств, оно составляло ещё 10^{-2} с, то сегодня благодаря применению транзисторов в интегральных схемах это время уже меньше 10^{-9} с. Для сложных комбинационных схем время срабатывания получается по существу как суммарное время прохождения сигнала через отдельные элементы схемы.

Из сказанного прежде всего следует, что при фактическом построении переключательных схем в качестве одного из важных факторов надо принимать во внимание временну́ю задержку. Однако при чисто функциональном описании мы можем продолжать условно считать, что в комбинационных схемах временно́й задержки нет. То обстоятельство, что в течение некоторого времени выход комбинационной схемы не определён, обычно учитывают, вводя *временну́ю развёртку*, а именно фиксируя (равноотстоящие) *точки отсчёта времени*. Всякая переключательная переменная рассматривается лишь в точках отсчёта времени.

Поставленные перед необходимостью считаться с отличными от нуля временами переключения, мы „делаем из нужды добродетель", переходя в духе машинно-ориентированного подхода к переменным памяти (см. 3.6.4.3) для двоичных слов и символов.

4.3.1. Переменные памяти для двоичных слов

Используя переменные памяти для двоичных слов, можно повторительную рекурсию выполнять в форме простых повторений. Классическим примером их применения служит так называемый последовательный сумматор (см. 4.3.2.1).

4.3.1.1. Пример: сложение

При введении „переноса" итеративная версия „быстрого" сложения из 4.2.2.2 принимает вид, показанный на рис. 169[1]. Если значения переменной $\ddot{u}$ включить в разбор случаев, то после некоторых упрощений придём к подпрограмме, представленной на рис. 170. Переходя к двоичному представлению, получаем для *k*-разрядной циклической двоичной арифметики

---

[1] Корректность этой программы следует (см. 3.3.3 4) из того, что выражение $z + t \times (a + b + \ddot{u})$ при повторениях остаётся неизменным; но вначале оно равно $0 + 1 \times (A + B + 0) = A + B$, а в конце $z + t \times (0 + 0 + 0) = z$. То что окончание обеспечено, очевидно.

```
funct add ≡ (int A, B
             co A ≧ 0 ∧ B ≧ 0 co) int:

⌈(var int a, b, z, ü, t) := (A, B, 0, 0, 1);
 while a ≠ 0 ∨ b ≠ 0 ∨ ü ≠ 0
 do (a, b, z, ü, t) :=
    if odd a ∧ odd b
    then
     ((a − 1) ÷ 2, (b − 1) ÷ 2, z + t × ü, 1, 2 × t)
    ▯ ¬odd a ∧ odd b
    then
     (a ÷ 2, (b − 1) ÷ 2, z + t × ü + t, 0, 2 × t)
    ▯ odd a ∧ ¬odd b
    then
     ((a − 1) ÷ 2, b ÷ 2, z + t × ü + t, 0, 2 × t)
    ▯ ¬odd a ∧ ¬odd b
    then
     (a ÷ 2, b ÷ 2, z + t × ü, 0, 2 × t)    fi od;
 z                                            ⌋
```

```
function add(A, B : integer
             {(A ≧ 0) and (B ≧ 0)}) : integer;
  var a, b, z, ü, t : integer;
begin
  (a, b, z, ü, t) := (A, B, 0, 0, 1);
  while (a ≠ 0) or (b ≠ 0) or (ü ≠ 0)
    do
    if odd(a) and odd(b)
    then (a, b, z, ü, t) :=
      ((a − 1) div 2, (b − 1) div 2, z + t*ü, 1, 2*t)
    ▯ not odd(a) and odd(b)
    then (a, b, z, ü, t) :=
      (a div 2, (b − 1) div 2, z + t*ü + t, 0, 2*t)
    ▯ odd(a) and not odd(b)
    then (a, b, z, ü, t) :=
      ((a − 1) div 2, b div 2, z + t*ü + t, 0, 2*t)
    ▯ not odd(a) and not odd(b)
    then (a, b, z, ü, t) :=
      ((a div 2, b div 2, z + t*ü, 0, 2*t);
  add ⇐ z
end
```

Рис. 169.

```
funct add ≡ (int A, B
             co A ≧ 0 ∧ B ≧ 0 co) int:

⌈(var int a, b, z, ü, t) := (A, B, 0, 0, 1);
 while a ≠ 0 ∨ b ≠ 0 ∨ ü ≠ 0
 do (a, b, z, ü, t) :=
    if odd a ∧ odd b ∧ ü = 1
    then
      ((a − 1) ÷ 2, (b − 1) ÷ 2, z + t, 1, 2 × t)
    ▯ odd a ∧ odd b ∧ ü = 0

    then
      ((a − 1) ÷ 2, (b − 1) ÷ 2, z, 1, 2 × t)
    ▯ ¬odd a ∧ odd b ∧ ü = 1
    then
      (a ÷ 2, (b − 1) ÷ 2, z, 1, 2 × t)
    ▯ ¬odd a ∧ odd b ∧ ü = 0
    then
      (a ÷ 2, (b − 1) ÷ 2, z + t, 0, 2 × t)
    ▯ odd a ∧ ¬odd b ∧ ü = 1
    then
      ((a − 1) ÷ 2, b ÷ 2, z, 1, 2 × t)
    ▯ odd a ∧ ¬odd b ∧ ü = 0
    then
      ((a − 1) ÷ 2, b ÷ 2, z + t, 0, 2 × t)
    ▯ ¬odd a ∧ ¬odd b ∧ ü = 1
    then
      (a ÷ 2, b ÷ 2, z + t, 0, 2 × t)
    ▯ ¬odd a ∧ ¬odd b ∧ ü = 0
    then
      (a ÷ 2, b ÷ 2, z, 0, 2 × t)          fi od;
 z                                            ⌋
```

```
function add(A, B : integer
             {(A ≧ 0) and (B ≧ 0)}) : integer;
  var a, b, z, ü, t : integer;
begin
  (a, b, z, ü, t) := (A, B, 0, 0, 1);
  while (a ≠ 0) or (b ≠ 0) or (ü ≠ 0) do

    if odd(a) and odd(b) and (ü = 1)
    then (a, b, z, ü, t) :=
      ((a − 1) div 2, (b − 1) div 2, z + t, 1, 2*t)
    ▯ odd(a) and odd(b) and (ü = 0)
    then (a, b, z, ü, t) :=
      ((a − 1) div 2, (b − 1) div 2, z, 1, 2*t)
    ▯ not odd(a) and odd(b) and (ü = 1)
    then (a, b, z, ü, t) :=
      (a div 2, (b − 1) div 2, z, 1, 2*t)
    ▯ not odd(a) and odd(b) and (ü = 0)
    then (a, b, z, ü, t) :=
      (a div 2, (b − 1) div 2, z + t, 0, 2*t)
    ▯ odd(a) and not odd(b) and (ü = 1)
    then (a, b, z, ü, t) :=
      ((a − 1) div 2, b div 2, z, 1, 2*t)
    ▯ odd(a) and not odd(b) and (ü = 0)
    then (a, b, z, ü, t) :=
      ((a − 1) div 2, b div 2, z + t, 0, 2*t)
    ▯ not odd(a) and not odd(b) and (ü = 1)
    then (a, b, z, ü, t) :=
      (a div 2, b div 2, z + t, 0, 2*t)
    ▯ not odd(a) and not odd(b) and (ü = 0)
    then (a, b, z, ü, t) :=
      (a div 2, b div 2, z, 0, 2*t);
  add ⇐ z
end
```

Рис. 170.

```
funct binadd = (bits A, bits B
        co length(A) = length(B) co) bits:


Γ(var bits a, b, z, var bit ü) := (A, B, ◇, O);
 while a ≠ ◇
 do (a, b, z, ü) :=
    if last(a) = L ∧ last(b) = L ∧ ü = L
    then
      (lead(a), lead(b), ⟨L⟩ + z, L)
    ▯ (last(a) = L ∧ last(b) = L ∧ ü = O)
     ∨ (last(a) = O ∧ last(b) = L ∧ ü = L)
     ∨ (last(a) = L ∧ last(b) = O ∧ ü = L)
    then
      (lead(a), lead(b), ⟨O⟩ + z, L)
    ▯ (last(a) = L ∧ last(b) = O ∧ ü = O)
     ∨ (last(a) = O ∧ last(b) = L ∧ ü = O)
     ∨ (last(a) = O ∧ last(b) = O ∧ ü = L)
    then
      (lead(a), lead(b), ⟨L⟩ + z, O)
    ▯ (last(a) = O ∧ last(b) = O ∧ ü = O)
    then
      (lead(a), lead(b), ⟨O⟩ + z, O) fi od;
 z                                          ⌋
```

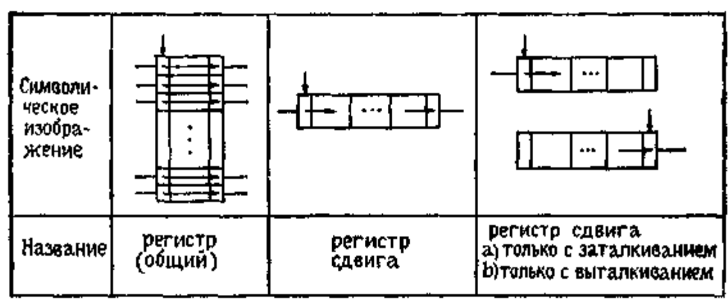
```
function binadd [illegible]
        [illegible]
  var a, b, z [illegible];
begin
  (a, b, z, ü) := (A, B, empty, O);
  while not isempty(a) do

  if (last(a) = L) and (last(b) = L) and (ü = L)
  then (a, b, z, ü) :=
     (lead(a), lead(b), prefix(L, z), L)
  ▯ ((last(a) = L) and (last(b) = L) and (ü = O)) or
     ((last(a) = O) and (last(b) = L) and (ü = L)) or
     ((last(a) = L) and (last(b) = O) and (ü = L))
  then (a, b, z, ü) :=
     (lead(a), lead(b), prefix(O, z), L)
  ▯ ((last(a) = L) and (last(b) = O) and (ü = O)) or
     ((last(a) = O) and (last(b) = L) and (ü = O)) or
     ((last(a) = O) and (last(b) = O) and (ü = L))
  then (a, b, z, ü) :=
     (lead(a), lead(b), prefix(L, z), O)
  ▯ (last(a) = O) and (last(b) = O) and (ü = O)
  then (a, b, z, ü) :=
     (lead(a), lead(b), prefix(O, z), O);
  binadd := z
end
```

Рис. 171.

подпрограмму *binadd*, приведённую на рис. 171 (см. 4.2.3; *ü* содержит в конце возможный „бит переполнения").

4.3.1.2. Двоичные регистры и элементы задержки

Переменные памяти для двоичных слов называются обычно ***регистрами*** (***задержки***) (рис. 172). Используемые в *binadd* при построении *a*, *b* и построении *z* переменные для двоичных слов с типичными „разовыми" операциями $a := lead(a)$ и $z :=$



Символическое изображение			
Название	регистр (общий)	регистр сдвига	регистр сдвига а) только с заталкиванием b) только с выталкиванием

Рис. 172. Символические изображения регистров.

$\langle x \rangle + z$ соответственно называются ***регистрами сдвига***; в первом случае имеем „выталкивание" из регистра сдвига, во втором случае — „заталкивание" в регистр. Что касается $\ddot{u}$, то это переменная для двоичного знака — двоичный (*тактированный*, или *управляемый тактовыми импульсами*) ***элемент задержки***

Символическое изображение	a ... r
Формула	$r = Da$
Название	элемент задержки (управляемый тактовыми импульсами)

Рис. 173. Символическое изображение элемента задержки.

(рис. 173). Там, где программист видит присваивание $r := a$, инженеры пишут $r = Da$ У собственно элемента задержки „хватает памяти" только до следующей точки отсчёта времени.

4.3.1.3. Присоединение элементов задержки к логическим элементам

Для соединений элементов задержки с логическими элементами справедливо несколько правил упрощения. В частности,

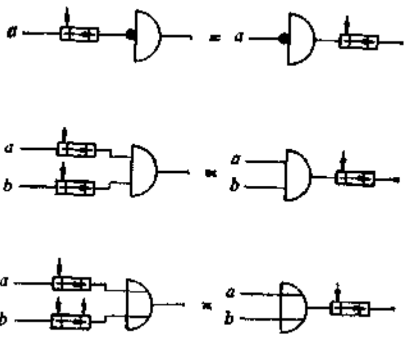


Рис. 174. „Протаскивание" элемента задержки через логические элементы.

справедливы закон коммутативности

$$\neg (Da) = D(\neg a)$$

и законы дистрибутивности

$$(Da) \wedge (Db) = D(a \wedge b),$$
$$(Da) \vee (Db) = D(a \vee b)$$

(см. рис. 174).

4.3.2. Построение переключательных схем

Структуру, содержащую, с одной стороны, переменные памяти для двоичных слов и двоичных знаков, т. е. регистры и элементы задержки (рис. 172, 173), а с другой — комбинационную схему, входы которой „питаются" от регистров и элементов задержки и выходы которой задают новые установки регистрам и элементам задержки, мы назовем ***переключательной схемой.*** Такая переключательная схема работает *тактами,* при этом предполагается, что длительность такта велика по сравнению с временем срабатывания комбинационной схемы. Такая ***тактированная*** переключательная схема получает ***тактовые импульсы*** либо от некоторых центральных часов (***синхронное тактирование***), либо от других переключательных схем (***асинхронное тактирование***).

Регистры и элементы задержки переключательной схемы могут находиться в конечном числе ***внутренних состояний***[1], переключательная схема вызывает ***переходы состояний***[2]. Эти переходы зависят также от ***входных параметров***; некоторые определённые функции состояния передаются „наружу" как результаты работы схемы.

4.3.2.1. Последовательный и параллельный сумматоры

На рис. 175 изображена переключательная схема для подпрограммы *binadd* из 4.3.1.1. Используется один полный сумматор, таблица значений которого (рис. 156) соответствует восьмичленному разбору случаев в *binadd.* Элемент задержки для переноса *ü* имеет два внутренних состояния со значениями $\mathbf{L} \mathrel{\widehat{=}}$ »перенос есть« и $\mathbf{O} \mathrel{\widehat{=}}$ »переноса нет«.

В то время как комбинационная схема сложения (рис. 167) выполняет сложение „одним махом"[3], последовательный сумматор требует для случая n-разрядных слагаемых n единиц времени (тактов). В известном смысле последовательный сумматор более „способен", чем комбинаторная схема сложения: число разрядов слагаемых не ограничено[4], так что сложить можно любую пару чисел; арифметика натуральных чисел ограничена лишь длиной регистра сдвига.

---

[1] Состояния в смысле раздела 3.3.2.1.

[2] В гл. 7 эта тема получит дальнейшее развитие в понятии конечного автомата.

[3] Строго говоря, это не так, поскольку перенос может „пробежать" все разряды. Однако этого можно избежать с помощью особых мер

[4] Нельзя сказать, что оно „бесконечно". Числа с бесконечным числом разрядов вообще не могут быть сложены с помощью оканчивающегося алгоритма.

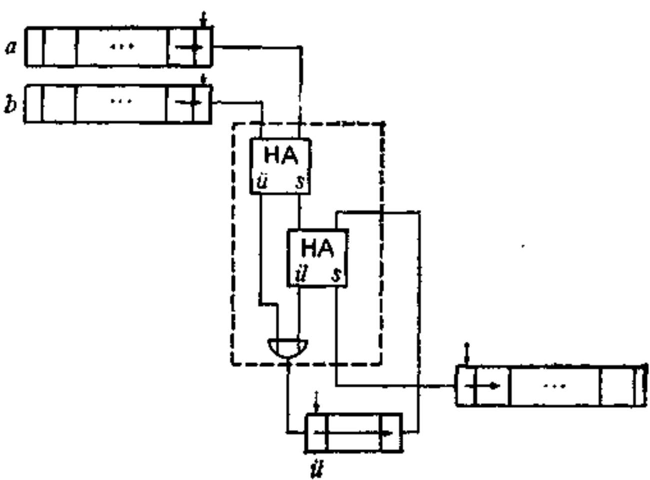


Рис. 175. Переключательная схема последовательного сумматора.

Переключательная схема с m двоичными элементами задержки может иметь до 2^m внутренних состояний, по числу 2^m возможных комбинаций значений. Примером с максимальным числом внутренних состояний служит переключательная схема ***двоичного счётчика*** (рис. 176). Она получается, если соединить выходы комбинационной схемы двоичного счётчика (рис. 165) с элементами задержки, а выходы этих элементов — с входами комбинационной схемы.

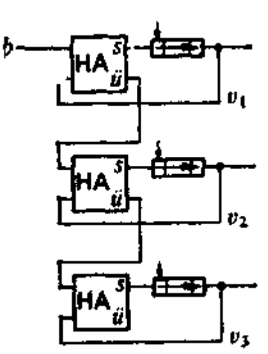


Рис. 176. Переключательная схема двоичного счётчика.

Другой пример получается аналогичным образом из комбинационной схемы сложения (рис. 168). Отдельные элементы задержки образуют совместно регистр — ***накапливающий регистр (накопитель)*** параллельного сумматора (рис. 177). Для сложения счётный вход c постоянно установлен на O.

Параллельный (четырёхразрядный) сумматор на рис. 177 может сложить лишь два четырёхразрядных числа и в случае переполнения установить выход переноса $ü$ на L. Опять мы имеем „кирпичик"; если связать $ü$-выход такого параллельного сумматора с c-входом другого, то получится параллельный сумматор с соответственно бо́льшим накопителем.

Фактически этим способом всегда может быть реализовано лишь фиксированное число разрядов. По образцу переключа-

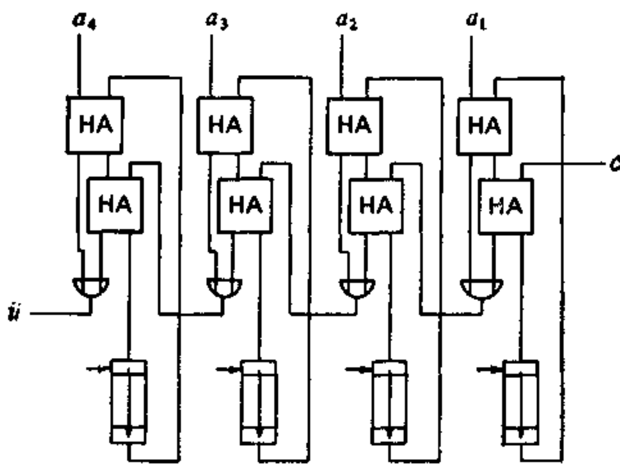


Рис. 177. Параллельный сумматор с накопителем.

тельной схемы на рис. 175 это ограничение можно всё-таки обойти, заменив фигурирующий там полный сумматор на k-разрядную комбинационную схему сложения. В таком случае слагаемые a и b также подготавливаются ***последовательно-параллельно*** в двух k-элементных ***батареях*** регистров сдвига. На рис. 178

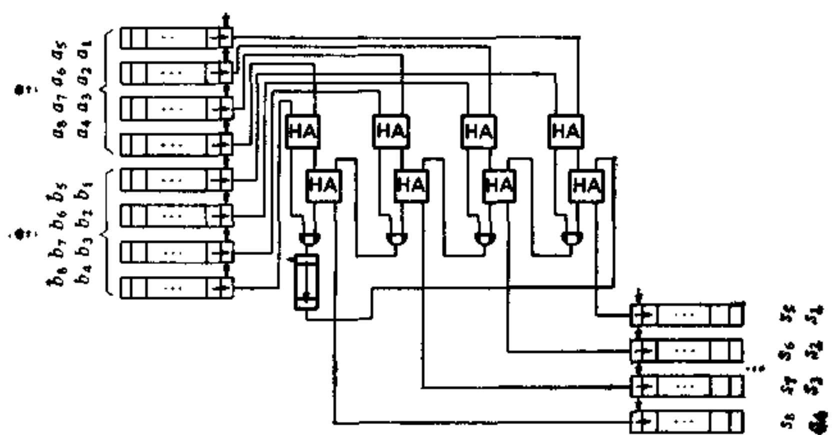


Рис. 178. Четырёхразрядный последовательно-параллельный сумматор.

представлена переключательная схема такого ***последовательно-параллельного сумматора*** для $k=4$. Снова арифметика натуральных чисел ограничена лишь длиной регистра сдвига.

4.3.2.2. Переключательная схема сдвига

В других переключательных схемах используются не все комбинации значений. Примером может служить ***переключательная схема сдвига***, применяемая для порождения трёхразрядного циклического кода (см. 1.4.2). Она представлена на рис. 179, где

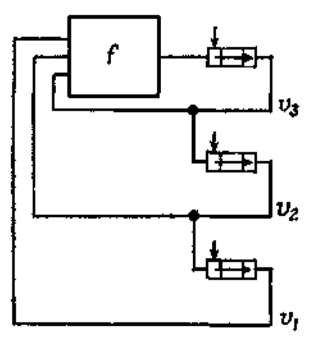


Рис. 179. Порождение циклического кода.

$$v_1 = Dv_2, \quad v_2 = Dv_3,$$
$$v_3 = Df(v_1, v_2, v_3),$$

или, в „программистской" записи,

$$(v_1, v_2, v_3) :=$$
$$(v_2, v_3, f(v_1, v_2, v_3)).$$

Это переключательная схема без входов, её внутренние состояния будут (рано или поздно) периодически повторяться. Для случаев

(A) $f(v_1, v_2, v_3) = v_1$,
(B) $f(v_1, v_2, v_3) = \neg v_1$,
(C) $f(v_1, v_2, v_3) = (v_1 \wedge v_2) \vee (\neg v_1 \wedge \neg v_2) = v_1 \leftrightarrow v_2$

будут пробегаться следующие комбинации значений (v_1, v_2, v_3):

(A)
↱(OOO)↲

↱(OOL)
(OLO)
(LOO)↲

↱(OLL)
(LLO)
(LOL)↲

↱(LLL)↲

(B)
↱(OOO)
(OOL)
(OLL)
(LLL)
(LLO)
(LOO)↲

↱(LOL)
(OLO)↲

(C)
↱(OOO)
(OOL)
(OLL)
(LLO)
(LOL)
(OLO)
(LOO)↲

↱(LLL)↲

В случае (C) получаются цикл ↱OOOLLOL↲ и тривиальный цикл ↱L↲.

Этот пример показывает также, каким образом можно построить регистр сдвига, „нанизывая“ один за другим элементы задержки.

4.3.3. Триггеры

Переключательная схема называется ***мультистабильной***, если для неё существует такая комбинация входных значений, которая оставляет неизменным любое её внутреннее состояние.

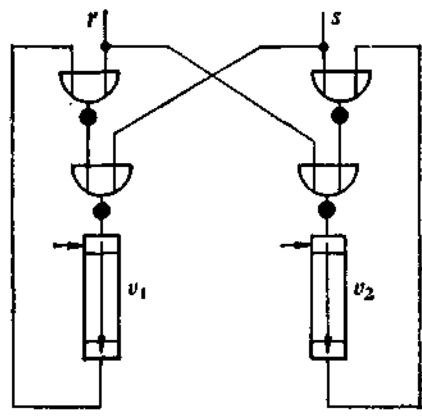


Рис. 180. Управляемый тактовыми импульсами RS-триггер как переключательная схема.

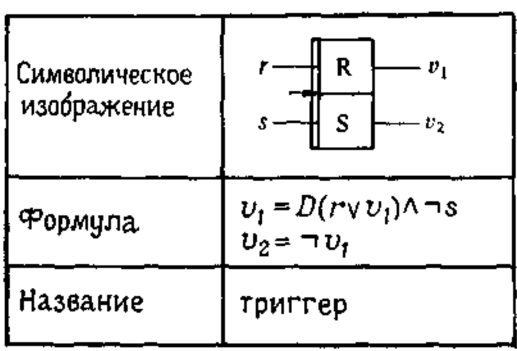
Такую переключательную схему можно использовать для запоминания. Основополагающим благодаря своей простоте является ***триггер***[1], который ***бистабилен***: у него два внутренних состояния; их можно характеризовать, скажем, значениями одной внутренней двоичной переменной v или двух переменных (v_1, v_2), которые могут принимать лишь комбинации значений (O,L) и (L,O), так что всегда выполнено условие $v_2 = \neg v_1$.

Самый распространённый триггер (так называемый „RS-триггер“[2]) имеет два симметричных входа (r, s), причём комбинация (L,L) исключается. Для комбинации входных значений (O,O) внутреннее состояние остаётся неизменным;

---

[1] В оригинале Flipflop — заимствованное из английского звукоподражательное слово (flip — щёлкать, flop — хлопаться, flap — хлопать, flip-flap — хлопанье). В немецком языке используется также термин Trigger, тоже английского присхождение (trigger — защёлка, спусковой крючок). — *Прим. изд. ред.*

[2] S и R — соответственно от английских set (установка) и reset (сброс). — *Прим. изд. ред.*



Символическое изображение	
Формула	$v_1 = D(r \vee v_1) \wedge \neg s$ $v_2 = \neg v_1$
Название	триггер

Рис. 181. Символическое изображение управляемого тактовыми импульсами RS-триггера.

Рис. 182. Реализация элемента задержки с помощью триггера.

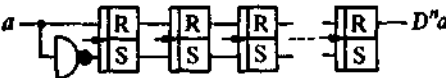


Рис. 183. Реализация регистра сдвига с помощью цепочки триггеров.

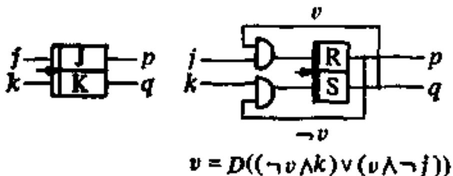


Рис. 184. JK-триггер.

для $(r, s) = (\mathsf{L}, \mathsf{O})$ будет $(v_1, v_2) = (\mathsf{L}, \mathsf{O})$, а для $(r, s) = (\mathsf{O}, \mathsf{L})$ будет $(v_1, v_2) = (\mathsf{O}, \mathsf{L})$.

Поведение триггера описывается двумя симметричными между собой равенствами

$$v_1 = D((r \vee v_1) \wedge \neg s),$$
$$v_2 = D((s \vee v_2) \wedge \neg r)$$

с дополнительным условием $v_1 = \neg v_2$, которые означают ровно то же, что и групповое присваивание

$$(v_1, v_2) := ((r \vee v_1) \wedge \neg s,\ (s \vee v_2) \wedge \neg r)$$

с условием-предохранителем $r \wedge s = \mathsf{O}$.

На рис. 180 изображена соответствующая переключательная схема с двумя элементами задержки, а на рис. 181 представлено символическое изображение триггера.

По техническим причинам как отдельные элементы задержки, так и целые регистры экономичнее всего реализовывать с помощью триггеров или соответственно батарей триггеров (***триггерные регистры***). Для отдельного элемента задержки соответствующая схема изображена на рис. 182; комбинация (L, L) здесь автоматически исключается. В случае регистра сдвига выходы одного триггера используют в качестве входов следующего (рис. 183).

Во всех случаях на входе каждого триггера должно выполняться условие $s \wedge r = \mathsf{O}$. В случае необходимости его можно обеспечить, как на

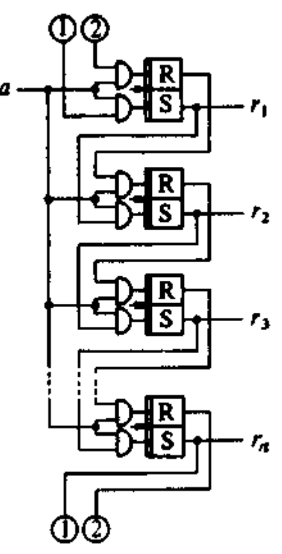


Рис. 185. Кольцевой счётчик на триггерах.

рис. 184, за счёт обратных связей и включения конъюнкций перед триггерами („JK-триггер"). Иногда работают также с ***двойным сигналом***, т. е. вместе со всякой переключательной переменной поступает и её отрицание, тогда на входах и выходах RS-триггеров всегда будут нужные пары. Примерами служат цепочка триггеров на рис. 183, а также приведённый на рис. 185 ***кольцевой*** (***циклический***) ***счётчик***, в котором „циркулирует" пара (O, L).

При двойном сигнале комбинация значений (O, O) никогда не появляется на входе триггера. Однако её использование часто оказывается выгодным (см. рис. 184).

4.3.4. Триггерные переключательные схемы

Переключательная схема на триггерах, или ***триггерная переключательная схема***, — это комбинационная схема, выходы которой через триггеры соединяются обратной связью со входами.

В принципе обычную переключательную схему можно сразу заменить триггерной, поскольку в соответствии с рис. 182 каждый элемент задержки можно заменить на триггер. Дополнительно требуемое на входе отрицание обеспечивается за счёт того, что выходом мы располагаем вместе с его отрицанием. Последнее обстоятельство позволяет, в частности, осуществлять непосредственное взаимодействие переключательных схем на триггерах с двухэтапными перекодировщиками рассмотренного в 4.1.5.4 вида. В общем случае отрицание можно „протаскивать" по схеме, используя законы де Моргана, и тем самым строить любую переключательную схему на триггерах без привлечения отрицания либо же строить двухкаскадные схемы из одних NOR-элементов.

4.3.4.1. Триггерные переключательные схемы для арифметики

В качестве примера триггерной переключательной схемы рассмотрим изображённую на рис. 186a схему параллельного сумматора с триггерным накопителем; обратите внимание, что полусумматоров здесь используется в два раза меньше, чем в обычной переключательной схеме (рис. 177). На рис. 186b показано, как можно дополнить эту схему комбинационной схемой сдвига и тем самым сделать пригодной и для умножения. С помощью команд сложения или сдвига открываются нужные для выполнения соответственно сложения или сдвига пути („шины"); при отсутствии новых команд в точке отсчета времени содержимое накапливающего регистра сохраняется неизменным.

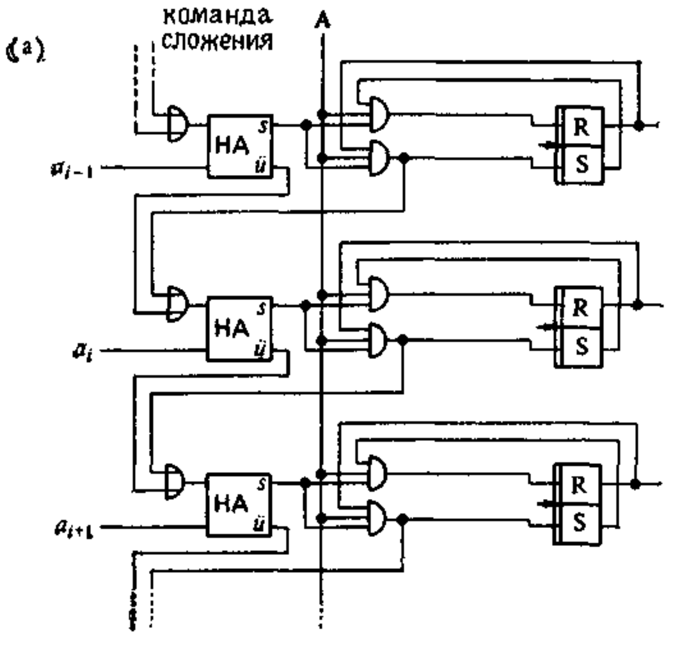
Умножение на натуральное число можно реализовать как повторное сложение со сдвигом разрядов. Так как множимое должно быть доступно в продолжение всего процесса умножения, то его, как правило, помещают в специальный ***регистр множимого*** („MD-регистр"[1]). Схематично это показано на рис. 187. AC-регистр[2], накопитель, удлиняется вправо (вместе со схемой сдвига, но без схемы сложения) и принимает в конце всё произведение. Командой *B* (как на рис. 186b) открывают шины, нужные для сдвига[3], командой *A* открывают шины от MD-регистра в комбинационную схему сложения; управляются команды И-элементом из позиции множителя. Множитель сам может располагаться в некотором регистре сдвига („MR-регистре"), тогда очередная используемая позиция всегда находится в правом конце этого регистра и оттуда считывается.

---

[1] MD — от латинского multiplicand (множимое), а появляющееся ниже MR — от mutliplicator (множитель). — *Прим. изд. ред.*

[2] AC — от английского accumulator (накопитель). — *Прим. изд. ред.*

[3] Сдвиг вправо с заполнением освободившихся мест „нулями" O.



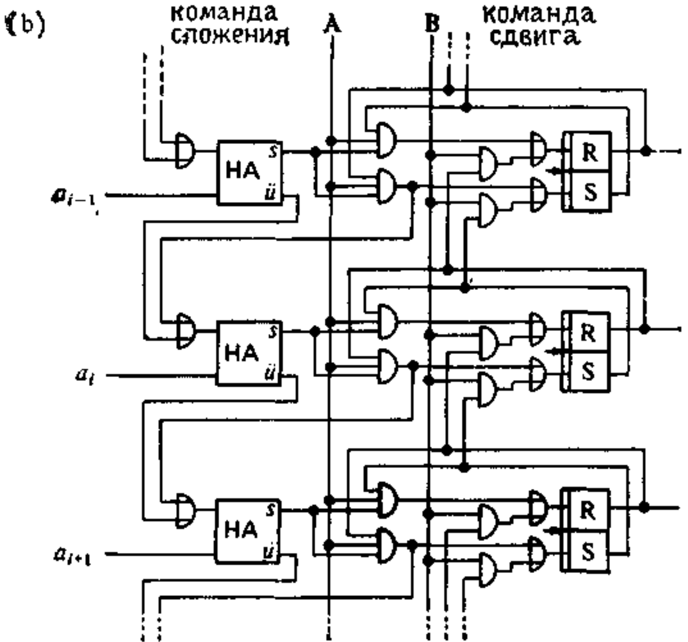


Рис. 186. Фрагмент схемы триггерного параллельного сумматора (a) без комбинационной схемы сдвига, (b) со схемой сдвига.

Из соображений экономии часто в качестве MD-регистра используется правая, первоначально незаполненная половина AC-регистра.

Для выполнения вычитания достаточно, как было сказано в 4.2.3, образовать в MD-регистре обратный код вычитаемого, т. е. подключиться к выходам-отрицаниям триггеров MD-регистра и снабдить комбинационную схему сложения (рис. 167)

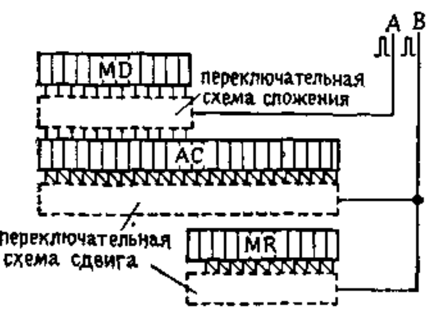


Рис. 187. Реализация последовательного умножения с помощью параллельного сложения.

полным сумматором также и для разряда единиц, причем на вход переноса этого сумматора при сложении будем подавать O, а при вычитании L.

Деление очевидным образом сводится к повторному вычитанию и сдвигу.

Для этого нужно попеременно уменьшать и увеличивать (на единицу) содержимое регистра частного, в котором формируется частное, а управление процессом вычитания и сложения должно учитывать знак остатка. При этом остаток обычно помещается в AC-регистре, делитель — в MD-регистре, а частное формируется в MR-регистре или, ради экономии, в освобождающейся части AC-регистра. В остальном по сравнению со схемой умножения ничего не меняется.

В настоящее время при реализации операций монтажными схемами почти всегда ограничиваются четырьмя основными арифметическими действиями.

4.3.4.2. Триггерные переключательные схемы для неарифметических операций

Различные части триггерных переключательных схем, построенных для реализации базовых операций арифметики, можно использовать и для реализации операций над нечисловыми объектами, при условии что мы всё время работаем с

одним и тем же числом разрядов. ***Ширина обработки***, т. е. число битов, которые обрабатываются в данной переключательной схеме, часто совпадает с числом битов, которые при нормальном способе работы сообща извлекаются из памяти (***ширина памяти***, на жаргоне — „длина слова"). Ширина памяти составляет иногда лишь 4, 6 или 8 бит и по порядку величины соответствует обычно представлению отдельных знаков (сорта **char**, соотв. *char*).

Наборы из шести или восьми битов часто называют ***байтами***. Впрочем, сегодня имеются уже микропроцессоры с шириной обработки 16 бит, а скоро появятся и 32-битовые. У больших вычислительных машин ширина обработки составляет 24, 32, 48, 64 бит.

Если наряду с комбинационной схемой сложения предусмотреть ещё и комбинационные схемы для поразрядной реализации конъюнкции и дизъюнкции, то можно выполнять (поразрядно) и булевы операции (см. табл. 11, группы 2 и 3). С помощью ***масок***, т. е. слов, нужные разряды которых заполнены знаком O, можно, используя конъюнкцию, ***очистить*** не интересующую нас часть слова. Вместе со сдвигами и сложением это позволяет выполнять и конкатенацию произвольных последовательностей знаков.

4.3.5. Техническая реализация переключательных схем

Триггеры сегодня строятся почти исключительно на интегральных транзисторных переключателях. Время срабатывания для триггеров в интегральных схемах — порядка 10^{-9} с. Недавно на базе стандартной микроэлектронной технологии освоено[1] промышленное производство больших блоков памяти ёмкостью 2^{18} бит по цене 100 марок за штуку.

Технически триггер реализуется чаще всего при помощи двух запирающих друг друга переключательных элементов (схема Экклса — Джордана, 1919 г.). Звукоподражательное „флип-флоп"[2] призвано напоминать об „опрокидывании", „перекидке" из одного состояния в другое. На рис. 188 изображена принципиальная схема триггера на транзисторах.

В настоящее время применяются в основном ***импульсно-управляемые*** (т. е. управляемые тактовыми импульсами) триггеры, у которых поступившая на вход комбинация передаётся во внутреннее состояние только в точках отсчёта времени.

Весьма дешёвым, а потому долгое время преобладавшим при построении больших запоминающих устройств является использование для реализации триггеров магнитного материала

---

[1] В ФРГ — *Прим изд. ред.*

[2] См первое подстрочное примечание в разделе 4.3.3. — *Прим. изд. ред.*

с чётко выраженным, почти прямоугольным гистерезисом. Два возможных направления намагничивания дают нам два состояния триггера. Особенно часто употребляется кольцевой ферритовый сердечник, который находит применение в так называемых ***устройствах памяти на сердечниках***. Если запись и стирание осуществить здесь очень просто, то при считывании запомненного состояния возникают — в отличие от электронного

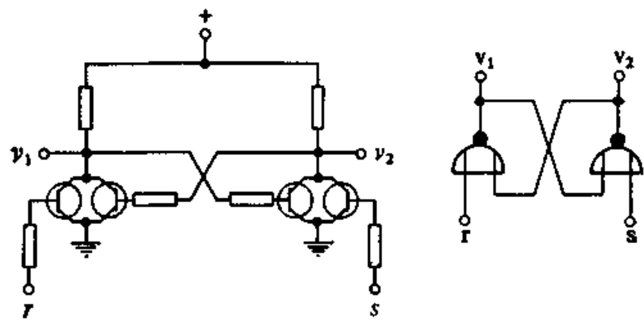


Рис. 188. Принципиальная схема триггера на *n-p-n*-транзисторах.

триггера — определённые трудности. Поэтому времена опросов относительно велики и составляют, как и времена срабатывания, величину порядка 10^{-7} с.[1]

Далее, под принцип действия триггера подпадает также всё ещё играющее важную роль запоминание на движущихся магнитных лентах. Более подробно об этом будет сказано в гл. 6.

4.4. Успехи и предельные возможности технологии

Поразительные успехи, которых достигла технология производства ЭВМ за последние три десятилетия, наиболее наглядно сказываются в размерах элементов переключательных схем. Отдельный элемент, например триггер, при ламповой технологии имел в 1952 г. размеры приблизительно $50 \times 60 \times 160$ (в мм) и занимал тем самым объём в $5 \cdot 10^{-4}$ м^3. Таким образом, плотность информации составляла не более $2 \cdot 10^3$ бит/м^3. А уже в 1975 г., используя технологию интегральных схем, на одном полупроводниковом кристалле размера $6 \times 3 \times 2$, т. е. в объёме $3.6 \cdot 10^{-8}$ м^3, размещали 1024 триггеров, что соответствует плотности информации в $3 \cdot 10^{10}$ бит/м^3. К 1985 г. на одной плате размера $7 \times 5 \times 0.5$, т. е. в объёме $2 \cdot 10^{-8}$ м^3, будут[2] размещать

[1] За дальнейшими подробностями отсылаем к специальной литературе (см., например, [41]).

[2] Напомним, что первая часть третьего немецкого издания вышла в 1982 г. — *Прим. изд. ред.*

2^{20} триггеров (что отвечает плотности информации 10^{13} бит/м^3), т. е. целый микропроцессор с шириной обработки 32 бит.

В технологии интегральных схем производство электронных элементов и групп осуществляется с помощью сложных процессов напыления и диффузионного внедрения. При этом для покрытия используются так называемые маски, которые должны воспроизводить детали с точностью до 1—3 мкм, а толщина эффективного слоя составляет всего 1 мкм. То есть это примерно те же размеры, что и у нервных клеток. Между тем уже достигнуты ширина эмиттера транзисторов, равная 1 мкм, и толщина базы, равная 0.25 мкм. Так как видимый свет лежит в диапазоне волн 0.4—0.8 мкм, то тем самым достигнута граница, обусловленная использованием оптических изображений. Для дальнейшего уменьшения размеров элементов памяти нужно применять электронно-оптические методы или работать с рентгеновскими лучами.

Почему же размеры вычислительных машин не уменьшились по сравнению с 1952 г. в 10^{10} раз? Прежде всего потому, что машины стали функционально богаче, а также потому, что остались прежними ограничения на зазоры между соединительными проводниками, а механические части и устройства питания едва ли занимают сейчас меньше места, чем раньше. Заметим, что из-за большей регулярности устройств памяти плотность их упаковки примерно в пять раз выше, чем у микропроцессоров.

Плотность, получаемая в более крупных единицах, в так называемых выводах интегральных схем, до сих пор составляет лишь сотую долю от плотности, достигаемой на полупроводниковом кристалле. Значительную часть места занимают корпуса полупроводниковых чипов и соединения между ними. Поэтому целью дальнейшего развития должно быть объединение в одной интегральной схеме не тысяч логических элементов и элементов памяти, а миллионов и более („большая интеграция").

Однако здесь есть предел, связанный с отводом тепла. Выделение тепла не должно превышать 10 Вт на „кирпичик". К 1985 г. следует рассчитывать на ЭВМ с 10^8 элементами (без памяти). На элементарный переключательный элемент в зависимости от технологии приходится мощность тепловых потерь от 10^{-6} до 10^{-4} Вт. Дальнейшее увеличение плотности упаковки возможно лишь при одновременном снижении потребляемой мощности.

Применение КМОП-технологии[1] сводит тепловые потери на элемент памяти к 10^{-7} Вт, и указанную проблему, по крайней мере для случая большой памяти, можно считать решённой,

[1] В которой используется комплементарная структура: металл — окисел — полупроводник. Отсюда сокращение. — *Прим. изд. ред.*

поскольку при этой технологии время переключения („срабатывания вентиля") в 2 нс и время выборки из памяти („время цикла") в 40 нс уже не так непропорционально велики.

Уменьшение размеров сокращает и время прохождения сигналов — путь в 30 см соответствует времени прохождения в 1 нс = 10^{-6} с. Того же порядка (и даже несколько меньше) достигнутое на сегодня время срабатывания. Что касается прогнозов на будущее, то по опыту последних лет можно считать, что каждые 6 лет происходит уменьшение времени срабатывания в 10 раз. Чтобы это могло в полной мере отразиться на повышении скорости работы машины, необходимы дальнейшие успехи „большой интеграции".

При потреблении мощности в 10^{-7} Вт и времени срабатывания в 1 нс мы получаем на каждый элементарный процесс переключения работу в 10^{-16} Втс. Это пока ещё на 4 десятичных порядка выше того предела, который устанавливается энергией теплового кванта $kT\ln 2$ и составляет 10^{-20} Втс при температуре $T = 300°$ К = 27 °С. Энергия электромагнитного кванта $h\nu$ (при частоте 10^9 Гц она составляет всего лишь 10^{-24} Втс) ещё меньше, и на работе ЭВМ квантовые эффекты пока никак не сказываются.

Дальнейший прогресс обещает использование эффекта Джозефсона, которое позволяет достичь времени переключения порядка 1 пс = 10^{-12} с. Здесь требуется охлаждение до температуры, близкой к абсолютному нулю, — при таком охлаждении предел, устанавливаемый энергией теплового кванта, отодвигается далеко вниз. Очень перспективен также полевой транзистор, изготовленный из арсенида галлия, который обещает достичь (при комнатной температуре) времени переключения всего лишь в несколько пикосекунд.

Интересно провести сравнение с нервной клеткой человека. Она работает сравнительно медленно, со временем срабатывания в несколько мс. Однако число входов в неё велико: отдельные нейроны имеют до тысячи синапсов. Плотность упаковки также значительна: в коре головного мозга на 1 м^3 приходится до 10^{14} нейронов, что соответствует 10^{14} бит/м^3 — на десятки порядков больше, чем при современной технологии. Располагая общим количеством нейронов около 10^{10}, человек уже чисто количественно превосходит самые большие современные машины на несколько порядков, причём потребление энергии близко к тому пределу, который устанавливается квантовыми эффектами.

Ещё более поразительное явление представляет собой запоминание генетической информации в двойных спиралях генов. Здесь плотность информации составляет уже величину порядка 10^{27} бит/м^3; она реализуется кодированием с помощью групп молекул, содержащих по нескольку десятков атомов.

Содержание части 2

Вступление

Глава 5. Блочная структура и динамическое распределение памяти

5.1. Блоки и распределение памяти

5.1.1. Блочная структура. 5.1.2. Магазинное распределение памяти. 5.1.3. Память с делением на слова. 5.1.4. Относительная адресация. 5.1.5. Массивы с динамически устанавливаемыми границами индексов. 5.1.6. Заключительные замечания.

5.2. Процедуры и блочная структура

5.2.1. Включение процедур в блочную структуру. 5.2.2. Дерево блочной структуры, дополненное стрелками вызова. 5.2.3. Дерево динамической блочной структуры. 5.2.4. Статические и динамические цепочки ссылок. 5.2.5. Динамическое распределение памяти при наличии процедур.

Глава 6. Внешняя память и связь с внешним миром, структуры данных, организация памяти

6.1. Технические характеристики устройств внешней памяти и ввода/вывода

6.1.1. Память с прямым доступом. 6.1.2. Память с непрямым доступом. 6.1.3. Единицы передачи и обмена.

6.2. Функциональное описание внешней памяти и устройств ввода/вывода

6.2.1. Перфокарты и колоды перфокарт, перфоленты: носители однократного использования. 6.2.2. Память на магнитной ленте с зонами переменной длины: носитель многократного использования с последовательным доступом к зонам переменной длины. 6.2.3. Память на магнитной ленте и на магнитных дисках с фиксированным разбиением на блоки: носители многократного использования с последовательным доступом к фиксированным блокам, структурированная память. 6.2.4. Память на магнитных дисках: носитель многократного использования с вращательным доступом.

6.3. Введение новых вычислительных структур

6.3.1. Подструктуры. 6.3.2. Операционное обогащение. 6.3.3. Образование пар и произвольных наборов. 6.3.4. Вариантные образования. 6.3.5. Рекурсивное определение вычислительных структур: рекурсивные структуры данных. 6.3.6. Термы и диаграммы.

6.4. Организация данных: списки и указатели

6.4.1. Списки. 6.4.2. Структурированная память. 6.4.3. Указатели. 6.4.4. Элементы сцеплений.

6.5. Реализация различных типов структурированной памяти с помощью указателей

6.5.1. Реализация стеков. 6.5.2. Реализация последовательностей. 6.5.3. Реализация деревьев с размеченными листьями.
6.6. Реализация структурированной памяти с помощью линейной памяти
6.6.1. Разбросанное распределение памяти. 6.6.2. Последовательное распределение памяти.

Глава 7. Формальные языки
7.1. Отношения и формальные системы
7.1.1. Бинарные отношения и ориентированные графы. 7.1.2. Нётеровы и конфлюэнтные отношения. 7.1.3. Формальные языки: общие понятия.
7.2. Формальные языки над последовательностями символов
7.2.1. Согласованные системы редукций. 7.2.2. Полутуэвские системы. 7.2.3. Полутуэвские алгоритмы. 7.2.4. Языки Хомского и грамматики Хомского. 7.2.5. Бэкусова нотация и обобщённая бэкусова нотация. 7.2.6. Регулярные выражения. 7.2.7. Подстановка грамматик.
7.3. Графы редукции и деревья редукции
7.3.1. Двудольные графы. 7.3.2. Графы и деревья редукции. 7.3.3. Построение графов редукции. 7.3.4. Однозначность. 7.3.5. Критерий однозначности. 7.3.6. Структурные грамматики.
7.4. Проблема анализа
7.4.1. Тупики. 7.4.2. Методы последовательного анализа для регулярных грамматик. 7.4.3. Методы последовательного анализа для КС-грамматик. 7.4.4. Методы последовательного анализа сверху-вниз. 7.4.5. Метод рекурсивного спуска.
7.5. Вычислимость и разрешимость

Глава 8. Определение синтаксиса и семантики алгоритмических языков
8.1. Синтаксис алгоритмических языков
8.1.1. Синтаксическое описание составных объектов. 8.1.2. Синтаксическое описание деревьев Канторовича.
8.2. Операционная семантика
8.2.1. Строение формул и вычисление их значений. 8.2.2. Частичная определимость. 8.2.3. Нестрогие операции. 8.2.4. Недетерминизм. 8.2.5. Семантика передачи параметров в подпрограммах. 8.2.6. Операционная семантика рекурсии. 8.2.7. Редукционные машины.
8.3. Семантика состояний
8.3.1. Исчисление состояний по Маккарти. 8.3.2. Исчисление утверждений по Флойду, Хоару и Дейкстре.
8.4. Математическая семантика
8.4.1. Теория неподвижных точек. 8.4.2. Абстрактные типы (данных). 8.4.3. Абстрактные типы и характеризация базовых вычислительных структур. 8.4.4. Абстрактные типы и описания синтаксиса и семантики языков программирования.

Приложение А. Системы счисления
А.1. Позиционные системы счисления и перевод целых чисел из одной системы счисления в другую
А.2. Представление отрицательных чисел
А.3. Четыре основные арифметические операции
А.4. Числа с плавающей запятой

Приложение B. Теория информации Шеннона
B.1. Дискретизация
B.1.1. Развёртка. B.1.2. Квантование.
B.2. Вероятностная теория информации
B.2.1. Шенноновские сообщения. B.2.2. Количество информации.
B.2.3. Пропускная способность канала.
B.2.4. Надёжность кода.

Приложение C. Соответствия и функции
C.1. Некоторые специальные свойства соответствий
C.1.1. Функции. C.1.2. Отображения. C.1.3. Многозначные функции. C.1.4. Представление соответствий и функций.
C.2. Диаграммы для соответствий и функций
C.3. Возведение в степень для множеств

Приложение D. Устройства ввода/вывода данных
D.1. Требования и возможности
D.2. Вывод
D.2.1. Устройства посимвольной печати. D.2.2. Устройства построчной печати. D.2.3. Графические устройства.
D.2.4. Экранные устройства. D.2.5. Речевой вывод.
D.3. Ввод
D.3.1. Клавиатуры. D.3.2. Точечный ввод с дисплея.
D.3.3. Устройства считывания маркировок. D.3.4. Устройства считывания со стандартных формуляров.

Приложение E. К истории информатики
E.1. Введение
E.1.1. Лейбниц. E.1.2. Корни информатики.
E.2. История цифровых и символьных вычислений.
E.2.1. Цифровые вычисления. E.2.2. Символьные вычисления.
E.3. История связи
E.3.1. Передача сообщений. E.3.2. Принцип двоичного кодирования. E.3.3. Теория кодирования и теория информации, теория экстраполяции. E.3.4. Регулирование.
E.4. Автоматы и алгоритмы
E.4.1. Принцип автомата. E.4.2. Программное управление. E.4.3. Алгоритмы. E.4.4. Алгоритмические языки. E.4.5. Рекурсивность.

Литература. Синтаксические диаграммы для использованных в книге версий алгола-68 и паскаля. Предметно-именной указатель.

Оглавление

Предисловие к русскому изданию . . . 5
От переводчиков . . . 7
Из предисловия к первому изданию части 1 . . . 8
Из предисловия к первому изданию части 2 . . . 9
Предисловие ко второму изданию части 1 . . . 9
Предисловие ко второму изданию части 2 . . . 10
Предисловие к третьему изданию части 1 . . . 10
Предисловие к третьему изданию части 2 . . . 13
Вступление . . . 16

Глава 1. Информация и сообщение . . . 18

1.1. Сообщение и информация . . . 18
1.1.1. Языковые сообщения . . . 21
1.1.2. Письмо . . . 23
1.2. Органы чувств . . . 24
1.2.1. Работа органов чувств, проведение возбуждения . . . 24
1.2.2. Обработка раздражения в мозге . . . 28
1.2.3. Значение информационных представлений . . . 35
1.3. Устройства связи . . . 36
1.3.1. Типы устройств связи . . . 36
1.3.2. Сигналы и параметры сигналов . . . 39
1.4. Дискретные сообщения . . . 40
1.4.1. Знаки, наборы знаков, алфавиты . . . 40
1.4.2. Коды и кодирования . . . 51
1.4.3. Последовательная и параллельная передача . . . 60
1.4.4. Символы . . . 62
1.5. Обработка сообщений и обработка информации . . . 62
1.5.1. Обработка сообщений как кодирование . . . 62
1.5.2. Интерпретация обработки сообщений . . . 64
1.6. Алгоритмы . . . 66
1.6.1. Характеристические свойства алгоритмов . . . 67
1.6.2. Примеры алгоритмов . . . 68
1.6.3. Рекурсия и итерация . . . 71
1.6.4. Специальные формы описания алгоритмов . . . 72
1.6.4.1. Пример: алгоритмы Маркова . . . 73
1.6.4.2. Рекурсивные алгоритмы по Маккарти . . . 75

Глава 2. Основные понятия программирования . . . 77

2.1. Основные вычислительные структуры . . . 79
2.1.1. Объекты . . . 79

2111 Сорта объектов 79
2112 Стандартные обозначения объектов . . 82
212 Операции . 84
213 Вычислительные структуры 86
2131 Вычислительная структура Z целых чисел 87
2132 Вычислительная структура N натуральных чисел . 90
2133 Вычислительные структуры для вычислений с рациональными, вещественными и комплексными числами . . 91
2134 Вычислительные структуры для нечисловых вычислений последовательности знаков 93
2135 Вычислительные структуры для нечисловых вычислений· размеченные бинарные деревья 96
2136 Вычислительная структура $\mathbb{B}_2$ значений истинности . . . 99
2137 Переходы между сортами 101
2138 Составные объекты 103
22 Формулы 103
221 Обозначения параметров 103
222 Формулы и формуляры 105
2221 Построение формул 105
2222 Формуляры 109
2223 Строгость операций 114
2224 Преобразование формул . . 115
223 Условные формулы 117
2231 Альтернатива и последовательный разбор случаев 117
2232 Охраняемый разбор случаев 120
2233 Проведение вычислений на формулярах с разбором случаев 122
2234 Нестрогий характер разбора случаев . . 124
23 Подпрограммы 125
231 Описание подпрограмм 125
2311 Нотация 125
2312 Системы подпрограмм 127
2313 Предохранители 128
2314 Связанные обозначения 129
232 Рекурсия . . . 129
233 Рекурсивная машина обработки формуляров . 138
24 О технике рекурсивного программирования 144
241 Как приходят к рекурсивным подпрограммам? 144
2411 Органически рекурсивные определения 144
2412 Извлечение рекурсии из постановки задачи 144
2413 Вложение 145
2414 Метод родственных задач 148
2415 Использование характеристических свойств 149
2416 Обращение . 151
242 Как доказывать свойства алгоритмов? 153
243 Некоторые замечания об окончании и о роли предохранителей и стражей 155
2431 155
2432 155
2433 156
2434 156
2435 157

2.5. Подчинение подпрограмм 157
2.5.1. Подчинённые подпрограммы 157
2.5.2. Подавленные параметры 159
2.5.2.1. Глобальные и нелокальные параметры . . . 159
2.5.2.2. Экранирование 161
2.5.2.3. Неподвижные параметры 162
2.5.3. Подпрограммы, свободные от параметров 163
2.6. Подпрограммы в качестве параметров и в качестве результатов 164
2.6.1. Подпрограммы в качестве параметров 164
2.6.2. Задержанные вычисления, осуществляемые посредством использования подпрограмм, свободных от параметров, в качестве параметра 166
2.6.3. Подпрограммы в качестве результата 168

Глава 3. Машинно-ориентированные алгоритмические языки 170

3.1. Предложения общего вида 171
3.1.1. Описания промежуточных результатов 171
3.1.1.1. Упрощённая нотация 173
3.1.1.2. Обозначения промежуточных результатов в рекурсивных подпрограммах 176
3.1.1.3. Линеаризация хода вычислений 180
3.1.1.4. Замечание относительно паскаля . . . . .
3.1.2. Групповые описания промежуточных результатов . 180
3.2. Программирование с переменными 182
3.2.1. Повторно используемые обозначения промежуточных результатов 182
3.2.2. Описания и присваивания 183
3.2.3. Неизменные переменные 184
3.2.4. Операторы 185
3.2.5. Примеры 186
3.2.6. Групповые описания переменных и групповые присваивания 187
3.3. Итеративное программирование 188
3.3.1. Повторительные программы с точки зрения итерации 188
3.3.2. Повторение 192
3.3.2.1. Итеративные программы 194
3.3.2.2. Иерархические повторительные системы и вложенные циклы 197
3.3.2.3. Диаграммы Насси — Шнейдермана 199
3.3.3. Решение задач с помощью итеративных форм . . . 200
3.3.3.1. 201
3.3.3.2. 201
3.3.3.3. 202
3.3.3.4. 203
3.3.4. Линеаризация 204
3.3.5. Условные операторы 206
3.3.5.1. Операторы альтернативы 207
3.3.5.2. Последовательные условные операторы . . 209
3.3.5.3. Охраняемые операторы 211
3.3.6. Пустой оператор 212
3.4. Операторы перехода 214
3.4.1. Гладкие вызовы и операторы перехода 214
3.4.2. Реализация повторения при помощи переходов . . 218
3.4.2.1. 218
3.4.2.2. 218
3.4.2.3. 219

3.4.3. Блок-схемы программ 220
3.4.3.1. 220
3.4.3.2. 223
3.4.3.3. 227
3.5. Процедуры 227
3.5.1. Параметры-переменные 228
3.5.2. Описания процедур 229
3.5.3. Вызовы процедур 231
3.5.3.1. 232
3.5.3.2. 232
3.5.4. Транзитные параметры и параметры-результаты . . 233
3.5.5. Входные параметры 235
3.5.6. Подавляемые параметры 236
3.5.7. Процедуры как средство структурирования 237
3.5.8. Рекурсивное определение повторения 239
3.5.8.1. Переход как гладкий вызов процедуры без параметров 239
3.5.8.2. Рекурсивное определение условного повторения 239
3.5.8.3. Рекурсивное определение повторения с пересчётом 240
3.6. Массивы 242
3.6.1. Индексированные переменные 243
3.6.1.1. 244
3.6.1.2. 245
3.6.2. Многомерные массивы 249
3.6.2.1. 249
3.6.2.2. 250
3.6.2.3. 251
3.6.3. Сведение многомерных массивов к одномерным . . 251
3.6.4. Статическое распределение памяти 253
3.6.4.1. 253
3.6.4.2. 254
3.6.4.3. 255
3.7 Декомпозиция формул 255
3.7.1. Декомпозиция по принципу магазина 256
3.7.2. Использование магазина промежуточных результатов 259
3.7.3. Перевод в трёхадресную форму 260
3.7.4. Перевод в одноадресную форму 263
3.7.5. Границы применимости принципа магазина . . . 266
3.7.6. Обработка разбора случаев 266
3.7.7. Исключение логических операций 269

Глава 4. Двоичные комбинационные и переключательные схемы 271

4.1. Булева алгебра 271
4.1.1. Абстрактное определение булевой алгебры 271
4.1.1.1. 271
4.1.1.2. 272
4.1.1.3. 273
4.1.1.4. 274
4.1.1.5. 275
4.1.2. Теорема о булевой нормальной форме 276
4.1.3. Отношение порядка в булевой алгебре. Импликация 279
4.1.3.1. Отношение „сильнее“ 279

4.1.3.2. Отношение „сильнее" на булевых функциях 280
4.1.3.3. Отношение порядка между булевыми выражениями 281
4.1.3.4. Приложения к высказываниям и предикатам 281
4.1.4. Таблицы решений 284
4.1.4.1. Совместные таблицы решений 284
4.1.4.2. Последовательные таблицы решений 287
4.1.5. Переключательные функции 289
4.1.5.1. Символические изображения переключательных функций 290
4.1.5.2. Соединение символических изображений . . 291
4.1.5.3. Пример: полусумматор, полный сумматор . 292
4.1.5.4. Пример: перекодировщики 293
4.1.6. Техническая реализация комбинационных схем . . 294
4.2. Двоичное кодирование 295
4.2.1. Двоичное сравнение 296
4.2.2. Двоичная арифметика 297
4.2.2.1. Двоичный счётчик 297
4.2.2.2. Сложение и вычитание 299
4.2.2.3. Умножение 301
4.2.2.4. Операции с двоично представленными последовательностями знаков и размеченными деревьями 302
4.2.3. Арифметика с ограниченным числом разрядов . . 303
4.3. Переключательные схемы 306
4.3.1. Переменные памяти для двоичных слов 307
4.3.1.1. Пример: сложение 307
4.3.1.2. Двоичные регистры и элементы задержки . . 309
4.3.1.3. Присоединение элементов задержки к логическим элементам 310
4.3.2. Построение переключательных схем 311
4.3.2.1. Последовательный и параллельный сумматоры 311
4.3.2.2. Переключательная схема сдвига 314
4.3.3. Триггеры 315
4.3.4. Триггерные переключательные схемы 317
4.3.4.1. Триггерные переключательные схемы для арифметики 318
4.3.4.2. Триггерные переключательные схемы для неарифметических операций 320
4.3.5. Техническая реализация переключательных схем . 321
4.4. Успехи и предельные возможности технологии 322

Содержание части 2 I

Учебное пособие

Фридрих Л. Бауэр
Герхард Гооз

ИНФОРМАТИКА

Вводный курс

В двух частях

Часть 1

Заведующий редакцией чл.-корр. АН СССР В. И. Арнольд
Зам. зав. редакцией А. С. Попов
Ст. научн. редактор В. И. Авербух
Мл. научн. редактор Л. А. Королёва
Художник А. Я. Мусин
Художественный редактор В. И. Шаповалов
Технический редактор И. М. Кренделёва
Корректор В. И. Киселёва

ИБ № 6903

Сдано в набор 07.02.89. Подписано к печати 26.01.90. Формат 60×90¹/₁₆. Бумага тип. № 1. Печать высокая. Гарнитура литературная. Объем 10,50 бум. л. Усл. печ. л. 21. Усл. кр.-отт. 21. Уч.-изд. л. 18,33. Изд. № 1/6116. Тираж 20 000 экз. Зак. 132. Цена 1 р. 60 к.

Издательство «Мир» В/О «Совэкспорткнига» Госкомпечати СССР. 129820, ГСП, Москва, И-110, 1-й Рижский пер., 2.

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Государственного комитета СССР по печати. 198052, г. Ленинград, Л-52, Измайловский проспект, 29.