Н.Вирт

АЛГОРИТМЫ + СТРУКТУРЫ ДАННЫХ = ПРОГРАММЫ

Монография известного швейцарского специалиста по системному программированию, знакомого советским читателям по переводу его книги «Систематическое программирование. Введение.» (М.: Мир. 1977). Она содержит описание и анализ основных алгоритмов, методов построения программ. Книгу можно использовать и как руководство по применению языка Паскаль в задачах математического обеспечения ЭВМ.

Для научных работников, преподавателей, аспирантов и студентов, специализирующихся по математическому обеспечению ЭВМ.

Содержание

Указатель программ

Указатель

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Уже много лет говорят о кризисе в области программирования и создания программного обеспечения. Появилась масса теорий и направлений, авторы и апологеты которых обещают быстро доставить корабль программирования к земле обетованной. Уже были и языки высокого и очень высокого уровня, и структурное программирование, и доказательство правильности, и доказательное программирование, и масса всяческих технологий. Один из авторов языка Ада в своем интервью, опубликованном в старом и уважаемом программистами журнале Communication of the ACM (1984, № 4), договорился до того, что легкость написания программ, оказывается, и не была целью разработки этого нового языка программирования. В этой ситуации все труднее становится учить программистов хорошо программировать.

На словах все признают, что в основе программирования лежит творческий акт. У учеников нужно развивать способность творчески мыслить. И здесь огромна роль учителя, не столько рассказывающего о чем-то, сколько показывающего, как он делает то-то и то-то. Заметим, что в основе обучения и действий учителя лежит тот же творческий акт. Творчество не подвластно канонам, методикам и, тем более, технологиям. Представьте себе учебник по технологии физики или еще лучше по технологии математики, по технологии соответствующего мышления. Блестящие книги Пойи лишь подтверждают правило, что творчеству учат Учителя. При обучении «творческим специальностям» ученики не столько слушают, сколько смотрят, что и как делает учитель. Они наблюдают весь процесс его творчества.

Но что же делать, если нужно обучать не десятки или сотни людей, а многие тысячи? В этом случае надо полагаться на «школы», во главе которых стоят такие крупные Учителя. В школах процесс обучения и воспитания опять-таки основан на показе, но носит более сложный характер и захватывает значительно большее количество учеников. В программировании таких школ несколько. Одна из зарубежных школ находится в Цюрихе, во главе ее стоит Н. Вирт. Именно отсюда пришел элегантный Паскаль, завоевавший

почти весь мир, отсюда пришли Модула и Модула-2. Здесь появилась книга «Систематическое программирование. Введение» (Пер. с англ. — М.: Мир, 1977), и отсюда же появляется книга, перевод которой читатель держит в руках.

В ней обобщен авторский опыт многолетнего обучения программированию. Искусство автора проявилось в том, что в своей книге он подчеркивает основополагающие принципы программирования, а не конкретные особенности того или иного модного языка программирования. Благодаря этому его книге суждена долгая жизнь. Здесь почти нет никаких рецептов, рекомендаций, методик. Это набор примеров программ, про которые автор говорит: «Смотрите, как и почему я это делаю».

Действительно, читатель, посмотрите. Если Вы только начинающий программист, то для Вас книга будет очень хорошим самоучителем. Если Вы достаточно опытны, то в ней Вы обнаружите многие тонкости, которые позволят Вам усовершенствовать Ваш стиль программирования. И наконец, если Вы первый раз окунаетесь в «море программирования» и не знаете, что такое ЭВМ, то лучше оставьте эту книгу на прилавке; пусть ее купят другие: ведь программистов так много, а хороших книг для них, к сожалению, так мало.

Д. Б. Подшивалов

Посвящается Нани

ПРЕДИСЛОВИЕ

В последние годы *программирование для вычислительных машин* стало не только средством, владение которым оказывается решающим для успешной работы во многих прикладных областях, а также и предметом научного изучения. Из ремесла программирование превратилось в академическую дисциплину. Первые крупные шаги в этом направлении были сделаны в работах Э. Дейкстры и К. Хоора. «Заметки по структурному программированию» Дейкстры *) определили новый взгляд на программирование как на предмет научного изучения и поле для интеллектуальной деятельности; этот подход получил название «революции» в программировании. В статье «Аксиоматическая основа программирования для вычислительных машин» **) Хоор продемонстрировал, что программы поддаются точному анализу, основанному на математических рассуждениях. В этих работах убедительно показано, что можно избежать многих ошибок программирования, если программисты со знанием дела будут применять те методы и приемы, которые они ранее использовали интуитивно и часто неосознанно. Основное внимание в них уделено построению и анализу программ, или, более конкретно, структуре алгоритмов, представленных текстами программ. Причем совершенно ясно, что систематический и научный подход к построению программ важен в первую очередь в случае больших программ со сложными данными. Таким образом, методы программирования включают также и все варианты структурирования данных. *Программы* представляют собой в конечном счете конкретные формулировки абстрактных *алгоритмов*, основанные на конкретных представлениях и структурах *данных*. Важный вклад в упорядочение широкого разнообразия терминов и концепций, относящихся к структурам данных, был сделан Хоором в статье «О структурной организации данных» ***). Стало ясно, что решения о структурировании данных нельзя принимать без

*) В книге: О. Дал, Э. Дейкстра, К. Хоор, Структурное программирование: Пер. с англ. — М.: Мир, 1975.

**) В Comm, ACM, 12, № 10 (1969), 576—583.

***) В книге «Структурное программирование» см. сноску выше.

знания алгоритмов, применяемых к этим данным, и наоборот, структура и выбор алгоритмов существенным образом зависят от структуры данных. Говоря короче, строение программ и структуры данных неразрывно связаны.

Предлагаемая книга начинается главой о структуре данных по двум причинам. Во-первых, мы интуитивно чувствуем, что данные предшествуют алгоритмам: нужно иметь некоторые объекты, прежде чем выполнять действия с ними. Во-вторых (и это более непосредственная причина), хотя здесь и предполагается, что читатель знаком с основными понятиями программирования, но по традиции курсы введения в программирование явно уделяют больше внимания алгоритмам, оперирующим данными со сравнительно простой структурой. В связи с этим возникла необходимость в вводной главе о структуре данных.

На протяжении всей книги, и в частности в гл. 1, мы следуем теории и терминологии, которые были предложены Хоором *) и реализованы в языке программирования Паскаль **). Суть этой теории состоит в том, что данные представляют собой прежде всего абстракции реальных объектов и формулируются предпочтительно как абстрактные структуры, не обязательно реализованные в распространенных языках программирования. В процессе конструирования программы представление данных постепенно уточняется вслед за уточнением алгоритма, все более подчиняясь ограничениям, накладываемым конкретной системой программирования. Поэтому мы определим несколько основных строительных конструкций — структур для данных, называемых *фундаментальными структурами*. Особенно важно, что эти конструкции довольно легко реализуются на современных вычислительных машинах, поскольку только в этом случае их можно действительно рассматривать как элементы реального представления данных, т. е. как «молекулы», возникающие на окончательном этапе уточнения описаний данных. Это следующие структуры: *запись*, *массив* (фиксированного размера) и *множество*. Неудивительно, что эти основные строительные блоки соответствуют математическим обозначениям, которые также являются фундаментальными.

Краеугольным камнем этой теории структур данных служит различие между фундаментальными и усложненными структурами. Фундаментальные структуры — это как бы *молекулы* (в свою очередь состоящие из атомов); они являются компонентами, из которых состоят усложненные структуры.

---

*) См. первую сноску на предыдущей странице.

**) N. Wirth, The Programming Language Pascal, Acta Informatica, 1, No. 1 (1971), 35—63.

Переменные фундаментальной структуры могут менять только свое значение, сохраняя форму или множество значений, которые они могут принимать. Таким образом, размер занимаемой ими памяти остается постоянным. Напротив, усложненные структуры характеризуются изменением не только значения, но *и* формы во время выполнения программы. Поэтому для их реализации нужно применять более сложные приемы.

Последовательный файл, или просто последовательность, в этой классификации является промежуточным. Его длина, естественно, изменяется, но это изменение формы тривиально. Поскольку последовательный файл играет важную роль практически во всех вычислительных системах, мы рассмотрим его среди фундаментальных структур в гл. 1.

Во второй главе описываются различные *алгоритмы сортировки*. Математический анализ некоторых из них раскрывает преимущества и недостатки разных методов и помогает программисту понять важность анализа при выборе подходящего способа решений стоящей перед ним задачи. Разграничение методов сортировки массивов и методов сортировки файлов (часто называемых внутренней и внешней сортировкой) демонстрирует решающее влияние представления данных на выбор алгоритмов и их сложность. Сортировке уделяется столько внимания, так как с ее помощью можно прекрасно иллюстрировать многие принципы программирования и ситуации, возникающие и в других задачах. Создается впечатление, что можно построить целый курс программирования, выбирая примеры только из задач сортировки.

Другая тема, которой часто пренебрегают в курсах введения в программирование, но которая важна для понимания большого числа алгоритмических решений, — это рекурсия. Поэтому третья глава посвящена *рекурсивным алгоритмам*. В ней показано, что рекурсия — это обобщение повторения (итерации) и поэтому является важным и мощным средством программирования. К сожалению, при обучении программированию рекурсивные методы нередко демонстрируют на примерах, в которых достаточно простой итерации. В гл. 3, напротив, приводятся несколько примеров задач, где рекурсия позволяет получить решение наиболее естественным образом, тогда как использование итерации сделало бы программы громоздкими и трудными для понимания. Идеальным приложением рекурсии служит класс алгоритмов с *возвратом*, но наиболее очевидно ее использование в алгоритмах, работающих с данными, структура которых определена рекурсивно. Подобные случаи рассматриваются в двух последних главах; третья глава дает для них соответствующую подготовку.

В гл. 4 рассматриваются *динамические структуры данных*, т. е. такие структуры, которые изменяются во время выполнения программы. Показано, что рекурсивные структуры данных являются важным подклассом обычно используемых динамических структур. Хотя в таких случаях возможно и естественно рекурсивное определение, его обычно на практике не применяют. Вместо этого программист получает доступ к механизму реализации путем использования явных *ссылок*, или *указателей*. Данная книга следует этому принципу, отражающему современное положение вещей: гл. 4 посвящена программированию со ссылками, а также спискам, деревьям и примерам, требующим еще более сложных совокупностей данных. В ней рассматривается процесс, который часто (и не совсем верно) называют «обработкой списков». Довольно много места отведено организации деревьев, и в частности деревьев поиска. Глава заканчивается рассмотрением метода рассеянных таблиц, или «хеширования», который часто предпочитают деревьям поиска. Это дает возможность сравнить два принципиально различных метода решения часто встречающейся задачи.

Последняя глава состоит из краткого введения в теорию *формальных языков* и *грамматического разбора* и из описания *транслятора* для небольшого и простого языка программирования для простой вычислительной машины. Мы включили эту главу по следующим причинам. Во-первых, квалифицированный программист должен иметь некоторое представление о методах трансляции языков программирования. Во-вторых, постоянно растет число задач, в которых для удобства работы нужно определить некоторый простой язык ввода или управления. В-третьих, поскольку формальные языки определяют рекурсивную структуру на последовательности символов, то процессоры для них служат хорошими примерами успешного применения рекурсии, которая позволяет добиться ясной структуры там, где программы оказываются большими и даже огромными. Для наших примеров мы использовали язык, называемый ПЛ/0, так как он является компромиссом между языком слишком простым, чтобы служить хорошим примером, и языком, транслятор для которого оказался бы столь большим, что его не имело бы смысла включать в книгу, предназначенную не только для разработчиков трансляторов.

Программирование — это искусство конструирования. Как можно научить конструкторской, изобретательской деятельности? Есть такой метод: выделить простейшие строительные блоки из многих уже существующих программ и дать их систематическое описание. Но программирование представляет собой обширную и разнообразную деятельность, часто

требующую сложной умственной работы. Ошибочно считать, что ее можно свести к использованию готовых рецептов. В качестве метода обучения нам остается тщательный выбор и рассмотрение характерных примеров. Конечно, не следует считать, что изучение примеров всем одинаково полезно. При этом подходе многое зависит от сообразительности и интуиции обучающегося. Это особенно верно для относительно сложных и длинных примеров программ. Они не случайно включены в эту книгу. Длинные программы обычно часто встречаются на практике, и они лучше всего подходят для выявления того неуловимого, но важного свойства, которое называют стилем или дисциплиной. Кроме того, они служат упражнением в искусстве читать программы, которым часто пренебрегают по сравнению с искусством писать программы. Главным образом по этой причине в качестве примеров берутся целиком большие программы. Читателю показывается, как постепенно создается программа, ему даются различные «моментальные снимки» ее развития, причем эти разработки демонстрируют метод *поэтапного уточнения* деталей. Я считаю важным, рассматривая программы в их окончательном виде, уделять достаточно внимания деталям, поскольку именно в них кроются основные трудности в программировании. Представить алгоритмы в чистом виде и дать их математический анализ было бы интересно с чисто академической точки зрения, но было бы нечестно по отношению к программисту-практику. Поэтому я строго придерживался принципа представлять программы в их окончательном виде на том языке, на котором они могут реально выполняться в вычислительной машине.

Разумеется, здесь возникает задача найти такую форму представления, которая может быть реализована на ЭВМ и одновременно является достаточно машинно-независимой, чтобы здесь использоваться. Для этого не подходят ни широко употребительные языки, ни абстрактная нотация. Нужный компромисс обеспечивает язык Паскаль, который разработан специально для этой цели, поэтому и используется в данной книге. Программисты, знакомые с другими языками высокого уровня, смогут легко разобраться в программах на Паскале, так как его выражения поясняются в тексте. Но это не значит, что некоторая подготовка была бы излишней. Идеальную подготовку дает книга «Систематическое программирование» *), так как она тоже основана на Паскале.

---

*) N. Wirth (Englewood Cliffs, N. J.: Prentice-Hall, INC., 1973). [Имеется перевод: Вирт Н. Систематическое программирование. Введение. — М.: Мир, 1977.]

Но она не может служить учебником языка Паскаль — для этого существуют более подходящие книги *).

Настоящая книга представляет собой сжатое и переработанное изложение нескольких курсов программирования, прочитанных в Федеральном технологическом институте (ETH) в Цюрихе. Многими идеями и взглядами, изложенными в этой книге, я обязан беседам со своими сотрудниками в ETH. В частности, мне хотелось бы выразить благодарность м-ру Г. Сандмейеру за внимательное прочтение рукописи и мисс Хейди Тейлер за внимание и терпение при перепечатке текста. Я хотел бы также отметить большое влияние, оказанное встречами рабочих групп 2.1 и 2.3 IFIP и особенно многочисленными беседами, которые я вел при этом с Э. Дэйкстрой и К. Хоором. Наконец, что не менее важно, ETH щедро предоставлял вычислительные машины, без которых была бы невозможна подготовка этой книги.

Н. Вирт

*) K. Jensen and N. Wirth, PASCAL — User Manual and Report Lecture Notes in Computer Science, Vol. 18 (Berlin, New York; Springer-Verlag, 1974). [Имеется перевод: Йенсен К., Вирт Н., Паскаль. Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982.]

Наш высокочтимый г. Л. Эйлер делает нам в назидание следующее заявление. Он откровенно признает:

...

III. что, являясь королем математиков, он все же вечно будет краснеть за вызов здравому смыслу и повседневному опыту, брошенный выводом из его формулы, согласно которой тело под действием силы притяжения к центру сферы внезапно изменит направление движения к центру;

IV. что он сделает все возможное, чтобы больше не изменять разуму, доверяясь ошибочной формуле. Он на коленях молит прощения за то, что как-то, имея в виду парадоксальный результат, он заявил: «Вычислениям следует доверять больше, чем чувствам, даже если кажется, что это противоречит действительности»;

V. что впредь он никогда больше не станет делать вычисления на шестидесяти страницах для получения результата, который по здравом размышлении можно вывести в десяти строках; и если он когда-нибудь вновь соберется, засучив рукава, считать три дня и три ночи подряд, то он прежде потратит четверть часа на раздумья о том, какие методы вычисления для этого наиболее подходящи

Вольтер, Памфлет доктора Акакия, ноябрь 1752 г.

1

ФУНДАМЕНТАЛЬНЫЕ СТРУКТУРЫ ДАННЫХ

1.1. ВВЕДЕНИЕ

Современная цифровая вычислительная машина первоначально предназначалась для облегчения и ускорения сложных и длительных вычислений. Однако при ее использовании обычно более важной оказывается способность хранить большой объем информации и обеспечивать доступ к нему, а способность вычислять, т. е. производить арифметические действия, во многих случаях отходит на второй план.

При этом обрабатываемая информация представляет собой в некотором смысле *абстракцию* какой-то части реального мира. Информация, доступная вычислительной машине, состоит из некоторых *данных* о действительности — таких данных, которые считаются относящимися к решаемой задаче и из которых, как предполагается, можно получить нужный результат. Данные являются абстракцией действительности, поскольку в них игнорируются некоторые свойства и характеристики реальных объектов, не существенные для решаемой задачи. Поэтому абстракция — это одновременно упрощение.

В качестве примера можно рассмотреть персональный файл служащего. Каждый служащий представлен (абстрагирован) в этом файле множеством данных, существенных либо для его характеристики, либо для процедур расчета. Это множество может включать некоторую идентификацию служащего, например его имя и заработную плату. Но вряд ли оно будет содержать такие несущественные данные, как цвет волос, вес и рост.

При решении какой-либо задачи как с помощью ЭВМ, так и без нее нужно выбрать некоторую абстракцию действительности, т. е. определить множество данных, описывающих реальную ситуацию. Этот выбор зависит от задачи, которую нужно решить. Затем следует выбрать способ представления этой информации. Здесь выбор определяется инструментами, применяемыми для решения задачи, т. е. средствами, которые предоставляет вычислительная машина. В большинстве случаев эти два этапа взаимозависимы.

Выбор представления данных часто бывает затруднителен, он не определяется однозначно имеющимися средствами. Его

следует осуществлять с учетом действий, производимых с данными. Хороший пример здесь — представление чисел, которые уже сами являются абстракциями свойств объектов. Если единственная (или по крайней мере основная) выполняемая операция — сложение, то лучший способ представить число n — это написать n черточек. При этом правило сложения окажется абсолютно простым и очевидным. Подобный принцип используется в римских цифрах, где правила сложения для небольших чисел также достаточно просты. С другой стороны, правила сложения небольших чисел, представленных арабскими цифрами, далеко не очевидны и требуют запоминания. Но при сложении больших чисел, а также при умножении и делении положение меняется. Представление чисел с помощью арабских цифр позволяет намного легче разложить эти операции на более простые благодаря системе записи, основанной на позиционном весе цифр.

Известно, что вычислительные машины используют внутреннее представление данных, основанное на двоичных цифрах (разрядах). Для человека такое представление неудобно из-за большого количества цифр в числе, но оно является наиболее подходящим для электронных схем, поскольку два значения (0 и 1) можно удобно и надежно кодировать наличием или отсутствием электрического тока, электрического заряда или магнитного поля.

Из приведенного примера видно, что при решении вопроса о представлении данных обычно имеется несколько уровней детализации. Пусть, например, нужно изобразить положение объекта в пространстве. На первом этапе берется пара вещественных чисел, например декартовы или полярные координаты. На втором этапе они представляются как числа с плавающей запятой: каждому вещественному числу x ставится в соответствие пара целых чисел, обозначающих мантиссу f и порядок e (например, $x = f \cdot 2^e$). На третьем этапе, когда учитывается, что данные должны располагаться в памяти ЭВМ, мы получаем двоичное позиционное представление целых чисел, и на последнем этапе двоичные числа могут представляться направлением магнитного поля в магнитном запоминающем устройстве. Ясно, что первый этап определяется в основном самой задачей, а последний тесно связан с используемым вычислительным устройством. Поэтому вряд ли следует требовать, чтобы программист сам определял способы представления чисел или даже характеристики запоминающего устройства. Эти «решения низшего уровня» можно предоставить разработчикам ЭВМ, так как они располагают наибольшей информацией о ее технологии, что позволяет им выбрать способ представления чисел, пригодный для всех (или почти всех) случаев.

С этой точки зрения очевидно значение *языков программирования*. Язык программирования описывает некоторую абстрактную вычислительную машину, понимающую термины этого языка, что соответствует какому-то уровню абстракции от объектов, используемых реальной ЭВМ. Следовательно, работая с таким «языком высокого уровня», программист освобождается (и отстраняется) от вопросов представления чисел, если число — элементарный объект этого языка.

Применение языка, который предоставляет подходящее множество основных абстракций, общих для большинства задач обработки данных, увеличивает надежность программ. Легче написать программу, основанную на знакомых нотациях для чисел, множеств, последовательностей и циклов, чем на разрядах, «словах» и переходах. Разумеется, в самой ЭВМ все данные: и числа, и множества, и последовательности — будут представлены в виде большой совокупности разрядов. Но для программиста это несущественно, если он не заботится о подробностях представления выбранных им абстракций и если он уверен, что представление, которое выберет машина (или транслятор), подходит для его целей.

Чем ближе абстракции к конкретной ЭВМ, тем легче разработчику языка выбрать представление данных и тем больше вероятность, что оно будет пригодно для всех (или почти всех) возможных задач. Это накладывает определенные ограничения на уровень абстракции от реальной вычислительной машины. Например, не имеет смысла включать в язык общего назначения в качестве основных элементов данных геометрические объекты, так как из-за сложности их представления оно будет сильно зависеть от выполняемых с ними действий. Но природа этих действий и их частота будут неизвестны разработчику универсального языка и его транслятора, поэтому любое его решение в некоторых случаях будет непригодным.

В настоящей книге эти соображения определяют выбор нотаций для описания алгоритмов и представления данных. Понятно, что мы хотим использовать привычную математическую нотацию, т. е. числа, множества, последовательности и т. д., а не такие машинно-зависимые понятия, как последовательности разрядов. Но ясно также, что желательно использовать язык, для которого *существует* эффективный транслятор. В равной мере неразумно как использовать машинно-ориентированный или машинно-зависимый язык, так и составлять программу для вычислительной машины с помощью абстрактных нотаций, не затрагивая проблему представления.

В качестве компромисса между этими крайностями был разработан язык программирования Паскаль, который и ис-

пользуется в данной книге [1.3, 1.5]. Этот язык был успешно реализован на нескольких ЭВМ, и было показано, что он достаточно близок к реальным машинам, а его свойства и их реализация довольно понятны. Кроме того, этот язык близок к другим языкам, особенно к Алголу-60, поэтому наши выводы можно использовать также применительно и к этим языкам.

1.2. КОНЦЕПЦИЯ ТИПА ДЛЯ ДАННЫХ

В математике принято классифицировать переменные в соответствии с некоторыми важными характеристиками. Проводится строгое разграничение, во-первых, между вещественными, комплексными и логическими переменными, во-вторых, между переменными, представляющими отдельные значения, множества значений или множества множеств, в-третьих, между функциями, функционалами, множествами функций и т. д. При обработке данных такая классификация не менее (если не более) важна. Мы будем придерживаться того принципа, что *каждая константа, переменная, выражение или функция бывают определенного типа.* Этот тип существенным образом характеризует множество значений, к которому принадлежит константа, которые может принимать переменная или выражение или которые может вырабатывать функция.

В математических текстах тип переменной обычно определяется по ее виду без обращения к контексту, но в программах для вычислительных машин это неприменимо, поскольку в них обычно используются буквы только одного вида, который допускает оборудование ЭВМ (латинские буквы). Поэтому широко используется правило, что тип явно задается в *описании* константы, переменной или функции, которое предшествует в тексте их использованию. Это правило особенно важно потому, что транслятор должен выбрать представление объекта в памяти ЭВМ. Очевидно, что объем памяти, выделяемой для переменной, должен устанавливаться в зависимости от того, какие значения она может принимать. Если эта информация известна транслятору, то можно избежать так называемого динамического распределения памяти. Это часто позволяет реализовать алгоритм более эффективно.

Таким образом, рассматриваемая здесь концепция типа, которая включена в язык программирования Паскаль, имеет следующие основные свойства [1.2]:

1. Любой тип данных определяет множество значений, к которому принадлежит константа, которые может принимать переменная (или выражение), или вырабатывать операция (или функция).

2. Тип значения, задаваемого константой, переменной или выражением, можно определить по их виду или описанию без необходимости выполнять какие-либо вычисления.
3. Каждая операция или функция требует аргументов фиксированного типа и выдает результат фиксированного типа. Если операция допускает аргументы нескольких типов (например, «+» используется для сложения как целых, так и вещественных чисел), то тип результата можно определить по специальным правилам языка.

Следовательно, транслятор может использовать информацию о типах для проверки вычислимости и правильности различных конструкций. Например, присваивание арифметической (вещественной) переменной булевского (логического) значения можно выявить без выполнения программы. Такая избыточность в тексте программы является важным вспомогательным средством разработки программ и рассматривается как существенное преимущество хороших языков высокого уровня перед машинным кодом или символическим языком ассемблера. Конечно, в конце концов данные будут представлены в виде большого количества двоичных цифр независимо от того, была ли программа написана на языке высокого уровня с использованием концепции типа или на языке ассемблера, где типы отсутствуют. Для вычислительной машины память — это однородная совокупность разрядов без какой-либо структуры. Но именно абстрактная структура позволяет программисту определять типы данных на фоне однообразных записей в памяти ЭВМ.

Теория, которая используется в этой книге, и язык программирования Паскаль предполагают некоторые методы определения типов данных. В большинстве случаев новые типы данных определяются с помощью ранее определенных типов данных. Значения, принадлежащие к такому типу, обычно представляют собой совокупности *значений компонент*, принадлежащих к определенным ранее *типам компонент*, такие составные значения называются *структурированными*. Если имеется только один тип компонент, т. е. все компоненты принадлежат одному типу, то он называется *базовым*.

Число различных значений, принадлежащих типу T, называется *кардинальным числом* T. Кардинальное число определяет размер памяти, нужной для размещения переменной x типа T. Этот факт обозначается так — x: T.

Поскольку типы компонент могут также быть составными, можно построить целую иерархию структур, но конечные компоненты структуры, разумеется, должны быть атомарными. Следовательно, система нотаций должна допускать описание

и простых, неструктурированных типов. Самый простой метод описания простого типа — это *перечисление* значений этого типа. Например, в программе, связанной с плоскими геометрическими фигурами, может описываться простой тип, называемый *фигурой*, значения которого задаются идентификаторами *прямоугольник*, *квадрат*, *эллипс*, *круг*. Но кроме типов, задаваемых программистом, нужно иметь некоторые *стандартные типы*, которые называются предопределенными. Они обычно включают *числа* и *логические переменные*. Если значения некоторого типа упорядочены, то такой тип называется упорядоченным или *скалярным*. В Паскале предполагается, что все неструктурированные типы упорядочены, в случае когда значения явно перечисляются, считается, что они упорядочены в порядке перечисления.

Применяя эти правила, можно описывать простые типы и строить из них структурированные типы любой степени сложности. Однако на практике недостаточно иметь только один общий метод объединения типов компонент в структуру. С учетом практических задач представления и использования данных универсальный язык должен располагать несколькими *методами структурирования*. Они могут быть эквивалентны в математическом смысле и различаться только операциями построения их значений и выбора компонент этих значений. Основные рассматриваемые здесь методы позволяют строить следующие структуры: *массив*, *запись*, *множество* и *последовательность* (*файл*). Более сложные структуры обычно не описываются как «статические» типы, а «динамически» создаются во время выполнения программы, причем их размер и вид могут изменяться. Такие структуры рассматриваются в гл. 4 — это списки, кольца, деревья и общие конечные графы.

Переменные и типы данных вводят в программу для того, чтобы их использовать в каких-либо вычислениях. Следовательно, нужно еще иметь и некоторое множество операций. Вместе с типами данных язык программирования задает некоторые простые, стандартные (атомарные) операции и методы структурирования, которые позволяют описывать сложные действия в терминах простых операций. Сутью искусства программирования обычно считается умение составлять операции. Однако мы увидим, что не менее важно умение составлять данные.

Важнейшие основные операции — это *сравнение* и *присваивание*, т. е. проверка равенства (и порядка в случае упорядоченных типов) и команда «установки равенства». Принципиальное различие этих двух операций выражается четким различием их обозначений в тексте (хотя оно, к сожалению, скрыто в таких широко распространенных языках, как

Фортран и ПЛ/1, которые используют знак равенства в качестве оператора присваивания):

Проверка равенства: $x = y$

Присваивание: $x := y$

Эти основные операции определены для большинства типов данных, но следует заметить, что для данных, имеющих большой объем и сложную структуру, выполнение этих операций может сопровождаться довольно сложными вычислениями.

Кроме проверки равенства и присваивания имеется еще один класс основных, неявно определенных операций — так называемых *операций преобразования*. Эти операции отображают одни типы данных в другие. Особенно они важны для составных типов. Составные значения строятся из значений компонент с помощью так называемых *конструкторов*, а значения компонент извлекаются с помощью так называемых *селекторов*. Таким образом, конструкторы и селекторы — это операции преобразования, отображающие типы компонент в составные типы и наоборот. Каждому методу структурирования соответствует своя пара конструкторов и селекторов, обозначения которых четко различаются.

Стандартным простым типам данных соответствует также некоторое множество стандартных простых операций. Следовательно, вместе со стандартными типами данных: числами и логическими значениями — вводятся также соответствующие арифметические и логические операции.

1.3. ПРОСТЫЕ ТИПЫ ДАННЫХ

Во многих программах целые числа используются в том случае, когда их собственно числовое значение несущественно и когда целое число указывает на выбор значения из небольшого множества возможных вариантов. В подобных случаях мы вводим новый, простой, неструктурированный тип T, перечисляя множества всех его возможных значений $c_1, c_2, \ldots, c_n$:

$$\textbf{type}\ T = (c_1, c_2, \ldots, c_n) \qquad (1.1)$$

Кардинальное число T есть $card(T) = n$.

Примеры:

type *фигура*=(*прямоугольник, квадрат, эллипс, круг*)
type *цвет*=(*красный, желтый, зеленый*)
type *пол* = (*мужской, женский*)
type *Boolean* = (*false, true*)

type *день* = (*понедельник*, *вторник*, *среда*, *четверг*, *пятница*, *суббота*, *воскресенье*)
type *валюта* = (*франк*, *марка*, *фунт*, *доллар*, *шиллинг*, *лира*, *гульден*, *крона*, *рубль*, *крузейро*, *иена*)
type *обитель* = (*ад*, *чистилище*, *рай*)
type *транспорт* = (*поезд*, *автобус*, *автомобиль*, *пароход*, *самолет*)
type *звание* = (*рядовой*, *капрал*, *сержант*, *лейтенант*, *капитан*, *майор*, *полковник*, *генерал*)
type *объект* = (*константа*, *тип*, *переменная*, *процедура*, *функция*)
type *структура* = (*файл*, *массив*, *запись*, *множество*)
type *состояние* = (*выключено*, *пустое*, *ошибка*, *перекос*)

При определении таких типов вводится не только новый идентификатор типа, но одновременно — множество идентификаторов, соответствующих значениям этого типа. Эти идентификаторы могут затем использоваться в программе как константы, при этом программа становится намного понятней. Если, например, мы определим переменные *s*, *d*, *r* и *b*

var *s*: *пол*
var *d*: *день*
var *r*: *звание*
var *b*: *Boolean*

то возможны следующие операторы присваивания:

s := *мужской*
d := *воскресенье*
r := *майор*
b := *true*

Очевидно, что они намного более информативны, чем операторы

$$s := 1 \qquad d := 7 \qquad r := 6 \qquad b := 2$$

которые будут им соответствовать, если *s*, *d*, *r* и *b* определить как переменные целого типа, а соответствующие константы изображать натуральными числами, определенными порядком перечисления. В дальнейшем транслятор может выявлять неуместное использование арифметических операций на таких нечисловых типах, как, например:

$$s := s + 1$$

Однако если тип считается упорядоченным, то полезно определить функции, которые выдают предшествующее и последующее значения для своего аргумента. Эти функции

обозначаются как $succ(x)$ (последующее значение) и $pred(x)$ (предшествующее значение). Упорядоченность значений типа T определяется правилом

$$(c_i < c_j) \equiv (i < j) \tag{1.2}$$

1.4. СТАНДАРТНЫЕ ПРОСТЫЕ ТИПЫ

Стандартные простые типы — это типы, которые являются встроенными для большинства ЭВМ. Они включают целые числа, логические значения и множество символов печати. В крупных вычислительных машинах имеются также вещественные числа и соответствующее множество простых операций. Мы обозначаем эти типы идентификаторами:

integer, Boolean, real, char.

Тип *integer* содержит подмножество целых чисел, размер которого может быть различным в разных вычислительных системах. Но предполагается, что все действия с данными этого типа являются точными и выполняются по обычным правилам арифметики и что вычисление прерывается, если результат оказывается за границами допустимого подмножества. Стандартные операции — это четыре действия арифметики: сложение (+), вычитание (—), умножение (*) и деление (**div**). Последнее должно давать целый результат, опуская возможный остаток, так, что для любых m и n

$$m - n < (m \,\mathbf{div}\, n)^* n \leqslant m. \tag{1.3}$$

Операция взятия остатка определяется с помощью деления уравнением

$$(m \,\mathbf{div}\, n)^* n + (m \,\mathbf{mod}\, n) = m. \tag{1.4}$$

Таким образом, m **div** n — это целое от деления m на n, а m **mod** n — остаток от деления.

Тип *real* обозначает подмножество вещественных чисел. В то время как арифметические действия с целыми числами дают точные результаты, для арифметических действий со значениями типа *real* допускается неточность в пределах ошибок округления, так как в вычислениях участвует конечное число цифр. В этом состоит явное различие между типами *integer* и *real*, существующее в большинстве языков программирования.

Деление вещественных чисел, дающее вещественный результат, мы обозначаем косой чертой (/), а деление целых чисел — **div**.

Два значения стандартного типа *Boolean* (булевское) обозначаются идентификаторами *true* (истина) и *false* (ложь),

Таблица 1.1. Булевские операции

p	q	$p \vee q$	$p \wedge q$	$\neg p$
true	*true*	*true*	*true*	*false*
true	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*
false	*false*	*false*	*false*	*true*

Операции над булевскими значениями — это логические конъюнкция, дизъюнкция и отрицание, значения которых приведены в табл. 1.1. Логическая конъюнкция обозначается символом $\wedge$ (или **and**), логическая дизъюнкция — символом $\vee$ (или **or**), а отрицание — символом $\neg$ (или **not**). Заметим, что операции сравнения дают результат типа *Boolean*. Следовательно, результат сравнения можно присваивать какой-либо булевской переменной или использовать в качестве операнда в булевских выражениях. Например, пусть даны булевские переменные p и q и целые переменные $x = 5$, $y = 8$, $z = 10$, тогда присваивания

$$p := x = y$$
$$q := (x < y) \wedge (y \leqslant z)$$

дают $p = false$ и $q = true$.

Стандартный тип *char* включает множество печатаемых символов. К сожалению, не существует общего стандартного множества символов, принятого во всех вычислительных системах. Поэтому использование слова «стандартный» может здесь ввести в заблуждение; его следует понимать в смысле «стандартный для вычислительной системы, на которой должна выполняться данная программа».

По-видимому, наиболее широко используется множество символов, определенное Международной организацией по стандартизации (ISO), и особенно его американская версия ASCII (американский стандартный код для обмена информацией). Поэтому в приложении А дана таблица символов ASCII. Она включает 95 печатаемых (*графических*) символов и 33 управляющих символа; последние используются в основном для передачи данных и для управления печатающим устройством. Широко распространено подмножество из 64 печатаемых символов (только прописные буквы), которое называется *ограниченным* множеством ASCII.

Для того чтобы описывать алгоритмы, работающие с символами, т. е. значениями типа *char*, независимо от вычислительной системы мы определим некоторые свойства

множества символов, делающие его связным. Это следующие свойства:

1. Тип *char* содержит 26 латинских *букв*, 10 арабских *цифр* и некоторое количество других графических символов таких, как знаки препинания.
2. Подмножества букв и цифр *упорядочены* и *связны*, т. е.

$$\begin{aligned}('A' \leqslant x) \wedge (x \leqslant 'Z') &\equiv x \text{ — буква,}\\ ('0' \leqslant x) \wedge (x \leqslant '9') &\equiv x \text{ — цифра.}\end{aligned} \tag{1.5}$$

3. Тип *char* содержит непечатаемый, пустой символ (пробел), который может использоваться как разделитель. (На рис. 1.1 пробелы обозначаются как ␣.)

Для написания программ в машинно-независимом виде особенно важно наличие функций преобразования между

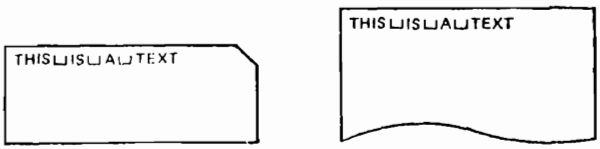


Рис. 1.1. Представления текста.

двумя стандартными типами *char* и *integer*. Мы называем эти функции $ord(c)$ — порядковый номер символа c в множестве *char* и $chr(i)$ — i-й символ множества *char*. Таким образом, *chr* — это обратная функция от *ord* и наоборот, т. е.

$$\begin{aligned}&ord(chr(i)) = i \quad \text{(если } chr(i) \text{ определена)},\\ &chr(ord(c)) = c.\end{aligned} \tag{1.6}$$

Особого внимания заслуживают функции

$$\begin{aligned}f(c) = ord(c) - ord('0') &= \text{положение } c \text{ среди цифр},\\ g(i) = chr(i + ord('0')) &= i\text{-я цифра.}\end{aligned} \tag{1.7}$$

Например, $f('3') = 3$, $g(5) = '5'$. Таким образом, f — обратная функция от g и наоборот, т. е.

$$\begin{aligned}&f(g(i)) = i \quad (0 \leqslant i \leqslant 9),\\ &g(f(c)) = c \quad ('0' \leqslant c \leqslant '9').\end{aligned} \tag{1.8}$$

Эти функции преобразования используются для перевода внутреннего представления чисел в последовательности цифр и наоборот. Они фактически и представляют собой такой перевод на простейшем уровне — для одной цифры.

1.5. ОГРАНИЧЕННЫЕ ТИПЫ

Часто бывает, что переменная принимает значения некоторого типа только в определенном интервале. Это можно выразить, определив переменную ограниченного типа, который описывается так:

$$\boxed{\textbf{type}\ T = min \ldots max} \qquad (1.9)$$

где *min* и *max* — границы интервала.

Примеры:

type *год* = 1900 .. 1999
type *буква* = 'A' .. 'Z'
type *цифра* = '0' .. '9'
type *офицер* = *лейтенант* .. *генерал*

Пусть даны переменные:

var *y*: *год*
var *L*: *буква*

тогда присваивания $y := 1973$ и $L := 'W'$ разрешены, а $y := 1291$ и $L := '9'$ не разрешены. Транслятор может проверять законность таких присваиваний только в том случае, если присваиваемое значение суть константа или переменная того же типа. Допустимость присваиваний вида

$$y := i \quad \text{и} \quad L := c$$

где i — типа *integer*, а c — типа *char*, можно проверить только во время выполнения программы. Системы, выполняющие такие проверки, оказались на практике чрезвычайно полезными для разработки программ. Использование ими избыточной информации для выявления возможных ошибок также является одной из основных причин применения языков высокого уровня.

1.6. МАССИВЫ

Массив — это, по-видимому, наиболее широко известная структура данных, так как во многих языках, включая Алгол-60 и Фортран, это единственная структура, которая существует в явном виде. Массив — это *регулярная* структура: все его компоненты — одного типа, называемого *базовым типом*. Массив — также структура с так называемым *случайным доступом*, все его компоненты могут выбираться произвольно и являются одинаково доступными. Для обозначения отдельной компоненты к имени всего массива

добавляется так называемый *индекс*, позволяющий выбрать компоненту. Индекс должен иметь значение типа, определенного как *тип индексов* массива. Описание регулярного типа T задает, таким образом, не только базовый тип T_0, но и тип индексов I:

$$\boxed{\textbf{type}\ T = \textbf{array}[I]\ \textbf{of}\ T_0} \tag{1.10}$$

Примеры:

```
type Row  = array[1 . . 5] of real
type Card = array[1 . . 80] of char
type alfa = array[1 . . 10] of char
```

Конкретное значение переменной

```
var x: Row
```

если каждая компонента удовлетворяет равенству $x_i = 2^{-i}$, может иметь вид, как показано на рис. 1.2.

x_1	0,5
x_2	0,25
x_3	0,125
x_4	0,0625
x_5	0,03125

Рис. 1.2. Массив типа *row*.

Составное значение x типа T со значениями компонент $c_1, \ldots, c_n$ может задаваться с помощью *конструктора* *) массива и оператора присваивания:

$$x := T(c_1, \ldots, c_n) \tag{1.11}$$

Операция, обратная конструктору, — *селектор*. Он позволяет выбрать из массива отдельную компоненту. Если в качестве переменной x рассматривать массив, то селектор массива обозначается с помощью имени массива, дополненного соответствующим индексом компоненты i:

$$\boxed{x[i]} \tag{1.12}$$

При работе с массивами, особенно большими, обычно выборочно изменяют отдельные компоненты, а не строят заново

*) В языке Паскаль такие конструкторы отсутствуют. — *Прим. ред.*

все составное значение. При этом переменная-массив рассматривается как массив составляющих переменных и допускается присваивание значения отдельным компонентам.

Пример:

$$x[i] := 0.125$$

Хотя при выборочном присваивании меняется только значение отдельной компоненты, с точки зрения построения концепции следует считать, что изменилось все составное значение.

То, что индексы массива, т. е. «имена» его компонент, должны быть определенного (скалярного) типа, имеет весьма важные следствия. Индексы могут вычисляться, вместо индексной константы можно использовать индексное *выражение*. Значение этого выражения вычисляется, и результат определяет выбираемую компоненту. Такая общность дает не только одно из важнейших и мощных средств программирования, но и приводит к одной из самых частых ошибок, так как полученное значение выражения может не попасть в интервал, заданный в качестве диапазона для индексов данного массива. Мы будем считать, что в случае такого ошибочного обращения к несуществующей компоненте массива адекватная вычислительная система выдает предупреждение.

Обычно тип индексов должен быть скалярным, т. е. неструктурированным, типом, на котором определено отношение порядка. Если базовый тип массива также упорядоченный, то на таком регулярном типе имеется естественное отношение порядка. Для двух массивов упорядочение определяется с помощью сравнения компонент с наименьшими индексами. Формально это можно описать следующим образом:

Если имеются два массива x и y, то отношение $x < y$ выполняется в том и только том случае, если существует индекс k такой, что

$$x[k] < y[k] \quad \text{и} \quad x[i] = y[i] \quad \text{для всякого } i < k. \quad (1.13)$$

Например,

$$(2, 3, 5, 7, 9) < (2, 3, 5, 7, 11)$$
$$'LABEL' < 'LIBEL'$$

Однако обычно считается, что массивы никак не упорядочены.

Кардинальное число составного типа равно произведению кардинальных чисел типов его компонент. Поскольку все компоненты регулярного типа A принадлежат к одному и тому

же базовому типу B, мы получим

$$\text{cardinality}(A) = (\text{cardinality}(B))^n \tag{1.14}$$

где $n = \text{cardinality}(I)$, а I — тип индексов массива.

В следующем небольшом фрагменте программы показано использование селектора для массива. Цель этой программы — найти наименьший индекс i компоненты со значением x. Поиск выполняется с помощью последовательного просмотра массива a, описанного как

```
var a: array[1..N] of T; {N > 0}
i := 0;
repeat i := i + 1 until (a[i] = x) ∨ (i = N);
if a[i] ≠ x then «в a нет такого элемента»
```
(1.15)

В другом варианте этой программы применяется распространенный прием *фиктивного элемента*, или *барьера*, расположенного в конце массива. Использование барьера позволяет упростить условие окончания цикла:

```
var a: array[1..N + 1] of T;
i := 0; a[N + 1] := x;
repeat i := i + 1 until a[i] = x;
if i > N then «в a нет такого элемента»
```
(1.16)

Присваивание $a[N+1] := x$ является примером *выборочного изменения*, т. е. изменения отдельной компоненты составной переменной. В обеих версиях (1.15) и (1.16) основным условием, выполняющимся вне зависимости от того, сколько раз выполняется оператор $i := i + 1$, является

$$a[j] \neq x \quad \text{для} \quad j = 1 \ldots i - 1$$

Поэтому оно называется *инвариантом цикла*.

Разумеется, поиск можно значительно ускорить, если компоненты уже упорядочены (рассортированы). В этом случае чаще всего применяется метод повторного деления пополам интервала, в котором ищется нужный элемент. Такой прием называется *методом деления пополам* или *бинарным поиском*, он показан в программе (1.17). При каждом повторении просматриваемый интервал между индексами i и j делится пополам. Поэтому максимальное число требующихся сравнений равно $\lceil \log_2(N) \rceil$.

```
i := 1; j := N;
repeat k := (i+j) div 2;
  if x > a[k] then i := k+1 else j := k-1
until (a[k]=x) ∨ (i>j)
```
(1.17)

(Соответствующим инвариантным условием выхода из цикла является

$$a[h] < x \quad \text{для} \quad h = 1 \ldots i-1$$
$$a[h] \geqslant x \quad \text{для} \quad h = j+1 \ldots N$$

Следовательно, если программа заканчивается при $a[h] \neq x$, то не существует $a[h] = x$ для $1 \leqslant h \leqslant N$.)

Компоненты массива могут в свою очередь быть составными. Переменная-массив, компоненты которой являются массивами, называется *матрицей*. Например,

$$M\text{: }\mathbf{array}[1\ldots 10]\ \mathbf{of}\ Row$$

— это массив, состоящий из десяти компонент (строк), каждая из которых состоит из пяти компонент вещественного типа. Этот массив называется матрицей 10×5 с вещественными компонентами. Селекторы могут соответствующим образом следовать один за другим, так что

$$M[i][j]$$

обозначает j-ю компоненту строки $M[i]$, являющейся i-й компонентой M. Обычно это записывается короче, как

$$M[i, j]$$

и точно так же описание

$$M\text{: }\mathbf{array}[1\ldots 10]\ \mathbf{of}\ \mathbf{array}[1\ldots 5]\ \mathbf{of}\ real$$

можно записать проще, как

$$M\text{: }\mathbf{array}[1\ldots 10,\ 1\ldots 5]\ \mathbf{of}\ real$$

Если нужно выполнить некоторое действие со *всеми* компонентами массива или с расположенными подряд компонентами какой-то части массива, то для этого удобно использовать оператор цикла, как показано в следующем примере.

Пусть дробь f представляется с помощью массива d, так, что

$$f = \sum_{i=1}^{k-1} d_i * 10^{-i}$$

т. е. в десятичном виде с $k-1$ цифрами. Теперь пусть f нужно разделить на 2. Для этого обычную операцию деления производят со *всеми* $k-1$ цифрами d_i, начиная с $i = 1$. При этом деление цифры на 2 выполняется с учетом возможного переноса из предыдущей позиции, и на следующий шаг

передается возможный остаток (см. 1.18)

```
r := 10*r+d[i];
d[i] := r div 2;                    (1.18)
r := r-2*d[i]
```

Этот процесс используется в программе 1.1 для получения таблицы отрицательных степеней 2. Оператор цикла удобно использовать также и для деления пополам при вычислении 2^{-1}, 2^{-2}, ..., 2^{-n}; таким образом получается вложенность двух операторов цикла.

```
program power (output);
{десятичное представление отрицательных степеней двойки}
const n = 10;
type digit = 0..9;
var i,k,r: integer;
    d: array [1..n] of digit;
begin  for k := 1 to n do
    begin write('.'); r := 0;
        for i := 1 to k-1 do
        begin r := 10*r+d[i]; d[i] := r div 2;
              r := r-2*d[i]; write(chr(d[i]+ord('0')))
        end ;
        d[k] := 5; writeln('5')
    end
end .
```

Программа 1.1. Вычисление степеней двойки.

Результат для $n = 10$ имеет вид

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.7. ЗАПИСИ

Самый общий метод получения составных типов — это объединение компонент, принадлежащих к произвольным, возможно, тоже составным типам, в один составной тип. При-

меры из математики — это комплексные числа, состоящие из двух вещественных чисел, и координаты точек, состоящие из двух или более вещественных чисел в зависимости от размерности пространства, заданного системой координат. Пример из обработки данных — это описание людей с помощью нескольких существенных характеристик, таких, как имя и фамилия, дата рождения, пол и семейное положение.

В математике такой составной тип называется *декартовым произведением* типов компонент. Это связано с тем, что множество значений такого составного типа состоит из всех возможных комбинаций значений, взятых по одному из каждого типа компонент. Следовательно, число таких наборов из n чисел равно произведению количеств элементов всех составляющих множеств, так что кардинальное число составного типа равно произведению кардинальных чисел всех типов компонент.

В обработке данных комбинированные типы, такие, как описания людей или объектов, часто встречаются в файлах или «банках данных» и представляют собой записи существенных характеристик человека или объекта. Поэтому слово «запись» (**record**) стало широко принятым для обозначения подобной совокупности данных, и мы будем использовать этот термин вместо термина «декартово произведение».

В общем виде составной тип T определяется следующим образом:

```
type T = record s1: T1;
                s2: T2;
                ...
                sn: Tn
         end
```

(1.19)

$$\text{Cardinality}(T) = \text{cardinality}(T_1) * \ldots * \text{cardinality}(T_n)$$

Примеры:

```
type Complex = record re: real;
                      im: real
               end

type Date = record day: 1..31;
                   month: 1..12;
                    year: 1..2000
            end
```

```
type Person = record name: alfa;
                     firstname: alfa;
                     birthdate: Date;
                     sex: (male, female);
                     marstatus: (single, married,
                        widowed, divorced)
              end
```

Значение типа T можно строить с помощью *конструктора записи* *) и, следовательно, присваивать его переменной этого типа:

$$x := T(x_1, x_2, \ldots, x_n) \tag{1.20}$$

где x_i — значение типа компоненты T_i.

Пусть даны переменные-записи:

```
z: Complex
d: Date
p: Person
```

им могут присваиваться отдельные значения, например, следующим образом (см. рис. 1.3):

```
z := Complex (1.0, — 1.0)
d := Date (1, 4, 1973)
p := Person ('WIRTH','CHRIS', Date(18, 1, 1966), male, single)
```

Идентификаторы $s_1, \ldots, s_n$, которые вводятся при определении комбинированного типа, являются именами отдельных

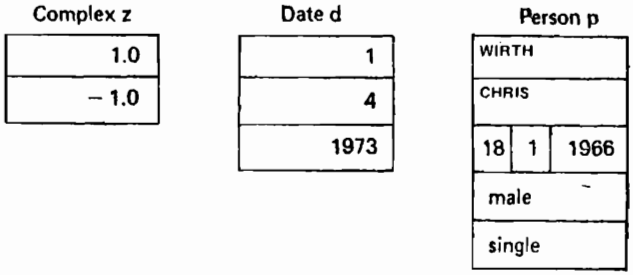


Рис. 1.3. Записи типов *Complex*, *Date*, *Person*.

компонент переменных этого типа, они употребляются в *селекторах записи*, где их добавляют к переменной, обозначающей всю запись. Если имеется переменная x: T, то ее

*) В языке Паскаль такие конструкторы также отсутствуют. — *Прим. ред.*

i-я компонента обозначается как

$$x.s_i \qquad (1.21)$$

Если такой селектор стоит в левой части оператора присваивания, то происходит выборочное изменение x:

$$x.s_i := x_i$$

где x_i — значение выражения типа T_i.

Если даны переменные:

z: *Complex*
d: *Date*
p: *Person*

то их можно использовать, например, со следующими селекторами:

z.im	(типа *real*)
d.month	(типа 1..12)
p.name	(типа *alfa*)
p.birthdate	(типа *Date*)
p.birthdate.day	(типа 1..31)

На примере типа *Person* мы видим, что компоненты записи могут в свою очередь быть составными. Таким образом, селекторы могут добавляться один к другому. Кроме того, разные составные типы могут комбинироваться различными способами. Например, i-я компонента массива a, который является компонентой записи r, обозначается как

$$r.a\,[i]$$

а компонента с селектором s, входящая в i-ю компоненту-запись массива записей a, обозначается как

$$a\,[i].s$$

Декартово произведение в принципе содержит *все* комбинации значений типов компонент. Однако следует заметить, что на практике не все такие комбинации могут быть «законными», т. е. иметь смысл. Например, тип *Date*, определенный выше, включает значения

(31, 4, 1973) и (29, 2, 1815)

хотя дней с такими датами не существует. Таким образом, определение этого типа не отражает реального положения

вещей. Все же оно достаточно близко к практическим целям, и ответственность за то, чтобы при выполнении программы не возникали подобные бессмысленные значения, возлагается на программиста.

В следующем небольшом фрагменте программы показано использование записей. Его задача — сосчитать число «людей» в массиве a, которые одновременно принадлежат женскому полу и одиноки:

```
var a: array[1 .. N] of Person;
    count: integer;
count := 0;                                                    (1.22)
for i := 1 to N do
    if (a[i].sex = female) ∧ (a[i].marstatus = single) then
        count := count+1
```

Инвариант цикла здесь

$$count = C(i)$$

где $C(i)$ — число одиноких женщин в подмножестве $a_1, \ldots, a_i$.

В другом варианте записи этого оператора используется конструкция, которая называется оператором присоединения:

```
for i := 1 to N do
    with a[i] do
        if (sex = female) ∧ (marstatus = single) then          (1.23)
            count := count+1
```

Выражение **with** r **do** s означает, что внутри оператора s селекторы переменной r можно использовать без префикса: считается, что все они ссылаются на переменную r. Таким образом, оператор присоединения позволяет сократить текст программы, а также предотвращает повторное вычисление адреса индексированной компоненты $a[i]$.

В следующем примере мы предполагаем, что некоторые группы людей в массиве a чем-то объединены (возможно, чтобы их можно было быстрее находить). Связующая информация выражается дополнительной компонентой записи *Person*, называемой *link* (связь). Эти компоненты соединяют записи в линейный список, так что для каждого человека легко можно найти предшествующую и последующую записи. Интересно, что при таком методе связывания можно легко просматривать список в обоих направлениях, используя только одно число, хранящееся в каждой записи. Это делается следующим образом.

Предположим, что индексы трех последовательных элементов списка есть i_{k-1}, i_k, i_{k+1}. Значение *link* для k-го элемента берется равным $i_{k+1} - i_{k-1}$. При проходе по списку вперед i_{k+1} определяется двумя текущими индексными переменными $x = i_{k-1}$ и $y = i_k$ по формуле

$$i_{k+1} = x + a[y].link$$

а при проходе по списку в обратном направлении i_{k-1} определяется с помощью $x = i_{k+1}$ и $y = i_k$ по формуле

$$i_{k-1} = x - a[y].link$$

Пример объединения при помощи *link* всех лиц одного пола показан в табл. 1.2.

Таблица 1.2. Массив элементов типа *Person*

	First Name	Sex	Link
1	Carolyn	F	2
2	Chris	M	2
3	Tina	F	5
4	Robert	M	3
5	Jonathan	M	3
6	Jennifer	F	5
7	Raytheon	M	5
8	Mary	F	3
9	Anne	F	1
10	Mathias	M	3

Запись и массив имеют общее свойство: оба являются структурами со «случайным доступом». Запись — более универсальная структура, поскольку не требуется, чтобы типы всех ее компонент были одинаковы. С другой стороны, массив предоставляет бо́льшие возможности, так как селекторы его компонент могут вычисляться (если они представлены выражениями), тогда как селекторы компонент записи — это фиксированные идентификаторы, задаваемые в описании типа.

1.8. ЗАПИСИ С ВАРИАНТАМИ

В практической работе часто кажется удобным и естественным рассматривать два типа как *варианты* одного и того же типа. Например, тип *Coordinate*, введенный в предыдущем разделе, можно рассматривать как объединение двух

вариантов: декартовых и полярных координат, компонентами которых являются соответственно (a) две длины и (b) длина и угол. Для того чтобы определить, какой вариант принят в данный момент, вводится третья компонента. Она называется *дискриминантом типа* или *полем признака.*

```
type Coordinate =
    record case kind: (Cartesian, polar) of
                  Cartesian: (x, y: real);
                  polar: (r: real; φ: real)
    end
```

Здесь имя поля признака — *kind*, а имена координат — либо x и y в случае значения *Cartesian* (декартовы), либо r и φ в случае значения *polar* (полярные).

Множество значений типа *Coordinate* есть объединение двух типов:

$$T_1 = (x, y: real)$$
$$T_2 = (r: real; \varphi: real)$$

а его кардинальное число равно сумме кардинальных чисел T_1 и T_2.

Однако чаще всего приходится объединять не два полностью различных типа, а два типа с частично совпадающими компонентами. Для такой ситуации применяется термин «запись с вариантами». Примером может служить тип *Person* определенный в предыдущем разделе, если существенные характеристики должны записываться в файл в зависимости от пола. Например, для мужчины могут считаться в какой-то определенной ситуации существенными такие признаки, как вес и наличие бороды, а для женщины можно считать важными три ее основных размера (тогда как вес она может хранить в тайне). Исходя из этих допущений, получим следующее описание типа:

```
type Person =
    record name, firstname: alfa;
           birthdate: Date;
           marstatus: (single, married, widowed, divorced);
       case sex: (male, female) of
       male: (weight: real;
              bearded: Boolean);
       female: (size: array[1 . . 3] of integer)
    end
```

Общий вид описания составного типа с вариантами:

$$
\begin{array}{l}
\textbf{type } T = \\
\quad \textbf{record } s_1 : T_1; \ldots; s_{n-1} : T_{n-1}; \\
\quad\quad \textbf{case } s_n : T_n \textbf{ of} \\
\quad\quad\quad c_1 : (s_{1,1} : T_{1,1}; \ldots; s_{1,n_1} : T_{1,n_1}); \\
\quad\quad\quad\quad \ldots\ldots\ldots\ldots \\
\quad\quad\quad c_m : (s_{m,1} : T_{m,1}; \ldots; s_{m,n_m} : T_{m,n_m}) \\
\quad\quad \textbf{end}
\end{array}
\qquad (1.24)
$$

Здесь s_i и s_{ij} — селекторы компонент, принадлежащих к типам компонент T_i и T_{ij}, а s_n — имя различающего поля признака типа T_n. Переменная x типа T состоит из компонент

$$x.s_1,\ x.s_2,\ \ldots,\ x.s_n,\ x.s_{k,1},\ \ldots,\ x.s_{k,n_k}$$

в том и только том случае, когда текущее значение $x.s_n = c_k$. Компоненты $x.s_1, \ldots, x.s_n$ составляют *общую часть* m вариантов.

Таким образом, использование селектора $x.s_{k,h}$ $(1 \leqslant h \leqslant n_k)$ при $x.s_n \neq c_k$ следует рассматривать как серьезную ошибку программирования. Это может, например, означать (для типа *Person*, определенного выше,) проверку, является ли некая леди бородатой, или (в случае выборочного присваивания) приписывание ей этого свойства!

Поэтому при использовании записей с вариантами требуется особое внимание. Лучше всего действия, связанные с каждым из вариантов, группировать в выбирающем селекторе, так называемом *операторе варианта*; его структура отражает структуру описания типа записи с вариантами:

$$
\begin{array}{l}
\textbf{case } x.s_n \textbf{ of} \\
\quad c_1 : S_1; \\
\quad c_2 : S_2; \\
\quad\quad \ldots \\
\quad c_m : S_m \\
\textbf{end}
\end{array}
\qquad (1.25)
$$

Оператор S_k выполняется в случае, когда для x выбирается k-й вариант, т. е. поле признака $x.s_n$ принимает значение c_k. Следовательно, для того чтобы предотвратить неправильное использование селекторов, нужно следить, чтобы каждый S_k содержал только селекторы

$$x.s_1, \ldots, x.s_{n-1}$$

и

$$x.s_{k,1}, \ldots, x.s_{k,n_k}$$

В следующем небольшом фрагменте программы вычисляется расстояние между двумя точками A и B, заданными переменными a и b типа *Coordinate* (запись с вариантами). Способ вычисления выбирается в зависимости от четырех воз-

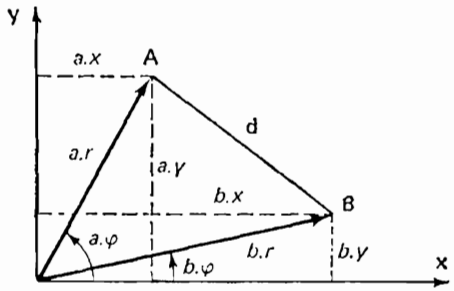


Рис. 1.4. Декартовы и полярные координаты.

можных комбинаций декартовых и полярных координат (см. рис. 1.4).

```
case a.kind of
   Cartesian: case b.kind of
                 Cartesian: d := sqrt(sqr(a.x-b.x)+sqr(a.y-b.y));
                 Polar:     d := sqrt(sqr(a.x-b.r*cos(b.φ)
                                     +sqr(a.y-b.r*sin(b.φ))
              end;
   Polar:     case b.kind of
                 Cartesian: d := sqrt(sqr(a.r*cos(a.φ)-b.x)
                                     +sqr(a.r*sin(a.φ)-b.y));
                 Polar:     d := sqrt(sqr(a.r)+sqr(b.r)
                                     -2*a.r*b.r*cos(a.φ-b.φ))
              end
end
```

1.9. МНОЖЕСТВО

Кроме массива и записи имеется третья фундаментальная структура данных — *множество*. Соответствующий тип описывается следующим образом:

$$\boxed{\textbf{type } T = \textbf{set of } T_0} \qquad (1.26)$$

Значениями переменной x типа T являются множества элементов типа T_0. Множество всех подмножеств множества T_0 называется *множеством-степенью* T_0. Таким образом, тип T — это множество-степень своего *базового типа* T_0.

Примеры:

```
type intset = set of 0..30
type charset = set of char
type tapestatus = set of exception
```

Во втором примере базовым типом является стандартное подмножество символов — тип *char*, в третьем примере — тип исключительных состояний магнитных лент, описанный как скалярный тип:

```
type exception = (unloaded, manual, parity, skew)
```

значения которого соответствуют различным состояниям устройства лентопротяжек. Если даны переменные:

```
is : intset
cs : charset
t  : array[1..6] of tapestatus
```

то формировать и присваивать значения переменных-множеств можно, скажем, так: *)

```
is := [1, 4, 9, 16, 25]
cs := ['+', '-', '*', '/']
t[3] := [manual]
t[5] := [ ]
t[6] := [unloaded..skew]
```

Здесь значение, присваиваемое $t[3]$,— это множество, состоящее из одного элемента *manual*, $t[5]$ присваивается пустое множество, что соответствует рабочему состоянию 5 лентопротяжки (какие-либо исключительные состояния отсутствуют), а $t[6]$ присваивается значение множества, включающего все четыре исключительных состояния.

Кардинальное число множества типа T равно

$$\text{cardinality}(T) = 2^{\text{cardinality}(T_0)}. \qquad (1.27)$$

Это следует из того, что каждый из элементов cardinality (T_0) представляется в множестве одним из двух значений: «присутствует» или «отсутствует», и эти элементы входят в множество независимо друг от друга. Очевидно, что для эффективной и экономной реализации не только базовый тип

*) В отличие от принятой нотации мы используем для множества не фигурные, а квадратные скобки. В фигурные скобки заключаются комментарии в программах.

множества должен быть конечным, но и его кардинальное число — достаточно небольшим.

На всех множествах определены следующие элементарные операции:

*	пересечение множеств
+	объединение множеств
—	разность множеств
in	принадлежность множеству

Пересечение и объединение двух множеств часто называют соответственно *умножением* и *сложением* множеств; соответствующим образом определены приоритеты операций: операция пересечения имеет приоритет перед операциями объединения и разности, а они в свою очередь имеют приоритет перед операцией принадлежности. Операция принадлежности относится к классу операций отношений. Ниже приведены примеры выражений с множествами и полностью эквивалентные им выражения со скобками:

$$r * s + t = (r * s) + t$$
$$r - s * t = r - (s * t)$$
$$r - s + t = (r - s) + t$$
$$x \textbf{ in } s + t = x \textbf{ in } (s + t)$$

Наш первый пример использования множеств — программа простого сканера в трансляторе. Сканер — это процедура, задача которой — преобразовать последовательность символов в последовательность текстовых единиц транслируемого языка, так называемых *лексем*. При каждом вызове сканер считывает нужное число входных символов и выдает одну выходную лексему. Конкретные правила трансляции следующие:

1. Имеются следующие выходные лексемы: *идентификатор*, *число*, *меньше-равно*, *больше-равно*, *присвоить* — и лексемы, соответствующие отдельным символам, таким, как +, —, * и т. д.
2. Лексема *идентификатор* выдается по прочтении последовательности букв и цифр, начинающейся с буквы.
3. Лексема *число* выдается по прочтении последовательности цифр.
4. Лексемы *меньше-равно*, *больше-равно* и *присвоить* выдаются по прочтении соответствующих пар символов $<=$, $>=$, $:=$.
5. Пробелы и концы строк опускаются.

В нашем распоряжении имеется простая процедура *read*(*x*), которая читает очередной символ из входной после-

довательности и присваивает его переменной x. Полученная выходная лексема присваивается глобальной переменной *sym*. Кроме того, имеются глобальные переменные *id* и *num*, назначение которых будет видно из программы 1.2, а также *ch*, содержащая текущий символ входной последовательности. Массив лексем S задает отображение символов в лексемы, его индексы ограничены лишь теми символами, которые не являются ни цифрами, ни буквами. Как мы видим, использование множеств символов позволяет программировать сканер независимо от их упорядоченности.

Второй пример использования множеств — программа составления школьного расписания. Предположим, что каждый из M учеников выбирает для изучения какие-либо предметы из их общего числа N. Теперь нужно так построить расписание, чтобы можно было некоторые предметы читать одновременно и при этом не возникало бы конфликтов [1.1].

В принципе построение расписания — сложная комбинаторная задача. При ее решении нужно учитывать много различных факторов. Но в этом примере мы значительно упростим задачу и отвлечемся от реальной ситуации, для которой составляется расписание.

Прежде всего для того чтобы решить, какие предметы можно читать в одно и то же время, нужно проанализировать индивидуальные списки выбранных предметов, составленные учениками. Эти списки представляют собой перечисления предметов, которые нельзя читать одновременно. Поэтому вначале мы программируем процесс сокращения данных. Ученикам присваиваются номера от 1 до M, а предметам — от 1 до N.

```
type course = 1..N;
     student = 1..M;
     selection = set of course;
var s: course;
    i: student;
    registration: array[student] of selection;
    conflict: array[course] of selection;                    (1.28)
{Определение множества курсов, вступающих в конфликт,
 по спискам курсов, выбранных отдельными учащимися}
for s := 1 to N do conflict[s] := [ ];
for i := 1 to M do
   for s := 1 to N do
   if s in registration[i] then
      conflict[s] := conflict[s] + registration[i]
```

(Заметим, что из этого алгоритма следует s **in** *conflict*[s].)

```
var ch: char;
    sym: symbol;
    num: integer;
    id: record
          k: 0..maxk;
          a: array [1..maxk] of char
        end ;
procedure scanner;
   var ch1: char;
begin {пропуск пробелов}
   while ch = '␣' do read(ch);
   if ch in ['A'..'Z'] then
      with id do
      begin sym := identifier; k := 0;
         repeat if k < maxk then
                   begin k := k+1; a[k] := ch
                   end ;
            read(ch)
         until ¬(ch in ['A'..'Z' , '0'..'9'])
      end else
   if ch in ['0'..'9'] then
      begin sym := number; num := 0;
         repeat num := 10*num+ord(ch)-ord('0');
            read(ch)
         until ¬(ch in ['0'..'9'])
      end else
   if ch in ['<', ':', '>'] then
      begin ch1 := ch; read(ch);
         if ch = '=' then
         begin
            if ch1 = '<' then sym := leq else
            if ch1 = '>' then sym := geq else sym := becomes;
            read(ch)
         end
         else sym := S[ch1]
      end else
   begin {другие символы}
      sym := S[ch]; read(ch)
   end
end {scanner}
```

Программа 1.2. Сканер.

Основная теперь задача — составить расписание, т. е. список читаемых предметов, так, чтобы они следовали в нужном порядке и не противоречили друг другу. Из множества всех курсов мы выбираем подмножества «неконфликтующих» предметов. Подмножества выбираются из переменной *remaining*, до тех пор пока множество оставшихся предметов не станет пустым.

```
var k: integer;
    remaining, session: selection;
    timetable: array[1 .. N] of selection;
k := 0; remaining := [1 .. N];
while remaining ≠ [ ] do                                    (1.29)
  begin session := следующая выборка;
        remaining := remaining − session;
        k := k+1; timetable[k] := session
  end
```

Как определяется «следующая выборка»? Вначале берется любой из множества оставшихся предметов. Затем из этого множества выбираются все такие предметы, которые «не конфликтуют» с выбранными ранее. Назовем множество таких предметов *trialset*. Затем будем исследовать каждый элемент множества *trialset*. Включение такого элемента в *session* зависит от того, пусто или нет пересечение множества предметов, уже включенных в *session*, с множеством предметов, конфликтующих с данным. Оператор «*session* := := *следующая выборка*» принимает вид

```
var s,t: course;
    trialset: selection;
begin s := 1;
      while ¬(s in remaining) do s := s+1;
      session := [s]; trialset := remaining − conflict[s];
      for t := 1 to N do                                    (1.30)
        if t in trialset then
        begin if conflict[t] * session = [ ] then
                session := session + [t]
        end
end
```

Конечно, такой способ выбора параллельно читаемых предметов не позволяет строить расписание оптимальным образом. В неудачных случаях множество выборок параллельных курсов может оказаться столь же велико, как и множество всех курсов, даже если существуют курсы, которые можно было бы читать параллельно.

1.10. ПРЕДСТАВЛЕНИЕ МАССИВОВ, ЗАПИСЕЙ И МНОЖЕСТВ

Основная цель использования абстракций в программировании — обеспечить разработку, анализ и проверку программы на основе законов, управляющих этими абстракциями. При этом нет необходимости знать, какими способами эти абстракции реализуются на конкретной вычислительной машине. Однако квалифицированному программисту полезно разбираться в наиболее часто применяемых приемах представления основных абстракций программирования — таких, как фундаментальные структуры данных. Эти знания могут помочь программисту строить программу и описывать данные с учетом не только абстрактных свойств структур, но и их реализации на конкретной ЭВМ, принимая во внимание ее свойства и присущие ей ограничения.

Проблема представления данных есть проблема отображения абстрактной структуры в память вычислительной машины. В первом приближении эта память представляет собой массив отдельных ячеек памяти, называемых словами. Индексы этих слов называются адресами:

var *store*: **array**[*address*] **of** *word* (1.31)

Кардинальные числа типов *address* и *word* различны для разных вычислительных машин. Особенная сложность заключается в разнообразии кардинальных чисел для слова. Логарифм кардинального числа называется *размером слова*, поскольку он равен количеству разрядов, из которых состоит ячейка памяти.

1.10.1. Представление массивов

Представление массива — это отображение (абстрактного) массива компонент типа T в память, которая представляет собой массив компонент типа *word*.

Массив следует отображать таким способом, чтобы можно было максимально просто и потому эффективно вычислять адреса его компонент. Адрес, или индекс памяти, i j-й компоненты массива вычисляется с помощью линейной функции отображения

$$i = i_0 + j * s \tag{1.32}$$

где i_0 — адрес первой компоненты, а s — число слов, которые «занимает» компонента. Так как по определению слово есть минимальная доступная единица памяти, то, по-видимому, желательно, чтобы s было целым числом; в простейшем случае $s = 1$. Если s — не целое число слов памяти (а так бывает довольно часто), то s обычно округляется до

Таблица 1.3. Фундаментальные структуры данных

Структура	Описание	Селектор	Доступ к компонентам с помощью	Типы компонент	Кардинальное число
Массив	a: **array**[I] **of** T_0	$a[i]$ $(i \in I)$	Селектора с вычисляемым индексом i	Все компоненты одного типа T_0	$card(T_0)^{card(I)}$
Запись	r: **record** s_1: T_1; s_2: T_2; . . . s_n: T_n **end**	$r.s$ $(s \in [s_1 \ldots s_n])$	Селектора с именем компоненты поля	Могут быть различными	$\prod_{i=1}^{n} card(T_i)$
Множество	s: **set-of** T_0	Отсутствует	Проверки принадлежности с операцией отношения **in**	Все компоненты одного типа (скалярного T_0)	$2^{card(T_0)}$

ближайшего большего целого числа $\lceil s \rceil$. В этом случае каждая компонента массива занимает $\lceil s \rceil$ слов, причем часть слова величиной $\lceil s \rceil - s$ остается неиспользованной (см.

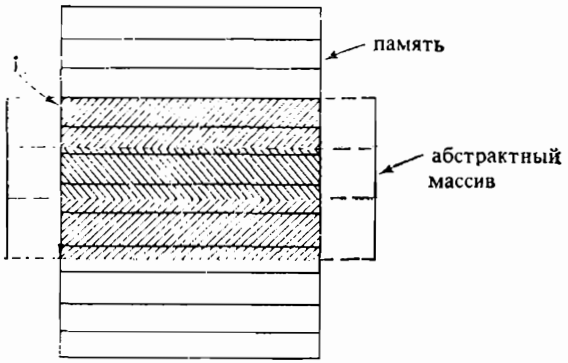


Рис. 1.5. Отображение массива в память.

рис. 1.5 и 1.6). Округление числа занимаемых слов до ближайшего целого называется *выравниванием*. Отношение размера памяти, которая отводится для описания структуры

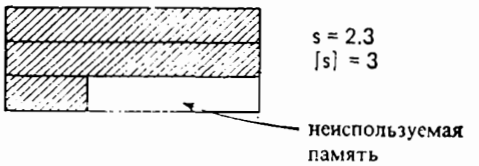


Рис. 1.6. Представление записи с выравниванием.

данных, к размеру действительно занятой памяти называется коэффициентом использования памяти:

$$u = \frac{s}{s'} = \frac{s}{\lceil s \rceil}. \tag{1.33}$$

С одной стороны, разработчик стремится получить коэффициент использования памяти, близкий к 1. Но поскольку, с другой стороны, доступ к частям слова — неясный и довольно неэффективный процесс, разработчику приходится идти на некоторый компромисс. При этом он должен учитывать следующие обстоятельства:

1. Выравнивание понижает коэффициент использования памяти.
2. Отказ от выравнивания может привести к неэффективному обращению к частям слова.

3. Обращение к частям слова может удлинить программу (оттранслированную) и этим свести на нет выигрыш, достигнутый отказом от выравнивания.

В действительности положения 2 и 3 обычно более важны, и трансляторы всегда автоматически применяют выравнивание. Заметим, что при $s > 0.5$ коэффициент использования памяти всегда будет $u > 0.5$. Однако, если $s \leqslant 0.5$, этот

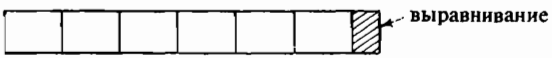


Рис. 1.7. Упаковка шести компонент в одно слово

коэффициент можно значительно увеличить, помещая в каждое слово более одной компоненты массива. Этот прием называется *упаковкой*. Если в слово упаковано n компонент, то коэффициент использования памяти равен (см. рис. 1.7)

$$u = \frac{n \cdot s}{[n \cdot s]}. \tag{1.34}$$

Доступ к i-й компоненте упакованного массива требует вычисления j — адреса слова, в котором расположена эта компонента, а также k — относительного адреса расположения компоненты внутри слова:

$$\begin{aligned} j &= i \ \mathbf{div}\ n \\ k &= i \ \mathbf{mod}\ n = i - j * n \end{aligned} \tag{1.35}$$

Большинство языков программирования не дает программисту возможности управлять реализацией абстрактных структур данных. Однако полезно иметь возможность указывать желательность упаковки хотя бы в тех случаях, когда в одно слово можно поместить более одной компоненты, поскольку при этом достигается экономия памяти в 2 и более раз. Мы вводим соглашение, что желательность упаковки будет обозначаться словом **packed** перед словом **array** (или **record**).

Пример:

type *alfa* = **packed array**[1 . . *n*] **of** *char*

Эта особенность наиболее ценна для вычислительных машин с длинными словами и сравнительно удобным доступом к отдельным частям слов. Важное свойство этого префикса состоит в том, что он не изменяет значение и правильность программы. Это означает, что можно выбирать любое из альтернативных представлений с уверенностью, что это не повлияет на смысл программы.

Обычно можно существенно уменьшить затраты на доступ к компонентам упакованного массива, если сразу распаковать (или упаковать) весь массив целиком. Дело в том, что при этом возможен эффективный последовательный проход по всему массиву и пропадает необходимость вычислять сложную функцию отображения для каждой отдельной компоненты. Поэтому мы вводим две стандартные процедуры: *pack* (упаковать) и *unpack* (распаковать). Пусть имеются переменные

$$u: \textbf{array}[a..d]\ \textbf{of}\ T$$
$$p: \textbf{packed array}[b..c]\ \textbf{of}\ T$$

где $a \leqslant b \leqslant c \leqslant d$ — одного и того же скалярного типа. Тогда

$$pack(u, i, p) \quad (a \leqslant i \leqslant b - c + d) \tag{1.36}$$

эквивалентно

$$p[j] := u[j + i - b], \quad j = b..c$$

а

$$unpack(p, u, i) \quad (a \leqslant i \leqslant b - c + d) \tag{1.37}$$

эквивалентно

$$u[j + i - b] := p[j], \qquad j = b..c$$

1.10.2. Представление записей

Записи отображаются в память (размещаются) так, что их компоненты располагаются последовательно. Адрес какой-либо компоненты (поля) r_i относительно начального адреса записи r называется *смещением* компоненты k_i. Оно вычисляется следующим образом:

$$k_i = s_1 + s_2 + \ldots + s_{i-1}, \tag{1.38}$$

где s_j — размер в словах j-й компоненты. Поскольку у массива все компоненты одного типа, то

$$s_1 = s_2 = \ldots = s_n$$

и, следовательно,

$$k_i = s_1 + s_2 + \ldots + s_{i-1} = (i - 1) \cdot s.$$

Универсальность записи не позволяет вычислять относительные адреса ее компонент с помощью такой же простой линейной функции, поэтому очень полезно ограничить доступ к ее компонентам и пользоваться лишь фиксированными идентификаторами. Это ограничение позволяет узнать относительные адреса во время трансляции, что намного увеличивает эффективность доступа к полям записи.

Если несколько компонент записи умещаются в одном слове памяти, то может идти речь об упаковке (см. рис. 1.8). Так же как для массива, желательность упаковки можно указать в описании с помощью слова **packed** перед словом **record**. Поскольку смещения компонент вычисляются при трансляции, смещение компоненты внутри слова также мо-

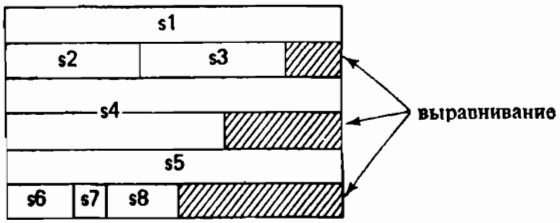


Рис. 1.8. Представление упакованной записи.

жет определяться транслятором. Это значит, что для многих машин упаковка записей приводит к значительно меньшей потере эффективности, чем упаковка массивов.

1.10.3. Представление множеств

Множество s наилучшим образом представляется в памяти машины с помощью *характеристической функции* $C(s)$. Характеристическая функция — это массив логических значений, i-я компонента которого означает наличие или отсутствие i-го значения базового типа в множестве s. Размер этого массива равен кардинальному числу базового типа множества:

$$C(s_i) \equiv (i \text{ in } s) \tag{1.39}$$

Например, множество небольших целых чисел

$$s = [1, 4, 8, 9]$$

представляется последовательностью логических значений F(false) и T(true)

$$C(s) = (FTFFTFFFTT)$$

если базовый тип множества s — целые числа, принадлежащие диапазону 0..9. В памяти машины последовательность логических значений изображается так называемой *строкой разрядов* (см. рис. 1.9).

Представление множеств их характеристической функцией позволяет реализовать операции объединения, пересечения и разности двух множеств с помощью элементарных логических операций. Для любого элемента i, принадлежащего

к базовому типу множеств x и y, имеют место следующие эквивалентности между операциями над множествами и логическими операциями:

$$\begin{aligned} i \textbf{ in } (x+y) &\equiv (i \textbf{ in } x) \vee (i \textbf{ in } y) \\ i \textbf{ in } (x*y) &\equiv (i \textbf{ in } x) \wedge (i \textbf{ in } y) \\ i \textbf{ in } (x-y) &\equiv (i \textbf{ in } x) \wedge \neg(i \textbf{ in } y) \end{aligned} \tag{1.40}$$

Такие логические операции имеются на всех вычислительных машинах, более того, они выполняются *одновременно* над всеми элементами (разрядами) слова. Поэтому для более

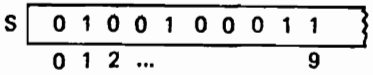


Рис. 1.9. Представление множества в виде разрядной строки.

эффективной реализации основных операций над множествами желательно, чтобы множеству соответствовало небольшое фиксированное число слов, над которыми кроме основных логических операций можно было бы выполнять также операции сдвига. В этом случае проверка принадлежности выполняется с помощью сдвига и последующей проверки знакового разряда. Следовательно, проверку

$$x \textbf{ in } [c_1, c_2, \ldots, c_n]$$

можно реализовать значительно эффективнее, чем при помощи эквивалентного булевского выражения

$$(x = c_1) \vee (x = c_2) \vee \ldots \vee (x = c_n)$$

Поэтому следует использовать множества только с *небольшими базовыми типами*. Наибольшее значение кардинального числа базового типа, при котором реализация достаточно эффективна, зависит от длины слова соответствующей вычислительной машины. Разумеется, в этом отношении предпочтительны машины с большой длиной слова. Если размер слова сравнительно невелик, можно использовать несколько слов.

1.11. ПОСЛЕДОВАТЕЛЬНЫЙ ФАЙЛ

Общее свойство структур данных, которые до сих пор обсуждались, а именно массива, записи и множества, заключается в том, что *их кардинальное число конечно*. Предполагается, что кардинальные числа типов их компонент конечны. Поэтому они не слишком трудны для реализации; со-

ответствующее представление легко находится для любой вычислительной машины.

Большинство так называемых усложненных структур: последовательности, деревья, графы и т. д. — характеризуются тем, что их кардинальные числа бесконечны. Это отличие от базовых структур с конечными кардинальными числами очень важно и имеет существенные практические следствия. Например, определим *последовательную* структуру следующим образом.

Последовательность с базовым типом T_0 — это либо пустая последовательность, либо конкатенация последовательности (с базовым типом T_0) и значения типа T_0.

Тип T, определенный таким образом, содержит бесконечное число значений. Каждое отдельное значение состоит из конечного числа компонент, но это число не ограничено, т. е. для каждой данной последовательности можно построить более длинную.

Аналогичные рассуждения применимы ко всем другим усложненным структурам данных. Из этого прежде всего следует, что объем памяти, необходимый для размещения структуры усложненного типа, неизвестен во время трансляции и может изменяться во время выполнения программы. Это требует *динамического распределения памяти*, при котором память занимается, если соответствующие значения «растут», и, возможно, освобождается, когда они «убывают». Поэтому проблема представления усложненных структур — чрезвычайно тонкая и сложная, и ее решение существенно влияет на эффективность работы и экономию памяти. Здесь можно принимать решения только с учетом того, какие простейшие операции и насколько часто должны выполняться на данной структуре. Поскольку эта информация неизвестна разработчику языка и транслятора, ему приходится исключать усложненные структуры из языка (универсального). Отсюда также следует, что программист должен избегать использования таких структур, если при решении задачи можно ограничиться фундаментальными структурами данных.

В большинстве языков и трансляторов учитывается и используется тот факт, что все усложненные структуры состоят либо из неструктурированных элементов, либо из фундаментальных структур. Это позволяет использовать преимущества усложненных структур, не имея информации об их возможном применении. Если в языке имеются средства для динамического размещения компонент, для динамической связи компонент и ссылки на них, то в нем могут создаваться произвольные структуры с помощью явных операций, определяемых программистом. Способы создания таких структур и работы с ними рассматриваются в гл. 4.

Однако существует структура, которая является усложненной, поскольку ее кардинальное число не ограничено, но которая так широко и часто используется, что ее приходится включить в число фундаментальных структур. Это — *последовательность*. Для описания абстрактного понятия последовательности мы вводим следующую терминологию и нотацию:

1. $\langle\ \rangle$ обозначает пустую последовательность.
2. $\langle x_0 \rangle$ обозначает последовательность, состоящую из единственной компоненты x_0, она называется *единичной* последовательностью.
3. Если $x = \langle x_1, \ldots, x_m \rangle$ и $y = \langle y_1, \ldots, y_n \rangle$ — последовательности, то

$$x \,\&\, y = \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle \tag{1.41}$$

есть *конкатенация* x и y.

4. Если $x = \langle x_1, \ldots, x_n \rangle$ — непустая последовательность, то

$$first\,(x) = x_1 \tag{1.42}$$

обозначает первый элемент x.

5. Если $x = \langle x_1, \ldots, x_n \rangle$ — непустая последовательность, то

$$rest\,(x) = \langle x_2, \ldots, x_n \rangle \tag{1.43}$$

есть последовательность x без первой компоненты. Следовательно, мы получаем инвариантное отношение

$$\langle first\,(x) \rangle \,\&\, rest\,(x) \equiv x \tag{1.44}$$

Введение этих обозначений не предполагает, что они будут использоваться в конкретных программах и обрабатываться реальными вычислительными машинами. В действительности весьма существенно, что операция конкатенации *не* используется в общем виде и обработка последовательностей ограничивается применением тщательно отобранного множества операций, предполагающих определенный порядок использования. Сами операторы определяются с помощью абстрактных понятий последовательности и конкатенации. Тщательный выбор множества операторов, работающих с последовательностями, позволяет при реализации находить удобное и эффективное представление последовательности на любом данном запоминающем устройстве. В результате соответствующий механизм динамического распределения памяти может быть достаточно простым, что позволяет программисту работать, не вникая в его тонкости.

Для того чтобы было ясно, что последовательность, вводимая в качестве базового типа, допускает применение только ограниченного множества операторов, основанных на строго последовательном доступе к компонентам, эта струк-

тура называется *последовательным файлом* или просто *файлом*. По аналогии с определениями типа для массивов и множеств файловый тип определяется так:

$$\boxed{\textbf{type}\ T = \textbf{file of}\ T_0} \tag{1.45}$$

Это значит, что любой файл типа T состоит из 0 или более компонент типа T_0.

Примеры:

type *text* = **file of** *char*
type *deck* = **file of** *card*

Смысл *последовательного доступа* заключается в том, что в каждый момент доступна лишь одна определенная компонента последовательности. Эта компонента определяется *текущей позицией* механизма доступа. Позиция с помощью файловых операций может меняться, определяя либо следующую компоненту (см. *get*), либо первую компоненту всей последовательности (см. *reset*). Формально мы определим позицию файла, считая, что файл состоит из двух частей: части x_L слева от текущей позиции и части x_R справа от нее. Очевидно, что всегда справедливо равенство (инвариант)

$$x \equiv x_L \,\&\, x_R \tag{1.46}$$

Второе, более важное следствие последовательного доступа заключается в том, что процессы формирования и просмотра последовательности не могут произвольно чередоваться. Таким образом, файл вначале строится при помощи последовательного добавления компонент (в конец), а затем может последовательно просматриваться от начала до конца. Поэтому принято считать, что файл находится в одном из двух состояний: либо формирования (записи), либо просмотра (чтения).

Преимущество строго последовательного доступа особенно ощутимо, если файлы размещаются на вспомогательных запоминающих устройствах, т. е. если происходит обмен между устройствами. Последовательный доступ — единственный метод, позволяющий успешно скрывать от программиста сложность механизмов такого обмена. В частности, он допускает применение буферизации — простого приема, который обеспечивает оптимальное использование ресурсов сложной вычислительной системы.

Некоторые запоминающие устройства на самом деле допускают только последовательный доступ к находящейся на них информации. Очевидно, что к таким устройствам отно-

сятся все виды лент. Но даже на магнитных барабанах и дисках каждая отдельная дорожка представляет собой запоминающее устройство с последовательным доступом. Строго последовательный доступ — основное свойство всех устройств с механическим перемещением, а также некоторых других.

1.11.1. Элементарные операции над файлами

Теперь мы попытаемся сформулировать абстрактное понятие последовательного доступа с помощью некоторого множества *элементарных операций над файлами*, которые имеются в распоряжении программиста. Они определяются в терминах понятий последовательности и конкатенации. Существует операция, инициализирующая процесс формирования файла, операция, инициализирующая просмотр, операция, добавляющая компоненту в конец последовательности, и операция, позволяющая при просмотре переходить к следующей компоненте. Две последние здесь определяются в форме, предполагающей наличие явной вспомогательной переменной, которая представляет собой буфер. Мы считаем, что такой буфер автоматически связывается с каждой файловой переменной x, и обозначаем его через $x\uparrow$. Ясно, что если x — типа T, то $x\uparrow$ принадлежит его базовому типу $T_э$.

1. Построение пустой последовательности. Операция

$$rewrite\,(x) \tag{1.47}$$

означает присваивание

$$x := \langle\ \rangle$$

Эта операция используется для уничтожения текущего значения x и инициации процесса построения новой последовательности, она соответствует разметке ленты.

2. Увеличение последовательности. Операция

$$put\,(x) \tag{1.48}$$

означает присваивание

$$x := x\ \&\ \langle x\uparrow \rangle$$

которое фактически добавляет значение $x\uparrow$ к последовательности x.

3. Инициация просмотра. Операция

$$reset\,(x) \tag{1.49}$$

означает одновременные присваивания

$$x_L := \langle\ \rangle$$
$$x_R := x$$
$$x\uparrow := first\,(x)$$

Эта операция используется для инициации процесса чтения последовательности.

4. Переход к следующей компоненте. Операция

$$get(x) \tag{1.50}$$

означает одновременные присваивания

$$x_L := x_L \,\&\, \langle first(x_R)\rangle$$
$$x_R := rest(x_R)$$
$$x\uparrow := first(rest(x_R))$$

Заметим, что $first(s)$ определено только при $s \neq \langle\,\rangle$.

Операции *rewrite* и *reset* не зависят от позиции буфера файла перед их выполнением. В любом случае они возвращают его к началу файла.

При просмотре последовательности необходимо иметь возможность распознавать ее конец, поскольку при достижении конца последовательности операция

$$x\uparrow := first(x_R)$$

становится неопределенной. Достижение конца файла, очевидно, равнозначно тому, что правая часть x_R пуста. Поэтому мы вводим предикат

$$eof(x) = x_R = \langle\,\rangle \tag{1.51}$$

который означает, что достигнут конец файла (*e*nd *o*f *f*ile). Следовательно, операция $get(x)$ может выполняться только при $eof(x) = false$.

В принципе все действия с файлами можно выразить с помощью четырех основных файловых операций. На практике же часто бывает естественно объединять операции продвижения по файлу (*get* или *put*) с обращением к буферной переменной. Поэтому мы введем еще две процедуры, которые можно выразить в терминах основных операций. Пусть v — переменная, а e — выражение базового типа T_0 файла. Тогда

$read(x, v)$ эквивалентно
$v := x\uparrow;\ get(x)$

а

$write(x, e)$ эквивалентно
$x\uparrow := e;\ put(x)$

Преимущество использования *read* и *write* вместо *get* и *put* связано не только с краткостью, но и с простотой концепции, поскольку теперь можно игнорировать существование буферной переменной $x\uparrow$, значение которой может быть и неопределенным. Однако буферная переменная бывает полезна для «заглядывания вперед».

Для выполнения этих двух процедур необходимы следующие условия:

$$\neg eof(x) \quad \text{для} \quad read(x, v)$$
$$eof(x) \quad \text{для} \quad write(x, e)$$

При чтении файла предикат $eof(x)$ становится истинным, как только прочитана последняя компонента файла x. На этом основаны две *схемы программ* для последовательного формирования и обработки файла x. Дополнительными параметрами в этих схемах являются операторы R и S и предикат p.

Запись файла x:

```
rewrite(x);
while p do
    begin R(v); write(x,v)
    end
```
(1.52)

Чтение файла x:

```
reset(x);
while ¬eof(x) do
    begin read(x,v); S(v)
    end
```
(1.53)

1.11.2. Файлы со сложной структурой

При решении многих прикладных задач необходимо в большие файлы ввести некоторую подструктуру. Например, книга, хотя и может рассматриваться как единая последовательность букв, подразделяется на главы и абзацы. Назначение подструктуры — задание неких явных точек отсчета, некоторых координат, которые позволят легче ориентироваться в длинной последовательности информации. Существующие запоминающие устройства часто дают определенные средства для представления таких точек отсчета (например, маркеры магнитной ленты) и позволяют находить их со скоростью большей, чем скорость просмотра информации между этими точками.

В рамках принятой нами системы обозначений естественный способ ввести первый уровень подструктуры — это рассматривать подобный файл как последовательность элементов, которые в свою очередь являются последовательностями, т. е. как файл файлов. Предположим, что конечные элементы

или единицы принадлежат к типу U, тогда подструктуры будут типа

$$T' = \textbf{file of } U$$

а весь файл — типа

$$T = \textbf{file of } T'$$

Очевидно, что таким образом можно строить файлы с любым уровнем вложенности. В общем виде тип T_n можно определить с помощью рекурсивного соотношения

$$T_i = \textbf{file of } T_{i-1}, \quad i = 1..n$$

и $T_0 = U$. Такие файлы часто называют *многоуровневыми файлами*, а компоненту типа T_i называют *сегментом* *) i-го уровня. Примером многоуровневого файла является книга, в которой уровням сегментации соответствуют главы, разделы, абзацы и строки. Но наиболее общий случай — это файл с одним уровнем сегментации.

Такой файл, сегментированный на одном уровне, никоим образом не идентичен массиву файлов. Прежде всего число сегментов файла может меняться, и файл по-прежнему может увеличиваться только с конца. В рамках введенных выше обозначений и при определении файла как

$$x\textbf{: file of file of } U$$

$x\uparrow$ будет обозначать доступный в текущий момент сегмент, а $x\uparrow\uparrow$ — доступную в текущий момент единичную компоненту. Соответственно $put(x\uparrow)$ и $get(x\uparrow)$ ссылаются на единичную компоненту, а $put(x)$ и $get(x)$ означают операции добавления очередного сегмента и перехода на очередной сегмент.

Сегментированные файлы удовлетворительно реализуются практически на всех запоминающих устройствах с последовательным доступом, включая ленты. Сегментация не меняет их важнейшего свойства — последовательного доступа либо к отдельным компонентам, либо (возможно, при помощи более быстрого механизма пропуска) к сегментам. Другие запоминающие устройства, а именно магнитные *барабаны* и *диски*, обычно содержат некоторое количество дорожек, каждая из которых представляет собой запоминающее устройство с последовательным доступом, но которое обычно слишком мало, чтобы вместить целый файл. Таким образом, на дисках файлы обычно распределяются на несколько дорожек и содержат соответствующую библиотечную информацию, связывающую эти дорожки. Очевидно, что

*) Слово «сегмент» встречается, пожалуй, только в этой книге. — *Прим. ред.*

указатель начала каждой дорожки служит естественным маркером сегмента, и, возможно, обращение к нему легче и непосредственнее, чем к маркерам на каком-либо последовательном устройстве. Для адресации дорожек, с которых начинаются сегменты, и обозначения текущей длины сегментов может использоваться, например, индексированная таблица в основной памяти (рис. 1.10).

Это приводит нас к так называемым *индексированным файлам* (иногда также называемым *файлами с прямым доступом*). В настоящее время барабаны и диски организованы

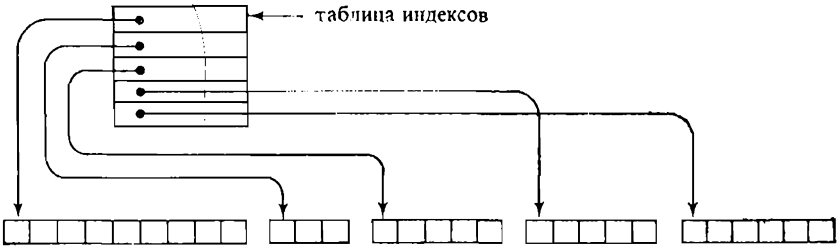


Рис. 1.10. Индексированный файл с пятью сегментами.

таким образом, что каждая дорожка содержит много физических меток, с которых может начинаться чтение или запись. Поэтому нет необходимости, чтобы каждый сегмент занимал ее целиком, поскольку это может привести к неэкономному расходованию памяти, если сегменты коротки по сравнению с длиной дорожки. Область памяти между двумя метками называется *физическим сегментом* или *сектором* в отличие от *логического сегмента*, который является понятием, относящимся к структуре данных программы. Разумеется, каждый физический сегмент содержит самое большее один логический сегмент, а каждый логический сегмент (даже пустой) занимает по меньшей мере один физический сегмент. Следует иметь в виду, что, хотя речь и идет о файлах «с прямым доступом», среднее время нахождения сегмента, так называемое *латентное* время, равно половине времени полного оборота диска.

У индексированных файлов сохраняется также то основное свойство, что запись производится последовательно в их конец. Поэтому они особенно полезны в случаях, когда изменения происходят сравнительно редко. Изменения производятся либо при помощи увеличения файла, либо при помощи копирования и обновления всего файла. Просмотр может осуществляться намного быстрее, если используются индексные указатели. Это типичная ситуация для так называемых *банков данных*.

Системы, которые допускают выборочное изменение фрагментов в середине файла, обычно сложны и пользоваться ими рискованно, поскольку новые порции информации должны быть того же размера, что и старые, на место которых они записываются. Кроме того, при обработке большого объема данных не рекомендуется выборочное изменение, поскольку при любой неудаче — чем бы она ни была вызвана: ошибкой программы или сбоем оборудования — по правилам должно существовать некоторое состояние, к которому следует вернуться, с тем чтобы возобновить и повторить прерванную работу. Поэтому обновление обычно происходит целиком: старый файл заменяется новой, измененной копией только после того, как последовательная проверка установит правильность нового файла. Для обновления последовательная организация является наиболее надежной. Ее следует предпочесть более сложным способам организации данных большого объема. Эти способы могут быть эффективнее, но они часто приводят к полной потере данных при сбое оборудования.

1.11.3. Тексты

Файлы, состоящие из компонент типа *char* (символьного), играют особо важную роль в вычислениях и обработке данных: они обеспечивают взаимодействие между вычислительными системами и пользователями. *Читаемый вход*, организуемый программистом, так же как *читаемый выход*, содержащий результаты вычислений, представляют собой последовательности символов. Поэтому таким типам данных присваивается стандартное имя:

type *text* = **file of** *char*

В конечном счете взаимодействие между вычислительным процессом и человеком можно представить двумя текстовыми файлами. Один из них содержит входную информацию (*input*) для вычислительного процесса, другой — результаты вычисления, называемые выходной информацией (*output*). С этого момента мы будем считать, что каждая программа содержит описания этих двух файлов, имеющие следующий вид:

var *input*, *output*: *text*

Учитывая, что эти файлы соответствуют стандартным средствам ввода и вывода вычислительной системы (таким, как устройство чтения с перфокарт и устройство вывода на печать), мы будем считать, что файл *input* можно только читать, а в файл *output* можно только писать.

Поскольку эти два стандартных файла используются очень часто, мы определим, что если первый параметр

процедур *read* и *write не является* файловой переменной, то по умолчанию предполагаются соответственно файлы *input* и *output*. Кроме того, мы позволим этим двум стандартным процедурам иметь произвольное число аргументов. Суммируем введенные выше обозначения:

read(*x*1, ..., *xn*) означает *read*(*input*, *x*1, ..., *xn*)
write(*x*1, ..., *xn*) означает *write*(*output*, *x*1, ..., *xn*)
read(*f*, *x*1, ..., *xn*) означает
begin *read*(*f*, *x*1); ...; *read*(*f*, *xn*) **end**
write(*f*, *x*1, ..., *xn*) означает
begin *write*(*x*, *x*1); ...; *write*(*f*, *xn*) **end**

Тексты являются типичным примером последовательностей, в которых обнаруживается подструктура. Принятые единицы этой подструктуры — это главы, абзацы и строки. Обычный способ изображения подструктуры текста — использование специальных разделительных символов. Наиболее известный пример — символ пробела, но подобные символы могут использоваться и для указания концов строк, абзацев и глав. Например, широко распространенное множество символов ISO, включая его американскую версию ASCII, содержит несколько таких элементов, называемых управляющими символами (см. приложение А).

В этой книге мы не будем использовать специальные разделительные символы и задавать какой-либо способ представления подструктуры. Вместо этого мы рассмотрим текст как файл, состоящий из символьных последовательностей, представляющих отдельные строки. Кроме того, мы ограничимся одним уровнем подструктуры, а именно строкой. Однако вместо того, чтобы определить тексты как файлы файлов печатаемых символов, мы рассматриваем их как файлы символов и вводим дополнительные операции и предикаты для управления, т. е. для отметки и распознавания строк. Их смысл легче понять, если предположить, что строки разделяются (гипотетическими) разделительными символами (не принадлежащими к типу *char*), а задача этих операций и предикатов — поиск и распознавание таких символов-разделителей. Дополнительные операции следующие:

writeln(*f*) — добавить признак конца строки к файлу *f*.
readln(*f*) — пропустить символы файла *f* до символа, который непосредственно следует за очередным символом конца строки.
eoln(*f*) — булевская функция. Истинна, если позиция файла указывает на признак конца строки, иначе ложна. Предполагается, что если *eoln*(*f*) истинна, то $f\uparrow$ = пробел.

Теперь мы можем привести две схемы программ для записи и чтения текстов, подобно схемам для «записи» и «чтения» других файлов [см. (1.52) и (1.53)]. В этих схемах участвует текстовый файл f и уделяется должное внимание формированию и распознаванию строчной структуры. Пусть $R(x)$ — оператор, присваивающий переменной x (типа *char*) некоторое значение. В нем также определяются условия p и q, означающие «это был последний символ строки» и «это был последний символ файла». При чтении файла в начале каждой строки выполняется оператор U, а для каждого символа x выполняется оператор $S(x)$. В конце каждой строки выполняется оператор V.

Запись текста f:

```
rewrite(f);
while ¬q do
    begin
        while ¬p do
            begin R(x); write(f,x)
            end ;
        writeln(f)
    end
```
(1.54)

Чтение текста f:

```
reset(f);
while ¬eof(f) do
    begin U;
        while ¬eoln(f) do
            begin read(f,x); S(x)
            end ;
        V; readln(f)
    end
```
(1.55)

Бывают случаи, когда построчная структура текста не содержит никакой существенной информации. Наше соглашение о значении буферной переменной при появлении признака конца строки [см. определение $eoln(f)$] позволяет в таких случаях использовать простую схему. Заметим, что в соответствии с определением *eoln* каждый конец строки дает добавочный символ пробела.

```
while ¬eof(f) do
    begin read(f,x); S(x)
    end
```
(1.56)

В большинстве языков программирования принято допускать для процедур чтения и записи аргументы типа *integer* или *real*. Такое обобщение было бы строгим, если бы типы *integer* и *real* представлялись как массивы символов, компоненты которых обозначают отдельные цифры числа. Языки, ориентированные только на коммерческие приложения, действительно удовлетворяют такому определению: они требуют представления чисел в десятичных цифрах и в десятичной системе счисления. Но введение типов *integer* и *real* в качестве фундаментальных обладает важным преимуществом: можно опустить подобные детальные спецификации. При этом в системе можно использовать такие представления чисел, которые больше ей соответствуют. В действительности в системах, ориентированных на научные расчеты, всегда выбирают двоичное представление, так как оно почти во всех отношениях имеет преимущества перед десятичным.

Но из этого следует, что программист должен понимать, что невозможно читать числа или записывать их в файлы без соответствующих операций преобразования. Обычно эти операции неявно содержатся в операциях *read* и *write* с аргументами числовых типов. Однако профессиональный программист сознает, что такие операции (так называемые *I/O-операции*) состоят из двух различных действий: обмена данными между различными запоминающими устройствами *и* преобразования представлений данных. Последнее действие может быть довольно сложным и занимать много времени.

В последующих главах этой книги операции чтения и записи с числовыми аргументами будут использоваться в соответствии с правилами языка программирования Паскаль. Эти правила допускают некоторые спецификации формата для управления процессом преобразования. Спецификации формата указывают число желательных цифр при операции записи. Это число символов, называемое также «шириной поля», записывается сразу после аргумента следующим образом:

$$write(f,\ x:\ n)$$

Аргумент x должен записываться в файл f; его значение преобразуется в последовательность из (по крайней мере) n символов. Если необходимо, цифрам предшествует знак и соответствующее число пробелов.

Для того чтобы понять примеры программ, приводимые далее в этой книге, дальнейшие подробности не нужны. Однако мы включили сюда две подпрограммы (программы 1.3 и 1.4), преобразующие представления чисел, чтобы показать, насколько *сложны* подобные действия, неявно предполагающиеся в операторах записи. Эти процедуры преобразуют ве-

```
procedure readreal (var f: text; var x: real);
  {чтение вещественного числа x из файла f}
  {далее идут константы, связанные с отдельной
  вычислительной системой}
  const t48 = 281474976710656;      {= 2**48}
    limit = 5629499534213l;         {= t48 div 5}
    z = 27;      {= ord('0') }
    lim 1 = 322;       {максимальный порядок}
    lim 2 = -292;    {минимальный порядок}
  type posint = 0 .. 323;
  var ch: char; y: real; a,i,e: integer;
    s,ss: boolean;      {знаки}
  function ten(e: posint): real; { = 10**e, 0<e<322 };
    var i: integer; t: real;
  begin i := 0; t := 1.0;
    repeat if odd(e) then
      case i of
       0: t := t * 1.0E1;
       1: t := t * 1.0E2;
       2: t := t * 1.0E4;
       3: t := t * 1.0E8;
       4: t := t * 1.0E16;
       5: t := t * 1.0E32;
       6: t := t * 1.0E64;
       7: t := t * 1.0E128;
       8: t := t * 1.0E256
      end ;
      e := e div 2; i := i+1
    until e = 0;
    ten := t
  end ;
  begin
    {пропуск начальных пробелов}
    while f↑=' ' do get(f);
    ch := f↑ ;
    if ch = '-' then
      begin s := true; get(f); ch := f↑
      end else
      begin s := false;
        if ch = '+' then
        begin get(f); ch := f↑
        end
      end ;
    if ¬(ch in ['0' .. '9']) then
```

```
    begin message (' DIGIT EXPECTED'); halt;
    end ;
    a := 0; e := 0;
    repeat if a < limit then a := 10*a + ord(ch)-z else e := e+1;
       get(f); ch := f↑
    until ¬(ch in ['0'..'9']);
    if ch = '.' then
    begin {чтение дробной части} get(f); ch := f↑;
       while ch in ['0'..'9'] do
       begin if a < limit then
             begin a := 10*a + ord(ch)-z; e := e-1
             end ;
          get(f); ch := f↑
       end
    end ;
    if ch = 'E' then
    begin {чтение порядка}   get(f); ch := f↑;
       i := 0;
       if ch = '-' then
       begin ss := true; get(f); ch := f↑
       end else
       begin ss := false; if ch = '+' then
          begin get(f); ch := f↑
          end
       end ;
       while ch in ['0'..'9'] do
       begin if i<limit then begin i := 10*i + ord(ch)-z end;
          get(f); ch := f↑
       end ;
       if ss then e := e-i else e := e+i
    end ;
       if e < lim.2 then
          begin a := 0; e := 0
          end else
       if e > lim 1 then
       begin message(' NUMBER TOO LARGE '); halt end;
       { 0 < a < 2**49 }
       if a ≥ t48 then y := ((a+1) div 2) * 2.0 else y := a;
       if s then y := -y;
       if e < 0 then x := y/ten(-e) else
       if e ≠ 0 then x := y*ten(e) else x := y ;
       while (f↑ = ' ')∧(¬eof(f)) do get(f);
end {readreal}
```

Программа 1.3. Чтение вещественного числа.

```
procedure writereal (var f: text; x: real; n: integer);
  {печать вещественного числа x с n символами в десятичном виде
  с плавающей запятой}
  {следующие константы зависят от используемого представления
  вещественных чисел с плавающей запятой}
  const t48 = 281474976710656; { = 2 ** 48; 48-размер мантиссы}
        z = 27;     { ord('0') }
  type posint = 0..323;     {диапазон для десятичного порядка}
  var c,d,e,e0,e1,e2,i: integer;

  function ten(e: posint): real;     { 10**e, 0<e<322 }
    var i: integer; t: real;
  begin i := 0; t := 1.0;
    repeat if odd(e) then
      case i of
       0: t := t * 1.0E1;
       1: t := t * 1.0E2;
       2: t := t * 1.0E4;
       3: t := t * 1.0E8;
       4: t := t * 1.0E16;
       5: t := t * 1.0E32;
       6: t := t * 1.0E64;
       7: t := t * 1.0E128;
       8: t := t * 1.0E256
      end ;
      e := e div 2; i := i+1
    until e = 0;
    ten := t
  end { ten } ;
  begin {требуются по крайней мере 10 символов:b+9.9E+999 }
    if x = 0 then
    begin repeat write(f, ' '); n := n-1
          until n <= 1;
      write(f, '0')
    end else
    begin
      if n <= 10 then n := 3 else n := n-7;
      repeat write(f, ' '); n := n-1
      until n <= 15;
      { 1 < n <= 15,  число печатаемых цифр}
      begin {проверка знака, определение порядка}
        if x < 0 then
          begin write(f, '-'); x := -x
```

```
    end else write (f, ' ');
e := expo (x);  {e = entier(log2(abs(x)))}
if e ≥ 0 then
    begin e := e*77 div 256 +1; x := x/ten(e);
       if x ≥ 1.0 then
          begin x := x/10.0; e := e+1
          end
    end else
    begin e := (e+1)*77 div 256; x := ten(-e)*x;
       if x < 0.1 then
          begin x := 10.0*x; e := e-1
          end
    end ;
{ 0.1 ≤ x < 1.0 }
case n of     {округление}
  2: x := x+0.5E-2;
  3: x := x+0.5E-3;
  4: x := x+0.5E-4;
  5: x := x+0.5E-5;
  6: x := x+0.5E-6;
  7: x := x+0.5E-7;
  8: x := x+0.5E-8;
  9: x := x+0.5E-9;
 10: x := x+0.5E-10;
 11: x := x+0.5E-11;
 12: x := x+0.5E-12;
 13: x := x+0.5E-13;
 14: x := x+0.5E-14;
 15: x := x+0.5E-15
end ;
if x ≥ 1.0 then
    begin x := x * 0.1; e := e+1;
    end ;
c := trunc(x,48); {= trunc(x*(2**48))}
c := 10*c; d := c div t48;
write(f, chr(d+z), '.');
for i := 2 to n do
begin c := (c - d*t48) * 10; d := c div t48;
    write(f, chr(d+z))
end ;
write(f, 'E'); e := e-1;
if e < 0 then
```

```
            begin write(f, '—'); e := —e
            end else write(f, '+');
        e1 := e * 205 div 2048; e2 := e — 10*e1;
        e0 := e1 * 205 div 2048; e1 := e1 — 10*e0;
        write(f, chr(e0+z), chr(e1+z), chr(e2+z))

    end
end {writereal}
```

Программа 1.4. Печать вещественного числа.

щественные числа из десятичного в произвольное «внутреннее» представление и наоборот. (Константы в заголовках связаны с особенностями формата чисел с плавающей запятой в вычислительной машине CDC 6000: 11-разрядный двоичный порядок и 48-разрядная мантисса. Функция *expo*(*x*) означает порядок *x*).

1.11.4. Программа редактирования файлов

В качестве примера применения последовательной структуры мы приведем следующую задачу, которая одновременно продемонстрирует методику разработки и пояснения программ. Этот метод называется *методом поэтапного уточнения* [1.4, 1.6], мы будем использовать его в этой книге при разборе многих алгоритмов.

Задача состоит в том, чтобы разработать программу, которая редактирует текст *x*, превращая его в текст *y*. Редактирование означает исключение или замену определенных строк или включение новых строк. Редактированием управляет последовательность *команд редактирования*, представленных стандартным текстом *input*. Эти команды имеют такой вид:

I, *m*.	Вставка в текст после *m*-й строки.
D, *m*, *n*.	Исключение строк от *m* до *n*.
R, *m*, *n*.	Замена строк от *m* до *n*.
E.	Окончание редактирования.

Каждая команда занимает одну строку в стандартном файле *input*, который мы называем файлом команд, *m* и *n* — десятичные номера строк, а вставляемые тексты должны непосредственно следовать за командами I и R. Они заканчиваются пустой строкой.

Мы требуем, чтобы номера строк в командах редактирования шли в строго возрастающем порядке. Это правило обеспечивает строго *последовательную обработку* входного текста *x*. Очевидно, что состояние работы определяется теку-

щей позицией x, т. е. номером строки, которая рассматривается в данный момент.

Предположим, что программа редактирования работает в интерактивном режиме и что, следовательно, файл команд представляет собой, например, данные, вводимые с терминала. В таком режиме работы весьма желательно, чтобы пользователь имел некоторую обратную связь. Подходящая и полезная форма обратной связи — это распечатка той строки, на которую продвинулся процесс редактирования после выполнения последней команды. Мы назовем эту строку *текущей строкой*. Вследствие нового требования, чтобы после выполнения каждой команды текущая строка выводилась на печать, нужно иметь явную переменную, в которой эта строка будет храниться после чтения из x и перед записью в y. Этот прием называется «заглядыванием вперед». Теперь можно представить программу редактирования следующим образом:

```
program editor (x, y, input, output);
var lno: integer;      {номер текущей строки}
    cl : line;         {текущая строка}
    x,y: text;
begin read instruction;                                  (1.57)
   repeat interpret instruction;
          write line;
          read instruction
   until instruction = 'E'
end.
```

Попробуем теперь более подробно определить некоторые операторы. Уточняя *«читать команду»* и *«выполнить команду»*, мы обращаем внимание, что команда обычно состоит из трех частей: кода команды и двух параметров. Поэтому мы вводим три переменные: *code*, m и n — для обмена между этими двумя операторами.

```
var code,ch: char;
    m,n: integer
```

Читать команду:

```
read(code,ch);
if ch = ',' then read(m,ch) else m := lno;          (1.58)
if ch = ',' then read(n) else n := m;
```

Эта формулировка допускает команды с 0, 1 или 2 параметрами, так как для «пропущенных» спецификаций подставляются значения по умолчанию.

Выполнить команду:

```
copy;
if code = 'I' then
begin putline;
      insert;
end else
if code = 'D' then skip else                    (1.59)
if code = 'R' then
begin insert;
      skip
end else
if code = 'E' then copyrest else Error
```

На следующем этапе уточнения мы выразим операторы *copy* (копировать), *insert* (вставить) и *skip* (пропустить), использованные в (1.59), с помощью операций с отдельными строками *getline* и *putline*. Их общим свойством является цикличность структуры. *Copy* служит для переписи строк из x в y, начиная с текущей и кончая m-й строкой. *Skip* читает строки из x до n-й строки, не переписывая их в y.

```
Copy:     while lno < m do
          begin putline;
                getline
          end

Skip:     while lno < n do getline

Insert:   readline;
          while noend do
              begin putline; readline
              end;                              (1.60)
          getline;

Copyrest: while ¬eof(x) do
              begin putline; getline
              end;
          putline
```

На третьем, последнем этапе уточнения мы выражаем операции *getline*, *putline*, *readline* и *writeline* с помощью операций с отдельными символами. Мы видим, что до сих пор все действия касались исключительно целых строк и не делалось никаких специальных предположений о детальной структуре строки. Мы знаем, что строки являются последовательностями символов. Было бы заманчиво описать переменную

cl (содержащую текущую строку) как последовательность

var *cl*: **file of** *char*

Однако вспомним совет никогда не использовать структуру с бесконечным кардинальным числом, если имеется эквивалентная фундаментальная структура (такая, как массив). Действительно, в этом случае рекомендуется использовать массив. Это возможно, если мы ограничим длину строки, например, 80 символами. Итак, мы определим

var *cl*: **array**[1..80] **of** *char*

Следующие четыре подпрограммы используют с этим массивом индексную переменную *i*. Фактически же эта переменная используется локально и может в каждой процедуре описываться как локальная. Кроме того, теперь нужно еще ввести глобальную переменную *L* для обозначения длины текущей строки.

```
Getline:  i := 0; lno := lno + 1;
          while ¬eoln(x) do
             begin i := i+1; read (x, cl[i])
             end ;
          L := i; readln(x)

Putline:  i := 0;
          while i < L do
             begin i := i+1; write (y, cl[i])
             end ;
          writeln(y)                                   (1.61)

Readline: i := 0;
          while ¬eoln(input) do
             begin i := i+1; read(cl[i])
             end ;
          L := i; readln

Writeline: i := 0; write (lno);
          while i < L do
             begin i := i+1; write(cl[i])
             end ;
          writeln
```

Условие *noend* в программе *insert* теперь легко можно выразить как

$$L \neq 0$$

На этом разработка программы редактирования файлов завершается.

УПРАЖНЕНИЯ

1.1. Пусть кардинальные числа стандартных типов *integer*, *real* и *char* обозначены через c_I, c_R и c_C. Каковы кардинальные числа следующих типов данных, определенных в этой главе в качестве примеров: *пол*, *Boolean*, *день*, *буква*, *цифра*, *офицер*, *row*, *alfa*, *date*, *complex*, *person*, *coordinate*, *charset*, *tapestatus*?

1.2. Как бы вы представили переменные типов, перечисленных в упр. 1.1:
(a) в памяти вычислительной машины, которой вы пользуетесь?
(b) на Фортране?
(c) на предпочитаемом вами языке программирования?

1.3. Какова последовательность команд (на вашей ЭВМ) для:
(a) операций размещения в памяти компонент упакованных записей и массивов и обращения к ним?
(b) операций над множествами, включая проверку принадлежности?

1.4. Можно ли во время выполнения программы контролировать правильность использования записей с вариантами? А во время трансляции?

1.5. По каким причинам некоторые совокупности данных определяют как последовательные файлы, а не как массивы?

1.6. Предположим, что вам нужно представить последовательные файлы, определенные в разд. 1.11, на ЭВМ с очень большой оперативной памятью. Вам разрешается ввести ограничение, что длина файла никогда не превышает определенную величину L. Следовательно, вы можете представить файлы с помощью массивов.

Опишите возможную реализацию, включая выбранное представление данных и процедуры для элементарных файловых операций *get*, *put*, *reset* и *rewrite*, которые определены с помощью аксиом в разд. 1.11.

1.7. Выполните упр. 1.6 для сегментированных файлов.

1.8. Имеется железнодорожное расписание, содержащее список ежедневных рейсов поездов на нескольких линиях железной дороги. Найдите такое представление этих данных с помощью массивов, записей или файлов, которое было бы удобно для поиска времени прибытия и отправления поезда в нужном направлении для определенной станции.

1.9. Дан текст T в виде файла и небольшой список слов в виде двух массивов A и B. Предположим, что слова — это небольшие массивы символов, максимальная длина которых фиксирована.

Напишите программу, которая преобразует текст T в текст S, заменяя каждый раз слово A_i соответствующим словом B_i.

1.10. Какие изменения (переопределение констант и т. п.) необходимы, чтобы переделать программы 1.3 и 1.4 для имеющейся у вас вычислительной машины?

1.11. Напишите программу, подобную программе 1.4, с заголовком

procedure *writereal*(**var** *f*: *text*; *x*: *real*; *n*, *m*: *integer*)

Требуется преобразовать значение x в последовательность, состоящую по крайней мере из n символов (их надо добавить в файл f), представляющих x в десятичном виде с фиксированной запятой с m цифрами после запятой. Если необходимо, числу может предшествовать соответствующее количество пробелов и/или знак.

1.12. Перепишите текстовый редактор из разд. 1.11.4 в виде завершенной программы.

1.13. Сравните три следующие версии бинарного поиска с (1.17). Какие из этих трех программ правильны? Какие более эффективны? Мы предполагаем, что имеются следующие переменные и константа $N > 0$:

```
var i,j,k: integer;
    a: array[1 .. N] of T;
    x: T
```

Программа A:

```
i := 1; j := N;
repeat k := (i+j) div 2;
   if a[k] < x then i := k else j := k
until (a[k]=x) ∨ (i≥j)
```

Программа B:

```
i := 1; j := N;
repeat k := (i+j) div 2;
   if x ≤ a[k] then j := k−1;
   if a[k] ≤ x then i := k+1
until i > j
```

Программа C:

```
i := 1; j := N;
repeat k := (i+j) div 2;
   if x < a[k] then j := k else i := k+1
until i ≥ j
```

Указание. Все программы должны заканчиваться при $a[k] = x$, если такая компонента существует, или при $a[k] \neq x$, если нет компоненты со значением x.

1.14. Некоторая компания проводит опрос, чтобы выяснить спрос на свою продукцию. Ее продукция — это пластинки и магнитофонные ленты с песнями; самые популярные песни будут переданы по радио. Опрашиваемое население делится на четыре категории согласно полу и возрасту (скажем, моложе 20 и старше 20). Каждый опрашиваемый должен назвать пять любимых песен. Песням ставятся в соответствие числа от 1 до N (например, пусть $N = 30$). Результаты опроса представлены в файле *poll* такого типа

```
type hit = 1 .. N;
     sex = (male, female);
     response =
        record name, firstname: alfa;
               s: sex;
               age: integer;
               choice: array [1 .. 5] of hit
        end ;
var poll: file of response
```

Итак, каждый элемент файла представляет опрашиваемого и содержит его имя, фамилию, пол, возраст и пять любимых песен в порядке

предпочтения. Этот файл является входным для программы, которая должна получить следующие результаты:

1. Список песен в порядке их популярности. Каждый элемент этого списка содержит номер песни и число упоминаний при опросе. Песни, которые ни разу не упоминались, исключаются из списка.
2. Четыре отдельных списка с именами и фамилиями всех отвечающих, которые назвали на первом месте одну из трех песен, наиболее популярных в их категории.

Всем пяти спискам должны предшествовать соответствующие заголовки.

ЛИТЕРАТУРА

1.1. DAHL O. J., DIJKSTRA E. W., HOARE C. A. R., Structured Programming. — New York: Academic Press, 1972. [Имеется перевод: Дал О., Дейкстра Э., Хоор К., Структурное программирование. — М.: Мир, 1975.]
1.2. HOARE C. A. R. Notes on Data Structuring. — Structured Programming. Dahl, Dijkstra, Hoare, 83—174. [Имеется перевод: Хоор К. Заметки о структурной организации данных. — В кн.: Дал О., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975, с. 98—197.]
1.3. JENSEN K., WIRTH N. PASCAL, User Manual and Report. — Lecture Notes in Computer Science. — Berlin: Springer-Verlag, 18, 1974. [Имеется перевод: Йенсен К., Вирт Н. ПАСКАЛЬ; Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982.]
1.4. WIRTH N. Program Development by Stepwise Refinement. — *Comm. ACM*, **14**, No 4, 1971, 221—227.
1.5. WIRTH N. The Programming Language PASCAL. — *Acta Informatica*. No. 1, 1971, 35—63.
1.6. WIRTH N. On the Composition of Well-Structured Programs. — *Computing Surveys*, **6**, No. 4, 1974, 247—259.

2

СОРТИРОВКА

2.1. ВВЕДЕНИЕ

Основная цель этой главы — показать на множестве примеров, как используются структуры данных, описанные в предыдущей главе, и продемонстрировать влияние выбранной структуры данных на алгоритмы, выполняющие некоторое задание. Кроме того, сортировка служит хорошим примером того, что одна и та же цель может достигаться с помощью различных алгоритмов, причем каждый из них имеет свои определенные преимущества и недостатки, которые нужно оценить с точки зрения конкретной ситуации.

Под *сортировкой* обычно понимают процесс перестановки объектов данного множества в определенном *порядке*. Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве. В этом смысле элементы сортировки присутствуют почти во всех задачах. Упорядоченные объекты содержатся в телефонных книгах, в ведомостях подоходных налогов, в оглавлениях, в библиотеках, в словарях, на складах, да и почти всюду, где их нужно разыскивать. Даже маленьких детей приучают приводить вещи «в порядок», и они сталкиваются с некоторым видом сортировки задолго до того, как узнают что-либо об арифметике.

Следовательно, методы сортировки очень важны, особенно при обработке данных. Казалось бы, что́ легче рассортировать, чем набор данных? Однако с сортировкой связаны многие фундаментальные приемы построения алгоритмов, которые и будут нас интересовать в первую очередь. Почти все такие приемы встречаются в связи с алгоритмами сортировки. В частности, сортировка является идеальным примером огромного разнообразия алгоритмов, выполняющих одну и ту же задачу, многие из которых в некотором смысле являются оптимальными, а большинство имеет какие-либо преимущества по сравнению с остальными. Поэтому на примере сортировки мы убеждаемся в необходимости сравнительного анализа алгоритмов. Кроме того, здесь мы увидим, как при помощи усложнения алгоритмов можно добиться значительного увеличения эффективности по сравнению с более простыми и очевидными методами.

Зависимость выбора алгоритмов от структуры данных — явление довольно частое, и в случае сортировки она настолько

сильна, что методы сортировки обычно разделяют на две категории: *сортировка массивов* и *сортировка* (последовательных) *файлов*. Эти два класса часто называют *внутренней* и *внешней* сортировкой, так как массивы располагаются во «внутренней» (оперативной) памяти ЭВМ; для этой памяти характерен быстрый произвольный доступ, а файлы хранятся в более медленной, но более вместительной «внешней» памяти, т. е. на запоминающих устройствах с механическим

Рис. 2.1. Сортировка массива.

передвижением (дисках и лентах). Это существенное различие можно наглядно показать на примере сортировки пронумерованных карточек. Представление карточек в виде массива соответствует тому, что все они располагаются перед сортирующим так, что каждая карточка видна и доступна (см. рис. 2.1). Представление карточек в виде файла предполагает, что видна только верхняя карточка из каждой стопки (см. рис. 2.2). Очевидно, что такое ограничение приведет к существенному изменению методов сортировки, но оно неизбежно, если карточек так много, что их число на столе не уменьшается.

Прежде всего мы введем некоторую терминологию и систему обозначений, которые будем использовать в этой главе. Нам даны элементы

$$a_1, a_2, \ldots, a_n.$$

Сортировка означает перестановку этих элементов в таком порядке:

$$a_{k_1}, a_{k_2}, \ldots, a_{k_n},$$

что при заданной *функции упорядочения* f справедливо отношение

$$f(a_{k_1}) \leqslant f(a_{k_2}) \leqslant \ldots \leqslant f(a_{k_n}). \qquad (2.1)$$

Обычно функция упорядочения не вычисляется по какому-то специальному правилу, а содержится в каждом элементе в виде явной компоненты (поля). Ее значение называется *ключом* элемента. Следовательно, для представления элемента a_i особенно хорошо подходит структура записи. Поэтому мы определяем тип *item* (элемент), который будет

Рис. 2.2. Сортировка файла.

использоваться в последующих алгоритмах сортировки следующим образом:

```
type item = record key: integer;
                   {описание других компонент}
            end
```
(2.2)

«Прочие компоненты» — это все существенные данные об элементе; поле *key* — *ключ* служит лишь для идентификации элементов. Однако, когда мы говорим об алгоритмах сортировки, ключ для нас — единственная существенная компонента, и нет необходимости как-то определять остальные. Выбор в качестве типа ключа целого типа достаточно произволен; ясно, что точно так же можно использовать и любой тип, на котором задано отношение всеобщего порядка.

Метод сортировки называется *устойчивым*, если относительный порядок элементов с одинаковыми ключами не меняется при сортировке. Устойчивость сортировки часто бывает желательна, если элементы упорядочены (рассортированы) по каким-то вторичным ключам, т. е. по свойствам, не отраженным в первичном ключе.

Не следует считать, что данная глава представляет собой исчерпывающий обзор методов сортировки. Здесь лишь особенно подробно разбираются некоторые избранные методы. Заинтересованного читателя, желающего получить полное представление о сортировке, мы отсылаем к блестящей и всеобъемлющей работе Д. Кнута [2.7] (см. также [2.10]).

2.2. СОРТИРОВКА МАССИВОВ

Основное требование к методам сортировки массивов — экономное использование памяти. Это означает, что переупорядочение элементов нужно выполнять *in situ* (на том же месте) и что методы, которые пересылают элементы из массива a в массив b, не представляют для нас интереса. Таким образом, выбирая метод сортировки, руководствуясь критерием экономии памяти, классификацию алгоритмов мы проводим в соответствии с их эффективностью, т. е. экономией времени или быстродействием. Удобная *мера эффективности* получается при подсчете числа C — необходимых *сравнений ключей* и M — *пересылок элементов*. Эти числа определяются некоторыми функциями от числа n сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка $n \cdot \log n$ сравнений, мы сначала обсудим несколько несложных и очевидных способов сортировки, называемых *простыми методами*, которые требуют порядка n^2 сравнений ключей. Мы решили рассмотреть простые методы прежде, чем перейти к более быстрым алгоритмам, по следующим трем важным причинам:

1. Простые методы особенно хорошо подходят для разъяснения свойств большинства принципов сортировки.
2. Программы, основанные на этих методах, легки для понимания и коротки. Следует помнить, что программы также занимают память!
3. Хотя сложные методы требуют меньшего числа операций, эти операции более сложны; поэтому при достаточно малых n простые методы работают быстрее, но их не следует использовать при больших n.

Методы, сортирующие элементы *In situ*, можно разбить на три основных класса в зависимости от лежащего в их основе приема:

1. Сортировка включениями.
2. Сортировка выбором.
3. Сортировка обменом.

Теперь мы рассмотрим и сравним эти три принципа. Программы работают с переменной-массивом a, компоненты

которой нужно рассортировать *in situ*. В этих программах используются типы данных *item* (2.2) и *index*, определенные так:

```
type index = 0..n;
var a: array[1..n] of item
```
(2.3)

2.2.1. Сортировка простыми включениями.

Этот метод обычно используют игроки в карты. Элементы (карты) условно разделяются на готовую последовательность $a_1, \ldots, a_{i-1}$ и входную последовательность $a_i, \ldots, a_n$. На каждом шаге, начиная с $i = 2$ и увеличивая i на единицу, берут i-й элемент входной последовательности и передают в готовую последовательность, *вставляя* его на подходящее место.

Таблица 2.1. Пример сортировки простыми включениями

Начальные ключи	44	55	12	42	94	18	06	67
$i = 2$	44	55	12	42	94	18	06	67
$i = 3$	12	44	55	42	94	18	06	67
$i = 4$	12	42	44	55	94	18	06	67
$i = 5$	12	42	44	55	94	18	06	67
$i = 6$	12	18	42	44	55	94	06	67
$i = 7$	06	12	18	42	44	55	94	67
$i = 8$	06	12	18	42	44	55	67	94

Процесс сортировки включениями показан на примере восьми случайно взятых чисел (см. табл. 2.1). Алгоритм сортировки простыми включениями выглядит следующим образом:

```
for i := 2, to n do
  begin x := a[i];
    «вставить x на подходящее место в a_1 .. a_i»
  end
```

При поиске подходящего места удобно чередовать сравнения и пересылки, т. е. как бы «просеивать» x, сравнивая его с очередным элементом a_j и либо вставляя x, либо пересылая a_j направо и продвигаясь налево. Заметим, что «просеивание» может закончиться при двух различных условиях:

1. Найден элемент a_j с ключом меньшим, чем ключ x.
2. Достигнут левый конец готовой последовательности.

Этот типичный пример цикла с двумя условиями окончания дает нам возможность рассмотреть хорошо известный прием

фиктивного элемента («барьера»). Его можно легко применить в этом случае, установив барьер $a_0 = x$. (Заметим, что для этого нужно расширить диапазон индексов в описании a до $0, \ldots, n$.) Окончательный алгоритм представлен в виде программы 2.1.

```
procedure straightinsertion;
  var i,j: index; x: item;
begin
  for i := 2 to n do
  begin x := a[i]; a[0] := x; j := i-1;
    while x.key < a[j].key do
      begin a[j+1] := a[j]: j := j-1;
      end ;
    a[j+1] := x
  end
end
```

Программа 2.1. Сортировка простыми включениями.

Анализ сортировки простыми включениями. Число C_i сравнений ключей при i-м просеивании составляет самое большее $i-1$, самое меньшее 1 и, если предположить, что все перестановки n ключей равновероятны, в среднем равно $i/2$. Число M_i пересылок (присваиваний) равно $C_i + 2$ (учитывая барьер). Поэтому общее число сравнений и пересылок есть

$$\begin{aligned} C_{\min} &= n-1 & M_{\min} &= 2(n-1) \\ C_{\text{ср.}} &= \tfrac{1}{4}(n^2+n-2) & M_{\text{ср.}} &= \tfrac{1}{4}(n^2+9n-10) \\ C_{\max} &= \tfrac{1}{2}(n^2+n)-1 & M_{\max} &= \tfrac{1}{2}(n^2+3n-4) \end{aligned} \qquad (2.4)$$

Наименьшие числа появляются, если элементы с самого начала упорядочены, а наихудший случай встречается, если элементы расположены в обратном порядке. В этом смысле сортировка включениями демонстрирует вполне *естественное поведение*. Ясно также, что данный алгоритм описывает *устойчивую* сортировку: он оставляет неизменным порядок элементов с одинаковыми ключами.

Алгоритм сортировки простыми включениями легко можно улучшить, пользуясь тем, что готовая последовательность $a_1, \ldots, a_{i-1}$, в которую нужно включить новый элемент, уже упорядочена. Поэтому место включения можно найти значительно быстрее. Очевидно, что здесь можно применить бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения. Модифицированный алгоритм сор-

тировки называется *сортировкой бинарными включениями*, он показан в программе 2.2.

```
procedure binaryinsertion;
  var i,j,l,r,m: index;  x: item;
begin
  for i := 2 to n do
  begin x := a[i]; l := 1; r := i-1;
    while l ≤ r do
    begin m := (l+r) div 2;
      if x.key < a[m].key then r := m-1 else l := m+1
    end ;
    for j := i-1 downto l do a[j+1] := a[j];
    a[l] := x;
  end
end
```

Программа 2.2. Сортировка бинарными включениями.

Анализ сортировки бинарными включениями. Место включения найдено, если $a_l.key \leqslant x.key < a_r.key$. Таким образом, интервал поиска в конце должен быть равен 1; это означает, что интервал из i ключей делится пополам $\lceil \log_2 i \rceil$ раз. Итак,

$$C = \sum_{i=1}^{n} \lceil \log_2 i \rceil.$$

Мы аппроксимируем эту сумму с помощью интеграла

$$\int_1^n \log x\, dx = x(\log x - c)\Big|_1^n = n(\log n - c) + c, \qquad (2.5)$$

где $c = \log e = 1/\ln 2 = 1.44269...$. Количество сравнений не зависит от исходного порядка элементов. Но из-за округления при делении интервала поиска пополам действительное число сравнений для i элементов может быть на 1 больше ожидаемого. Природа этого «перекоса» такова, что в результате места включения в нижней части находятся в среднем несколько быстрее, чем в верхней части. Это дает преимущество в тех случаях, когда элементы изначально далеки от правильного порядка. На самом же деле минимальное число сравнений требуется, если элементы вначале расположены в обратном порядке, а максимальное — если они уже упорядочены. Следовательно, это случай *неестественного поведения* алгоритма сортировки:

$$C \doteq n(\log n - \log e \pm 0.5).$$

К сожалению, улучшение, которое мы получаем, используя метод бинарного поиска, касается только числа сравнений,

а не числа необходимых пересылок. В действительности поскольку пересылка элементов, т. е. ключей и сопутствующей информации, обычно требует значительно больше времени, чем сравнение двух ключей, то это улучшение ни в коей мере не является решающим: важный показатель M по-прежнему остается порядка n^2. И в самом деле, пересортировка уже рассортированного массива занимает больше времени, чем при сортировке простыми включениями с последовательным поиском! Этот пример показывает, что «очевидное улучшение» часто оказывается намного менее существенным, чем кажется вначале, и в некоторых случаях (которые *действительно* встречаются) может на самом деле оказаться ухудшением. В конечном счете сортировка включениями оказывается не очень подходящим методом для цифровых вычислительных машин: включение элемента с последующим сдвигом всего ряда элементов на одну позицию неэкономна. Лучших результатов можно ожидать от метода, при котором пересылки элементов выполняются только для отдельных элементов и на большие расстояния. Эта мысль приводит к сортировке выбором.

2.2.2. Сортировка простым выбором

Этот метод основан на следующем правиле:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом a_1.

Эти операции затем повторяются с оставшимися $n-1$ элементами, затем с $n-2$ элементами, пока не останется только один элемент — наибольший. Этот метод продемонстрирован на тех же восьми ключах в табл. 2.2.

Таблица 2.2. Пример сортировки простым выбором

Начальные ключи								
	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

Программу можно представить следующим образом:

```
for i := 1 to n — 1 do
    begin «присвоить k индекс наименьшего элемента из a[i]...
          ... a[n]»;
          «поменять местами a_i и a_k»
    end
```

Этот метод, называемый *сортировкой простым выбором*, в некотором смысле противоположен сортировке простыми включениями; при сортировке простыми включениями на каждом шаге рассматривается только *один* очередной элемент *входной последовательности* и *все* элементы *готового массива* для нахождения места включения; при сортировке простым выбором рассматриваются *все* элементы *входного массива* для нахождения элемента с наименьшим ключом, и этот *один* очередной элемент отправляется в *готовую последовательность*. Весь алгоритм сортировки простым выбором представлен в виде программы 2.3.

```
procedure straightselection;
   var i,j,k: index;  x: item;
begin for i := 1 to n-1 do
      begin k := i; x := a[i];
         for j := i+1 to n do
            if a[j].key < x.key then
            begin k := j; x := a[j]
            end ;
         a[k] := a[i]; a[i] := x;
      end
end
```

Программа 2.3. Сортировка простым выбором.

Анализ сортировки простым выбором. Очевидно, что число C сравнений ключей не зависит от начального порядка ключей. В этом смысле можно сказать, что сортировка простым выбором ведет себя менее естественно, чем сортировка простыми включениями. Мы получаем

$$C = \tfrac{1}{2}(n^2 - n).$$

Минимальное число пересылок равно

$$M_{\min} = 3(n-1) \tag{2.6}$$

в случае изначально упорядоченных ключей и принимает наибольшее значение:

$$M_{\max} = \operatorname{trunc}\left(\frac{n^2}{4}\right) + 3(n-1),$$

если вначале ключи расположены в обратном порядке. Среднее $M_{\text{ср.}}$ трудно определить, несмотря на простоту алгоритма. Оно зависит от того, сколько раз определяется, что k_i меньше всех предшествующих величин $k_1, \ldots, k_{i-1}$ при просмотре последовательности чисел $k_1, \ldots, k_n$. Это значение, взятое

в среднем для всех перестановок n ключей, число которых равно $n!$, есть

$$H_n - 1,$$

где H_n — n-е *гармоническое число*

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \tag{2.7}$$

(см. Д. Кнут, т. 1).

Число H_n можно выразить как

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \ldots, \tag{2.8}$$

где $\gamma = 0.577216\ldots$ — эйлерова константа. Для достаточно больших n мы можем опустить дробные слагаемые и, таким образом, аппроксимировать среднее число присваиваний на i-м проходе следующим образом:

$$F_i = \ln i + \gamma + 1.$$

Тогда среднее число пересылок $M_{\text{ср.}}$ при сортировке выбором есть сумма F_i, где i принимает значения от 1 до n:

$$M_{\text{ср.}} = \sum_{i=1}^{n} F_i = n(\gamma + 1) + \sum_{i=1}^{n} \ln i.$$

Аппроксимируя далее сумму отдельных слагаемых с помощью интеграла

$$\int_1^n \ln x\, dx = x(\ln x - 1)\Big|_1^n = n \ln n - n + 1,$$

получаем приближенное значение

$$M_{\text{ср.}} \doteq n(\ln n + \gamma). \tag{2.9}$$

Мы можем сделать вывод, что обычно алгоритм сортировки простым выбором предпочтительней алгоритма сортировки простыми включениями, хотя в случае, когда ключи заранее рассортированы или почти рассортированы, сортировка простыми включениями все же работает несколько быстрее.

2.2.3. Сортировка простым обменом

Классификация методов сортировки не всегда четко определена. Оба представленных ранее метода можно рассматривать как сортировку обменом. Однако в этом разделе мы остановимся на методе, в котором обмен двух элементов является основной характеристикой процесса. Приведенный ниже алгоритм сортировки простым обменом основан на

принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут рассортированы все элементы.

Как и в предыдущих методах простого выбора, мы совершаем повторные проходы по массиву, каждый раз просеивая наименьший элемент оставшегося множества, двигаясь к левому концу массива. Если, для разнообразия, мы будем рассматривать массив, расположенный вертикально, а не горизонтально и — при помощи некоторого воображения — представим себе элементы пузырьками в резервуаре с водой, обладающими «весами», соответствующими их ключам, то каждый проход по массиву приводит к «всплыванию» пузырька на соответствующий его весу уровень (см. табл. 2.3). Этот ме-

Таблица 2.3. Пример сортировки методом пузырька

Начальные ключи	*i* = 2	*i* = 3	*i* = 4	*i* = 5	*i* = 6	*i* = 7	*i* = 8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

тод широко известен как *сортировка методом пузырька*. Его простейший вариант приведен в программе 2.4.

```
procedure bubblesort;
   var i,j: index;  x: item;
begin for i := 2 to n do
      begin for j := n downto i do
            if a[j-1].key > a[j].key then
            begin x := a[j-1]; a[j-1] := a[j]; a[j] := x
            end
      end
end {bubblesort}
```

Программа 2.4. Сортировка методом пузырька.

, Этот алгоритм легко оптимизировать. Пример в табл. 2.3 показывает, что три последних прохода никак не влияют на порядок элементов, поскольку те уже рассортированы. Очевидный способ улучшить данный алгоритм — это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то это означает, что алгоритм может закончить работу. Этот процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и место (индекс) последнего

обмена. Ведь ясно, что все пары соседних элементов с индексами, меньшими этого индекса k, уже расположены в нужном порядке. Поэтому следующие проходы можно заканчивать на этом индексе, вместо того чтобы двигаться до установленной заранее нижней границы i. Однако внимательный программист заметит здесь странную асимметрию: один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только на один шаг на каждом проходе. Например, массив

12 18 42 44 55 67 94 06

будет рассортирован при помощи метода пузырька за один проход, а сортировка массива

94 06 12 18 42 44 55 67

потребует семи проходов. Эта неестественная асимметрия подсказывает третье улучшение: менять направление следующих один за другим проходов. Мы назовем полученный в результате алгоритм *шейкер-сортировкой*. Его работа показана в табл. 2.4 на тех же восьми ключах, которые использовались в табл. 2.3.

Таблица 2.4. Пример шейкер-сортировки

l = 2	3	3	4	4
r = 8	8	7	7	4
44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	42	44
18	94	18	55	55
06	18	67	67	67
67	67	94	94	94

Анализ сортировки методом пузырька и шейкер-сортировки. Число сравнений в алгоритме простого обмена равно

$$C = \tfrac{1}{2}(n^2 - n), \tag{2.10}$$

минимальное, среднее и максимальное количества пересылок (присваиваний элементов) равны

$$M_{\min} = 0, \quad M_{\text{ср.}} = \tfrac{1}{4}(n^2 - n), \quad M_{\max} = \tfrac{1}{2}(n^2 - n). \tag{2.11}$$

```
procedure shakersort;
   var j,k,l,r: index;  x: item;
begin l := 2; r := n; k := n;
   repeat
      for j := r downto l do
         if a[j-1] .key > a[j] .key then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end ;
      l := k+1;
      for j := l to r do
         if a[j-1] .key > a[j] .key then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end ;
      r := k-1;
   until l > r
end {shakersort}
```

Программа 2.5. Шейкер-сортировка.

Анализ улучшенных методов, особенно метода шейкер-сортировки, довольно сложен. Наименьшее число сравнений есть $C_{\min} = n - 1$. Для усовершенствованного метода пузырька Кнут получил, что среднее число проходов пропорционально $n - k_1\sqrt{n}$ и среднее число сравнений пропорционально $\frac{1}{2}[n^2 - n(k_2 + \ln n)]$. Но мы замечаем, что все предложенные выше усовершенствования никоим образом не влияют на число обменов; они лишь уменьшают число избыточных повторных проверок. К сожалению, обмен двух элементов — обычно намного более дорогостоящая операция, чем сравнения ключей; поэтому все наши усовершенствования дают значительно меньший эффект, чем можно было бы ожидать.

Анализ показывает, что сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором, и действительно, сортировка методом пузырька вряд ли имеет какие-то преимущества, кроме своего легко запоминающегося названия. Алгоритм шейкер-сортировки выгодно использовать в тех случаях, когда известно, что элементы уже почти упорядочены — редкий случай на практике.

Можно показать, что среднее расстояние, на которое должен переместиться каждый из n элементов во время сортировки, — это $n/3$ мест. Это число дает ключ к поиску усовершенствованных, т. е. более эффективных, методов сорти-

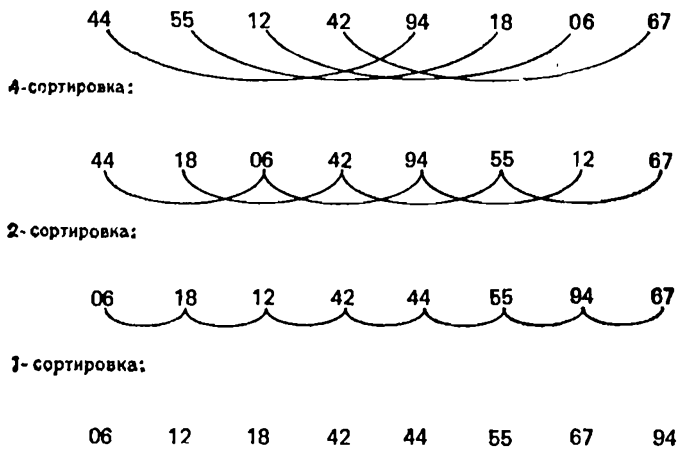
ровки. Все простые методы в принципе перемещают каждый элемент на одну позицию на каждом элементарном шаге. Поэтому они требуют порядка n^2 таких шагов. Любое улучшение должно основываться на принципе пересылки элементов за один цикл на большее расстояние.

Далее мы обсудим три усовершенствованных метода — по одному для каждого основного метода сортировки: включения, выбора и обмена.

2.2.4. Сортировка включениями с убывающим приращением

Некоторое усовершенствование сортировки простыми включениями было предложено Д. Л. Шеллом в 1959 г. Этот метод мы объясним и продемонстрируем на нашем стандартном примере из восьми элементов (см. табл. 2.5). На первом

Таблица 2.5. Сортировка включениями с убывающим приращением



проходе отдельно группируются и сортируются все элементы, отстоящие друг от друга на четыре позиции. Этот процесс называется 4-сортировкой. В нашем примере из восьми элементов каждая группа содержит ровно два элемента. После этого элементы вновь объединяются в группы с элементами, отстоящими друг от друга на две позиции, и сортируются заново. Этот процесс называется 2-сортировкой. Наконец, на третьем проходе все элементы сортируются обычной сортировкой, или 1-сортировкой.

Сначала может показаться, что необходимость нескольких проходов сортировки, в каждом из которых участвуют все элементы, больше работы потребует, чем сэкономит. Однако

на каждом шаге сортировки либо участвует сравнительно мало элементов, либо они уже довольно хорошо упорядочены и требуют относительно мало перестановок.

Очевидно, что этот метод в результате дает упорядоченный массив, и также совершенно ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая i-сортировка объединяет две группы, рассортированные предыдущей $2i$-сортировкой. Также ясно, что приемлема любая последовательность приращений, лишь бы последнее было равно 1, так как в худшем случае вся работа будет выполняться на последнем проходе. Однако менее очевидно, что метод убывающего приращения дает даже *лучшие* результаты, когда приращения не являются степенями двойки.

Таким образом, программа разрабатывается вне связи с конкретной последовательностью приращений. Все t приращений обозначаются через

$$h_1, h_2, \ldots, h_t$$

с условиями

$$h_t = 1, \quad h_{t+1} < h_t. \tag{2.12}$$

Каждая h-сортировка программируется как сортировка простыми включениями, при этом, для того чтобы условие окончания поиска места включения было простым, используется барьер.

Ясно, что каждая h-сортировка требует собственного барьера и что программа должна определять его место как можно проще. Поэтому массив a нужно дополнить не одной компонентой $a[0]$, а h_1 компонентами, так что теперь он описывается как

$$a: \textbf{array}[-h_1 \,..\, n] \textbf{ of } item$$

Этот алгоритм представлен в виде процедуры, названной *Shellsort* [2.11] («сортировка Шелла») в программе 2.6 для $t = 4$.

Анализ сортировки Шелла. При анализе этого алгоритма возникают некоторые очень сложные математические задачи, многие из которых еще не решены. В частности, неизвестно, какая последовательность приращений дает лучшие результаты. Однако выявлен удивительный факт, что они не должны быть кратны друг другу. Это позволяет избежать явления, которое видно в приведенном выше примере, где каждый проход сортировки объединяет две цепочки, которые ранее никак не взаимодействовали. В действительности желательно, чтобы взаимодействие между разными цепочками

```
procedure shellsort;
   const t = 4;
   var i,j,k,s: index; x: item; m: 1..t;
      h: array [1..t] of integer;
begin h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
   for m := 1 to t do
   begin k := h[m]; s := -k; {место барьера}
      for i := k+1 to n do
      begin x := a[i]; j := i-k;
         if s=0 then s := -k; s := s+1; a[s] := x;
         while x.key < a[j].key do
         begin a[j+k] := a[j]; j := j-k
         end ;
         a[j+k] := x
      end
   end
end
```

Программа 2.6. Сортировка Шелла.

происходило как можно чаще. Можно сформулировать следующую теорему:

Если k-рассортированная последовательность i-сортируется, то она остается k-рассортированной.

Кнут [2.8] указывает, что разумным выбором может быть такая последовательность приращений (записанная в обратном порядке):

$$1, 4, 13, 40, 121, \ldots,$$

где $h_{k-1} = 3h_k + 1$, $h_t = 1$ и $t = [\log_3 n] - 1$. Он рекомендует также последовательность

$$1, 3, 7, 15, 31, \ldots,$$

где $h_{k-1} = 2h_k + 1$, $h_t = 1$ и $t = [\log_2 n] - 1$. Дальнейший анализ показывает, что в последнем случае затраты, которые требуются для сортировки n элементов с помощью алгоритма сортировки Шелла, пропорциональны n. Хотя это — значительное улучшение по сравнению с n^2, мы не будем в дальнейшем обращаться к этому методу, поскольку известны алгоритмы, работающие еще лучше.

2.2.5. Сортировка с помощью дерева

Метод сортировки простым выбором основан на повторном выборе наименьшего ключа среди n элементов, затем среди $n-1$ элементов и т. д. Понятно, что поиск наименьшего

ключа из n элементов требует $n-1$ сравнений, а поиск его среди $n-1$ элементов требует $n-2$ сравнений. Итак, как можно улучшить эту сортировку выбором? Это можно сделать только в том случае, если получать от каждого прохода больше информации, чем просто указание на один, наименьший элемент. Например, с помощью $n/2$ сравнений можно определить наименьший ключ из каждой пары, при помощи

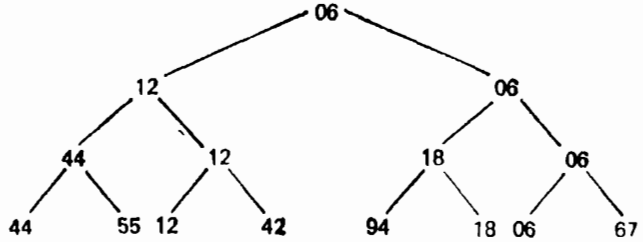


Рис. 2.3. Циклический выбор из двух ключей.

следующих $n/4$ сравнений можно выбрать наименьший из каждой пары таких наименьших ключей и т. д. Наконец, при помощи всего $n-1$ сравнений мы можем построить дерево выбора, как показано на рис. 2.3, и определить корень как наименьший ключ [2.2].

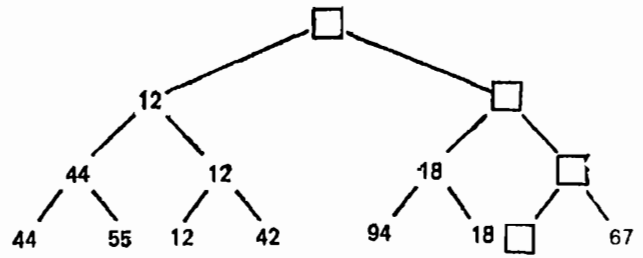


Рис. 2.4. Выбор наименьшего ключа.

На втором шаге мы спускаемся по пути, указанному наименьшим ключом, и исключаем его, последовательно заменяя либо на «дыру» (или ключ ∞), либо на элемент, находящийся на противоположной ветви промежуточного узла (см. рис. 2.4 и 2.5). Элемент, оказавшийся в корне дерева, вновь имеет наименьший ключ (среди оставшихся) и может быть исключен. После n таких шагов дерево становится пустым (т. е. состоит из «дыр»), и процесс сортировки закончен. Отметим, что каждый из n шагов требует лишь $\log_2 n$ сравнений. Поэтому вся сортировка требует лишь порядка $n \cdot \log_2 n$ элементарных операций, не считая n шагов, которые необходимы для построения дерева. Это — значительное улуч-

шение по сравнению с простым методом, требующим n^2 шагов и даже по сравнению с сортировкой Шелла, которая требует $n^{1.2}$ шагов.

Конечно, при сортировке с помощью дерева задача хранения информации стала сложнее и поэтому увеличилась сложность отдельных шагов; в конечном счете для хранения возросшего объема информации, получаемой на начальном проходе, нужно строить некую древовидную структуру. Наша очередная задача — найти способы эффективной организации этой информации.

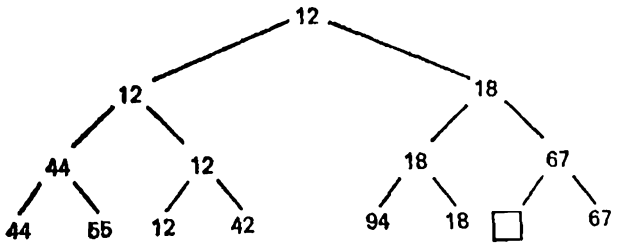


Рис. 2.5. Заполнение «дыр».

Разумеется, было бы весьма желательно избавиться от необходимости в дырах ($-\infty$), которые в конце заполняют все дерево и приводят к большому количеству ненужных сравнений. Кроме того, нужно найти способ представить дерево из n элементов в n единицах памяти вместо $2n-1$ единиц, как показано выше. Это действительно можно сделать с помощью метода, который его изобретатель Дж. Уильямс [2.14] назвал *пирамидальной сортировкой*. Ясно, что этот метод дает существенное улучшение по сравнению с более привычными способами сортировки по дереву.

Пирамида определяется как последовательность ключей

$$h_l, h_{l+1}, \ldots, h_r$$

такая, что

$$\begin{aligned} h_i &\leqslant h_{2i}, \\ h_i &\leqslant h_{2i+1} \end{aligned} \qquad (2.13)$$

для всякого $i = l, \ldots, r/2$. Если двоичное дерево представлено в виде массива, как показано на рис. 2.6, то, следовательно, деревья сортировки на рис. 2.7 и 2.8 являются пирамидами, и, в частности, элемент h_1 пирамиды является ее *наименьшим* элементом

$$h_1 = \min(h_1 \ldots h_n).$$

Теперь предположим, что дана пирамида с элементами $h_{l+1}, \ldots, h_r$ для некоторых значений l и r и нужно добавить

новый элемент x для того, чтобы сформировать расширенную пирамиду $h_l, \ldots, h_r$. Возьмем, например, исходную пирамиду $h_1, \ldots, h_7$, показанную на рис. 2.7, и расширим эту пирамиду «влево», добавив элемент $h_1 = 44$. Новый элемент x сначала помещается в вершину дерева, а затем «просеивается» по пути, на котором находятся меньшие по сравнению с ним элементы, которые одновременно поднимаются вверх; таким

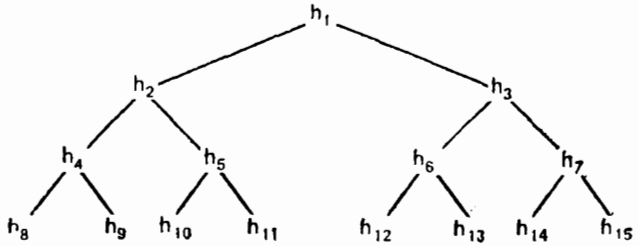


Рис. 2.6. Массив h, расположенный в виде бинарного дерева.

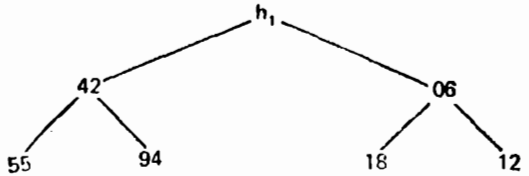


Рис. 2.7. Пирамида из семи элементов.

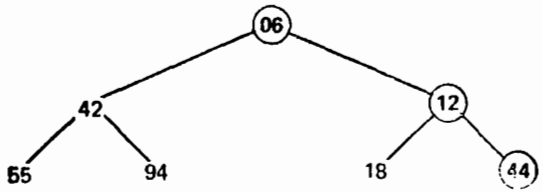


Рис. 2.8. Просеивание ключа 44 через пирамиду.

образом формируется новая пирамида. В данном примере значение 44 сначала меняется местами с 06, затем с 12, и так формируется дерево, показанное на рис. 2.8. Далее мы процесс просеивания будем формулировать следующим образом: i, j — пара индексов, обозначающих элементы, которые нужно менять местами на каждом шаге просеивания. Мы предоставляем читателю возможность самому убедиться, что предложенный способ просеивания действительно позволяет сохранить условия (2.13), определяющие пирамиду.

Изящный способ построения пирамиды *in situ* был предложен Р. У. Флойдом. В нем используется следующая про-

цедура просеивания (программа 2.7). Дан массив $h_1, \ldots, h_n$; ясно, что элементы $h_{n/2+1} \ldots h_n$ уже образуют пирамиду, поскольку не существует двух индексов i, j, таких, что $j = 2i$ (или $j = 2i + 1$). Эти элементы составляют последовательность, которую можно рассматривать как нижний ряд соответствующего двоичного дерева (см. рис. 2.6), где не тре-

```
procedure sift(l,r: index);
   label 13;
   var i,j: index;  x: item;
begin i := l; j := 2*i; x := a[i];
   while j ≤ r do
   begin if j < r then
            if a[j] .key > a[j+1] .key then j := j+1;
         if x .key ≤ a[j] .key then goto 13;
         a[i] := a[j]; i := j; j := 2*i {sift}
   end;
13: a[i] := x
end
```

Программа 2.7. Просеивание.

буется никакого упорядочения. Теперь пирамида расширяется влево: на каждом шаге добавляется новый элемент и при помощи просеивания помещается на соответствующее место.

Таблица 2.6. Построение пирамиды

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
44	55	06	42	94	18	12	67
44	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

Этот процесс иллюстрируется табл. 2.6 и приводит к пирамиде, показанной на рис. 2.6. Следовательно, процесс построения пирамиды из n элементов *in situ* можно описать следующим образом:

```
l := (n div 2) + 1;
while l > 1 do
     begin l := l-1; sift(l,n)
     end
```

Для того чтобы рассортировать элементы, надо выполнить n шагов просеивания: после каждого шага очередной элемент берется с вершины пирамиды. Вновь встает вопрос, куда помещать элементы с вершины и возможна ли сортировка *in situ*. Да, такое решение существует! На каждом шаге из пирамиды выбирается последняя компонента (скажем, x), верхний элемент пирамиды помещается на освободившееся место x, а x просеивается на свое место. В этом случае необходимо совершить $n-1$ шагов, что показано на примере

Таблица 2.7. Пример пирамидальной сортировки

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

пирамиды, приведенной в табл. 2.7. Этот процесс описывается с помощью процедуры *sift* (программа 2.7) следующим образом:

```
r := n;
while r > 1 do
   begin x := a[1]; a[1] := a[r]; a[r] := x;
      r := r-1; sift(1,r)
   end
```

Из табл. 2.7 видно, что на самом деле в результате мы получаем последовательность в обратном порядке. Но это легко можно исправить, изменив направление отношения порядка в процедуре *sift*. В результате мы получаем процедуру *Heapsort*, показанную в программе 2.8.

Анализ пирамидальной сортировки. С первого взгляда неочевидно, что этот метод сортировки дает хорошие результаты. Ведь элементы с большими ключами вначале просеиваются влево, прежде чем, наконец, окажутся справа. Действительно, эта процедура не рекомендуется для такого небольшого числа элементов, как, скажем, в нашем примере. Однако для больших n пирамидальная сортировка оказы-

```
procedure heapsort;
   var l,r: index;  x: item;
   procedure sift;
      label 13;
      var i,j: index;
   begin i := l; j := 2*i; x := a[i];
      while j ≤ r do
      begin if j < r then
               if a[j] .key < a[j+1] .key then j := j+1;
         if x .key ≥ a[j] .key then goto 13;
         a[i] := a[j]; i := j; j := 2*i
      end ;
   13: a[i] := x
   end ;
begin l := (n div 2) + 1; r := n;
   while l > 1 do
      begin l := l-1; sift
      end ;
   while r > 1 do
      begin x := a[1]; a[1] := a[r]; a[r] := x;
         r := r-1; sift
      end
end {heapsort}
```

Программа 2.8. Пирамидальная сортировка.

вается очень эффективной, и чем больше n, тем она эффективнее — даже по сравнению с сортировкой Шелла.

В худшем случае необходимы $n/2$ шагов, которые просеивают элементы через $\log(n/2)$, $\log(n/2-1)$, ..., $\log(n-1)$ позиций (здесь берется целая часть логарифма по основанию 2). Следовательно, на фазе сортировки происходит $n-1$ просеиваний с самое большее $\log(n-1)$, $\log(n-2)$, ..., 1 пересылками. Кроме того, требуются $n-1$ пересылок для того, чтобы отложить просеянный элемент вправо. Отсюда видно, что пирамидальная сортировка требует $n \cdot \log(n)$ шагов даже *в худшем случае*. Такие отличные характеристики для худшего случая — одно из самых выгодных качеств пирамидальной сортировки.

Не совсем ясно, в каких случаях можно ожидать наименьшей (или наибольшей) эффективности. Но в принципе для пирамидальной сортировки, видимо, больше всего подходят случаи, когда элементы более или менее рассортированы в обратном порядке, т. е. для нее характерно неестественное

поведение. Очевидно, что при обратном порядке фаза построения пирамиды не требует никаких пересылок. Для восьми элементов из нашего примера минимальное и максимальное количества пересылок дают следующие исходные последовательности:

$M_{\min} = 13$ для последовательности
94 67 44 55 12 42 18 6

$M_{\max} = 24$ для последовательности
18 42 12 44 6 55 67 94

Среднее число пересылок равно приблизительно $\frac{1}{2} n \cdot \log n$ и отклонения от этого значения сравнительно малы.

2.2.6. Сортировка с разделением

После того как мы обсудили два усовершенствованных метода сортировки, основанных на принципах включения и выбора, мы введем третий, улучшенный метод, основанный на принципе обмена. Учитывая, что сортировка методом пузырька в среднем была наименее эффективной из трех алгоритмов простой сортировки, мы должны требовать значительного улучшения. Однако неожиданно оказывается, что усовершенствование сортировки, основанной на обмене, которое мы здесь будем обсуждать, дает вообще лучший из известных до сего времени метод сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель К. Хоор окрестил его *быстрой сортировкой* [2.5, 2.6].

Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно производить обмены элементов на больших расстояниях. Предположим, что нам даны n элементов с ключами, расположенными в обратном порядке. Их можно рассортировать, выполнив всего $n/2$ обменов, если сначала поменять местами самый левый и самый правый элементы и так постепенно продвигаться с двух концов к середине. Разумеется, это возможно, только если мы знаем, что элементы расположены строго в обратном порядке. Но все же этот пример наводит на некоторые мысли.

Попробуем рассмотреть следующий алгоритм: выберем случайным образом какой-то элемент (назовем его x), просмотрим массив, двигаясь слева направо, пока не найдем элемент $a_i > x$, а затем просмотрим его справа налево, пока не найдем элемент $a_j < x$. Теперь поменяем местами эти два элемента и продолжим процесс «просмотра с обменом», пока два просмотра не встретятся где-то в середине массива. В результате массив разделится на две части: левую — с клю-

чами меньшими, чем x, и правую — с ключами большими x. Теперь запишем этот алгоритм разделения в виде процедуры в программе 2.9. Заметим, что отношения $>$ и $<$ заменены на $\geqslant$ и $\leqslant$, отрицания которых в операторе цикла с предусловием — это $<$ и $>$. При такой замене x действует как барьер для обоих просмотров.

```
procedure partition;
var w,x: item;
begin i := 1; j := n;
    выбор случайного элемента x;
    repeat
        while a[i].key < x .key do i := i+1;
        while x.key < a[j] .key 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
    until i > j
end
```

Программа 2.9. Разделение.

Если, например, в качестве x выбрать средний ключ, равный 42, из массива ключей

$$44\ 55\ 12\ \mathit{42}\ 94\ 06\ 18\ 67,$$

то для того, чтобы разделить массив, потребуются два обмена

$$18\ 06\ 12\ |\ 42\ |\ 94\ 55\ 44\ 67,$$

конечные значения индексов $i = 5$ и $j = 3$. Ключи $a_1, \ldots, a_{i-1}$ меньше или равны ключу $x = 42$, ключи $a_{j+1}, \ldots, a_n$ больше или равны x. Следовательно, мы получили два подмассива

$$\begin{aligned} a_k.key &\leqslant x.key \quad \text{для} \quad k = 1, \ldots, i-1, \\ a_k.key &\geqslant x.key \quad \text{для} \quad k = j+1, \ldots, n \end{aligned} \tag{2.14}$$

и, следовательно,

$$a_k.key = x.key \quad k = j+1, \ldots, i-1.$$

Этот алгоритм очень прост и эффективен, так как основные величины, участвующие в сравнениях: i, j и x, можно во время просмотра хранить на быстрых регистрах. Но он также может быть весьма неуклюжим, как видно из примера

с n одинаковыми ключами, в котором выполняется $n/2$ обменов. Эти ненужные обмены можно легко устранить, заменив операторы просмотра на

while $a[i]$ $.key \leq x$ $.key$ **do** $i := i+1$;
while x $.key \leq a[j]$ $.key$ **do** $j := j-1$;

Но тогда выбранный для сравнения элемент x, который находится в массиве, перестанет служить в качестве барьера для двух разнонаправленных просмотров. В случае когда в массиве все ключи одинаковы, просмотры выйдут за его границы, если не использовать более сложные условия окончания. Простота условий в программе 2.9 оправдывает излишние обмены, которые в среднем для «случайных» массивов происходят довольно редко. Небольшую экономию можно, однако, получить, заменив условие, управляющее обменом, на

$$i < j$$

вместе $i \leqslant j$. Но такую замену нельзя распространить на оба оператора

$$i := i + 1; \quad j := j - 1$$

которые поэтому требуют использования различных условий. Необходимость условия $i \leqslant j$ можно проиллюстрировать следующим примером при $x = 2$:

1 1 1 *2* 1 1 *1*

Первый просмотр и обмен дают

1 1 1 1 1 1 2

причем $i = 5$, $j = 6$. Второй просмотр не изменяет массив и заканчивается с $i = 7$ и $j = 6$. Если бы обмен не подчинялся условию $i \leqslant j$, то произошел бы ошибочный обмен a_6 и a_7.

В правильности алгоритма разделения можно убедиться на основании того, что оба утверждения (2.14) являются вариантами оператора цикла с постусловием. Вначале, при $i = 1$ и $j = n$, они, очевидно, истинны, а при выходе с $i > j$ они предполагают, что получен нужный результат.

Теперь нам пора вспомнить, что наша цель — не только разделить исходный массив элементов на бо́льшие и меньшие, но также рассортировать его. Однако от разделения до сортировки всего лишь один небольшой шаг: разделив массив, нужно сделать то же самое с обеими полученными частями, затем с частями этих частей и т. д., пока каждая часть не

будет содержать только один элемент. Этот метод представлен программой 2.10.

Процедура *sort* рекурсивно вызывает сама себя. Такое использование рекурсии в алгоритмах — очень мощное средство. Мы обсудим его позже в гл. 3. В некоторых языках программирования старого образца рекурсия по некоторым техническим причинам запрещена. Сейчас мы покажем, как

```
procedure quicksort;
   procedure sort (l,r: index);
      var i,j: index;  x,w: item;
   begin i := l; j := r;
      x := a[(l+r) div 2];
      repeat
         while a[i] .key < x .key do i := i+1;
         while x .key < a[j] .key 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
      until i > j;
      if l < j then sort(l,j);
      if i < r then sort(i,r)
   end ;
begin sort(1,n)
end {quicksort}
```

Программа 2.10. Быстрая сортировка.

можно выразить тот же алгоритм в виде нерекурсивной процедуры. Очевидно, что при этом рекурсия представляется как итерация, причем необходимы некоторые дополнительные операции для хранения информации.

Основа итеративного решения — ведение списка запросов на разделения, которые еще предстоит выполнить. После каждого шага нужно произвести два очередных разделения, и лишь одно из них можно выполнить непосредственно при следующей итерации, запрос на другое заносится в список. Важно, разумеется, что запросы из списка выполняются в обратной последовательности. Это предполагает, что первый занесенный в список запрос выполняется последним и наоборот; список ведет себя как пульсирующий стек. В следующей нерекурсивной версии быстрой сортировки каждый запрос представлен просто левым и правым индексами, определяющими границы части, которую впоследствии нужно

будет разделить. Итак, мы вводим переменную-массив, называемую *stack*, и индекс *s*, указывающий на самую последнюю запись в этом стеке (см. программу 2.11). Каким дол-

```
procedure quicksort 1;
   const m = 12;
   var i,j,l,r: index;
      x,w: item;
      s: 0 .. m;
      stack: array [1 .. m] of
            record l,r: index end;
begin s := 1; stack[1] .l := 1; stack[1] .r := n;
   repeat {выбор запроса из вершины стека}
      l := stack[s] .l; r := stack[s] .r; s := s-1;
      repeat {split a[l] ... a[r]}
         i := l; j := r; x := a[(l+r) div 2];
         repeat
            while a[i] .key < x .key do i := i+1;
            while x .key < a[j] .key 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
         until i > j;
         if i < r then
         begin {запись в стек запроса на сортировку
                правой части}
            s := s+1; stack[s] .l := i; stack[s] .r := r
         end ;
         r := j.
      until l ≥ r
   until s = 0
end {quicksort 1}
```

Программа 2.11. Нерекурсивная версия быстрой сортировки.

жен быть размер стека *m*, мы обсудим при анализе быстрой сортировки.

Анализ быстрой сортировки. Для того чтобы проанализировать свойства быстрой сортировки, мы должны сначала изучить поведение процесса разбиения. После выбора гра-

ницы x процессу разбиения подвергается весь массив. Таким образом, выполняется ровно n сравнений. Число обменов можно оценить при помощи следующего вероятностного рассуждения.

Предположим, что множество данных, которые нужно разделить, состоит из n ключей $1, \ldots, n$, и мы выбрали x в качестве границы. После разделения x будет занимать в массиве позицию x. Число требующихся обменов равно числу элементов в левой части $x-1$, умноженному на вероятность того, что ключ нужно обменять. Ключ обменивается, если он не меньше чем x. Вероятность этого равна $(n-x+1)/n$. Ожидаемое число обменов вычисляется при помощи суммирования всех возможных вариантов выбора границы и деления этой суммы на n:

$$M = \frac{1}{n}\sum_{x=1}^{n} \frac{x-1}{n} \cdot (n-x+1) = \frac{n}{6} - \frac{1}{6n}. \qquad (2.15)$$

Следовательно, ожидаемое число обменов равно приблизительно $n/6$.

Если предположить, что нам очень везет и мы всегда выбираем в качестве границы медиану *), то каждое разделение разбивает массив на две равные части и число проходов, необходимых для сортировки, равно $\log n$. Тогда общее число сравнений составит $n \cdot \log n$, а общее число обменов — $(n/6) \cdot \log n$. Разумеется, нельзя ожидать, что мы все время будем попадать на медиану. На самом деле, вероятность этого равна всего лишь $1/n$. Но к удивлению, если граница выбирается случайным образом, эффективность быстрой сортировки в среднем хуже оптимальной лишь в $2 \cdot \ln 2$ раз.

Однако быстрая сортировка все же имеет свои «подводные камни». Прежде всего при небольших значениях n ее эффективность невелика, как и у всех усовершенствованных методов. Ее преимущество по сравнению с другими усовершенствованными методами заключается в том, что для сортировки уже разделенных небольших подмассивов легко можно применить какой-либо простой метод. Это особенно ценно, если говорить о рекурсивной версии программы.

Тем не менее остается проблема наихудшего случая. Как тогда ведет себя быстрая сортировка? Ответ, к сожалению, разочаровывает. Здесь проявляется слабость быстрой сортировки (которая в таких случаях становится «медленной сортировкой»). Рассмотрим, например, неблагоприятный случай,

*) См. следующий раздел. — *Прим. перев.*

когда каждый раз в качестве x выбирается наибольшее значение в подмассиве. Тогда каждый шаг разбивает сегмент из n элементов на левую часть из $n-1$ элементов и правую часть, состоящую из одного элемента. В результате вместо $\log n$ необходимо n разбиений, и скорость работы в наихудшем случае оказывается порядка n^2.

Очевидно, что эффективность алгоритма быстрой сортировки определяется выбором элемента x. В нашем примере программа выбирает в качестве x элемент, расположенный посередине. Заметим, что почти с тем же успехом можно было бы выбрать либо первый, либо последний элемент: $a[l]$ или $a[r]$. Но при таком выборе наихудший вариант встретится, когда массив уже предварительно рассортирован; быстрая сортировка в этом случае проявляет определенную «неприязнь» к тривиальной работе и предпочитает беспорядочные массивы. При выборе среднего элемента в качестве границы это странное свойство быстрой сортировки менее заметно, так как уже рассортированный массив становится оптимальным случаем! Действительно, средняя скорость работы здесь оказывается несколько выше, если выбирается средний элемент. Хоор считает, что выбор x должен быть «случайным», или в качестве x нужно выбирать медиану из небольшого числа ключей (скажем, из 3-ех) [2.12, 2.13]. Такой осмотрительный выбор почти не влияет на среднюю скорость быстрой сортировки, но значительно улучшает скорость в худшем случае. Как мы видим, быстрая сортировка напоминает азартную игру, где следует заранее рассчитать, сколько можно позволить себе проиграть в случае невезения.

Из этого нужно сделать важный вывод, на который программист должен обратить особое внимание. К чему приводит наихудший случай, разобранный выше в связи со скоростью выполнения программы 2.11? Мы видим, что при каждом разбиении правая часть подмассива состоит из одного элемента; запрос на сортировку этой части заносится в стек для последующего выполнения. Следовательно, максимальное число запросов и поэтому необходимый общий размер стека оказываются равными n. Конечно же, это абсолютно неприемлемо. (Заметим, что с рекурсивной версией дело обстоит не лучше, а на самом деле даже хуже, поскольку система, допускающая рекурсивный вызов процедур, должна автоматически сохранять значения локальных переменных и параметров при всех вызовах процедур, и для этой цели она использует неявный стек.) Это можно исправить, если хранить в стеке запрос на сортировку более длинной части и сразу продолжать дальнейшее разделение коротких частей. В этом случае размер стека можно ограничить до $m = \log_2 n$.

Необходимое изменение программы 2.11 касается лишь части, фиксирующей новые запросы. Она теперь имеет вид

```
if j-l < r-i then
begin if i < r then
    begin {записи запроса на сортировку правой части}
        s := s+1; stack[s] .l := i; stack[s] .r := r
    end;
    r := j  {продолжение сортировки левой части}
end else
begin if l < j then
    begin {запись в стек запроса на сортировку левой части}
        s := s+1; stack[s] .l := k; stack[s] .r := j
    end;
    l := i  {продолжение сортировки правой части}
end
```

(2.16)

2.2.7. Поиск медианы

Медианой последовательности из n элементов называется элемент, значение которого меньше (или равно) половины n элементов и больше (или равно) другой половины. Например, медиана

16 12 99 95 18 87 10

есть 18.

Задачу поиска медианы принято связывать с сортировкой, так как медиану всегда можно найти следующим способом: рассортировать n элементов и затем выбрать средний элемент. Но разделение, которое выполняет программа 2.9, позволяет потенциально найти медиану значительно быстрее. Рассматриваемый здесь метод дает возможность решать и более общую задачу поиска элемента с k-м по величине значением из n элементов. Поиск медианы является частным случаем для $k = n/2$.

Алгоритм, изобретенный К. Хоором [2.4], работает следующим образом. Прежде всего применяется операция разделения, используемая при быстрой сортировке, с $l = 1$, $r = n$ и с $a[k]$, выбранным в качестве разделяющего значения (границы) x. Получаются значения индексов i и j, такие, что

$$\begin{aligned} &1)\ a[h] \leqslant x \quad \text{для всех} \quad h < i, \\ &2)\ a[h] \geqslant x \quad \text{для всех} \quad h > j, \\ &3)\ i > j. \end{aligned} \tag{2.17}$$

Возможны три варианта:

1. Разделяющее значение x было слишком мало; в результате граница между двумя частями ниже искомого значения k. Процесс разбиения следует повторить для элементов $a[i], \ldots, a[r]$ (см. рис. 2.9).

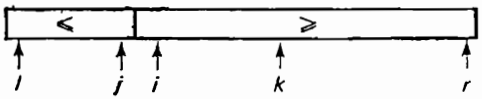


Рис. 2.9. Граница слишком низко.

2. Выбранная граница x была слишком велика. Операцию разбиения следует повторить на подмассиве $a[l], \ldots, a[j]$ (см. рис. 2.10).

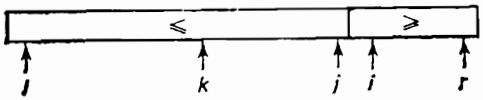


Рис. 2.10. Граница слишком высоко.

3. Значение k лежит в интервале $j < k < i$: элемент $a[k]$ разделяет массив в заданной пропорции и, следовательно, является искомым (см. рис. 2.11).

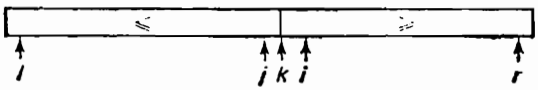


Рис. 2.11. Граница проведена правильно.

Процесс разбиения повторяется до появления случая 3. Этой итерации соответствует следующий фрагмент программы:

```
l := 1; r := n;
while l < r do
   begin x := a[k];
      partition(a[l] ... a[r]);
      if j < k then l := i;
      if k < i then r := j
   end
```
(2.18)

За формальным доказательством корректности этого алгоритма мы отсылаем читателя к статье Хоора. Теперь мы можем целиком написать всю программу *Find*.

```
procedure find (k; integer);
    var l,r,i,j,w,x: integer;
begin l := 1; r := n;
    while l < r do
    begin x := a[k]; i := l; j := r;
        repeat {split}
            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
        until i > j;
        if j < k then l := i;
        if k < i then r := j
    end
end {find}
```

Программа 2.12. Поиск k-го элемента.

Если предположить, что в среднем каждое разбиение уменьшает вдвое размер подмассива, в котором содержится искомый элемент, то число необходимых сравнений равно

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \doteq 2n, \tag{2.19}$$

т. е. порядка n. Этим объясняется эффективность программы *Find* при нахождении медиан и других квантилей и ее преимущество по сравнению с приходящим вначале в голову методом сортировки всего множества элементов для выбора k-го по величине (такой метод в лучшем случае дает порядок $n \cdot \log n$). Однако в худшем случае каждый шаг разбиения уменьшает размер множества, в котором ищется нужный элемент, только на 1, и поэтому требуется порядок n^2 сравнений. Следовательно, этот алгоритм тоже вряд ли стоит использовать для небольшого числа элементов (скажем, меньше 10).

2.2.8. Сравнение методов сортировки массивов

В завершение нашего обзора методов сортировки мы попытаемся сравнить их эффективность. Пусть n по-прежнему обозначает число сортируемых элементов, а C и M — соответственно количество необходимых сравнений ключей и

пересылок элементов. Для всех трех простых методов сортировки можно дать замкнутые аналитические формулы. Они приведены в табл. 2.8. Заголовки столбцов Min, Max, Средн. определяют соответственно минимумы, максимумы и ожидаемые средние значения для всех $n!$ перестановок n элементов.

Таблица 2.8. Сравнение простых методов сортировки

	Min	Средн	Max
Простые включения	$C = n - 1$ $M = 2(n - 1)$	$(n^2 + n - 2)/4$ $(n^2 - 9n - 10)/4$	$(n^2 - n)/2 - 1$ $(n^2 + 3n - 4)/2$
Простой выбор	$C = (n^2 - n)/2$ $M = 3(n - 1)$	$(n^2 - n)/2$ $n(\ln n + 0{,}57)$	$(n^2 - n)/2$ $n^2/4 + 3(n - 1)$
Простой обмен (метод пузырька)	$C = (n^2 - n)/2$ $M = 0$	$(n^2 - n)/2$ $(n^2 - n) * 0{,}75$	$(n^2 - n)/2$ $(n^2 - n) * 1{,}5$

Для усовершенствованных методов нет достаточно простых и точных формул. Все, что можно сказать, — это что стоимость вычислений равна $c_i \cdot n^{1.2}$ в случае сортировки Шелла и $c_i \cdot n \cdot \log n$ в случаях пирамидальной и быстрой сортировок.

Эти формулы дают лишь приблизительную оценку эффективности как функции от n; они допускают классификацию алгоритмов сортировки на простые (n^2) и усовершенствованные, или «логарифмические» $(n \cdot \log n)$. Однако для практических целей полезно иметь некоторые экспериментальные данные, которые могут пролить свет на коэффициенты c_i, позволяющие проводить дальнейшую оценку различных методов. Кроме того, в этих формулах не учитываются затраты на другие операции, отличные от сравнений ключей и пересылок элементов, такие, как управление циклами и т. д. Разумеется, эти факторы в какой-то степени зависят от конкретных систем, но тем не менее некоторый пример экспериментально полученных данных является информативным. В табл. 2.9 приведено время (в миллисекундах), которое затратила система Паскаль на вычислительной машине CDC 6400 на выполнение сортировки описанными здесь методами. В трех столбцах указано время, потребовавшееся для сортировки уже рассортированного массива, случайной перестановки и массива с обратным порядком элементов. Левое

Таблица 2.9. Время выполнения программ сортировки

	Упорядоченный массив		Случайный массив		Упорядоченный в обратном порядке массив	
Простое включение	12	23	366	1444	704	2836
Бинарное включение	56	125	373	1327	662	2490
Простой выбор	489	1907	509	1956	695	2675
Метод пузырька	540	2165	1026	4054	1492	5931
Метод пузырька с ограничением	5	8	1104	4270	1645	6542
Шейкер-сортировка	5	9	961	3642	1619	6520
Сортировка Шелла	58	116	127	349	157	492
Пирамидальная сортировка	116	253	110	241	104	226
Быстрая сортировка	31	69	60	146	37	79
Сортировка слиянием*)	99	234	102	242	99	232

*) См. разд. 2.3.1.

число в каждой колонке дано для массива из 256 элементов, а правое — для 512 элементов. Эти данные демонстрируют явное отличие методов n^2 от методов $n \cdot \log n$. Примечательны следующие моменты:

1. Преимущество сортировки бинарными включениями по сравнению с сортировкой простыми включениями действительно ничтожно, а в случае уже имеющегося порядка вообще отсутствует.
2. Сортировка методом пузырька определенно является наихудшей среди всех сравниваемых методов. Ее улучшенная версия — шейкер-сортировка все-таки хуже, чем сортировка простыми включениями и простым выбором (кроме патологического случая сортировки уже рассортированного массива).
3. Быстрая сортировка превосходит пирамидальную сортировку в отношении 2 к 3. Она сортирует массив с элементами, расположенными в обратном порядке практически так же, как уже рассортированный.

Следует добавить, что эти данные были получены при сортировке элементов, состоящих только из ключа без сопутствующей информации. Это — не слишком реалистичное допущение; в табл. 2.10 показано, как влияет увеличение размера элементов на скорость работы программ. В выбранном примере сопутствующие данные занимают в 7 раз больше памяти, чем ключ. Левое число в каждой колонке показывает время, нужное для сортировки записей без сопутствующих

Таблица 2.10. Время выполнения программ сортировки (Ключи с сопутствующей информацией)

	Упорядоченный массив		Случайный массив		Упорядоченный в обратном порядке массив	
Простые включения	12	46	366	1129	704	2150
Бинарные включения	56	76	373	1105	662	2070
Простой выбор	489	547	509	607	695	1430
Метод пузырька	540	610	1026	3212	492	5599
Метод пузырька с ограничением	5	5	1104	3237	1645	5762
Шейкер-сортировка	5	5	961	3071	1619	5757
Сортировка Шелла	58	186	127	373	157	435
Пирамидальная сортировка	116	264	110	246	104	227
Быстрая сортировка	31	55	60	137	37	75
Сортировка слиянием*	99	196	102	195	99	187

*) См. разд. 2.3.1.

данных, правое — отражает сортировку с сопутствующими данными; $n = 256$. Обратите внимание на следующие детали:

1. Сортировка простым выбором дает существенный выигрыш и оказывается лучшим из простых методов.
2. Сортировка методом пузырька по-прежнему является наихудшим методом (она еще больше сдала свои позиции!), и лишь ее «усовершенствование», называемое шейкер-сортировкой, еще чуть хуже в случае массива с обратным порядком.
3. Быстрая сортировка даже укрепила свою позицию в качестве самого быстрого метода и оказалась действительно лучшим алгоритмом сортировки.

2.3. СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНЫХ ФАЙЛОВ

2.3.1. Простое слияние

К сожалению, алгоритмы сортировки, рассмотренные в предыдущей главе, неприменимы, если сортируемые данные не помещаются в оперативной памяти, а, например, расположены на внешнем запоминающем устройстве с последовательным доступом, таком, как магнитная лента. В этом случае мы описываем данные как (последовательный) файл, который характеризуется тем, что в каждый момент имеется непосредственный доступ к одному и только одному элементу. Это — строгое ограничение по сравнению с возможностями, которые дает массив, и поэтому здесь приходится применять другие методы сортировки. Основной метод — это сортировка

слиянием. Слияние означает объединение двух (или более) упорядоченных последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент. Слияние — намного более простая операция, чем сортировка; она используется в качестве вспомогательной в более сложном процессе последовательной сортировки. Один из методов сортировки слиянием называется *простым слиянием* и состоит в следующем:

1. Последовательность a разбивается на две половины b и c.
2. Последовательности b и c сливаются при помощи объединения отдельных элементов в упорядоченные пары.
3. Полученной последовательности присваивается имя a, и повторяются шаги 1 и 2; на этот раз упорядоченные пары сливаются в упорядоченные четверки.
4. Предыдущие шаги повторяются: четверки сливаются в восьмерки, и весь процесс продолжается до тех пор, пока не будет упорядочена вся последовательность, ведь длины сливаемых последовательностей каждый раз удваиваются.

В качестве примера рассмотрим последовательность

44 55 12 42 94 18 06 67

На первом шаге разбиение дает последовательности

44 55 12 42
94 18 06 67

Слияние отдельных компонент (которые являются упорядоченными последовательностями длины 1) в упорядоченные пары дает

44 94 ′ 18 55 ′ 06 12 ′ 42 67

Новое разбиение пополам и слияние упорядоченных пар дают

06 12 44 94 ′ 18 42 55 67

Третье разбиение и слияние приводят, наконец, к нужному результату:

06 12 18 42 44 55 67 94

Операция, которая однократно обрабатывает все множество данных, называется *фазой*, а наименьший подпроцесс, который, повторяясь, образует процесс сортировки, называется *проходом* или *этапом*. В приведенном выше примере сортировка производится за три прохода, каждый проход состоит из фазы разбиения и фазы слияния. Для выполнения сортировки требуются три магнитные ленты, поэтому процесс называется *трехленточным слиянием*.

Собственно говоря, фазы разбиения не относятся к сортировке, поскольку они никак не переставляют элементы; в каком-то смысле они непродуктивны, хотя и составляют половину всех операций переписи. Их можно удалить, объединив фазы разбиения и слияния. Вместо того чтобы сливать элементы в одну последовательность, результат слияния сразу распределяют на две ленты, которые на следующем проходе будут входными. В отличие от двухфазного слияния этот метод называется *однофазным* или *сбалансированным слиянием*. Оно имеет явные преимущества, так как требует вдвое меньше операций переписи, но это достигается ценой использования четвертой ленты.

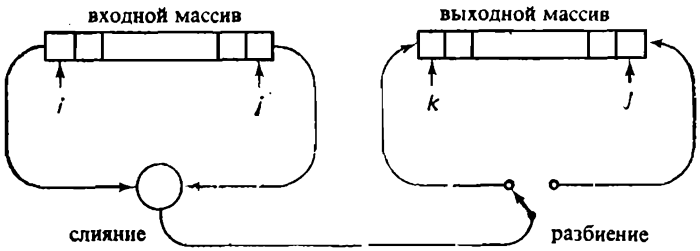


Рис. 2.12. Сортировка двух массивов методом простого слияния.

Разберем программу слияния подробно; предположим сначала, что данные расположены в виде массива, который, однако, можно просматривать только *строго последовательно*. Другая версия сортировки слиянием будет основана на файловой структуре, это позволит сравнить эти программы и показать строгую зависимость формы программы от представления ее данных.

Вместо двух файлов можно легко использовать один массив, если рассматривать его как последовательность с двумя концами. Вместо того чтобы сливать элементы из двух исходных файлов, мы можем брать их с двух концов массива. Таким образом, общий вид объединенной фазы слияния-разбиения можно изобразить, как показано на рис. 2.12. Направление пересылки сливаемых элементов меняется (переключается) после каждой упорядоченной пары на первом проходе, после каждой упорядоченной четверки на втором проходе и т. д.; таким образом равномерно заполняются две выходные последовательности, представленные двумя концами одного массива (выходного). После каждого прохода два массива меняются ролями: входной становится выходным и наоборот.

Программу можно еще больше упростить, объединив два концептуально различных массива в один массив двойной

длины. Итак, данные будут представлены следующим образом:

$$a: \textbf{array}[1..2*n]\ \textbf{of}\ item \tag{2.20}$$

Пусть индексы i и j указывают два исходных элемента, тогда как k и l обозначают два места пересылки (см. рис. 2.12). Исходные данные — это, разумеется, элементы $a_1, \ldots, a_n$. Очевидно, что нужна булевская переменная up для указания направления пересылки данных; $up = true$ будет означать, что на текущем проходе компоненты $a_1, \ldots, a_n$ будут пересылаться «вверх» — в переменные $a_{n+1}, \ldots, a_{2n}$, тогда как $up = false$ будет указывать, что $a_{n+1}, \ldots, a_{2n}$ должны переписываться «вниз» — в $a_1, \ldots, a_n$. Значение up строго чередуется между двумя последовательными проходами. И наконец, вводится переменная p для обозначения длины сливаемых подпоследовательностей (p-наборов). Ее начальное значение равно 1, и оно удваивается перед каждым очередным проходом. Для простоты мы будем считать, что n — всегда степень двойки. Итак, первая версия программы простого слияния имеет такой вид:

```
procedure mergesort;
   var i,j,k,l: index;
      up: Boolean; p: integer;
begin up := true; p := 1;
   repeat {инициация индексов}
      if up then
         begin i := 1; j := n; k := n+1; l := 2*n
         end else                                          (2.21)
         begin k := 1; l := n; i := n+1; j := 2*n
         end;
      «слияние p-наборов последовательностей i и j
       в последовательности k и l»;
      up := —up; p := 2*p
   until p = n
end
```

На следующем этапе мы уточняем действие, описанное на естественном языке (внутри кавычек). Ясно, что этот проход, обрабатывающий n элементов, состоит из последовательных слияний p-наборов. После каждого отдельного слияния направление пересылки переключается из нижнего в верхний конец выходного массива или наоборот, чтобы обеспечить одинаковое распределение в обоих направлениях. Если сливаемые элементы посылаются в нижний конец массива, то индексом пересылки служит k и k увеличивается на 1 после

каждой пересылки элемента. Если же они пересылаются в верхний конец массива, то индексом пересылки является l и l после каждой пересылки уменьшается на 1. Чтобы упростить операцию слияния, мы будем считать, что место пересылки всегда обозначается через k, и будем менять местами значения k и l после слияния каждого p-набора, а приращение индекса обозначим через h, где h равно либо 1, либо -1. Уточнив таким образом «конструкцию», мы получаем

```
h := 1; m := n; {m-номера сливаемых элементов}
repeat q := p; r := p; m := m-2*p;
    «слияние q элементов из i и r элементов из j,
    индекс засылки есть k с приращением h»;          (2.22)
  h := -h;
  обмен значениями k и l
until m = 0
```

На следующем этапе уточнения нужно сформулировать саму операцию слияния. Здесь следует учесть, что остаток подпоследовательности, которая остается непустой после слияния, добавляется к выходной последовательности при помощи простого копирования.

```
while (q≠0) ∧ (r≠0) do
begin {выбор элемента из i или j}
    if a[i] .key < a[j] .key then
        begin «пересылка элемента из i в k,
              увеличение i и k» ; q := q-1
        end else
        begin «пересылка элемента из j в k,          (2.23)
              увеличение j и k» ; r := r-1
        end
end;
«копирование остатка последовательности i»;
«копирование остатка последовательности j»
```

После уточнения операций копирования остатков программа будет ясна во всех деталях. Перед тем как записать ее полностью, мы хотим устранить ограничение, в соответствии с которым n должно быть степенью двойки. На какую часть алгоритма это повлияет? Легко убедиться в том, что в более общей ситуации лучше всего использовать прежний метод до тех пор, пока это возможно. В данном случае это означает, что мы продолжаем слияние p-наборов, пока длина остатков входных последовательностей не станет меньше p. Это влияет только на ту часть, где определяются значения

q и r — длины последовательностей, которые предстоит слить. Вместо трех операторов

$$q := p; \quad r := p; \quad m := m - 2 * p$$

используются следующие четыре оператора, и, как может убедиться читатель, здесь эффективно применяется описанная выше стратегия; заметим, что m обозначает общее число элементов в двух входных последовательностях, которые осталось слить:

if $m \geqslant p$ **then** $q := p$ **else** $q := m$; $\quad m := m - q$;
if $m \geqslant p$ **then** $r := p$ **else** $r := m$; $\quad m := m - r$;

И наконец, чтобы обеспечить окончание работы программы, нужно заменить условие $p = n$, управляющее внешним циклом, на $p \geqslant n$. После этих модификаций мы можем попытаться описать весь алгоритм в виде законченной программы (см. программу 2.13).

Анализ сортировки слиянием. Поскольку на каждом проходе p удваивается и сортировка заканчивается, как только $p \geqslant n$, она требует $\lceil \log_2 n \rceil$ проходов. По определению при каждом проходе все множество из n элементов копируется ровно один раз. Следовательно, общее число пересылок равно

$$M = n \cdot \lceil \log_2 n \rceil \qquad (2.24)$$

Число C сравнений по ключу еще меньше, чем M, так как при копировании остатка последовательности сравнения не производятся. Но, поскольку сортировка слиянием обычно применяется при работе с внешними запоминающими устройствами, стоимость операций пересылки часто на несколько порядков превышает стоимость сравнений. Поэтому подробный анализ числа сравнений не представляет особого практического интереса.

Алгоритм сортировки слиянием выдерживает сравнение даже с усовершенствованными методами сортировки, которые обсуждались в предыдущем разделе. Но затраты на управление индексами довольно высоки, кроме того, существенным недостатком является использование памяти размером $2n$ элементов *). Поэтому сортировка слиянием редко применяется при работе с массивами, т. е. данными, расположенными в оперативной памяти. Цифры, характеризующие быстродействие сортировки слиянием в режиме реального времени, содержатся в последних строках табл. 2.9 и 2.10. Эти показатели лучше, чем у пирамидальной сортировки, но хуже, чем у быстрой сортировки.

*) Напомним, что в итеративном варианте алгоритма быстрой сортировки требуется стек размера n, а в рекурсивном варианте затраты памяти значительно больше. — *Прим. ред.*

```
procedure mergesort;
    var i,j,k,l,t: index;
        h,m,p,q,r: integer; up: boolean;
    {a имеет индексы 1, . . . , 2*n}
begin up := true; p := 1;
    repeat h := 1; m := n;
        if up then
        begin i := 1; j := n; k := n+1; l := 2*n
        end else
        begin k := 1; l := n; i := n+1; j := 2*n
        end ;
        repeat {слияние серий из i и j в k}
            {q — длина серии из i, r — длина серии из j}
            if m ≥ p then q := p else q := m; m := m−q;
            if m ≥ p then r := p else r := m; m := m−r;
            while (q≠0) ∧ (r≠0) do
            begin {слияние}
                if a[i] .key < a[j] .key then
                begin a[k] := a[i]; k := k+h; i := i+1; q := q−1
                end else
                begin a[k] := a[j]; k := k+h; j := j−1; r := r−1
                end
            end ;
            {копирование остатка серии из j}
            while r ≠ 0 do
                begin a[k] := a[j]; k := k+h; j := j−1; r := r−1
                end ;
            {копирование остатка серии из i}
            while q ≠ 0 do
                begin a[k] := a[i]; k := k+h; i := i+1; q := q−1
                end ;
            h := −h; t := k; k := l; l := t
        until m = 0;
        up := ¬up; p := 2*p
    until p ≥ n;
    if ¬up then
    for i := 1 to n do a[i] := a[i+n]
end {mergesort}
```

Программма 2.13. Сортировка простым слиянием.

2.3.2. Естественное слияние

В случае простого слияния мы ничего не выигрываем, если данные уже частично рассортированы. На k-м проходе длина всех сливаемых подпоследовательностей меньше или равна 2^k без учета того, что более длинные подпоследовательности уже могут быть упорядочены и их можно было бы сливать. Фактически можно было бы сразу сливать какие-либо упорядоченные подпоследовательности длиной m и n в одну последовательность из $m+n$ элементов. Метод сортировки, при котором каждый раз сливаются две самые длинные возможные подпоследовательности, называется *естественным слиянием*.

Упорядоченную подпоследовательность часто называют *цепочкой*. Но, поскольку слово «цепочка» чаще используется для обозначения последовательности символов, мы будем использовать слово *серия*, когда речь идет об упорядоченной подпоследовательности. Мы называем подпоследовательность $a_i, \ldots, a_j$, такую, что

$$\begin{aligned} &a_k \leqslant a_{k+1} \quad \text{для} \quad k = i, \ldots, j-1, \\ &a_{i-1} > a_i, \\ &a_j > a_{j+1}, \end{aligned} \tag{2.25}$$

максимальной серией или, короче, серией. Итак, сортировка естественным слиянием сливает не последовательности фиксированной, заранее заданной длины, а (максимальные) серии. Серии имеют то свойство, что при слиянии двух последовательностей, каждая из которых содержит n серий, возникает одна последовательность, содержащая ровно n серий. Таким образом, на каждом проходе общее число серий уменьшается вдвое, и число необходимых пересылок элементов в худшем случае равно $n \cdot [\log_2 n]$, а в обычном случае даже меньше. Ожидаемое число сравнений, однако, намного больше, так как кроме сравнений, необходимых для упорядочения элементов, требуются еще сравнения соседних элементов каждого файла для определения концов серии.

Следующим нашим упражнением будет разработка алгоритма естественного слияния тем же поэтапным методом, который использовался при объяснении алгоритма простого слияния. Вместо массива он обрабатывает последовательный файл и представляет собой несбалансированную двухфазную трехленточную сортировку слиянием. Пусть исходная последовательность элементов задана в виде файла c, который в конце работы должен содержать результат сортировки. (Разумеется, в реальных процессах обработки исходные данные для сохранности сначала переписываются с ленты в рабочий файл c.) Используются две вспомогательные ленты a и b. Каждый проход состоит из фазы распределения, которая распределяет серии поровну из c в a и b, и фазы слияния.

которая сливает серии из a и b в c. Этот процесс показан на рис. 2.13.

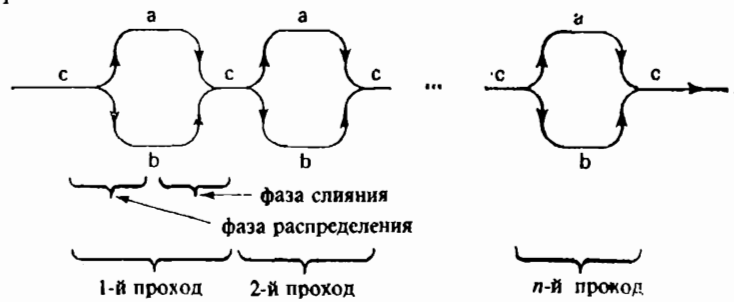


Рис. 2.13. Фазы сортировки и проходы сортировки.

В качестве примера в табл. 2.11 показан файл c в исходном состоянии (строка 1) и после каждого прохода (строки

Таблица 2.11. Пример сортировки естественным слиянием

17	31′	5	59′	13	41	43	67′	11	23	29	47′	3	7	71′	2	19	57′	37	61
5	17	31	59′	11	13	23	29	41	43	47	67′	2	3	7	19	57	71′	37	61
5	11	13	17	23	29	31	41	43	47	59	67′	2	3	7	19	37	57	61	71
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

2—4). В естественном слиянии участвуют 20 чисел. Заметим, что требуются только три прохода. Сортировка заканчивается, как только число серий в c будет равно 1. (Предполагается, что в исходном файле имеется хотя бы одна непустая серия.) Итак, пусть переменная l используется для подсчета числа серий, сливаемых в c. Если мы определим глобальные объекты

```
type tape = file of item;
var c: tape
```
(2.26)

то программу можно написать следующим образом:

```
procedure naturalmerge;
    var l: integer;
        a,b: tape;
begin
    repeat rewrite(a); rewrite(b); reset(c);
        distribute;
        reset(a); reset(b); rewrite(c);
        l := 0; merge
    until l = 1
end
```
(2.27)

Ясно, что две фазы выражаются двумя отдельными операторами. Теперь их надо уточнить, т. е. описать более подробно. Подробные описания можно либо непосредственно вставить в текст, либо представить в виде процедур, и тогда сокращенно записанные операторы следует рассматривать как вызовы процедур. На этот раз мы изберем последний способ и определим

```
procedure distribute; { из c в a и b}
begin
    repeat copyrun(c,a);
        if ¬eof(c) then copyrun(c,b)                    (2.28)
    until eof(c)
end
```

и

```
procedure merge;
begin  {из a и b в c}
    repeat mergerun; l := l+1
    until eof(b);                                       (2.29)
    if ¬eof(a) then
        begin copyrun(a,c); l := l+1
        end
end
```

Предполагается, что при таком способе распределения в файлах a и b оказывается либо равное число серий, либо файл a содержит на одну серию больше, чем b. Поскольку соответствующие пары серий сливаются, в файле a может оказаться лишняя серия, которую следует просто переписать. Процедуры *merge* и *distribute* формулируются с помощью подчиненных процедур *mergerun* и *copyrun*, задачи которых понятны. Теперь опишем эти процедуры более подробно; они требуют введения глобальной булевской переменной *eor* (end of the run), значение которой показывает, достигнут ли конец серии.

```
procedure copyrun(var x,y: tape);
begin {перепись одной серии из x в y}                   (2.30)
    repeat copy(x,y) until eor
end
```

```
procedure mergerun;
begin {слияние серий из a и b в c}
    repeat if a↑.key < b↑.key then
        begin copy(a,c);
            if eor then copyrun(b,c)
        end else                                    (2.31)
        begin copy(b,c);
            if eor then copyrun(a,c)
        end
    until eor
end
```

Процесс сравнения и выбора по ключу при слиянии серий завершается, как только будет исчерпана одна из двух серий. После этого остаток другой серии, который еще не исчерпан, нужно переслать в выходную серию с помощью простого копирования. Это осуществляется вызовом процедуры *copyrun*.

Обе эти процедуры определены с помощью подчиненной процедуры *copy*, которая пересылает элемент из файла x в файл y и определяет, достигнут ли конец серии. Ее легко написать, используя операторы *read* и *write*. Для того чтобы найти конец серии, нужно сохранять ключ последнего прочитанного (переписанного) элемента для сравнения со следующим. Это «заглядывание вперед» достигается использованием буферной переменной файла $x\uparrow$.

```
procedure copy(var x,y: tape);
    var buf: item;
begin read(x, buf); write(y, buf);                  (2.32)
    if eof(x) then eor := true else eor := buf.key > x↑.key
end
```

На этом построение процедуры сортировки естественным слиянием закончено. К сожалению, как может заметить внимательный читатель, эта программа некорректна, поскольку в некоторых случаях она неправильно производит сортировку. Рассмотрим, например, такую последовательность входных данных:

3 2 5 11 7 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67

Распределяя последовательные серии поочередно в файлы a и b, мы получим

$a = 3\ '\ 7\ 13\ 19\ '\ 29\ 37\ 43\ '\ 57\ 61\ 71\ '$

$b = 2\ 5\ 11\ '\ 17\ 23\ 31\ '\ 41\ 47\ 59\ '\ 67$

Эти последовательности легко сливаются в одну серию, после чего сортировка заканчивается. Хотя этот пример и не при-

водит к ошибочному поведению программы, он показывает, что простое распределение серий в несколько файлов может дать в результате меньшее число выходных серий, чем входных. Это происходит потому, что первый элемент $(i+2)$-й серии может быть больше, чем последний элемент i-й серии, что приведет к автоматическому слиянию двух серий в одну.

Хотя и предполагается, что процедура *distribute* посылает серии поровну в оба файла, действительные количества выходных серий в a и b могут значительно различаться. Однако наша процедура будет только сливать пары серий и заканчиваться, как только будет прочитан файл b, теряя при этом остаток одного из файлов. Рассмотрим такие исходные данные, которые сортируются (и усекаются) за два последовательных прохода:

Таблица 2.12. Неправильный результат работы программы сортировки слиянием

17	**19**	**13**	**57**	**23**	**29**	**11**	**59**	**31**	**37**	**7**	**61**	**41**	**43**	**5**	**67**	**47**	**71**	**2**	**3**
13	**17**	**19**	**23**	**29**	**31**	**37**	**41**	**43**	**47**	**57**	**71**	**11**	**59**						
11	**13**	**17**	**19**	**23**	**29**	**31**	**37**	**41**	**43**	**47**	**57**	**59**	**71**						

Эта ошибка типична для многих ситуаций. Она вызвана тем, что упускается из виду одно из возможных последствий, казалось бы, простой операции. Она также типична в том смысле, что существует несколько способов ее исправления и нужно выбрать один из них. Часто имеются две возможности, различие между которыми носит принципиальный характер:

1. Мы видим, что операция распределения написана некорректно и не удовлетворяет требованию, чтобы число серий на двух лентах было одинаковым (или различалось не более чем на 1). Придерживаемся принятой ранее схемы и соответствующим образом исправляем неправильную процедуру.
2. Мы видим, что исправление неправильно написанной части требует серьезных модификаций, и ищем способы изменить другие части алгоритма, чтобы повлиять на работу некорректной части.

Вообще говоря, первый способ выглядит более понятным и надежным, а также более честным; он в достаточной мере свободен от непредусмотренных последствий и сложных побочных эффектов. Следовательно, это тот путь, который обычно рекомендуется.

Однако надо указать, что бывают случаи, когда не следует пренебрегать второй возможностью. Поэтому ниже мы покажем на этом примере, как можно исправить ошибку,

```
program mergesort (input, output);
{3-ленточная, 2-фазная сортировка естественным слиянием}
type item = record key: integer
                   {прочие поля}
            end ;
   tape = file of item;
var c: tape; n: integer; buf: item;
procedure list (var f: tape);
   var x: item;
begin reset(f);
   while ¬eof(f) do
      begin read(f,x); write(output, x.key)
      end ;
   writeln
end {list} ;
procedure naturalmerge;
   var l: integer;  {число сливаемых серий}
      eor: boolean; {индикатор конца серии}
      a,b: tape;
   procedure copy(var x,y: tape);
      var buf: item;
   begin read(x, buf); write(y,buf);
      if eof(x) then eor := true else eor := buf.key ≥ x↑.key
   end ;
   procedure copyrun (var x,y: tape);
   begin {перепись одной серии из x в y}
      repeat copy(x,y) until eor
   end ;
   procedure distribute;
   begin {из c в a и b}
      repeat copyrun (c,a);
         if ¬eof(c) then copyrun (c,b)
      until eof(c)
   end ;
   procedure mergerun;
   begin {из a и b в c}
      repeat
         if a↑.key ≤ b↑.key then
         begin copy (a,c);
            if eor then copyrun (b,c)
         end else
         begin copy (b,c);
```

```
                if cor then copyrun (a,c)
            end
        until eor
    end ;
    procedure merge;
    begin {из a и b в c}
        while ¬eof(a) ∧ ¬eof(b) do
        begin mergerun; l := l+1
        end;
        while ¬eof(a) do
        begin copyrun (a,c); l := l+1
        end;
        while ¬eof(b) do
        begin copyrun (b,c); l := l+1
        end ;
        list (c)
    end ;
begin
    repeat rewrite(a); rewrite(b); reset(c);
        distribute;
        reset(a); reset(b); rewrite(c);
        l := 0; merge;
    until l = 1
end ;
begin {основная программа; чтение входной
      последовательности с 0 в конце}
    rewrite(c); read(buf.key);
    repeat write(c.buf); read(buf.key)
    until buf .key = 0:
    list (c);
    naturalmerge
    list(c)
end .
```

Программа 2.14. Сортировка естественным слиянием.

изменив процедуру слияния, а не процедуру распределения, которая изначально является ошибочной.

Это значит, что схему распределения мы оставляем нетронутой, а отказываемся от требования, чтобы серии распределялись поровну на две ленты. В результате программа может работать неоптимальным образом. Однако в худшем варианте она сохраняет те же характеристики; кроме того,

случай существенно неравномерного распределения статистически крайне *маловероятен*. Поэтому соображения эффективности не являются серьезным аргументом против этого решения.

Если требование о распределении серий поровну отменено, то процедуру слияния следует изменить таким образом, чтобы после достижения конца одного из файлов копировался *весь* остаток другого файла, а не только одна серия.

Это — несложное изменение, оно намного проще, чем какое-либо изменение схемы распределения. (Мы предлагаем читателю самому убедиться в правоте этого утверждения.) Пересмотренная версия алгоритма слияния включена в окончательную программу 2.14.

2.3.3. Сбалансированное многопутевое слияние

Затраты на последовательную сортировку пропорциональны числу проходов, так как по определению на каждом проходе происходит перепись всего множества данных. Один из способов уменьшить это число — распределять серии на более чем две ленты. Слияние r серий, которые поровну распределены на N лентах, дает в результате последовательность из r/N серий. Второй проход уменьшает это число до r/N^2, а третий — до r/N^3, и после k проходов остается r/N^k отрезков. Итак, общее число проходов при сортировке n элементов *N-путевым слиянием* $k = [\log_N n]$. Поскольку на каждом проходе производится n операций переписи, общее число операций переписи в худшем случае будет

$$M = n \cdot [\log_N n].$$

В качестве следующего упражнения мы построим программу сортировки, основанной на многопутевом слиянии. Чтобы подчеркнуть различие между этой программой и описанной выше процедурой естественного двухфазного слияния, мы определим многопутевое слияние как однофазную сбалансированную сортировку слиянием. Это означает, что на каждом проходе имеется равное число входных и выходных файлов, в которые поочередно распределяются следующие одна за другой серии. При использовании N файлов алгоритм будет основан на $N/2$-путевом слиянии, если считать, что N четно. Следуя принятой ранее стратегии, мы не будем заботиться о том, чтобы предотвратить автоматическое слияние двух соседних серий, попавших на одну ленту. Следовательно, нам приходится разрабатывать программу слияния без требования, чтобы на входных лентах содержалось строго одинаковое число серий.

В этой программе мы впервые встречаем естественное использование структуры данных, представляющей собой массив файлов. В самом деле, удивительно, насколько велико отличие этой программы от предыдущей в результате перехода от двухпутевого к многопутевому слиянию. Это происходит прежде всего потому, что теперь процесс слияния не может просто завершаться после того, как будет исчерпан один из входных файлов. Вместо этого нужно вести список входных файлов, которые пока активны, т. е. не исчерпаны. Другое усложнение возникает из-за необходимости переключать группы входных и выходных лент после каждого прохода.

Вначале, в дополнение к двум знакомым нам типам *item* и *tape*, определим тип номера ленты:

$$tapeno = 1..N \tag{2.33}$$

Очевидно, что номера лент нужны, для того чтобы индексировать массив файлов. Будем считать, что исходная последовательность элементов задана переменной

$$f0: tape \tag{2.34}$$

и для сортировки мы имеем в распоряжении N лент, где N четно:

$$f: \textbf{array}[tapeno]\ \textbf{of}\ tape \tag{2.35}$$

Для решения проблемы переключения лент рекомендуется использовать карту ленточных индексов. Вместо того чтобы обращаться к ленте непосредственно с помощью индекса i, к ней адресуются через карту t, т. е. вместо

$$f[i] \text{ мы пишем } f[t[i]],$$

где карта определена как

$$t: \textbf{array}[tapeno]\ \textbf{of}\ tapeno \tag{2.36}$$

Если первоначально $t[i] = i$ для любого i, то переключение производится просто при помощи обмена пар компонент карты

$$\begin{aligned} t[1] &\leftrightarrow t[nh+1] \\ t[2] &\leftrightarrow t[nh+2] \\ &\cdots \\ t[nh] &\leftrightarrow t[n], \end{aligned}$$

где $nh = n/2$. Следовательно, мы всегда можем считать

$$f[t[1]], \ldots, f[t[nh]]$$

входными лентами, а

$$f[t[nh+1]], \ldots, f[t[n]]$$

выходными лентами. (В дальнейшем мы будем называть $f[t[j]]$ просто «лентой j».) Теперь в первом приближении алгоритм можно записать следующим образом:

```
procedure tapemergesort;
    var i,j: tapeno;
        l: integer; {число распределяемых серий}
        t: array [tapeno] of tapeno;
begin {распределение начальных серий на t [1] . . . t [nh]}
    j := nh; l := 0;
    repeat if j < nh then j := j+1 else j := 1;
           «перепись одного отрезка с f 0 на ленту j»;
           l := l+1
    until eof(f0);
    for i := 1 to n do t[i] := i;
    repeat {слияние из t [1] . . . t [nh] в t [nh + 1] . . . t [n]}      (2.37)
           «установка входных лент»;
           l := 0;
           j := nh+1; {j-индекс выходной ленты}
           repeat l := l+1;
                  «слияние серии а с входных лент на t [j]»
              if j < n then j := j+1 else j := nh+1
           until «все входные ленты исчерпаны»;
           «переключение ленты»
    until l = 1;
    {отсортированный файл находится на ленте t [1]}
end
```

Прежде всего уточним операцию копирования, которая используется при начальном распределении серий; вновь введем вспомогательную переменную для буферизации последнего считанного элемента

$$buf: item$$

и заменим «перепись одного отрезка с $f0$ на ленту j» оператором

```
repeat read(f0, buf);
    write(f[j], buf)                                                    (2.38)
until (buf.key > f0↑.key) ∨ eof(f0)
```

Перепись серии заканчивается, либо когда встречается первый элемент следующей серии ($buf.key > f0\uparrow.key$), либо когда достигается конец всего входного файла ($eof(f0)$).

Теперь в алгоритме сортировки остались операторы

1) установка входных лент;
2) слияние серии с входных лент на ленту $t[j]$;
3) переключение ленты

и предикат

4) все входные ленты исчерпаны,

которые нужно определить более подробно. Во-первых, мы должны аккуратно вести учет текущих входных файлов. Заметим, что количество «активных» входных файлов может быть меньше чем $N/2$. Действительно, максимальное число входных файлов может быть равно числу серий; сортировка заканчивается в том случае, когда остается только один файл. Причем может оказаться, что количество серий в начале последнего прохода сортировки меньше чем nh. Поэтому мы вводим переменную $k1$ для обозначения числа текущих входных файлов. Инициацию $k1$ мы включим в оператор (1) следующим образом:

```
if l < nh then k1 := l else k1 := nh;
for i := 1 to k1 do reset(f[t[i]]);
```

Понятно, что оператор (2) должен уменьшать значение $k1$, как только исчерпывается какой-либо входной файл. Следовательно, предикат (4) легко можно выразить отношением

$$k1 = 0$$

Оператор (2) уточнить труднее; он содержит циклический выбор наименьшего ключа из текущих входных данных; выбранный таким образом элемент, посылается в выходной файл, т. е. на текущую выходную ленту. Этот процесс осложняется еще тем, что нужно искать конец каждой серии. Конец серии считается достигнутым, если (1) очередной ключ меньше текущего, или (2) достигнут конец входного файла. В последнем случае лента исключается из работы путем уменьшения $k1$, в первом случае отрезок закрывается, т. е. файл перестает участвовать в дальнейшем выборе элементов, но только до тех пор, пока не окончится формирование текущей выходной серии. Отсюда ясно, что необходима вторая переменная $k2$ для обозначения числа входных лент, которые в текущий момент используются для выбора следующего элемента. Это значение вначале полагается равным k_1 и уменьшается, когда какая-либо серия закрывается по условию (1).

К сожалению, недостаточно ввести переменную $k2$: мало знать число лент — нужно знать точно, какие именно ленты

```
program balancedmerge (output);
{сбалансированная n-путевая сортировка слиянием}
const n = 6; nh = 3;          {число лент}
type item = record
                key: integer
            end ;
   tape = file of item;
   tapeno = 1 . . n;
var leng, rand: integer;      {используются для формирования файла}
   eot: boolean;              {конец ленты}
   buf: item;
   f0: tape; {f0 — входная лента со случайными числами}
   f: array [1 . . n] of tape;
procedure list(var f: tape; n: tapeno);
   var z: integer;
begin writeln('TAPE', n:2); z := 0;
   while ¬eof(f) do
   begin read(f, buf); write(output, buf.key: 5); z := z+1;
      if z = 25 then
         begin writeln(output); z := 0;
         end
   end ;
   if z ≠ 0 then writeln (output); reset(f)
end {list} ;
procedure tapemergesort;
   var i,j,mx,tx: tapeno;
      k1,k2,l; integer;
      x, min: integer;
      t, ta: array [tapeno] of tapeno;
begin {распределение начальных серий на t[1] . . . t[nh]}
   for i := 1 to nh do rewrite(f[i]);
   j := nh; l := 0;
   repeat if j < nh then j := j+1 else j := 1;
      {перепись одной серии с f0 на ленту j}
      l := l+1;
      repeat read(f0, buf); write(f[j], buf)
      until (buf .key > f0↑ .key) ∨ eof(f0)
   until eof(f0);
   for i := 1 to n do t[i] := i;
   repeat {слияние с t[1] . . . t[nh] на t[nh + 1] . . . t[n]}
      if l < nh then k1 := l else k1 := nh,
      {k1-число входных лент на этой фазе}
      for i := 1 to k1 do
         begin reset(f[t[i]]); list(f[t[i]], t[i]); ta[i] := t[i]
         end ;
```

```
      l := 0;  {l-число сливаемых серий}
      j := nh+1;     {j-индекс выходной ленты}
      repeat {слияние серий с t[1] . . . t[k1] на t[j]}
         k2 := k1; l := l+1;  {k2-число входных лент,
                                участвующих в слиянии}
         repeat {выбор наименьшего элемента}
            i := 1; mx := 1; min := f[ta[1]]↑.key;
            while i < k2 do
            begin i := i+1; x := f[ta[i]]↑.key;
                  if x < min then
                  begin min := x; mx := i
                  end
            end ;
            {наименьший элемент на ta[mx], пересылка его на t[j]}
            read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
            write(f[t[j]], buf);
            if eot then
            begin rewrite(f[ta[mx]]); {исключение ленты}
               ta[mx] := ta[k2]; ta[k2] := ta[k1];
               k1 := k1-1; k2 := k2-1
            end else
            if buf.key > f[ta[mx]]↑.key then
            begin tx := ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx;
               k2 := k2-1
            end
         until k2 = 0;
         if j < n then j := j+1 else j := nh+1
      until k1 = 0;
      for i := 1 to nh do
         begin tx := t[i]; t[i] := t[i+nh]; t[i+nh] := tx
         end
   until l = 1;
   reset(f[t[1]]); list(f[t[1]], t[1]);  {отсортированный файл
                                          находится на t[1]}
end {tapemergesort} ;

begin {формирование случайного файла f0}
   leng := 200; rand := 7789; rewrite(f0);
   repeat rand := (131071*rand) mod 2147483647;
      buf.key := rand div 2147484; write(f0, buf); leng := leng - 1
   until leng = 0;
   reset(f0); list(f0, 1);
   tapemergesort
end .
```

Программа 2.15. Сортировка сбалансированным слиянием.

еще участвуют в игре. Очевидный прием — использовать массив булевских переменных, отражающих активность лент. Однако мы предпочитаем другой метод, при котором более эффективно работает процедура выбора, являющаяся в конечном счете наиболее часто повторяющейся частью алгоритма. Вместо булевского массива вводится вторая карта лент *ta*. Эта карта используется вместо *t* так, что *ta*[1] ... ... *ta*[*k*2] — индексы лент, участвующих в работе. Итак, оператор (2) можно записать следующим образом:

```
k2 := k1;
repeat «выбор минимального ключа, пусть
        ta[mx] — номер его ленты»;
       read(f[ta[mx]], buf);
       write(f[t[j]], buf);                                    (2.39)
       if eof(f[ta[mx]]) then «исключение ленты» else
       if buf.key > f[ta[mx]]↑.key then «закрыть серию»
until k2 = 0
```

Поскольку количество устройств-носителей магнитных лент, доступных в вычислительной системе, обычно довольно мало, алгоритм выбора, который нужно уточнить на следующем этапе, может также представлять собой простой линейный поиск. Оператор «исключение ленты» предполагает уменьшение *k*1 и *k*2, а также изменение значений индексов в карте *ta*. Оператор «закрыть серию» просто уменьшает *k*2 и должным образом переупорядочивает компоненты *ta*. Подробно это показано в программе 2.15, которая является окончательным уточнением (2.37) при помощи (2.39). Заметим, что ленты освобождаются процедурой *rewrite*, как только прочитана последняя серия. Оператор «переключение ленты» разработан в соответствии с данными ранее пояснениями.

2.3.4. Многофазная сортировка

Теперь мы знаем необходимые приемы и подготовлены к тому, чтобы разработать и запрограммировать другой алгоритм сортировки, работающий более эффективно, чем алгоритмы сбалансированной сортировки. Мы видели, что при сбалансированном слиянии устраняются операции простого копирования, поскольку распределение и слияние объединены в одну фазу. Возникает вопрос: можно ли еще лучше использовать имеющиеся ленты? Да, это действительно возможно; очередное усовершенствование заключается в том, чтобы отказаться от строгого понятия прохода, т. е. использовать ленты более хитрым способом, чем тот, когда считают, что всегда имеется $N/2$ входных лент и столько же выход-

ных, и меняют ролями входные и выходные ленты после каждого отдельного прохода. При этом понятие прохода становится нечетким. Этот метод был изобретен Р. Л. Гилстадом [2.3] и назван *многофазной сортировкой*.

Вначале проиллюстрируем его на примере работы с тремя лентами. В каждый момент элементы сливаются с двух лент на третью. Как только одна из входных лент окажется исчерпанной, она сразу становится выходной лентой для слияния с той ленты, которая еще не исчерпана, и с той, которая до этого была выходной.

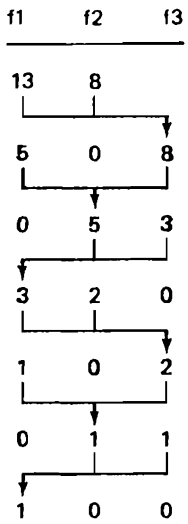


Рис. 2.14. Многофазная сортировка слиянием с тремя лентами, содержащими 21 серию.

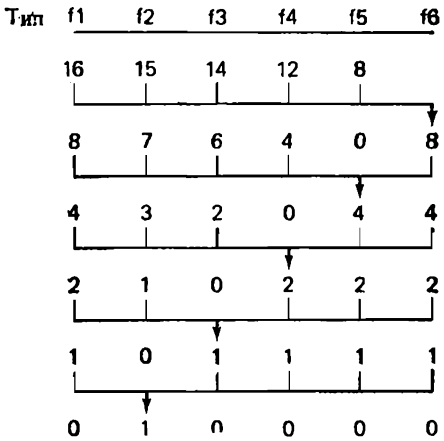


Рис. 2.15. Многофазная сортировка слиянием с шестью лентами, содержащими 65 серий.

Поскольку мы знаем, что n серий на каждой входной ленте превращаются в n серий на выходной ленте, нам нужно только вести список числа серий на каждой ленте (вместо того, чтобы определять действительные ключи). На рис. 2.14 предполагается, что вначале две входные ленты $f1$ и $f2$ содержат соответственно 13 и 8 серий. Таким образом, на первом «проходе» 8 серий сливаются с $f1$ и $f2$ на $f3$, на втором «проходе» оставшиеся 5 серий сливаются с $f3$ и $f1$ на $f2$ и т. д. В конце работы на ленте $f1$ содержится отсортированный файл.

Второй пример демонстрирует многофазный метод с 6 лентами. Пусть вначале имеются 16 серий на $f1$, 15 — на $f2$, 14 — на $f3$, 12 — на $f4$ и 8 — на $f5$; на первом частичном

проходе 8 серий сливаются на $f6$; в конце работы на ленте $f2$ содержится отсортированное множество элементов (см. рис. 2.15).

Многофазная сортировка более эффективна, чем сбалансированная сортировка, поскольку, если даны N лент, она всегда имеет дело с $(N-1)$-путевым слиянием вместо $N/2$-путевого слияния. Поскольку число требующихся проходов приблизительно равно $\log_N n$, где n — число сортируемых элементов, а N — число входных лент для слияния, многофазный метод обещает дать значительное улучшение по сравнению со сбалансированным слиянием.

Разумеется, в приведенных примерах было тщательно подобрано распределение начальных серий. Для того чтобы узнать, какие исходные распределения серий требуются для правильной работы алгоритма, мы пойдем обратным путем, начиная с окончательного распределения (последняя строка на рис. 2.15). Переписывая таблицы для этих двух примеров и поворачивая каждый ряд на одну позицию по отношению к предыдущему ряду, мы получаем табл. 2.13 и 2.14 для шести проходов и для трех и шести лент соответственно.

Таблица 2.13. Идеальное распределение серий на двух лентах

l	$a_1^{(l)}$	$a_2^{(l)}$	$\sum a_i^{(l)}$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Таблица 2.14. Идеальное распределение серий на пяти лентах

l	$a_1^{(l)}$	$a_2^{(l)}$	$a_3^{(l)}$	$a_4^{(l)}$	$a_5^{(l)}$	$\sum a_i^{(l)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Из табл. 2.13 можно вывести соотношения

$$\left.\begin{aligned} a_2^{(l+1)} &= a_1^{(l)}, \\ a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} \end{aligned}\right\} \quad \text{для } l > 0 \qquad (2.40)$$

и $a_1^{(0)} = 1$, $a_2^{(0)} = 0$. Полагая $a_1^l = f_l$, мы получаем

$$\begin{aligned} f_{l+1} &= f_l + f_{l-1}, \quad \text{для} \quad l \geqslant 1, \\ f_1 &= 1, \\ f_0 &= 0. \end{aligned} \tag{2.41}$$

Это — рекурсивные правила (или рекуррентные соотношения), определяющие так называемые *числа Фибоначчи*:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots .$$

Каждое число Фибоначчи представляет собой сумму двух предшествующих чисел. Итак, для того, чтобы многофазный метод с тремя лентами работал правильно, числа начальных серий на двух входных лентах должны быть двумя соседними в ряду Фибоначчи. А что можно сказать относительно второго примера (табл. 2.14) с шестью лентами? Правила построения чисел легко записать в таком виде:

$$\begin{aligned} a_5^{(l+1)} &= a_1^{(l)}, \\ a_4^{(l+1)} &= a_1^{(l)} + a_5^{(l)} = a_1^{(l)} + a_1^{(l-1)}, \\ a_3^{(l+1)} &= a_1^{(l)} + a_4^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)}, \\ a_2^{(l+1)} &= a_1^{(l)} + a_3^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)}, \\ a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} + a_1^{(l-4)}. \end{aligned} \tag{2.42}$$

Подстановка f_l вместо a_1^l дает

$$\begin{aligned} f_{l+1} &= f_l + f_{l-1} + f_{l-2} + f_{l-3} + f_{l-4} \quad \text{для} \quad l \geqslant 4, \\ f_4 &= 1, \\ f_l &= 0 \quad \text{для} \quad l < 4. \end{aligned} \tag{2.43}$$

Эти числа являются так называемыми числами Фибоначчи порядка 4. В общем виде *числа Фибоначчи порядка p* определяются следующим образом:

$$\begin{aligned} f_{l+1}^{(p)} &= f_l^{(p)} + f_{l-1}^{(p)} + \ldots + f_{l-p}^{(p)} \quad \text{для} \quad l \geqslant p, \\ f_p^{(p)} &= 1, \\ f_l^{(p)} &= 0 \quad \text{для} \quad 0 \leqslant l < p. \end{aligned} \tag{2.44}$$

Заметим, что обычные числа Фибоначчи имеют порядок 1.

Теперь мы убедились, что исходные числа серий для идеальной многофазной сортировки с n лентами должны быть суммами $n-1$, $n-2$, ..., 1 (см. табл. 2.15) последовательных чисел Фибоначчи порядка $n-2$. Из этого следует, что алгоритм многофазного слияния применим только к таким входным данным, в которых число серий есть сумма

Таблица 2.15. Количество серий, при которых возможно идеальное распределение

l \ n	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001
15	1597	13745	34513	56417	76806	95617
16	2584	25281	66526	110913	152351	190465
17	4181	46499	128233	218049	302201	379399
18	6765	85525	247177	428673	599441	755749
19	10946	157305	476449	842749	1189041	1505425
20	17711	289329	918385	1656801	2358561	2998753

$n-1$ таких сумм Фибоначчи. Итак, возникает важный вопрос: что делать, если число начальных серий не является такой идеальной суммой? Ответ прост (и типичен для подобных ситуаций): мы предполагаем существование гипотетических пустых серий, таких, что сумма реальных и гипотетических серий дает идеальную сумму. Пустые серии называются *фиктивными сериями*. Но этот ответ на самом деле неудовлетворителен, так как сразу вызывает следующий, более трудный вопрос: как распознаются фиктивные серии при слиянии? Перед тем как ответить на него, мы вернемся к упомянутой выше проблеме распределения действительных и фиктивных серий на $n-1$ лентах.

Однако, для того чтобы найти подходящее правило распределения, мы должны знать, как сливаются настоящие и фиктивные серии. Ясно, что выбор фиктивной серии с i-й ленты означает, что i-я лента не участвует в слиянии; в результате слияние происходит с менее чем $n-1$ лент. Слияние фиктивных серий со всех $n-1$ входных лент не предполагает никакой действительной операции слияния, а означает просто запись фиктивной серии на выходную ленту. Из этого можно

сделать вывод, что фиктивные серии нужно распределять на $n-1$ лент как можно более равномерно, так как мы заинтересованы в активном слиянии с наибольшего возможного числа входных лент.

Забудем на некоторое время о фиктивных сериях и рассмотрим задачу распределения *неизвестного* числа серий на $n-1$ лент. Ясно, что в процессе распределения можно получать числа Фибоначчи порядка $n-2$, определяющие желательные количества серий на каждой ленте. Предположив, например, что $n=6$, и ссылаясь на табл. 2.14, мы начинаем с распределения серий, указанных в строке с номером $l=1$ (1, 1, 1, 1, 1); если имеются еще серии, мы переходим ко второй строке (2, 2, 2, 2, 1); если входные данные еще не исчерпаны, распределение производится в соответствии со следующей строкой (4, 4, 4, 3, 2) и т. д. Номер строки мы будем называть *уровнем*. Очевидно, что чем больше число серий, тем выше будет уровень чисел Фибоначчи, который в данном случае равен количеству проходов, или переключений лент, необходимых для последующей сортировки.

Алгоритм распределения теперь, в первом приближении, можно сформулировать следующим образом:

1. Пусть перед нами стоит цель — числа Фибоначчи порядка $n-2$, уровня 1.
2. Распределяем серии согласно поставленной цели.
3. Если цель достигнута, вычисляем следующий уровень чисел Фибоначчи; разность между числами этого уровня и числами предыдущего уровня представляет собой новую цель распределения. Возвращаемся к шагу 2. Если цели нельзя достичь, потому что входные данные исчерпаны, распределение заканчивается.

Правила вычисления следующего уровня чисел Фибоначчи содержатся в их определении (2.44). Итак, мы можем сосредоточить внимание на шаге 2, где при заданной цели последовательные серии должны распределяться поочередно на $n-1$ лент. Именно здесь в наших рассуждениях должны вновь появиться фиктивные серии.

Предположим, что, повышая уровень, мы записываем следующую цель с помощью разностей d_i для $i=1, \ldots, n-1$, где d_i обозначает число серий, которые на данном шаге нужно отправить на ленту i. Теперь можно считать, что мы сразу помещаем d_i фиктивных серий на ленту i и рассматриваем последующее распределение как замену фиктивных серий действительными так, что при каждой замене d_i уменьшается на 1. Таким образом, d_i будет указывать число фиктивных серий на ленте i, когда входные данные будут исчерпаны.

Неизвестно, какой алгоритм дает оптимальное распределение, но следующий, предлагаемый нами метод оказался очень хорошим. Он называется *горизонтальным распределением* (см. [2.7], т. 3, стр. 322); этот термин становится понятным, если представить себе серии сложенными в виде пирамиды, как показано на рис. 2.16 для $n = 6$ уровня 5 (см. табл. 2.14).

Для того чтобы получить равномерное распределение оставшихся фиктивных серий как можно более быстрым способом, при замене их на действительные уменьшается высота пирамиды: фиктивные серии берутся с горизонтальных уровней слева направо. При таком способе серии распределяются на ленты в последовательности, указанной числами на рис. 2.16.

8	1				
7	2	3	4		
6	5	6	7	8	
5	9	10	11	12	
4	13	14	15	16	17
3	18	19	20	21	22
2	23	24	25	26	27
1	28	29	30	31	32

Рис. 2.16. «Горизонтальное распределение» серий.

Теперь мы можем описать алгоритм в виде процедуры, называемой *selecttape*, которая вызывается каждый раз, когда переписана какая-либо серия и нужно выбрать ленту, с которой берется очередная серия. Мы предполагаем, что существует переменная j, обозначающая индекс текущей выходной ленты. Переменные a_i и d_i обозначают числа идеального и фиктивного распределений для ленты i:

```
j: tapeno;
a,d: array [tapeno] of index;                    (2.45)
level: integer
```

Эти переменные инициируются следующими значениями:

$$a_1 = 1, \quad d_i = 1 \quad \text{для} \quad i = 1 \ldots n-1,$$
$$a_n = 0, \quad d_n = 0 \quad \text{(фиктивные)},$$
$$j = 1,$$
$$level = 1.$$

Отметим, что *selecttape* должна вычислять следующую строку табл. 2.14, т. е. значения $a_1^{(l)}, \ldots, a_{n-1}^{(l)}$, каждый раз при уве-

личении уровня. В это же время вычисляется также «очередная цель», т. е. разности $d_i = a_i^{(l)} - a_i^{(l-1)}$. Приведенный алгоритм основан на том, что результирующее значение d_i уменьшается при увеличении номера строки (ведущие вниз ступени на рис. 2.16). (Отметим, что исключением является переход с уровня 0 на уровень 1; следовательно, этот алгоритм должен использоваться, начиная с уровня 1.) *Selecttape* заканчивает работу, уменьшая d_j на 1; это соответствует замене фиктивной серии на ленте j действительной серией:

```
procedure selecttape;
  var i: tapeno; z: integer;
begin
  if d[j] < d[j+1] then j := j+1 else
  begin if d[j] = 0 then
    begin level := level + 1; z := a[1];
      for i := 1 to n-1 do
      begin d[i] := z+a[i+1]-a[i]; a[i] := z+a[i+1]
      end
    end ;
    j := 1
  end ;
  d[j] := d[j] -1
end
```
(2.46)

Предполагая, что у нас есть процедура для переписи серии с $f0$ на $f[j]$, мы можем записать начальную фазу распределения следующим образом (как обычно, считаем, что на входе имеется по крайней мере одна серия):

```
repeat selecttape, copyrun
until eof(f0)
```
(2.47)

Но здесь мы должны остановиться и вспомнить эффект, который наблюдался при распределении серий в рассмотренном ранее алгоритме естественного слияния: из-за того, что две серии, последовательно записанные на одну ленту, могут образовать одну серию, реальное количество серий на лентах может не совпасть с ожидаемым. Когда алгоритм сортировки разрабатывался таким образом, что его работа не зависела от количества серий, этот побочный эффект можно было спокойно игнорировать. Но при многофазной сортировке мы особенно заботимся о том, чтобы знать точные количества серий на каждой ленте. Следовательно, нам приходится учитывать возможность такого случайного слияния.

Поэтому нельзя избежать нового усложнения алгоритма распределения. Оказывается необходимым сохранять ключ

последнего элемента последней серии каждой ленты. Для этого мы вводим переменную

last: **array**[*tapeno*] **of** *integer*

Теперь алгоритм распределения можно представить следующим образом:

```
repeat selecttape;
   if last[j] ≤ f0↑ .key then
      «продолжение прежней серии»;                    (2.48)
   copyrun; last[j] := f0↑ .key
until eof(f0)
```

Здесь содержится очевидная ошибка: мы забыли о том, что *last*[*j*] получает (определенное) значение только после переписи первой серии! Правильное решение состоит в том, что вначале распределяется по одной серии на каждую из $n-1$ лент без обращения к *last*[*j*]. Оставшиеся серии распределяются согласно (2.49):

```
while ¬eof(f0) do
begin selecttape;
   if last[j] ≤ f0↑ .key then
   begin {продолжение прежней серии}
      copyrun;                                        (2.49)
      if eof(f0) then d[j] := d[j] + 1 else copyrun
   end
   else copyrun
end
```

Предполагается, что присваивание значений *last*[*j*] включено в процедуру *copyrun*.

Теперь мы, наконец, готовы взяться за основной алгоритм многофазной сортировки слиянием. Его принципиальная структура подобна основной части программы n-путевого слияния: имеется внешний цикл, в теле которого сливаются серии, пока не будут исчерпаны все входные данные, внутренний цикл, в теле которого сливается по одной серии с каждой входной ленты, и самый внутренний цикл, в теле которого выбирается начальный ключ и элемент передается в выходной файл. Принципиальные отличия следующие:

1. На каждом проходе имеется только одна выходная лента вместо $n/2$.
2. Вместо переключения $n/2$ входных и $n/2$ выходных лент после каждого прохода ленты *чередуются*. Это достигается с помощью карты ленточных индексов *t*.

3. Число входных лент меняется от серии к серии; в начале каждой серии оно определяется по счетчикам d_i фиктивных серий. Если $d_i > 0$ для всех i, то $n-1$ фиктивных серий сливаются в одну фиктивную серию при помощи простого увеличения счетчика d_n выходной ленты. В противном случае со всех лент, у которых $d_i = 0$, сливается по одной серии, а для всех остальных лент d_i уменьшается, что означает исключение одной фиктивной серии. Число входных лент, участвующих в слиянии, мы обозначаем через k.
4. Невозможно установить окончание фазы при помощи состояния конца файла $(n-1)$-й ленты, поскольку могут понадобиться дальнейшие слияния, в которых участвуют фиктивные серии с этой ленты. Вместо этого теоретически необходимое число серий определяется по коэффициентам a_i. Коэффициенты a_i были вычислены на фазе распределения; теперь их можно вычислить в «обратном порядке».

Теперь, согласно этим правилам, сформулируем основную часть алгоритма многофазной сортировки, предполагая, что все $n-1$ лент с начальными сериями перемотаны на начало и установлены начальные значения в карте ленточных индексов $t_i = i$:

```
repeat {слияние с t[1] . . . t[n − 1] на t[n]}
   z := a[n−1]; d[n] := 0; rewrite(f[t[n]]);
   repeat k := 0; {слияние одной серии}
      {определение числа k входных лент, участвующих в слиянии}
      for i := 1 to n−1 do
      if d[i] > 0 then d[i] := d[i] −1 else
         begin k := k+1; ta[k] := t[i]
         end ;
      if k = 0 then d[n] := d[n] + 1 else
         «слияние одной действительной серии с t[1] . . .t[k]»
      z := z−1
   until z = 0;
   reset(f[t[n]]);
   «переключение лент в карте t; вычисление a[j] для следующего
   уровня»;
   rewrite(f[t[n]]); level := level − 1
until level = 0;
{отсортированный файл находится на t[1]}
```

(2.50)

Операция действительного слияния почти идентична с программой n-путевой сортировки слиянием, единственная

```
program polysort (output);
{многофазная сортировка с n лентами}
const n = 6;                {число лент}
type item = record
              key: integer
            end ;
   tape = file of item;
   tapeno = 1 . . n;
var leng, rand: integer;    {используются для формирования файла}
   eot: boolean;
   buf: item;
   f0: tape;  {f0 -входная лента со случайными числами}
   f: array [1 . . n] of tape;
procedure list (var f: tape; n: tapeno);
   var z: integer;
begin z := 0;
   writeln ('TAPE', n: 2);
   while ¬eof(f) do
   begin read(f, buf); write(output, buf.key: 5); z := z+1;
      if z = 25 then
         begin writeln (output); z := 0
         end
   end ;
   if z ≠ 0 then writeln (output); reset(f)
end {list} ;
procedure polyphasesort;
   var i,j,mx,tn: tapeno;
      k, level: integer;
      a, d: array [tapeno] of integer;
         {a[j]-идеальное число серий на ленте j}
         {d[j]-число фиктивных серий на ленте j}
      dn,x,min,z: integer;
      last: array [tapeno] of integer;
         {last[j]-ключ конечной серии на ленте j}
      t,ta: array [tapeno] of tapeno;
         {карты номеров лент}
   procedure selecttape;
      var i: tapeno; z: integer;
   begin
      if d[j] < d[j+1] then j := j+1 else
      begin if d[j] = 0 then
            begin level := level + 1; z := a[1];
```

```
                for i := 1 to n−1 do
                begin d[i] := z + a[i+1] − a[i]; a[i] := z + a[i+1]
                end
            end ;
            j := 1
        end ;
        d[j] := d[j] −1
    end ;
    procedure copyrun;
    begin {перепись одной серии с f0 на ленту j}
        repeat read(f0, buf); write(f[j], buf);
        until eof(f0) ∨ (buf .key > f0↑ .key);
        last[j] := buf .key
    end ;
begin {распределение начальных серий}
    for i := 1 to n−1 do
        begin a[i] := 1; d[i] := 1; rewrite(f[i])
        end ;
    level := 1; j := 1; a[n] := 0; d[n] := 0;
    repeat selecttape; copyrun
    until eof(f0) ∨ (j=n−1);
    while ¬eof(f) do
    begin selecttape;
        if last[j] ≤ f0↑ .key then
        begin {продолжение прежней серии}
            copyrun;
            if eof(f0) then d[j] := d[j] + 1 else copyrun
        end
        else copyrun
    end ;
    for i := 1 to n−1 do reset(f[i]);
    for i := 1 to n do t[i] := i;
    repeat {слияние с t[1] . . . t[n − 1] на t[n]}
        z := a[n−1]; d[n] := 0; rewrite(f[t[n]]);
        repeat k := 0;     {слияние одной серии}
            for i := 1 to n−1 do
            if d[i] > 0 then d[i] := d[i]−1 else
                begin k := k+1; ta[k] := t[i]
                end ;
            if k = 0 then d[n] := d[n] + 1 else
            begin {слияние одного действительного отрезка
                с t[1] . . . t[k]}
```

```
            repeat i := 1; mx := 1;
               min := f[ta[1]]↑ .key;
               while i < k do
               begin i := i+1; x := f[ta[i]]↑ .key;
                  if x < min then
                  begin min := x; mx := i
                  end
               end ;
               {ta[mx] содержит наименьший элемент, пересылка
                его на t[n]}
               read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
               write(f[t[n]], buf);
               if (buf.key > f[ta[mx]]↑.key) ∨ eot then
               begin {сброс этой ленты}
                  ta[mx] := ta[k]; k := k-1
               end
            until k = 0
         end ;
         z := z-1
      until z = 0;
      reset(f[t[n]]); list(f[t[n]], t[n]); {переключение ленты}
      tn := t[n]; dn := d[n]; z := a[n-1];
      for i := n downto 2 do
         begin t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
         end ;
      t[1] := tn; d[1] := dn; a[1] := z;
      {отсортированный файл находится на t[1]}
      list(f[t[1]], t[1]); level := level - 1
   until level = 0;
end {polyphasesort} ;
begin {формирование случайного файла}
   leng := 200; rand := 7789;
   repeat rand := (131071*rand) mod 2147483647;
      buf.key := rand div 2147484; write(f0, buf); leng := leng - 1
   until leng = 0;
   reset(f0); list(f0, 1);
   polyphasesort
end
```

Программа 2.16. Многофазная сортировка.

разница заключается в том, что алгоритм исключения ленты несколько проще. Как производится поворот карты ленточных индексов и соответствующих счетчиков d_i (и перевычисление коэффициентов a_i при переходе на низший уровень) — очевидно, это можно подробно видеть в программе 2.16, которая полностью описывает алгоритм многофазной сортировки.

2.3.5. Распределение начальных серий

Мы пришли к сложным программам последовательной сортировки, поскольку более простые методы, работающие с массивами, требуют наличия достаточно большой памяти с произвольным доступом, чтобы хранить все сортируемое множество данных. Очень часто такой памяти нет, вместо нее приходится использовать достаточно вместительные запоминающие устройства с последовательным доступом. Мы видим, что методы последовательной сортировки, рассмотренные выше, не требуют практически никакой оперативной памяти, кроме буферов для файлов и, разумеется, самой программы. Но в действительности даже небольшие вычислительные машины обладают некоторой оперативной памятью с произвольным доступом, которая почти всегда больше той, которая требуется для разработанных здесь программ. Непростительно было бы не попытаться использовать ее оптимальным образом.

Решение заключается в *комбинировании* методов сортировки массивов и файлов. В частности, адаптированную сортировку массивов можно использовать на фазе распределения начальных серий так, чтобы в результате эти серии имели длину l, приблизительно равную размеру имеющейся оперативной памяти. Очевидно, что на последующих проходах никакие дополнительные сортировки массивов не дадут какого-либо улучшения, так как длина участвующих в них серий постоянно растет и, следовательно, всегда будет больше имеющейся оперативной памяти. Поэтому мы можем сосредоточить внимание на оптимизации алгоритма, который формирует начальные серии.

Конечно, мы сразу обращаемся к логарифмическим методам сортировки массивов. Наиболее подходящий из них — это сортировка с помощью дерева, или пирамидальная сортировка (см. разд. 2.2.5). Пирамиду можно рассматривать как туннель, через который должны пройти все компоненты файла, некоторые — быстрее, некоторые — медленнее. Наименьший ключ легко извлекается с вершины пирамиды, а его замещение — очень эффективный процесс. Пропуск компоненты с входной ленты $f0$ через весь «пирамидальный тун-

нель» h на выходную ленту $f[j]$ можно просто описать следующим образом:

$$\begin{aligned} &write(f[j],\ h[1]);\\ &read(f0,\ h[1]);\\ &sift(1,\ n) \end{aligned} \qquad (2.51)$$

«*Sift*» («просеивание») — это процесс, описанный в разд. 2.2.5. Новая вставляемая компонента $h[1]$ просеивается на соответствующее место. Отметим, что $h[1]$ — наименьший элемент в пирамиде. Пример дан на рис. 2.17.

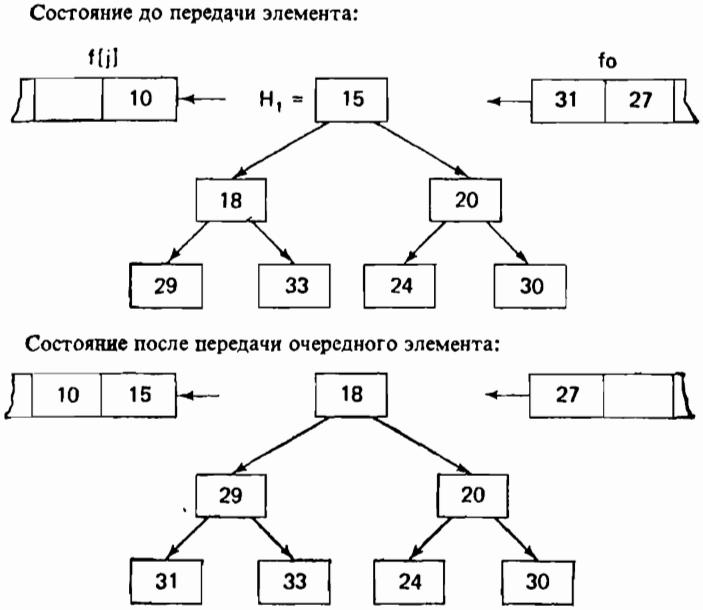


Рис. 2.17. Просеивание ключа через пирамиду.

В конечном счете программа значительно усложняется, поскольку:

1. Пирамида h вначале пуста и прежде всего должна быть заполнена.
2. К концу работы пирамида заполнена лишь частично, а в самом конце становится пустой.
3. Нужно сохранять информацию о начале новых серий, для того чтобы вовремя сменить индекс выходной ленты j.

Прежде всего опишем формально переменные, явно участвующие в работе:

```
var f0: tape;
    f: array[tapeno] of tape;
    h: array[1..m] of item;
    l,r: integer
```
(2.52)

m — размер пирамиды h. Для обозначения $m/2$ мы используем константу mh; l и r — индексы в h. Процесс пропуска элементов через пирамиду можно теперь разбить на пять отдельных этапов:

1. Прочесть mh первых элементов с $f0$ и записать их в верхнюю половину пирамиды, где не требуется никакого упорядочения ключей.
2. Прочесть mh остальных элементов и записать их в нижнюю половину пирамиды, просеивая каждый элемент на соответствующее место (построить пирамиду).
3. Установить l в m и выполнить для оставшихся на $f0$ элементах следующий шаг: отправить $h[1]$ на текущую выходную ленту. Если его ключ меньше или равен ключу следующего элемента входной ленты, то этот следующий элемент принадлежит той же серии и его можно просеять на подходящее место. В противном случае надо уменьшить размер пирамиды и поместить новый элемент во вторую, «верхнюю» пирамиду, которая строится для следующей серии. Границу между двумя пирамидами обозначим индексом l. Итак, «нижняя»», или текущая, пирамида состоит из элементов $h[1] \ldots h[l]$, а «верхняя», или следующая, пирамида — из $h[l+1] \ldots h[m]$. Если $l = 0$, то нужно сменить выходную ленту и вновь установить l в m.
4. Теперь входные ленты исчерпаны. Вначале установить r в m, затем сбросить на выходную ленту нижнюю часть пирамиды, заканчивающую текущую серию, одновременно построить верхнюю часть и постепенно переместить ее в позиции $h[l+1] \ldots h[r]$.
5. Последняя серия формируется из оставшихся элементов пирамиды.

Теперь мы можем подробно описать эти пять этапов в виде законченной программы, вызывающей процедуру *selecttape* как только найден конец серии и нужно совершить некоторое действие, изменяющее индекс выходной ленты. В программе 2.17 вместо этого используется фиктивная процедура: она просто подсчитывает число сформированных серий. Все элементы записываются на ленту $f1$.

Если теперь мы попытаемся объединить эту программу, например, с программой многофазной сортировки, то столкнемся с серьезной трудностью. Она возникает по следующим причинам: программа сортировки содержит вначале довольно сложную процедуру для переключения лент и предполагает наличие процедуры *copyrun*, которая записывает на выбранную ленту ровно одну серию. С другой стороны, программа пирамидальной сортировки представляет собой сложную процедуру, предполагающую наличие закрытой процедуры *selecttape*, которая просто выбирает новую ленту. Проблемы не возникало бы, если бы в одной (или обеих) программе нужная процедура вызывалась лишь в одном месте, однако она вызывается в нескольких местах в обеих программах.

В такой ситуации лучше всего использовать так называемую *сопрограмму*, как обычно рекомендуется, когда сосуществуют несколько процессов. Самый типичный пример — комбинация процесса, который порождает поток информации, и процесса, который ее использует. Связь порождение/использование можно выразить в виде двух сопрограмм. Одна из них может быть основной программой.

Сопрограмму можно рассматривать как процедуру, или подпрограмму, которая содержит одну или несколько точек прерывания. Если встречается такая точка прерывания, то управление возвращается в программу, вызвавшую эту сопрограмму. При повторном вызове сопрограммы ее выполнение возобновляется с этой точки прерывания. В нашем примере мы можем рассматривать многофазную сортировку как основную программу, вызывающую *copyrun*, которая построена в виде сопрограммы. Она состоит из основного тела программы 2.17, где каждый вызов *selecttape* теперь представляет собой точку прерывания. Проверку на конец файла нужно всюду заменить проверкой, достигла ли сопрограмма своего конца. Логично заменить *eof*(*f*0) на *eoc*(*copyrun*).

Анализ и выводы. Какой эффективности можно ожидать от многофазной сортировки с начальным распределением серий с помощью пирамидальной сортировки? Вначале мы обсудим, каких улучшений следует ожидать от использования пирамиды.

В последовательности со случайно распределенными ключами средняя длина серий равна 2. Какова эта длина после того, как последовательность пропущена через пирамиду размером m? Казалось бы, нужно ответить m, но, к счастью, результат вероятностного анализа на самом деле намного лучше, а именно равен $2m$ (см. [2.7], т. 3, с. 304). Поэтому ожидаемый коэффициент улучшения равен m.

```
program distribute(f0,f1,output);
{начальное распределение серий с помощью пирамидальной
сортировки}
const m = 30; mh = 15; {размер пирамиды}
type item = record
                key: integer
            end ;
   tape = file of item;
   index = 0 .. m;
var l,r: index;
   f0,f1: tape;
   count: integer;      {счетчик серий}
   h: array [1 .. m] of item; {пирамида}

procedure selecttape;
begin count := count + 1;
   {фиктивная процедура; подсчитывает число распределенных
   серий}
end {selecttape} ;

procedure sift(l,r: index);
   label 13;
   var i,j: integer; x: item;
begin i := l; j := 2*i; x := h[i];
   while j ≤ r do
   begin if j < r then
            if h[j] .key > h[j+1] .key then j := j+1;
         if x .key ≤ h[j] .key then goto 13;
         h[i] := h[j]; i := j; j := 2*i
   end ;
13: h[i] := x
end ;
begin {формирование начальных серий с помощью пирамидальной
      сортировки}
   count := 0; reset(f0); rewrite(f1);
   selecttape;
  {этап 1: заполнение верхней части пирамиды h}
   l := m;
   repeat read(f0, h[l]); l := l-1
   until l = mh;
  {этап 2: заполнение нижней части пирамиды h}
   repeat read(f0, h[l]); sift(l,m); l := l-1
   until l = 0;
  {этап 3: пропуск серий через пирамиду}
```

```
    l := m;
    while ¬eof(f0) do
    begin write(f1, h[1]);
      if h[1].key ≤ f0↑.key then
      begin {новая запись принадлежит той же серии}
        read(f0, h[1]); sift(1,l);
      end else
      begin {новая запись принадлежит следующей серии}
        h[1] := h[l]; sift(1,l-1);
        read(f0, h[l]); if l ≤ mh then sift(l,m); l := l-1;
        if l = 0 then
        begin {пирамида заполнена; начать новую серию}
          l := m; selecttape;
        end
      end
    end ;
  {этап 4: сброс нижней части пирамиды}
    r := m;
    repeat write(f1, h[1]);
      h[1] := h[l]; sift(1,l-1);
      h[l] := h[r]; r := r-1;
      if l ≤ mh then sift(l,r); l := l-1
    until l = 0;
  {этап 5: сброс верхней части пирамиды; формирование
   последней серии}
    selecttape;
    while r > 0 do
    begin write(f1,h[1]);
      h[1] := h[r]; sift(1,r); r := r-1
    end ;
    writeln (count)
  end .
```

Программа 2.17. Распределение начальных серий с помощью пирамиды.

Оценку свойств многофазной сортировки можно получить из табл. 2.15, определив максимальное число начальных серий, которые можно отсортировать за данное количество частичных проходов (уровней) при заданном числе n лент. Например, при $n = 6$ лентах и пирамиде размером $m = 100$ файл, содержащий до 165 680 100 начальных серий, можно отсортировать за 20 частичных проходов. Это — отличные характеристики.

Рассматривая программу, представляющую собой комбинацию многофазной и пирамидальной сортировки, нельзя не поразиться ее сложности. Ведь она выполняет ту же самую, легко определимую задачу переупорядочения множества элементов, что и любая из коротких программ, основанных на простых методах сортировки массивов. Из всего сказанного в этой главе можно сделать следующие выводы:

1. Между алгоритмом и структурой данных существует тесная связь; структура данных оказывает большое влияние на вид программы.
2. При помощи усложнения программы можно значительно повысить ее эффективность, даже когда структура данных, с которой она работает, плохо соответствует поставленной задаче.

УПРАЖНЕНИЯ

2.1. Какие из алгоритмов, представленных программами 2.1—2.6, 2.8, 2.10 и 2.13, являются методами устойчивой сортировки?

2.2. Будет ли программа 2.2 работать правильно, если в условии окончания цикла заменить $l \leqslant r$ на $l < r$? Будет ли она по-прежнему правильной, если операторы $r := m - 1$ и $l := m + 1$ упростить до $r := m$ и $l := m$? Если нет, найдите множества значений $a_1 \ldots a_n$, при которых измененная программа будет работать неправильно.

2.3. Запрограммируйте три метода простой сортировки и измерьте время их работы. Найдите веса, на которые нужно умножить коэффициенты C и M, чтобы получить оценки реального времени.

2.4. Протестируйте программу пирамидальной сортировки 2.8 с различными произвольными входными последовательностями и определите, сколько раз в среднем выполняется оператор **goto** 13. Поскольку это число сравнительно мало, интересен следующий вопрос: имеется ли способ извлечь проверку

$$x.key \geqslant a[j].key$$

из цикла с предусловием?

2.5. Рассмотрите следующую «очевидную» версию программы разделения 2.9:

```
i := 1; j := n;
x := a[(n+1) div 2].key;
repeat
    while a[i].key < x do i := i+1;
    while x < a[j].key do j := j-1;
    w := a[i]; a[i] := a[j]; a[j] := w
until i > j
```

Найдите множества значений $a_1 \ldots a_n$, для которых эта версия работает неправильно.

2.6. Напишите программу, которая комбинирует алгоритмы быстрой сортировки и сортировки методом пузырька следующим образом: используется быстрая сортировка для получения (неотсортированных)

подмассивов длиной $m (1 \leqslant m \leqslant n)$; затем для завершения задачи используется сортировка методом пузырька. Отметим, что последняя теперь может проходить по всему массиву, минимизируя тем самым затраты на управление. Найдите значение m, минимизирующее общее время сортировки.

Примечание. Ясно, что оптимальное значение m будет достаточно мало. Тогда, может быть, стоит позволить, чтобы сортировка методом пузырька проходила по всему массиву ровно $m - 1$ раз, вместо того чтобы определять последний проход, на котором не производилось никаких обменов.

2.7. Проведите тот же эксперимент, что и в упр. 6 с сортировкой простым выбором вместо сортировки методом пузырька. Конечно, сортировка простым выбором не может проходить по всему массиву, поэтому ожидаемый объем работы с индексами несколько больше.

2.8. Напишите рекурсивный алгоритм быстрой сортировки согласно указанию, что сортировку меньшего подмассива следует выполнять раньше сортировки более длинного подмассива. Выполните первую задачу при помощи итеративного оператора, а последнюю — при помощи рекурсивного вызова. (Следовательно, ваша процедура сортировки будет содержать один рекурсивный вызов в отличие от программы 2.10, содержащей 2 вызова, и программы 2.11, не содержащей ни одного.)

2.9. Найдите перестановку ключей 1, 2, ..., n, для которой быстрая сортировка ведет себя наихудшим (наилучшим) образом ($n = 5, 6, 8$).

2.10. Напишите программу естественного слияния, которая, подобно программе простого слияния 2.13, работает на массиве двойной длины с двух концов в середину. Сравните ее характеристики с характеристиками программы 2.13.

2.11. Заметьте, что при двухпутевом естественном слиянии мы не просто слепо выбираем наименьшее значение из имеющихся ключей. Вместо этого, когда встречается конец серии, остаток другой серии просто переписывается в выходную последовательность. Например, слияние

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

дает последовательность

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

вместо последовательности

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

которая кажется лучше упорядоченной. Почему принята такая стратегия?

2.12. Зачем нужна переменная *ta* в программе 2.15? При каких условиях выполняется оператор

begin *rewrite*(*f*[*ta*[*mx*]]); ...

а при каких — оператор

begin *tx* := *ta*[*mx*]; ...?

2.13. Почему в программе многофазной сортировки 2.16 требуется переменная *last*, а в программе 2.15 она не нужна?

2.14. Существует метод сортировки, похожий на многофазную сортировку: так называемое *каскадное слияние* [2.1, 2.9]. Он использует другой

принцип слияния. Если, например, даны шесть лент $T1, \ldots, T6$, каскадное слияние, начиная также с «идеального распределения» серий на $T1, \ldots, T5$, выполняет пятипутевое слияние с $T1, \ldots, T5$ на $T6$, пока $T5$ не станет пустой, затем (не затрагивая $T6$) четырехпутевое слияние на $T5$, затем трехпутевое слияние на $T4$, двухпутевое слияние на $T3$ и, наконец, перепись с $T1$ на $T2$. Следующий проход работает таким же образом, начиная с пятипутевого слияния на $T1$ и т. д. Хотя эта схема кажется хуже многофазного слияния, поскольку временами оставляет некоторые ленты без работы, а также выполняет операции простого копирования, она, к удивлению, оказывается лучше многофазной для очень больших файлов и для шести и более лент. Напишите хорошо структурированную программу каскадного слияния.

ЛИТЕРАТУРА

2.1. BETZ B. K., CARTER. — *ACM National Conf.*, **14**, 1959, Paper 14.
2.2. FLOYD R. W. Treesort (Algorithms 113, 243), *Comm. ACM*, **5**, No. 8, 1962, 434, *Comm. ACM*, 7. No. 12, 1964, 701.
2.3. GILSTAD R. L. Polyphase Merge Sorting — An Advanced Technique. — *Proc. AFIPS Easter Jt. Comp. Conf.* **18**, 1960, 143—148.
2.4. HOARE C. A R. Proof of Program. FIND. — *Comm. ACM*, **13**, No. 1, 1970, 39—45.
2.5. HOARE C. A. R. Proof of Recursive Program: Quicksort. — *Comp. J.*, **14**, No. 4, 1971, 391—395.
2.6. HOARE C. A. R. Quicksort. — *Comp. J.*, **5**, No. 1, 1962, 10—15.
2.7. KNUTH D. E. The art of Computer Programming, 3, — Reading, Mass.: Addison-Wesley, 1973. [Имеется перевод: Кнут Д. Искусство программирования для ЭВМ, т. 3. — М.: Мир, 1978.]
2.8. KNUTH D. E. The art of Computer Programming, 3, 86—95. [Имеется перевод: Кнут Д. Искусство программирования для ЭВМ, т. 3. — М.: Мир, 1978, с. 108—119.]
2.9. KNUTH D. E. The art of Computer Programming, 3, 289. [Имеется перевод: Кнут Д. Искусство программирования для ЭВМ, т. 3. — М.: Мир, 1978, с. 342.]
2.10. LORIN H. A Guided Bibliography to Sorting. — *IBM Syst. J.*, **10**, No. 3, 1971, 244—254.
2.11. SHELL D. L. A Highspeed Sorting Procedure. — *Comm. ACM*, **2**, No. 7, 1959, 30—32.
2.12. SINGLETON R. C. An Efficient Algorithm for Sorting with Minimal Storage (Algorithm 347). — *Comm. ACM*, **12**, No. 3, 1969, 185.
2.13. Van EMDEN M. H. Increasing the Efficiency of Quicksort (Algorithm) 402). — *Comm. ACM*, **13**, No. 9, 1970, 563—566, 693.
2.14. WILLIAMS J. W. J. Heapsort (Algorithm 232). — *Comm. ACM*, **7**, No. 6, 1964, 347—348.

3

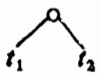
РЕКУРСИВНЫЕ АЛГОРИТМЫ

3.1. ВВЕДЕНИЕ

Объект называется *рекурсивным*, если он содержит сам себя или определен с помощью самого себя. Рекурсия встречается не только в математике, но и в обыденной жизни. Кто не видел рекламной картинки, которая содержит свое собственное изображение?

Рекурсия является особенно мощным средством в математических определениях. Известны примеры рекурсивных определений натуральных чисел, древовидных структур и некоторых функций:

1. Натуральные числа:
   (a) 1 есть натуральное число;
   (b) целое число, следующее за натуральным, есть натуральное число.
2. Древовидные структуры:
   (a) O есть дерево (называемое пустым деревом);
   (b) если t_1 и t_2 — деревья, то



   есть дерево (нарисованное сверху вниз).
3. Функция факториал $n!$ для неотрицательных целых чисел:
   (a) $0! = 1$,
   (b) если $n > 0$, то $n! = n \cdot (n-1)!$

Очевидно, что мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. Точно так же бесконечные вычисления можно описать с помощью конечной рекурсивной программы, даже если эта программа не содержит явных циклов. Однако лучше всего использовать рекурсивные алгоритмы в тех случаях, когда решаемая задача, или вычисляемая функция, или обрабатываемая структура данных определены с помощью рекурсии. В общем виде рекурсивную программу P можно изобразить как композицию $\mathscr{P}$ базовых

операторов S_i (не содержащих P) и самой P:

$$P \equiv \mathscr{P}[S_i, P]. \tag{3.1}$$

Необходимое и достаточное средство для рекурсивного представления программ — это описание процедур, или подпрограмм, так как оно позволяет присваивать какому-либо оператору имя, с помощью которого можно вызывать этот оператор. Если процедура P содержит явное обращение к самой себе, то она называется *прямо рекурсивной*; если P содержит обращение к процедуре Q, которая содержит (прямо

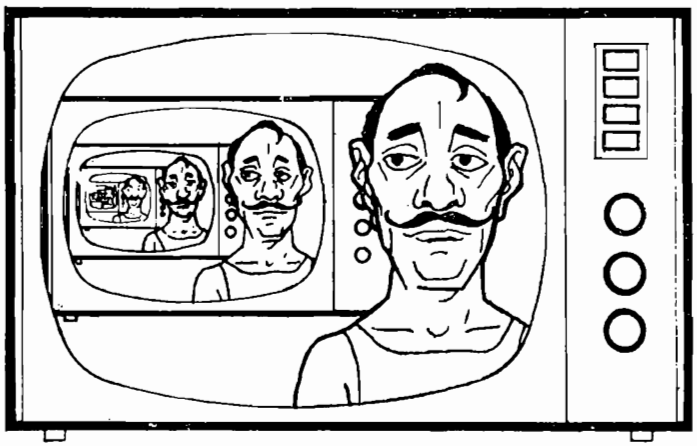


Рис. 3.1. Рекурсивное изображение.

или косвенно) обращение к P, то P называется *косвенно рекурсивной*. Поэтому использование рекурсии не всегда сразу видно из текста программы.

С процедурой принято связывать некоторое множество локальных объектов, т. е. переменных, констант, типов и процедур, которые определены локально в этой процедуре, а вне ее не существуют или не имеют смысла. Каждый раз, когда такая процедура рекурсивно вызывается, для нее создается новое множество локальных переменных. Хотя они имеют те же имена, что и соответствующие элементы множества локальных переменных, созданного при предыдущем обращении к этой же процедуре, их значения различны. Следующие правила области действия идентификаторов позволяют исключить какой-либо конфликт при использовании имен: идентификаторы всегда ссылаются на множество переменных, созданное последним. То же правило относится к параметрам процедуры.

Подобно операторам цикла, рекурсивные процедуры могут привести к бесконечным вычислениям. Поэтому необходимо рассмотреть проблему *окончания работы* процедур. Очевидно, что для того, чтобы работа когда-либо завершилась, необходимо, чтобы рекурсивное обращение к процедуре P подчинялось условию B, которое в какой-то момент перестает выполняться. Поэтому более точно схему рекурсивных алгоритмов можно представить так:

$$P \equiv \textbf{if } B \textbf{ then } \mathscr{P}[S_i, P] \tag{3.2}$$

или

$$P \equiv \mathscr{P}[S_i, \textbf{ if } B \textbf{ then } P] \tag{3.3}$$

Основной способ доказать, что выполнение операторов цикла когда-либо заканчивается, — определить функцию $f(x)$ (x — множество переменных программы), такую, что $f(x) \leqslant 0$ удовлетворяет условию окончания цикла (с предусловием или с постусловием), и доказать, что при каждом повторении $f(x)$ уменьшается. Точно так же можно доказать, что выполнение рекурсивной процедуры P когда-либо завершится, показав, что каждое выполнение P уменьшает $f(x)$. Наиболее надежный способ обеспечить окончание процедуры — связать с P параметр (значение), скажем n, и рекурсивно вызывать P со значением этого параметра $n-1$. Тогда замена условия B на $n > 0$ гарантирует окончание работы. Это можно изобразить следующими схемами программ:

$$P(n) \equiv \textbf{if } n > 0 \textbf{ then } \mathscr{P}[S_i, P(n-1)] \tag{3.4}$$

$$P(n) \equiv \mathscr{P}[S_i, \textbf{ if } n > 0 \textbf{ then } P(n-1)] \tag{3.5}$$

На практике нужно обязательно убедиться, что наибольшая глубина рекурсии не только конечна, но и достаточно мала. Дело в том, что при каждом рекурсивном вызове процедуры P выделяется некоторая память для размещения ее переменных. Кроме этих локальных переменных нужно еще сохранять текущее состояние вычислений, чтобы вернуться к нему, когда закончится выполнение новой активации P и нужно будет вернуться к старой. Мы уже наблюдали подобную ситуацию при разборе процедуры быстрой сортировки в гл. 2. Было обнаружено, что при «наивном» составлении программы из оператора, разделяющего n элементов на две части, и двух рекурсивных вызовов, сортирующих эти две части, глубина рекурсии в худшем случае может приближаться к n. При разумном изменении процедуры оказалось, что можно ограничить эту глубину $\log n$. Разница между значениями n и $\log n$ вполне достаточна для того, чтобы случай, крайне не подходящий для использования рекурсии, превратить в тот, в котором рекурсия вполне практична.

3.2. КОГДА НЕ НУЖНО ИСПОЛЬЗОВАТЬ РЕКУРСИЮ

Рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. Но это не значит, что при наличии таких рекурсивных определений лучшим способом решения задачи непременно является рекурсивный алгоритм. В действительности из-за того, что обычно понятие рекурсивных алгоритмов объяснялось на неподходящих примерах, в основном и возникло широко распространенное предубеждение против использования рекурсии в программировании и приравнивание ее к неэффективности. Повлиял на это и тот факт, что широко распространенный язык программирования Фортран запрещает рекурсивное использование подпрограмм и тем самым не допускает рекурсию, даже когда ее применение оправданно.

Программы, в которых следует избегать использования рекурсии, можно охарактеризовать следующей схемой, изображающей их строение. Это схема (3.6) и эквивалентная ей (3.7):

$$P \equiv \textbf{if } B \textbf{ then } (S;\ P) \tag{3.6}$$

$$P \equiv (S;\ \textbf{if } B \textbf{ then } P) \tag{3.7}$$

Эти схемы естественно применять в тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений. Рассмотрим, например, широко известный пример вычислений факториалов $f_i = i!$:

$$\begin{aligned} i &= 0,\ 1,\ 2,\ 3,\ \ 4,\ \ \ 5,\ \ldots, \\ f_i &= 1,\ 1,\ 2,\ 6,\ 24,\ 120,\ \ldots. \end{aligned} \tag{3.8}$$

«Нулевое» число определяется явным образом как $f_0 = 1$, а последующие числа обычно определяются рекурсивно — с помощью предшествующего значения:

$$f_{i+1} = (i+1) \cdot f_i. \tag{3.9}$$

Эта формула предполагает использование рекурсивного алгоритма для вычисления n-го факториального числа. Если мы введем две переменные I и F для значений i и f_i на i-м уровне рекурсии, то увидим, что для перехода к следующему числу в последовательности (3.8) необходимы следующие вычисления:

$$I := I + 1; \quad F := I * F \tag{3.10}$$

и, подставив (3.10) вместо S в (3.6), мы получаем рекурсивную программу

$$\begin{aligned} &P \equiv \textbf{if } I < n \textbf{ then } (I := I + 1;\ F := I * F;\ P) \\ &I := 0;\ F := 1;\ P \end{aligned} \tag{3.11}$$

Первую строку в (3.11) можно так записать на принятом нами языке программирования Паскаль:

```
procedure P;
begin if I < n then
      begin I := I + 1; F := I*F; P
      end
end
```
(3.12)

Чаще употребляемая, но эквивалентная, по существу, форма дана в (3.13). Вместо процедуры здесь вводится так называемая процедура-функция, т. е. некоторая процедура, с которой явно связывается вычисляемое значение. Поэтому функцию можно использовать непосредственно как элемент выражения. Тем самым переменная F становится излишней, а роль I выполняет явный параметр процедуры.

```
function F(I: integer): integer;
begin if I > 0 then F := I*F(I - 1)
               else F := 1
end
```
(3.13)

Совершенно ясно, что здесь рекурсию можно заменить обычной итерацией, а именно программой

```
I := 0; F := 1;
while I < n do
   begin I := I + 1; F := I*F
   end
```
(3.14)

В общем виде программы, соответствующие схемам (3.6) или (3.7), нужно преобразовать так, чтобы они соответствовали схеме (3.15):

$$P \equiv (x := x_0;\ \textbf{while}\ B\ \textbf{do}\ S) \tag{3.15}$$

Есть и другие, более сложные рекурсивные схемы, которые можно и должно переводить в итеративную форму. Примером служит вычисление чисел Фибоначчи, определяемых с помощью рекуррентного соотношения

$$\mathrm{fib}_{n+1} = \mathrm{fib}_n + \mathrm{fib}_{n-1} \quad \text{для} \quad n > 0 \tag{3.16}$$

и $\mathrm{fib}_1 = 1$, $\mathrm{fib}_0 = 0$. При непосредственном, «лобовом» подходе мы получим программу

```
function Fib(n: integer): integer;
begin if n = 0 then Fib := 0 else
      if n = 1 then Fib := 1 else
      Fib := Fib(n-1) + Fib(n-2)
end
```
(3.17)

При вычислении fib_n обращение к функции $Fib(n)$ приводит к рекурсивным активациям этой процедуры. Сколько раз? Мы можем заметить, что каждое обращение при $n > 1$ приводит к двум дальнейшим обращениям, т. е. общее число обращений растет экспоненциально (см. рис. 3.2). Ясно, что такая программа непригодна для практического использования.

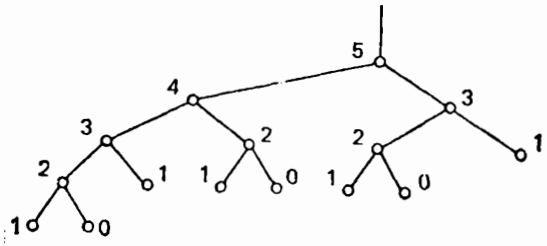


Рис. 3.2. 15 вызовов $Fib(n)$ при $n = 5$.

Однако очевидно, что числа Фибоначчи можно вычислять по итеративной схеме, при которой использование вспомогательных переменных $x = \mathrm{fib}_i$ и $y = \mathrm{fib}_{i-1}$ позволяет избежать повторного вычисления одних и тех же значений:

```
{вычисление x = fib_n для n > 0}
i := 1; x := 1; y := 0;
while i < n do                                   (3.18)
   begin z := x; i := i + 1;
         x := x + y; y := z
   end
```

(Заметим, что три присваивания x, y и z можно выразить всего лишь двумя присваиваниями без использования вспомогательной переменной z: $x := x + y$; $y := x - y$.)

Итак, вывод таков: следует избегать рекурсии, когда имеется *очевидное* итеративное решение поставленной задачи.

Но это не означает, что всегда нужно избавляться от рекурсии любой ценой. Во многих случаях она вполне применима, как будет показано в следующих разделах этой главы и в последующих главах. Тот факт, что рекурсивные процедуры можно реализовать на нерекурсивных по сути машинах, говорит о том, что для практических целей любую рекурсивную программу можно преобразовать в чисто итеративную. Но это требует явного манипулирования со стеком рекурсий, и эти операции до такой степени заслоняют суть программы, что понять ее становится очень трудно. Следовательно, алгоритмы, которые по своей природе скорее рекурсивны, чем итеративны, нужно представлять в виде

рекурсивных процедур. Чтобы лучше понять это, мы предлагаем читателю сравнить программы 2.10 и 2.11.

Оставшаяся часть этой главы посвящена разработке некоторых рекурсивных программ в тех случаях, когда рекурсия полностью оправданна. Кроме того, в гл. 4 и 5 также широко используется рекурсия, если, конечно, структуры данных естественно и очевидно приводят к рекурсивным решениям.

3.3. ДВА ПРИМЕРА РЕКУРСИВНЫХ ПРОГРАММ

Симпатичный узор на рис. 3.5 состоит из суперпозиции пяти кривых. Эти кривые строятся на основе некоторого регулярного образца, и предполагается, что их можно нарисовать с помощью графопостроителя, управляемого вычислительной машиной. Наша задача — найти рекурсивную схему, по которой можно написать программу, управляющую графопостроителем. Рассматривая рисунок, мы обнаруживаем, что

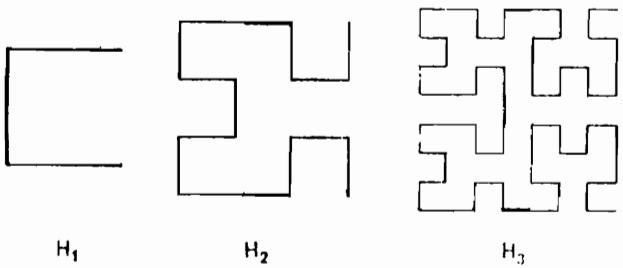


Рис. 3.3. Кривые Гильберта порядка 1, 2 и 3.

три наложенные друг на друга кривые имеют форму, показанную на рис. 3.3. Мы обозначаем их через H_1, H_2 и H_3. На рисунках видно, что H_{i+1} получается соединением четырех H_i вдвое меньшего размера, соответствующим образом повернутых и связанных вместе тремя соединительными линиями. Отметим, что можно считать, что H_1 состоит из четырех пустых H_0, связанных тремя прямыми линиями. Кривая H_i называется *кривой Гильберта i-го порядка* в честь его первооткрывателя Д. Гильберта (1891).

Предположим, что у нас имеются следующие основные средства для построения графов: две координаты — переменные x и y, процедура *setplot* (устанавливающая перо в точку с координатами x и y) и процедура *plot* (передвигающая перо, которое при этом чертит прямую из текущей точки в точку, обозначенную x и y).

Поскольку каждая кривая H_i состоит из четырех вдвое меньших копий H_{i-1}, то естественно построить процедуру, ри-

```
program Hilbert(pf,output);
 изображение кривых Гильберта порядка  от 1 до n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
   pf: file of integer;      {plot file}
procedure A(i: integer);
begin if i > 0 then
      begin D(i-1); x := x-h; plot;
            A(i-1); y := y-h; plot;
            A(i-1); x := x+h; plot;
            B(i-1)
      end
end ;
procedure B(i: integer);
begin if i > 0 then
      begin C(i-1); y := y+h; plot;
            B(i-1); x := x+h; plot;
            B(i-1); y := y-h; plot;
            A(i-1)
      end
end ;
procedure C(i: integer);
begin if i > 0 then
      begin B(i-1); x := x+h; plot;
            C(i-1); y := y+h; plot;
            C(i-1); x := x-h; plot;
            D(i-1)
      end
end ;
procedure D(i: integer);
begin if i > 0 then
      begin A(i-1); y := y-h; plot;
            D(i-1); x := x-h; plot;
            D(i-1); y := y+h; plot;
            C(i-1)
      end
end ;
begin startplot;
   i := 0; h := h0; x0 := h div 2; y0 := x0;
   repeat {изображение кривой Гильберта порядка i}
      i := i+1; h := h div 2;
      x0 := x0 + (h div 2); y0 := y0 + (h div 2);
      x := x0; y := y0; setplot;
```

```
        A(i)
    until i = n;
    endplot
end .
```

Программа 3.1. Кривые Гильберта.

сующую H_i в виде композиции четырех частей, каждая из которых рисует H_{i-1} соответствующего размера и с нужным поворотом. Если мы обозначим эти четыре части A, B, C и D, а подпрограммы, рисующие соединительные линии, — в виде стрелок, указывающих соответствующее направление, то получим следующую рекурсивную схему (см. рис. 3.3):

$$\begin{aligned} &\sqsubset A: \quad D \leftarrow A \downarrow A \rightarrow B \\ &\sqcap B: \quad C \uparrow B \rightarrow B \downarrow A \\ &\sqsupset C: \quad B \rightarrow C \uparrow C \leftarrow D \\ &\sqcup D: \quad A \downarrow D \leftarrow D \uparrow C \end{aligned} \tag{3.19}$$

Если длину соединительной линии обозначить через h, то процедуру, соответствующую схеме A, можно легко выразить с помощью рекурсивных обращений к описанным аналогичным образом процедурам B и D и самой процедуры A:

```
procedure A(i: integer);
begin if i > 0 then
    begin D(i−1); x := x−h; plot;
          A(i−1); y := y−h; plot;
          A(i−1); x := x+h; plot;
          B(i−1)
    end
end
```
(3.20)

Эта процедура инициируется один раз основной программой для каждой кривой Гильберта, которые накладываются одна на другую, образуя данный рисунок. Основная программа задает исходную точку для кривой, т. е. начальные значения x и y, и единичное приращение h. Величина h_0 соответствует ширине всей страницы и должна удовлетворять равенству $h0 = 2^k$ для некоторого $k \geqslant n$ (см. рис. 3.4). Программа рисует всего n кривых Гильберта (см. программу 3.1 и рис. 3.5).

Похожий, но несколько более сложный и эстетически утонченный рисунок приведен на рис. 3.7. Он также получен с помощью наложения нескольких кривых; две такие кривые показаны на рис. 3.6. Кривая S_i называется кривой *Серпинского* i-го порядка. Какова рекурсивная схема для такой

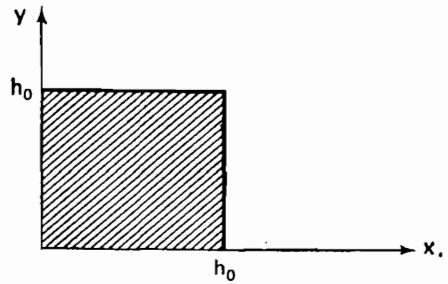


Рис. 3.4. Рамка для кривых.

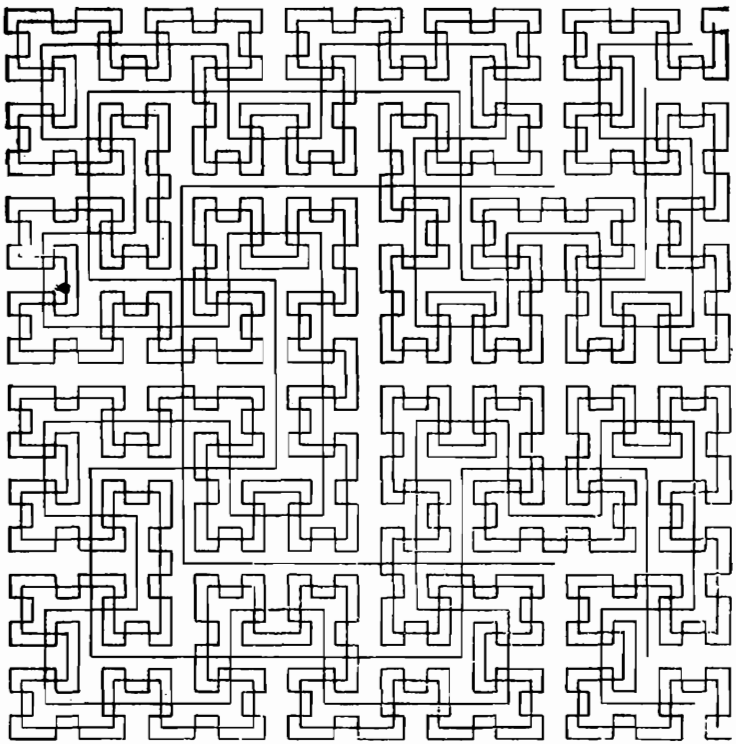


Рис. 3.5. Кривые Гильберта порядка $H_1, \ldots, H_8$.

кривой? Попробуем в качестве основного строительного блока выделить лист S_1, возможно, без одного ребра. Но это не приводит нас к нужному решению. Принципиальное различие между кривыми Серпинского и Гильберта заключается в том, что кривые Серпинского являются замкнутыми (без соединительных линий). Это означает, что основная рекурсивная схема должна давать разомкнутую кривую, а четыре части соединяются линиями, не принадлежащими самому рекурсивному узору. Действительно, эти связи представляют собой

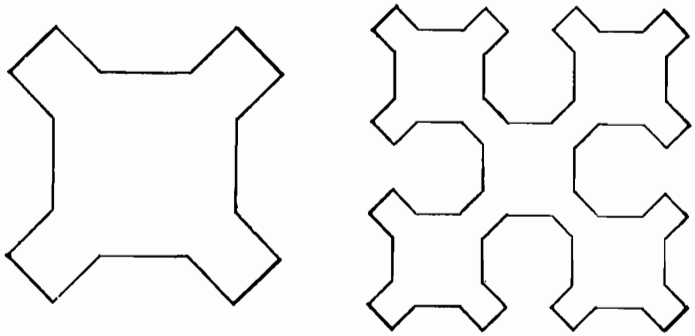


Рис. 3.6. Кривые Серпинского порядка 1 и 2.

четыре прямые в четырех внешних «углах», изображенных на рис. 3.6 жирными линиями. Можно считать, что они принадлежат к непустой начальной кривой S_0, представляющей собой квадрат, стоящий на одном угле.

Теперь легко построить рекурсивную схему. Четыре составляющие фигуры вновь обозначаются A, B, C и D, а соединительные линии рисуются явно. Заметим, что четыре рекурсивные кривые действительно одинаковы с точностью до поворота на 90°.

Основной образ кривых Серпинского следующий:

$$S\colon A \searrow B \swarrow C \nwarrow D \nearrow \tag{3.21}$$

а объединенные рекурсивные фигуры строятся по таким схемам:

$$\begin{aligned} A&\colon A \searrow B \Rightarrow D \nearrow A \\ B&\colon B \swarrow C \Downarrow A \searrow B \\ C&\colon C \nwarrow D \Leftarrow B \swarrow C \\ D&\colon D \nearrow A \Uparrow C \nwarrow D \end{aligned} \tag{3.22}$$

(Двойные стрелки обозначают линии двойной длины.)

```
program Sierpinski (pf,output);
{изображение кривых Серпинского порядка от 1 до n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
   pf: file of integer;
procedure A(i: integer);
begin if i > 0 then
      begin A(i-1); x := x+h; y := y-h; plot;
            B(i-1); x := x + 2*h; plot;
            D(i-1); x := x+h; y := y+h; plot;
            A(i-1)
      end
end ;
procedure B(i: integer);
begin if i > 0 then
      begin B(i-1); x := x-h; y := y-h; plot;
            C(i-1); y := y - 2*h; plot;
            A(i-1); x := x+h; y := y-h; plot;
            B(i-1)
      end
end ;
procedure C(i: integer);
begin if i > 0 then
      begin C(i-1); x := x-h; y := y+h; plot;
            D(i-1); x := x - 2*h; plot;
            B(i-1); x := x-h; y := y-h; plot;
            C(i-1)
      end
end ;
procedure D(i: integer):
begin if i > 0 then
      begin D(i-1); x := x+h; y := y+h; plot;
            A(i-1); y := y + 2*h; plot;
            C(i-1); x := x-h; y := y+h; plot;
            D(i-1)
      end
end ;
begin startplot;
   i := 0; h := h0 div 4; x0 := 2*h; y0 := 3*h;
   repeat i := i+1; x0 := x0-h;
      h := h div 2; y0 := y0+h;
      x := x0; y := y0; setplot;
      A(i); x := x+h; y := y-h; plot;
```

```
        B(i); x := x−h; y := y−h; plot;
        C(i); x := x−h; y := y+h; plot;
        D(i); x := x+h; y := y+h; plot;
    until i = n;
    endplot
end .
```

Программа 3.2. Кривые Серпинского.

Используя те же примитивы для операций построения, что и в случае кривых Гильберта, приведенную выше рекурсивную схему легко преобразовать в (прямо и косвенно) рекурсивный алгоритм:

```
procedure A(i: integer);
begin if i > 0 then
    begin A(i−1); x := x+h; y := y−h; plot;
        B(i−1); x := x+2*h; plot;
        D(i−1); x := x+h; y := y+h; plot;
        A(i−1)
    end
end
```
(3.23)

Эта процедура соответствует первой строке рекурсивной схемы (3.22). Процедуры, соответствующие фигурам B, C и D, строятся аналогично. Основная программа строится по схеме (3.21). Она должна установить начальные значения для координат рисунка и задать длину единичной линии h в зависимости от формата бумаги, как показано в программе 3.2. Результат работы этой программы при $n = 4$ показан на рис. 3.7. Заметим, что S_0 не рисуется.

Как можно убедиться, в этих примерах рекурсия используется весьма элегантно. Правильность программ легко следует из их структуры и схем построения. Кроме того, использование явного параметра уровня i в соответствии со схемой (3.5) гарантирует окончание программ, так как глубина рекурсии не может быть больше n. По сравнению с этой рекурсивной формулировкой эквивалентные программы, которые избегают явного использования рекурсии, чрезвычайно сложны и трудны для понимания. Мы предлагаем читателю самому в этом убедиться, попытавшись разобраться в программах, приведенных в [3.3].

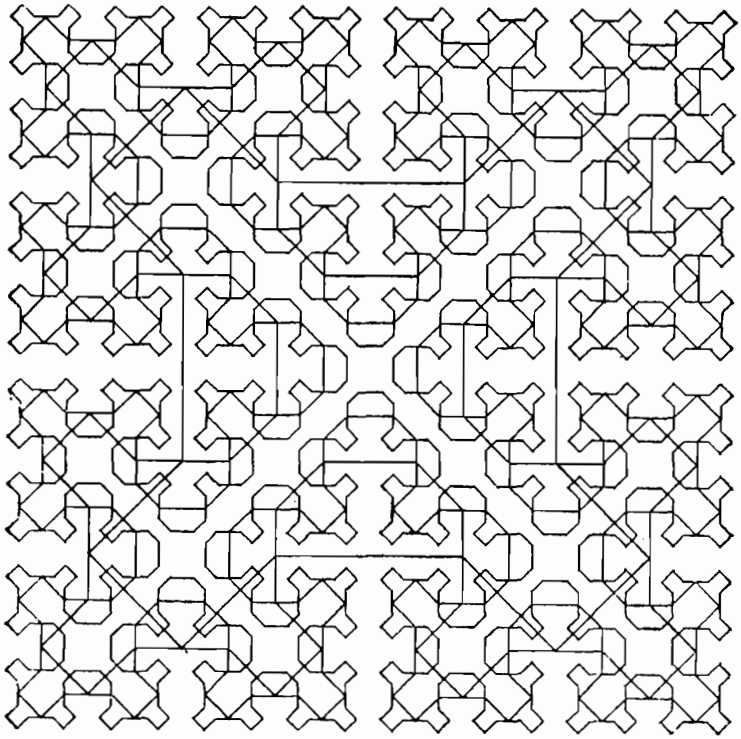


Рис. 3.7. Кривые Серпинского $S_1, \ldots, S_4$.

3.4. АЛГОРИТМЫ С ВОЗВРАТОМ

Особенно интересный раздел программирования — это задачи из области «искусственного интеллекта». Здесь нужно строить алгоритмы, которые находят решение определенной задачи не по фиксированным правилам вычисления, а методом проб и ошибок. Обычно процесс проб и ошибок разделяется на отдельные подзадачи. Часто эти подзадачи наиболее естественно описываются с помощью рекурсии. Процесс проб и ошибок можно рассматривать в общем виде как поисковый процесс, который постепенно строит и просматривает (а также обрезает) дерево подзадач. Во многих случаях такие деревья поиска растут очень быстро, обычно экспоненциально, в зависимости от заданного параметра. Соответственно увеличивается стоимость поиска. Часто дерево поиска можно обрезать, используя только эвристические соображения, и тем самым сводить количество вычислений к разумным пределам.

Здесь мы не собираемся обсуждать общие эвристические правила. Предмет этой главы — общий принцип разбиения таких задач на подзадачи и использование в них рекурсии. Вначале мы продемонстрируем основные принципы на хорошо известном примере — задаче о ходе коня.

Дана доска $n \times n$, содержащая n^2 полей. Конь, который ходит согласно шахматным правилам, помещается на поле с начальными координатами x_0, y_0. Нужно покрыть всю доску ходами коня, т. е. вычислить обход доски, если он существует, из $n^2 - 1$ ходов, такой, что каждое поле посещается ровно один раз.

Очевидно, что задачу покрытия n^2 полей можно свести к более простой: или выполнить очередной ход, или установить, что никакой ход невозможен. Поэтому мы будем строить алгоритм, который пытается сделать очередной ход. Первая попытка выглядит так:

```
procedure попытка следующего хода;
begin инициация выборки ходов;
   repeat выбор следующего возможного хода из списка очередных ходов;
      if он приемлем then
      begin запись хода;
         if доска не заполнена then                                  (3.24)
         begin попытка следующего хода;
            if неудача then стирание предыдущего хода
         end
      end
   until (ход был удачным) ∨ (нет других возможных ходов)
end
```

Если мы хотим более точно описать этот алгоритм, то должны выбрать некоторое представление для данных. Очевидно, что доску можно представить в виде матрицы, скажем, h. Введем также тип индексирующих значений:

```
type index = 1..n;                                                   (3.25)
var h: array[index, index] of integer
```

Так как мы хотим сохранять историю последовательного «захвата» доски, то мы будем представлять каждое поле доски целым числом, а не булевским значением, которое отражало бы просто факт занятия поля. Очевидно, можно остановиться на таких соглашениях:

$h[x, y] = 0$: поле $\langle x, y \rangle$ не посещалось, (3.26)
$h[x, y] = i$: поле $\langle x, y \rangle$ посещалось на i-м ходу $(1 \leqslant i \leqslant n^2)$.

Теперь нужно выбрать подходящие параметры. Они должны определять начальные условия для следующего хода, а

также сообщать о его удаче или неудаче. Первая задача выполняется заданием координат поля x, y, с которого делается ход, а также номером хода i (для его фиксации). Для решения второй задачи нужен булевский параметр-результат: $q = true$ означает удачу, $q = false$ — неудачу.

Какие операторы можно теперь уточнить на основе этих решений? Разумеется, «доска не заполнена» можно выразить как «$i < n^2$». Кроме того, если ввести две локальные переменные u и v для обозначения координат возможного хода, определяемых по правилам хода коня, то предикат «приемлем» можно выразить как логическую конъюнкцию двух условий: чтобы новое поле находилось на доске, т. е. $1 \leqslant u \leqslant n$ и $1 \leqslant v \leqslant n$, и чтобы оно ранее не посещалось, т. е. $h[u, h] = 0$. Фиксация допустимого хода выполняется с помощью присваивания $h[u, v] := i$, а отмена хода (стирание) — как $h[u, v] := 0$. Если при рекурсивном вызове этого алгоритма в качестве параметра-результата передается локальная переменная $q1$, то вместо «ход был удачным» можно подставить $q1$. Таким образом, мы приходим к программе (3.27):

```
procedure try (i: integer; x,y: index; var q: boolean);
var u,v: integer; q1: boolean;
begin инициация выбора ходов;
    repeat пусть u, v — координаты следующего хода,
        определяемого шахматными правилами;
        if (1≤u≤n) ∧ (1≤v≤n) ∧ (h[u,v]=0) then
        begin h[u,v] := i;
            if i < sqr(n) then                                  (3.27)
            begin try(i+1,u,v,q1);
                if ¬q1 then h[u,v] := 0
            end else q1 := true
        end
    until q1 ∨ (нет других ходов);
    q := q1
end
```

Еще один этап уточнения, и мы напишем программу уже полностью на нашем языке программирования. Надо заметить, что до сих пор программа разрабатывалась совершенно независимо от правил хода коня. Мы вполне умышленно откладывали рассмотрение частных особенностей задачи. Но теперь пора обратить на них внимание.

Если задана начальная пара координат $\langle x, y \rangle$, то имеется восемь возможных координат $\langle u, v \rangle$ следующего хода. На рис. 3.8 они пронумерованы от 1 до 8.

Получать u, v из x, y просто — будем прибавлять к ним разности координат, помещенные либо в массиве пар разностей, либо в двух массивах отдельных разностей. Пусть эти массивы обозначены через a и b и соответствующим образом инициированы. Для нумерации следующего возможного хода можно использовать индекс k. Подробности показаны в программе 3.3. Рекурсивная процедура вызывается

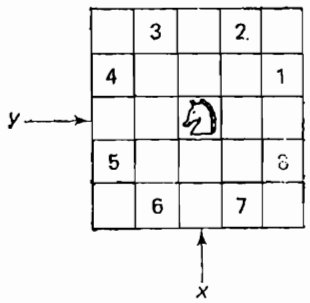


Рис. 3.8. Восемь возможных ходов коня.

в первый раз с параметрами x_0, y_0 — координатами поля, с которого начинается обход. Этому полю присваивается значение 1, остальные поля маркируются как свободные:

$$h[x_0, y_0] := 1; \quad try(2, x_0, y_0, q)$$

Не следует упускать еще одну деталь. Переменная $h[u, v]$ существует лишь в том случае, когда u и v находятся внутри границ массива $1 \ldots n$. Следовательно, выражение в (3.27), подставленное вместо «он приемлем» в (3.24), осмысленно, только если его первые две составляющие истинны. В программе 3.3 это условие подходящим образом переформулировано, кроме того, двойное отношение $1 \leqslant u \leqslant n$ заменено выражением u **in** $[1, 2, \ldots, n]$, которое при достаточно малых n обычно бывает более эффективным (см. разд. 1.10.3). В табл. 3.1 приведены решения, полученные при исходных позициях $\langle 1,1 \rangle$, $\langle 3,3 \rangle$ для $n = 5$ и $\langle 1,1 \rangle$ для $n = 6$.

Параметр-результат q и локальную переменную $q1$ можно заменить глобальной переменной и тем самым несколько упростить программу.

Каким образом теперь можно обобщить этот пример? Какой схеме, типичной для задач подобного рода, он следует? Какие выводы можно сделать, изучая его? Характерная черта этого алгоритма состоит в том, что он предпринимает какие-то шаги по направлению к общему решению, эти шаги фиксируются (записываются), но можно возвращаться обратно и стирать записи, если оказывается, что шаг не приводит к

```
program knightstour (output);
const n = 5; nsq = 25;
type index = 1..n;
var i,j: index;
    q: boolean;
    s: set of index;
    a,b: array [1..8] of integer;
    h: array [index, index] of integer;
procedure try (i: integer; x,y: index; var q: boolean);
    var k,u,v: integer; q1: boolean;
begin k := 0;
    repeat k := k+1; q1 := false;
        u := x + a[k]; v := y + b[k];
        if (u in s)∧(v in s) then
        if h[u,v] = 0 then
        begin h[u,v] := i;
            if i < nsq then
                begin try (i+1,u,v,q1);
                    if ¬q1 then h[u,v] := 0
                end else q1 := true
        end
    until q1 ∨ (k=8);
    q := q1
end {try};
begin s := [1,2,3,4,5];
    a[1] :=  2; b[1] :=  1;
    a[2] :=  1; b[2] :=  2;
    a[3] := -1; b[3] :=  2;
    a[4] := -2; b[4] :=  1;
    a[5] := -2; b[5] := -1;
    a[6] := -1; b[6] := -2;
    a[7] :=  1; b[7] := -2;
    a[8] :=  2; b[8] := -1;
    for i := 1 to n do
        for j := 1 to n do h[i,j] := 0;
    h[1,1] := 1; try(2,1,1,q);
    if q then
        for i := 1 to n do
        begin for j := 1 to n do write(h[i,j]:5);
            writeln
        end
    else writeln(' NO SOLUTION ')
end .
```

Программа 3.3. Ход коня.

Таблица 3.1. Три обхода конем

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

решению, а заводит в «тупик». Такое действие называется *возвратом*. Из схемы (3.24) можно вывести общую схему (3.28), если предположить, что число возможных дальнейших путей на каждом шаге конечно:

```
procedure try;
begin инициировать выборку возможных шагов;
   repeat выбрать следующий шаг;
      if приемлемо then
      begin записать его;
         if решение неполно then                    (3.28)
         begin попробовать очередной шаг;
            if неудачно then стереть запись
         end
      end
   until удача ∨ больше нет путей
end
```

Разумеется, в конкретных программах эта схема может воплощаться различными способами. Часто в них используется явный параметр уровня, обозначающий глубину рекурсии и допускающий простое условие окончания.

Если, кроме того, на каждом шаге число исследуемых дальнейших путей фиксировано, скажем равно m, то используется схема (3.29), вызываемая оператором «*try*(1)»:

```
procedure try(i: integer);
   var k: integer;
begin k:=0;
   repeat k:=k+1; выбрать k-й возможный путь;
      if приемлемо then
      begin записать его;
         if i<n then                                        (3.29)
         begin try(i+1);
            if неудачно then стереть запись
         end
      end
   until удачно ∨ (k=m)
end
```

В остальной части этой главы рассматриваются еще три примера. В них представлены различные воплощения абстрактной схемы (3.29). Эти примеры также призваны иллюстрировать подходящее использование рекурсии.

3.5. ЗАДАЧА О ВОСЬМИ ФЕРЗЯХ

Задача о восьми ферзях — хорошо известный пример использования метода проб и ошибок и алгоритмов с возвратом. В 1850 г. ею занимался К. Ф. Гаусс, но полного ее решения он не дал. Это и не удивительно. Для подобных задач характерно отсутствие аналитического решения. Вместо этого они требуют большого объема изнурительных вычислений, терпения и аккуратности. Поэтому такие задачи стали почти исключительно прерогативой вычислительных машин, которые обладают этими свойствами в гораздо большей степени, чем люди, даже гениальные.

Задача о восьми ферзях поставлена следующим образом (см. также [3.4]): нужно так расставить восемь ферзей на шахматной доске, чтобы ни один ферзь не угрожал другому.

Используя схему (3.29) в качестве образца, мы легко получаем следующую предварительную версию алгоритма:

```
procedure try(i: integer);
begin
   инициировать выбор позиции для i-го ферзя;
   repeat выбрать позицию;
      if безопасно then
      begin поставить ферзя;                                    (3.30)
         if i < 8 then
         begin try(i + 1);
            if неудачно then убрать ферзя
         end
      end
   until удачно ∨ нет больше позиции
end
```

Чтобы идти дальше, нужно выбрать некоторое представление для данных. Поскольку мы знаем, что по шахматным правилам ферзь бьет все фигуры, расположенные на той же горизонтали, вертикали или диагонали доски, то мы заключаем, что каждая вертикаль может содержать одного и только одного ферзя, так что i-го ферзя можно сразу помещать на i-ю вертикаль. Итак, параметр i становится индексом вертикали, а выбор позиции ограничивается восемью возможными значениями индекса горизонтали j.

Осталось решить, как представить расположение восьми ферзей на доске. Очевидно, что доску можно было бы вновь изобразить в виде квадратной матрицы, но после некоторого размышления мы обнаруживаем, что такое представление значительно усложнило бы проверку безопасности позиции. Это крайне нежелательно, поскольку такая операция выполняется наиболее часто. Поэтому нужно выбрать представление, которое насколько возможно упростит эту проверку. Лучше всего сделать наиболее доступной ту информацию, которая действительно важна и чаще всего используется. В нашем случае это не расположение ферзей, а информация о том, помещен ли ферзь на данной горизонтали или диагонали. (Мы уже знаем, что на каждой k-й вертикали уже помещен один ферзь для $1 \leqslant k \leqslant i$.) Это приводит к следующим описаниям переменных:

```
var x : array [1 .. 8] of integer;
    a : array [1 .. 8] of Boolean;                              (3.31)
    b : array [b1 .. b2] of Boolean;
    c : array [c1 .. c2] of Boolean;
```

где

$x[i]$ указывает позицию ферзя на i-й вертикали;
$a[j]$ означает, что на j-й горизонтали нет ферзя;
$b[k]$ означает отсутствие ферзя на k-й ↙-диагонали;
$c[k]$ означает отсутствие ферзя на k-й ↘-диагонали.

Выбор индексных границ $b1$, $b2$, $c1$, $c2$ определяется, исходя из способа, которым вычисляются индексы b и c; мы замечаем, что на ↙-диагонали все поля имеют одну и ту же сумму координат i и j, а на ↘-диагонали постоянна разность координат $i - j$. Соответствующее решение показано в программе 3.4.

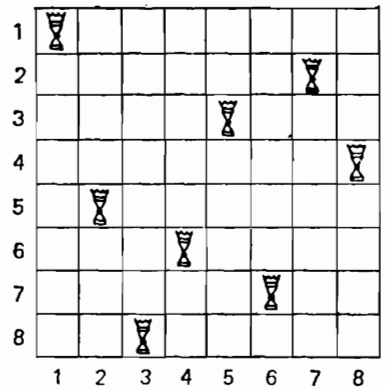


Рис. 3.9. Одно из решений задачи о восьми ферзях.

При таких данных оператор «*поставить ферзя*» принимает следующую форму:

$$x[i] := j; \quad a[j] := false; \quad b[i+j] := false; \quad c[i-j] := false \tag{3.32}$$

При уточнении оператора «*убрать ферзя*» мы получаем

$$a[j] := true; \quad b[i+j] := true; \quad c[i-j] := true \tag{3.33}$$

а условие «*безопасно*» выполняется, если поле $\langle i, j\rangle$ находится на горизонтали и диагонали, которые еще свободны (представлены как *true*), что можно описать логическим выражением

$$a[j] \wedge b[i+j] \wedge c[i-j]. \tag{3.34}$$

Этим завершается разработка алгоритма, который полностью описан в программе 3.4. Она дает решение $x = (1, 5, 8, 6, 3, 7, 2, 4)$, которое изображено на рис. 3.9.

```
program eightqueen1(output);
{поиск одного решения задачи о восьми ферзях}
var i: integer; q: boolean;
    a: array [ 1 .. 8] of boolean;
    b: array [ 2 .. 16] of boolean;
    c: array [-7 .. 7] of boolean;
    x: array [ 1 .. 8] of integer;

procedure try(i: integer; var q: boolean);
    var j: integer;
begin j := 0;
    repeat j := j+1; q := false;
        if a[j] ∧b[i+j] ∧c[i-j] then
        begin x[i] := j;
            a[j] := false; b[i+j] := false; c[i-j] := false;
            if i < 8 then
            begin try (i+1,q);
                if ¬q then
                begin a[j] := true; b[i+j] := true; c[i-j] := true
                end
            end else q := true
        end
    until q ∨ (j=8)
end {try};

begin
    for i :=  1 to  8 do a[i] := true;
    for i :=  2 to 16 do b[i] := true;
    for i := -7 to  7 do c[i] := true;
    try (1,q);
    if q then
        for i := 1 to 8 do write (x[i]: 4);
    writeln
end
```

Программа 3.4. Восемь ферзей (одно решение).

Прежде чем закончить разбор задач, связанных с шахматной доской, мы воспользуемся задачей о восьми ферзях для иллюстрации важного обобщения алгоритма проб и ошибок. В общих словах это обобщение заключается в том, чтобы находить не одно, а все решения поставленной задачи.

К этому обобщению легко перейти. Вспомним, что множество возможных путей строится по строго определенной системе, так чтобы никакой путь не предлагался более одного

раза. Это свойство алгоритма соответствует поиску по дереву, при котором каждый узел посещается только один раз. Это позволяет — после того как решение найдено и должным образом зафиксировано — просто переходить на следующий возможный путь. Общая схема такого алгоритма (3.35) получена из (3.29):

```
procedure try(i: integer);
   var k: integer;
begin
   for k := 1 to m do
   begin выбор k-го пути;
      if приемлемо then                                    (3.35)
      begin запись его
         if i < n then try(i+1) else печать решения;
         стирание записи
      end
   end
end
```

Заметим, что благодаря тому, что условие окончания цикла упростилось до одной составляющей $k = m$, оператор цикла с постусловием естественно заменить на оператор цикла с параметром. К удивлению, поиск *всех* возможных решений описывается более простой программой, чем поиск одного решения.

Обобщенный алгоритм, который находит все 92 решения задачи о восьми ферзях, представлен в программе 3.5. На самом деле существует только 12 принципиально различных решений; наша программа не учитывает симметрию.

Таблица 3.2. Двенадцать решений задачи о восьми ферзях

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

```
program eightqueens (output);
var i: integer;
    a: array [ 1 .. 8] of boolean;
    b: array [ 2 ..16] of boolean;
    c: array [-7 .. 7] of boolean;
    x: array [ 1 .. 8] of integer;
procedure print;
    var k: integer;
begin for k := 1 to 8 do write(x[k]: 4);
    writeln
end {print} ;

procedure try(i: integer);
    var j: integer;
begin
    for j := 1 to 8 do
        if a[j] ∧b[i+j] ∧c[i-j] then
        begin x[i] := j;
            a[j] := false; b[i+j] := false; c[i-j] := false;
            if i < 8 then try(i+1) else print;
            a[j] := true; b[i+j] := true; c[i-j] := true
        end
end {try} ;

begin
    for i :=  1 to  8 do a[i] := true;
    for i :=  2 to 16 do b[i] := true;
    for i := -7 to  7 do c[i] := true;
    try (1)
end
```

Программа 3.5. Восемь ферзей (все решения).

В табл. 3.2 приведены первые 12 решений. Число N указывает частоту проверок безопасности полей. Ее среднее значение для всех 92 решений равно 161.

3.6. ЗАДАЧА ОБ УСТОЙЧИВЫХ БРАКАХ

Пусть даны два непересекающихся множества A и B с одинаковыми кардинальными числами, равными n. Нужно найти некоторое множество пар $\langle a, b\rangle$ из n, такое, чтобы a in A и b in B удовлетворяли некоторым ограничениям. Для выбора таких пар существует много разных критериев; один из них называется «*правилом устойчивых браков*».

Предположим, что A — множество мужчин, а B — множество женщин. Каждый мужчина и каждая женщина устанавливают определенный порядок предпочтения для своих возможных партнеров по браку. Если n пар выбрано таким образом, что существуют какие-то мужчина и женщина, не состоящие в браке друг с другом, но предпочитающие друг друга своим действительным супругам, то такое множество браков считается неустойчивым. Если же такой пары не существует, то множество называется *устойчивым*.

Эта ситуация типична для многих похожих задач, в которых распределение зависит от каких-то предпочтений, как, например, выбор школы учащимися, выбор рекрутами различных родов войск и т. д. Пример с браками отчасти выбран интуитивно; заметим, однако, что установленный порядок предпочтений инвариантен и не изменяется по мере образования пар. Это упрощает задачу, но и несколько искажает действительность (как любая абстракция).

Решение можно искать следующим образом: пробовать последовательно объединять в пары члены двух множеств, пока оба множества не будут исчерпаны. Намереваясь найти *все* устойчивые распределения, мы можем легко набросать решение, используя в качестве образца схему программы (3.35). Пусть $try(m)$ — алгоритм поиска партнерши для мужчины m, и пусть поиск происходит согласно списку предпочтений этого мужчины. Приведем первую версию, основанную на этих допущениях:

```
procedure try(m: man);
   var r: rank;
begin
   for r := 1 to n do
   begin выбрать r-ю женщину из списка предпочтений мужчины m;
      if приемлемо then
      begin записать брак;                                    (3.36)
        if m — не последний мужчина then try(succ(m))
           else записать устойчивое множество;
        отменить брак
      end
   end
end
```

Вновь мы не можем двигаться дальше, пока не решим, как представлять данные. Определим три скалярных типа; для простоты пусть их значения будут целыми числами от 1 до n. Хотя формально эти три типа одинаковы, присваивание им различных имен значительно проясняет программу.

В частности, сразу понятно, что означает какая-либо переменная:

$$\begin{aligned}&\textbf{type } man = 1..n;\\ &\quad woman = 1..n;\\ &\quad rank = 1..n;\end{aligned} \tag{3.37}$$

Исходные данные представляются двумя матрицами, в которых указан порядок предпочтений мужчин и женщин их партнерами:

$$\begin{aligned}&\textbf{var } wmr\text{: } \textbf{array}[man, rank]\ \textbf{of}\ woman\\ &\quad mwr\text{: } \textbf{array}[woman, rank]\ \textbf{of}\ man\end{aligned} \tag{3.38}$$

$wmr[m]$ обозначает список предпочтений, установленный мужчиной m, т. е. $wmr[m][r] = wmr[m, r]$ — женщина, которая занимает r-е место в списке предпочтений мужчины m. Точно так же $mwr[w]$ — список предпочтений женщины w, а $mwr[w, r]$ — мужчина, занимающий r-е место в ее списке.

Результат представляется в виде массива женщин x, такого, что $x[m]$ обозначает партнершу мужчины m. Для сохранения симметрии (называемой также «равноправием») между мужчинами и женщинами, вводится дополнительный массив y, такой, что $y[w]$ обозначает партнера женщины w:

$$\begin{aligned}&\textbf{var } x\text{: } \textbf{array}[man]\ \textbf{of}\ woman;\\ &\quad y\text{: } \textbf{array}[woman]\ \textbf{of}\ man;\end{aligned} \tag{3.39}$$

Ясно, что в массиве y нет особой нужды, поскольку он содержит информацию, которая уже представлена в массиве x. В самом деле, для любых m и w, состоящих между собой в браке, выполняются равенства

$$x[y[w]] = w, \quad y[x[m]] = m. \tag{3.40}$$

Следовательно, значение $y[w]$ можно установить просто поиском по x; но использование массива y явно повышает эффективность. Информация, представленная в массивах x и y, нужна для определения устойчивости предлагаемого множества браков. Поскольку это множество строится постепенно, путем выбора отдельных пар и проверки устойчивости множества после каждого такого предлагаемого брака, x и y используются еще до того, как будут заполнены. Для того чтобы знать, какие компоненты уже определены, можно ввести булевские массивы

$$\begin{aligned}&singlem\text{: } \textbf{array}[man]\ \textbf{of}\ boolean\\ &singlew\text{: } \textbf{array}[woman]\ \textbf{of}\ boolean\end{aligned} \tag{3.41}$$

с такими значениями:

¬singlem [m] предполагает, что x [m] определено,

¬singlew [w] предполагает, что y [w] определено.

Но, рассматривая предлагаемый алгоритм, легко обнаружить, что семейное положение мужчины можно определить просто по значению m следующим образом:

$$\neg singlem\,[k] \equiv k < m. \tag{3.42}$$

Поэтому массив *singlem* можно удалить; соответственно мы упростим имя *singlew* до *single*.

После соответствующих соглашений алгоритм принимает вид (3.43). Предикат «*приемлемо*» можно изобразить в виде конъюнкции *single* и *stable* («*устойчивый*»), где *stable* — функция, которую еще надо будет уточнить.

```
procedure try(m: man);
   var r: rank;  w: woman;
begin for r := 1 to n do
   begin w := wmr[m,r];
      if single[w] ∧ stable then
      begin x[m] := w;  y[w] := m;  single[w] := false;
         if m < n then try(succ(m))
            else запись устойчивого множества;
         single[w] := true
      end
   end
end
```
(3.43)

Здесь все еще заметно большое сходство с программой 3.5.

Теперь основная задача — уточнить алгоритм определения устойчивости. К сожалению, устойчивость нельзя представить таким же простым выражением, как безопасность позиции ферзя в программе 3.5. Нужно прежде всего иметь в виду, что устойчивость по определению следует из сравнения предпочтений, или рангов. Но ранги мужчин и женщин нигде в наших описанных до сих пор данных явно не представлены. Конечно, ранг женщины w с точки зрения мужчины m можно вычислить, но лишь с помощью трудоемкого поиска w в $wmr[m]$.

Так как вычисление устойчивости — очень частое действие, желательно, чтобы эта информация была более доступна. Для этого мы будем использовать две матрицы

```
rmw: array[man, woman] of rank;
rwm: array[woman, man] of rank;
```
(3.44)

такие, что $rmw[m, w]$ означает ранг w-й женщины в списке предпочтений m-го мужчины, а $rwm[w, m]$ — ранг m-го мужчины в списке w-й женщины. Ясно, что значения этих вспомогательных массивов постоянны и могут быть получены с самого начала из значений wmr и mwr.

Значение предиката *stable* (*устойчивый*) теперь вычисляется в строгом соответствии с его исходным определением. Вспомним, что мы исследуем возможность брака между m и w, где $w = wmr[m, r]$, т. е. w имеет ранг r в m-м списке предпочтений. Будучи оптимистами, мы вначале предполагаем, что устойчивость сохранилась, а затем пытаемся найти возможные источники неприятностей. Где они могут таиться? Есть две симметричные возможности:

1. Может существовать женщина pw, предпочтительная для m по сравнению с w, которая сама предпочитает m своему мужу.
2. Может существовать мужчина pm, предпочтительный для w по сравнению с m, который сам предпочитает w своей жене.

Исследуя источник неприятностей 1, мы сравниваем ранги $rwm[pw, m]$ и $rwm[pw, y[pw]]$ для всех женщин, которых m предпочитает своей невесте w, т. е. для всех $pw = wmr[m, i]$, таких, что $i < r$. Мы знаем, что все эти женщины уже выданы замуж, так как если бы какая-то из них была еще одинока, m выбрал бы ее раньше, чем w. Этот процесс проверки можно сформулировать в виде простого линейного поиска; s означает устойчивость:

```
s := true; i := 1;
while (i < r) ∧ s do
   begin pw := wmr[m,i]; i := i+1;                    (3.45)
      if ¬single[pw] then s := rwm[pw, m] > rwm[pw, y[pw]]
   end
```

Исследуя источник неприятностей 2, мы должны рассмотреть всех мужчин pm, которых w предпочитает своему предполагаемому партнеру m, т. е. всех $pm = mwr[w, i]$, таких, что $i < rwm[w, m]$. Как и при исследовании источника 1, нужно сравнивать ранги $rmw[pm, w]$ и $rmw[pm, x[pm]]$. Но здесь нужно быть внимательными, чтобы не производить сравнений с участием $x[pm]$, где pm еще не женат. Необходимой предосторожностью будет проверка $pm < m$, поскольку мы знаем, что все мужчины, предшествующие m, уже женаты.

Весь алгоритм представлен в программе 3.6. Табл. 3.3 содержит множество входных данных, соответствующих масси-

Таблица 3.3. Пример входных данных для задачи об устойчивых браках

Ранг		1	2	3	4	5	6	7	8
Мужчина	1 выбирает женщину	7	2	6	5	1	3	8	4
	2	4	3	2	6	8	1	7	5
	3	3	2	4	1	8	5	7	6
	4	3	8	4	2	5	6	7	1
	5	8	3	4	5	6	1	7	2
	6	8	7	5	2	4	3	1	6
	7	2	4	6	3	1	7	5	8
	8	6	1	4	2	7	5	3	8
Женщина	1 выбирает мужчину	4	6	2	5	8	1	3	7
	2	8	5	3	1	6	7	4	2
	3	6	8	1	2	3	4	7	5
	4	3	2	4	7	6	8	5	1
	5	6	3	1	4	5	7	2	8
	6	2	1	3	8	7	4	6	5
	7	3	5	7	2	4	1	8	6
	8	7	2	8	4	5	6	3	1

вам wmr и mwr. И наконец, в табл. 3.4 приведены девять полученных решений.

Таблица 3.4. Решения задачи об устойчивых браках

		x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	rm	rw	c^*
Решение	1	7	4	3	8	1	5	2	6	16	32	21
	2	2	4	3	8	1	5	7	6	22	27	449
	3	2	4	3	1	7	5	8	6	31	20	59
	4	6	4	3	8	1	5	7	2	26	22	62
	5	6	4	3	1	7	5	8	2	35	15	47
	6	6	3	4	8	1	5	7	2	29	20	143
	7	6	3	4	1	7	5	8	2	38	13	47
	8	3	6	4	8	1	5	7	2	34	18	758
	9	3	6	4	1	7	5	8	2	43	11	34

*c = количество проверок устойчивости.
Решение 1 = решение, оптимальное для мужчин.
Решение 9 = решение, оптимальное для женщин.

По своей сути этот алгоритм основан на простой схеме поиска с возвратом. Его эффективность зависит в основном от удачного построения схемы усекания дерева решения. Несколько более быстрый, но более сложный и менее понятный алгоритм предложен Мак-Вити и Уилсоном [3.1, 3.2], которые, кроме того, распространили его на случай множеств (мужчин и женщин) неодинакового размера.

```
program marriage (input,output);
{задача о стабильных браках}
const n = 8;
type man = 1..n; woman = 1..n; rank = 1..n;
var m: man; w: woman; r: rank;
    wmr: array [man, rank] of woman;
    mwr: array [woman, rank] of man;
    rmw: array [man, woman] of rank;
    rwm: array [woman, man] of rank;
    x:   array [man] of woman;
    y:   array [woman] of man;
    single: array [woman] of boolean;

procedure print;
    var m: man; rm, rw: integer;
begin rm := 0; rw := 0;
    for m := 1 to n do
    begin write (x[m]:4);
        rm := rm + rmw[m,x[m]]; rw := rw + rwm[x[m],m]
    end ;
    writeln (rm:8,rw:4);
end {print} ;

procedure try(m: man);
    var r: rank; w: woman;

    function stable: boolean;
        var pm: man; pw: woman;
            i, lim: rank;  s: boolean;
    begin s := true; i := 1;
        while (i<r) ∧ s do
        begin pw := wmr[m,i]; i := i+1;
            if ¬single[pw] then s := rwm[pw,m]  > rwm[pw,y[pw]]
        end ;
        i := 1; lim := rwm[w,m];
        while (i<lim) ∧ s do
        begin pm := mwr[w,i]; i := i+1;
            if pm < m then s := rmw[pm,w] > rmw[pm,x[pm]]
        end ;
        stable := s
    end {stable} ;
begin {try}
    for r := 1 to n do
    begin w := wmr[m,r];
```

```
        if single[w] then
            if stable then
            begin x[m] := w; y[w] := m; single[w] := false;
                if m < n then try(succ(m)) else print;
                single[w] := true
            end
    end
end {try} ;
begin {основная программа}
    for m := 1 to n do
        for r := 1 to n do
        begin read(wmr[m,r]); rmw[m,wmr[m,r]] := r
        end ;
    for w := 1 to n do
        for r := 1 to n do
        begin read(mwr[w,r]); rwm[w,mwr[w,r]] := r
        end ;
    for w := 1 to n do single[w] := true;
    try (1)
end .
```

Программа 3.6. Устойчивые браки.

Алгоритмы такого вида, как в двух последних примерах, дающие все возможные решения задачи (при определенных ограничениях), часто используются для выбора одного или нескольких решений, которые в каком-то смысле являются оптимальными. В данном примере можно было бы, допустим, интересоваться решением, которое в среднем больше удовлетворяет мужчин, или женщин, или всех.

Отметим, что в табл. 3.4 указаны суммы рангов всех женщин с точки зрения их мужей и суммы рангов всех мужчин с точки зрения их жен. Это значения

$$rm = \sum_{m=1}^{n} rmw\,[m,\ x\,[m]\,], \quad rw = \sum_{m=1}^{n} rwm\,[x\,[m],\ m]. \quad (3.46)$$

Решение с наименьшим значением rm называется устойчивым решением, оптимальным для мужчин, а с наименьшим rw — устойчивым решением, оптимальным для женщин. При избранной стратегии поиска первыми вычисляются решения, хорошие с точки зрения мужчин, а благоприятные для женщин решения даются в конце работы. В этом смысле алгоритм ориентирован на сильный пол. Но его можно быстро

изменить, систематически поменяв ролями мужчин и женщин, т. е. заменив *mwr* на *wmr* и *rmw* на *rwm*.

Здесь мы не будем далее обобщать эту программу, а оставим поиск оптимального решения в качестве очередного и последнего примера алгоритма с возвратом.

3.7. ЗАДАЧА ОПТИМАЛЬНОГО ВЫБОРА

Наш последний пример алгоритма с возвратом — это логическое обобщение двух предыдущих алгоритмов, которые строятся по общей схеме (3.35). Вначале мы использовали принцип возврата для нахождения *одного* решения данной задачи. Это было показано на примерах хода коня и задачи о восьми ферзях. Затем мы задались целью найти *все* решения задачи; примерами были восемь ферзей и устойчивые браки. Теперь мы хотим найти *оптимальное решение*.

Для этого нужно получить все возможные решения и в процессе их получения оставлять только то, которое в некотором смысле является оптимальным. Если предположить, что оптимальность определена с помощью некоторой функции $f(s)$, принимающей положительные значения, то алгоритм можно получить из схемы (3.35) заменой оператора «*печать решения*» на оператор

$$\textbf{if } f(solution) > f(optimum) \textbf{ then } optimum := solution \quad (3.47)$$

В переменной *optimum* хранится лучшее из полученных до сих пор решений. Разумеется, ее нужно должным образом инициировать; кроме того, значение $f(optimum)$ принято хранить в другой переменной, чтобы избежать ее частого перевычисления.

В качестве примера общей задачи поиска оптимального решения мы выбираем следующую важную и часто встречающуюся задачу: найти *оптимальную выборку* из заданного множества объектов, подчиненную некоторым ограничениям. Выборки, составляющие приемлемые решения, строятся постепенно, с помощью исследования отдельных объектов базового множества. Процедура *try* описывает процесс исследования пригодности объекта для включения в выборку; она вызывается рекурсивно при переходе к следующему объекту, пока все объекты не будут рассмотрены.

Мы видим, что из рассмотрения каждого объекта можно сделать два возможных вывода: либо включить объект в текущую выборку, либо не включать его. Поэтому здесь не удастся использовать операторы цикла; вместо этого нужно явно описать оба случая. Это показано в (3.48); предполо-

жим, что объекты пронумерованы 1, 2, ..., n:

```
procedure try(i: integer);
begin
1: if включение приемлемо then
   begin включить i-й объект;
      if i < n then try(i + 1) else проверить оптимальность;
      удалить i-й объект                                    (3.48)
   end;
2: if невключение приемлемо then if i < n then try(i + 1)
   else проверить оптимальность
end
```

Из этой схемы видно, что всего имеются 2^n возможные выборки; поэтому ясно, что критерии приемлемости должны значительно ограничить количество рассматриваемых возможностей. Для пояснения возьмем конкретный пример: пусть каждый из n объектов $a_1, \ldots, a_n$ обладает весом w и ценностью v. Оптимальным пусть считается множество с наибольшей суммарной ценностью компонент, а ограничением пусть служит их предельный общий вес. Эта задача хорошо знакома всем отправляющимся в путешествие, которые упаковывают чемоданы, стараясь так выбрать n предметов, чтобы их общая ценность была максимальной, а общий вес не превышал какого-то допустимого предела.

Теперь мы можем решить, как представить известные факты в виде данных. Из вышесказанного легко получаются такие описания:

```
type index = 1..n;
     object = record w,v: integer end
var a: array [index] of object;                             (3.49)
    limw, totv, maxv: integer;
    s, opts: set of index
```

Переменные *limw* и *totv* обозначают предельный вес и общую ценность всех n объектов. Фактически эти два значения в течение всего процесса отбора постоянны; через s обозначается текущая выборка объектов, где каждый объект представлен своим именем (индексом); *opts* есть оптимальная выборка, полученная до сих пор, а *maxv* — ее ценность.

Теперь посмотрим, каковы критерии приемлемости объекта для текущей выборки. Когда речь идет о *включении*, объект можно включить в выборку, если он удовлетворяет допустимому весу. Если же не удовлетворяет, то можно прекратить попытки добавлять новые объекты в текущей выборке. Но если рассматривать *невключение*, то критерием приемлемости, т. е. возможности продолжать построение текущей

```
program selection (input,output);
{поиск оптимальной выборки объектов при ограничениях}
const n = 10;
type index = 1..n;
   object = record v,w: integer end;
var i: index;
   a: array [index] of object;
   limw, totv, maxv: integer;
   w1, w2, w3: integer;
   s, opts: set of index;
   z: array [boolean] of char;

procedure try(i: index; tw,av: integer);
   var av1: integer;
begin {попытка включения объекта i}
   if tw + a[i].w ≤ limw then
   begin s := s + [i];
      if i < n then try(i+1, tw+a[i].w, av) else
         if av > maxv then
         begin maxv := av; opts := s
         end ;
      s := s - [i]
   end ;
   {попытка исключения объекта i} av1 := av - a[i].v;
   if av1 > maxv then
   begin if i < n then try(i+1, tw, av1) else
            begin maxv := av1; opts := s
            end
   end
end {try} ;
begin totv := 0;
   for i := 1 to n do
      with a[i] do
      begin read(w,v); totv := totv + v
      end ;
   read(w1,w2,w3);
   z[true] := '*'; z[false] := ' ';
   write(' WEIGHT   ');
   for i := 1 to n do write (a[i].w: 4);
   writeln; write (' VALUE   ');
   for i := 1 to n do write (a[i].v: 4);
   writeln;
   repeat limw := w1; maxv := 0; s := [ ]; opts := [ ];
```

```
        try(1,0,totc);
        write(limw);
        for i := 1 to n do write('     ', z[i in opts]);
        writeln; w1 := w1 + w2
    until w1 > w3
end
```

Программа 3.7. Оптимальная выборка.

выборки, будет возможность получить без этого объекта такую общую ценность выборки, которая была бы не меньше полученного до сих пор оптимума. Ведь иначе продолжение поиска, хотя и будет давать какие-то решения, никогда не приведет к оптимальному, следовательно, на этом пути бесполезен какой-либо дальнейший поиск. С учетом этих двух условий мы определяем, какие существенные величины нужно вычислять для каждого шага в процессе отбора:

1. Общий вес *tw* выборки *s*, полученной до сих пор.
2. Общую ценность *av* текущей выборки *s*, которой еще можно достичь.

Эти два значения удобно представить в виде параметров процедуры *try*.

Условие «*включение приемлемо*» в (3.48) теперь можно сформулировать как

$$tw + a[i].w \leqslant limw \tag{3.50}$$

а последующую проверку оптимальности — как

```
if av > maxv then
   begin {запись нового оптимума}
      opts := s; maxv := av
   end
```
(3.51)

Последнее присваивание связано с тем соображением, что достижимое значение будет получено после просмотра всех *n* объектов.

Условие «*невключение приемлемо*» в (3.48) выражается как

$$av - a[i].v > maxv \tag{3.52}$$

Так как позднее оно снова используется, значение $av - a[i].v$ присваивается переменной *av1*, чтобы избежать повторного вычисления.

Всю программу полностью теперь можно получить из (3.48) с помощью (3.52), добавив соответствующие операторы инициации глобальных переменных. Следует отметить, что здесь удобно используются операции над множествами.

Результат выполнения программы 3.7 с заданными предельными весами от 10 до 120 показан в табл. 3.5.

Таблица 3.5. Пример результата работы программы оптимальной выборки

Вес	10	11	12	13	14	15	16	17	18	19
Значение	18	20	17	19	25	21	27	23	25	24
10	*									
20							*			
30					*		*			
40	*				*		*			
50	*	*		*			*			
60	*	*	*	*	*					
70	*	*			*		*		*	
80	*	*	*		*		*	*		
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*

УПРАЖНЕНИЯ

3.1. (Ханойские башни.) Даны три стержня и n дисков разного размера. Диски можно надевать на стержни, строя таким образом «башни». Пусть вначале диски находятся на стержне A в порядке убывающего размера, как показано на рис. 3.10 для $n = 3$. Нужно переместить n

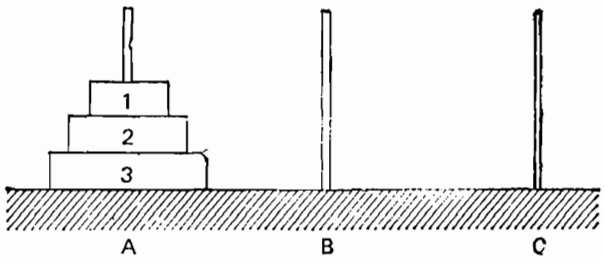


Рис. 3.10. Ханойские башни.

дисков на стержень C так, чтобы они остались в том же порядке. Этого нужно добиться, соблюдая следующие правила:

1. На каждом шаге ровно один диск перемещается с одного стержня на другой.
2. Диск большего размера нельзя помещать на меньший.
3. Стержень B можно использовать в качестве промежуточного.

Постройте алгоритм, который выполняет эту задачу. Заметим, что башню удобно рассматривать как состоящую из одного диска на самом верху и из башни, состоящей из остальных дисков. Опишите этот алгоритм в виде рекурсивной программы.

3.2. Напишите процедуру, которая формирует все $n!$ перестановок n элементов $a_1, \ldots, a_n$ *in situ*, т. е. без использования другого массива. Получив очередную перестановку, вызвать параметрическую процедуру Q, которая может, например, выводить полученную перестановку.

Указание. Рассматривайте задачу получения всех перестановок элементов $a_1, \ldots, a_m$ как состоящую из m подзадач, строящих все перестановки $a_1, \ldots, a_{m-1}$, за которыми следует a_m, и так, что в i-й подзадаче сначала меняются местами два элемента a_i и a_m.

3.3. Определите рекурсивную схему для рис. 3.11, который представляет собой наложение четырех кривых W_1, W_2, W_3, W_4. Эта структура сход-

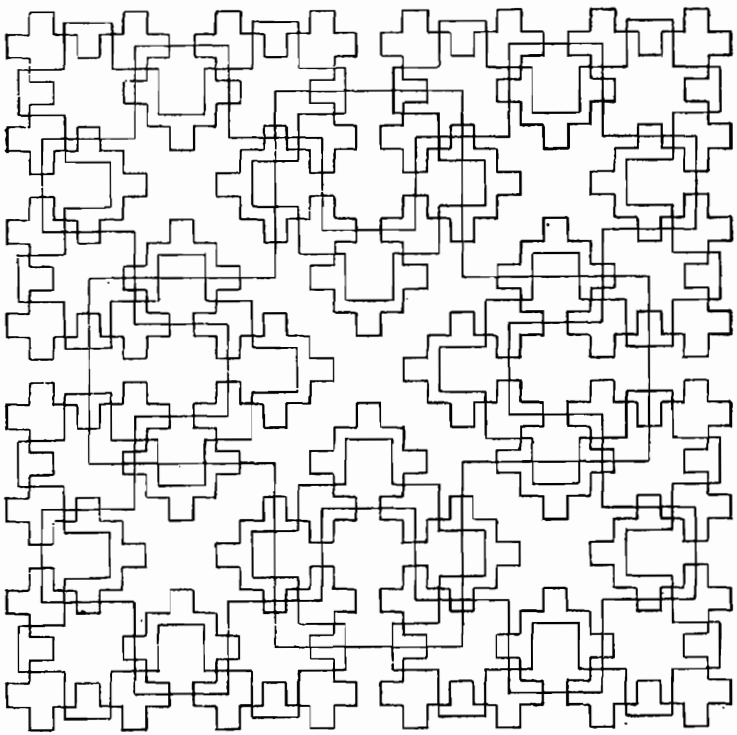


Рис. 3.11. W-кривые порядка 1—4.

на с кривыми Серпинского (3.21) и (3.22). На основе рекурсивной схемы получите рекурсивную программу, которая чертит эти кривые.

3.4. Лишь 12 из 92 решений, вычисляемых программой восьми ферзей существенно различны. Остальные можно получить с помощью осевой или центральной симметрий. Придумайте программу, которая находит 12 основных решений. Заметим, что, например, поиск в вертикали 1 можно ограничить позициями 1—4.

3.5. Измените программу устойчивых браков так, чтобы она находила оптимальное решение (для мужчин и для женщин). Следовательно, она станет программой, работающей по принципу «ветвления с ограничением» подобно программе 3.7.

3.6. Некоторая железнодорожная компания обслуживает n станций $S_1, \ldots, S_n$. Она предполагает улучшить информационное обслуживание клиентов с помощью информационных терминалов, управляемых вычислительной машиной. Клиент набирает название своей станции отправления S_A и станции назначения S_B, и ему должна выдаваться схема расписаний поездов с минимальным общим временем поездки.

Разработайте программу для вычисления нужной информации. Пусть расписание (представляющее собой ваш банк данных) изображено соответствующей структурой данных, содержащей время отправления (= прибытия) всех имеющихся поездов. Разумеется, не все станции связаны непрерывными линиями (см. также упр. 1.8).

3.7. Функция Аккермана A определена для всех неотрицательных целых аргументов m и n следующим образом:

$$A(0,n) = n + 1$$
$$A(m,0) = A(m-1,1) \quad (m > 0)$$
$$A(m,n) = A(m-1,A(m,n-1)) \quad (m,n > 0)$$

Разработайте программу, которая вычисляет $A(m, n)$ без использования рекурсии.

В качестве образца используйте программу 2.11 — нерекурсивную версию быстрой сортировки. Определите множество правил для общего случая преобразования рекурсивной программы в итеративную.

ЛИТЕРАТУРА

3.1. McVITIE D. G., WILSON L. B. The Stable Marriage Problem. — *Comm. ACM*, **14**, No. 7, 1971, 486—492.
3.2. McVITIE D. G., WILSON L. B. Stable Marriage Assignment for Unequal Sets. — *BIT*, **10**, 1970, 295—309.
3.3. Spase Filling Curves, or How to Waste Time on a Plotter. — *Software — Practice and Experience*, **1**, No. 4, 1971, 403—440.
3.4. WIRTH H. Program Development by Stepwise Refinement. — *Comm. ACM*, **14**, No. 4, 1971, 221—227.

4

ДИНАМИЧЕСКИЕ ИНФОРМАЦИОННЫЕ СТРУКТУРЫ

4.1. РЕКУРСИВНЫЕ ТИПЫ ДАННЫХ

В гл. 1 мы определили фундаментальные структуры данных: массив, запись и множество. Эти структуры называются фундаментальными, так как, во-первых, они представляют собой строительные блоки, из которых формируются более сложные структуры, и, во-вторых, они чаще встречаются на практике. Цель описания типа данных и последующего определения некоторых переменных как относящихся к этому типу состоит в том, чтобы зафиксировать раз и навсегда размер значений, которые могут присваиваться этим переменным, и соответственно размер выделяемой для них памяти. Поэтому описанные таким образом переменные называются *статическими*. Однако многие задачи требуют более сложных информационных структур. Для таких задач характерно, что используемые в них структуры изменяются во время выполнения. Поэтому они называются *динамическими* структурами. Разумеется, на каком-то уровне детализации компоненты этих структур являются статическими, т. е. относятся к одному из фундаментальных типов данных. Эта глава посвящена конструированию и анализу динамических информационных структур, а также работе с ними.

Примечательно, что существуют некоторые близкие аналоги между методами структурирования алгоритмов и методами структурирования данных. Как и при любых аналогиях, остаются некоторые различия (иначе мы имели бы дело с идентичностью), тем не менее сравнение методов структурирования программ и данных полезно для их понимания.

Элементарным, неструктурированным оператором является присваивание. Соответствующий ему тип даных — скалярный, неструктурированный. Оба они являются атомарными строительными блоками для составных операторов и типов данных. Простейшие структуры, получаемые с помощью перечисления, или следования, — это составной оператор и запись. Оба состоят из конечного (обычно небольшого) количества явно перечисляемых компонент, которые могут различаться. Если все компоненты одинаковы, их не нужно выписывать отдельно: для того чтобы описать повторения, число которых известно и конечно, мы пользуемся

оператором цикла с параметром (**for**) и массивом. Выбор из двух или более вариантов выражается условным или выбирающим оператором и соответственно записью с вариантами. И наконец, повторение неизвестное (потенциально бесконечное) количество раз выражается оператором цикла с предусловием (**while**) или с постусловием (**repeat**). Соответствующая структура данных — последовательность (файл) — простейший вид структуры, допускающей построение типов с бесконечным кардинальным числом.

Возникает вопрос: а существует ли структура данных, которая подобным же образом соответствует оператору процедуры? Разумеется, наиболее интересная и новая по сравнению с другими операторами особенность процедур — это возможность *рекурсии*. Значения типа данных, который можно назвать рекурсивным, должны содержать одну или более компонент того же типа, что и все значение, по аналогии с процедурой, содержащей один или более вызовов самой себя. Как и в процедурах, в таких определениях типов рекурсия может быть прямой или косвенной.

Простой пример объекта, тип которого можно определить рекурсивно, — арифметическое выражение, встречающееся в языках программирования. Рекурсия отражает возможность вложенности, т. е. использования заключенных в скобки подвыражений в качестве операндов в выражении. Итак, пусть выражение здесь определяется неформально следующим образом:

> *Выражение* состоит из терма, за которым следует знак операции, за которым следует терм. (Эти два терма являются операндами соответствующей операции.) *Терм* — это либо переменная, представленная идентификатором, либо выражение, заключенное в скобки.

Тип данных, значениями которого являются подобные выражения, легко можно описать с помощью известного уже средства — рекурсии *).

```
type expression = record op: operator;
                         opd1, opd2: term,
                  end;
type term = record                                    (4.1)
              if t then (id: alfa)
                   else (subex: expression)
            end
```

*) Описание типа *term* в (4.1) не принадлежит языку Паскаль: в нем не используется конструкция **if** ... **then** ... **else** для выражения варианта в записи. Более правильной с точки зрения Паскаля была бы формули-

Итак, каждая переменная типа *term* состоит из двух компонент: поля признака t и, если t истинно, поля *id*, иначе —

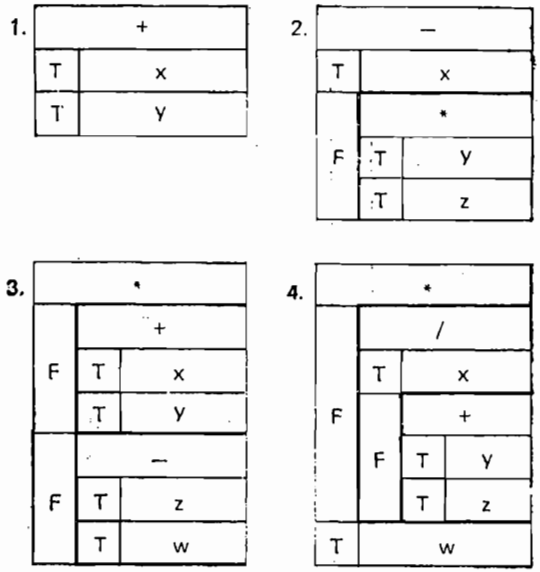


Рис. 4.1. Расположение в памяти рекурсивных записей.

поля *subex*. Теперь рассмотрим в качестве примеров следующие четыре выражения:

$$\begin{aligned} &1.\ x + y \\ &2.\ x - (y * z) \\ &3.\ (x + y) * (z - w) \\ &4.\ (x/(y + z)) * w \end{aligned} \tag{4.2}$$

Эти выражения можно представить, как показано на рис. (4.1), на котором наглядно видна их вложенная, рекур-

---

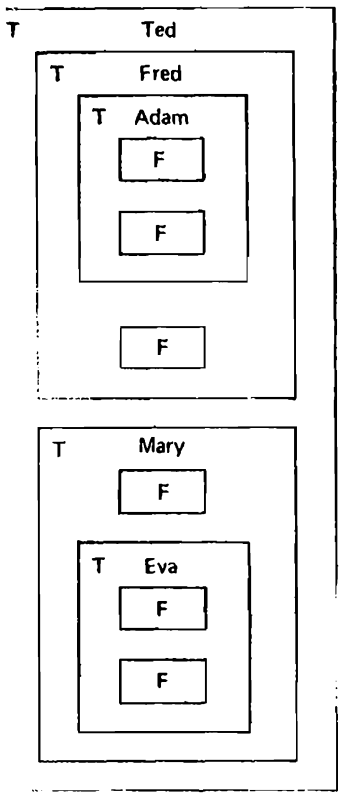
ровка:

```
type term = record
              case t: Boolean of
              true: (id: alfa)
              false: (subex: expression)
            end
```

Однако «зацикливание» в описаниях типов: *term* определяется через *expression*, *a expression* через *term* — обычно также запрещается. Таким образом, подобного рода рекурсивные описания типов в Паскале не применяются. Для построения динамических структур в нем используется аппарат динамического размещения переменных и ссылок, описанный ниже. — *Прим. перев.*

ивная структура. Эти схемы определяют также размещение тих выражений в памяти.

Другой пример рекурсивной информационной структуры — енеалогическое дерево. Пусть генеалогическое дерево состоит из имени человека и двух генеалогических деревьев его родителей. Такое определение неизбежно приводит к бесконечной структуре. Реальные генеалогические деревья конечны, потому что на каком-то уровне сведения о предках отсутствуют, Если мы вновь используем рекурсивную структуру, как показано в (4.3), то этот факт мы можем учесть:



ис. 4.2. Структура генеалогического дерева.

```
type ped = record
             if known then
               (name: alfa;
                father, mother: ped)
           end
```

(4.3)

(Заметим, что каждая переменная типа *ped* (от *pedigrec* — «генеалогическое дерево». — *Перев.*) содержит по крайней мере одну компоненту — поле признака *known* («*известен*»). Если его значение истинно, то имеются еще три поля; иначе больше нет полей.) Отдельное значение, обозначенное (рекурсивным) конструктором записи

$$x = (T,Ted,(T,Fred,(T,Adam,(F),(F)),F)),\\ (T,Mary,(F),(T,Eva,(F),(F)))$$

зображено на рис. 4.2, причем, глядя на рисунок, можно акже сделать выводы о возможном расположении данных памяти. (Так как рассматривается только одно определение ипа, мы пропускаем идентификатор типа *ped* перед каждым онструктором.)

Здесь очевидна важная роль вариантов: это единственное редство, позволяющее сделать рекурсивную структуру коечной, поэтому в рекурсивном определении всегда участвуют

варианты. В этом особенно наглядно прослеживается аналогия между структурами программ и данных. Каждая рекурсивная процедура также должна обязательно содержать условный оператор, чтобы ее выполнение могло когда-нибудь закончиться. Ясно, что окончание выполнения для процедуры соответствует конечности кардинального числа для типа данных.

4.2. ССЫЛКИ ИЛИ УКАЗАТЕЛИ

Характерная особенность рекурсивных структур, которая отличает их от основных структур (массивов, записей, множеств), — их способность изменять размер. Поэтому для рекурсивно определенных структур невозможно установить фиксированный размер памяти, и поэтому транслятор не может приписать компонентам такой переменной определенные адреса. Для решения этой проблемы чаще всего применяется метод *динамического распределения памяти*, т. е. выделения памяти для отдельных компонент в тот момент, когда они появляются во время выполнения программы, а не во время трансляции. В этом случае транслятор выделяет фиксированный объем памяти для хранения *адреса* динамически размещаемой компоненты, а не самой компоненты. Например, генеалогическое дерево, изображенное на рис. 4.2, можно представить в виде отдельных, вполне возможно, не расположенных рядом в памяти записей — по одной для каждого человека. Эти записи связываются с помощью адресов, находящихся в соответствующих полях *father* («*отец*») и *mother* («*мать*»). Графически это лучше всего изобразить стрелками (см. рис. 4.3).

Следует подчеркнуть, что использование ссылок для реализации рекурсивных структур — чисто технический прием. Программисту необязательно знать об их существовании. Память может выделяться автоматически, как только упоминается новая переменная. Однако если использование ссылок или указателей сделать явным, то можно строить более разнообразные структуры данных, чем те, которые можно задать лишь с помощью рекурсивных определений. В частности, в этом случае можно определять «бесконечные», или циклические, структуры. Кроме того, на одну и ту же подструктуру можно ссылаться из разных мест, т. е. относить ее к нескольким разным структурам. Поэтому в современных языках программирования принято обеспечивать явное манипулирование не только данными, но и ссылками на них. Это предполагает четкое разграничение в обозначениях данных и ссылок на них. Следовательно, нужно ввести типы данных, значениями которых являются ссылки (указатели)

на другие данные. Для этого мы используем такие обозначения:

$$\textbf{type}\ T_p = \uparrow T \tag{4.4}$$

В описании типа (4.4) имеется в виду, что значениями типа T_p являются ссылки на данные типа T. Таким образом, стрелка в (4.4) читается как «ссылка на». Существенно, что тип элементов, на которые ссылаются значения типа T_p, задан в его определении. Мы говорим, что T_p связан с T. Эта

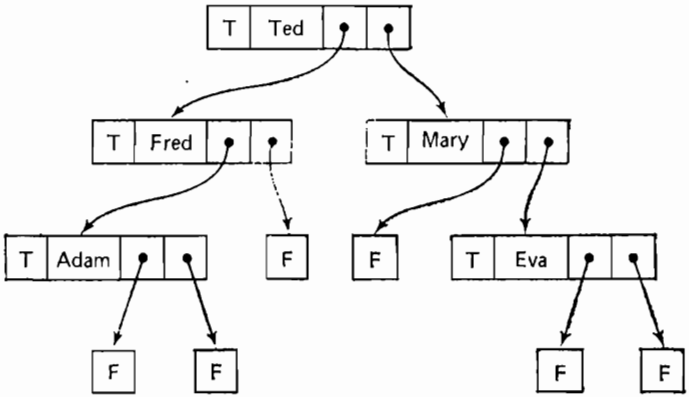


Рис. 4.3. Структура, связанная ссылками.

связь отличает ссылки в языках высокого уровня от адресов в языке ассемблера и является очень важным средством увеличения надежности программ с помощью избыточности обозначений.

Значения ссылочных типов создаются всякий раз, когда динамически размещается какой-либо элемент данных. Мы будем придерживаться соглашения, чтобы каждый такой случай явно выделялся в программе в отличие от ситуации, когда предполагается, что при первом упоминании элемента он автоматически размещается в памяти. Для динамического размещения данных мы вводим встроенную процедуру *new*. Если дана ссылочная переменная p типа T_p, то оператор

$$new(p) \tag{4.5}$$

выделяет память для переменной типа T, создает ссылку типа T_p на эту новую переменную и присваивает значение этой ссылки переменной p (см. рис. 4.4). Теперь сама ссылка обозначается как p (т. е. является значением ссылочной переменной p). В отличие от этого через $p\uparrow$ обозначается переменная, на которую указывает p (т. е. динамически размещенная переменная типа T).

Выше упоминалось, что каждый рекурсивный тип для того, чтобы его кардинальное число было конечным, должен содержать некоторый вариант. Наиболее типичный случай показан на примере генеалогического дерева: после признака принимает одно из двух значений (булевских); когда оно принимает значение *false*, все прочие компоненты отсутствуют. Это изображается такой схемой описаний:

$$\textbf{type}\ T = \textbf{record if}\ p\ \textbf{then}\ S(T)\ \textbf{end} \qquad (4.6)$$

$S(T)$ означает последовательность определений полей, содержащую хотя бы одно поле типа T, что приводит к рекурсии.

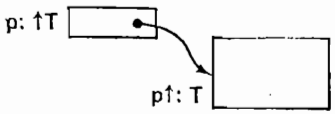


Рис. 4.4. Динамическое размещение переменной $p\uparrow$.

Все структуры типов, описанных по образу (4.6), представляют собой древовидную (или списковую) структуру, подобную изображенной на рис. 4.3. Недостаток такой структуры — наличие ссылок на элементы, состоящие только из одного поля признака, т. е. не содержащие никакой существенной информации. Применение метода ссылок дает возможность легко экономить память, так как позволяет включить информацию поля признака в значение самой ссылки. Обычно принято расширять область значений типа T_p, добавляя к ней значение, которое не ссылается ни на какой элемент. Мы обозначаем его специальным символом **nil** и считаем, что **nil** автоматически является элементом всех ссылочных типов, описанных в программе. Это расширение области ссылочных значений позволяет создавать конечные структуры без явного использования вариантов (условий) в (рекурсивных) описаниях.

Новые формулировки описаний типов данных (4.1) и (4.3), основанные на явных ссылках, даны соответственно в (4.7) и (4.8). Заметим, что в случае (4.8) (который соответствует схеме (4.6)) отсутствует вариант, поскольку $p\uparrow.known = false$ теперь выражается как $p = \textbf{nil}$. Изменение названия типа *ped* на *person* («*человек*») отражает новую точку зрения на структуру, связанную с использованием явных ссылок. Вместо того чтобы рассматривать вначале данную структуру как целостное образование, а затем выделять в ней подструктуры и их компоненты, мы обращаем внимание в первую очередь на компоненты, а их взаимосвязь (основанная на ссылках)

из любого фиксированного описания не очевидна.

```
type expression = record op: operator;
                         opd1, opd2: ↑term
                  end;
type term       = record                                    (4.7)
                    if t then (id: alfa)
                         else (sub: ↑expression)
                  end

type person     = record name: alfa;
                    father, mother: ↑person                 (4.8)
                  end
```

Структура данных, представляющая генеалогическое дерево, показанная на рис. 4.2 и 4.3, здесь вновь изображается на рис. 4.5, где ссылки на неизвестных лиц обозначены через **nil**. Очевидно, что при этом достигается значительная экономия памяти.

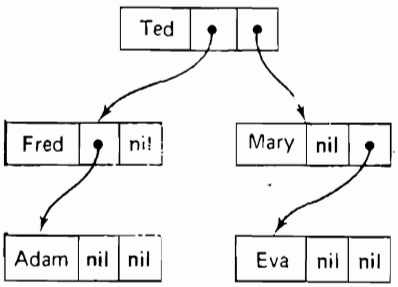


Рис. 4.5. Структура с **nil**.

Обращаясь вновь к рис. 4.5, предположим, что Фред и Мэри — брат и сестра, т. е. имеют общих отца и мать. Такой случай легко выразить, заменив два значения **nil** в соответствующих полях двух записей (*mother* — у Фреда и *father* — у Мэри). Реализация, при которой концепция ссылок скрыта или используются другие приемы управления памятью, вынуждала бы программиста представить записи Адама и Евы по два раза каждую. Хотя при просмотре данных не имеет значения, представлены эти два идентичных отца (или две матери) двумя записями или одной, разница *важна, когда допускается выборочное изменение.* Когда ссылки используются как явные элементы данных, а не как скрытые вспомогательные средства реализации, программист может явно указывать случаи «*разделения*» памяти (в смысле совместного владения какой-либо областью памяти).

Другое следствие использования явных ссылок — возможность определять и обрабатывать циклические структуры данных. Разумеется, эта дополнительная гибкость не только увеличивает возможности, но и требует большей осторожности, так как работа с циклическими структурами легко может привести к бесконечным вычислениям.

То, что мощность и гибкость тесно связаны с опасностью ошибочного использования, — явление, широко известное в программировании. Особенно это касается оператора безусловного перехода (**goto**). Возможно, продолжая аналогию между структурами программ и структурами данных, чисто рекурсивные структуры данных можно поместить на уровень, соответствующий процедуре, тогда как введение ссылок можно сравнить с использованием операторов (**goto**). Так же как с помощью оператора (**goto**), можно строить любые программные схемы (включая циклы), так и с помощью ссылок можно создавать структуры данных любого вида (в том числе циклические). Соответствия между структурами программ и структурами данных указаны в табл. 4.1.

Таблица 4.1. Соответствия между структурами программ и данных

Схема строения	Оператор	Тип данных
Атомарный элемент	Присваивание	Скалярный тип
Перечисление	Составной оператор	Запись
Известное число повторений	Оператор цикла с параметром	Массив
Выбор	Условный оператор	Запись с вариантами, объединение типов
Неизвестное число повторений	Оператор цикла с предусловием или постусловием	Последовательность, или файл
Рекурсия	Оператор процедуры	Рекурсивный тип данных
Универсальный граф	Оператор безусловного перехода	Структура, связанная ссылками

В гл. 3 было показано, что итерация есть частный случай рекурсии и что вызов рекурсивной процедуры P, описанной по схеме

```
procedure P;
begin
   if B then begin P0; P end
end
```
(4.9)

где P_0 — оператор, не содержащий P, эквивалентен итеративному оператору

while B **do** P_0

Аналогии, указанные в табл. 4.1, позволяют обнаружить такую же связь между рекурсивными типами данных и после-

довательностью. В самом деле, рекурсивный тип, определяемый по схеме

```
type T = record
           if B then(t0: T0; t: T)
         end
```
(4.19)

где T_0 — тип, не содержащий T, эквивалентен файловому типу данных

file of T_0

и может быть им заменен. Отсюда видно, что рекурсию можно заменять итерацией в программах и описаниях данных в том (и только в том) случае, если имя процедуры или типа появляется только один раз в конце (или начале) своего описания.

Остальная часть этой главы посвящена созданию и обработке структур данных, компоненты которых связаны явными ссылками. Особое значение придается структурам простой формы; приемы работы с более сложными структурами можно получить из способов работы с основными видами структур. Эти виды — линейные списки, или цепочки, самый простой случай, а также деревья. Предпочтение, которое мы оказываем этим «строительным блокам», не означает, что на практике не встречаются более сложные структуры. В действительности следующая история, напечатанная в цюрихской газете в июле 1922 г., доказывает, что иррегулярность может появляться даже в структурах, которые обычно считаются образцом регулярности, таких, как (семейные) деревья. В этой истории говорится о человеке, который так описывает несчастье своей жизни:

> Я женился на вдове, у которой была взрослая дочь. Мой отец, который довольно часто нас навещал, влюбился в мою падчерицу и женился на ней. Следовательно, мой отец стал моим зятем, а моя падчерица стала моей матерью. Спустя несколько месяцев, моя жена родила сына, который стал шурином моего отца и одновременно моим дядей. У жены моего отца, то есть моей падчерицы, тоже родился сын. Таким образом, у меня появился брат и одновременно внук. Моя жена является моей бабушкой, так как она мать моей матери. Следовательно, я муж моей жены и одновременно ее внук, другими словами, я — свой собственный дедушка.

4.3. ЛИНЕЙНЫЕ СПИСКИ

4.3.1. Основные операции

Самый простой способ соединить, или связать, множество элементов — это расположить их линейно в *списке*, или в *очереди*. В этом случае каждый элемент содержит только одну ссылку, связывающую его со следующим элементом списка.

Пусть тип T описан, как показано в (4.11). Каждая переменная этого типа состоит из трех компонент: идентифицирующего ключа, ссылки на следующий элемент и, возможно, другой информации, не указанной в (4.11).

```
type T = record key: integer;
                next: ↑ T;
                . . . . . .
         end
```
(4.11)

Список элементов типа T показан на рис. 4.6. Переменная-ссылка p указывает на первую компоненту списка. По-видимому, самое простое действие, которое можно выполнить со

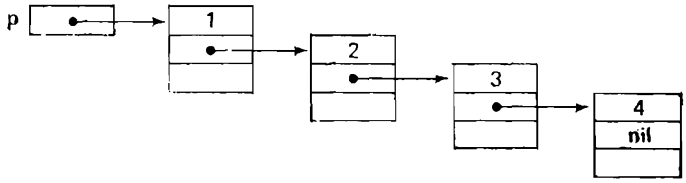


Рис. 4.6. Пример списка.

списком, показанным на рис. 4.6, — вставить в его начало некоторый элемент. Прежде всего этот элемент типа T размещается в памяти, ссылка на него присваивается вспомогательной ссылочной переменной q. После этого ссылкам присваиваются новые значения, как показано в (4.12):

$$new(q); \quad q\uparrow.next := p; \quad p := q \tag{4.12}$$

Заметим, что здесь важен порядок следования этих трех операторов.

Операция включения элемента в начало списка определяет, как можно построить такой список: начиная с пустого списка, последовательно добавлять элементы в его начало. Процесс *формирования списка* описан в (4.13); здесь число связываемых элементов равно n.

```
p := nil; {начало с пустого списка}
while n > 0 do
   begin new(q); q↑.next := p; p := q;
         q↑.key := n; n := n−1
   end
```
(4.13)

Это — самый простой способ построения списка. Но при этом полученный порядок элементов обратен порядку их «поступления». В некоторых случаях это нежелательно; следовательно, новые элементы должны добавляться в конец списка.

Хотя конец легко найти проходом по списку, такой непосредственный подход потребовал бы затрат, которых просто избежать, используя вторую ссылку q, которая всегда указывает на последний элемент. Такой метод применяется, например, в программе 4.4, формирующей перекрестные ссылки на заданный текст. Недостаток такого метода состоит в том, что первый включаемый элемент приходится обрабатывать иначе, чем остальные.

Явное использование ссылок намного упрощает некоторые операции, которые иначе были бы сложными и запутанными; среди элементарных действий со списками есть включение

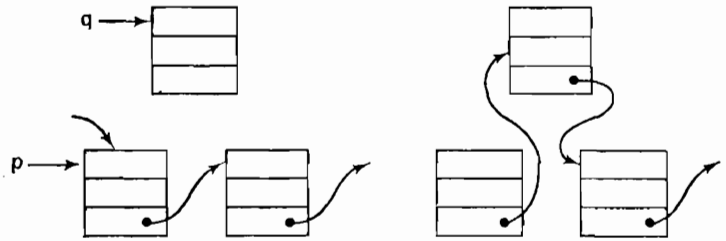


Рис. 4.7. Включение в список после $p\uparrow$.

и удаление элементов (выборочное изменение списка) и, разумеется, просмотр списка. Вначале мы рассмотрим *включение в список*.

Предположим, что элемент, на который указывает ссылка q, нужно включить в список после элемента, на который указывает ссылка p. Необходимые присваивания значений ссылкам показаны в (4.14), а их результат изображен на рис. 4.7.

$$q\uparrow.next := p\uparrow.next; \quad p\uparrow.next := q \tag{4.14}$$

Если требуется включение *перед* элементом, указанным $p\uparrow$, а не после него, то кажется, что однонаправленная цепочка связей создает трудность, поскольку нет «прохода» к элементам, предшествующим данному. Однако простой «трюк» позволяет решить эту проблему; он показан в (4.15) и на рис. 4.8. Допустим, что ключ нового элемента есть $k=8$.

$$\begin{aligned} &new(q); \quad q\uparrow := p\uparrow; \\ &p\uparrow.key := k; \quad p\uparrow.next := q \end{aligned} \tag{4.15}$$

«Трюк» состоит в том, что новая компонента в действительности вставляется после $p\uparrow$, но затем происходит обмен значениями между новым элементом и $p\uparrow$.

Теперь мы рассмотрим процесс *удаления из списка*. Удаление элемента, следующего за $p\uparrow$, очевидно. В (4.16) оно показано в комбинации с одновременным добавлением уда-

ляемого элемента в начало другого списка (на которое указывает q), причем r — вспомогательная переменная типа $\uparrow T$.

$$r := p\uparrow.next; \quad p\uparrow.next := r\uparrow.next; \quad r\uparrow.next := q; \quad q := r \tag{4.16}$$

Рис. 4.9 иллюстрирует процесс (4.16) и показывает, что он состоит из циклического обмена значениями трех ссылок.

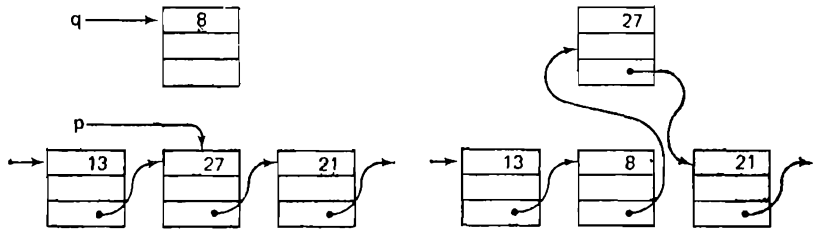


Рис. 4.8. Включение в список перед $p\uparrow$.

Труднее удалить сам указанный элемент (а не следующий за ним), поскольку мы сталкиваемся с той же проблемой, что и при включении перед $p\uparrow$: возврат к элементу, который

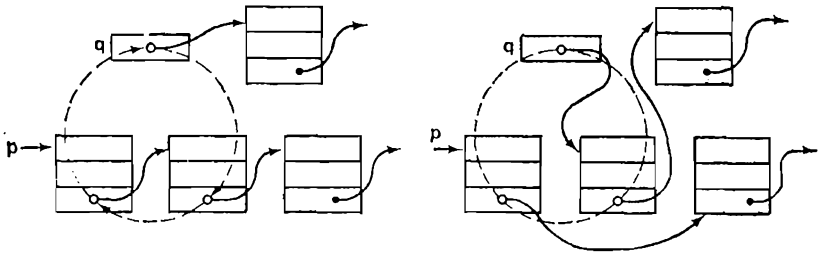


Рис. 4.9. Удаление из списка и включение в другой список.

предшествует указанному, невозможен. Но можно удалить последующий элемент, предварительно переслав его значение ближе к началу списка. Это довольно очевидный и простой прием, но его можно применить только в случае, когда у $p\uparrow$ есть последующий элемент, т. е. он не является последним элементом списка.

Теперь мы перейдем к основной операции *прохода по списку*. Предположим, что операция $P(x)$ должна выполняться с каждым элементом списка, первый элемент которого есть $p\uparrow$. Эту задачу можно выразить следующим образом:

while *список, на который указывает p, непуст* **do**
begin *выполнить операцию P*;
    перейти к следующему элементу
end

Подробнее это действие описывается оператором (4.17):

```
while p≠ nil do
   begin P(p↑); p := p↑.next                    (4.17)
   end
```

Из определения оператора цикла с предусловием и списковой структуры следует, что P будет выполнено для всех элементов списка и ни для каких других.

Очень частая операция — *поиск в списке* элемента с заданным ключом x. Так же как в случае файлов, поиск ведется строго последовательно. Он заканчивается, либо когда элемент найден, либо когда достигнут конец списка. Снова предположим, что начало списка обозначено ссылкой p. Первая попытка сформулировать задачу такого поиска приводит к следующему:

$$\textbf{while } (p \neq \textbf{nil}) \wedge (p\uparrow.key \neq x) \textbf{ do } p := p\uparrow.next \qquad (4.18)$$

Однако следует заметить, что при $p = \textbf{nil}$ не существует $p\uparrow$. Следовательно, вычисление условия окончания может потребовать обращения к несуществующей переменной (в отличие от переменной с неопределенным значением) и может привести к ошибке при выполнении программы. Это можно исправить, используя либо явный выход из цикла, выраженный оператором безусловного перехода (4.19), либо вспомогательную булевскую переменную, отмечающую, найден или нет нужный ключ (4.20).

```
while p ≠ nil do
   if p↑.key = x then goto Found                (4.19)
             else p := p↑.next
```

Использование оператора безусловного перехода требует присутствия в каком-то месте метки для перехода; несовместимость этого оператора с оператором цикла видна из того факта, что условие **while** при этом вводит в заблуждение: тело цикла *не* обязательно выполняется, пока $p \neq \textbf{nil}$.

```
b := true;
while (p ≠ nil) ∧ b do
   if p↑.key = x then b := false                (4.20)
             else p := p↑.next
{(p=nil)∨¬b}
```

4.3.2. Упорядоченные списки и реорганизация списков

Алгоритм (4.20) сильно напоминает подпрограммы поиска при просмотре массива или файла. В самом деле, файл — это в сущности линейный список, в котором техника

связи с последующим элементом остается неопределенной или неявной. Поскольку элементарные операции над файлами не допускают включение новых элементов (разве что в конец) или удаление (разве что уничтожение *всех* элементов), у разработчика есть большая возможность выбора способов представления, и он может использовать также последовательное расположение, помещая следующие одна за другой компоненты в смежные области памяти. Линейные списки с явными ссылками обеспечивают *бо́льшую гибкость* и поэтому их следует использовать, когда требуется такая дополнительная гибкость.

В качестве примера мы рассмотрим теперь задачу, к которой будем постоянно обращаться в этой главе, чтобы продемонстрировать на ней работу различных методов. Она состоит в чтении некоторого текста, выборе из него всех остальных слов и подсчете частоты их появления, т. е. в составлении *частотного словаря*.

Очевидно, что для этого нужно составить *список* слов, найденных в тексте. Каждое очередное слово, прочитанное в тексте, ищется в списке. Если слово найдено, счетчик его частоты увеличивается, в противном случае слово добавляется к списку. Мы будем называть этот процесс просто *поиском*, хотя ясно, что в него входит также и *включение*.

Чтобы сосредоточить внимание на основной задаче обработки списка, мы предположим, что слова уже выделены из исследуемого текста, закодированы целыми числами и находятся во входном файле.

Формулировка этой процедуры, называемой *search* (*поиск*), непосредственно следует из (4.20). Переменная *root* указывает на начало списка, в который вставляются новые слова, это действие определено в (4.12). Полный алгоритм описан в программе 4.1; здесь есть подпрограмма для распечатки полученного списка слов в виде таблицы. Печать таблицы служит примером действия, выполняемого с каждым элементом списка один раз, схема этого процесса уже была приведена в (4.17).

Алгоритм линейного просмотра в программе 4.1 напоминает процедуру поиска в массивах и файлах, где, в частности, используется простой способ упрощения условия окончания цикла: использование *барьера*. При поиске по списку также можно пользоваться барьером, он представляется фиктивным элементом в конце списка. Новая процедура (4.21), заменяющая процедуру поиска в программе 4.1, предполагает, что добавлена глобальная переменная *sentinel* и что инициация переменной *root* заменена операторами

new(*sentinel*); *root* := *sentinel*;

```
program list (input,output);
{простое включение в список}
   type ref = ↑word;
      word = record key: integer;
                    count: integer;
                    next: ref
             end ;
   var k: integer; root: ref;
   procedure search (x: integer; var root: ref);
      var w: ref; b: boolean;
   begin w := root; b := true;
      while (w≠nil) ∧ b do
         if w↑.key = x then b := false else w := w↑.next;
      if b then
      begin {новый элемент} w := root; new (root);
         with root↑ do
         begin key := x; count := 1; next := w
         end
      end else
      w↑.count := w↑.count + 1
   end {search} ;
    procedure printlist (w: ref);
    begin while w ≠ nil do
          begin writeln (w↑.key, w↑.count);
                w := w↑.next
          end
    end {printlist} ;
begin root := nil; read(k);
   while k ≠ 0 do
      begin search (k, root); read(k)
      end ;
   printlist(root)
end .
```

Программа 4.1. Включение в список.

создающими элемент, который будет использоваться в качестве барьера (*sentinel*):

```
procedure search(x: integer; var root: ref);
   var w: ref;
begin w := root; sentinel↑.key := x;
   while w↑.key ≠ x do w := w↑.next;
   if w ≠ sentinel then w↑.count := w↑.count + 1 el
      begin {новый элемент}  w := root; new(root); (4.21)
         with root↑ do
         begin key := x; count := 1; next := w
         end
      end
end {search}
```

Разумеется, в этом примере плохо используется мощность и гибкость связанного списка; при поиске можно допустить линейный просмотр всего списка только в том случае, если число элементов ограниченно. Однако легко найти подходящее усовершенствование: *поиск в упорядоченном списке*. Если список упорядочен (например, по возрастанию ключей), то поиск заканчивается не позднее чем встретится первый ключ, больший, чем искомый. Упорядочение списка достигается включением новых элементов не в начало, а в соответствующие по порядку места. Фактически благодаря легкости включения в связанный список упорядочение обеспечивается почти без дополнительных затрат, т. е. его гибкость используется полностью. Массивы и файлы такой возможности не дают. (Однако заметим, что даже упорядоченные списки не предоставляют ничего эквивалентного бинарному поиску в массивах.)

Поиск в упорядоченном списке — типичный пример ситуации, описанной в (4.15), когда элемент нужно вставлять перед данным, а именно перед первым элементом, имеющим больший ключ. Однако предложенный здесь прием отличен от того, который применялся в (4.15). Вместо копирования значений при проходе списка используются две ссылки: $w2$ отстает на один шаг от $w1$ и, таким образом, указывает на место включения, когда $w1$ находит слишком большой ключ. В общем виде этап включения показан на рис. 4.10. Предварительно мы должны рассмотреть два обстоятельства:

1. Ссылка на новый элемент ($w3$) должна присваиваться $w2$↑*.next*, кроме случая, когда список еще пуст. Для простоты и эффективности мы предпочитаем не использовать для проведения этого различия условный оператор. Един-

ственный способ избежать этого — добавить в начало списка фиктивный элемент.

2. Проход по списку с двумя ссылками, спускающимися на расстоянии в один шаг, требует, чтобы список содержал по меньшей мере один элемент (кроме фиктивного). Поэтому включение первого элемента следует проводить иначе, чем всех остальных.

Процедура, построенная в соответствии с этими указаниями, приведена в (4.23). Она использует вспомогательную

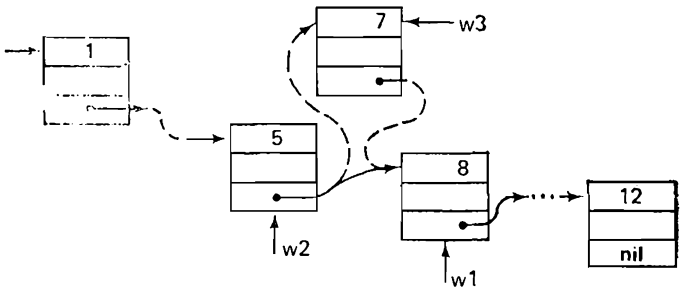


Рис. 4.10. Включение в упорядоченный список.

процедуру *insert* («*включение*»), которую нужно локально описать в *search*. Она размещает в памяти и инициирует новый элемент *w*3, таким образом:

```
procedure insert(w: ref);
   var w3: ref;
begin new(w3);
   with w3↑ do
      begin key := x; count := 1; next := w
      end ;
   w2↑.next := w3
end {insert}
```
(4.22)

Инициация «*root* := **nil**» в программе 4.1 соответственно заменяется на

$$new(root); \quad root\uparrow.next := \mathbf{nil}$$

В соответствии с рис. 4.10 мы определяем условие перехода к следующему элементу при просмотре списка; оно состоит из двух частей, а именно:

$$(w1\uparrow.key < x) \wedge (w1\uparrow.next \neq \mathbf{nil})$$

Ниже приведена процедура поиска:

```
procedure search(x: integer; var root: ref);
   var w1,w2: ref;
begin w2 := root; w1 := w2↑.next;
   if w1 = nil then insert (nil) else
   begin
      while (w1↑.key < x) ∧ (w1↑.next ≠ nil) do
         begin w2 := w1; w1 := w2↑.next                    (4.23)
         end ;
      if w1↑.key = x then w1↑.count := w1↑.count + 1 else
        insert(w1)
   end
end {search} ;
```

К сожалению, несмотря на всю нашу осторожность, сюда вкралась ошибка. Мы предлагаем читателю, прежде чем двигаться дальше, найти здесь логический подвох. Тем же, кто предпочитает избежать этой работы детектива, достаточно сказать, что (4.23) будет всегда проталкивать включенный первым элемент в конец списка. Ошибку можно исправить, указав, что, если поиск заканчивается при невыполнении второй части условия, новый элемент нужно включать *после* $w1\uparrow$, а не *перед* ним. Следовательно, оператор «*insert* (*w*1)» заменяется на

```
begin if w1↑.next = nil then
         begin w2 := w1; w1 := nil
         end;                                              (4.24)
      insert(w1)
end
```

Увы, доверчивый читатель вновь введен в заблуждение, так как алгоритм (4.24) по-прежнему неверен. Чтобы обнаружить ошибку, предположим, что новый ключ лежит между последним и предпоследним ключами. Это приведет к тому, что обе части условия продолжения цикла окажутся ложными, когда будет достигнут конец списка, и, следовательно, включение произойдет после конечного элемента. Если тот же ключ появится позже, он будет включен правильно и, таким образом, окажется в таблице в двух местах. Это можно исправить, заменив условие

$$w1\uparrow.next = \textbf{nil}$$

в (4.24) на

$$w1\uparrow.key < x$$

Чтобы ускорить поиск, условие продолжения в операторе цикла можно вновь упростить, используя барьер. Это требует присутствия *фиктивного элемента* как в начале, так и в конце. Следовательно, список должен инициироваться следующими операторами:

new(*root*); *new*(*sentinel*); *root*↑.*next* := *sentinel*;

и процедура поиска заметно упрощается, что видно из (4.25)

```
procedure search(x: integer; var root: ref);
   var w1,w2,w3: ref;
begin w2 := root; w1 := w2↑.next; sentinel↑.key := x;
   while w1↑.key < x do
      begin w2 := w1; w1 := w2↑.next
      end ;
   if (w1↑.key = x) ∧ (w1 ≠ sentinel) then,
      w1↑.count := w1↑.count + 1 else                    (4.25)
   begin new(w3); {включение w3 между w1 и w2}
      with w3↑ do
         begin key := x; count := 1; next := w1
         end ;
      w2↑.next := w3
   end
end {search}
```

Теперь пора спросить, какого выигрыша можно ждать от поиска в упорядоченном списке. Учитывая, что дополнительное усложнение невелико, не следует ожидать каких-то потрясающих результатов.

Допустим, что все слова встречаются в тексте с одинаковой частотой. В этом случае, как только все слова окажутся в списке, выигрыш, достигнутый при помощи лексикографического упорядочения, фактически будет ничтожен; здесь позиция слова не имеет значения, поскольку важна лишь сумма всех шагов и все слова ищутся с одинаковой частотой. Однако выигрыш достигается при включении нового слова. Вместо всего списка просматривается в среднем только около половины списка. Следовательно, включения в упорядоченный список стоит использовать лишь в случае, когда нужно построить словарь с большим числом различных слов, по сравнению с частотой появления. Поэтому приведенные выше процедуры служат в основном в качестве упражнений в программировании, а не для практического применения.

Помещать данные в связанный список рекомендуется в том случае, если число элементов мало (скажем, <100), заранее неизвестно и, более того, нет никакой информации

о частоте обращения к ним. Типичный пример — таблица имен в трансляторах с языков программирования. Как только встречается описание, новое имя добавляется к списку, а при выходе из области действия описания имя удаляется из списка. Простые связанные списки стоит использовать, если речь идет о сравнительно коротких программах. Даже и в этом случае можно значительно повысить эффективность доступа благодаря очень простому приему, который мы здесь еще раз рассмотрим в первую очередь в связи с тем, что он служит хорошим примером для демонстрации гибкости структуры связанного списка.

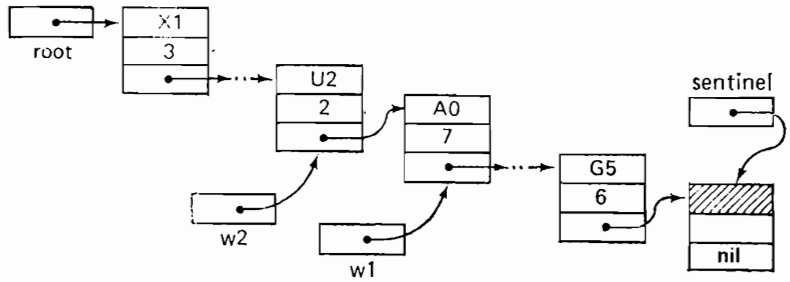


Рис. 4.11. Список до переупорядочения.

Для текста программ характерно частое скопление одного и того же идентификатора, т. е. за одним вхождением часто следует одно или более повторных вхождений того же слова. Это наводит на мысль реорганизовать список после каждого обращения, переставляя найденное слово в начало списка, так как тем самым минимизируется длина прохода по списку при следующем поиске того же слова. Этот метод называется *поиском по списку с переупорядочением*, или — несколько претенциозно — *самоорганизующимся поиском по списку*. Описывая соответствующий алгоритм в виде процедуры, которую можно подставить в программу 4.1, мы учтем предыдущий опыт и с самого начала введем барьер. Действительно, его наличие в этом случае не только ускоряет поиск, но и упрощает программу. С самого начала список не пуст, а уже содержит барьер. Начальные операторы следующие:

new(*sentinel*); *root* := *sentinel*;

Заметим, что основное различие между новым алгоритмом и простым поиском по списку (4.21) — переупорядочение при нахождении элемента. Найденный элемент отделяется, или удаляется со своего старого места и вставляется в начало. Это удаление слова требует использования двух ссылок при поиске, чтобы можно было установить местонахождение эле-

мента $w2\uparrow$, предшествующего найденному элементу $w1\uparrow$, что в свою очередь требует особого обращения с первым элементом (т. е. с пустым списком). Чтобы читатель мог наглядно представить себе процесс изменения связей, мы отсылаем его к рис. 4.11. На нем изображены две ссылки в момент, когда $w1\uparrow$ опознан в качестве искомого элемента. Конфигурация списка после соответствующего переупорядочения показана на рис. 4.12, а новая процедура поиска целиком описана ниже:

```
procedure search(x: integer; var root: ref);
   var w1,w2: ref;
begin w1 := root; sentinel↑.key := x;
   if w1 = sentinel then
   begin {первый элемент} new(root);
      with root↑ do
         begin key := x; count := 1; next := sentinel
         end
   end else
   if w1↑.key = x then w1↑.count := w1↑.count + 1 else
   begin {search}
      repeat w2 := w1; w1 := w2↑.next
      until w1↑.key = x;                                        (4.26)
      if w1 = sentinel then
      begin {включение}
         w2 := root; new(root);
         with root↑ do
            begin key := x; count := 1; next := w2
            end
      end else
      begin {найден, теперь переупорядочивание списка}
         w1↑.count := w1↑.count + 1;
         w2↑.next := w1↑.next; w1↑.next := root; root := w1
      end
   end
end {search}
```

Выигрыш при таком методе поиска сильно зависит от степени скопления входных данных. При заданном коэффициенте скопления улучшение более ощутимо в случае больших списков. Для того чтобы получить представление о примерной величине выигрыша, который можно ожидать, были проведены эмпирические измерения. Приведенная выше программа составления частотного словаря применялась к коротким

и относительно длинным текстам, затем сравнивались методы линейного упорядочения (4.21) и переупорядочения списков

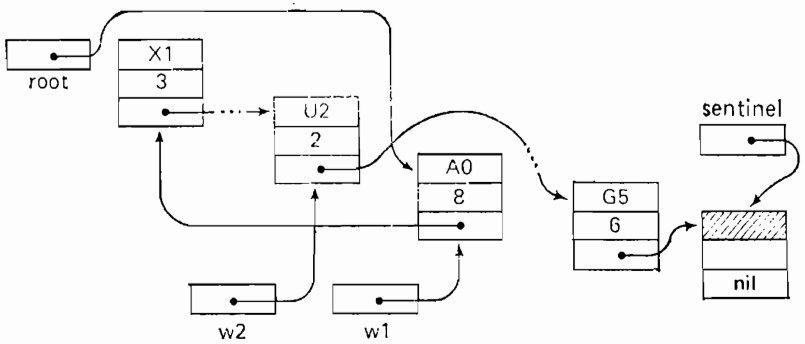


Рис. 4.12. Список после переупорядочения.

(4.26). Результаты измерений приводятся в табл. 4.2. К сожалению, наибольший выигрыш достигается в случаях, когда

Таблица 4.2. Сравнение методов поиска по списку

	Тест 1	Тест 2
Число различных ключей	53	582
Число появлений ключей	315	14 341
Время поиска с упорядочением	6 207	3 200 622
Время поиска с переупорядочением	4 529	681 584
Коэффициент улучшения	1,37	4,70

почему-либо требуется другая организация данных. Мы вернемся к этому примеру в разд. 4.4.

4.3.3. Приложение: топологическая сортировка

Хороший пример использования гибких, динамических структур данных — процесс *топологической сортировки*. Имеется в виду сортировка элементов, для которых определен *частичный порядок*, т. е. упорядочение задано не на всех, а только на некоторых парах элементов. Это довольно типичная ситуация. Приведем несколько таких примеров.

1. В толковом словаре слова определяются с помощью других слов. Если слово v определено с помощью другого слова w, мы обозначим это как $v \prec w$. Топологическая сортировка слов в словаре означает расположение их в таком порядке, чтобы все слова, участвующие в определении данного слова, находились раньше его в словаре.

2. Задача (например, технический проект) разбивается на ряд подзадач. Выполнение одних подзадач обычно должно предшествовать выполнению других подзадач. Если подзадача v должна предшествовать подзадаче w, мы пишем $v \prec w$. Топологическая сортировка означает выполнение подзадач в таком порядке, чтобы перед началом выполнения каждой подзадачи все необходимые для этого подзадачи были уже выполнены.
3. В университетской программе одни предметы опираются на материал других, поэтому некоторые курсы студенты должны прослушать раньше других. Если курс v содержит материал для курса w, мы пишем $v \prec w$. Топологическая сортировка означает чтение курсов в таком порядке, чтобы ни один курс не читался раньше того, на материале которого он основан.
4. В программе некоторые процедуры могут содержать вызовы других процедур. Если процедура v вызывается в процедуре w, мы обозначаем это как $v \prec w$. Топологическая сортировка предполагает расположение описаний процедур в таком порядке, чтобы вызываемые процедуры описывались раньше тех, которые их вызывают *).

В общем виде частичный порядок на множестве S — это отношение между элементами этого множества. Оно обозначается символом $\prec$, читается «предшествует» и удовлетворяет трем следующим свойствам (аксиомам) для любых различных элементов x, y и z из S:

(1) если $x \prec y$ и $y \prec z$, то $x \prec z$ (транзитивность),

(2) если $x \prec y$, то не $y \prec x$ (асимметричность), (4.27)

(3) не $x \prec x$ (иррефлексивность).

По понятным соображениям мы будем считать, что множество S, которое нужно топологически рассортировать, является конечным. Поэтому отношение частичного порядка можно проиллюстрировать с помощью диаграммы или графа, в котором вершины обозначают элементы S, а стрелки изображают отношение порядка. Пример приведен на рис. 4.13.

Цель топологической сортировки — преобразовать частичный порядок в линейный. Графически это означает расположение вершин графа в ряд так, чтобы все стрелки были направлены вправо, как показано на рис. 4.14. Свойства (1) и (2) частичного порядка обеспечивают отсутствие циклов. Это как раз и есть то необходимое условие, при котором возможно преобразование к линейному порядку.

*) Очевидно, что использование рекурсии делает невозможным такое упорядочение. — *Прим. перев.*

Как найти одно из возможных линейных упорядочений? Рецепт достаточно прост. Мы начинаем с того, что выбираем какой-либо элемент, которому не предшествует никакой другой (хотя бы один такой элемент существует, иначе имелся бы цикл). Этот элемент помещается в начало списка и исключается из множества S. Оставшееся множество по-прежнему

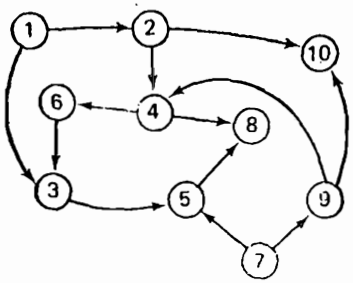


Рис. 4.13. Частично упорядоченное множество.

частично упорядочено; таким образом, можно вновь применить тот же самый алгоритм, пока множество не станет пустым.

Для того чтобы подробнее сформулировать этот алгоритм, нужно описать структуры данных, а также выбрать представление S и отношения порядка. Это представление зависит от

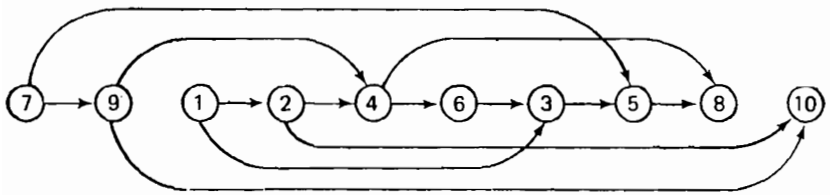


Рис. 4.14. Линейное расположение частично упорядоченного множества, приведенного на рис. 4.13.

выполняемых действий, особенно от операции выбора элемента без предшественников. Поэтому каждый элемент удобно представить тремя характеристиками: идентифицирующим ключом, множеством следующих за ним элементов («последователей») и счетчиком предшествующих элементов («предшественников»). Поскольку n — число элементов в S не задано *a priori*, это множество удобно организовать в виде связанного списка. Следовательно, каждый дескриптор элемента содержит еще поле, связывающее его со следующим элементом списка. Мы будем считать, что ключи — это целые

числа (необязательно последовательные от 1 до n). Аналогично множество последователей каждого элемента можно представить в виде связанного списка. Каждый элемент списка последователей некоим образом идентифицирован и связан со следующим элементом этого списка. Если мы назовем дескрипторы главного списка, в котором каждый элемент из S содержится ровно один раз, *ведущими* (*leaders*), а дескрипторы списка последователей *ведомыми* (*trailers*), то мы получим такие описания типов данных:

```
type lref = ↑leader;
     tref = ↑trailer;
     leader = record key, count: integer;
                  trail: tref;
                  next: lref
              end;
     trailer = record id: lref;
                  next: tref
               end
```
(4.28)

Предположим, что множество S и отношения порядка на нем первоначально заданы в виде последовательности пар ключей во входном файле. Входные данные для примера, изображенного на рис. 4.13, показаны в (4.29), где символы $\prec$ добавлены для ясности:

$$\begin{array}{llllll} 1\prec 2 & 2\prec 4 & 4\prec 6 & 2\prec 10 & 4\prec 8 & 6\prec 3 \\ 1\prec 3 & 3\prec 5 & 5\prec 8 & 7\prec 5 & 7\prec 9 & 9\prec 4 \\ 9\prec 10 & & & & & \end{array} \tag{4.29}$$

Первая часть программы топологической сортировки должна прочитать входной файл и преобразовать входные данные в структуру списка. Это производится последовательным чтением пар ключей x и y ($x\prec y$). Обозначим ссылки на их представления в списке ведущих через p и q. Эти записи ищутся в списке и, если их там нет, добавляются к нему. Эту задачу выполняет функция, называемая L (*located*). Затем к списку ведомых для элемента x добавляется новый дескриптор, идентифицированный как y, счетчик предшественников для y увеличивается на 1. Такой алгоритм соответствует *фазе ввода* (4.30). На рис. 4.15 показана структура, сформированная при обработке входных данных (4.29) с помощью алгоритма (4.30). В этом фрагменте программы есть обращения к функции $L(w)$, дающей ссылку на компоненту списка с ключом w (см. также программу 4.2). Мы предполагаем, что последовательность входных пар клю-

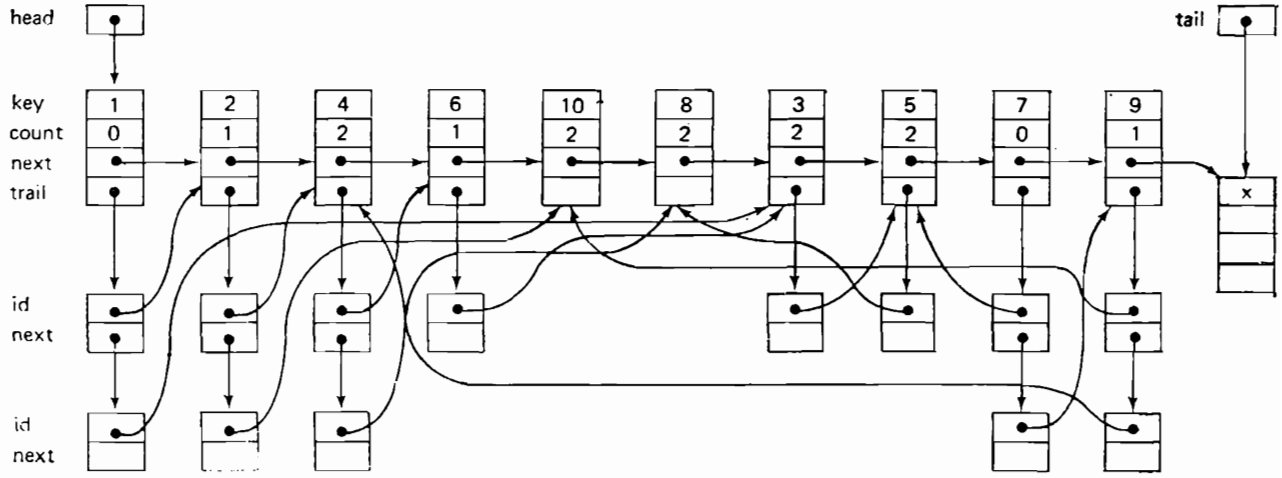


Рис. 4.15. Список, построенный программой топологической сортировки.

чей заканчивается дополнительным нулем.

```
{фаза ввода} read(x);
new(head); tail := head; z := 0;
while x ≠ 0 do
begin read(y); p := L(x); q := L(y);
   new(t); t↑.id := q; t↑.next := p↑.trail;
   p↑.trail := t; q↑.count := q↑.count + 1;
   read(x)
end
```
(4.30)

После того как на фазе ввода построена структура данных, показанная на рис. 4.15, можно провести саму топологическую сортировку, описанную выше. Но поскольку она состоит в последовательном выборе элемента с нулевым счетчиком предшественников, видимо, разумно вначале собрать все

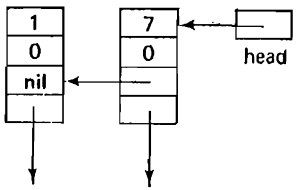


Рис. 4.16. Список ведущих с нулевыми счетчиками.

такие элементы в связанную цепочку. Поскольку мы знаем, что исходная цепочка ведущих впоследствии не понадобится, то же самое поле *next* можно использовать повторно для связывания в цепочку ведущих, не имеющих предшественников. Такая замена одной цепочки на другую часто встречается при работе со списками. Это подробно описано в (4.31); для удобства новая цепочка строится в *обратном порядке*.

```
{поиск ведущих с 0 предшественников}
   p := head; head := nil;
   while p ≠ tail do
      begin q := p; p := q↑.next;
         if q↑.count = 0 then
         begin {включение q↑ в новую цепочку}
            q↑.next := head; head := q
         end
      end
```
(4.31)

Если обратиться к рис. 4.15, то мы увидим, что цепочка *next* ведущих заменяется на цепочку, изображенную на рис. 4.16. Связи, отсутствующие на этом рисунке, остались прежними.

После всех этих подготовительных действий, направленных на то, чтобы выработать подходящее представление частично упорядоченного множества S, мы можем, наконец, перейти к собственно топологической сортировке, т. е. формированию выходной последовательности. В первом, грубом приближении это можно описать следующим образом:

```
q := head;
while q≠nil do
begin {вывести этот элемент, затем исключить его}
   writeln(q↑.key); z := z - 1;                                  (4.32)
   t := q ↑.trail; q := q↑.next;
   «уменьшить счетчик предшественников у всех его последователей в списке ведомых t; если какой-либо счетчик
   стал равен 0, добавить этот элемент к списку ведущих q»
end
```

Оператор в (4.32), который осталось уточнить, осуществляет еще один проход по списку [см. схему (4.17)]. На каждом шаге вспомогательная переменная p указывает на ведущий дискриптор, счетчик которого нужно уменьшить и проверить на равенство нулю.

```
while t ≠ nil do
begin p := t↑.id; p↑.count := p↑.count-1;
      if p↑.count = 0 then
      begin {включение p↑ в список ведущих}                      (4.33)
         p↑.next := q; q := p
      end;
      t := t↑.next
end
```

На этом завершается разработка программы топологической сортировки. Обратите внимание, что был введен счетчик z для подсчета ведущих дескрипторов, сформированных на фазе ввода. Этот счетчик уменьшается каждый раз, когда ведущий дескриптор выводится на фазе вывода. Поэтому он должен вновь стать равным 0 в конце работы программы. Если он не смог вернуться к 0, это указывает, что в структуре остались элементы и среди них нет таких, у которых отсутствуют предшественники. Очевидно, что в этом случае множество S не является частично упорядоченным.

Приведенная выше программа фазы вывода служит примером работы со списком, который «пульсирует», т. е. элементы которого добавляются и удаляются в непредсказуемом порядке. Следовательно, это пример процесса, полностью

```
program topsort(input,output);
type lref = ↑leader;
   tref = ↑trailer;
   leader = record key: integer;
                count: integer;
                trail: tref;
                next: lref;
            end ;
   trailer = record id: lref;
                next: tref
            end ;
var head, tail, p,q: lref;
   t: tref; z: integer;
   x,y: integer;
function L(w: integer): lref;
   {ссылка на ведущего с ключом w}
   var h: lref;
begin h := head; tail↑.key := w;
   while h↑.key ≠ w do h := h↑.next;
   if h = tail then
      begin {в списке нет элемента с ключом w}
         new(tail); z := z+1;
         h↑.count := 0; h↑.trail := nil; h↑.next := tail
      end ;
   L := h
end {L} ;
begin {инициация списка ведущих фиктивным элементов}
   new(head); tail := head; z := 0;
{фаза ввода} read(x);
   while x ≠ 0 do
   begin read(y); writeln(x,y);
      p := L(x); q := L(y);
      new (t); t↑.id := q; t↑.next := p↑.trail;
      p↑.trail := t; q↑.count := q↑.count + 1;
      read(x)
   end ;
{поиск ведущих со счетчиком-0}
   p := head; head := nil;
   while p ≠ tail do
   begin q := p; p := p↑.next;
      if q↑.count = 0 then
         begin q↑.next := head; head := q
```

```
            end
        end ;
    {фаза вывода}  q := head;
        while q ≠ nil do
        begin writeln(q↑.key); z := z-1;
            t := q↑.trail; q := q↑.next;
            while t ≠ nil do
            begin p := t↑.id; p↑.count := p↑.count - 1;
                if p↑.count = 0 then
                begin {включение p↑ в q-список}
                    p↑.next := q; q := p
                end ;
                t := t↑.next
            end
        end ;
        if z ≠ 0 then writeln ('THIS SET IS NOT PARTIALLY ORDERED')
    end .
```

Программа 4.2. Топологическая сортировка.

использующего гибкость, которую обеспечивает явно связанный список.

4.4. ДРЕВОВИДНЫЕ СТРУКТУРЫ

4.4.1. Основные понятия и определения

Мы видели, что последовательности и списки можно определить следующим образом: любая последовательность (список) с базовым типом T — это либо:

1) пустая последовательность (список); либо
2) конкатенация (цепочка) из элемента типа T и последовательности с базовым типом T.

Здесь для определения принципов структурирования (следования или итерации) используется рекурсия. Следование и итерация встречается настолько часто, что их обычно считают фундаментальными «образами» как структур данных, так и «управления» в программах. Однако всегда следует помнить, что с помощью рекурсий их только *можно* определять, но рекурсии можно эффективно и элегантно использовать для определения более сложных структур.

Хорошо известным примером служат деревья. Пусть древовидная структура определяется следующим образом: *древовидная структура* с базовым типом T — это либо:

1) пустая структура; либо

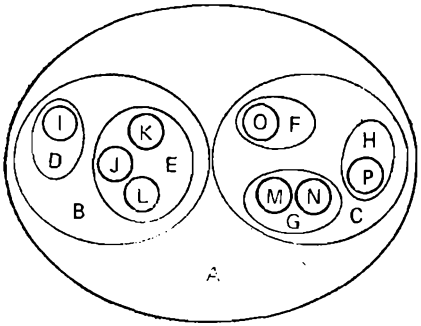
2) узел типа T, с которым связано конечное число древовидных структур с базовым типом T, называемых *поддеревьями*.

Из сходства рекурсивных определений последовательностей и древовидных структур видно, что последовательность (список) есть древовидная структура, у которой каждый узел имеет не более одного «поддерева». Поэтому последовательность (список) называется также *вырожденным деревом*.

Существует несколько способов изображения древовидной структуры. Например, пусть базовый тип T есть множество букв; такая древовидная структура разными способами изображена на рис. 4.17. Все эти представления демонстрируют одну и ту же структуру и поэтому эквивалентны. С помощью графа можно наглядно представить разветвляющиеся связи, которые по понятным причинам привели к общеупотребительному термину «дерево». Однако довольно странно, что деревья принято рисовать перевернутыми или — если кто-то предпочитает иначе выразить этот факт — изображать корни дерева. Но последняя формулировка вводит в заблуждение, так как верхний узел (A) обычно называют *корнем*. Хотя мы сознаем, что в природе деревья представляют собой несколько более сложные образования, чем наши абстракции, мы будем в дальнейшем древовидные структуры называть *просто деревьями*.

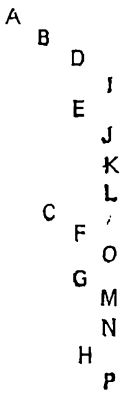
Упорядоченное дерево — это дерево, у которого ветви каждого узла упорядоченны. Следовательно, два упорядоченных дерева на рис. 4.18 — это особые, отличные друг от друга деревья. Узел y, который находится непосредственно под узлом x, называется (непосредственным) *потомком* x; если x находится на *уровне* i, то говорят, что y — на уровне $i+1$. Наоборот, узел x называется (непосредственным) *предком* y. Считается, что корень дерева расположен на уровне 1. Максимальный уровень какого-либо элемента дерева называется его *глубиной* или *высотой*.

Если элемент не имеет потомков, он называется *терминальным* элементом или *листом*, а элемент, не являющийся терминальным, называется *внутренним* узлом. Число (непосредственных) потомков внутреннего узла называется его *степенью*. Максимальная степень всех узлов есть степень дерева. Число ветвей, или ребер, которые нужно пройти, чтобы продвинуться от корня к узлу x, называется *длиной пути* к x. Корень имеет длину пути 1, его непосредственные потомки — длину пути 2 и т. д. Вообще, узел на уровне i имеет длину пути i. Длина пути дерева определяется как сумма длин путей всех его узлов. Она также называется длиной *внутреннего пути*. Например, длина внутреннего пути дерева, изобра-

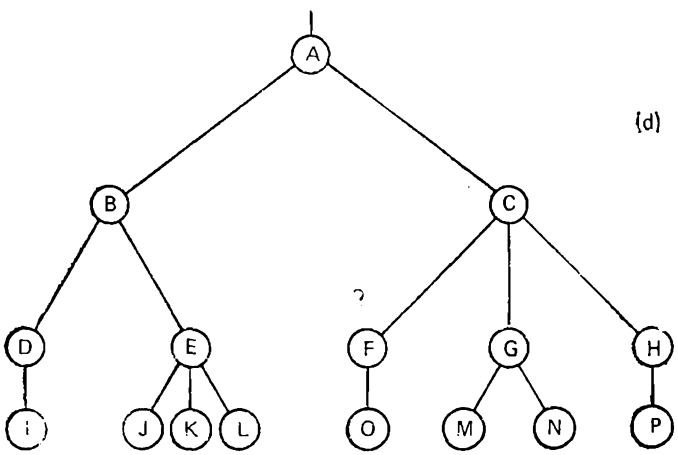


(a)

{A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P)))} (b)



(c)



(d)

Рис. 4.17. Представления древовидной структуры: (a) вложенные множества; (b) вложенные скобки; (c) ломаная последовательность; (d) граф.

женного на рис. 4.17, равна 52. Очевидно, что средняя длина пути P_I есть

$$P_I = \frac{1}{n} \sum_i n_i \cdot i, \tag{4.34}$$

где n_i — число узлов на уровне i. Для того чтобы определить, что называется длиной внешнего пути, мы будем дополнять дерево специальным узлом каждый раз, когда в нем встречается нулевое поддерево. При этом мы считаем, что все узлы должны иметь одну и ту же степень — степень дерева. Следовательно, подобное расширение дерева предполагает запол-

Рис. 4.18. Два различных бинарных дерева.

нение пустых ветвей, разумеется, при этом специальные узлы не имеют дальнейших потомков. Дерево на рис. 4.17, дополненное специальными узлами, показано на рис. 4.19, где специальные узлы изображены квадратиками.

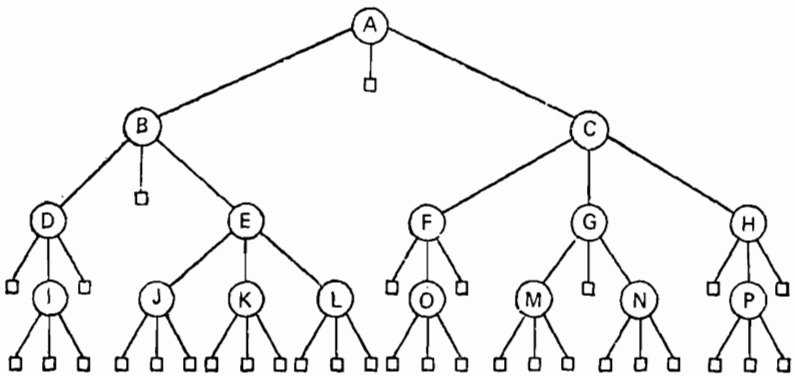


Рис. 4.19. Тернарное дерево со специальными узлами.

Длина внешнего пути теперь определяется как сумма длин путей всех специальных узлов. Если число специальных узлов на уровне i есть m_i, то средняя длина внешнего пути P_E равна

$$P_E = \frac{1}{m} \sum_i m_i \cdot i. \tag{4.35}$$

У дерева, приведенного на рис. 4.19, длина внешнего пути равна 153.

Число специальных узлов m, которые нужно добавить к дереву степени d, непосредственно зависит от числа n исходных узлов. Заметим, что на каждый узел указывает ровно одна ветвь. Следовательно, в расширенном поддереве имеется $m + n$ ветвей. С другой стороны, из каждого исходного узла выходят d ветвей, а из специальных узлов — ни одной. Поэтому всего имеется $dn + 1$ ветвей (1 дает ветвь, указывающую на корень). Из этих двух формул мы получаем следующее равенство между числом m специальных узлов и n исходных узлов: $dn + 1 = m + n$, или

$$m = (d - 1)n + 1. \tag{4.36}$$

Максимальное число узлов в дереве заданной высоты h достигается в случае, когда все узлы имеют d поддеревьев, кроме узлов уровня h, не имеющих ни одного. Тогда в дереве степени d первый уровень содержит 1 узел (корень), уровень 2 содержит d его потомков, уровень 3 содержит d^2 потомков d узлов уровня 2 и т. д. Это дает следующую величину:

$$N_d(h) = 1 + d + d^2 + \ldots + d^{h-1} = \sum_{i=0}^{h-1} d^i \tag{4.37}$$

в качестве максимального числа узлов для дерева с высотой h и степенью d. При $d = 2$ мы получаем

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1. \tag{4.38}$$

Упорядоченные деревья степени 2 играют особо важную роль. Они называются *бинарными деревьями*. Мы определяем упорядоченное бинарное дерево как *конечное множество элементов (узлов), каждый из которых либо пуст, либо состоит из корня (узла), связанного с двумя различными бинарными деревьями*, называемыми *левым* и *правым поддеревом* корня. В следующих пунктах этого раздела мы будем рассматривать исключительно бинарные деревья и поэтому будем употреблять слово «дерево», имея в виду «упорядоченное бинарное дерево». Деревья, имеющие степень больше 2, называются *сильно ветвящимися деревьями* (*multiway trees*), они рассматриваются в разд. 5 этой главы.

Знакомыми примерами *бинарных* деревьев являются фамильное (генеалогическое) дерево с отцом и матерью человека в качестве его потомков (!), история теннисного турнира, где узлом является каждая игра, определяемая ее победителем, а поддеревьями — две предыдущие игры соперников; арифметическое выражение с двухместными операциями, где каждая

операция представляет собой ветвящийся узел с операндами в качестве поддеревьев (см. рис. 4.20).

Теперь мы обратимся к проблеме представления деревьев. Ясно, что изображение таких рекурсивных структур (точнее, рекурсивно определенных. — *Прим. ред.*) с разветвлениями предполагает использование ссылок. Очевидно, что не имеет смысла описывать переменные с фиксированной древовидной структурой, вместо этого *узлы* определяются как переменные

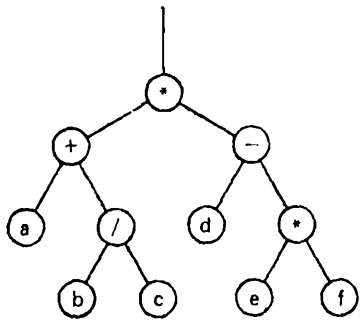


Рис. 4.20. Выражение $(a + b/c)*(d - e*f)$, представленное в виде дерева.

с фиксированной структурой, т. е. фиксированного типа, где степень дерева определяет число компонент-ссылок, указывающих на поддеревья данного узла. Ясно, что ссылка на пустое поддерево обозначается через **nil**. Следовательно, дерево на рис. 4.20 состоит из компонент такого типа:

```
type node = record op: char;
                   left, right: ↑node
            end                                    (4.39)
```

и может строиться, как показано на рис. 4.21.

Ясно, что существуют способы представления абстрактной древовидной структуры в терминах других типов данных, например таких, как массив. Это — общепринятый способ во всех языках, где нет средств динамического размещения компонент и указания их с помощью ссылок. В этом случае дерево на рис. 4.20 можно представить переменной-массивом, описанной как

```
t: array[1..11] of
     record op: char;
        left, right: integer                       (4.40)
     end
```

и со значениями компонент, приведенными в табл. 4.3.

Хотя подразумевается, что массив t представляет абстрактную структуру дерева, мы будем называть его все же не деревом, а массивом согласно явному определению. Мы не будем обсуждать другие возможные представления деревьев в системах, где отсутствует динамическое распределение памяти,

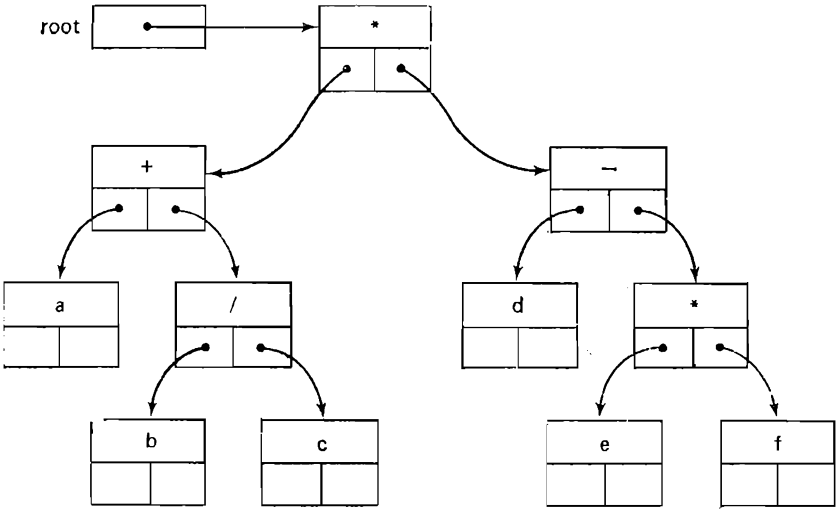


Рис. 4.21. Дерево, представленное как структура данных.

поскольку мы считаем, что системы программирования и языки, имеющие это свойство, являются или станут широко распространенными.

Таблица 4.3. Дерево, представленное с помощью массива

1	*	2	3
2	+	6	4
3	–	9	5
4	/	7	8
5	*	10	11
6	*a*	0	0
7	*b*	0	0
8	*c*	0	0
9	*d*	0	0
10	*e*	0	0
11	*f*	0	0

Прежде чем обсуждать, как лучше использовать деревья и как выполнять операции с деревьями, мы покажем на при-

мере, как программа может строить дерево. Предположим, что нужно сформировать дерево, содержащее узлы типа, описанного в (4.39), а значениями узлов будут n чисел, прочитанных из входного файла. Для усложнения задачи потребуем построить дерево с n узлами и минимальной высотой.

Чтобы достичь минимальной высоты при данном числе узлов, нужно располагать максимально возможное число узлов на всех уровнях, кроме самого нижнего. Это можно сделать очень просто, если распределять все поступающие узлы

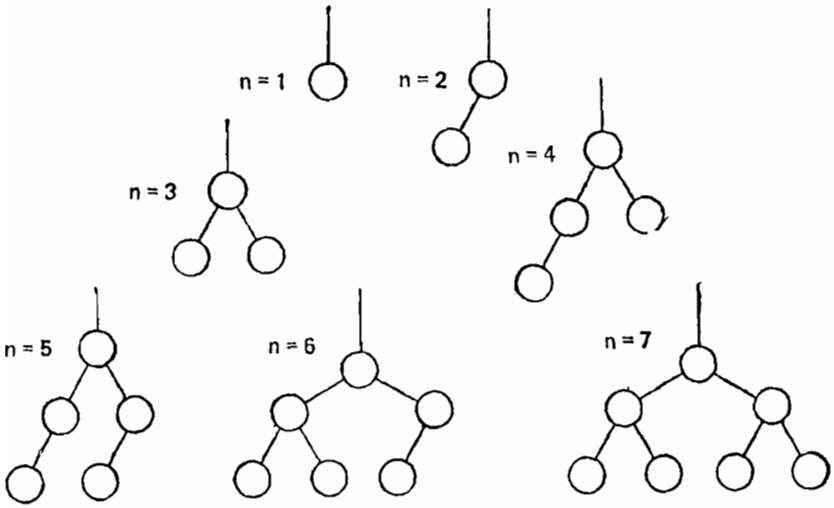


Рис. 4.22. Идеально сбалансированные деревья.

поровну слева и справа от каждого узла. В результате построенное дерево при данном n имеет вид, как показано на рис. 4.22 для $n = 1, \ldots, 7$.

Правило равномерного распределения при известном числе узлов n лучше всего формулируется с помощью рекурсии:

1. Взять один узел в качестве корня.
2. Построить левое поддерево с $nl = n$ **div**2 узлами тем же способом.
3. Построить правое поддерево с $nr = n - nl - 1$ узлами тем же способом.

Это правило описано рекурсивной процедурой *tree*, входящей в программу 4.3, которая читает входной файл и строит идеально сбалансированное дерево. Мы получаем такое определение:

```
program buildtree(input,output);
type ref = ↑node;
   node = record key: integer;
                 left, right: ref
          end ;
var n: integer; root: ref;
function tree(n: integer): ref;
   var newnode: ref;
      x, nl, nr: integer;
begin {построение идеально сбалансированного дерева с n узлами}
   if n = 0 then tree := nil else
   begin nl := n div 2; nr := n-nl-1;
      read(x); new(newnode);
      with newnode↑ do
         begin key := x; left := tree(nl); right := tree(nr)
         end ;
      tree := newnode
   end
end {tree} ;
procedure printtree(t: ref; h: integer);
   var i: integer;
begin {печать дерева t со сдвигом h}
   if t ≠ nil then
   with t↑ do
   begin printtree(left, h+1);
      for i := 1 to h do write('    ');
      writeln(key);
      printtree(right, h+1)
   end
end {printtree} ;
begin {первое целое число есть число узлов}
   read(n);
   root := tree(n);
   printtree(root,0)
end .
```

Программа 4.3. Построение идеально сбалансированного дерева.

Дерево *идеально сбалансировано*, если для каждого его узла количества узлов в левом и правом поддереве различаются не более чем на 1.

Предположим, например, что имеются следующие входные данные для дерева с 21 узлом:

```
21 8  9 11 15 19 20 21  7 3 2 1 5
 6 4 13 14 10 12 17 16 18
```

Тогда программа 4.3 строит идеально сбалансированное дерево, показанное на рис. 4.23.

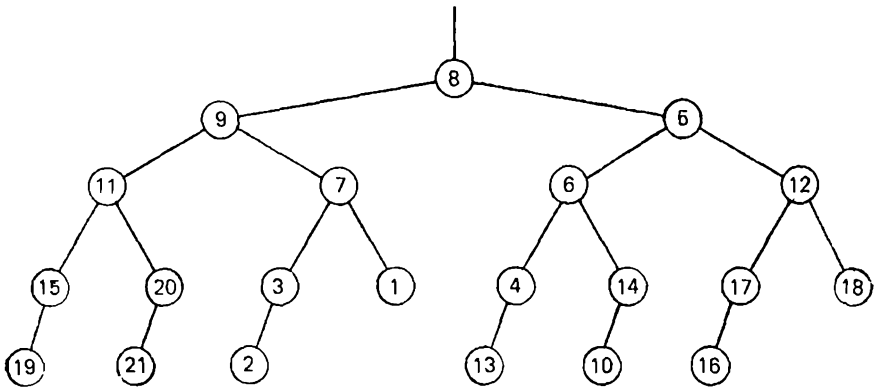


Рис. 4 23. Дерево, построенное с помощью программы 4.3.

Отметим простоту и ясность этой программы, достигнутые благодаря использованию рекурсивных процедур. Очевидно, что рекурсивные алгоритмы особенно уместны, когда программа должна обрабатывать данные, структура которых определена рекурсивно. Это вновь отражается в процедуре *printtree*, которая печатает полученное дерево: пустое дерево не печатается для поддерева уровня L, а вначале печатается его левое поддерево, затем узел, который выделяется предшествующими L пробелами, и, наконец, печатается его правое поддерево.

Преимущество рекурсивного алгоритма особенно наглядно по сравнению с его нерекурсивной формулировкой. Читателю предлагается проявить свою изобретательность и написать нерекурсивную программу, строящую такие же деревья, прежде чем смотреть на (4.41). Эта программа приведена без дальнейших комментариев и может служить упражнением для читателя. Ему предлагается выяснить, как и почему она работает.

```
program buildtree(input,output);
type ref = ↑node;
   node = record key: integer;
             left, right: ref
          end ;
var i,n,nl,nr,x: integer;
   root,p,q,r,dmy: ref;
   s: array [1 . . 30] of  {стек}
         record n: integer; rf: ref
         end ;
begin {первое целое число есть число узлов}
   read(n); new(root); new(dmy);  {фиктивный элемент}
   i := 1; s[1].n := n; s[1].rf := root;                          (4.41)
   repeat n := s[i] .n; r := s[i] .rf; i := i−1;  {из стека}
      if n = 0 then r↑.right := nil else
      begin p := dmy;
         repeat nl := n div 2; nr := n−nl−1;
            read(x); new(q); q↑.key := x;
            i := i+1; s[i].n := nr; s[i].rf := q; {в стек}
            n := nl; p↑.left := q; p := q
         until n = 0;
         q↑.left := nil; r↑.right := dmy↑.left
      end
   until i = 0;
   printtree (root↑.right,0)
end .
```

4.4.2. Основные операции с бинарными деревьями

Имеется много задач, которые можно выполнять на древовидной структуре; распространенная задача — выполнение заданной операции P с каждым элементом дерева. Здесь P рассматривается как параметр более общей задачи посещения всех узлов, или, как это обычно называют, *обхода дерева*.

Если рассматривать эту задачу как единый последовательный процесс, то отдельные узлы посещаются в некотором определенном порядке и могут считаться расположенными линейно. В самом деле, описание многих алгоритмов существенно упрощается, если можно говорить о переходе к следующему элементу дерева, имея в виду некоторое упорядочение.

Существуют три принципа упорядочения, которые естественно вытекают из структуры деревьев. Так же как и саму

древовидную структуру, их удобно выразить с помощью рекурсии. Обращаясь к бинарному дереву на рис. 4.24, где R обозначает корень, а A и B — левое и правое поддеревья, мы можем определить такие три упорядочения:

1. *Сверху вниз*: R, A, B (посетить корень *до* поддеревьев)
2. *Слева направо*: A, R, B
3. *Снизу вверх*: A, B, R (посетить корень *после* поддеревьев)

Обходя дерево на рис. 4.20 и выписывая символы, находящиеся в узлах, в том порядке, в котором они встречаются, мы получаем следующие последовательности:

1. *Сверху вниз*: $* + a/b\ c - d * e f$
2. *Слева направо*: $a + b/c * d - e * f$
3. *Снизу вверх*: $abc/+\ def * - *$

Мы узнаем три формы записи выражений: обход сверху вниз дает *префиксную* запись, обход снизу вверх — *постфиксную* запись, а обход слева направо дает привычную *инфиксную* запись, хотя и без скобок, необходимых для определения порядка выполнения операций.

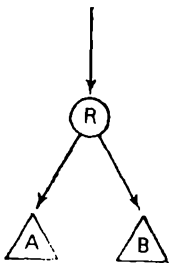


Рис. 4.24. Бинарное дерево.

Теперь выразим эти три метода обхода как три конкретные программы с явным параметром t, означающим дерево, с которым они имеют дело, и неявным параметром P, означающим операцию, которую нужно выполнить с каждым узлом. Введем следующие определения:

```
type ref = ↑node
   node = record . . .                    (4.42)
             left,right: ref
          end
```

Эти три метода легко сформулировать в виде рекурсивных процедур; они вновь служат примером того, что действия

с рекурсивно определенными структурами данных лучше всего описываются рекурсивными алгоритмами.

```
procedure preorder(t: ref);
begin if t ≠ nil then
      begin P(t);
            preorder(t↑.left);
            preorder(t↑.right)
      end
end
```
(4.43)

```
procedure postorder(t: ref);
begin if t ≠ nil then
      begin postorder(t↑.left);
            postorder(t↑.right);
            P(t)
      end
end
```
(4.44)

```
procedure inorder(t: ref);
begin if t ≠ nil then
      begin inorder(t↑.left);
            P(t);
            inorder(t↑.right)
      end
end
```
(4.45)

Отметим, что ссылка t передается как параметр-значение. Это отражает тот факт, что здесь существенна сама *ссылка* (указание) на рассматриваемое поддерево, а не переменная, значение которой есть эта ссылка и которая могла бы изменить значение, если бы t передавался как параметр-переменная.

Пример подпрограммы, осуществляющей обход дерева, — это подпрограмма печати дерева с соответствующим сдвигом, выделяющим каждый уровень узлов (см. программу 4.3).

Бинарные деревья часто используются для представления множеств данных, элементы которых ищутся по уникальному, только им присущему ключу. Если дерево организовано таким образом, что для каждого узла t_i все ключи в левом поддереве меньше ключа t_i, а ключи в правом поддереве больше ключа t_i, то это дерево называется *деревом поиска*. В дереве поиска можно найти место каждого ключа, двигаясь начиная от корня и переходя на левое или правое поддерево каждого узла в зависимости от значения его ключа. Как мы видели,

n элементов можно организовать в бинарное дерево с высотой не более чем $\log n$. Поэтому для поиска среди n элементов может потребоваться не более $\log n$ сравнений, если дерево идеально сбалансировано. Очевидно, что дерево — намного более подходящая форма организации такого множества данных, чем линейный список, который рассматривался в предыдущем разделе.

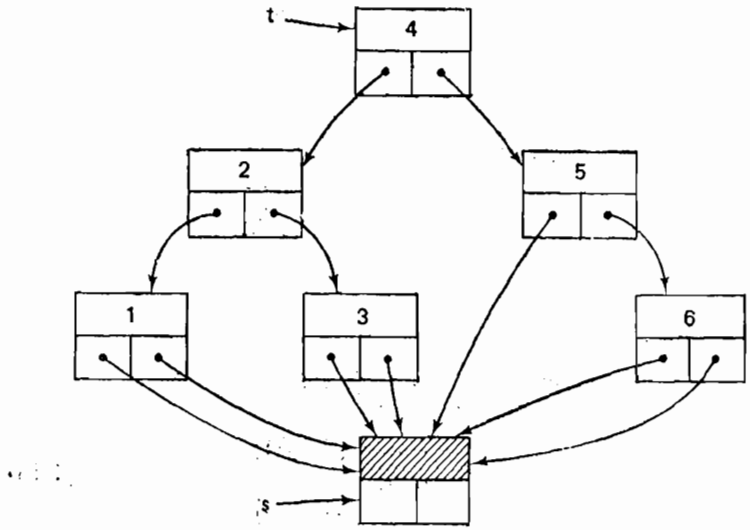


Рис. 4.25. Дерево поиска с барьером.

Так как этот поиск проходит по единственному пути от корня к искомому узлу, его можно запрограммировать с помощью итерации (4.46):

```
function loc(x: integer; t: ref): ref;
   var found: boolean;
begin found := false;
   while (t ≠ nil) ∧ ¬found do
   begin                                                    (4.46)
      if t↑.key = x then found := true else
      if t↑.key > x then t := t↑.left else t := t↑.right
   end;
   loc := t
end
```

Функция $loc(x, t)$ имеет значение nil, если в дереве с корнем t не найдено ключа со значением x. Так же как в случае поиска по списку, сложность условия окончания цикла за-

ставляет искать лучшее решение. При поиске по списку в конце его помещается *барьер*. Этот прием можно применить и в случае поиска по дереву. Использование ссылок позволяет связать все терминальные узлы дерева с одним и тем же барьером. Полученная структура — это уже не просто дерево, а скорее, дерево, все листья которого прицеплены внизу к одному якорю (см. рис. 4.25). Барьер можно также считать общим представлением всех внешних (специальных) узлов, которыми дополняется исходное дерево (см. рис. 4.19). Полученная в результате упрощенная процедура поиска описана ниже:

```
function loc(x; integer; t: ref): ref;
begin s↑.key := x; {барьер}
   while t↑.key ≠ x do
      if x < t↑.key then t := t↑.left else t := t↑.right;
   loc := t
end
```
(4.47)

Отметим, что если в дереве с корнем t не найдено ключа со значением x, то в этом случае $loc(x, t)$ принимает значение s, т. е. ссылки на барьер. Ссылка на s просто принимает на себя роль ссылки **nil**.

4.4.3. Поиск по дереву с включением

Возможности техники динамического размещения переменных с доступом к ним через ссылки вряд ли полностью проявляются в тех примерах, где построенная структура данных остается неизменной. Более подходящими примерами служат задачи, в которых сама структура дерева изменяется, т. е. дерево растет и/или уменьшается во время выполнения программы. Это также случай, когда другие представления данных, такие, как массив, не подходят и когда дерево с элементами, связанными ссылками, как раз и есть подходящая структура.

Прежде всего рассмотрим случай постоянно растущего, но никогда не убывающего дерева. Хорошим примером этого является задача построения частотного словаря, которая уже разбиралась, когда речь шла о связанных списках. Вернемся к ней снова. В этой задаче задана последовательность слов и нужно установить число появлений каждого слова. Это означает, что, начиная с пустого дерева, каждое слово ищется в дереве. Если оно найдено, увеличивается его счетчик появлений, если нет — в дерево вставляется новое слово (с начальным значением счетчика, равным 1). Мы называем эту задачу

поиском по дереву с включением. Предполагаются следующие описания типов:

```
type ref = ↑word;
     word = record
              key: integer;
              count: integer;
              left, right: ref
            end
```
(4.48)

Считая, кроме того, что у нас есть исходный файл ключей *f*, а переменная *root* указывает на корень дерева поиска, мы можем записать программу следующим образом:

```
reset(f);
while ¬eof(f) do
   begin read(f, x); search(x,root) end
```
(4.49)

Определение пути поиска здесь вновь очевидно. Но если он приводит в «тупик», т. е. к пустому поддереву, обозначен-

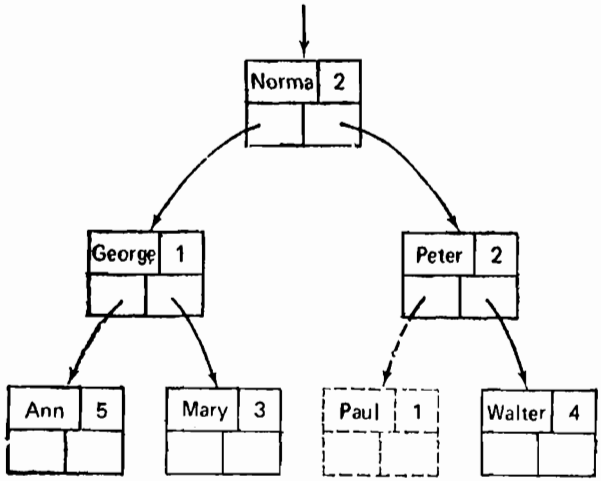


Рис. 4.26. Включение в упорядоченное бинарное дерево.

ному ссылочным значением **nil**, то данное слово нужно вставить в дерево на место пустого поддерева. Рассмотрим, например, бинарное дерево, показанное на рис. 4.26, и включение в него слова «Paul». Результат показан пунктирными линиями на том же рисунке.

Целиком работа алгоритма приведена в программе 4.4. Процесс поиска представлен в виде рекурсивной процедуры.

Отметим, что ее параметр *p* передается как параметр-переменная, а *не* как параметр-значение. Это существенно, поскольку в случае включения *переменной* должно присваиваться некоторое новое значение ссылке, которая перед этим имела значение **nil**. Для входной последовательности, состоящей из 21 числа, которая обрабатывалась с помощью программы 4.3, построившей дерево на рис. 4.23, программа 4.4 строит бинарное дерево поиска, показанное на рис. 4.27.

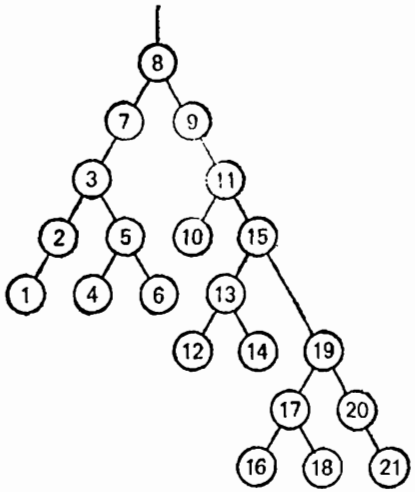


Рис. 4.27. Дерево поиска, построенное с помощью программы 4.4.

Использование барьера вновь несколько упрощает задачу, что показано в (4.50). Понятно, что в начале программы переменная *root* должна инициироваться ссылкой на барьер, а не значением **nil**, и перед каждым поиском очередного слова искомое значение *x* должно присваиваться полю ключа в барьере.

```
procedure search(x: integer; var p: ref);
begin
   if x < p↑.key then search(x, p↑.left) else
   if x > p↑.key then search(x, p↑.right) else
   if p ≠ s then p↑.count := p↑.count + 1 else
     begin {включение} new(p);                                    (4.50)
        with p↑ do
        begin key := x; left := s; right := s; count := 1
        end
     end
end
```

```
program treesearch(input,output);
{поиск с включением по двоичному дереву}
type ref = ↑word;
   word = record key: integer;
                 count: integer;
                 left, right: ref;
          end ;
var root: ref; k: integer;

procedure printtree(w: ref; l: integer);
   var i: integer;
begin if w ≠ nil then
      with w↑ do
      begin printtree(left, l+1);
            for i := 1 to l do write('   ');
            writeln(key);
            printtree(right, l+1)
      end
end ;
procedure search(x: integer; var p: ref);
begin
   if p = nil then
   begin {слова нет в дереве; включить его}
      new (p);
      with p↑ do
         begin key := x; count := 1; left := nil; right := nil
         end
   end else
   if x < p↑.key then search(x, p↑.left) else
   if x > p↑.key then search(x, p↑.right) else
      p↑.count := p↑.count + 1
end {search} ;
begin root := nil;
   while ¬eof(input) do
      begin read(k); search(k, root)
      end ;
   printtree(root,0)
end
```

Программа 4.4. Поиск с включениями.

Еще раз, теперь уже последний, построим альтернативную версию этой программы, отказавшись от использования рекурсии. Но сейчас избежать рекурсии не так просто, как в случае без включения, так как для того, чтобы производить включение, нужно помнить пройденный путь по крайней мере на один шаг назад. В программе 4.4 он запоминается автоматически при использовании параметра-переменной.

Чтобы правильно привязать включаемую компоненту, мы должны иметь ссылку на ее предка и знать, включается она в качестве правого или левого поддерева. Для этого вводятся две переменные: $p2$ и d (для направления):

```
procedure search(x: integer; root: ref);
    var p1,p2: ref; d: integer;
begin p2 := root; p1 := p2↑.right; d := 1;
    while (p1≠nil) ∧ (d≠0) do
    begin p2 := p1;
        if x < p1↑.key then
            begin p1 := p1↑.left; d := -1 end else
        if x > p1↑.key then
            begin p1 := p1↑.right; d := 1 end else
        d := 0
    end ;
    if d = 0 then p1↑.count := p1↑.count + 1 else
        begin {включение} new(p1);
            with p1↑ do
            begin key := x; left := nil; right := nil; count := 1
            end ;
            if d < 0 then p2↑.left := p1 else p2↑.right := p1
        end
end
```

(4.51)

Как и в случае поиска с включением по списку, используются две ссылки $p1$ и $p2$, такие, что в процессе поиска $p2$ всегда указывает на предикат $p1\uparrow$. Чтобы удовлетворить этому условию в начале поиска, вводится вспомогательный фиктивный элемент, на который указывает *root*. Начало действительного дерева поиска обозначается ссылкой *root*↑.*right*. Поэтому программа должна начинаться операторами

new(*root*); *root*↑.*right* := **nil**

вместо начального присваивания

root := **nil**

Хотя задача этого алгоритма — поиск, его можно применить и для сортировки. В самом деле, он очень напоминает метод сортировки включением, а поскольку вместо массива используется дерево, пропадает необходимость перемещения компонент выше места включения. Сортировку с помощью дерева можно запрограммировать почти столь же эффективно, как и лучшие методы сортировки массивов. Но небходимо принять некоторые меры предосторожности. Разумеется, при появлении одинаковых ключей, теперь надо поступать иначе. Если в случае $x = p\uparrow.key$ алгоритм работает так же, как и в случае $x > p\uparrow.key$, то он представляет метод устойчивой сортировки, т. е. элементы с одинаковыми ключами появляются в той же последовательности при обычном обходе дерева, что и в процессе их включения в дерево.

Вообще говоря, имеются лучшие способы сортировки, но в задачах, где требуется и поиск, и сортировка, алгоритм поиска по дереву с включением весьма рекомендуется. Он действительно очень часто применяется в трансляторах и программах работы с банками данных для организации объектов, которые нужно хранить и искать в памяти. Подходящий пример — построение таблицы перекрестных ссылок для заданного текста. Исследуем эту задачу подробно.

Наша цель — написать программу, которая (читая текст f и печатая его с добавлением последовательных номеров строк) собирает все слова этого текста, сохраняя при этом номера строк, в которых они встречались. Когда этот просмотр закончится, нужно построить таблицу, содержащую все собранные слова в алфавитном порядке, со списками соответствующих строк.

Очевидно, что дерево поиска (называемое также *лексикографическим* деревом) лучше всего подходит для представления слов, встречающихся в тексте. Теперь каждый узел не только содержит слово в качестве значения ключа, но одновременно представляет собой начало списка номеров строк. Каждую запись номера строки мы будем называть *отметкой*. Следовательно, в этом примере мы встречаем и деревья, и линейные списки. Программа состоит из двух основных частей (см. программу 4.5): фазы чтения текста и построения дерева и фазы печати таблицы. Ясно, что последняя является частным случаем процедуры обхода дерева, где посещение каждого узла предполагает печать значения ключа слова и проход по связанному с ним списку номеров строк (отметок). Кроме того, полезно привести еще некоторые пояснения, относящиеся к программе 4.5:

1. Словом считается любая последовательность букв и цифр, начинающаяся с буквы.

```
program crossref(f,output);
{построение таблицы перекрестных ссылок с использованием
двоичного дерева}
const c1 = 10;    {длина слова}
      c2 =  8;    {количество слов в строке}
      c3 =  6;    {количество цифр в числе}
      c4 = 9999;  {максимальный номер строки}
type alfa = packed array [1 .. c1] of char;
     wordref = ↑word;
     itemref = ↑item;
     word = record key: alfa;
                   first, last: itemref;
                   left, right: wordref
            end ;
     item = packed record
                   lno: 0 .. c4;
                   next: itemref
            end ;
var root: wordref;
    k,k1: integer;
    n: integer;          {номер текущей строки}
    id: alfa;
    f: text;
    a: array [1 .. c1] of char;
procedure search (var w1: wordref);
  var w: wordref; x: itemref;
begin w := w1;
  if w = nil then
  begin new(w); new(x);
    with w↑ do
    begin key := id; left := nil; right := nil;
      first := x; last := x
    end ;
    x↑.lno := n; x↑.next := nil; w1 := w
  end else
  if id < w↑.key then search(w↑.left) else
  if id > w↑.key then search(w↑.right) else
  begin new(x); x↑.lno := n; x↑.next := nil;
    w↑.last↑.next := x; w↑.last := x
  end
end {search} ;
procedure printtree(w: wordref);
```

```
    procedure printword(w: word);
        var l: integer; x: itemref;
    begin write (' ', w.key);
        x := w.first; l := 0;
        repeat if l = c2 then
                begin writeln;
                    l := 0; write (' ':c1+1)
                end ;
            l := l+1; write (x↑.lno:c3); x := x↑.next
        until x = nil;
        writeln
    end {printword} ;
begin if w ≠ nil then
    begin printtree(w↑.left);
        printword(w↑); printtree(w↑.right)
    end
end {printtree} ;
begin root := nil; n := 0; k1 := c1;
    page (output); reset(f);
    while ¬eof (f) do
    begin if n = c4 then n := 0;
        n := n+1; write (n:c3);          {следующая строка}
        write (' ');
        while ¬eoln (f) do
        begin {просмотр непустой строки}
            if f↑ in ['A'..'z'] then
            begin k := 0;
                repeat if k < c1 then
                    begin k := k+1; a[k] := f↑;
                    end ;
                    write (f↑); get (f)
                until ¬(f↑ in ['A'..'z','0'..'9']);
                if k ≥ k1 then k1 := k else
                repeat a[k1] := ' '; k1 := k1-1
                until k1 = k;
                pack (a,1,id); search(root)
            end else
            begin {проверка на кавычку или комментарий}
                if f↑ = '''' then
                    repeat write(f↑); get(f)
                    until f↑ = '''' else
                if f↑ = '{' then
```

```
                repeat write(f↑); get(f)
                until f↑ = '}';
              write (f↑); get (f)
            end
        end ;
        writeln; get(f)
    end ;
    page(output); printtree(root);
end
```

Программа 4.5. Построение таблицы перекрестных ссылок.

2. В качестве ключа хранятся только первые $c1$ символов. Таким образом, два слова, у которых первые $c1$ символов не различаются, считаются одинаковыми.
3. Эти $c1$ символов упаковываются в массив *id* (типа *alfa*). Если $c1$ достаточно мало, во многих вычислительных машинах такие упакованные массивы могут сравниваться с помощью одной команды.
4. Переменная $k1$ — это индекс, который используется в следующем инвариантном условии, касающемся буфера символов a:

$$a[i] = ' \ ' \quad \text{для} \quad i = k1 + 1 \ldots c1.$$

   Это означает, что слова, состоящие из менее чем $c1$ символов, дополняются соответствующим количеством пробелов.
5. Желательно, чтобы номера строк в таблице перекрестных ссылок печатались в возрастающем порядке. Поэтому список отметок должен формироваться в том же порядке, в каком они печатаются. Это требование предполагает использование в каждом слове-узле двух ссылок, из которых одна указывает на первый, а вторая — на последний элемент списка отметок.
6. Сканер строится таким образом, что слова в кавычках и внутри комментариев не включаются в таблицу перекрестных ссылок; при этом предполагается, что кавычки и комментарии не переходят через концы строк.

В табл. 4.4 показан результат обработки некоторого текста программы.

4.4.4. Удаление из дерева

Теперь мы переходим к задаче, обратной включению, а именно удалению. Нам нужно построить алгоритм для удаления узла с ключом x из дерева с упорядоченными ключами. К сожалению, удаление элемента обычно не так просто, как

Таблица 4.4. Пример распечатки, полученной в результате работы программы 4.5.

```
 1 PROGRAM PERMUTE (OUTPUT);
 2    CONST N = 4;
 3    VAR I: INTEGER;
 4        A: ARRAY [1..N] OF INTEGER;
 5
 6    PROCEDURE PRINT;
 7       VAR I: INTEGER;
 8    BEGIN FOR I := 1 TO N DO WRITE (A[I]:3);
 9       WRITELN
10    END {PRINT} ;
11
12    PROCEDURE PERM (K: INTEGER);
13       VAR I,X: INTEGER;
14    BEGIN
15       IF K = 1 THEN PRINT ELSE
16       BEGIN PERM (K-1);
17          FOR I := 1 TO K-1 DO
18          BEGIN X := A[I]; A[I] := A[K]; A[K] := X;
19             PERM (K-1);
20                X := A[I]; A[I] := A[K]; A[K] := X;
21          END
22       END
23    END {PERM} ;
24
25 BEGIN
26    FOR I := 1 TO N DO A[I] := I;
27    PERM (N)
28 END .
```

ARRAY	4							
A	4	8	18	18	18	18	20	20
	20	20	26					
BEGIN	8	14	16	18	25			
CONST	2							
DO	8	17	26					
ELSE	15							
END	10	21	22	23	28			
FOR	8	17	26					
IF	15							
INTEGER	3	4	7	12	13			
I	3	7	8	8	13	17	18	18
	20	20	26	26	26			

Продолжение

K	12	15	16	17	18	18	13	20
	20							
N	2	4	8	26	27			
OF	4							
OUTPUT	1							
PERMUTE	1							
PERM	12	16	19	27				
PRINT	6	15						
PROCEDURE	6	12						
PROGRAM	1							
THEN	15							
TO	8	17	26					
VAR	3	7	13					
WRITELN	9							
WRITE	8							
X	13	18	18	20	20			

включение. Оно просто в случае, когда удаляемый элемент является терминальным узлом или имеет одного потомка.

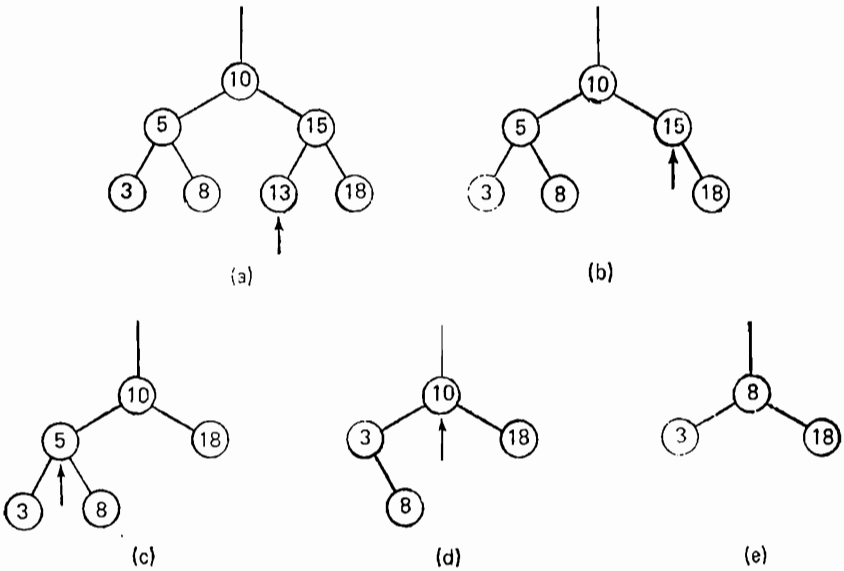


Рис. 4.28. Удаление из дерева.

Трудность заключается в удалении элементов с двумя потомками, поскольку мы не можем указывать одной ссылкой на два направления. В этом случае удаляемый элемент нужно заменить либо на самый правый элемент его левого подде-

рева, либо на самый левый элемент его правого поддерева. Ясно, что такие элементы не могут иметь более одного потомка. Подробно это показано в рекурсивной процедуре, называемой *delete* (4.52). Она различает три случая:

1. Компоненты с ключом, равным x, нет.
2. Компонента с ключом x имеет не более одного потомка.
3. Компонента с ключом x имеет двух потомков.

```
procedure delete (x: integer; var p: ref);
   var q: ref;
   procedure del (var r: ref);
   begin if r↑.right ≠ nil then del (r↑.right) else
         begin q↑.key := r↑.key; q↑.count := r↑.count;
            q := r; r := r↑.left
         end
   end ;
begin {delete}
   if p = nil then writeln (' WORD IS NOT IN TREE') else                (4.52)
   if x < p↑.key then delete(x, p↑.left) else
   if x > p↑.key then delete(x, p↑.right) else
   begin {delete p↑}  q := p;
      if q↑.right = nil then p := q↑.left else
      if q↑.left = nil then p := q↑.right else del (q↑.left);
      {dispose(q)}
   end
end {delete}
```

Вспомогательная рекурсивная процедура *del* вызывается только в 3-м случае. Она «спускается» вдоль самой правой ветви левого поддерева удаляемого узла $q\uparrow$ и затем заменяет существенную информацию (ключ и счетчик) в $q\uparrow$ соответствующими значениями самой правой компоненты $r\uparrow$ этого левого поддерева, после чего от $r\uparrow$ можно освободиться. Процедуру *dispose*(q) можно рассматривать как обратную процедуре *new*(q). Последняя занимает память для новой компоненты, а первая может применяться для указания вычислительной системе, что память, которую занимает $q\uparrow$, можно освободить и потом вновь использовать (некоторый вид чистки памяти).

Для иллюстрации работы процедуры (4.52) мы отсылаем читателя к рис. 4.28. Задано начальное дерево (a), из которого последовательно удаляются узлы с ключами 13, 15, 5, 10. Полученные деревья показаны на рис. 4.28 (b — e).

4.4.5. Анализ поиска с включениями по дереву

Довольно естественно испытывать некоторое недоверие к алгоритму поиска по дереву с включениями. Во всяком случае, до тех пор, пока мы не узнаем более детально о его работе, мы будем испытывать некоторые сомнения. Прежде всего многих программистов беспокоит то, что обычно мы не знаем, каким образом будет расти дерево, и не имеем никакого представления о форме, которую оно примет. Мы лишь можем догадаться, что оно, скорее всего, не будет идеально сбалансированным. Поскольку среднее число сравнений, необходимых для нахождения ключа в идеально сбалансированном дереве с n узлами, приблизительно равно $h = \log n$, то число сравнений в дереве, сформированном этим алгоритмом, будет больше h. Но насколько больше?

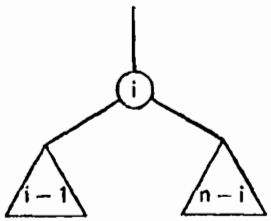


Рис. 4.29. Распределение весов по ветвям.

Прежде всего легко найти наихудший случай. Допустим, что ключи поступают уже в строго возрастающем (или убывающем) порядке. Тогда каждый ключ вставляется непосредственно справа (или слева) от предшествующего, и построенное дерево оказывается полностью вырожденным, т. е. оно превращается в линейный список. В этом случае средние затраты на поиск равны $n/2$ сравнениям. Очевидно, что в таком наихудшем случае алгоритм поиска малоэффективен, и, кажется, что наши сомнения оправдываются. Конечно, встает вопрос, насколько вероятен такой случай. Точнее, мы хотели бы знать длину a_n пути поиска, усредненную по всем n ключам и усредненную по всем $n!$ деревьям, которые получаются в результате $n!$ перестановок n исходных различных ключей. Эта задача анализа алгоритмов оказывается достаточно простой и приводится здесь не только как типичный пример такого анализа, но и из-за практической важности полученного результата.

Пусть даны n различных ключей со значениями 1, 2, ..., n. Предположим, что они появляются в случайном порядке. Вероятность того, что первый ключ, который становится корневым узлом, будет иметь значение i, есть $1/n$. Его левое поддерево в конце работы будет содержать $i - 1$ узлов, а правое поддерево — $n - i$ узлов (см. рис. 4.29). Пусть средняя длина пути в левом поддереве обозначается через

a_{i-1}, а в правом поддереве a_{n-i}. Вновь предполагается, что все возможные перестановки оставшихся $n-1$ ключей равновероятны. Средняя длина пути в дереве с n узлами равна сумме произведений уровня каждого узла и вероятности обращения к нему. Если предположить, что все узлы ищутся с одинаковой вероятностью, то

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i, \tag{4.53}$$

где p_i есть длина пути до узла i.

В дереве на рис. 4.29 мы разделяем узлы на три класса:

1. $i-1$ узлов в левом поддереве имеют среднюю длину пути $a_{i-1}+1$.
2. Корень имеет длину пути, равную 1.
3. $n-i$ узлов в правом поддереве имеют среднюю длину пути $a_{n-i}+1$.

Следовательно, (4.53) можно представить в виде суммы трех слагаемых:

$$a_n^{(i)} = (a_{i-1}+1)\frac{i-1}{n} + 1\cdot\frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n}. \tag{4.54}$$

Искомая величина a_n теперь получается как среднее a_n^i для всех $i=1, \ldots, n$, т. е. для всех деревьев с ключами $1, 2, \ldots, n$ в корне:

$$\begin{aligned} a_n &= \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n}\right] = \\ &= 1 + \frac{1}{n^2}\sum_{i=1}^{n}[(i-1)a_{i-1} + (n-i)a_{n-i}] = \\ &= 1 + \frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\cdot a_i. \end{aligned} \tag{4.55}$$

Уравнение (4.55) представляет собой реккурентное соотношение для a_n вида $a_n = f_1(a_1, a_2, \ldots, a_{n-1})$. Отсюда мы можем получить более простое реккурентное соотношение вида $a_n = f_2(a_{n-1})$ следующим образом:

Из (4.55) непосредственно получаем

$$(1)\quad a_n = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\cdot a_i = 1 + \frac{2}{n^2}(n-1)a_{n-1} + \frac{2}{n^2}\sum_{i=1}^{n-2} i\cdot a_i,$$

$$(2)\quad a_{n-1} = 1 + \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i\cdot a_i.$$

Умножив (2) на $(n-1/n)^2$, мы получим

$$(3)\quad \frac{2}{n^2}\sum_{i=1}^{n-2} i\cdot a_i = \frac{(n-1)^2}{n^2}(a_{n-1}-1)$$

и, подставив (3) в (1), получим

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1}+2n-1). \tag{4.56}$$

Оказывается, что a_n можно представить в нерекурсивной, закрытой форме с помощью гармонической функции

$$\begin{aligned} H_n &= 1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n},\\ a_n &= 2\cdot\frac{n+1}{n}H_n-3. \end{aligned} \tag{4.57}$$

[Недоверчивый читатель может проверить, что (4.57) удовлетворяет рекурсивному соотношению (4.56).]

Из формулы Эйлера (используя константу $\gamma\cong 0{,}577$)

$$H_n = \gamma+\ln(n)+\frac{1}{12n^2}+\ldots$$

мы получаем для больших n соотношение

$$a_n \cong 2[\ln(n)+\gamma]-3 = 2\ln(n)-c.$$

Поскольку средняя длина пути в идеально сбалансированном дереве приблизительно равна

$$a'_n = \log(n)-1, \tag{4.58}$$

опуская постоянное слагаемое, которое становится незначительным для больших n, мы получаем

$$\lim_{n\to\infty}\frac{a_n}{a'_n} = \frac{2\ln n}{\log n} = 2\cdot\ln 2 = 1.386. \tag{4.59}$$

Что дает нам результат этого анализа (4.59)? Из него мы можем сделать вывод, что, стараясь всегда строить идеально сбалансированное дерево вместо «случайного» дерева, получаемого программой 4.4, мы могли бы, по-прежнему предполагая, что все ключи появляются с равной вероятностью, ожидать среднего выигрыша в длине пути поиска не более 39 %. Ударение следует сделать на слове «среднего», поскольку, разумеется, выигрыш может быть нáмного больше в неудачном случае, когда формируемое дерево полностью вырождается в список, но вероятность этого случая невелика (если все перестановки n ключей равновероятны). В связи

с этим следует отметить, что ожидаемая средняя длина пути в «случайном» дереве растет тоже строго логарифмически по отношению к числу его узлов, несмотря на то что в худшем случае длина пути увеличивается в линейной зависимости.

Цифра 39 % накладывает ограничения на объем дополнительных затрат, которые имеет смысл вкладывать в какую-либо доперестройку структуры дерева при включении элементов. Разумеется, отношение r между частотой обращений (поиска) к узлам (информации) и частотой включений существенно влияет на границу, до достижения которой эти затраты выгодны. Чем больше этот коэффициент, тем больше выигрыш от такой процедуры перестройки. Цифра 39 % достаточно низка, и в большинстве случаев выигрыш от такой процедуры по сравнению с простым алгоритмом включения в дерево не оправдывает затрат, если только число узлов и соотношение между поиском и включением не оказываются велики (или если можно не опасаться худшего случая).

4.4.6. Сбалансированные деревья

Из предыдущих рассуждений ясно, что процедура включения, восстанавливающая идеальную сбалансированность структуры дерева, вряд ли будет выгодна, поскольку такое восстановление после случайного включения — довольно сложная операция. Но ее можно упростить, если дать менее строгое определение «сбалансированности». Такой несовершенный критерий сбалансированности может потребовать более простой перестройки дерева при небольшом уменьшении среднего быстродействия поиска.

Одно такое определение сбалансированности было дано Адельсоном-Вельским и Ландисом [4.1]. Критерий сбалансированности следующий:

> Дерево является *сбалансированным* тогда и только тогда, когда для каждого узла высота его двух поддеревьев различается не более чем на 1.

Деревья, удовлетворяющие этому условию, часто называют АВЛ-деревьями (по фамилиям их изобретателей). Мы будем называть их просто *сбалансированными деревьями*, так как их критерий сбалансированности оказывается наиболее подходящим. (Отметим, что все идеально сбалансированные деревья являются также АВЛ-сбалансированными.)

Это определение не только простое, но также приводит к легко выполнимой балансировке, а средняя длина поиска остается практически такой же, как у идеально сбалансированного дерева.

Со сбалансированными деревьями можно выполнять следующие операции за $O(\log n)$ единицу времени даже в худшем случае:

1. Найти узел с данным ключом.
2. Включить узел с данным ключом.
3. Удалить узел с данным ключом.

Это является прямым следствием теоремы, доказанной Адельсоном-Вельским и Ландисом, которая утверждает, что сбалансированное дерево никогда не будет более чем на 45 % выше соответствующего идеально сбалансированного дерева независимо от количества узлов. Если мы обозначим высоту сбалансированного дерева с n узлами через $h_b(n)$, то

$$\log(n+1) \leqslant h_b(n) \leqslant 1.4404 \cdot \log(n+2) - 0.328 \quad (4.60)$$

Разумеется, оптимум достигается, если дерево идеально сбалансировано, при $n = 2^k - 1$. Но какова структура наихудшего АВЛ-сбалансированного дерева?

Чтобы найти максимальную высоту h всех сбалансированных деревьев с n узлами, возьмем фиксированное h и попробуем построить сбалансированное дерево с минимальным количеством узлов. Такая стратегия рекомендуется, поскольку, как и в случае минимального h, это значение может быть достигнуто только при некоторых, определенных значениях n.

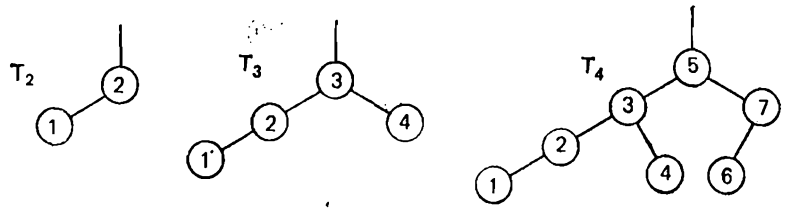


Рис. 4.30. Деревья Фибоначчи высотой 2, 3 и 4.

Обозначим такое дерево с высотой h через T_h. Очевидно, что T_0 — пустое дерево, а T_1 — дерево с одним узлом. Чтобы построить дерево T_h для $h > 1$, мы зададим корень с двумя поддеревьями, которые также имеют минимальное число узлов. Следовательно, поддеревья также являются T-деревьями. Очевидно, что одно поддерево *обязано* иметь высоту $h-1$, а другому позволено иметь высоту на единицу меньше, т. е. $h-2$. На рис. 4.30 показаны деревья высотой 2, 3 и 4. Поскольку принцип их организации напоминает принцип построения чисел Фибоначчи, подобные деревья называются *деревьями Фибоначчи*. Они определяются следующим образом:

1. Пустое дерево есть дерево Фибоначчи с высотой 0.

2. Один узел есть дерево Фибоначчи с высотой 1.
3. Если T_{h-1} и T_{h-2} — деревья Фибоначчи с высотой $h-1$ и $h-2$, то $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$ есть дерево Фибоначчи с высотой h.
4. Никакие другие деревья не являются деревьями Фибоначчи.

Число узлов в T_h определяется простым рекуррентным соотношением:

$$N_0 = 0, \quad N_1 = 1,$$
$$N_h = N_{h-1} + 1 + N_{h-2}, \tag{4.61}$$

N_i — это количества узлов, для которых можно получить наихудший случай (верхнюю границу h) из (4.60).

4.4.7. Включение в сбалансированное дерево

Посмотрим, что может произойти, когда в сбалансированное дерево включается новый узел. Пусть дан корень r с левым и правым поддеревьями L и R. Предположим, что в L

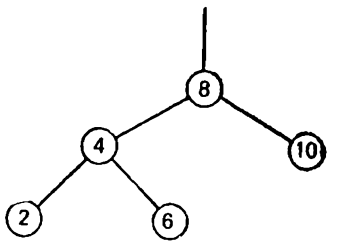


Рис 4.31. Сбалансированное дерево.

включается новый узел, вызывая увеличение его высоты на 1. Возможны три случая:

1. $h_L = h_R$: L и R становятся неравной высоты, но критерий сбалансированности не нарушается.
2. $h_L < h_R$: L и R приобретают равную высоту, т. е. сбалансированность даже улучшается.
3. $h_L > h_R$: критерий сбалансированности нарушается, и дерево нужно перестраивать.

Рассмотрим дерево на рис. 4.31. Узлы с ключами 9 и 11 можно вставить без балансировки; дерево с корнем 10 становится односторонним (случай 1), а с корнем 8 улучшает свою сбалансированность (случай 2). Однако включение узлов 1, 3, 5 или 7 требует последующей балансировки.

При внимательном изучении этой ситуации можно обнаружить, что имеются лишь две существенно различные возмож-

ности, требующие индивидуального подхода. Оставшиеся могут быть получены симметричными преобразованиями этих двух. Случай 1 определяется включением ключа 1 или 3 в дерево на рис. 4.31, случай 2 — включением узла 5 или 7.

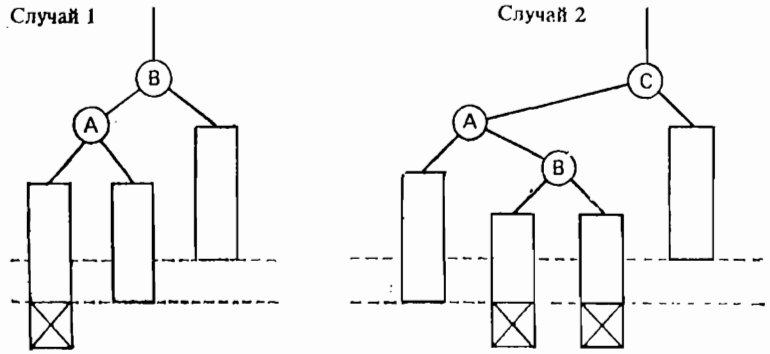


Рис. 4.32. Несбалансированность, возникающая при включении.

Эти два случая в общем виде показаны на рис. 4.32, где поддеревья обозначены прямоугольниками, а увеличение высоты при включении указано перечеркнутыми квадратами. Простые преобразования этих двух структур восстанавливают

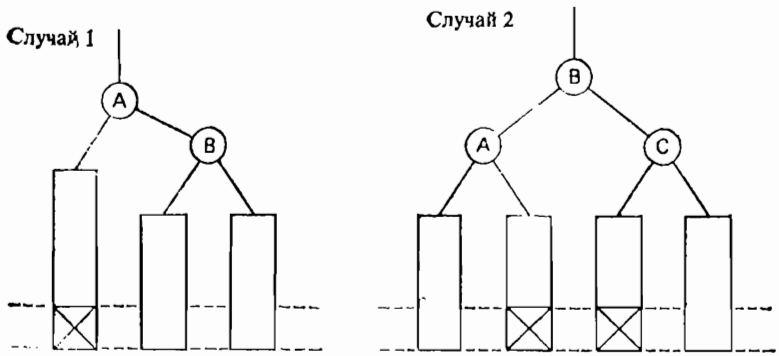


Рис. 4.33. Восстановление баланса.

нужную сбалансированность. Их результат приведен на рис. 4.33; отметим, что допускаются перемещения лишь в вертикальном направлении, в то время как относительное горизонтальное расположение показанных узлов и поддеревьев должно оставаться без изменений.

Алгоритм включения и балансировки полностью определяется способом хранения информации о сбалансированности дерева. Крайнее решение состоит в хранении этой информа-

ции полностью неявно в самой структуре дерева. Но в этом случае показатель сбалансированности узла должен заново вычисляться каждый раз, когда узел затрагивается включением, что приводит к чрезвычайно высоким затратам. Другая крайность — явно хранить показатель сбалансированности в информации, связанной с каждым узлом. Тогда определение (4.48) типа узла расширяется до

```
type node = record key: integer;
                   count: integer;
                   left,right: ref;
                   bal: −1..+1
            end
```
(4.62)

В дальнейшем мы будем интерпретировать показатель сбалансированности узла как высоту его правого поддерева минус высота его левого поддерева и будем строить алгоритм, исходя из узлов описанного в (4.62) типа.

В общих чертах процесс включения узла состоит из последовательности таких трех этапов:

1. Следовать по пути поиска, пока не окажется, что ключа нет в дереве.
2. Включить новый узел и определить новый показатель сбалансированности.
3. Пройти обратно по пути поиска и проверить показатель сбалансированности у каждого узла.

Хотя этот метод требует некоторой избыточной проверки (если сбалансированность установлена, то для предков соответствующего узла ее проверять уже не надо), мы будем вначале придерживаться этой, очевидно, корректной схемы, так как ее можно реализовать с помощью простого расширения уже разработанной процедуры поиска с включением из программы 4.4. Эта процедура описывает операцию поиска для каждого отдельного узла, и благодаря рекурсивной формулировке ее можно дополнить операцией, выполняемой «по дороге назад вдоль пути поиска». На каждом шаге должна передаваться информация о том, увеличилась ли высота поддерева (в которое произведено включение). Поэтому мы добавим к списку параметров процедуры булевскую переменную h, означающую «*высота поддерева увеличилась*». Очевидно, что h должна быть параметром-переменной, поскольку используется для передачи результата.

Теперь предположим, что алгоритм возвращается к узлу с левой ветви (см. рис. 4.32) с указанием, что ее высота уве-

личилась. Мы должны различать три возможные ситуации в зависимости от высоты поддеревьев перед включением:

1. $h_L < h_R$, $p\uparrow.bal = +1$, предыдущая несбалансированность в p уравновешивается.
2. $h_L = h_R$, $p\uparrow.bal = 0$, вес склонился влево.
3. $h_L > h_R$, $p\uparrow.bal = -1$, необходима балансировка.

В третьем случае показатель сбалансированности корня левого поддерева (скажем, $p1\uparrow.bal$) определяет, который из случаев (1 или 2 на рис. 4.32) имеет место. Если левое поддерево этого узла тоже выше правого, то мы имеем дело со случаем 1, иначе — со случаем 2. (Убедитесь, что в этом случае левое поддерево корня не может иметь показатель сбалансированности, равный 0.) Необходимые операции балансировки полностью заключаются в обмене значениями ссылок. Фактически ссылки обмениваются значениями по кругу, что приводит к однократному или двукратному «повороту» двух или трех узлов. Кроме «вращения» ссылок следует также изменить соответствующие показатели сбалансированности узлов. Подробно это показано в процедуре поиска, включения и балансировки (4.63).

Принцип работы алгоритма показан на рис. 4.34. Рассмотрим бинарное дерево (a), которое состоит только из двух узлов. Включение ключа 7 вначале дает несбалансированное дерево (т. е. линейный список). Его балансировка требует однократного правого (RR) поворота, давая в результате идеально сбалансированное дерево (b). Последующее включение узлов 2 и 1 дает несбалансированное поддерево с корнем 4. Это поддерево балансируется однократным левым (LL) поворотом (d). Далее включение ключа 3 сразу нарушает критерий сбалансированности в корневом узле 5. Сбалансированность теперь восстанавливается с помощью более сложного двукратного поворота налево и направо (LR); результатом является дерево (e). Теперь при следующем включении потерять сбалансированность может лишь узел 5. Действительно, включение узла 6 должно привести к четвертому виду балансировки, описанному в (4.63): двукратному повороту направо и налево (RL). Окончательное дерево показано на рис. 4.34 (f). С эффективностью алгоритма включения в сбалансированное дерево связаны следующие два особо интересных вопроса:

1. Если все $n!$ перестановок n ключей появляются с одинаковой вероятностью, то какова ожидаемая высота формируемого сбалансированного дерева?
2. Какова вероятность, что включение приведет к балансировке?

```
procedure search(x: integer; var p: ref; var h: boolean);
   var p1,p2: ref;      {h = false}
begin
   if p = nil then
   begin {слова нет в дереве; включить его}
      new(p); h := true;
      with p↑ do
      begin key := x; count := 1;
         left := nil; right := nil; bal := 0
      end
   end else
   if x < p↑.key then
   begin search(x, p↑.left, h);
      if h then      {выросла левая ветвь}
      case p↑.bal of
     1:  begin p↑.bal := 0; h := false
         end ;
     0:  p↑.bal := -1;
    -1:  begin {балансировка} p1 := p↑.left;
            if p1↑.bal = -1 then
            begin {однократный LL-поворот}
               p↑.left := p1↑.right; p1↑.right := p;
               p↑.bal := 0; p := p1
            end else
            begin {двукратный LR-поворот}  p2 := p1↑.right;
               p1↑.right := p2↑.left; p2↑.left := p1;
               p↑.left := p2↑.right; p2↑.right := p;
               if p2↑.bal = -1 then p↑.bal := +1 else p↑.bal := 0;
               if p2↑.bal = +1 then p1↑.bal := -1 else p1↑.bal := 0;
               p := p2
            end ;
            p↑.bal := 0; h := false
         end
      end
   end else
   if x > p↑.key then
   begin search(x, p↑.right, h);
      if h then      {выросла правая ветвь}
      case p↑.bal of
    -1:  begin p↑.bal := 0; h := false
         end ;
     0:  p↑.bal := +1;
     1:  begin {балансировка} p1 := p↑.right;
```

```
            if p1↑.bal = +1 then
            begin {однократный RR-поворот}
               p↑.right := p1↑.left; p1↑.left := p;
               p↑.bal := 0; p := p1
            end else
            begin {двукратный RL-поворот} p2 := p1↑.left;
               p1↑.left := p2↑.right; p2↑.right := p1;
               p↑.right := p2↑.left; p2↑.left := p;
               if p2↑.bal = +1 then p↑.bal := -1 else p↑.bal := 0;
               if p2↑.bal = -1 then p1↑.bal := +1 else p1↑.bal := 0;
               p := p2
            end;
            p↑.bal := 0; h := false
         end
      end
   end
   else
   begin p↑.count := p↑.count + 1; h := false
   end
end {search}
```

(4.63)

Математический анализ этого сложного алгоритма пока не произведен. Эмпирические проверки оправдывают предположение, что ожидаемая высота сбалансированного дерева, которое строится в (4.63), равна $h = \log(n) + c$, где c — малая константа ($c \cong 0{,}25$). Это значит, что на практике АВЛ-сбалансированные деревья ведут себя так же, как идеально сбалансированные деревья, хотя с ними намного легче работать. Эмпирически можно также предположить, что в среднем балансировка необходима приблизительно один раз на каждые два включения. При этом однократный и двукратный повороты одинаково вероятны. Пример на рис. 4.34 явно был тщательно подобран, чтобы показать как можно больше поворотов при минимальном числе включений.

Из-за сложности операций балансировки считается, что сбалансированные деревья следует использовать лишь в том случае, когда поиск информации происходит значительно чаще, чем включение. Это, в частности, верно потому, что узлы таких деревьев поиска обычно для экономии памяти реализуются как плотно упакованные записи. «Скорость» изменения показателей сбалансированности, занимающих только по два разряда каждый, и обращения к ним часто является решающим фактором, определяющим эффективность

операции перебалансировки. Эмпирические оценки говорят, что сбалансированные деревья теряют большую часть своей привлекательности, если нужна плотная упаковка записи.

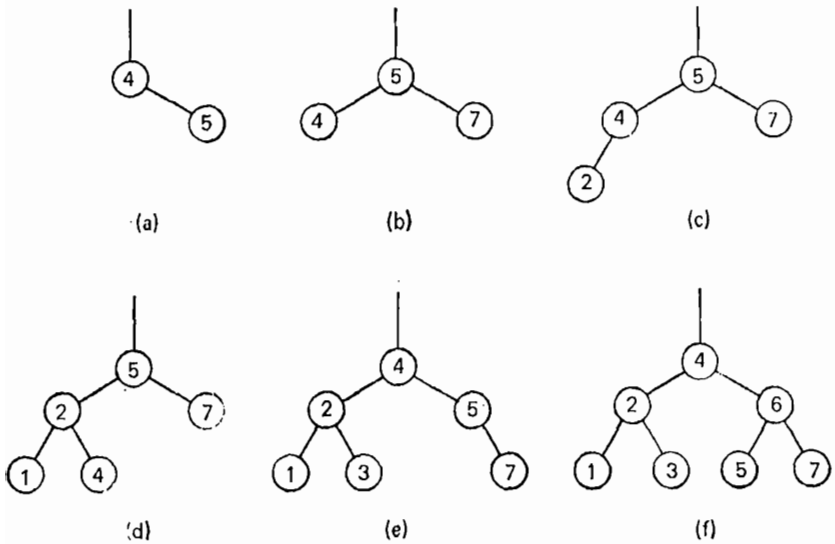


Рис. 4.34. Включение в сбалансированное дерево.

В самом деле, трудно превзойти простой и очевидный алгоритм включения в дерево!

4.4.8. Удаление из сбалансированного дерева

Учитывая наш опыт с удалением из дерева, мы можем предположить, что в случае сбалансированных деревьев удаление будет еще более сложным, чем включение. Это верно, хотя операция балансировки остается в основном такой же, что и при включении. В частности, балансировка состоит из однократного или двукратного поворота узлов.

Удаление из сбалансированного дерева основано на алгоритме (4.52). Простыми случаями являются удаление терминальных узлов и узлов с одним потомком. Если же узел, который нужно удалить, имеет два поддерева, мы вновь будем заменять его самым правым узлом левого поддерева. Как и в случае включения (4.63), добавляется булевский параметр-переменная h, означающий, что «высота поддерева уменьшилась». Вопрос о перебалансировке рассматривается только при $h = true$. Значение *true* присваивается h при нахождении и удалении узла или если сама балансировка уменьшает высоту поддерева. В программе (4.64) мы вводим две

(симметричные) операции балансировки в виде процедур, так как в алгоритме удаления к ним обращаются несколько раз. Отметим, что *balance 1* используется, когда уменьшается высота левого, а *balance 2* — правого поддерева.

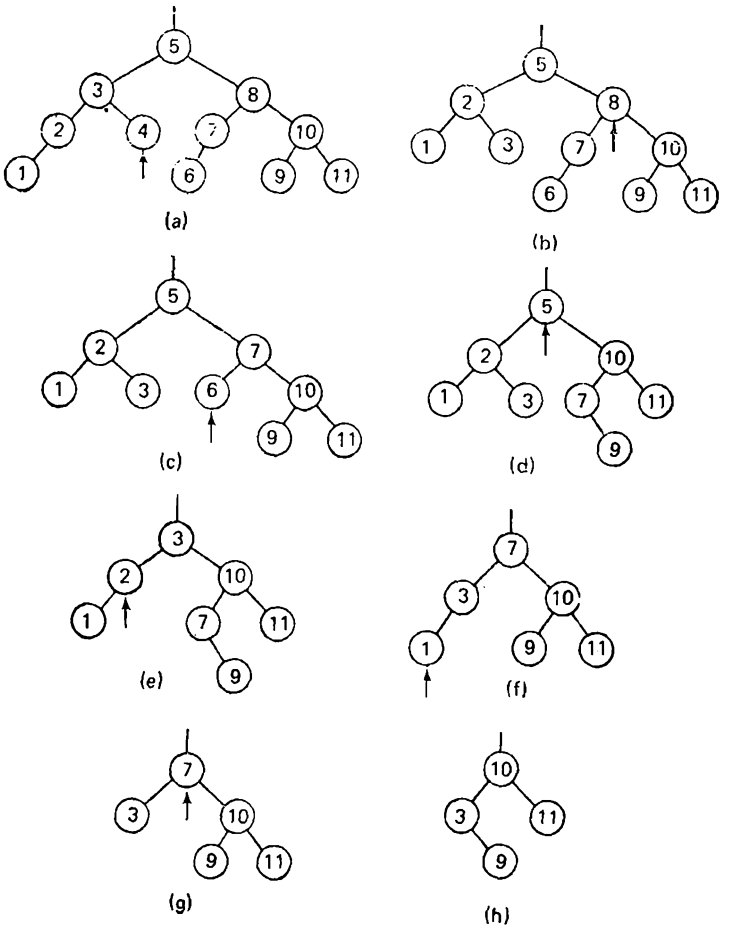


Рис. 4.35. Удаления из сбалансированного дерева.

Работа нашей процедуры иллюстрируется на рис. 4.35. Если задано сбалансированное дерево (a), то последовательное удаление узлов с ключами 4, 8, 6, 5, 2, 1 и 7 дает деревья (b), ..., (h).

Удаление ключа 4 само по себе просто, так как он представляет собой терминальный узел. Однако при этом появляется несбалансированность в узле 3. Его балансировка тре-

бует однократного поворота налево. Балансировка вновь становится необходимой после удаления узла 6. На этот раз правое поддерево корня балансируется однократным поворотом направо. Удаление узла 2, хотя само по себе просто, так как он имеет только одного потомка, вызывает сложный двукратный поворот направо и налево. И четвертый случай: двукратный поворот налево и направо вызывается удалением узла 7, который прежде заменяется самым правым элементом левого поддерева, т. е. узлом с ключом 3.

```
procedure delete(x: integer; var p: ref; var h: boolean);
  var q: ref;     {h = false}
  procedure balance1(var p: ref; var h: boolean);
    var p1,p2: ref; b1,b2: -1..+1;
  begin {h-true, левая ветвь стала короче}
    case p↑.bal of
  -1: p↑.bal := 0;
   0: begin p↑.bal := +1; h := false
      end ;
   1: begin {балансировка} p1 := p↑.right; b1 := p1↑.bal;
        if b1 ≥ 0 then
        begin {однократный RR-поворот}
          p↑.right := p1↑.left; p1↑.left := p;
          if b1 = 0 then
          begin p↑.bal := +1; p1↑.bal := -1; h := false
          end else
          begin p↑.bal := 0; p1↑.bal := 0
          end ;
          p := p1
        end else
        begin {двукратный RL-поворот}
          p2 := p1↑.left; b2 := p2↑.bal;
          p1↑.left := p2↑.right; p2↑.right := p1;
          p↑.right := p2↑.left; p2↑.left := p;
          if b2 = +1 then p↑.bal := -1 else p↑.bal := 0;
          if b2 = -1 then p1↑.bal := +1 else p1↑.bal := 0;
          p := p2; p2↑.bal := 0
        end
      end
  end
end {balance 1} ;
```

```
procedure balance2(var p: ref; var h: boolean);
   var p1,p2: ref; b1,b2: −1..+1;
begin {h-true, правая ветвь стала короче}
   case p↑.bal of
  1: p↑.bal := 0;
  0: begin p↑.bal := −1; h := false
     end ;
 −1: begin {балансировка} p1 := p↑.left; b1 := p1↑.bal;
        if b1 ≤ 0 then
        begin {однократный LL-поворот}
           p↑.left := p1↑.right; p1↑.right := p;
           if b1 = 0 then
           begin p↑.bal := −1; p1↑.bal := +1; h := false
           end else
           begin p↑.bal := 0; p1↑.bal := 0
           end ;
           p := p1
        end else
           begin {двукратный LR-поворот}
              p2 := p1↑.right; b2 := p2↑.bal;
              p1↑.right := p2↑.left; p2↑.left := p1;
              p↑.left := p2↑.right; p2↑.right := p;
              if b2 = −1 then p↑.bal := +1 else p↑.bal := 0;
              if b2 = +1 then p1↑.bal := −1 else p1↑.bal := 0;
              p := p2; p2↑.bal := 0
           end
        end
     end {balance2} ;
     procedure del(var r: ref; var h: boolean);
     begin {h = false}
        if r↑.right ≠ nil then
        begin del(r↑.right,h); if h then balance2(r,h)
        end else
        begin q↑.key := r↑.key; q↑.count := r↑.count;
           r := r↑.left; h := true
        end
     end ;
begin {delete}
   if p = nil then
      begin writeln ('KEY IS NOT IN TREE'); h := false
      end else
```

```
if x < p↑.key then
   begin delete(x,p↑.left,h); if h then balance1(p,h)
   end else
if x > p↑.key then
   begin delete(x,p↑.right,h); if h then balance2(p,h)
   end else
begin {удаление p↑} q := p;
   if q↑.right = nil then
      begin p := q↑.left; h := true
      end else                                          (4.64)
   if q↑.left = nil then
      begin p := q↑.right; h := true
      end else
   begin del(q↑.left,h);
      if h then balance1(p,h)
   end ;
   {dispose(q)}
end
end {delete}
```

Очевидно, что удаление элемента в сбалансированном дереве может также быть выполнено за (в худшем случае) $O(\log n)$ шагов. Тем не менее не следует упускать из виду существенную разницу в выполнении процедур включения и удаления. В то время как включение одного ключа может вызвать самое большее один поворот (двух или трех узлов), удаление может потребовать поворота в каждом узле вдоль пути поиска. Рассмотрим, например, удаление самого правого узла в дереве Фибоначчи. Удаление любого узла в дереве Фибоначчи вызывает уменьшение его высоты, а удаление самого правого узла требует максимального числа поворотов. Таким образом, мы получили самое неудачное сочетание: наихудший выбор узла в наиболее плохом варианте сбалансированного дерева! Но насколько вообще вероятны повороты? Удивительный результат эмпирических проверок показал, что в то время как один поворот вызывается приблизительно каждыми двумя включениями, тем не менее при удалении мы имеем дело с одним поворотом на целых пять удалений. Поэтому удаление из сбалансированного дерева примерно так же просто — или так же трудно, — как и включение.

4.4.9. Оптимальные деревья поиска

До сих пор в наших рассуждениях об организации деревьев поиска мы исходили из предположения, что частота обращения ко всем узлам одинакова, т. е. что все ключи

с равной вероятностью становятся аргументами поиска. По-видимому, это — наилучшее допущение, если нет сведений о распределении частоты обращений. Но существуют случаи (скорее исключение, чем правило), когда имеется информация о вероятности обращений к отдельным ключам. Для таких случаев обычно характерно, что ключи остаются постоянными, т. е. дерево поиска не подвергается ни включениям, ни удалениям, а сохраняет постоянную структуру. Типичным примером служит сканер транслятора, который для каждого слова (идентификатора) определяет, является ли оно

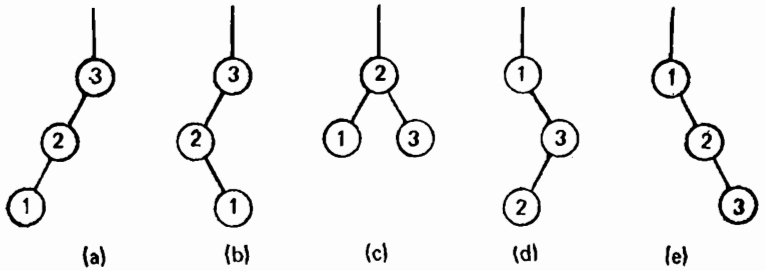


Рис. 4.36. Деревья поиска с тремя узлами.

зарезервированным (ключевым) словом. Статистические измерения, проведенные на сотнях транслируемых программ, могут в этом случае дать точные сведения о частоте появления отдельных ключей и, следовательно, о вероятности обращения к ним.

Предположим, что p_i — вероятность обращения к узлу i в дереве поиска:

$$Pr\{x = k_i\} = p_i, \quad \sum_{i=1}^{n} p_i = 1. \tag{4.65}$$

Теперь мы хотим организовать дерево поиска так, чтобы общее число шагов поиска, подсчитанное для достаточно большого количества опытов, было минимальным. Для этого припишем каждому узлу в определении длины пути (4.34) некоторый вес. Узлы, к которым часто обращаются, считаются тяжелыми, а посещаемые редко — легкими. Тогда *взвешенная длина пути* есть сумма всех путей от корня к каждому узлу, умноженных на вероятность обращения к этому узлу:

$$P_I = \sum_{i=1}^{n} p_i h_i, \tag{4.66}$$

h_i — уровень узла i (или его расстояние от корня $+1$). Наша цель — *минимизировать взвешенную длину пути* для данного распределения вероятностей.

В качестве примера рассмотрим множество ключей 1, 2, 3 с вероятностями обращения $p_1 = 1/7$, $p_2 = 2/7$ и $p_3 = 4/7$. Эти три ключа можно расположить в виде деревьев поиска пятью различными способами (см. рис. 4.36).

Взвешенные длины пути этих деревьев вычисляются в соответствии с (4.66):

$$P_I^{(a)} = \frac{1}{7}(1 \cdot 3 + 2 \cdot 2 + 4 \cdot 1) = \frac{11}{7},$$

$$P_I^{(b)} = \frac{1}{7}(1 \cdot 2 + 2 \cdot 3 + 4 \cdot 1) = \frac{12}{7},$$

$$P_I^{(c)} = \frac{1}{7}(1 \cdot 2 + 2 \cdot 1 + 4 \cdot 2) = \frac{12}{7},$$

$$P_I^{(d)} = \frac{1}{7}(1 \cdot 1 + 2 \cdot 3 + 4 \cdot 2) = \frac{15}{7},$$

$$P_I^{(e)} = \frac{1}{7}(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3) = \frac{17}{7}.$$

Итак, в этом примере оптимальным оказывается не идеально сбалансированное дерево, а вырожденное.

Пример сканера в трансляторе сразу наводит на мысль, что эту проблему следует рассматривать при несколько более общем условии: слова, встречающиеся в исходном тексте, не всегда являются зарезервированными словами; в действительности это скорее исключение, чем правило. Выяснение того, что данное слово не является ключом в дереве поиска, можно рассматривать как обращение к гипотетическому «специальному узлу», вставленному между меньшим и большим ключами (см. рис. 4.19) и имеющему соответствующую длину внешнего пути. Если известна также вероятность q_i того, что аргумент поиска x лежит между двумя ключами k_i и k_{i+1}, то это может существенно повлиять на структуру оптимального дерева поиска. Поэтому мы обобщим задачу, учитывая и неудачные поиски.

Общая взвешенная длина имеет теперь следующий вид:

$$P = \sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{m} q_j h'_j, \tag{4.67}$$

где

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{m} q_j = 1,$$

h_i — уровень внутреннего узла i, h'_j — уровень внешнего узла j. Среднюю взвешенную длину пути можно назвать «ценой» дерева поиска, так как она является мерой ожидаемого количества затрат на поиск. Дерево поиска, структура которого дает минимальную цену для всех деревьев с заданным мно-

жеством ключей k_i и вероятностями p_i и q_j обращений, называется *оптимальным деревом*.

Для нахождения оптимального дерева не обязательно, чтобы сумма всех p и q равнялась 1. На самом деле значения вероятностей обычно находятся с помощью экспериментов, в которых подсчитываются обращения к узлам. Вместо вероятностей p_i и q_i мы в дальнейшем будем использовать показатели частоты обращений, которые обозначим как

a_i = число поисков с аргументом x, равным k_i,

b_j = число поисков, когда аргумент x лежит между k_j и k_{j+1}.

По соглашению b_0 есть число поисков, когда x меньше k_1, а b_n — частота поисков, когда x больше k_n (см. рис. 4.37).

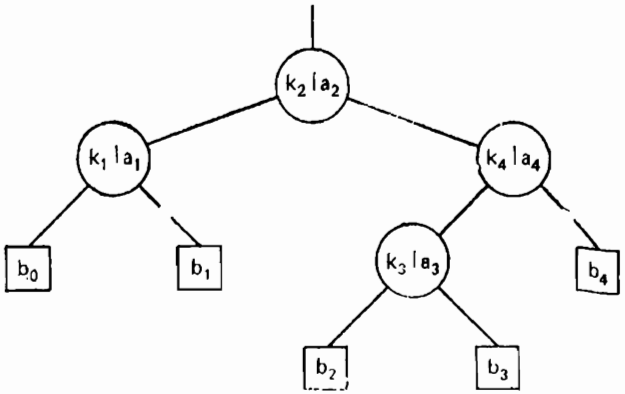


Рис. 4.37. Дерево поиска с указанием частоты обращений.

В дальнейшем мы будем через P обозначать общую взвешенную длину пути вместо средней длины пути:

$$P = \sum_{i=1}^{n} a_i h_i + \sum_{j=0}^{n} b_j h'_j. \qquad (4.68)$$

Итак, благодаря использованию экспериментально измеренных частот мы не только избавились от необходимости вычислять вероятность, но и получили возможность иметь дело только с целыми числами при нашем поиске оптимального дерева.

Если учесть, что число возможных конфигураций из n узлов растет экспоненциально вместе с n, задача нахождения оптимума при больших n кажется совершенно безнадежной. Однако оптимальные деревья имеют одно важное свойство, которое помогает их находить: все их поддеревья тоже являются оптимальными. Например, если дерево на рис. 4.37 оптимально для данных a и b, то поддерево с ключами k_3

и k_4, как известно, также оптимально. Эта особенность предполагает алгоритм, который систематически находит все бо́льшие и бо́льшие деревья, начиная с отдельных узлов как наименьших возможных поддеревьев. Таким образом, дерево растет «от листьев к корню», что является, поскольку мы привыкли рисовать деревья сверху вниз, направлением «снизу вверх» [4.6].

Основой нашего алгоритма служит уравнение (4.69). Пусть P — взвешенная длина пути дерева, а P_L и P_R — взвешенные длины левого и правого поддеревьев его корня. Понятно, что P — это сумма P_L, P_R и числа случаев, когда поиск проходит по единственному пути к корню, что является просто общим числом W случаев поиска.

$$P = P_L + W + P_R, \tag{4.69}$$

$$W = \sum_{i=1}^{n} a_i + \sum_{j=0}^{n} b_j. \tag{4.70}$$

Мы называем W *весом* дерева. Тогда его средняя длина пути будет P/W.

Из этих рассуждений видно, что необходимо обозначить веса и длины пути поддеревьев, состоящих из какого-то числа ключей. Пусть w_{ij} обозначает вес, а p_{ij} — длину пути оптимального поддерева T_{ij}, состоящего из узлов с ключами k_{i+1}, k_{i+2}, ..., k_j. Эти величины определяются рекуррентными соотношениями (4.71) и (4.72):

$$\begin{aligned} &w_{ii} = b_i && (0 \leqslant i \leqslant n), \\ &w_{ij} = w_{i,\,j-1} + a_j + b_j && (0 \leqslant i < j \leqslant n), \end{aligned} \tag{4.71}$$

$$\begin{aligned} &p_{ii} = w_{ii} && (0 \leqslant i \leqslant n), \\ &p_{ij} = w_{ij} + \min_{i<k\leqslant j} (p_{i,\,k-1} + p_{kj}) && (0 \leqslant i < j \leqslant n). \end{aligned} \tag{4.72}$$

Последнее равенство непосредственно следует из (4.69) и определения оптимальности.

Поскольку существует приблизительно $(1/2)n^2$ значений p_{ij} и так как (4.72) требует выбора среди $0 < j - i \leqslant n$ значений, поиск минимума займет приблизительно $(1/6)n^3$ операций. Кнут показал, что при помощи следующих рассуждений можно избавиться от множителя n и таким образом сохранить практическую ценность этого алгоритма.

Пусть r_{ij} — значение k, при котором достигается минимум в (4.72). Можно ограничить поиск r_{ij} намного меньшим интервалом, т. е. уменьшить число оценочных шагов $j - i$. Это основано на следующем наблюдении: если мы нашли корень r_{ij} оптимального поддерева T_{ij}, то ни добавление к дереву справа какого-либо узла, ни удаление его самого левого узла

никогда не смогут сдвинуть этот корень влево. Это свойство можно представить как

$$r_{i,\,j-1} \leqslant r_{ij} \leqslant r_{i+1,\,j}, \tag{4.73}$$

что ограничивает поиск возможных значений r_{ij} диапазоном $r_{i,j-1}, \ldots, r_{i+1,j}$ и дает общее число элементарных шагов $O(n^2)$. Теперь можно построить алгоритм оптимизации со всеми деталями. Мы используем следующие определения T — оптимальные деревья, состоящие из узлов с ключами $k_{i+1}, \ldots, k_j$:

1. a_i: частота поиска k_i.
2. b_j: частота поиска аргумента x, лежащего между k_j и k_{j+1}.
3. w_{ij}: вес T_{ij}.
4. p_{ij}: взвешенная длина пути в T_{ij}.
5. r_{ij}: индекс корня T_{ij}.

Если дан

$$\textbf{type } index = 0..n$$

мы описываем следующие массивы:

$$\begin{aligned} a&: \textbf{array}[1..n] \textbf{ of } integer;\\ b&: \textbf{array}[index] \textbf{ of } integer;\\ p,w&: \textbf{array}[index,index] \textbf{ of } integer;\\ r&: \textbf{array}[index,index] \textbf{ of } index \end{aligned} \tag{4.74}$$

Предположим, что вес w_{ij} получен из a и b очевидным способом (см. 4.71). Рассмотрим теперь w как аргумент процедуры, которую мы должны написать, а r будем считать ее результатом, так как r полностью описывает структуру. Переменную p можно рассматривать как промежуточный результат. Начиная рассмотрение с наименьших возможных поддеревьев, т. е. не содержащих вообще никаких узлов, мы переходим ко все бо́льшим и бо́льшим деревьям. Обозначим ширину $j-i$ поддерева T_{ij} через h. Тогда мы можем простым способом найти значения p_{ii} для всех деревьев с $h = 0$ согласно (4.72):

$$\textbf{for } i := 0 \textbf{ to } n \textbf{ do } p[i, i] := w[i, i] \tag{4.75}$$

В случае $h = 1$ мы имеем дело с деревьями, состоящими из одного узла, который, разумеется, является корнем (рис. 4.38):

```
for i := 0 to n - 1 do
    begin j := i + 1; p[i, j] := p[i, i] + p[j, j]; r[i, j] := j
    end
```
(4.76)

Заметим, что i обозначает левую, а j — правую границу индексов в рассматриваемом дереве T_{ij}. Для случаев $h > 1$ мы используем оператор цикла с параметром h, принимающим значения от 2 до n, причем случай $h = n$ перекрывает все дерево $T_{0,n}$. В каждом случае минимальная длина пути p_{ij} и соответствующий индекс корня r_{ij} определяются простым

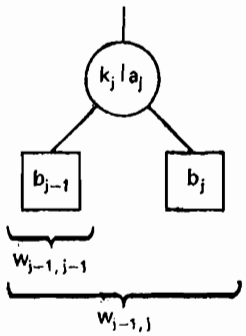


Рис. 4.38. Оптимальное дерево с одним узлом.

оператором цикла с параметром; индекс k принимает значения на интервале, заданном в (4.73):

or $h := 2$ **to** n **do**
for $i := 0$ **to** $n - h$ **do**
begin $j := i + h$; (4.77)
«найти m и $min =$ минимум$(p[i, m-1] + p[m, j])$ для всех m, таких, что $r[i, j-1] \leqslant m \leqslant r[i+1, j]$»; $p[i, j] := min + w[i, j]$; $r[i, j] := m$
end

Детальное описание оператора, заключенного в кавычки, можно найти в программе 4.6. Средняя длина пути в $T_{0,n}$ теперь определяется коэффициентом $p_{0,n}/w_{0,n}$, а его корень есть узел с индексом $r_{0,n}$.

Из алгоритма (4.77) видно, что затраты на определение оптимальной структуры имеют порядок $O(n^2)$, объем необходимой памяти составляет также $O(n^2)$. Это неприемлемо, если n очень велико. Поэтому крайне желательны алгоритмы с большей эффективностью. Один из них — алгоритм, разработанный Ху и Таккером [4.5], который требует только $O(n)$ объема памяти и $O(n \cdot \log n)$ вычислений. Но он подходит только для случаев, когда ключи имеют нулевую частоту ($a_i = 0$), т. е. когда учитываются только неудачные случаи поиска. Другой алгоритм, также требующий $O(n)$ единиц памяти и $O(n \cdot \log n)$ вычислений, был описан Уолкером

и Готлибом [4.11]. Однако с помощью этого алгоритма можно построить не оптимальное, а лишь почти оптимальное дерево. Поэтому он может основываться на *эвристических принципах*. Основная идея следующая:

Предположим, что узлы (настоящие и специальные) распределены на линейной шкале с весами, соответствующими их частотам (или вероятностям) обращения. Найдем узел, ближайший к «центру тяжести». Этот узел называется центроидом и имеет индекс

$$\frac{1}{w}\left(\sum_{i=1}^{n} i \cdot a_i + \sum_{j=0}^{n} j \cdot b_j\right), \tag{4.78}$$

округленный до ближайшего целого. Если все узлы имеют одинаковый вес, то корень искомого оптимального дерева очевидным образом совпадает с центроидом и — как нам представляется — в большинстве случаев будет находиться в близком соседстве с центроидом. Тогда на ограниченном интервале ищется локальный оптимум, после чего эта же процедура применяется к двум полученным поддеревьям. Вероятность того, что корень находится очень близко от центроида, возрастает с увеличением размера дерева n. Как только поддеревья достигают «обозримого» размера, их оптимум можно определять с помощью описанного выше точного алгоритма.

4.4.10. Изображение древовидной структуры

Теперь мы перейдем к сопутствующей задаче: как сформировать выходной файл, который *изображает* структуру дерева в достаточно ясной, графической форме, если в нашем распоряжении имеется только обычное печатающее устройство? Иначе говоря, мы хотели бы нарисовать дерево, печатая ключи в виде узлов и соединяя их соответственно горизонтальными и вертикальными линиями.

На устройстве построчной печати, входные данные которого представляются в виде текстового файла, т. е. в виде последовательности символов, мы можем «двигаться» лишь строго последовательно слева направо и сверху вниз. Поэтому кажется разумным вначале построить представление *дерева*, которое близко соответствует его топологической структуре. Затем на следующем этапе нужно упорядоченно *отобразить* эту картину на печатаемую страницу и вычислить точные координаты узлов и дуг.

Для решения первой задачи мы можем воспользоваться нашим опытом работ с алгоритмами формирования дерева, и поэтому мы без колебаний выбираем рекурсивное решение рекурсивно поставленной задачи. Мы строим функцию, назы-

ваемую *tree*, подобную той, которая использовалась в программе 4.3. Параметры *i* и *j* ограничивают значения индексов узлов, которые принадлежат дереву. Его корень определяется как узел с индексом r_{ij}. Но прежде всего нам нужно описать тип переменных, представляющих узлы. Они должны содержать две ссылки на поддеревья и ключ узла. Для целей, которые будут обсуждаться на следующем этапе, вводятся также два дополнительных поля, называемые *pos* и *link*. Описания типов показаны в (4.79), и процедура-функция включена в программу 4.6.

```
type ref = ↑node;
  node = record key: alfa;
                pos: lineposition;                (4.79)
                left, right, link: ref
         end
```

Отметим, что эта процедура подсчитывает число формируемых узлов с помощью глобальной переменной-счетчика *k*. *k*-му узлу присваивается *k*-й ключ, а поскольку ключи упорядочены в алфавитном порядке, *k*, умноженное на постоянный масштабный коэффициент, дает горизонтальную координату каждого ключа; это значение сразу запоминается вместе с остальной информацией. Заметим также, что мы отказались от соглашения, что ключи являются целыми числами, и считаем, что они относятся к типу *alfa*, означающему массивы символов определенного (максимального) размера, на которых задан алфавитный порядок.

Чтобы ясно представить себе, что мы теперь получили, рассмотрим рис. 4.39. Если дано множество из *n* ключей и вычисляемая матрица r_{ij}, то операторы

$$k := 0; \quad root := tree(0,n)$$

сформируют связанную древовидную структуру с горизонтальными позициями узлов, содержащимися в их записях, и вертикальными позициями, неявно заданными их уровнем в дереве.

Теперь мы можем перейти к следующему этапу: отображению дерева на страницу. В этом случае мы должны продвигаться строго последовательно от уровня корня вниз. На каждом шаге обрабатывается один ряд (уровень) узлов. Но каким образом мы находим узлы, лежащие в одном ряду? Для этой цели, а именно связывания вместе узлов, находящихся на одном уровне, ранее мы ввели поле записи, называемое *link*. Устанавливаемые связи показаны пунктирными линиями на рис. 4.39. На каждом этапе работы предполагается наличие цепочки, связывающей узлы, которые нужно напечатать в одном горизонтальном ряду. Мы назовем эту

цепочку *current* (текущей). Рассматривая каждый узел, мы определяем его потомков (если они имеются) и связываем их во вторую цепочку, которую мы называем *next* (следующей). Когда мы продвигаемся на один уровень вниз, цепочка *next* становится цепочкой *current*, а *next* присваивается значение **nil**.

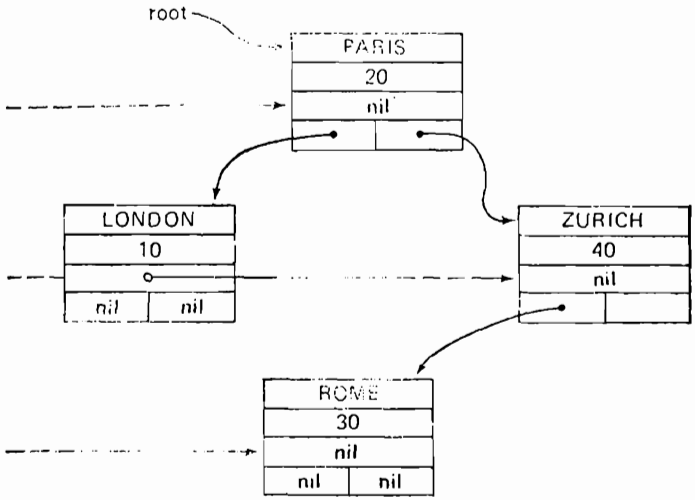


Рис. 4.39. Дерево, нарисованное с помощью программы 4.6.

Подробно этот алгоритм описан в программе 4.6. Следующие замечания могут прояснить некоторые моменты:

1. Списки (цепочки) узлов, находящихся на одном уровне, формируются слева направо, в результате самый левый узел оказывается последним. Поскольку узлы должны печататься в порядке появления, список нужно инвертировать. Это происходит в тот момент, когда цепочка *next* становится цепочкой *current*.
2. Строка, в которой печатаются ключи (главная строка), содержит также горизонтальные части «дуг» (см. рис. 4.40). Переменные $u1$, $u2$, $u3$, $u4$ обозначают начальные и конечные позиции левых и правых горизонтальных дуг узла.
3. Каждой главной строке предшествуют три строки, изображающие вертикальные части дуг.

Теперь опишем структуру программы 4.6. Ее две основные составляющие — это процедура построения оптимального дерева поиска при заданном распределении весов w и процедура изображения дерева с заданными индексами узлов r. Вся программа предназначена для обработки текстов программ,

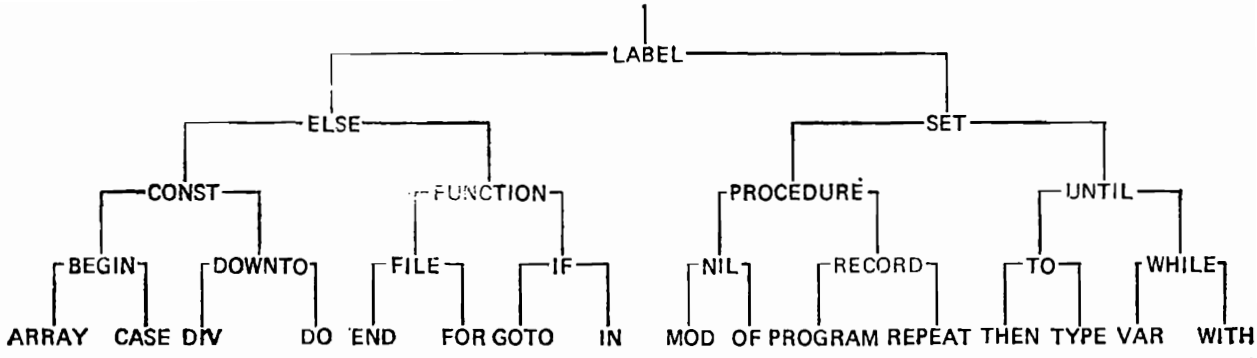


Рис. 4.40. Идеально сбалансированное дерево. (Средняя длина пути сбалансированного дерева = 5.566.)

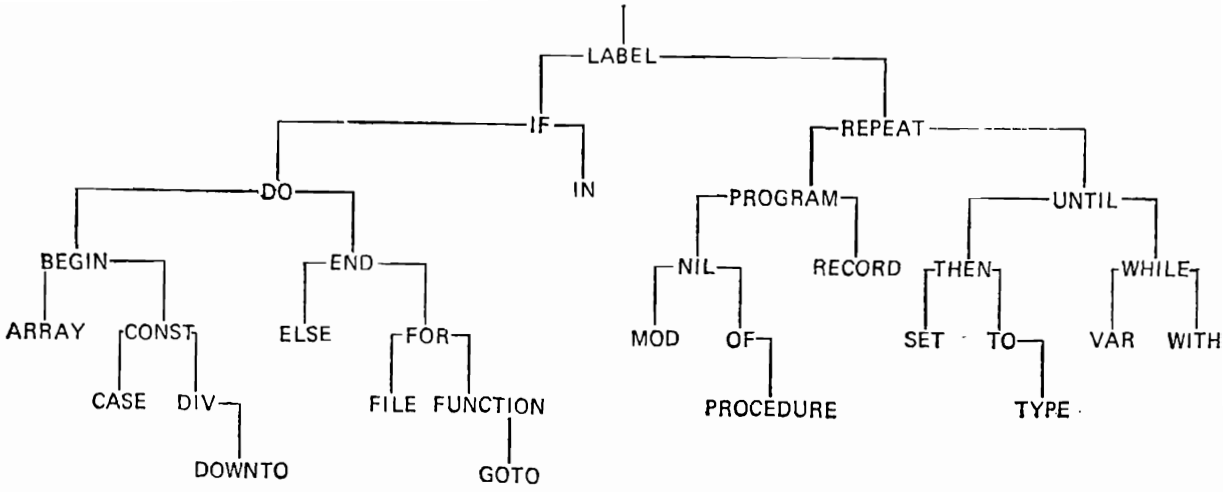


Рис. 4.41. Оптимальное дерево поиска. (Средняя длина пути оптимального дерева = 4.160.)

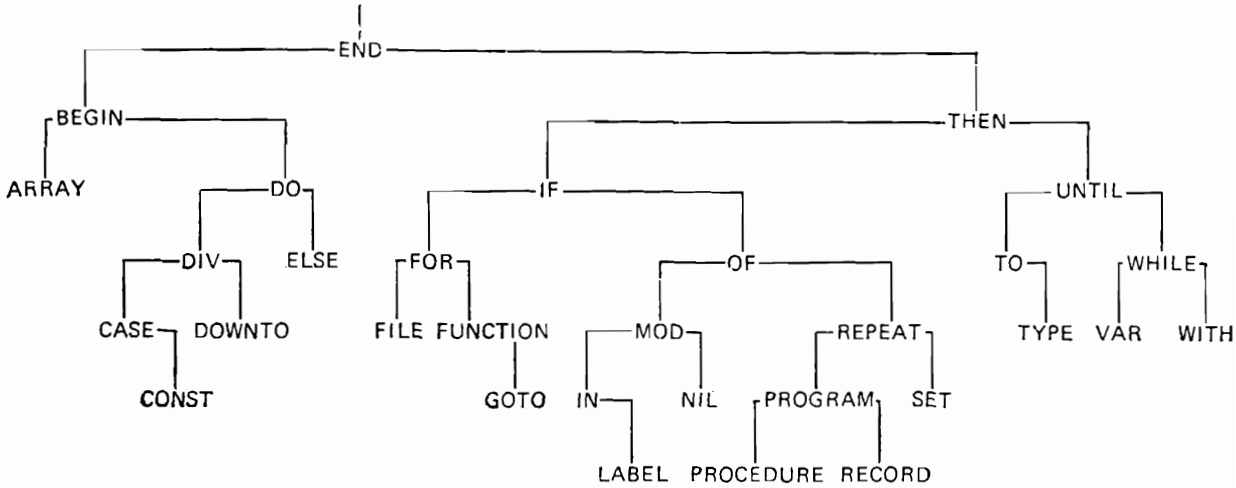


Рис. 4.42. Оптимальное дерево, построенное с учетом только ключевых слов.

в частности написанных на языке Паскаль. Такая программа вначале читается, ее идентификторы и зарезервированные слова распознаются с установкой счетчиков a_i и b_i при нахождении зарезервированного слова k_i и идентификаторов между k_i и k_{i+1}. После печати частотной статистики программа переходит к вычислению длины пути идеально сбалансированного дерева, определяя корни его поддеревьев. После этого печатается средняя длина пути и изображается полученное дерево.

Таблица 4.5. Частоты появлений ключевых слов

4	7	ARRAY
14	27	BEGIN
19	0	CASE
15	2	CONST
8	5	DIV
0	0	DOWNTO
0	20	DO
0	8	ELSE
0	28	END
1	0	FILE
0	12	FOR
0	2	FUNCTION
0	0	GOTO
9	13	IF
23	2	IN
208	0	LABEL
22	0	MOD
17	10	NIL
24	7	OF
17	2	PROCEDURE
0	1	PROGRAM
53	1	RECORD
6	8	REPEAT
16	0	SET
10	13	THEN
0	12	TO
6	2	TYPE
1	8	UNTIL
39	5	VAR
0	8	WHILE
0	0	WITH
37		
549	203	

```
program optimaltree(input,output);
const n = 31; {число ключей}
   kln = 10; {максимальная длина ключа}
type index = 0..n;
   alfa = packed array [1..kln] of char;
var ch: char;
      k1, k2: integer;
      id: alfa;    {идентификатор или служебное слово}
      buf: array [1..kln] of char;    {буфер символов}
      key: array [1..n] of alfa;
      i,j,k: integer;
      a: array [1..n] of integer;
      b: array [index] of integer;
      p,w: array [index,index] of integer;
      r: array [index,index] of index;
      suma, sumb: integer;
function baltree(i,j: index): integer;
   var k: integer;
begin k := (i+j+1) div 2; r[i,j] := k;
   if i ≥ j then baltree := b[k] else
      baltree := baltree(i,k-1) + baltree(k,j) + w[i,j]
end {baltree} ;
procedure opttree;
   var x, min: integer;
      i,j,k,h,m: index;
begin {аргумент: w, результат: p, r}
   for i := 0 to n do p[i,i] := w[i,i];  {ширина дерева h = 0}
   for i := 0 to n-1 do          {ширина дерева h = 1}
   begin j := i+1;
      p[i,j] := p[i,i] + p[j,j];  r[i,j] := j
   end ;
   for h := 2 to n do              {h-ширина текущего дерева}
   for i := 0 to n-h do            {i-левый индекс текущего дерева}
   begin j := i+h;                 {j-правый индекс текущего дерева}
      m := r[i,j-1];  min := p[i,m-1] + p[m,j];
      for k := m+1 to r[i+1,j] do
      begin x := p[i,k-1] + p[k,j];
         if x < min then
            begin m := k;  min := x
            end
      end ;
      p[i,j] := min + w[i,j];  r[i,j] := m
   end
```

```
end {opttree} ;
procedure printtree;
   const lw = 120;      {размер строки АЦПУ}
   type ref = ↑node;
      lineposition = 0 .. lw;
      node = record key: alfa;
                    pos: lineposition;
                    left, right, link: ref
             end ;
   var root, current, next: ref;
      q,q1,q2: ref;
      i, k: integer;
      u, u1, u2, u3, u4: lineposition;
   function tree(i,j: index): ref;
      var p: ref;
   begin if i = j then p := nil else
      begin new(p);
         p↑.left := tree(i, r[i,j]-1);
         p↑.pos := trunc((lw-kln)*k/(n-1)) + (kln div 2); k := k+1;
         p↑.key := key[r[i,j]];
         p↑.right := tree(r[i,j], j)
      end ;
      tree := p
   end ;
begin k := 0;  root := tree(0,n);
   current := root; root↑.link := nil;
   next := nil;
   while current ≠ nil do
   begin {продвижение вниз, сначала печать вертикальных строк}
      for i := 1 to 3 do
      begin u := 0; q := current;
         repeat u1 := q↑.pos;
            repeat write(' '); u := u+1
            until u = u1;
            write('|'); u := u+1; q := q↑.link
         until q = nil;
         writeln
      end ;
      {печать главной строки; сборка их потомков, спускаясь
      по узлам текущего списка,и формирование следующего
      списка}
```

```
        q := current; u := 0;
        repeat unpack(q↑.key, buf, 1);
          {центральный ключ} i := kln;
          while buf[i] = ' ' do i := i-1;
          u2 := q↑.pos - ((i-1) div 2); u3 := u2+l;
          q1 := q↑.left; q2 := q↑.right;
          if q1 = nil then u1 := u2 else
            begin u1 := q1↑.pos; q1↑.link := next; next := q1
            end ;
          if q2 = nil then u4 := u3 else
            begin u4 := q2↑.pos+1; q2↑.link := next; next := q2
            end ;
          i := 0;
          while u < u1 do begin write(' '); u := u+1 end ;
          while u < u2 do begin write('-'); u := u+1 end ;
          while u < u3 do begin i := i+1; write(buf[i]); u := u+1 end ;
          while u < u4 do begin write('-'); u := u+1 end ;
          q := q↑.link
        until q = nil;
        writeln;
        {инвертирование следующего списка и превращение его
        в текущий}
        current := nil;
        while next ≠ nil do
          begin q := next; next := q↑.link;
            q↑.link := current; current := q
          end
      end
    end {printtree} ;
    begin {инициация таблицы ключей и счетчиков}
      key[ 1] := 'ARRAY';    key[ 2] := 'BEGIN';
      key[ 3] := 'CASE';     key[ 4] := 'CONST';
      key[ 5] := 'DIV';      key[ 6] := 'DOWNTO';
      key[ 7] := 'DO';       key[ 8] := 'ELSE';
      key[ 9] := 'END';      key[10] := 'FILE';
      key[11] := 'FOR';      key[12] := 'FUNCTION';
      key[13] := 'GOTO';     key[14] := 'IF';
      key[15] := 'IN';       key[16] := 'LABEL';
      key[17] := 'MOD';      key[18] := 'NIL';
      key[19] := 'OF';       key[20] := 'PROCEDURE';
      key[21] := 'PROGRAM';  key[22] := 'RECORD';
      key[23] := 'REPEAT';   key[24] := 'SET';
      key[25] := 'THEN';     key[26] := 'TO';
```

```
key[27] := 'TYPE';      key[28] := 'UNTIL';
key[29] := 'VAR';       key[30] := 'WHILE';
key[31] := 'WITH';
for i := 1 to n do
   begin a[i] := 0;  b[i] := 0
   end ;
b[0] := 0; k2 := kln;
{просмотр входного текста и определение a и b}
while ¬eof(input) do
begin read(ch);
   if ch in ['A'..'Z'] then
   begin {идентификатор или служебное слово} k1 := 0;
      repeat if k1 < kln then
            begin k1 := k1+1; buf[k1] := ch
            end ;
         read(ch)
      until ¬ch in ['A'..'Z', '0'..'9'];
      if k1 ≥ k2 then k2 := k1 else
      repeat buf[k2] := ' '; k2 := k2-1
      until k2 = k1;
      pack(buf,1,id);
      i := 1; j := n;
      repeat k := (i+j) div 2;
         if key[k] ≤ id then i := k+1;
         if key[k] ≥ id then j := k-1;
      until i > j;
      if key[k] = id then a[k] := a[k] + 1 else
         begin k := (i+j) div 2; b[k] := b[k]+1
         end
   end else
     if ch = '''' then
        repeat read(ch) until ch = '''' else
     if ch = '{' then
        repeat read(ch) until ch = '}'
end ;
writeln ('KEYS AND FREQUENCIES OF OCCURRENCE:');
suma := 0; sumb := b[0];
for i := 1 to n do
begin suma := suma+a[i]; sumb := sumb+b[i];
   writeln(b[i-1], a[i], ' ', key[i])
end ;
writeln(b[n]);
```

```
    writeln('          ----------          ----------');
    writeln(sumb, suma);
    {определение w из a и b}
    for i := 0 to n do
    begin w[i,i] := b[i];
      for j := i+1 to n do w[i,j] := w[i,j-1] + a[j] + b[j]
    end ;
    write('AVERAGE PATH LENGTH OF BALANCED TREE= ');
    writeln(baltree(0,n)/w[0,n]:6:3); printtree;
    opttree;
    write('AVERAGE PATH LENGTH OF OPTIMAL TREE= ');
    writeln(p[0,n]/w[0,n]:6:3); printtree;
    {исследование только служебных слов, установив b = 0}
    for i := 0 to n do
    begin w[i,i] := 0;
      for j := i+1 to n do w[i,j] := w[i,j-1] + a[j]
    end ;
    opttree;
    writeln('OPTIMAL TREE CONSIDERING KEYS ONLY');
    printtree
end
```

Программа 4.6. Построение оптимального дерева поиска.

На третьем этапе вызывается процедура *opttree*, которая строит оптимальное дерево поиска; затем последнее изображается. И наконец, те же процедуры используются для построения и изображения оптимального дерева с учетом лишь частот обращений к ключам.

В табл. 4.5 и на рис. 4.40—4.42 приведены результаты, полученные программой 4.6 при обработке ее собственного текста. Различия трех рисунков показывают, что сбалансированное дерево нельзя считать даже близким к оптимальному и что частоты поиска незарезервированных слов сильно влияют на выбор оптимальной структуры.

4.5. СИЛЬНО ВЕТВЯЩИЕСЯ ДЕРЕВЬЯ

До сих пор мы ограничивали наши рассуждения деревьями, в которых каждый узел имеет самое большее двух потомков, т. е. бинарными деревьями. Этого вполне достаточно, если, например, мы хотим представить родственные отношения с предпочтением «восходящей линии», т. е. когда для каждого человека указываются его родители. В конце концов, ни у кого не бывает более двух родителей. Но как быть тому,

кто предпочитает изображать «нисходящую линию»? Ему придется столкнуться с тем фактом, что некоторые люди имеют более двух детей, поэтому его деревья будут содержать узлы со многими ветвями. За неимением лучшего термина мы будем называть их *сильно ветвящимися деревьями.*

Разумеется, в таких структурах нет ничего необычного, мы уже встречали все средства программирования и описания данных, нужные для того, чтобы справиться с такими ситуациями. Если, например, задана абсолютная верхняя граница количества детей (что, по-видимому, является неким футуристическим предположением), то можно представить детей в виде компоненты-массива в записи, представляющей человека. Но если число детей у разных людей сильно варьируется, то это может привести к неэкономному расходу памяти. В этом случае намного правильнее расположить потомство в виде линейного списка со ссылкой в записи родителей на самого младшего (или самого старшего) отпрыска. Возможное описание типа для такого случая показано в (4.80), а возможная структура данных приведена на рис. 4.43.

```
type person = record name: alfa;
                     sibling: ↑person;          (4.80)
                     offspring: ↑person
              end
```

Мы замечаем, что при повороте этого рисунка на 45° он будет выглядеть как идеальное бинарное дерево *). Но это неверная точка зрения, поскольку функционально эти две ссылки имеют совершенно различное значение. Обычно нельзя обращаться с братом, как с сыном, поэтому так не следует поступать даже при описании данных. Этот пример легко можно было бы распространить и на более сложные структуры данных, включив еще какие-то компоненты в запись для каждого лица; таким образом можно было бы изображать другие родственные связи. Одна из таких связей, которую нельзя в принципе вывести из отношений братства и потомства, — это связь между мужем и женой или обратное отношение отцовства и материнства. Такая структура быстро разрастается в сложный, реляционный «банк данных», который может отображаться в несколько деревьев. Алгоритмы, работающие с такими структурами, тесно связаны с их описаниями; поэтому не имеет смысла определять для них какие-либо общие правила или методы работы.

Однако имеется практически очень важная область применения сильно ветвящихся деревьев, которая *представляет*

*) О бинарности скорее свидетельствует наличие двух ссылок. — *Прим. ред.*

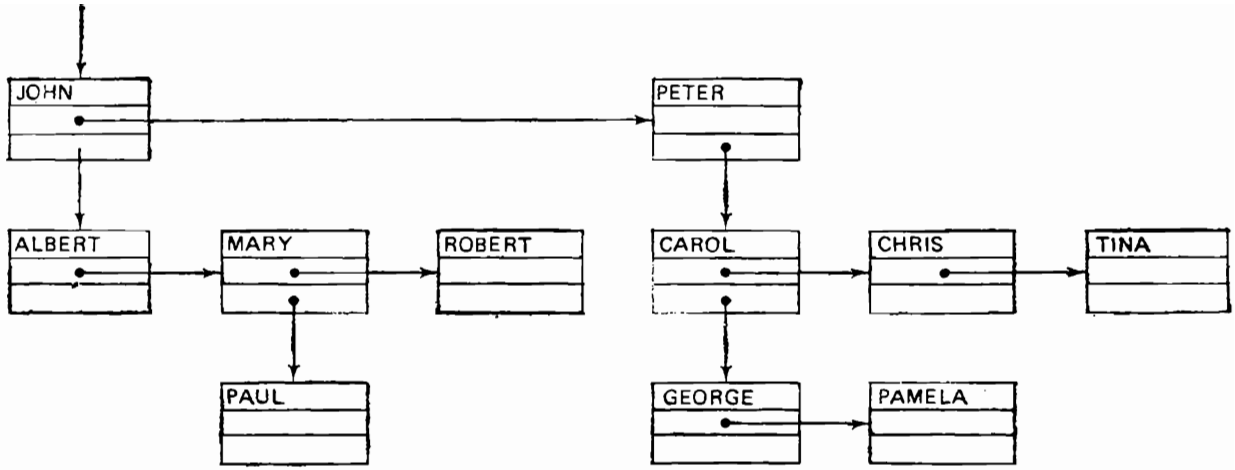


Рис. 4.43. Сильно ветвящееся дерево.

общий интерес. Это — формирование и использование крупномасштабных деревьев поиска, в которых необходимы и включения, и удаления, но для которых оперативная память недостаточно велика или слишком дорогостояща, чтобы использовать ее для долговременного хранения.

Предположим, что узлы дерева должны храниться на внешнем запоминающем устройстве, таком, как диск. Введенные в этой главе динамические структуры данных особенно подходят и для хранения на внешних запоминающих устройствах. Принципиально новое — это лишь то, что ссылки представляют собой адреса на диске, а не адреса оперативной

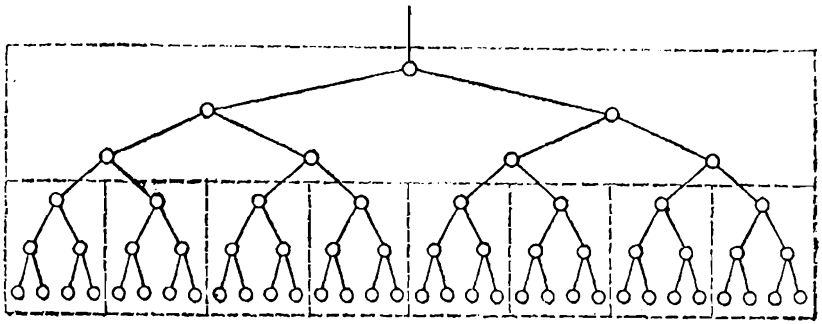


Рис. 4.44. Бинарное дерево, разделенное на «страницы».

памяти. Если множество данных, состоящее, например, из миллиона элементов, хранится в виде бинарного дерева, то для поиска элемента потребуется в среднем около $\log_2 10^6 \cong$ $\cong 20$ шагов поиска. Поскольку теперь каждый шаг включает обращение к диску (с собственным латентным временем), будет весьма желательна организация памяти, требующая меньше обращений. Сильно ветвящееся дерево является идеальным решением этой проблемы. Если происходит обращение к некоторому одиночному элементу, расположенному на внешнем устройстве, то без больших дополнительных затрат можно обращаться также к целой группе элементов. Отсюда следует, что дерево нужно разделить на поддеревья, считая, что все эти поддеревья одновременно полностью доступны. Мы будем называть такие поддеревья *страницами*. На рис. 4.44 показано бинарное дерево, разделенное на страницы, состоящие из 7 узлов каждая.

Уменьшение количества обращений к диску — а теперь обращение к каждой странице предполагает обращение к диску — может быть значительным. Предположим, что мы решили помещать на странице 100 узлов (это разумная цифра), тогда дерево поиска, содержащее миллион элементов,

потребует в среднем только $\log_{100} 10^6 = 3$ обращений к страницам вместо 20. Но конечно, если дерево растет «случайным образом», то наихудший случай может потребовать даже 10^4 обращений! Понятно, что в случае сильно ветвящихся деревьев почти обязательна схема управления их ростом.

4.5.1. Б-деревья

При поиске критерия управляемого роста нужно сразу отвергнуть идеальную сбалансированность, так как она требует слишком больших затрат на балансировку. Очевидно, что правила необходимо несколько смягчить. Очень разумный критерий был сформулирован Р. Бэйером [4.2]: каждая страница (кроме одной) содержит от n до $2n$ узлов при заданном постоянном n. Следовательно, в дереве с N элементами и максимальным размером страницы $2n$ узлов наихудший случай потребует $\log_n N$ обращений к страницам, а обращения к страницам составляют, как известно, основную часть затрат на поиск. Кроме того, важный коэффициент использования памяти составляет не менее 50 %, так как страницы заполнены хотя бы наполовину. При всех этих преимуществах данная схема требует сравнительно простых алгоритмов поиска, включения и удаления. В дальнейшем мы подробно их изучим.

Рассматриваемые структуры данных называются *Б-деревьями* и имеют следующие свойства (n называется *порядком* Б-дерева):

1. Каждая страница содержит не более $2n$ элементов (ключей).
2. Каждая страница, кроме корневой, содержит не менее n элементов.
3. Каждая страница является либо листом, т. е. не имеет потомков, либо имеет $m + 1$ потомков, где m — число находящихся на ней ключей.
4. Все листья находятся на одном и том же уровне.

На рис. 4.45 показано Б-дерево порядка 2 с 3 уровнями. Все страницы содержат 2, 3 или 4 элемента. Исключением является корень, которому разрешается содержать только один элемент. Все листья находятся на уровне 3. Ключи расположены в возрастающем порядке слева направо, если спроектировать дерево на один уровень, вставляя потомков между ключами, находящимися на странице-предке. Такое расположение представляет естественное развитие принципа организации бинарных деревьев поиска и определяет метод поиска элемента с заданным ключом. Рассмотрим страницу, имеющую вид, как показано на рис. 4.46, и пусть задан аргумент

поиска x. Предполагая, что страница считана в оперативную память, мы можем использовать известные методы поиска среди ключей $k_1, \ldots, k_m$. Если m достаточно велико, можно применить бинарный поиск, если оно сравнительно небольшое, подойдет простой последовательный поиск. (Заметим, что

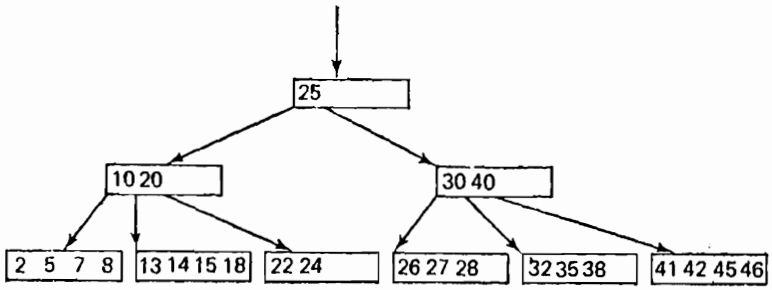
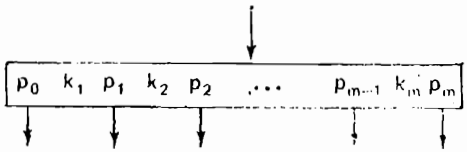


Рис. 4.45. Б-дерево порядка 2

время, требующееся для поиска в оперативной памяти, вероятно, пренебрежимо мало по сравнению со временем, которое занимает считывание страницы с внешнего устройства



Рис, 4.46. Страница Б-дерева, содержащая m ключей.

в оперативную память). Если поиск неудачен, мы имеем одну из следующих ситуаций:

1. $k_i < x < k_{i+1}$ для $1 \leqslant i < m$. Мы продолжаем поиск на странице $p_i\uparrow$.
2. $k_m < x$. Поиск продолжается на странице $p_m\uparrow$.
3. $x < k_1$. Поиск продолжается на странице $p_0\uparrow$.

Если в каком-то случае ссылка равна **nil**, т. е. нет соответствующего потомка, то элемента с ключом x нет во всем дереве и поиск заканчивается.

К удивлению, включение в Б-дерево также выполняется сравнительно просто. Если элемент вставляется в страницу, содержащую $m < 2n$ элементов, то процесс включения ограничивается этой страницей. Лишь включение в уже заполненную страницу влияет на структуру дерева и может вызвать появление новых страниц. Чтобы понять, что происходит в этом случае, рассмотрим рис. 4.47, на котором показано

включение ключа 22 в Б-дерево порядка 2. Оно состоит из следующих этапов:

1. Выясняется, что ключ 22 отсутствует. Включение в страницу C невозможно, так как C уже заполнена.
2. Страница C расщепляется на две страницы, т. е. размещается новая страница D.
3. Количество $m+1$ ключей поровну распределяется на C и D, а средний ключ перемещается на один уровень вверх, на страницу-предка A.

Эта весьма элегантная схема сохраняет все основные свойства Б-деревьев. В частности, при расщеплении получаются

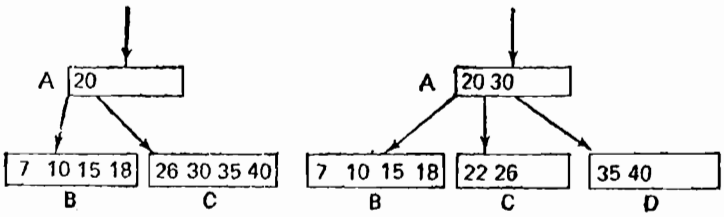


Рис. 4.47. Включение ключа 22 в Б-дерево.

страницы, содержащие ровно n элементов. Разумеется, включение элемента в страницу-предка может вновь вызвать переполнение этой страницы, что приведет к *распространению расщепления*. В экстремальном случае оно может распространиться до корня. Это и есть единственный случай увеличения высоты Б-дерева. Следовательно, Б-дерево растет странным способом: от листьев к корню.

Теперь на основе этих приблизительных описаний мы разработаем детальную программу. Уже ясно, что здесь лучше всего подойдет рекурсивная формулировка, так как процесс расщепления может распространяться назад вдоль пути поиска. Общая структура программы будет сходна со структурой программы включения в сбалансированное дерево, хотя детали будут отличаться.

Прежде всего нужно сформулировать описание страницы. Мы выбираем расположение элементов в виде массива:

```
type page = record m: index;
                   p0: ref;                    (4.81)
                   e: array[1 .. mm] of item
            end
```

где

```
const nn = 2*n;
type ref = ↑page;
     index = 0..nn
```

и

```
type item = record key: integer;
                   p: ref;
                   count: integer
            end
```
(4.82)

Здесь вновь компонента *count* заменяет всевозможную прочую информацию, которая может быть связана с каждым элементом, но не играет никакой роли в самом процессе поиска. Заметим, что каждая страница содержит пространство для $2n$ элементов. Поле m указывает, сколько элементов размещено в действительности. Поскольку $m \geqslant n$ (за исключением страницы-корня), использование памяти гарантируется по крайней мере на 50 %.

Алгоритм поиска с включением по Б-дереву является частью программы 4.7; он оформлен в виде процедуры *search*. Его основная структура проста и напоминает структуру простого поиска по бинарному дереву, с той разницей, что дальнейший путь выбирается не из двух возможных ветвей. Вместо этого «поиск внутри страницы» оформлен как бинарный поиск в массиве.

Алгоритм включения сформулирован в виде отдельной процедуры лишь для ясности. Эта процедура вызывается после того, как *search* указывает, что элемент нужно передать вверх по дереву (в направлении к корню). Для указания используется булевский параметр-результат h, он играет роль, подобную h в алгоритме включения в сбалансированное дерево, где он сообщает, что поддерево выросло. Если h истинно, то второй параметр-результат u представляет передаваемый вверх элемент. Отметим, что включение начинается с гипотетических страниц, а именно «специальных узлов» (см. рис. 4.19); новый элемент сразу отправляется через параметр u на страницу-лист для включения. Набросок схемы приведен в (4.83).

Если после вызова *search* в основной программе параметр h истинен, это означает расщепление страницы-корня. Поскольку эта страница играет особую роль, процесс нужно запрограммировать отдельно. Он состоит из размещения новой корневой страницы и включения в нее одного элемента, переданного через параметр u. Следовательно, новая стра-

```
procedure search(x: integer; a: ref; var h: boolean; var u: item);
begin if a = nil then
      begin  {x нет в дереве}
         Присвоение значения x элементу u, установка h в true,
         указывая, что элемент u передается вверх по дереву
      end else
      with a↑ do
      begin  {поиск x на странице a↑}
         двоичный поиск в массиве;
         if найдено then
           увеличение счетчика появлений элемента else
         begin search (x, потомок, h, u)
            if h then {передача вверх элемента u}
               if (число элементов на a↑) < 2n then
                  включение u в страницу a↑ и установка h в false
               else расщепление страницы и передача вверх
                    среднего элемента
         end
      end
end
```

(4.83)

ница-корень содержит только один элемент. Детали можно посмотреть в программе 4.7.

На рис. 4.48 показан результат работы программы 4.7 при построении Б-дерева со следующей последовательностью вставляемых ключей:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13
46 27 8 32; 38 24 45 25;

Точки с запятой указывают моменты размещения новых страниц. Включение последнего ключа вызывает два расщепления и размещение трех новых страниц.

Отметим особую роль в этой программе оператора присоединения **with**. Это видно уже в (4.83). В первую очередь он означает, что внутри оператора, перед которым стоит соответствующий заголовок, идентификаторы компонент (полей) страницы автоматически относятся к странице $a\uparrow$. Если реально страницы размещаются во внешней памяти, что, разумеется, необходимо в больших системах банков данных, то оператор **with** дополнительно означает передачу указанной страницы в оперативную память. Поэтому каждое обращение к *search* предполагает размещение в оперативной памяти

одной страницы, всего же необходимо самое большее $k = \log_n N$ рекурсивных обращений. Следовательно, если дерево содержит N элементов, мы должны иметь возможность разместить в оперативной памяти k страниц. Это накладывает

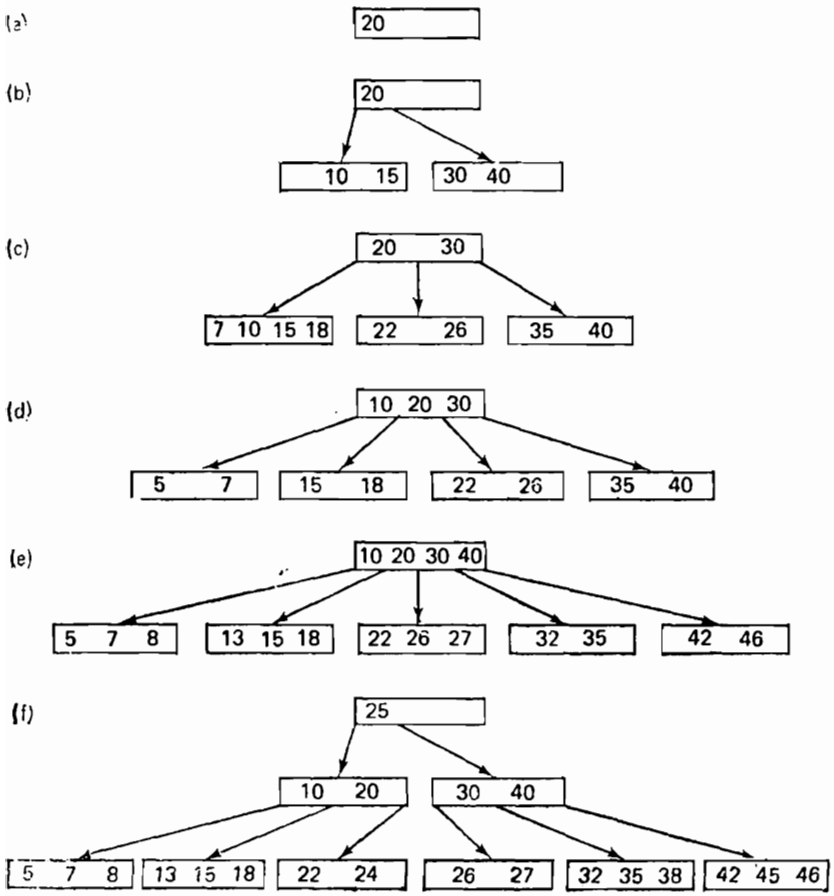


Рис. 4.48. Рост Б-дерева порядка 2.

ограничение на размер страницы $2n$. На самом деле нам нужно иметь возможность разместить даже больше, чем k страниц, так как включение может вызвать их расщепление. Естественно, что корневую страницу лучше постоянно хранить в оперативной памяти, поскольку каждый поиск всегда начинается с корня.

Еще одно положительное качество Б-деревьев — это их удобство и экономичность в случае чисто последовательного

изменения всего банка данных. При этом каждая страница вызывается в оперативную память ровно один раз.

Удаление элементов из Б-дерева очень просто в общих чертах, но сложно в деталях. Мы можем выделить два различных случая:

1. Элемент, который нужно удалить, находится на странице-листе; тогда алгоритм удаления прост и очевиден.
2. Этот элемент не на странице-листе; тогда его нужно заменить на один или два лексикографически смежных элемента, которые находятся на страницах-листьях и которые легко удалить.

В случае 2 поиск смежного ключа аналогичен поиску такого же ключа при удалении из бинарных деревьев. Мы спускаемся по самым правым указателям вниз к листу P, заменяем удаляемый элемент на самый правый элемент P и затем уменьшаем размер P на 1.

В любом случае после уменьшения размера нужно проверить число элементов m на уменьшенной странице, так как, если $m < n$, будет нарушено основное свойство Б-деревьев. Если это произошло, нужно совершить некоторые дополнительные действия; это условие *недостатка* обозначается булевской переменной-параметром h.

Единственный выход — одолжить или отобрать элемент с одной из соседних страниц, а поскольку это требует вызова страницы Q в оперативную память — относительно дорогостоящей операции, — то мы попытаемся наилучшим образом воспользоваться этой нежелательной ситуацией и заберем сразу больше одного элемента. Обычно элементы P и Q поровну распределяются на обе страницы. Это называется *балансировкой*.

Разумеется, может оказаться, что с Q нельзя забирать элементы, так как она тоже уже достигла своего минимального размера n. В этом случае общее число элементов на P и Q равно $2n - 1$, и мы можем слить эти две страницы в одну, добавив средний элемент со страницы-предка P и Q, а затем можем полностью располагать страницей Q. Это — процесс, в точности обратный расщеплению страниц. Его можно наблюдать, рассматривая удаление ключа 22 на рис. 4.47.

Удаление среднего ключа на странице-предке может вновь уменьшить ее размер ниже допустимой границы n, требуя тем самым дальнейших специальных мер (балансировки или слияния) на более высоком уровне. В экстремальном случае слияние страниц может распространиться по всему пути к корню. Если корень уменьшается до размера 0, он удаляется, что вызывает уменьшение высоты Б-дерева. Это единственный случай, когда высота Б-дерева может уменьшиться.

На рис. 4.49 показан постепенный распад Б-дерева с рис. 4.48 при последовательном удалении ключей:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26;
7 35 15;

Точки с запятой снова указывают места «скачков», т. е. освобождения страниц. Алгоритм удаления включен в программу

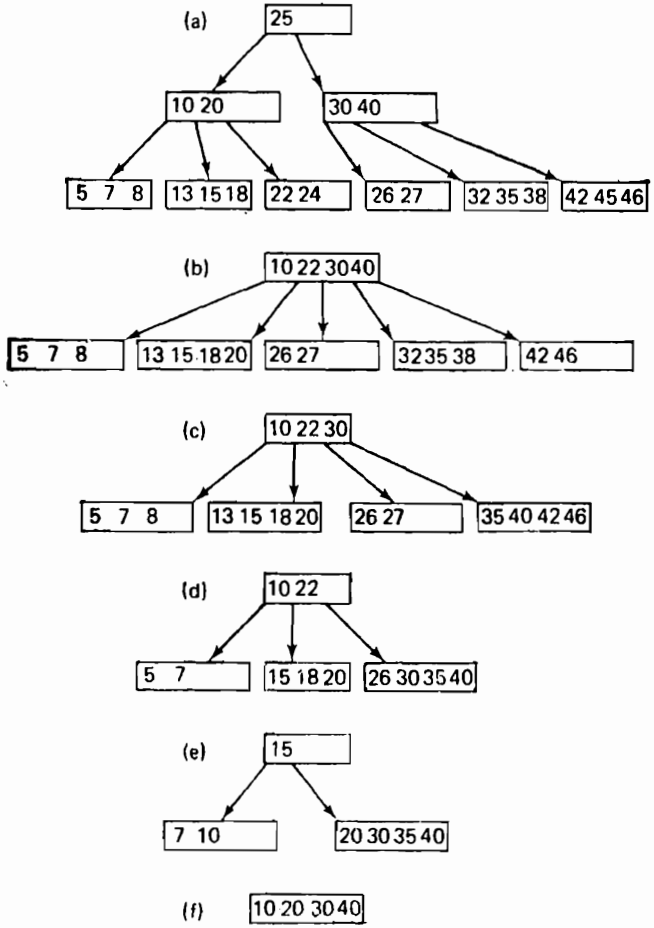


Рис. 4.49. Распад Б-дерева порядка 2.

4.7. Особенно примечательно его сходство с алгоритмом удаления из сбалансированного дерева.

Исчерпывающий анализ свойств Б-деревьев выполнен в статье Бэйера и Мак-Крейта [4.2]. В частности, в ней рас-

```
program Btree(input,output);
{поиск, включение и удаление в Б-дереве}
const n = 2; nn = 4; {размер страницы}
type ref = ↑page;
   item = record key: integer;
             p: ref;
             count: integer;
          end ;
   page = record m: 0 .. nn; {число элементов}
             p0: ref;
             e: array [1 .. nn] of item;
          end ;
var root, q: ref; x: integer;
   h: boolean; u: item;
procedure search(x: integer; a: ref; var h: boolean; var v: item);
{Поиск ключа x в Б-дереве с корнем a; если найден, увеличение
 счетчика, иначе включение в дерево элемента с ключом x и
 счетчиком 1. Если элемент должен передаваться на низший
 уровень, присвоить его v; h:-«дерево стало выше»}
   var k,l,r: integer; q: ref; u: item;
   procedure insert;
      var i: integer; b: ref;
   begin {включение и справа от a↑. e[r]}
      with a↑ do
      begin if m < nn then
         begin m := m+1; h := false;
            for i := m downto r+2 do e[i] := e[i−1];
            e[r+1] := u
         end else
         begin {страница a↑ заполнена; расщепить ее и присвоить
                полученный элемент v}          new(b);
            if r ≤ n then
            begin if r = n then v := u else
                  begin v := e[n];
                     for i := n downto r+2 do e[i] := e[i−1];
                     e[r+1] := u
                  end ;
               for i := 1 to n do b↑.e[i] := a↑.e[i+n]
            end else
            begin {включение u в первую страницу}
               r := r−n; v := e[n+1];
               for i := 1 to r−1 do b↑.e[i] := a↑.e[i+n+1];
               b↑.e[r] := u;
```

```
                for i := r+1 to n do b↑.e[i] := a↑.e[i+n]
            end ;
            m := n; b↑.m := n; b↑.p0 := v .p; v .p := b
        end
    end {with}
end {insert} ;
begin {поиск ключа x на странице a↑; h-false}
    if a = nil then
    begin {элемента с ключом x нет в дереве} h := true;
        with v do
        begin key := x; count := 1; p := nil
        end
    end else
    with a↑ do
    begin l := 1; r := m; {двоичный поиск в массиве}
        repeat k := (l+r) div 2;
            if x ≤ e[k] .key then r := k-1;
            if x ≥ e[k] .key then l := k+1;
        until r < l;
        if l-r > 1 then
        begin {найдено} e[k] .count := e[k] .count + 1; h := false
        end else
        begin {элемента нет на этой странице}
            if r = 0 then q := p0 else q := e[r] .p;
            search(x,q,h,u); if h then insert
        end
    end
end {search} ;
```

```
procedure delete(x: integer; a: ref; var h: boolean);
{поиск и удаление ключа x в Б-дереве a; если на странице
 не хватает элементов, то балансировка с соседней страницей,
 если это возможно, иначе — слияние;
 h:= «на странице a не хватает элементов»}
   var i,k,l,r: integer; q: ref;
   procedure underflow(c,a: ref; s: integer; var h: boolean);
      {a-страница с нехваткой, c-страница-предок}
      var b: ref; i,k,mb,mc: integer;
   begin mc := c↑.m;  {h = true, a↑.m = n−1}
      if s < mc then
      begin {b := страница справа от a} s := s+1;
         b := c↑.e[s].p; mb := b↑.m; k := (mb−n+1) div 2;
         {k-число элементов на соседней странице b}
         a↑.e[n] := c↑.e[s]; a↑.e[n] .p := b↑.p0;
         if k > 0 then
         begin {пересылка k элементов с b на a}
            for i := 1 to k−1 do a↑.e[i+n] := b↑.e[i];
            c↑.e[s] := b↑.e[k]; c↑.e[s] .p := b;
            b↑.p0 := b↑.e[k] .p; mb := mb−k;
            for i := 1 to mb do b↑.e[i] := b↑.e[i+k];
            b↑.m := mb; a↑.m := n−1+k; h := false
         end else
         begin {слияние страниц a и b}
            for i := 1 to n do a↑.e[i+n] := b↑.e[i];
            for i := s to mc−1 do c↑.e[i] := c↑.e[i+1];
            a↑.m := nn; c↑.m := mc−1; {dispose(b)}
            h:= c↑.m<n
         end
      end else
      begin {b := страница слева от a}
         if s = 1 then b := c↑.p0 else b := c↑.e[s−1] .p;
         mb := b↑.m + 1; k := (mb−n) div 2;
         if k > 0 then
         begin {пересылка k элементов со страница b на a}
            for i := n−1 downto 1 do a↑.e[i+k] := a↑.e[i];
            a↑.e[k] := c↑.e[s]; a↑.e[k] .p := a↑.p0; mb := mb−k;
            for i := k−1 downto 1 do a↑.e [i]:= b↑.e[i+mb];
            a↑.p0 := b↑.e[mb] .p;
            c↑.e[s] := b↑.e[mb]; c↑.e[s] .p := a;
            b↑.m := mb−1; a↑.m := n−1+k; h := false
         end else
         begin {слияние страниц a и b}
```

```
            b↑.e[mb] := c↑.e[s]; b↑.e[mb] .p := a↑.p0;
            for i := 1 to n−1 do b↑.e[i+mb] := a↑.e[i];
            b↑.m := nn; c↑.m := mc−1; {dispose(a)}
            h:= c↑.m<n
          end
        end
    end {underflow} ;
    procedure del(p: ref; var h: boolean);
      var q: ref;     {глобальный a, k}
    begin
      with p↑ do
      begin q := e[m] .p;
        if q ≠ nil then
        begin del(q,h); if h then underflow(p,q,m,h)
        end else
        begin p↑.e[m] .p := a↑.e[k] .p; a↑.e[k] := p↑.e[m];
          m := m−1; h := m<n
        end
      end
    end {del} ;
begin {delete}
    if a = nil then
    begin writeln ('KEY IS NOT IN TREE'); h := false
    end else
    with a↑ do
    begin l := 1; r := m; {двоичный поиск в массиве}
      repeat k := (l+r) div 2;
        if x ≤ e[k] .key then r := k−1;
        if x ≥ e[k] .key then l := k+1;
      until l > r;
      if r=0 then q := p0 else q := e[r] .p;
      if l−r > 1 then
      begin {найден, теперь удаление e[k]}
        if q = nil then
        begin {a-терминальная страница} m := m−1; h := m<n;
          for i := k to m do e[i] := e[i+1];
        end else
        begin del(q,h); if h then underflow(a,q,r,h)
        end
      end else
      begin delete(x,q,h); if h then underflow(a,q,r,h)
      end
```

```
    end
end {delete} ;
procedure printtree(p: ref; l: integer);
    var i: integer;
begin if p ≠ nil then
    with p↑ do
    begin for i := 1 to l do write('    ');
          for i := 1 to m do write(e[i].key: 4);
          writeln;
          printtree(p0,l+1);
          for i := 1 to m do printtree(e[i] .p, l+1)
    end
end ;
begin root := nil; read(x);
    while x ≠ 0 do
    begin writeln('SEARCH KEY', x);
        search(x,root,h,u);
        if h then
        begin {включение новой корневой страницы} q := root; new(root)
            with root↑ do
                begin m := 1; p0 := q; e[1] := u
                end
        end ;
        printtree(root,1); read(x)
    end ;
    read(x);
    while x ≠ 0 do
    begin writeln('DELETE KEY', x);
        delete(x,root,h);
        if h then
        begin {уменьшен размер корневой страницы}
            if root↑.m = 0 then
            begin q := root; root := q↑.p0; {dispose(q)}
            end
        end ;
        printtree(root,1); read(x)
    end
end
```

Программа 4.7. Поиск, включение и удаление в Б-дереве.

сматривается вопрос об оптимальном размере страницы n, который сильно зависит от характеристик памяти и вычислительной системы.

Вариации схемы Б-дерева обсуждаются в книге Кнута ([2.7], т. 3, с. 567—570). Одно важное замечание заключается в том, что расщепление страницы следует задерживать тем же способом, каким задерживается слияние страниц: балансировкой соседних страниц. Остальные предлагаемые улучшения, по-видимому, не дают ничего существенного.

4.5.2. Бинарные Б-деревья

Разновидность Б-деревьев, которая кажется наименее интересной, — Б-деревья первого порядка ($n = 1$). Но иногда стоит обратить внимание и на этот случай. Ясно, однако, что Б-деревья первого порядка бесполезны для представления больших, упорядоченных, индексированных множеств данных, требующих внешней памяти; примерно 50 % всех страниц будут содержать только один элемент. Поэтому мы забудем о внешней памяти и вновь рассмотрим деревья поиска, расположенные в *оперативной памяти*.

Бинарное Б-дерево (ББ-дерево) состоит из узлов (страниц) с одним или двумя элементами. Следовательно, страница содержит две или три ссылки на потомков, отсюда термин *2—3 дерево*. Согласно определению Б-деревьев, все листья находятся на одном уровне, а все нетерминальные страницы, в том числе корень, имеют двух или трех потомков. Поскольку теперь мы имеем дело только с оперативной памятью, то обязательно оптимальное использование памяти, и поэтому представление элементов узла в виде массива здесь не подходит. Альтернатива этому — динамическое, связанное размещение, т. е. внутри каждого узла имеется связанный список элементов длиной 1 или 2. Поскольку каждый узел имеет не более трех потомков и поэтому должен содержать самое большее три ссылки, мы попытаемся комбинировать ссылки на потомков и ссылки в списке элементов, как показано на рис. 4.50. Тем самым узел Б-дерева теряет свою целостность, и элементы выполняют роль узлов в обычном бинарном дереве. Но необходимо различать ссылки на потомков (вертикальные) и ссылки на «братьев» — элементы той же страницы (горизонтальные). Поскольку лишь ссылки направо могут быть горизонтальными, для указания этого различия достаточно одного разряда. Поэтому мы вводим булевское поле h, фиксирующее «горизонталь». Описание узла дерева, основанное на таком представлении, дано в (4.84). Оно было предложено Р. Бэйером [4.3] в 1971 г. Деревья поиска, по-

строенные из таких узлов, гарантируют максимальную длину пути $p = 2 \cdot \lceil \log N \rceil$.

```
type node = record key: integer;
                   ..........
                   left,right: ref;
                   h: boolean
            end
```
(4.84)

Рассматривая проблему включения, следует различать четыре возможные ситуации, которые возникают при увеличе-

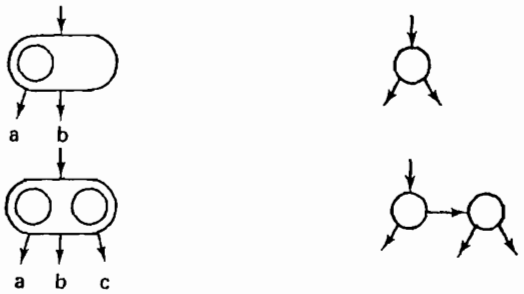


Рис. 4.50. Представление узлов ББ-дерева.

нии левого или правого поддерева. Эти четыре случая показаны на рис. 4.51. Вспомним, что Б-деревья обладают свойством расти от листьев к корню и что нужно следить, чтобы все листья оставались на одном уровне.

Самый простой случай — это (1), когда растет правое поддерево узла A и когда A — единственный ключ на (гипотетической) странице. Тогда потомок B просто становится братом A, т. е. вертикальная ссылка становится горизонтальной. Такой простой «подъем» правой ветви невозможен, если A уже имеет брата. Тогда мы получаем страницу с тремя узлами и должны расщепить ее (случай 2). Ее средний узел B передается на ближайший более высокий уровень.

Теперь предположим, что увеличилась высота левого поддерева узла B. Если узел B снова один на странице (случай 3), т. е. его правая ссылка указывает на потомка, то левое поддерево (A) может стать братом B. Нужен простой поворот ссылок, так как левая ссылка не может быть горизонтальной. Но если B уже имеет брата, подъем A дает страницу с тремя узлами, что требует расщепления. Это расщепление выполняется очень просто: C становится потомком B, который поднимается на ближайший более высокий уровень (случай 4).

Следует заметить, что при поиске ключей нет особой разницы, двигаемся мы по горизонтальной или вертикальной

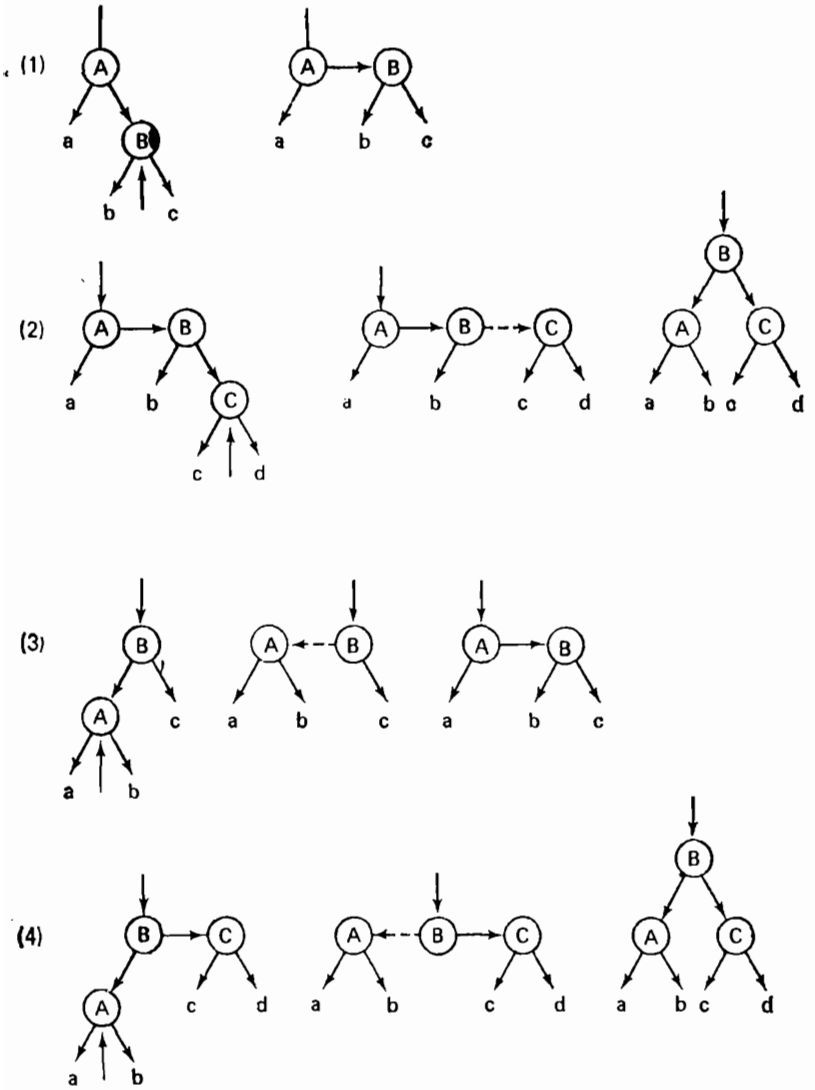


Рис. 4.51. Включение узлов в ББ-дерево.

ссылке. Поэтому забота о том, чтобы левая ссылка в случае 3 становилась горизонтальной, хотя страница по-прежнему содержит не более двух узлов, кажется надуманной. Действительно, алгоритм включения проявляет странную асимметрию

при увеличении роста левого и правого поддеревьев, поэтому организация ББ-дерева кажется несколько искусственной. У нас нет «доказательств» необычности такой организации, и лишь интуиция говорит нам, что здесь «что-то не то», и нам следует избавиться от такой асимметрии. Это ведет к понятию *симметричного бинарного Б-дерева* (СББ-дерева), которое также было предложено Бэйером [4.4] в 1972 г. Такое по-

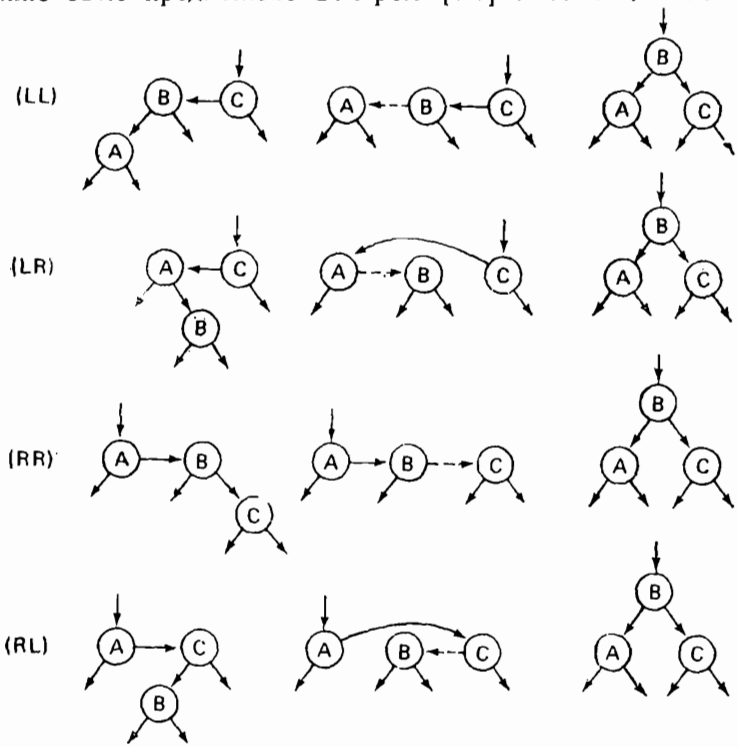


Рис. 4.52. Включение в СББ-деревья.

строение дает в среднем несколько более эффективные деревья поиска, но алгоритмы включения и удаления при этом несколько сложнее. Кроме того, теперь каждый узел требует двух разрядов (булевские переменные *lh* и *rh*) для обозначения природы его ссылок.

Поскольку мы собираемся детально остановиться на проблеме включения, то нам надо еще раз определить различия в четырех случаях роста поддеревьев. Они показаны на рис. 4.52, на котором наглядно видна полученная симметрия. Заметим, что всегда, когда растет поддерево узла *A*, не имеющего братьев, корень этого поддерева становится братом *A*. Этот случай не нуждается в дальнейшем обсуждении.

Четыре случая, приведенные на рис. 4.52, иллюстрируют переполнение страницы и последующее ее расщепление. Они отмечены в соответствии с направлениями горизонтальных ссылок, связывающих трех братьев на рисунках в среднем столбце. Исходная ситуация показана в левом столбце, в среднем столбце показано, что узел, находящийся внизу, поднимается с ростом поддерева; на рисунках правого столбца показан результат перестановки узлов при расщеплении страницы.

Желательно больше не возвращаться к понятию страниц, на основе которого разработана эта организация, так как все, к чему мы стремимся, — это ограничить максимальную длину пути поиска значением $2 \cdot \log N$. Для этого нужно только, чтобы нигде на пути поиска не встречались две последовательные горизонтальные ссылки. Но нет причины запрещать любые узлы с горизонтальными ссылками налево и направо. Поэтому мы определяем СББ-дерево как дерево со следующими свойствами:

1. Каждый узел содержит один ключ и не более двух поддеревьев (ссылок).
2. Каждая ссылка либо горизонтальная, либо вертикальная. Ни на каком пути поиска нет двух последовательных горизонтальных ссылок.
3. Все терминальные узлы (узлы, не имеющие потомков) находятся на одном (терминальном) уровне.

Из этого определения следует, что самый длинный путь поиска не более чем в два раза превосходит высоту дерева. Так как никакое СББ-дерево с N узлами не может иметь высоту, бо́льшую $[\log N]$, то $2 \cdot [\log N]$ является верхним пределом длины пути поиска.

Чтобы читатель наглядно представил себе, как растут эти деревья, мы отсылаем его к рис. 4.53. На нем показаны изменения четырех деревьев при последовательных включениях элементов с ключами, перечисленными в строках (4.85), где точки с запятой отмечают моменты увеличения высоты дерева:

$$\begin{array}{ll} (1)\ 1\ 2;\ 3;\ 4\ 5\ \ 6;\ 7; & \\ (2)\ 5\ 4;\ 3;\ 1\ 2\ \ 7\ \ 6; & (4.85) \\ (3)\ 6\ 2;\ 4;\ 1\ 7\ \ 3\ \ 5; & \\ (4)\ 4\ 2\ \ 6;\ 1\ 7;\ 3\ \ 5; & \end{array}$$

Эти рисунки особенно наглядно иллюстрируют третье свойство Б-деревьев: все терминальные узлы находятся на одном уровне. Поэтому хочется сравнить эти структуры с только что подстриженными садовыми кустарниками. Мы будем называть такие структуры *кустарниками*.

Алгоритм построения кустарников сформулирован в (4.87). Он основан на определении типа узла (4.86) с двумя компонентами *lh* и *rh*, обозначающими горизонтальность левой и правой ссылок.

```
type node = record key: integer;
                   count: integer;
                   left,right: ref;
                   lh,rh: boolean
            end
```
(4.86)

Рекурсивная процедура *search* вновь строится по основной схеме алгоритма включения в бинарное дерево [см. (4.87)]. Добавляется третий параметр h; он указывает, изменилась ли структура поддерева с корнем p, и полностью соответствует параметру h в программе поиска в Б-дереве. Однако нужно отметить последствия представления «страниц» в виде связанных списков: проход любой страницы происходит с помощью одного или двух обращений к процедуре поиска. Мы должны различать два случая: когда поддерево (обозначенное вертикальной ссылкой) выросло, и когда узел-брат (обозначенный горизонтальной ссылкой) получил другого брата и, следовательно, требуется расщепление. Эта проблема легко решается введением следующих трех значений h:

1. $h = 0$: никаких изменений структуры дерева не требуется.
2. $h = 1$: узел p получил брата.
3. $h = 2$: поддерево p увеличилось в высоте.

Заметим, что действия, предпринимаемые для переупорядочения узлов, очень напоминают те, которые были разработаны для алгоритма поиска в сбалансированном дереве (4.63). Из (4.87) видно, что все четыре случая можно реализовать простыми поворотами ссылок: однократными поворотами налево или направо и двукратными поворотами налево и направо или направо и налево. В самом деле, процедура (4.87) оказывается несколько проще, чем (4.63). Ясно, что схема кустарниковых деревьев является альтернативой критерию АВЛ-сбалансированности. Поэтому возможно и желательно сравнение их характеристик.

Мы отказываемся от анализа точными, математическими методами и обратим основное внимание на некоторые существенные различия. Можно доказать, что *АВЛ-сбалансированные деревья являются подмножеством кустарниковых деревьев.* Таким образом, класс последних шире. Отсюда следует, что их длина пути в среднем больше, чем у АВЛ-деревьев. Отметим, что в этом отношении «наихудший слу-

```
procedure search(x: integer; var p: ref; var h: integer);
   var p1,p2: ref;
begin
   if p = nil then
   begin {слова нет в дереве, вставить его}
      new(p); h := 2;
      with p↑ do
      begin key := x; count := 1; left := nil;
         right := nil; lh := false; rh := false
      end
   end else
   if x < p↑.key then
   begin search(x,p↑.left,h);
      if h ≠ 0 then
      if p↑.lh then
      begin p1 := p↑.left; h := 2; p↑.lh := false;
         if p1↑.lh then
         begin {LL} p↑.left := p1↑.right;
            p1↑.right := p; p1↑.lh := false; p := p1
         end else
         if p1↑.rh then
         begin {LR} p2 := p1↑.right; p1↑.rh := false;
            p1↑.right := p2↑.left; p2↑.left := p1;
            p↑.left := p2↑.right; p2↑.right := p; p := p2
         end
      end else
      begin h := h-1; if h ≠ 0 then p↑.lh := true
      end
   end else
   if x > p↑.key then
   begin search(x,p↑.right,h);
      if h ≠ 0 then
      if p↑.rh then
      begin p1 := p↑.right; h := 2; p↑.rh := false;
         if p1↑.rh then
         begin {RR} p↑.right := p1↑.left;
            p1↑.left := p; p1↑.rh := false; p := p1
         end else
         if p1↑.lh then
         begin {RL} p2 := p1↑.left; p1↑.lh := false;
            p1↑.left := p2↑.right; p2↑.right := p1;
            p↑.right := p2↑.left; p2↑.left := p; p := p2
         end
```

```
            end else
            begin h := h−1, if h ≠ 0 then p↑.rh := true
            end
        end else
        begin p↑.count := p↑.count + 1; h := 0
        end
    end {search}
```

(4.87)

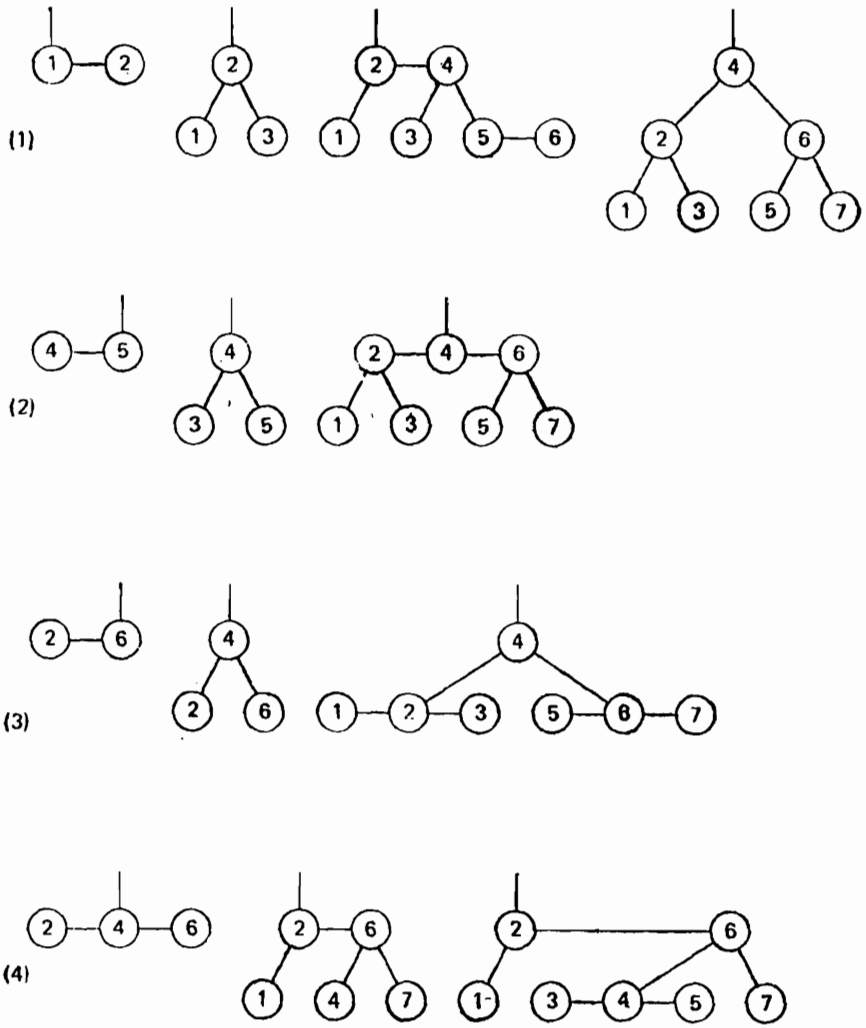


Рис. 4.53. Формирование «кустарниковых» деревьев при последовательностях включений (4.85).

чай» — дерево (4) на рис. 4.53. С другой стороны, в кустарниковых деревьях перестройка узлов будет происходить реже. Поэтому сбалансированные деревья предпочтительны в тех случаях, когда поиск ключей происходит намного чаще, чем включение (или удаление). Если это соотношение умеренное, можно предпочесть схему кустарниковых деревьев.

Очень трудно сказать, где проходит граница. Это во многом зависит не только от соотношения между частотой поиска и частотой изменения структуры, но и от особенностей реализации. В частности, записи узлов могут иметь плотно упакованное представление, следовательно, обращение к полям потребует выборки части слова. Во многих реализациях работа с булевскими полями (*lh*, *rh* в случае кустарниковых деревьев) может быть более эффективной, чем с полями из трех значений (*bal* в случае сбалансированных деревьев).

4.6. ПРЕОБРАЗОВАНИЯ КЛЮЧА (РАССТАНОВКА)

Сформулируем основную задачу, к которой мы обращаемся в последнем разделе, в дополнение к задачам, демонстрирующим методы динамического размещения данных:

> Задано множество S элементов, характеризующихся значениями ключей, на которых задано отношение порядка. Как организовать S, чтобы поиск элемента с заданным ключом k требовал как можно меньше затрат?

Очевидно, что в памяти ЭВМ к каждому элементу в конце концов обращаются с помощью его адреса a в памяти. Следовательно, поставленная задача — это в сущности задача нахождения подходящего отображения H ключей (K) в адреса (A).

$$H: K \rightarrow A$$

В предыдущих разделах это отображение было реализовано в виде алгоритмов поиска по спискам и деревьям, основанных на различных способах их организации. Здесь мы предлагаем другой, более простой и во многих случаях очень эффективный подход. Тот факт, что он также имеет некоторые недостатки, будет обсуждаться позже.

В этом методе используется организация данных в виде массива. Поэтому H — это отображение, которое преобразует ключи в индексы массива, что дало термин *преобразование ключей*, обычно применяемый для обозначения этого приема. Следует заметить, что нам не придется использовать какие-либо процедуры динамического размещения, поскольку массив — одна из базовых, статических структур. Таким образом, этот раздел несколько чужероден в главе о динамических

информационных структурах, но поскольку преобразование ключей часто используется в той же области, где наряду с ним в качестве конкурентов используются древовидные структуры, то, по-видимому, его обсуждение здесь уместно.

Основная трудность преобразования ключей заключается в том, что множество возможных значений ключей намного обширнее, чем множество имеющихся адресов памяти (индексов массива). Типичный пример — использование слов длиной, скажем, до 10 букв алфавита в качестве ключей для идентификации индивидуумов в множестве, например, до тысячи человек. Следовательно, имеются 26^{10} возможных ключей, которые нужно отобразить в 10^3 возможных индексов. Поэтому очевидно, что H — это функция, отображающая «много в один». Если дан некоторый ключ k, то первый этап в операции поиска — это вычисление соответствующего индекса $h = H(k)$, а второй — очевидно, необходимый этап — проверка, действительно ли элемент с ключом k находится в массиве (таблице) T по адресу h, т. е. проверка $T[H(k)] \cdot key = k$. Сразу возникают два вопроса:

1. Какую функцию H следует использовать?
2. Как поступать в ситуации, когда H не дает местонахождения нужного элемента?

Ответ на второй вопрос заключается в том, что нужно использовать какой-то метод для получения нового адреса, скажем, с индексом h', а если там вновь нет нужного элемента, то с третьим индексом h'' и т. д. Случай, когда на указанном месте находится другой ключ, а не искомый, называется *конфликтом*, задача получения альтернативных индексов называется *разрешением конфликтов*. Далее мы обсудим выбор функции преобразования и методы разрешения конфликтов.

4.6.1. Выбор функции преобразования

Основное требование к хорошей функции преобразования состоит в том, чтобы она распределяла ключи как можно более равномерно по шкале значений индексов. Кроме выполнения этого требования, распределение не связано никакой схемой, и даже желательно, чтобы оно производило впечатление совершенно случайного. Такая особенность дала этому методу несколько ненаучное название «расстановки» (хеширования), а H называется *функцией расстановки* *). Разумеется, она должна эффективно вычисляться, т. е. состоять

*) Следует отметить, что в советской литературе впервые этот метод был описан А. П. Ершовым и назван им «функциями расстановки». — *Прим. ред.*

из очень небольшого числа основных арифметических действий.

Предположим, что имеется функция $ord(k)$, которая определяет порядковый номер ключа k во множестве всех возможных значений ключей. Предположим далее, что индексы массива занимают интервал целых чисел $0 \ldots N-1$, где N — размер массива. Тогда очевидным решением является

$$H(k) = ord(k) \bmod N \qquad (4.88)$$

Эта функция обладает тем свойством, что значения ключей равномерно распределяются на всем интервале индексов, поэтому она служит основой большинства преобразований ключей. Кроме того, она очень эффективно вычисляется при N, равном степени двойки. Но как раз этого случая следует избегать, если ключи являются последовательностями букв. Предположение, что все ключи равновероятны, в этом случае совершенно ошибочно. В результате слова, различающиеся лишь несколькими символами, будут с большой вероятностью отображаться в одинаковые индексы, что приведет к чрезвычайно неравномерному распределению. Поэтому рекомендуется, чтобы N в (4.88) было *простым числом* [4.7]. Отсюда следует, что придется выполнять операцию целого деления, которую нельзя заменить простым маскированием двоичных разрядов. Но это не является препятствием для большинства современных вычислительных машин, которые имеют встроенную команду деления.

Часто употребляется свертка, состоящая из выполнения логических операций, таких, как «исключающее или» на некоторых частях ключа, представленного последовательностью двоичных цифр. Эти операции на некоторых машинах могут выполняться быстрее, чем деление, но не всегда можно быть уверенным, что они равномерно распределяют ключи на интервале индексов. Поэтому мы не будем подробно обсуждать такие методы.

4.6.2. Разрешение конфликтов

Если строка в таблице, соответствующая заданному ключу, не содержит нужный элемент, то имеет место конфликт, т. е. два элемента имеют ключи,отображающиеся в один и тот же индекс. Нужна вторая проба с использованием другого индекса, который однозначно получается на основе данного ключа. Существует несколько методов получения таких вторичных индексов. Очевидный и эффективный метод *) — свя-

*) Это, конечно, метод разрешения конфликтов, а не метод получения вторичных индексов. — *Прим. ред.*

зывание в цепочку всех элементов с одинаковым первичным индексом $H(k)$. Этот прием называется *непосредственным сцеплением*. Элементы такого списка могут либо находиться в первичной таблице, либо нет, в последнем случае память, в которой они размещаются, называется *областью переполнения*. Этот метод довольно эффективен, хотя он имеет тот недостаток, что нужно вести вторичные списки и что каждая строка должна содержать пространство для ссылки (или индекса) на список элементов, вступающих с ним в конфликт.

Другой метод разрешения конфликтов состоит в том, чтобы полностью отказаться от ссылок и просто просматривать один за другим различные элементы таблицы, пока не будет найден нужный элемент или не встретится свободное место, что означает отсутствие в таблице данного ключа. Этот метод называется *открытой адресацией* [4.9]. Разумеется, последовательность индексов при вторичных пробах для данного ключа должна быть всегда одной и той же. Можно набросать примерно такой алгоритм просмотра таблицы:

```
h := H(k); i := 0;
repeat
    if T[h].key = k then элемент найден else
    if T[h].key = свободно  then  элемента нет в
                    таблице else                    (4.89)
    begin {конфликт}
        i := i + 1, h := H(k) + G(i)
    end
until найден или нет в таблице (или таблица полна)
```

Для разрешения конфликтов в литературе предлагались различные функции. Обзор этой темы Моррисом в 1968 г. [4.8] стимулировал активную деятельность в этой области. Простейший метод — это, считая, что таблица круговая, исследовать следующее место и так до тех пор, пока не будет найден элемент с заданным ключом или не встретится свободное место. Следовательно, $G(i) = i$; индексы h_i, используемые для поиска, в этом случае имеют вид:

$$\begin{aligned} h_0 &= H(k), \\ h_i &= (h_0 + i) \bmod N, \quad i = 1 \ldots N - 1. \end{aligned} \tag{4.90}$$

Этот метод называется *линейным опробированием*, он имеет тот недостаток, что элементы обычно скапливаются вокруг первичных ключей (ключей, при которых мы не столкнулись с конфликтом при включении). В идеале, конечно, следует выбирать такую функцию G, которая вновь равномерно рассеивает ключи на оставшемся пространстве. Но на практике это требует слишком больших затрат, и предпочтение от-

дается методам, представляющим компромисс; будучи простыми для вычисления, они все же лучше линейной функции (4.90). Один из них состоит в использовании квадратичной функции, такой, что последовательность индексов для опробирования есть

$$\begin{aligned} h_0 &= H(k), \\ h_i &= (h_0 + i^2) \bmod N \quad (i > 0). \end{aligned} \tag{4.91}$$

Отметим, что вычисление следующего индекса не требует возведения в квадрат, если использовать рекуррентное соотношение (4.92) для $h_i = i^2$ и $d_i = 2i + 1$:

$$\begin{aligned} h_{i+1} &= h_i + d_i \\ d_{i+1} &= d_i + 2 \end{aligned} \quad (i > 0) \tag{4.92}$$

с $h_0 = 0$ и $d_0 = 1$. Этот метод называется *квадратичным опробированием*, он успешно позволяет избежать первичного скопления, хотя практически не требует никаких дополнительных вычислений. Небольшой недостаток заключается в том, что при опробировании рассматриваются не все строки таблицы, так что при включении можно не встретить свободного места, хотя такие места еще остаются. Фактически если ее размер N — простое число, то при квадратичном опробировании используется по меньшей мере половина таблицы. Это можно получить из следующих рассуждений: если i-я и j-я пробы приводят к одной и той же строке таблицы, то справедливо равенство

$$i^2 \bmod N = j^2 \bmod N$$

или

$$i^2 - j^2 \equiv 0 \pmod N.$$

Разбивая разность на два множителя, мы получаем

$$(i + j)(i - j) \equiv 0 \pmod N.$$

Поскольку $i \neq j$, мы видим, что либо i, либо j должно быть не менее $N/2$, чтобы получить $i + j = cN$, где c — целое число.

На практике этот недостаток не так существенен, поскольку необходимость выполнять $N/2$ вторичных проб для разрешения конфликтов встречается очень редко и лишь в том случае, когда таблица уже почти заполнена.

Чтобы на примере продемонстрировать метод рассеянных таблиц, мы переписали программу 4.5 — формирование таблицы перекрестных ссылок — в виде программы 4.8. Принципиальное отличие заключается в формулировке процедуры поиска (*search*) и в замене ссылочного типа *wordref* таблицей слов T. Функция расстановки H есть модуль размера таблицы; для разрешения конфликтов выбрано квадратичное

```
program crossref(f,output);
{построение таблицы перекрестных ссылок с использованием
расстановки}
label 13;
const c1 = 10;     {длина слова}
      c2 = 8;      {количество слов в строке}
      c3 = 6;      {количество цифр в числе}
      c4 = 9999;   {максимальный номер строки}
      p = 997;     {простое число}
      free = ' ';
type  index = 0..p;
      itemref = ↑item;
      word = record key: alfa;
                    first, last: itemref;
                    fol: index
             end;
      item = packed record
                    lno: 0..c4;
                    next: itemref
             end;
var i, top: index;
    k,k1: integer;
    n: integer;          {номер текущей строки}
    id: alfa;
    f: text;
    a: array [1..c1] of char;
    t: array [0..p] of word;      {массив для расстановки}
procedure search;
  var h,d,i: index;
      x: itemref; f: boolean;
  {глобальные переменные: t, id, top}
begin h := ord(id) mod p;
  f := false; d := 1;
  new(x); x↑.lno := n; x↑.next := nil;
  repeat
    if t[h].key = id then
      begin {найдено} f := true;
        t[h].last↑.next := x; t[h].last := x
      end else
    if t[h].key = free then
      begin {новый элемент} f := true;
        with t[h] do
        begin key := id; first := x; last := x; fol := top
```

```
            end ;
            top := h
          end else
          begin {конфликт} h := h+d; d := d+2;
            if h ≥ p then h := h-p;
            if d = p then
              begin writeln('TABLE OVERFLOW'); goto 13
              end
          end
    until f
end {search} ;
procedure printtable;
    var i,j,m: index;
    procedure printword(w: word);
        var l: integer; x: itemref;
    begin write('  ', w.key);
        x := w.first; l := 0;
        repeat if l = c2 then
                 begin writeln;
                   l := 0; write('  ':c1+1)
                 end ;
          l := l+1; write(x↑.lno:c3); x := x↑.next
        until x = nil;
        writeln
    end {printword} ;
begin i := top;
    while i ≠ p do
    begin {просмотр связанного списка и поиск минимального ключа}
        m := i; j := t[i].fol;
        while j ≠ p do
          begin if t[j].key < t[m].key then m := j;
            j := t[j].fol
          end ;
        printword(t[m]);
        if m ≠ i then
          begin t[m].key := t[i].key;
            t[m].first := t[i].first; t[m].last := t[i].last
          end ;
        i := t[i].fol
    end
end {printtable} ;
begin n := 0; k1 := c1; top := p; reset(f);
```

```
    for i := 0 to p do t[i].key := free;
    while ¬eof(f) do
    begin if n = c4 then n := 0;
       n := n+1; write(n:c3);      {следующая строка}
       write(' ');
       while ¬eoln(f) do
       begin {просмотр непустой строки}
          if f↑ in ['A'..'Z'] then
          begin k := 0;
             repeat if k < c1 then
                begin k := k+1;  a[k] := f↑;
                end ;
                write(f↑); get(f)
             until ¬(f↑ in ['A'..'Z', '0'..'9']);
             if k ≥ k1 then k1 := k else
             repeat a[k1] := ' '; k1 := k1-1
             until k1 = k;
             pack(a,1,id); search;
          end else
          begin {проверка на кавычку или комментарий}
             if f↑ = '''' then
                repeat write(f↑); get(f)
                until f↑ = '''' else
             if f↑ = '{' then
                repeat write(f↑); get(f)
                until f↑ = '}' ;
             write(f↑); get(f)
          end
       end ;
       writeln; get(f)
    end ;
 13: page; printable
 end .
```

Программа 4.8. Построение таблицы перекрестных ссылок с использованием функций расстановки.

опробирование. Отметим, что для эффективной работы существенно, чтобы размер таблицы был простым числом.

Несмотря на то что метод преобразования ключа в этом случае очень эффективен — намного эффективнее, чем использование деревьев, — он имеет недостаток. После просмотра текста и выбора слов мы хотим расположить эти слова

в алфавитном порядке. При работе с деревьями это очень просто, так как в их основе уже лежит упорядоченность. Но это не так в случае преобразования ключа. Вот здесь и сказывается, что таблицы — «рассеянные». Поэтому печати таблицы должна предшествовать сортировка (для простоты в программе 4.8 используется сортировка простым выбором); но кроме того, оказывается полезным сохранять историю включения элементов, для чего они связываются в специальный список. Поэтому преимущества метода расстановки при поиске отчасти уменьшаются из-за необходимости дополнительных действий при выполнении всей задачи построения упорядоченной таблицы перекрестных ссылок.

4.6.3. Анализ метода преобразования ключа

Очевидно, что в наихудшем случае включение и поиск при использовании метода расстановки будут иметь очень плохие характеристики. Ведь вполне может быть, что аргумент поиска будет таким, что все пробы будут попадать как раз на занятые места и пропускать нужные (или свободные). В самом деле, тому, кто использует метод расстановки, нужно свято верить в теорию вероятностей. Всё, в чем мы хотим быть уверенными, — это что *в среднем* число проб мало. Следующие вероятностные рассуждения показывают, что оно даже *очень мало*.

Вновь предположим, что все возможные ключи равновероятны и функция расстановки H равномерно рассеивает их по всему интервалу индексов. Затем предположим, что нужно вставить ключ в таблицу размером n, которая уже содержит k элементов. Вероятность попадания на свободное место с первого раза в этом случае равна $1 - k/n$. Одновременно это есть вероятность p_1 того, что потребуется только одно сравнение. Вероятность, что понадобится ровно одна вторая проба, равна вероятности конфликта при первой попытке, умноженной на вероятность попадания на свободное место в следующий раз. В целом мы получаем вероятность p_i того, что включение потребует i проб:

$$
\begin{aligned}
p_1 &= \frac{n-k}{n}, \\
p_2 &= \frac{k}{n} \cdot \frac{n-k}{n-1}, \\
p_3 &= \frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \frac{n-k}{n-2}, \\
&\cdots\cdots\cdots\cdots \\
p_i &= \frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \frac{k-2}{n-2} \cdot \ldots \cdot \frac{k-i+2}{n-i+2} \cdot \frac{n-k}{n-i+1}.
\end{aligned}
\tag{4.93}
$$

Следовательно, среднее значение числа проб, требующихся при вставке $(k+1)$-го ключа, равно

$$E_{k+1}=\sum_{i=1}^{k+1} i\cdot p_i = 1\cdot\frac{n-k}{n}+2\frac{k}{n}\cdot\frac{n-k}{n-1}+\ldots$$
$$\ldots+(k+1)\cdot\left(\frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{k-2}{n-2}\cdots\frac{1}{n-k+1}\right)=\frac{n+1}{n-k+1}. \quad (4.94)$$

Поскольку число проб при включении элемента равно числу проб при его поиске, результат (4.94) можно использовать для вычисления среднего числа E проб, требующихся при обращении к произвольному ключу в таблице. Вновь обозначим через n размер таблицы, а через m — число ключей, находящихся в таблице. Тогда

$$E=\frac{1}{m}\sum_{k=1}^{m}E_k=\frac{n+1}{m}\sum_{k=1}^{m}\frac{1}{n-k+2}=\frac{n+1}{m}\,(H_{n+1}-H_{n-m+1}), \quad (4.95)$$

где $H_n=1+\frac{1}{2}+\ldots+\frac{1}{n}$ — гармоническая функция. Функцию H_n можно аппроксимировать следующим образом: $H_n\cong\ln(n)+\gamma$, где γ — эйлерова константа. Если, кроме того, в (4.95) $m/(n+1)$ заменить на α, то мы получим

$$E=\frac{1}{\alpha}(\ln(n+1)-\ln(n-m+1))=$$
$$=\frac{1}{\alpha}\ln\frac{n+1}{n+1-m}=\frac{-1}{\alpha}\ln(1-\alpha), \quad (4.96)$$

где α приблизительно равно отношению занятой и имеющейся памяти, называемому *коэффициентом заполнения*; $\alpha=0$ предполагает пустую таблицу, $\alpha=n/(n+1)$ — заполненную таблицу. Среднее значение E числа проб при поиске или включении случайно выбранного ключа как функции от коэффициента заполнения α приведено в табл. 4.6.

Таблица 4.6. Среднее значение числа проб как функция от коэффициента загрузки

α	E
0,1	1,05
0,25	1,15
0,5	1,39
0,75	1,85
0,9	2,56
0,95	3,15
0,99	4,66

Полученные числа действительно вызывают удивление; они показывают чрезвычайно высокую эффективность метода преобразования ключа. Даже если таблица заполнена на 90 %, понадобится в среднем только 2,56 пробы, чтобы либо обнаружить местоположение ключа, либо найти свободное место! Особо отметим, что эта цифра не зависит от абсолютного числа имеющихся ключей, а зависит лишь от коэффициента заполнения.

Проведенный выше анализ основан на использовании метода разрешения конфликтов, который равномерно рассеивает ключи на оставшемся пространстве. Методы, применяемые на практике, несколько менее эффективны. Подробный анализ *линейного опробирования* дает следующее (4.97) среднее значение числа проб [4.10]:

$$E = \frac{1 - \alpha/2}{1 - \alpha}. \tag{4.97}$$

Некоторые значения $E(\alpha)$ перечислены в табл. 4.7. Результаты, полученные даже при худшем методе разрешения конфликтов, так высоки, что возникает соблазн рассматривать преобразования ключей (расстановку) как всеобщую панацею. Ведь эффективность этого метода даже выше, чем у самых утонченных методов организации деревьев, которые здесь рассматривались, во всяком случае, если сравнивать количество шагов, необходимых для поиска и включения. Поэтому важно отчетливо представлять себе некоторые недостатки метода расстановки, которые очевидны при беспристрастном изучении.

Таблица 4.7. Среднее значение числа проб при линейном опробировании

α	E
0,1	1,06
0,25	1,17
0,5	1,50
0,75	2,50
0,9	5,50
0,95	10,50

Разумеется, основной недостаток по сравнению с методами, использующими динамическое размещение, состоит в том, что *размер таблицы фиксирован* и не может приспосабливаться к действительным потребностям. Поэтому необходимо достаточно хорошо оценить *a priori* количество классифицируемых элементов данных, если мы хотим избежать неэкономного использования памяти, а также низкой эффективности (или даже переполнения таблицы). Даже если число элементов

точно известно, что бывает крайне редко, для получения хорошей эффективности нужно, чтобы размер таблицы был несколько больше (скажем, на 10 %).

Второй главный недостаток метода рассеянной памяти становится очевидным, если ключи надо не только вставлять и разыскивать, но и удалять, так как удаление из таблицы очень затруднено, если не используется прямое связывание в цепочку элементов из области переполнения. Итак, справедливость требует отметить, что древовидные структуры не теряют своей привлекательности и в самом деле предпочтительны, если объем данных сильно варьируется и временами даже уменьшается.

УПРАЖНЕНИЯ

4.1. Введем понятие *рекурсивного типа*

$$\textbf{rectype}\ T = T_0$$

как объединения множества значений, определенных типом T_0 с единственным значением **none** «ничто», т. е.

$$T = T_0 \cup \{\textbf{none}\}.$$

Определение типа *ped* [см (4.3)] можно, например, упростить до

```
rectype ped = record name: alfa;
                     father, mother: ped
              end
```

Как располагается в памяти рекурсивная структура, соответствующая рис. 4.2?

Вероятно, реализация такой структуры будет основана на схеме динамического распределения памяти, и поля, называемые *father* и *mother* в приведенном выше примере, будут содержать ссылки, получаемые автоматически, но скрытые от программиста. Какие трудности встретятся при реализации такой структуры?

4.2. Определите структуры данных, описанные в последнем параграфе разд. 4.2, в терминах записей и ссылок. Можно ли, кроме того, представить такую родственную совокупность с помощью рекурсивных типов, предложенных в предыдущем упражнении?

4.3. Предположим, что очередь «первым вошел — первым вышел» Q с элементами типа T_0 реализована в виде связанного списка. Определите соответствующую структуру данных, процедуры включения и удаления элемента из Q и функцию, проверяющую, пуста или нет очередь. В процедурах должны иметься собственные средства для экономного переиспользования памяти.

4.4. Предположим, что записи в связанном списке содержат ключевое поле типа *integer*. Напишите программу сортировки списка в порядке возрастания значений ключей. Затем сформулируйте процедуру, формирующую список, в котором элементы расположены в обратном порядке.

4.5. Циклические списки (см. рис. 4.54) обычно формируются с так называемым *заголовком списка*. Какой смысл имеет использование такого заголовка? Напишите процедуры включения, удаления и поиска элемента с заданным ключом. Сделайте это, как предполагая существование заголовка, так и без него.

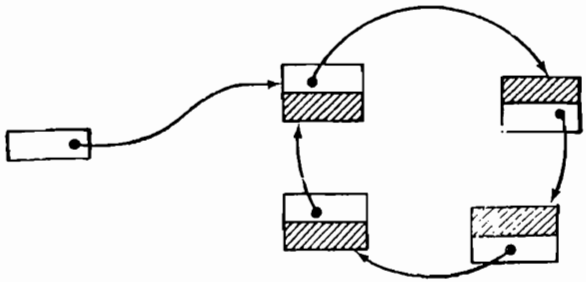


Рис. 4.54. Круговой список.

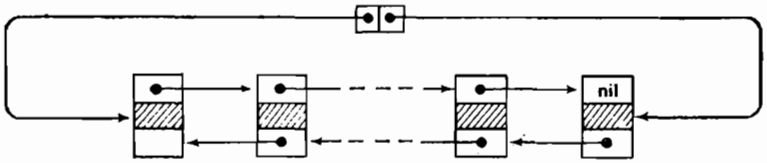


Рис. 4.55. Двунаправленный список.

4.6. *Двунаправленный список* — это список элементов, которые связаны с обеих сторон (см. рис. 4.55). Обе связи исходят из заголовка. Аналогично предыдущему упражнению напишите пакет процедур для поиска, включения и удаления элементов.

4.7. Будет ли программа 4.2 правильно работать, если некоторая пара $\langle x, y \rangle$ встретится во входном файле более одного раза?

4.8. Сообщение «'THIS SET IS NOT PARTIALLY ORDERED'» («ДАННОЕ МНОЖЕСТВО НЕ ЯВЛЯЕТСЯ ЧАСТИЧНО УПОРЯДОЧЕННЫМ») в программе 4.2 во многих случаях малоинформативно. Дополните программу, чтобы она выдавала последовательность элементов, которые образуют цикл, если он имеется.

4.9. Напишите программу, которая читает текст программ, находит все определения и вызовы процедур подпрограмм и пытается установить топологическое упорядочение на подпрограммах. Пусть $P \prec Q$ выполняется ,если P вызывается в Q.

4.10. Нарисуйте дерево, которое построит программа 4.3, если входной файл состоит из $n + 1$ чисел n, 1, 2, 3, . . . , n.

4.11. В каком порядке встречаются узлы при обходе дерева на рис. 4.23 сверху вниз, слева направо и снизу вверх?

4.12. Найдите правило построения последовательности из n чисел, для которой программа 4.4 сформирует идеально сбалансированное дерево.

4.13. Рассмотрим два порядка обхода бинарных деревьев:

(a)
(1) Обойти правое поддерево.
(2) Посетить корень.
(3) Обойти левое поддерево.

(b)
(1) Посетить корень.
(2) Обойти правое поддерево.
(3) Обойти левое поддерево.

Имеются ли какие-либо простые соотношения между последовательностями узлов, получаемыми при этих порядках обхода и теми, которые дают три порядка, определенные выше в тексте?

4.14. Определите структуру данных для представления n-арных деревьев. Затем напишите процедуру, которая обходит n-арное дерево и формирует бинарное дерево, содержащее те же элементы. Предположим, что ключ, расположенный в элементе, занимает k слов и каждая ссылка занимает одно слово памяти. Какова будет экономия памяти при использовании бинарного дерева по сравнению с n-арным?

4.15. Предположим, что дерево построено на основе следующего описания рекурсивной структуры данных (см. упр. 4.1):

```
rectype tree = record x: integer;
                      left, right: tree
               end
```

Сформулируйте процедуру, которая находит элемент с заданным ключом x и выполняет операцию P с этим элементом.

4.16. В файловой системе каталог файлов организован в виде упорядоченного бинарного дерева. Каждый узел обозначает файл и содержит имя файла, а также среди прочего дату последнего обращения к нему, закодированную в виде целого числа.

Напишите программу, которая обходит дерево и удаляет все файлы, последнее обращение к которым происходило до определенной даты.

4.17. В некоторой древовидной структуре частота обращения к каждому элементу измеряется эмпирически — приписыванием каждому узлу счетчика обращений. Через определенный интервал времени организация дерева изменяется при помощи обхода всего дерева и формирования нового дерева с использованием программы 4.4, которая вставляет элементы в порядке убывания счетчиков частоты обращений. Напишите программу, которая выполняет эту реорганизацию. Будет ли средняя длина пути в этом дереве равна, хуже или намного хуже, чем в оптимальном дереве?

4.18. Метод анализа алгоритма включения в дерево, описанный в разд. 4.5, можно также использовать для вычисления средних значений для числа сравнений C_n и числа пересылок (обменов) M_n, которые выполняются с помощью алгоритма быстрой сортировки (программа 2.10) при обработке n элементов массива, считая, что все $n!$ перестановок n ключей $\{1, 2, \ldots, n\}$ равновероятны. Найдите аналогию и определите C_n и M_n.

4.19. Нарисуйте сбалансированное дерево с 12 узлами, имеющее максимальную высоту среди всех сбалансированных деревьев с 12 узлами. В какой последовательности нужно включать узлы, чтобы процедура (4.63) сформировала это дерево?

4.20. Найдите такую последовательность из n включаемых элементов, чтобы процедура (4.63) выполняла каждое из четырех действий балансировки (*LL*, *RR*, *RL*, *LR*) по крайней мере один раз. Какова минимальная длина n такой последовательности?

4.21. Найдите сбалансированное дерево с ключами $1 \ldots n$ и перестановку этих ключей, такие, чтобы при работе процедуры удаления (4.64) она выполняла каждую из четырех подпрограмм балансировки по крайней мере один раз. Какова последовательность с минимальной длиной n?

4.22. Какова средняя длина пути в дереве Фибоначчи T_n?

4.23. Напишите программу, которая формирует дерево, близкое к оптимальному в соответствии с алгоритмом, основанным на выборе центроида в качестве корня (4.78).

4.24. Предположим, что ключи 1, 2, 3 ... вставляются в пустое Б-дерево порядка 2 (программа 4.7). Какие ключи вызывают расщепление страниц? Какие ключи вызывают увеличение высоты дерева?

Если ключи удаляются в том же порядке, то какие ключи вызывают слияние (и освобождение) страниц и какие ключи вызывают уменьшение высоты? Ответьте на этот вопрос для случаев (a) схемы удаления, использующей балансировку (как в программе 4.7), и (b) схемы без балансировки (при недостаче берется один элемент с соседней страницы).

4.25. Напишите программу поиска, включения и удаления ключей в бинарном Б-дереве. Используйте определение типа узла (4.84). Схема включения показана на рис. 4.51.

4.26. Найдите последовательность вставляемых ключей, которая, начиная с пустого симметричного бинарного Б-дерева, заставляет процедуру (4.87) выполнить все четыре действия балансировки (*LL*, *RR*, *LR*, *RL*) по крайней мере один раз. Какова самая короткая последовательность?

4.27. Напишите процедуру удаления элементов в симметричном бинарном Б-дереве. Затем найдите дерево и короткую последовательность удалений, вызывающую появление всех четырех ситуаций балансировки хотя бы по одному разу.

4.28. Сравните работу алгоритма включения и удаления для бинарных деревьев, АВЛ-сбалансированных деревьев и для симметричных бинарных Б-деревьев на вычислительной машине. В частности, исследуйте влияние упаковки данных, т. е. экономного представления данных с использованием только двух разрядов для хранения информации о сбалансированности каждого узла.

4.29. Модифицируйте алгоритм печати в программе 4.6 таким образом, чтобы его можно было использовать для изображения симметричных бинарных Б-деревьев с горизонтальными и вертикальными ветвями.

4.30. Если количество информации, связанной с каждым ключом, относительно велико (по сравнению с самим ключом), эту информацию не рекомендуется помещать в рассеянную таблицу. Объясните почему и предложите схему для представления такого множества данных.

4.31. Рассмотрите предложения о решении проблемы скопления с помощью деревьев переполнения вместо списков переполнения, т. е. организации тех ключей, которые вступают в конфликт, в виде дерева. Следовательно, каждый вход в рассеянную таблицу можно рассматривать как корень (возможно, пустого) дерева (древовидная расстановка).

4.32. Предложите схему выполнения включений и *удалений* в рассеянной таблице с использованием квадратичных приращений для разрешения конфликтов. Сравните экспериментально эту схему с простой организацией бинарного дерева, задавая случайные последовательности ключей для включения и удаления.

4.33. Основной недостаток метода расстановки состоит в том, что размер таблицы должен быть фиксированным, тогда как число элементов неизвестно. Предположим, что ваша вычислительная система содержит механизм динамического распределения памяти, который позволяет в любой момент выделять память. Следовательно, когда рассеянная

таблица H заполнена (или почти заполнена), то формируется бо́льшая таблица H', и все ключи из H пересылаются в H', после чего память, занятая H, возвращается в распоряжение системы. Этот процесс можно назвать *повторной расстановкой*. Напишите программу, которая выполняет повторную расстановку для таблицы H размером n.

4.34. Очень часто ключи являются не целыми числами, а последовательностями букв. Эти слова могут сильно различаться по длине и поэтому не могут удобно и экономно размещаться в полях фиксированного размера. Напишите программу, которая работает с рассеянной таблицей и *ключами переменной длины*.

ЛИТЕРАТУРА

4.1. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации. — *Доклады АН СССР*, **146**, 1962, с. 263—266.

4.2. BAYER R., McCREIGHT E. Organization and Maintenance of Large Ordered Indexes. — *Acta Informatica*, **1**, No. 3, 1972, 173—189.

4.3. BAYER R. Binary B-trees for Virtual Memory. — *Proc. 1971 ACM SIGFIDET Workshop*, San Diego, Nov. 1971, 219—235.

4.4. BAYER R. Symmetric Binary B-trees; Data Structure and Maintenance Algorithms. — *Acta Informatica*, **1**, No. 4, 1972, 290—306.

4.5. HU T. C., TUCKER A. C. — *SIAM J. Applied Math*, **21**, No. 4, 1971, 514—532.

4.6. KNUTH D. E. Optimum Binary Search Trees. — *Acta Informatica*, **1**, No. 1, 1971, 14—25.

4.7. MAURER W. D. An Improved Hash Code for Scatter Storage. — *Comm. ACM*, **11**, 1968, No. 1, 35—38.

4.8. MORRIS R. Scatter Storage Techniques. — *Comm. ACM*, **11**, No. 1, 1968, 38—43.

4.9. PETERSON W. W. Addressing for Random-access Storage. — *IBM J. Res. and Dev.*, **1**, 1957, 130—146.

4.10. SCHAY G., SPRUTH W. Analysis of a File Addressing Method. — *Comm. ACM*, **5**, No. 8, 1962, 459—462.

4.11. WALKER W. A., GOTLIEB C. C. A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees. — *Graph Theory and Computing*, New York: Academic Press, 1972, pp. 303—323.

5

СТРУКТУРА ЯЗЫКОВ И ТРАНСЛЯТОРЫ

В этой главе мы постараемся разработать транслятор для простого, «рудиментарного» языка программирования. Такая разработка может послужить примером систематического, хорошо структурированного подхода при написании программы нетривиальной сложности и размера. При этом можно продемонстрировать практическое применение методов, рассмотренных в предыдущих главах. Кроме того, мы постараемся дать общее представление о структуре трансляторов и принципах их работы. Знание этого предмета позволит лучше разобраться в искусстве программирования на языках высокого уровня, а также облегчит программисту разработку собственных систем, предназначенных для конкретных целей и областей применения. Но, поскольку, как известно, теория трансляторов — сложный и обширный предмет, в этом отношении данная глава будет носить лишь вводный и обзорный характер. По-видимому, главное, что следует уяснить, — это что структура транслятора отражает структуру языка и сложность — или простота — языка решающим образом влияет на сложность его транслятора. Поэтому мы начнем с описания строения языка, а затем сосредоточим внимание исключительно на простых структурах, для которых можно построить простые, модульные трансляторы. Такие простые языковые конструкции оказываются достаточными для удовлетворения практически всех потребностей, возникающих при использовании языков программирования.

5.1. ОПРЕДЕЛЕНИЕ И СТРУКТУРА ЯЗЫКА

В основе каждого языка лежит *словарь*. Его элементы обычно называют словами, но в теории формальных языков их называют *символами*. Языки характеризуются тем, что некоторые последовательности слов считаются правильными *предложениями* языка, а другие — неправильными, или не принадлежащими данному языку. Чем же определяется, является ли некоторая последовательность слов правильным предложением? Обычно это определяется грамматикой, синтаксисом (можно сказать — структурой) языка. *Синтаксис*

определяется как множество правил или формул, которые задают множество (формально правильных) предложений. Такое множество синтаксических правил не только позволяет установить, принадлежит ли некоторая заданная последовательность слов множеству предложений языка, но при этом определяет структуру предложения, которая устанавливает его смысл. Поэтому ясно, что синтаксис и *семантика* (=смысл) тесно связаны между собой. Следовательно, определения, связанные со структурой, всегда следует рассматривать как средство распознавания смысла. Но это не помешает нам изучить вначале исключительно структурные аспекты языка, отвлекаясь от их связи со смыслом, т. е. от их интерпретации.

Рассмотрим, например, предложение «Кошки спят». Слово «кошки» — подлежащее, а «спят» — сказуемое. Это предложение принадлежит языку, который можно описать, например, при помощи следующих синтаксических правил:

⟨предложение⟩ ::= ⟨подлежащее⟩ ⟨сказуемое⟩
⟨подлежащее⟩ ::= *кошки* | *собаки*
⟨сказуемое⟩ ::= *спят* | *едят*

Смысл этих трех строчек таков:

1. Предложение состоит из подлежащего, за которым следует сказуемое.
2. Подлежащее состоит либо из одного слова «кошки», либо из одного слова «собаки».
3. Сказуемое состоит либо из слова «спят», либо из слова «едят».

Идея заключается в том, что любое предложение можно получить из *начального символа* ⟨предложение⟩ последовательным применением *правил подстановки*.

Формализм, или нотация, использованный при написании этих правил, называется *бэкус-науровой формой* (БНФ). Впервые она была использована для описания Алгола-60 [5.7]. Синтаксические единицы ⟨предложение⟩, ⟨подлежащее⟩ и ⟨сказуемое⟩ называются *нетерминальными символами*, слова *кошки*, *собаки*, *спят* и *едят* — *терминальными символами*, а правила — *порождающими правилами*. Символы ::= и | — это *метасимволы**) языка БНФ. Если для краткости мы будем использовать отдельные заглавные буквы для нетерминальных символов, то данный пример можно переписать следующим образом:

*) Метасимволы не принадлежат описываемому языку, а относятся к языку описания. Метасимволами являются также ⟨ ⟩ — скобки нетерминальных символов. — *Прим. перев.*

Пример 1:

$$\begin{aligned} S &::= AB \\ A &::= x \mid y \\ B &::= z \mid w \end{aligned} \tag{5.1}$$

и язык, определенный этим синтаксисом, будет состоять из четырех предложений xz, yz, xw, yw.

Приведем теперь более точные, математические определения:

1. Пусть язык $L = L(T, N, P, S)$ задан:
   (a) словарем T терминальных символов;
   (b) множеством N нетерминальных символов (грамматических категорий);
   (c) множеством P порождающих правил (синтаксисом);
   (d) символом S (из N), называемым начальным символом.
2. Язык $L(T, N, P, S)$ есть множество последовательностей терминальных символов ξ, которые могут порождаться из S по правилу 3 (приведенному ниже):

$$L = \{\xi \mid S \xrightarrow{*} \xi \quad \text{и} \quad \xi \in T^*\} \tag{5.2}$$

   (для обозначения последовательностей символов мы используем греческие буквы), T^* означает множество всех последовательностей символов из T.
3. Последовательность σ_n может *порождаться* последовательностью σ_0 в том и только в том случае, если имеются последовательности $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}$, такие, что каждое σ_i может непосредственно порождаться σ_{i-1} по правилу 4 (приведенному ниже):

$$(\sigma_0 \xrightarrow{*} \sigma_n) \leftrightarrow (\sigma_{i-1} \rightarrow \sigma_i) \quad \text{для} \quad i = 1 \ldots n. \tag{5.3}$$

4. Последовательность η может *непосредственно порождаться* последовательностью ξ в том и только в том случае, если существуют последовательности α, β, ξ', η', такие, что:
   (a) $\xi = \alpha\xi'\beta$;
   (b) $\eta = \alpha\eta'\beta$;
   (c) P содержит порождающее правило $\xi' ::= \eta'$.

Примечание. Правило вида $\alpha ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ используется как сокращенная запись для множества порождающих правил:

$$\alpha ::= \beta_1, \quad \alpha ::= \beta_2, \ \ldots, \ \alpha ::= \beta_n.$$

Например, последовательность xz из примера 1 можно получить с помощью следующих шагов непосредственного порождения: $S \rightarrow AB \rightarrow xB \rightarrow xz$; следовательно, $S \xrightarrow{*} xz$, и по-

скольку $xz \in T^*$, то xz есть предложение языка, т. е. $xz \in L$. Отметим, что нетерминальные символы A и B появляются только на промежуточных шагах, а окончательный шаг должен дать последовательность, состоящую только из *терминальных* символов. Грамматические правила называются порождающими, потому что они определяют, как новые последовательности могут формироваться, или *порождаться*.

Язык называется *контекстно-свободным* в том и только в том случае, если он может быть определен с помощью контекстно-свободного множества порождающих правил. Множество порождающих правил контекстно-свободно тогда и только тогда, когда все его члены имеют вид

$$A ::= \xi \quad (A \in N,\ \xi \in (N \cup T)^*),$$

т. е. если его левая часть состоит из одного нетерминального символа, который может заменяться на последовательность, стоящую в правой части, независимо от контекста, в котором он встречается. Если порождающее правило имеет вид

$$\alpha A \beta ::= \alpha \xi \beta,$$

то оно называется *контекстно-зависимым*, так как замена A на ξ может иметь место только в контекстах α и β. Далее мы ограничим рассмотрение только контекстно-свободными языками.

Из примера 2 видно, как при помощи рекурсии конечное множество порождающих правил может задавать бесконечное множество предложений.

Пример 2:

$$\begin{aligned} S &::= xA, \\ A &::= z \mid yA. \end{aligned} \tag{5.4}$$

Начальный символ S может порождать следующие предложения:

xz
xyz
$xyyz$
$xyyyz$
.....

5.2. АНАЛИЗ ПРЕДЛОЖЕНИЙ

Задача трансляторов, или «языковых процессоров», — это в первую очередь не порождение, а *распознавание* предложений и их структуры. Это означает, что шаги порождения, которые формируют предложение, должны реконструироваться

при чтении предложения, т. е. проходиться в обратном порядке. В принципе это очень трудная, а иногда и невыполнимая задача. Ее сложность сильно зависит от правил порождения, которые определяют язык. Разработка алгоритмов распознавания для языков с достаточно сложной структурой — задача теории *синтаксического анализа*. Здесь же наша цель — разработать метод построения алгоритмов распознавания, достаточно простых и эффективных для практического применения. Это значит, что вычислительные затраты на анализ предложения должны находиться в линейной зависимости от длины предложения, в самом худшем случае функция зависимости может быть $n \cdot \log n$, где n — длина предложения. Разумеется, мы не можем ставить задачу поиска алгоритма распознавания для любого заданного языка, будем реалистами и поступим наоборот: построим некоторый эффективный алгоритм, а затем определим, с каким классом языков он может работать [5.3].

Из основного требования эффективности следует в первую очередь, что каждый очередной этап анализа должен выбираться лишь в зависимости от текущего состояния процесса и от одного следующего читаемого символа. Другое наиболее важное требование — чтобы ни к какому этапу не было повторного обращения. Эти два требования известны как технический термин *просмотр на один символ вперед без возврата*.

Основной метод, который мы здесь разберем, называется *нисходящим* грамматическим разбором, поскольку, применяя этот метод, мы будем пытаться реконструировать этапы порождения (которые в принципе образуют дерево) от начального символа к конечному предложению, т. е. сверху вниз [5.5, 5.6]. Вернемся к примеру 1: нам дано предложение «*Собаки едят*», и мы должны определить, принадлежит ли оно языку. По определению предложение принадлежит языку, только если оно может порождаться из начального символа ⟨предложение⟩. Из грамматических правил следует, что предложение должно состоять из подлежащего, за которым следует сказуемое. Теперь мы можем разделить задачу на две подзадачи: вначале нужно определить, может ли какая-либо начальная часть предложения порождаться из символа ⟨подлежащее⟩. Действительно, *собаки* может непосредственно порождаться из этого символа, поэтому мы убираем символ *собаки* из входного предложения (т. е. сдвигаемся на один шаг) и переходим к следующей подзадаче: определить, может ли оставшаяся часть предложения порождаться из символа ⟨сказуемое⟩. Поскольку ответ вновь положительный, результат анализа положительный. Процесс работы можно изобразить такой схемой, где слева показаны стоящие задачи,

а справа — еще не прочитанная часть входного предложения:

⟨предложение⟩		*собаки едят*
⟨подлежащее⟩	⟨сказуемое⟩	*собаки едят*
собаки	⟨сказуемое⟩	*собаки едят*
	⟨сказуемое⟩	*едят*
	едят	*едят*
	—	—

Вторая схема демонстрирует процесс анализа предложения *xyyz* в соответствии с порождающими правилами примера 2:

S	*xyyz*
xA	*xyyz*
A	*yyz*
yA	*yyz*
A	*yz*
yA	*yz*
A	*z*
z	*z*
—	—

Поскольку обратное прослеживание этапов порождения предложения называется *грамматическим разбором*, описанный выше алгоритм есть *алгоритм грамматического разбора*. В обоих примерах отдельные подстановки можно производить однозначно при проверке одного очередного символа во входном предложении. К сожалению, это не всегда бывает возможно, что видно из следующего примера:

Пример 3:

$$\begin{aligned} S &::= A \mid B \\ A &::= xA \mid y \\ B &::= xB \mid z \end{aligned} \qquad (5.5)$$

Мы пытаемся проанализировать предложение *xxxz*

S	*xxxz*
A	*xxxz*
xA	*xxxz*
A	*xxz*
xA	*xxz*
A	*xz*
xA	*xz*
A	*z*

и попадаем впросак. Трудность возникает на самом первом шаге, когда решение о замене *S* на *A* или *B* нельзя принять на основе лишь первого символа. Можно прослеживать один

из возможных выборов, а затем возвращаться, если этот путь не дает нужного решения. Такое действие называется *возвратом*. В языке примера 3 число возможных шагов, на которые приходится иногда возвращаться, не ограничено. Понятно, что подобная ситуация крайне нежелательна; следовательно, нужно избежать таких особенностей языка, которые приводят к возврату при грамматическом разборе. Поэтому мы принимаем решение, что будем рассматривать только такие грамматические системы, в которых начальные символы альтернативных правых частей порождающих правил различны.

Ограничение 1:

Если дано порождающее правило

$$A ::= \xi_1 | \xi_2 | \ldots | \xi_n,$$

то множества начальных символов всех предложений, которые могут порождаться из различных ξ_i, не должны пересекаться, т. е.

$$first(\xi_i) \cap first(\xi_j) = \varnothing \text{ для всех } i \neq j.$$

Множество $first(\xi)$ есть множество всех терминальных символов, которые могут встречаться в начале предложений, полученных из ξ. Пусть это множество вычисляется согласно следующим правилам:

1. Если первый символ аргумента терминальный, то

$$first(a\xi) = \{a\}.$$

2. Если первый символ нетерминальный и стоит в левой части порождающего правила

$$A ::= \alpha_1 | \alpha_2 | \ldots | \alpha_n,$$

то

$$first(A\xi) = first(\alpha_1) \cup first(\alpha_2) \cup \ldots \cup first(\alpha_n).$$

В примере 3 можно заметить, что $x \in first(A)$ и $x \in first(B)$. Следовательно, в первом порождающем правиле нарушено ограничение 1. На самом деле, легко найти синтаксис для языка примера 3, удовлетворяющий ограничению 1. Нужно отложить разделение на части до тех пор, пока не будут пройдены все x. Следующие порождающие правила *эквивалентны* правилам (5.5) в том смысле, что они порождают то же множество предложений:

$$\begin{aligned} S &::= C \mid xS, \\ C &::= y \mid z. \end{aligned} \tag{5.5a}$$

К сожалению, ограничение 1 недостаточно сильно, чтобы избавить нас от дальнейших неприятностей. Рассмотрим

Пример 4:

$$\begin{aligned} S &::= Ax, \\ A &::= x \mid \epsilon. \end{aligned} \tag{5.6}$$

Здесь ϵ обозначает нулевую последовательность символов. Если мы попытаемся разобрать предложение x, то можем попасть в следующий «тупик»:

S	x
Ax	x
xx	x
x	—

Трудность возникает из-за того, что мы должны были следовать правилу $A ::= \epsilon$ вместо $A ::= x$. Эта ситуация называется *проблемой пустой строки*, она связана со случаем, когда нетерминальный символ может порождать пустую последовательность. Чтобы ее избежать, мы вводим

Ограничение 2:

Для любого символа $A \in N$, который порождает пустую последовательность $(A \xrightarrow{*} \epsilon)$, множество начальных символов не должно пересекаться со множеством символов, которые могут появляться в предложениях языка справа от какой-либо последовательности, порождаемой A (внешними символами A), т. е.

$$first(A) \cap follow(A) = \varnothing.$$

Множество $follow(A)$ определяется так: берутся все порождающие правила P_i вида

$$X ::= \xi A \eta,$$

затем для каждой последовательности η_i, стоящей справа от A, определяется ее множество начальных символов $S_i = first(\eta_i)$. Множество $follow(A)$ — объединение всех таких множеств S_i. Если хотя бы одна η может порождать пустую последовательность, то множество $follow(X)$ следует также включить в $follow(A)$. В примере 4 ограничение 2 нарушается для символа A, поскольку

$$first(A) = follow(A) = \{x\}.$$

Повторение подобных последовательностей символов в предложениях обычно задается в порождающих правилах с помощью рекурсии. Например, порождающее правило

$$A ::= B \mid AB$$

описывает множество предложений B, BB, BBB, ... Однако использование такого правила теперь запрещено ограниче-

нием 1, так как

$$first(B) \cap first(AB) = first(B) \neq \varnothing.$$

Если мы заменим это правило его слегка модифицированной версией

$$A ::= \epsilon \mid AB,$$

порождающей последовательности ϵ, B, BB, BBB, ..., то нарушим ограничение 2, поскольку

$$first(A) = first(B)$$

и, следовательно,

$$first(A) \cap follow(A) \neq \varnothing.$$

Очевидно, что два эти ограничения запрещают использование определений с «левой рекурсией». Простой способ избежать таких форм — это либо использовать правую рекурсию

$$A ::= \epsilon \mid BA,$$

либо расширить БНФ-нотацию с тем, чтобы она допускала явное выражение повторений. Поэтому определим $\{B\}$ как множество последовательностей

$$\epsilon,\ B,\ BB,\ BBB,\ \ldots.$$

Конечно, нужно учитывать, что каждая такая конструкция может порождать пустую последовательность. (Фигурные скобки { } являются метасимволами расширенной БНФ.)

Эти рассуждения, а также пример преобразования порождающих правил (5.5) в (5.5а) могут навести на мысль, что «трюки» с преобразованием грамматик позволяют решить все проблемы синтаксического анализа. Но не следует забывать, что структура предложений связана с их смыслом и что смысл синтаксических конструкций обычно выражается через смысл их компонент. Рассмотрим, например, язык, состоящий из выражений, которые включают операнды a, b, c и знак минус, обозначающий вычитание:

$$S ::= A \mid S - A,$$
$$A ::= a \mid b \mid c.$$

Согласно этой грамматике, предложение $a - b - c$ имеет структуру, которую с использованием скобок можно выразить следующим образом: $((a - b) - c)$. Но если эту грамматику преобразовать в эквивалентную, но свободную от левой рекурсии

$$S ::= A \mid A - S,$$
$$A ::= a \mid b \mid c,$$

то это же предложение получит другую структуру, которую можно выразить как $(a-(b-c))$. Учитывая принятое значение вычитания, мы видим, что эти две формы вовсе не эквивалентны с точки зрения семантики.

Следует сделать вывод, что при определении языка, обладающего смыслом, нужно всегда принимать во внимание его семантическую структуру, поскольку синтаксис должен ее отражать.

5.3. ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО ГРАФА

В предыдущем разделе был описан алгоритм нисходящего распознавания, применимый к грамматикам, которые удовлетворяют ограничениям 1 и 2. Теперь мы перейдем к реализации этого алгоритма в виде конкретной программы. При этом можно использовать два различных метода. Один из них — это написать универсальную программу нисходящего грамматического разбора, пригодную для всех возможных грамматик (удовлетворяющих ограничениям 1 и 2). В этом случае конкретные грамматики задаются этой программе в виде данных некоторой структуры, которая в каком-то смысле управляет ее работой. Поэтому такая программа называется *таблично-управляемой*. Другой метод — разрабатывать программу нисходящего грамматического разбора специально для заданного конкретного языка; при этом его синтаксис по определенным правилам отображается в последовательность операторов, т. е. в программу. Мы по очереди рассмотрим оба этих метода, каждый из которых имеет свои преимущества и недостатки. При построении транслятора для конкретного языка программирования вряд ли потребуется высокая гибкость и параметризация, свойственные универсальной программе, тогда как программа грамматического разбора, предназначенная специально для данного языка, обычно оказывается более эффективной и с ней легче работать, поэтому такой подход предпочтителен. В обоих случаях полезно представлять заданный синтаксис в виде так называемого *синтаксического графа*, или *графа распознавания*. Такой граф отражает управление ходом работы при грамматическом анализе предложения.

Для нисходящего грамматического разбора характерно, что цель анализа известна с самого начала. Эта цель — распознать предложение, т. е. последовательность символов, которая может порождаться из начального символа. Применение порождающего правила, т. е. замена одного символа последовательностью символов, соответствует расщеплению одной цели на некоторое число подцелей, которые должны следовать в определенном порядке. Поэтому нисходящий метод можно называть также и *целеориентированным* грамматиче-

ским разбором. При построении программы грамматического разбора можно воспользоваться этим очевидным соответствием между нетерминальными символами и целями: для каждого нетерминального символа строится своя процедура грамматического разбора. Цель каждой такой процедуры — распознавание части предложения, которая может порождаться из соответствующего нетерминального символа. Поскольку мы хотим построить граф, представляющий всю программу грамматического разбора, то каждый нетерминальный символ будет отображаться в подграф. Поэтому мы приходим к таким правилам построения синтаксического графа:

Правила построения графа:

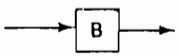
А1. Каждый нетерминальный символ A с соответствующим множеством порождающих правил

$$A ::= \xi_1 | \xi_2 | \ldots | \xi_n$$

отображается в синтаксический граф A, структура которого определяется правой частью порождающего правила в соответствии с А2 — А6.

А2. Каждое появление *терминального* символа x в ξ_i соответствует оператору распознавания этого символа во входном предложении. На графе это изображается ребром, помеченным символом x, заключенным в кружок.

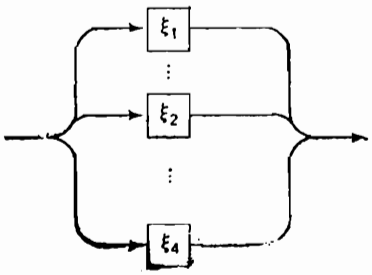
А3. Каждому появлению *нетерминального* символа B в ξ_i соответствует обращение к процедуре распознавания B. На графе это изображается ребром, помеченным символом B, заключенным в квадрат:



А4. Порождающее правило, имеющее вид

$$A ::= \xi_1 | \ldots | \xi_n$$

отображается в граф

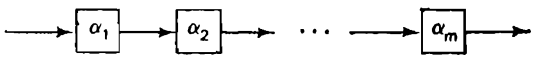


где каждое $\boxed{\xi_i}$ получено применением правил А2—А6 к ξ_i.

А5. Строка ξ, имеющая вид

$$\xi = \alpha_1\alpha_2 \ldots \alpha_m$$

отображается в граф

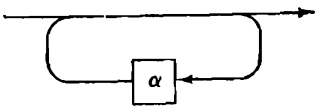


где каждое $\boxed{\alpha_i}$ получено применением правил А2—А6 к α_i.

А6. Строка ξ, имеющая вид

$$\xi = \{\alpha\}$$

отображается в граф



где $\boxed{\alpha}$ получено применением правил А2—А6 к α.

Пример 5:

$$\begin{aligned} A &::= x \mid (B), \\ B &::= AC, \\ C &::= \{+A\}. \end{aligned} \qquad (5.7)$$

Здесь «+», x, (, и) — терминальные символы, а { и } принадлежит расширенной БНФ и, следовательно, являются метасимволами. Язык, порождаемый из A, состоит из выражений с операндами x, знаком операции «+» и скобками. Примеры предложений:

$$\begin{array}{l} x \\ (x) \\ (x + x) \\ ((x)) \\ \ldots\ldots \end{array}$$

Графы, полученные с помощью применения шести правил построения графов, показаны на рис. 5.1. Заметим, что эту систему графов можно свести в один граф, подставив соответственно C в B и B в A (см. рис. 5.2).

Синтаксический граф является эквивалентным представлением грамматики языка; его можно использовать вместо

множества порождающих правил БНФ. Это очень удобная форма, и во многих (если не в большинстве) случаев она предпочтительнее БНФ. Разумеется, граф дает более ясное и точное представление о структуре языка, а также позволяет лучше представить себе процесс грамматического разбора.

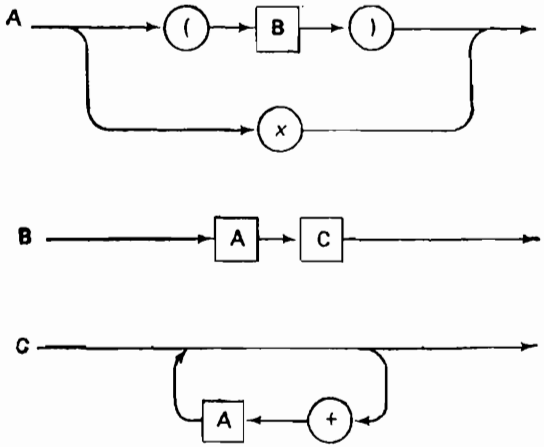


Рис. 5.1. Синтаксические графы для синтаксиса прим. 5.

Граф является подходящим представлением, которое может служить отправной точкой для разработчика языка. Примеры полных определений языков с помощью синтаксических графов даны в разд. 5.7 для ПЛ/0 и в приложении *В* для Паскаля.

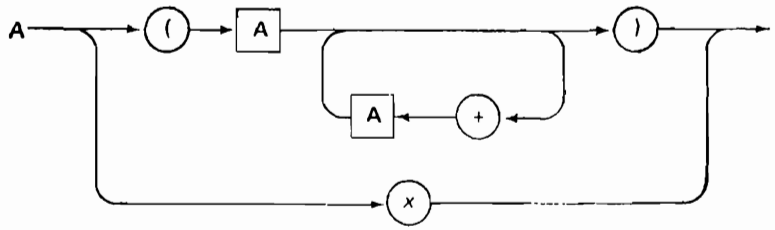


Рис. 5.2. Сводный синтаксический граф, соответствующий прим. 5.

Для того чтобы обеспечить детерминированный грамматический разбор с просмотром вперед на один символ, были установлены ограничения 1 и 2. Как проявляются эти ограничения при графическом представлении синтаксиса? Здесь особенно наглядно видны удобство и ясность такого представления.

1. Ограничению 1 соответствует требование, чтобы при каждом разветвлении можно было выбрать ветвь, по которой

будет идти дальнейший разбор по очередному символу на этой ветви. Это означает, что никакие две ветви не должны начинаться с одного и того же символа.

2. Ограничению 2 соответствует требование, чтобы если какой-либо граф A можно пройти, не читая вообще никаких входных символов, то такая «нулевая ветвь» должна помечаться всеми символами, которые могут следовать за A. (Это влияет на решение о переходе на эту ветвь.)

Легко проверить, удовлетворяет ли некоторая система графов этим двум ограничениям, не обращаясь к представлению грамматики с помощью БНФ. В качестве вспомогательного шага для каждого графа A определяются множества $first(A)$ и $follow(A)$. Затем непосредственно можно проверить выполнение ограничений 1 и 2. Систему графов, которая удовлетворяет этим двум ограничениям, мы будем называть *детерминированным синтаксическим графом*.

5.4. ПОСТРОЕНИЕ ПРОГРАММЫ ГРАММАТИЧЕСКОГО РАЗБОРА ДЛЯ ЗАДАННОГО СИНТАКСИСА

Программу, которая распознает какой-либо язык, легко построить на основе его детерминированного синтаксического графа (если такой граф существует). Этот граф фактически представляет собой блок-схему программы. Но при ее разработке рекомендуется строго следовать правилам преобразования, подобным тем, с помощью которых можно предварительно получить из БНФ графическое представление синтаксиса. Эти правила перечислены ниже. Они применяются в определенном контексте, который предполагает наличие основной программы, содержащей процедуры, которые соответствуют различным подцелям, а также процедуру перехода к очередному символу.

Для простоты мы будем считать, что предложение, которое нужно анализировать, представлено файлом *input* и что терминальные символы — отдельные значения типа *char*. Пусть символьная переменная *ch*: *char* всегда содержит очередной читаемый символ. Тогда переход к следующему символу выражается оператором

$$read(ch)$$

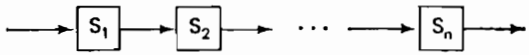
Основная программа будет состоять из оператора чтения первого символа, за которым следует оператор активации основной цели грамматического разбора. Отдельные процедуры, соответствующие целям грамматического разбора или графам, получаются по следующим правилам. Пусть оператор, полученный с помощью преобразования графа S, обозначается через $T(S)$.

Правила преобразования графа в программу:

В1. Свести систему графов к как можно меньшему числу отдельных графов с помощью соответствующих подстановок.

В2. Преобразовать каждый граф в описание процедуры в соответствии с приведенными ниже правилами В3—В7.

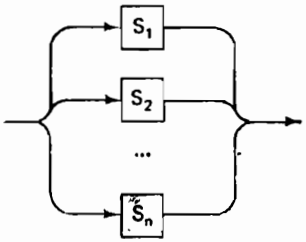
В3. *Последовательность* элементов



переводится в составной оператор

begin $T(S_1)$; $T(S_2)$; ...; $T(S_n)$ **end**

В4. *Выбор* элементов



переводится в выбирающий или условный оператор

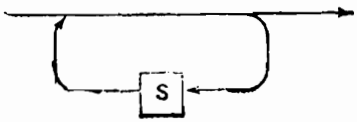
case *ch* **of**
L_1 : $T(S_1)$;
L_2 : $T(S_2)$;
.....
L_n : $T(S_n)$
end

if *ch* **in** L_1 **then** $T(S_1)$ **else**
if *ch* **in** L_2 **then** $T(S_2)$ **else**
.....
if *ch* **in** L_n **then** $T(S_n)$ **else**
error

где L_i означает множество начальных символов конструкции S_i ($L_i = first(S_i)$).

Примечание. Если L_i состоит из одного символа a, то, разумеется, вместо «*ch* **in** L_i» нужно писать «$ch = a$».

В5. Цикл вида



переводится в оператор

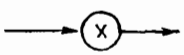
while *ch* **in** *L* **do** *T*(*S*)

где $T(S)$ есть отображение S в соответствии с правилами В3—В7, а L есть множество $L = first(S)$ (см. предыдущее примечание).

В6. Элемент графа, обозначающий другой граф A

переводится в оператор обращения к процедуре A.

В7. Элемент графа, обозначающий терминальный символ



переводится в оператор

if *ch* = *x* **then** *read*(*ch*) **else** *error*

где *error* — процедура, к которой обращаются при появлении неправильной конструкции.

Теперь покажем применение этих правил на примере преобразования редуцированного графа, изображенного на

```
program parse (input, output);
   var ch: char;
   procedure A;
   begin if ch = 'x' then read(ch) else
            if ch = '(' then
            begin read(ch); A;
               while ch = '+' do
                  begin read(ch); A
                  end ;
               if ch = ')' then read(ch) else error
            end else error
   end ;
begin read(ch); A
end
```

Программа 5.1. Грамматический разбор для синтаксиса из прим. 5.

рис. 5.2 (пример 5), в программу грамматического разбора (программа 5.1):

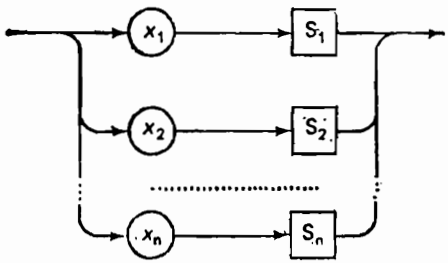
При этом преобразовании свободно применялись некоторые очевидные правила программирования, позволяющие упростить программу. Например, при буквальном переводе четвертая строка имела бы вид

```
if ch = 'x' then
    if ch = 'x' then read(ch) else error
else ....
```

Ясно, что ее можно сократить, как это сделано в программе. Операторы чтения в пятой и седьмой строках тоже получены с помощью такого же упрощения.

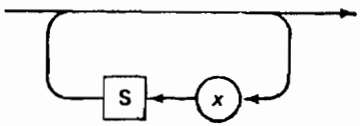
По-видимому, полезно определить, когда вообще возможны подобные упрощения, и показать это непосредственно в виде графов. Два основных случая покрываются следующими дополнительными правилами:

B4a




```
if ch = 'x1' then begin read(ch); T(S1) end else
if ch = 'x2' then begin read(ch); T(S2) end else
        ...........
if ch = 'xn' then begin read(ch); T(Sn) end else error
```

B5a




```
while ch = 'x' do
    begin read(ch); T(S) end
```

Кроме того, часто встречающуюся конструкцию

```
read(ch); T(S);
while B do
    begin read(ch); T(S) end
```

можно, разумеется, выразить короче:

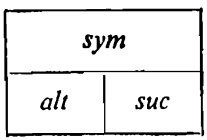
repeat *read*(*ch*); $T(S)$ **until** B (5.8)

Мы намеренно не описываем пока процедуру *error* («ошибка»). Поскольку сейчас нас интересует лишь, как определить, правильно ли входное предложение, мы можем считать, что эта процедура заканчивает работу программы. Конечно, на практике в случае появления неправильных конструкций нужно использовать более тонкие приемы. Они будут рассматриваться в разд. 5.9.

5.5. ПОСТРОЕНИЕ ТАБЛИЧНО-УПРАВЛЯЕМОЙ ПРОГРАММЫ ГРАММАТИЧЕСКОГО РАЗБОРА

Вместо того чтобы для каждого языка составлять специальную программу по правилам, изложенным в предыдущем разделе, можно построить одну, универсальную программу грамматического разбора. Конкретные грамматики задаются этой универсальной программе в виде исходных данных, предшествующих предложениям, которые нужно разобрать. Универсальная программа работает в строгом соответствии с методом простого нисходящего грамматического разбора; поэтому она довольно проста, если основана на детерминированном синтаксическом графе, т. е. если предложения можно анализировать с просмотром вперед на один символ без возврата.

Итак, грамматика, (мы предполагаем, что она представлена в виде детерминированного множества синтаксических графов) преобразуется в подходящую структуру данных, а не в структуру программ [5.2]. Естественный способ представить граф — это ввести узел для каждого символа и связать эти узлы с помощью ссылок. Следовательно, «таблица» — это не просто массив. Правила преобразования очевидны и приведены ниже. Узлы этой структуры представляют собой записи с вариантами, один для терминального, а другой — для нетерминального символа. Первый идентифицируется терминальным символом, который он обозначает, второй — ссылкой на структуру данных, представляющую соответствующий нетерминальный символ. Оба варианта содержат две ссылки: одна указывает на следующий символ, *последователь* (*suc*), а другая связана со списком возможных *альтернатив* (*alt*). Описание соответствующего типа данных приведено в (5.9), а графически узел можно изобразить как



Выясняется, что еще нужен элемент, представляющий пустую последовательность, символ «пусто». Мы обозначим его с помощью терминального элемента, называемого *empty* (5.9).

```
type pointer = ↑node;
   node =
   record suc,alt: pointer;
      case terminal: boolean of                    (5.9)
         true: (tsym: char);
         false: (nsym: hpointer)
   end
```

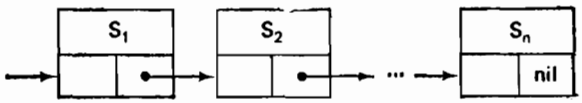
Правила преобразования графов в структуре данных аналогичны правилам B1—B7.

Правила преобразования графов в структурах данных

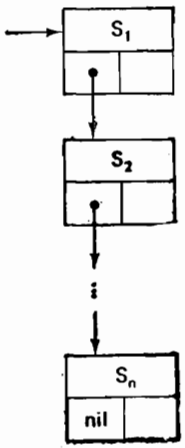
C1. Свести систему графов к как можно меньшему числу отдельных графов с помощью соответствующих подстановок.

C2. Преобразовать каждый граф в структуру данных согласно правилам C3—C5, приведенным ниже.

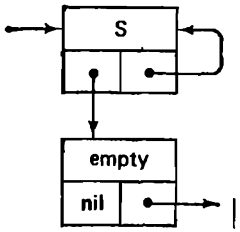
C3. Последовательность элементов (см. рисунок к правилу B3) преобразуется в следующий список узлов:



C4. Список альтернатив (см. рисунок к правилу B4) преобразуется в такую структуру данных:



C5. Цикл (см. рисунок к правилу B5) преобразуется в следующую структуру:



В качестве примера на рис. 5.3 показана структура, полученная из графа, соответствующего синтаксису примера 5 (рис. 5.2). Структура данных идентифицируется *узлом-заго-*

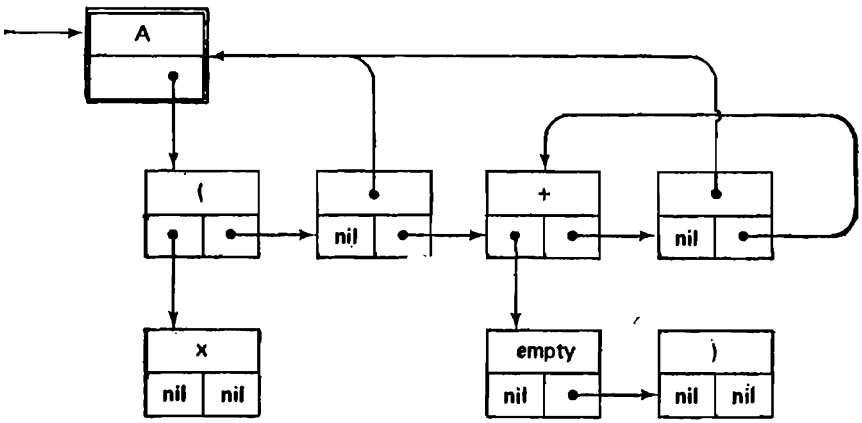


Рис. 5.3. Структура данных, представляющая граф рис. 5.2.

ловком, который содержит имя нетерминального символа (цели), к которому относится структура. Пока в заголовке необходимости нет, так как можно вместо поля цели указывать непосредственно на «вход» в соответствующую структуру. Однако заголовок можно использовать для хранения выводимого на печать имени структуры:

```
type hpointer = ↑header;
   header =
      record entry: pointer;
         sym: char
      end
```
(5.10)

Программа, производящая грамматический разбор предложения, представленного в виде последовательности символов входного файла, состоит из повторяющегося оператора,

описывающего переход от одного узла к следующему узлу. Она оформлена как процедура, задающая интерпретацию графа; если встречается узел, представляющий нетерминальный символ, то интерпретация графа, на который он ссылается предшествует завершению интерпретации текущего графа. Следовательно, процедура интерпретации вызывается *рекурсивно*. Если текущий символ (*sym*) входного файла совпадает с символом в текущем узле структуры данных, то процедура переходит к узлу, на который указывает поле *suc*, иначе — к узлу, на который указывает поле *alt*:

```
procedure parse(goal: hpointer; var match: boolean);
   var s : pointer;
begin s := goal↑.entry;
   repeat
      if s↑.terminal then
         begin if s↑.tsym = sym then
               begin match := true; getsym
               end                                          (5.11)
            else match := (s↑.tsym = empty)
         end
      else parse(s↑.nsym, match);
      if match then s := s↑.suc else s := s↑.alt
   until s = nil
end
```

Программа грамматического разбора (5.11) «стремится» к новой подцели G, как только она появляется, не проверяя даже, содержится ли текущий символ входного файла в множестве начальных символов соответствующего графа $first(G)$. Это предполагает, что в синтаксическом графе не должно существовать выбора между несколькими альтернативными нетерминальными элементами. В частности, если какой-либо нетерминальный символ может порождать пустую последовательность, то ни одна из правых частей соответствующих ему порождающих правил не должна начинаться с нетерминального символа.

На основе (5.11) можно построить более сложные таблично-управляемые программы грамматического разбора, которые могут работать с более широкими классами грамматик. Небольшая модификация позволяет также осуществлять и возвраты, но это будет сопровождаться значительной потерей эффективности.

Представление синтаксиса с помощью графа имеет один существенный недостаток: вычислительные машины не могут читать графы. Но перед началом грамматического разбора

нужно каким-то образом строить структуру данных, управляющих программой. В этом смысле представление грамматик в БНФ оказывается идеальным в качестве исходных данных для универсальной программы грамматического разбора. Поэтому следующий раздел посвящен разработке программы, которая читает правила БНФ и по правилам В1—В6 преобразует их во внутреннюю структуру данных, с которой может работать программа грамматического разбора (5.11) [5.8].

5.6. ПРЕОБРАЗОВАНИЕ БНФ В СТРУКТУРЫ ДАННЫХ, УПРАВЛЯЮЩИЕ ГРАММАТИЧЕСКИМ РАЗБОРОМ

Транслятор, распознающий порождающие правила БНФ и преобразующий их в какое-то другое представление, как раз служит примером программы, входные данные которой можно рассматривать как предложения, принадлежащие некоторому языку. Действительно, саму БНФ можно считать некоторым языком, имеющим свой собственный синтаксис, который, разумеется, также можно описать с помощью порождающих правил БНФ. Следовательно, его транслятор может служить примером распознавателя, который помимо этого преобразует входные данные, т. е., вообще говоря, является процессором. Поэтому мы будем действовать следующим образом:

Шаг 1. Определим синтаксис метаязыка, называемого РБНФ (расширенной БНФ).

Шаг 2. Построим распознающую программу для РБНФ в соответствии с правилами, приведенными в разд. 5.4.

Шаг 3. Расширив эту программу, превратим ее в транслятор и объединим с таблично-управляемой программой грамматического разбора.

Пусть метаязык, т. е. язык, на котором пишутся синтаксические правила, описан следующими порождающими правилами:

$$\begin{array}{ll} \langle\text{правило}\rangle & ::=\langle\text{символ}\rangle = \langle\text{выражение}\rangle \\ \langle\text{выражение}\rangle & ::=\langle\text{терм}\rangle\,\{,\langle\text{терм}\rangle\} \\ \langle\text{терм}\rangle & ::=\langle\text{фактор}\rangle\,\{\langle\text{фактор}\rangle\} \\ \langle\text{фактор}\rangle & ::=\langle\text{символ}\rangle \mid [\langle\text{терм}\rangle] \end{array} \qquad (5.12)$$

Заметим, что в порождающих правилах входного языка используются иные метасимволы, чем в БНФ. Это делается по двум причинам:

1. В (5.12) нужно отличать символы языка от метасимволов.

2. Желательно использовать более привычные символы печатающего устройства, в частности использовать один знак (=) вместо (::=).

Соответствие между обычной БНФ и нашей входной версией показано в табл. 5.1. Кроме того, каждое наше порождающее

Таблица 5.1. Метасимволы и символы языка

БНФ	РБНФ
::=	=
|	,
{	[
}	]

правило должно заканчиваться точкой. При использовании этого входного языка для описания синтаксиса примера 5 (5.7) мы получаем

$$\begin{aligned} &A = x,\ (B).\\ &B = AC.\\ &C = [+A]. \end{aligned} \qquad (5.13)$$

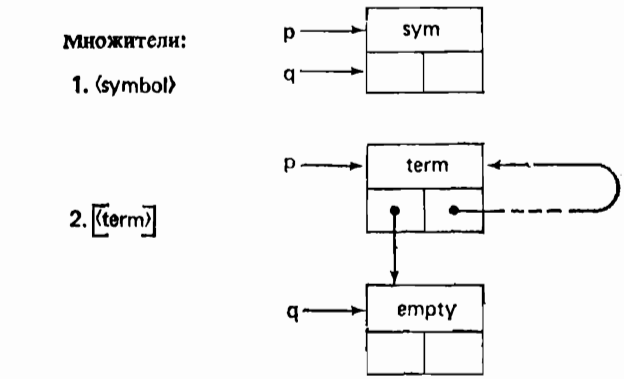
Чтобы упростить построение транслятора, мы будем считать, что терминальные символы — это *отдельные буквы* и каждое порождающее правило пишется в отдельной строке. Это позволяет использовать во входном тексте пробелы (чтобы его было удобнее читать), которые транслятор игнорирует. Но тогда оператор *read*(*ch*) в правиле B7 нужно заменить вызовом процедуры, которая определяет очередной учитываемый символ. Эта процедура — простейшая разновидность лексического сканера, или просто сканера. Задача сканера — выделить из входной последовательности обычных символов очередной *символ языка* *). Это не всегда бывает так легко, как в нашем примере, поскольку мы до сих пор предполагали, что символы языка — это отдельные буквы, а на самом деле — это особый и на практике редко встречающийся случай.

Наконец, мы постулируем, что в нашем входном языке БНФ нетерминальные символы представляются буквами $A - H$, а терминальные символы — буквами $I - Z$. Это чистая условность, не связанная ни с какими серьезными причинами, но она избавляет от необходимости задавать словари терминальных и нетерминальных символов перед списком порождающих правил.

*) Напомним, что символы языка суть множество терминальных и нетерминальных символов, «построенных» из обычно вводимых символов системы. — *Прим. ред.*

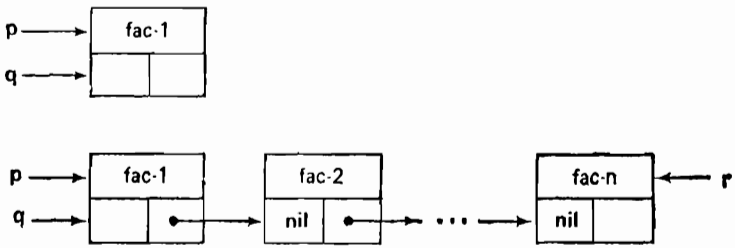
Убедившись, что (5.12) удовлетворяет ограничениям 1 и 2, и действуя строго по правилам В1 — В7, мы получаем программу 5.2, которая распознает язык, определенный в (5.12). Заметим, что сканер называется *getsym*.

На третьем шаге разработки транслятора нужно прочитанные порождающие правила БНФ преобразовать в структуру данных, которая может интерпретироваться процедурой грамматического разбора (5.11). К сожалению, этот этап не поддается формализации в отличие от этапа, связанного с построением программы распознавания. Поэтому мы просто нарисуем структуры, соответствующие каждой конструкции

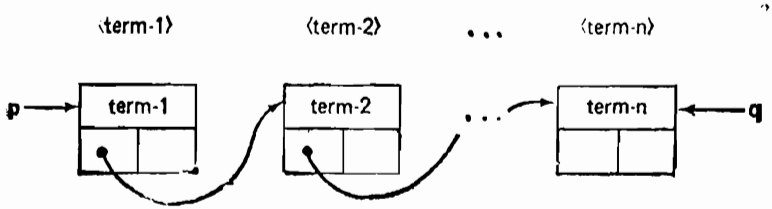


Слагаемые:

⟨factor-1⟩ ... ⟨factor-n⟩



Выражения:



```
program parser(input, output);
label 99;
const empty = '*';
var sym: char;

procedure getsym;
begin
   repeat read(sym); write(sym) until sym ≠ ' '
end {getsym} ;

procedure error;
begin writeln;
   writeln (' INCORRECT INPUT'); goto 99
end {error} ;

procedure term;
   procedure factor;
   begin
      if sym in ['A' . . 'Z', empty] then getsym else
      if sym = '[' then
      begin getsym; term;
         if sym = ']' then getsym else error
      end else error
   end {factor} ;
begin factor;
   while sym in ['A' . . 'Z', '[', empty] do factor
end {term} ;

procedure expression;
begin term;
   while sym = ',' do
   begin getsym; term
   end
end {expression} ;

begin {основная программа}
   while ¬eof(input) do
   begin getsym;
      if sym in ['A' . . 'Z'] then getsym else error;
      if sym = '=' then getsym else error;
      expression;
      if sym ≠ '.' then error;
      writeln; readln;
   end ;
99: end
```

Программа 5.2. Грамматический разбор для языка (5.12).

языка. Формируемые структуры выдаются как параметры-результаты соответствующих процедур распознавания языковых конструкций; таким образом, эти процедуры превращаются в процедуры трансляции. Естественно, в качестве результатов передаются не сами структуры, а ссылки на них *p*, *q*, *r* (см. рис. на с. 342).

Ясно, что процедура *factor* формирует новые элементы структуры данных; остальные же две процедуры связывают их в линейные списки, при этом *term* использует для связывания поле *suc*, а *expression* — поле *alt*. Подробности показаны в программе 5.3.

Метод обработки нетерминальных символов нуждается в некотором пояснении. Нетерминальный символ может встретиться в качестве фактора раньше, чем появится в левой части порождающего правила. Процедура *find*(*sym*, *h*) ищет заданный символ *sym* в линейном списке, где собраны все заголовки нетерминальных символов. Если символ найден, ссылка на него присваивается *h*, а если он еще не содержится в этом списке, то добавляется к нему. В процедуре *find* применяется метод барьера, подробно обсуждавшийся в гл. 4.

Программа 5.3 состоит из трех частей, каждая обрабатывает определенный раздел входного файла. Часть 1 читает *порождающие правила* и преобразует их в соответствующие структуры данных. Часть 2 читает и идентифицирует один символ — это *начальный символ*, с которого начинается порождение предложения языка. (Ему предшествует знак $, разграничивающий части 1 и 2 входных данных.) Часть 3 есть программа грамматического разбора (5.11), читающая *входные предложения* и анализирующая их в соответствии со структурами данных, сформированными в части 1.

Примечательно, что программа 5.3 получена просто с помощью включения добавочных операторов в *неизмененную* программу 5.2. Старая программа только распознавала правильно сформированные предложения, на ее основе можно построить новую, расширенную программу, которая не только распознает, но и транслирует предложения. Такой метод построения процессоров для работы с языком при помощи *поэтапного уточнения*, или, скорее, *поэтапного дополнения*, очень полезен. Он позволяет разработчику сосредоточить внимание исключительно на каком-то одном аспекте обработки языка, прежде чем обратиться к другим аспектам, и поэтому облегчает проверку правильности транслятора или, во всяком случае, обеспечивает высокий уровень надежности при разработке программы. В нашем довольно простом примере разработка транслятора состоит из двух этапов. Более сложные языки и более сложные задачи трасляции требуют значительно большего числа отдельных этапов дополнения. Очень

```
program generalparser (input, output);
label 99;
const empty = '•';
type pointer = ↑node;
     hpointer = ↑header;
   node = record suc, alt: pointer;
                 case terminal: boolean of
                   true: (tsym: char);
                   false: (nsym: hpointer)
          end ;
   header = record sym: char;
                   entry: pointer;
                   suc: hpointer
            end ;
var list, sentinel, h: hpointer;
   p: pointer;
   sym: char;
   ok: boolean;

procedure getsym;
begin
   repeat read(sym); write(sym) until sym ≠ ' '
end {getsym} ;
procedure find(s: char; var h: hpointer);
{поиск в списке нетерминального символа s, если его нет,
включение его}
   var h1: hpointer;
begin h1 := list; sentinel↑.sym := s;
   while h1↑.sym ≠ s do h1 := h1↑.suc;
   if h1 = sentinel then
      begin {включение} new (sentinel);
         h1↑.suc := sentinel; h1↑.entry := nil
      end ;
   h := h1
end {find} ;
procedure error;
begin writeln;
   writeln ('INCORRECT SYNTAX'); goto 99
end {error} ;
procedure term (var p,q,r: pointer);
   var a,b,c: pointer;
   procedure factor (var p,q: pointer);
```

```
        var u, v : pointer ; h : hpointer ;
    begin if sym in ['A' . . 'Z', empty] then
        begin {символ} new(a);
            if sym in ['A' . . 'H'] then
            begin {нетерминальный}  find(sym,h);
                a↑.terminal := false; a↑.nsym := h
            end else
            begin {терминальный}
                a↑.terminal := true; a↑.tsym := sym
            end ;
            p := a; q := a; getsym
        end else
        if sym = '[' then
        begin getsym; term(p,a,b); b↑.suc := p;
            new(b); b↑.terminal := true; b↑.tsym := empty;
            a↑.alt := b; q := b;
            if sym = ']' then getsym else error
        end else error
    end {factor} ;
begin factor(p,a); q := a;
    while sym in ['A' . . 'Z', '[', empty] do
    begin factor(a↑.suc, b); b↑.alt := nil; a := b
    end ;
    r := a
end {term} ;
procedure expression (var p,q: pointer);
    var a,b,c: pointer;
begin term(p,a,c): c↑.suc := nil;
    while sym = ',' do
    begin getsym;
        term(a↑.alt, b, c): c↑.suc := nil; a := b
    end ;
    q := a
end {expression} ;
procedure parse (goal: hpointer; var match: boolean);
    var s: pointer;
begin s := goal↑.entry;
    repeat
        if s↑.terminal then
        begin if s↑.tsym = sym then
                begin match := true; getsym
                end
            else match := (s↑.tsym = empty)
```

```
        end
        else parse(s↑.nsym, match);
        if match then s := s↑.suc else s := s↑.alt
    until s = nil
end {parse} ;
begin {порождающие правила}
    getsym; new(sentinel); list := sentinel;
    while sym ≠ '$' do
    begin find(sym,h);
        getsym; if sym = '=' then getsym else error;
        expression (h↑.entry, p); p↑.alt := nil;
        if sym ≠ '.' then error;
        writeln; readln; getsym
    end ;
    h := list; ok := true;  {проверка, все ли символы определены}
    while h ≠ sentinel do
    begin if h↑.entry = nil then
            begin writeln(' UNDEFINED SYMBOL ', h↑.sym);
                ok := false
            end ;
        h := h↑.suc
    end ;
    if ¬ok then goto 99;
{цель}


    getsym; find(sym,h); readln; writeln;
{предложения}
    while ¬eof(input) do
        begin write(' '); getsym; parse(h,ok);
            if ok ∧ (sym='.') then writeln (' CORRECT')
                              else writeln (' INCORRECT');
            readln
        end ;
99: end .
```

Программа 5.3. Транслятор для языка (5.13).

похожая разработка, состоящая из трех этапов, будет рассматриваться в разд. 5.8—5.11.

Как видно из разработки программы 5.3, программы, *управляемые синтаксическими таблицами*, или, вернее, управляемые структурой данных, обеспечивают свободу и гибкость, отсутствующие в специальных программах грамматического разбора. Хотя такая дополнительная гибкость в принципе не нужна, она оказывается весьма существенной в трансляторах для так называемых *расширяемых языков*. Расширяемые языки можно дополнять новыми синтаксическими конструкциями более или менее по усмотрению программиста. Так же как входной файл программы 5.3, входной файл для транслятора с расширяемого языка содержит раздел, определяющий расширения языка, используемые в последующей программе. Более сложные схемы позволяют даже изменять язык в процессе трансляции, чередуя части транслируемой программы с разделами новых определений языка.

Однако, хотя эти идеи могут показаться весьма привлекательными, попытки реализовать подобные трансляторы оказались довольно неудачными. Дело в том, что синтаксический анализ — лишь часть всей задачи трансляции и на самом деле даже не самая существенная часть. Ее легче всего формализовать и, следовательно, представить с помощью систематизированной табличной структуры. Гораздо труднее формализовать смысл языка, т. е. выход, или результат трансляции. До сих пор эта задача не была сколько-нибудь удовлетворительно решена, и этим объясняется то, почему разработчики трансляторов относятся к расширяемым языкам с гораздо большим энтузиазмом до их реализации, чем после. Остальную часть этой главы мы посвятим разработке скромного транслятора для конкретного, небольшого языка программирования.

5.7. ЯЗЫК ПРОГРАММИРОВАНИЯ ПЛ/0

Оставшиеся разделы этой главы посвящены разработке транслятора для языка, который мы назовем ПЛ/0. При создании этого языка учитывались два условия: во-первых, транслятор не должен оказаться слишком громоздким для этой книги, во-вторых, желательно было продемонстрировать большинство основных принципов трансляции языков программирования высокого уровня. Несомненно, можно было выбрать как более простой, так и более сложный язык; ПЛ/0 является одним из возможных компромиссов между языками достаточно простыми для ясности изложения и достаточно сложными, чтобы ими стоило заниматься. Значительно более

сложный язык — Паскаль, транслятор для которого был разработан с применением тех же методов. Его синтаксис дан в приложении В.

Если говорить о структуре программы, то ПЛ/0 достаточно полон. Конечно, в нем в качестве основной конструкции на уровне языка содержится оператор присваивания. Другие структурные концепции — это следование, условное выполнение и цикл, представленные знакомыми формами **begin/end**-, **if**-, **while**. В ПЛ/0 включено также и понятие подпрограммы, следовательно, там есть описания процедур и оператор вызова процедуры.

Что же касается типов данных, то ПЛ/0, бесспорно, удовлетворяет требованию простоты: единственный тип данных — это целые числа. Разумеется, в ПЛ/0 присутствуют обычные операции арифметики и сравнения.

Наличие процедур, т. е. более или менее самостоятельных частей программы, дает возможность ввести концепцию *локальности* объектов (констант, переменных и процедур). Поэтому в заголовке каждой процедуры есть описания объектов; эти объекты считаются локальными для процедуры, в которой они описаны.

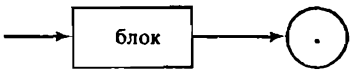
Это краткое введение позволяет представить себе синтаксис ПЛ/0. Этот синтаксис изображен на рис. 5.4 с помощью 7 диаграмм. Преобразовать диаграммы во множество эквивалентных порождающих правил БНФ мы предоставляем читателю. Рис. 5.4 является убедительным примером выразительности этих диаграмм, которые позволяют сформулировать синтаксическое описание целого языка программирования в столь краткой и хорошо воспринимаемой форме.

Следующая программа, написанная на ПЛ/0, демонстрирует некоторые свойства этого мини-языка. Эта программа содержит знакомые алгоритмы умножения, деления и нахождения наибольшего общего делителя двух натуральных чисел.

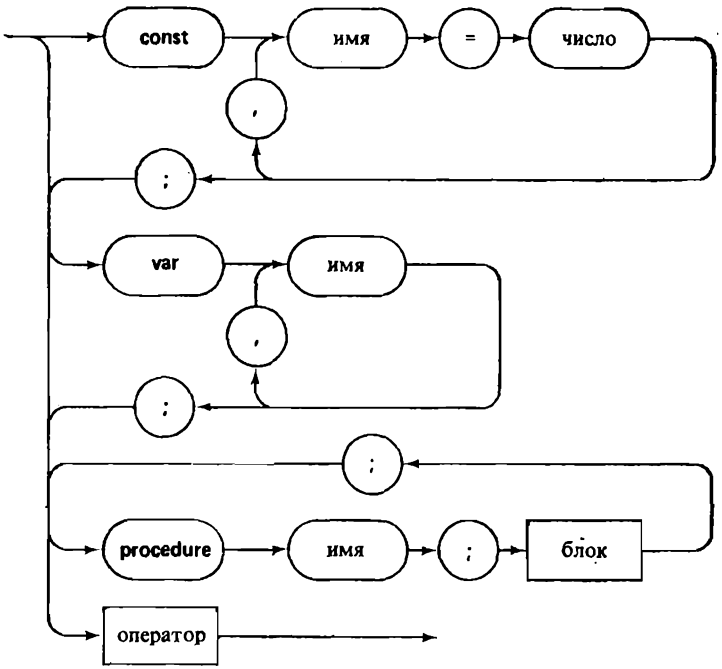
5.8. ПРОГРАММА ГРАММАТИЧЕСКОГО РАЗБОРА ДЛЯ ПЛ/0

В качестве первого этапа построения транслятора для ПЛ/0 нужно разработать программу грамматического разбора. Это можно сделать, строго следуя правилам построения В1 — В7, приведенным в разд. 5.4. Но этот метод применим, только если синтаксис удовлетворяет ограничениям 1 и 2. Поэтому мы обязаны проверить это условие в его формулировке для синтаксических графов.

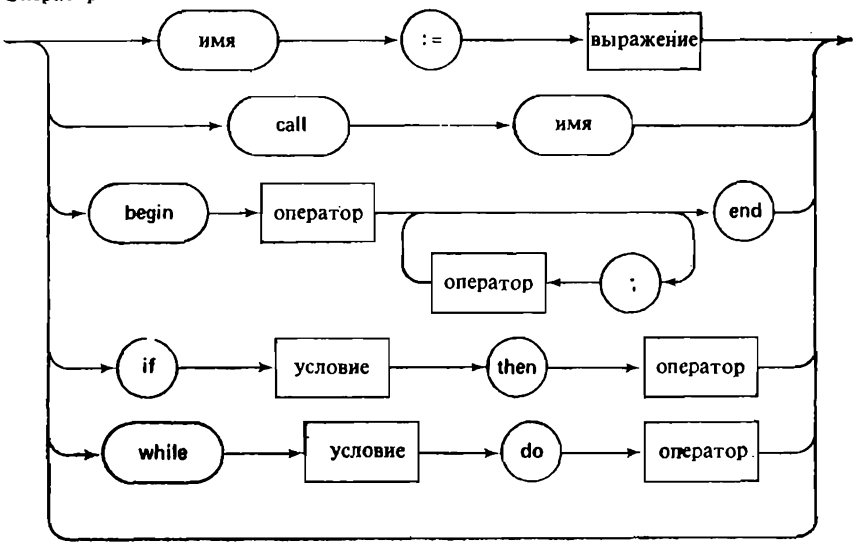
Ограничение 1 требует, чтобы каждая ветвь, выходящая из разветвления, вела к отличному от других начальному символу. Это очень просто проверить по синтаксическим диа-



Блок



Оператор



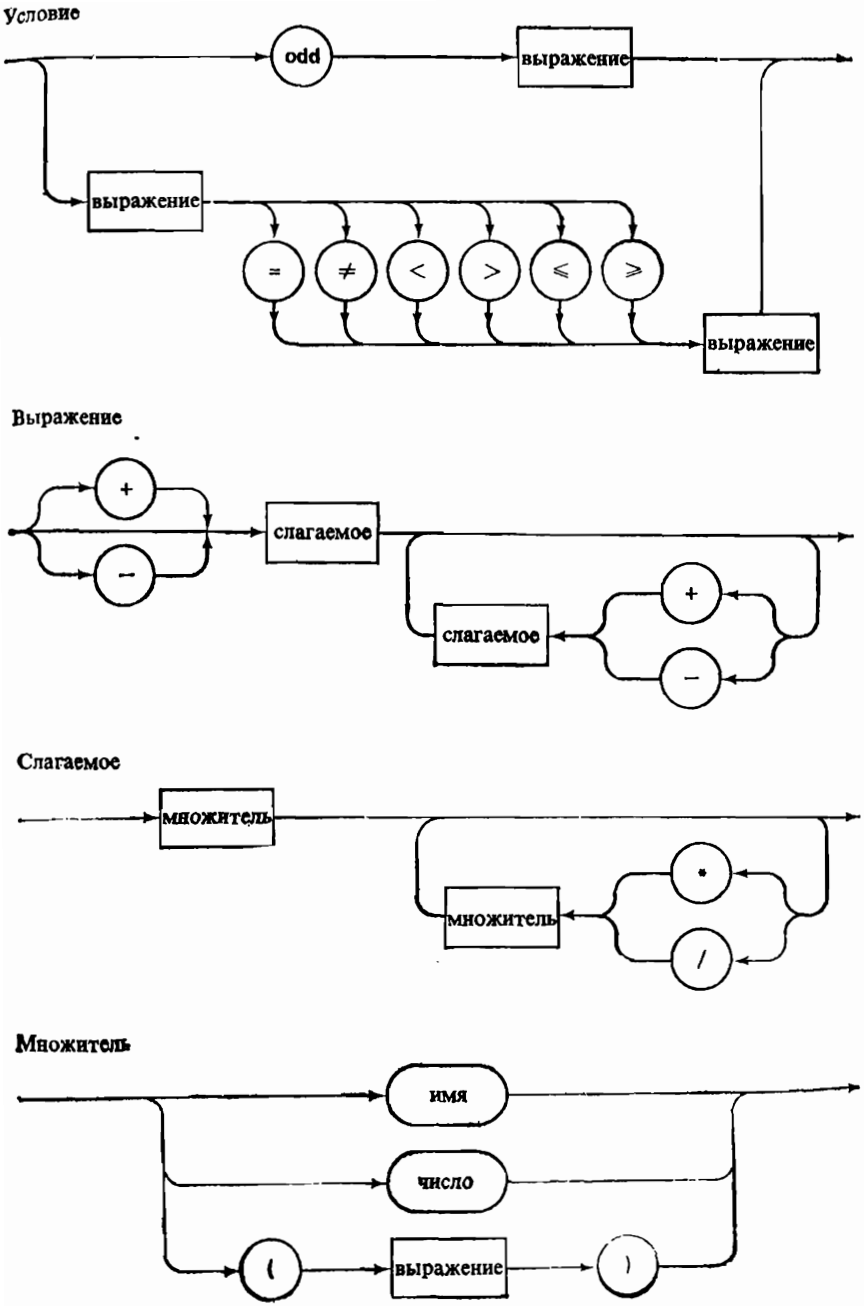


Рис. 5.4. Синтаксис ПЛ/0.

```
const m = 7, n = 85;
var x,y,z,q,r;

procedure multiply;
    var a,b;
begin a := x; b := y; z := 0;
        while b > 0 do                                    (5.1
        begin
            if odd b then z := z + a;
            a := 2*a; b := b/2;
        end
end ;


procedure divide;
    var w;
begin r := x; q := 0; w := y;
    while w ≤ r do w := 2*w;
    while w > y do
        begin q := 2*q; w := w/2;                         (5.1
            if w ≤ r then
                begin r := r-w; q := q+1
                end
        end
end ;


procedure gcd;
    var f,g;
begin f := x; g := y;
    while f ≠ g do
        begin if f < g then g := g-f;
              if g < f then f := f-g;
        end ;
    z := f                                                (5.16
end ;

begin
    x := m; y := n; call multiply;
    x := 25; y := 3; call divide;
    x := 84; y := 36; call gcd;
end .
```

граммам рис. 5.4. Правило 2 относится ко всем графам, которые могут проходиться без чтения какого-либо символа. Единственный такой граф в синтаксисе ПЛ/0 — это тот, который описывает операторы. Ограничение 2 требует, чтобы все начальные символы, которые могут стоять сразу после оператора, отличались от начальных символов операторов. Поскольку в дальнейшем будет полезно знать множества начальных и последующих символов для всех графов, мы определим эти множества для всех 7 нетерминальных символов (графов) синтаксиса ПЛ/0 (кроме «программы»). Используя табл. 5.2, можно убедиться в соблюдении нужного условия, т. е. в том, что множества начальных и последующих символов операторов не пересекаются. Тем самым разрешается применение правил построения программы грамматического разбора B1 — B7.

Таблица 5.2. Начальные и внешние символы в ПЛ/0

Нетерминальный символ S	Начальные символы $L(S)$	Внешние символы $F(S)$
Блок	**const var** *идентификатор* **procedure if call begin while**	. ;
Оператор	*идентификатор* **call begin if while**	. ; **end**
Условие	**odd** + — (*идентификатор число*	**then do**
Выражение	+ — (*идентификатор число*	. ;) R **end then do**
Слагаемое	*идентификатор число* (	. ;) R + — **end then do**
Множитель	*идентификатор число* (	. ;) R + — * / **end then do**

Внимательный читатель должен был заметить, что основные символы ПЛ/0 не являются обычными отдельными буквами, как было в предыдущих примерах. Они могут быть такими последовательностями, как, например, BEGIN или (:=). Как и в программе 5.3, для работы с чисто внешними представлениями или при лексической обработке входной последовательности символов используется так называемый сканер. Он оформлен в виде процедуры *getsym*, задача которой — выбрать из входного файла очередной основной символ. Сканер выполняет следующие действия:

1. Пропускает разделители (пробелы).
2. Распознает зарезервированные слова, такие, как BEGIN, END и т. п.

3. Распознает незарезервированные слова в качестве идентификаторов. Текущий идентификатор присваивается глобальной переменной, называемой *id*.
4. Распознает последовательности цифр в качестве чисел. Значение числа присваивается глобальной переменной *num*.
5. Распознает пары специальных знаков, такие, как (:=).

Рис. 5.5. Диаграмма зависимости для ПЛ/0.

При чтении входной последовательности сканер *getsym* использует локальную процедуру *getch*, которая читает очередной символ. Кроме этой основной задачи *getch* также:

1. Распознает и пропускает информацию о концах строк.
2. Переписывает входной файл в выходной, формируя, таким образом, распечатку программы.
3. Печатает в начале каждой строки ее номер или другую подобную информацию.

С помощью сканера осуществляется необходимый просмотр вперед на один символ. Кроме того, вспомогательная процедура *getch* допускает просмотр еще на один входной символ. Следовательно, наш транслятор «заглядывает» вперед на один основной символ плюс один входной символ.

Подробно эти процедуры показаны в программе 5.4, которая представляет собой полную программу грамматического разбора для ПЛ/0. В действительности она уже несколько расширена в том смысле, что попутно собирает идентификаторы описанных констант, переменных и процедур в *таблицу*. При появлении идентификатора внутри оператора он ищется в этой таблице, так как нужно установить, был ли идентификатор должным образом описан. Отсутствие такого описания можно рассматривать как синтаксическую ошибку, поскольку это формальная ошибка при построении текста программы, а именно использование «незаконного» символа. Тот факт, что эту ошибку можно обнаружить лишь с помощью хранения информации в таблице, есть следствие присущей языку *контекстной зависимости*, проявляющейся в следующем правиле: все идентификаторы соответствующего контекста должны быть описаны. На самом деле практически все языки программирования в этом смысле контекстно зависимы; тем не менее

контекстно-свободный синтаксис позволяет их описать наилучшим образом и очень полезен при построении трансляторов для этих языков. Проверка некоторых контекстных зависимостей может легко включаться в общие схемы, примером чему служит использование таблицы идентификаторов в рассматриваемой программе.

Перед тем как строить отдельные процедуры грамматического разбора, соответствующие отдельным синтаксическим графам, полезно установить, как эти графы взаимосвязаны. Для этого строится так называемая *диаграмма зависимости*; она изображает связи между графами, т. е. для каждого графа G указывает все такие графы $G_1 \ldots G_n$, с помощью

```
program PL0 (input, output);
{транслятор с ПЛ/0, только синтаксический анализ}
label 99;
const norw = 11;        {число зарезервированных слов}
   txmax = 100;         {длина таблицы имен}
   nmax = 14;           {максимальное число цифр в числах}
   al = 10;             {длина имен}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   object = (constant, variable, procedure);
var ch: char;           {последний прочитанный входной символ}
   sym: symbol;         {последний прочитанный символ языка}
   id: alfa;            {последнее прочитанное имя}
   num: integer;        {последнее прочитанное число}
   cc: integer;         {счетчик символов}
   ll: integer;         {длина строки}
   kk: integer;
   line: array [1 .. 81] of char;
   a: alfa;
   word: array [1 .. norw] of alfa;
   wsym: array [1 .. norw] of symbol;
   ssym: array [char] of symbol;
   table: array [0 .. txmax] of
          record name: alfa;
                 kind: object
          end ;
```

```
procedure error (n: integer);
begin writeln (' ':cc, '↑', n:2); goto 99
end {error} ;
procedure getsym;
   var i,j,k: integer;
   procedure getch;
   begin if cc = ll then
      begin if eof(input) then
                begin write (' PROGRAM INCOMPLETE'); goto 99
                end ;
         ll := 0; cc := 0; write(' ');
         while ¬eoln(input) do
            begin ll := ll+1; read(ch); write(ch); line[ll] := ch
            end ;
         writeln; ll := ll+1; read(line[ll])
      end ;
      cc := cc+1; ch := line[cc]
   end {getch} ;
begin {getsym}
while ch = ' ' do getch;
   if ch in ['A'..'Z'] then
   begin {имя, или зарезервированное слово}    k := 0;
      repeat if k < al then
         begin k := k+1; a[k] := ch
         end ;
         getch
      until ¬(ch in ['A'..'Z','0'..'9']);
      if k ≥ kk then kk := k else
         repeat a[kk] := ' '; kk := kk-1
         until kk = k;
      id := a; i := 1; j := norw;
      repeat k := (i+j) div 2;
         if id ≤ word[k] then j := k-1;
         if id ≥ word[k] then i := k+1
      until i > j;
      if i-1 > j then sym := wsym[k] else sym := ident
   end else
   if ch in ['0'..'9'] then
   begin {число} k := 0; num := 0; sym := number;
      repeat num := 10*num + (ord(ch)-ord('0'));
         k := k+1; getch
      until ¬(ch in ['0'..'9']);
      if k > nmax then error (30)
```

```
    end else
    if ch = ':' then
    begin getch;
        if ch = '=' then
        begin sym := becomes; getch
        end else sym := nul;
    end else
    begin sym := ssym[ch]; getch
    end
end {getsym} ;

procedure block (tx: integer);
    procedure enter (k: object);
    begin {запись объекта в таблицу}
        tx := tx + 1;
        with table[tx] do
        begin name := id; kind := k;
        end
    end {enter} ;

    function position (id: alfa): integer;
        var i: integer;
    begin {поиск имени id в таблице}
        table[0].name := id; i := tx;
        while table[i].name ≠ id do i := i-1;
        position := i
    end {position} ;

    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym = eql then
            begin getsym;
                if sym = number then
                    begin enter (constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
    end {constdeclaration} ;

    procedure vardeclaration;
    begin if sym = ident then
            begin enter (variable); getsym
            end else error (4)
    end {vardeclaration} ;
```

```
procedure statement;
    var i: integer;
    procedure expression;
        procedure term;
            procedure factor;
                var i: integer;
            begin
                if sym = ident then
                begin i := position(id);
                    if i = 0 then error (11) else
                    if table[i].kind ≠ procedure then error (21);
                    getsym
                end else
                if sym = number then
                begin getsym
                end else
                if sym = lparen then
                begin getsym; expression;
                    if sym = rparen then getsym else error (22)
                end
                else error (23)
            end {factor} ;
            begin {term} factor;
                while sym in [times, slash] do
                    begin getsym; factor
                    end
            end {term} ;
        begin {expression}
            if sym in [plus, minus] then
                begin getsym; term
                end else term;
            while sym in [plus, minus] do
                begin getsym; term
                end
        end {expression} ;
        procedure condition;
        begin
            if sym = oddsym then
            begin getsym; expression
            end else
            begin expression;
                if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
                    error (20) else
```

```
            begin getsym; expression
            end
        end
    end {condition} ;
    begin {statement}
    if sym = ident then
    begin i := position(id);
        if i = 0 then error (11) else
        if table [i] .kind ≠ variable then error (12);
        getsym; if sym = becomes then getsym else error (13);
        expression
    end else
    if sym = callsym then
    begin getsym;
        if sym ≠ ident then error (14) else
            begin i := position(id);
                if i = 0 then error (11) else
                if table[i] .kind ≠ procedure then error (15);
                getsym
            end
    end else
    if sym = ifsym then
    begin getsym; condition;
        if sym = thensym then getsym else error (16);
        statement;
    end else
    if sym = beginsym then
    begin getsym; statement;
        while sym = semicolon do
            begin getsym; statement
            end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; condition;
        if sym = dosym then getsym else error (18);
        statement
    end
end {statement} ;
begin {block}
    if sym = constsym then
    begin getsym; constdeclaration;
```

```
        while sym = comma do
            begin getsym; constdeclaration
            end ;
        if sym = semicolon then getsym else error (5)
    end ;
    if sym = varsym then
    begin getsym; vardeclaration;
        while sym = comma do
            begin getsym; vardeclaration
            end ;
        if sym = semicolon then getsym else error (5)
    end ;
    while sym = procsym do
    begin getsym;
        if sym = ident then
            begin enter (procedure); getsym
            end
        else error (4);
        if sym = semicolon then getsym else error (5);
        block (tx);
        if sym = semicolon then getsym else error (5);
    end ;
    statement
end {block} ;

begin {основная программа}
    for ch := 'A' to ';' do ssym[ch] := nul;
    word[ 1] := 'BEGIN ';   word[ 2] := 'CALL      ';
    word[ 3] := 'CONST';    word[ 4] := 'DO        ';
    word[ 5] := 'END   ';   word[ 6] := 'IF        ';
    word[ 7] := 'ODD   ';   word[ 8] := 'PROCEDURE';
    word[ 9] := 'THEN  ';   word[10] := 'VAR       ';
    word[11] := 'WHILE ';
    wsym[ 1] := beginsym;   wsym[ 2] := callsym;
    wsym[ 3] := constsym;   wsym[ 4] := dosym;
    wsym[ 5] := endsym;     wsym[ 6] := ifsym;
    wsym[ 7] := oddsym;     wsym[ 8] := procsym;
    wsym[ 9] := thensym;    wsym[10] := varsym;
    wsym[11] := whilesym;
    ssym['+'] := plus;      ssym['-'] := minus;
    ssym['*'] := times;     ssym['/'] := slash;
    ssym['('] := lparen;    ssym[')'] := rparen;
    ssym['='] := eql;       ssym[','] := comma;
```

```
    ssym['.'] := period;        ssym['≠'] := neq;
    ssym['<'] := lss;           ssym['>'] := gtr;
    ssym['≤'] := leq;           ssym['≥'] := geq;
    ssym[';'] := semicolon;
    page(output);
    cc := 0; ll := 0; ch := ' '; kk := al; getsym;
    block (0);
    if sym ≠ period then error (9);
99: writeln
end .
```

Программа 5.4. Грамматический разбор для ПЛ/0.

которых определен G. Соответственно это определяет, какие процедуры будут вызываться другими процедурами. Диаграмма зависимости для ПЛ/0 показана на рис. 5.5.

Циклы на рис. 5.5 обозначают появление рекурсии. Поэтому важно, чтобы в языке, на котором реализуется транслятор ПЛ/0, была разрешена рекурсия. Кроме того, диаграмма зависимости позволяет сделать выводы об иерархической организации программы грамматического разбора. Например, все процедуры могут содержаться (быть описаны как локальные) в процедуре, которая анализирует конструкцию ⟨программа⟩ (которая поэтому будет главной программой в программе грамматического разбора). Далее, все процедуры, изображенные на диаграмме ниже ⟨блок⟩, могут определяться как локальные в подпрограмме, представляющей цель разбора ⟨блок⟩. Разумеется, все эти процедуры вызывают сканер *getsym*, который в свою очередь вызывает *getch*.

5.9. ВОССТАНОВЛЕНИЕ ПРИ СИНТАКСИЧЕСКИХ ОШИБКАХ

До сих пор программа грамматического разбора лишь устанавливала, принадлежит ли входная последовательность символов языку. В качестве побочного результата она также определяла структуру предложения. Но если встречалась неправильная конструкция, задача программы могла считаться выполненной и она могла закончить работу. Разумеется, на практике такая схема неприемлема. Вместо этого транслятор должен выдавать соответствующую диагностику об ошибках и продолжать процесс грамматического разбора, возможно находя дальнейшие ошибки. Чтобы продолжать работу, нужно либо сделать какие-то предположения о том, что на самом

деле имел в виду автор неправильной программы, либо пропустить некоторую часть входной последовательности, либо сделать и то, и другое. Сделать достаточно разумные предположения о действительных намерениях программиста — довольно сложно. До сих пор это не удавалось формализовать, поскольку формальный подход к синтаксису и грамматическому разбору не учитывает многие факторы, сильно влияющие на человеческое сознание. Например, распространенной ошибкой является пропуск знаков пунктуации, таких, как точка с запятой (не только в программировании!), но весьма маловероятно, что кто-то пропустит знак «+» в арифметическом выражении. Для программы грамматического разбора и точка с запятой, и плюс — просто терминальные символы без какого-либо существенного различия; а для человека точка с запятой почти не имеет значения и в конце строки кажется избыточной, тогда как знак арифметической операции, бесспорно, осмыслен *). При разработке подходящей системы восстановления следует принимать во внимание многие подобные соображения, которые связаны с конкретным языком и не могут обобщаться для всех контекстно-свободных языков.

Все же существуют некоторые правила и рекомендации, действующие не только в рамках одного языка, такого, как ПЛ/0. Для них, пожалуй, характерно, что они в равной мере связаны как с исходной концепцией языка, так и с механизмом восстановления в программе грамматического разбора. Прежде всего совершенно ясно, что эффективное восстановление возможно или, во всяком случае, намного облегчается лишь в случае языка с *простой структурой*. В частности, если при обнаружении ошибки пропускается какая-то часть входной последовательности, то язык обязательно должен содержать *служебные слова*, неправильное употребление которых крайне маловероятно и которые поэтому могут использоваться для возобновления грамматического разбора. В ПЛ/0 это правило строго соблюдается: каждый оператор начинается с однозначного служебного слова, такого, как **begin**, **if**, **while**; то же относится к описаниям: они начинаются с **var**, **const** или **procedure**. Мы назовем это *правилом служебных слов*.

Второе правило более непосредственно связано с построением программы грамматического разбора. Для нисходящего

---

*) Может быть, эти соображения натолкнут читателя на мысль, что в современных языках программирования слишком уж много внимания было уделено формальной синтаксической проблематике. И это на этапе, когда мы ставим задачу общения с машиной на естественном языке! — *Прим. ред.*

анализа характерно, что цели разбиваются на подцели; при этом процедуры вызывают другие процедуры, соответствующие этим подцелям. Второе правило определяет, что если процедура грамматического разбора обнаруживает ошибку, то она не должна прекращать работу и сообщать о случившемся вызвавшей ее процедуре. Вместо этого она должна самостоятельно продолжать просмотр текста до того места, откуда можно возобновить анализ. Мы назовем это *правилом «не поднимай панику»*. Из него следует, что из процедуры грамматического разбора не может быть другого выхода, кроме обычного завершения работы.

Правило «не поднимай панику» можно интерпретировать следующим образом: при появлении неправильной конструкции процедура должна пропустить входной текст, пока не встретится символ, который по правилам может следовать за той конструкцией языка, которую она пыталась обнаружить. Это означает, что каждой процедуре грамматического разбора в момент текущей ее активации должно быть известно множество внешних символов.

Поэтому на первом этапе уточнения (или дополнения) мы снабдим каждую процедуру грамматического разбора явным параметром *fsys*, который задает возможные внешние символы. В конце каждой процедуры вставляется явная проверка: действительно ли следующий символ входного текста содержится среди этих внешних символов (если это уже не обусловлено логикой программы)?

Все же с нашей стороны было бы неразумно при всех обстоятельствах пропускать входной текст до следующего появления такого внешнего символа. В конце концов, программист мог по ошибке пропустить всего один символ (например, точку с запятой), а игнорирование текста до следующего внешнего символа может иметь гибельные последствия. Поэтому ко множествам символов, которые прекращают пропуск текста, мы добавим служебные слова, отмечающие начало конструкции, которую не следует пропускать. Таким образом, символы, передаваемые в качестве параметров процедуре грамматического разбора, — это *символы возобновления*, а не просто внешние символы. Мы можем считать, что множества символов возобновления с самого начала содержат отдельные служебные слова и при проходе иерархии подцелей грамматического разбора постепенно дополняются внешними символами этих подцелей. Для удобства вводится общая подпрограмма, называемая *test*, — она выполняет описанную выше проверку. Эта процедура (5.17) имеет три параметра:

1. Множество *s1* допустимых следующих символов; если текущий символ к нему не принадлежит, то имеет место ошибка

2. Множество $s2$ дополнительных символов возобновления, появление которых определенно является ошибкой, но которые ни в коем случае не следует пропускать.
3. Номер n, который присваивается ошибке, если процедура ее обнаружит.

```
procedure test (s1, s2: symset; n: integer);
begin if ¬(sym in s1) then
      begin error(n); s1 := s1+s2;
            while ¬(sym in s1) do getsym
      end
end
```
(5.17)

Процедуру (5.17) удобно использовать при *входе* в процедуру грамматического разбора для проверки, является ли текущий символ допустимым начальным символом анализируемой конструкции. Это рекомендуется во всех случаях, когда процедура X вызывается безусловно, как в операторе

$$\begin{aligned} &\textbf{if } sym = a_1 \textbf{ then } S_1 \textbf{ else}\\ &\quad\ldots\ldots\ldots\ldots\\ &\textbf{if } sym = a_n \textbf{ then } S_n \textbf{ else } X \end{aligned}$$

который получен из порождающего правила

$$A ::= a_1 S_1 \mid \ldots \mid a_n S_n \mid X. \tag{5.18}$$

В этих случаях параметр s1 должен содержать множество начальных символов X, а $s2$ — множество внешних символов A (см. табл. 5.2). Подробно эта процедура показана в программе 5.5, которая представляет собой дополненную версию программы 5.4. Для удобства читателя вся программа приведена полностью, за исключением инициации глобальных переменных и процедуры *getsym*, оставшихся без изменений.

Предполагаемая до сих пор схема пытается возобновить грамматический разбор, вернуться на правильный путь, пропустив один или более символов входного текста. Но это неудачная стратегия во всех случаях, когда ошибка вызвана *пропуском* символа. Опыт говорит о том, что такие ошибки фактически связаны с символами, выполняющими чисто синтаксические функции и не представляющими какого-либо действия. Примером служит точка с запятой в ПЛ/0. Однако, поскольку множества внешних символов дополнены определенными служебными словами, на самом деле грамматический разбор вовремя прекращает пропуск символов, т. е. ведет себя так, как если бы забытый символ был вставлен. Это видно из той части программы (5.19), которая анализирует

составные операторы. Она «вставляет» пропущенные точки с запятой перед ключевыми словами. Множество, называемое *statbegsys*, есть множество начальных символов конструкции «оператор».

```
if sym = beginsym then
begin getsym;
   statement([semicolon, endsym]+fsys);
   while sym in [semicolon]+statbegsys do
      begin
         if sym = semicolon then getsym else error;
         statement([semicolon, endsym]+fsys)
      end;
   if sym = endsym then getsym else error
end
```
(5.19)

О том, насколько успешно эта программа обнаруживает синтаксические ошибки и справляется с необычными ситуациями, можно судить по программе ПЛ/0 (5.20). Ее распечатка является результатом работы программы 5.5, а в табл. 5.3 перечислены возможные сообщения об ошибках, соответствующие номерам ошибок в программе 5.5.

Таблица 5.3. Сообщения об ошибках, выдаваемые транслятором с ПЛ/0

1. = вместо :=
2. Нет числа после =
3. Нет = после идентификатора
4. Нет идентификатора после **const, var, procedure**
5. Пропущена запятая или точка с запятой
6. Неверный символ после описания процедуры
7. Нет оператора
8. Неверный символ после операторной части блока
9. Нет многоточия
10. Пропущена точка с запятой между операторами
11. Неописанный идентификатор
12. Недопустимое присваивание константе или процедуре.
13. Требуется :=
14. Нет идентификатора после **call**
15. Вызов константы или переменной вместо процедуры
16. Требуется **then**
17. Требуется точка с запятой или **end**
18. Требуется **do**
19. Неверный символ после оператора
20. Требуется сравнение
21. Выражение содержит идентификатор процедуры
22. Отсутствует правая скобка
23. Неверный символ после множителя
24. Неверный символ в начале выражения
30. Слишком большое число

Программа (5.20) получена в результате намеренного введения синтаксических ошибок в (5.14)—(5.16).

```
const m = 7, n = 85
var x,y,z,q,r;
    ↑ 5
          ↑ 5
procedure multiply;
    var a,b
    begin a := x; b := y; z := 0
    ↑ 5
          ↑11
        while b > 0 do
          ↑10
        begin
          if odd b do z := z + a;
                      ↑16
                      ↑19
          a := 2a; b := b/2;
                ↑23
        end
    end ;
procedure divide
    var w;
        ↑ 5
    const two = 2, three := 3;
        ↑ 7
                          ↑ 1
begin r := x; q := 0; w := y;
          ↑13
          ↑24
    while w ≤ r do w := two*w;
    while w > y
        begin q := (2*q; w := w/2);
          ↑18
                    ↑22
                                ↑23
          if w ≤ r then
              begin r := r−w q := q+1
                              ↑23
              end
        end
end ;
procedure gcd;
    var f,g;
begin f := x; g := y
```

```
        while f ≠ g do
            ↑17
            begin if f < g then g := g−f;
                  if g < f then f := f−g;
        z := f
    end ;
    begin
        x := m; y := n; call multiply;
        x := 25; y := 3; call divide;
        x := 84; y := 36; call gcd;
        call x; x := gcd; gcd =x
            ↑15
                      ↑21
                            ↑12
                              ↑13
                              ↑24
    end .
        ↑17
        ↑ 5
        ↑ 7
    PROGRAM INCOMPLETE                                    (5.20)
```

Все же ясно, что никакая схема, которая достаточно эффективно транслирует правильные предложения, не сможет так же эффективно справляться со всеми возможными неправильными конструкциями. Да и как может быть иначе!

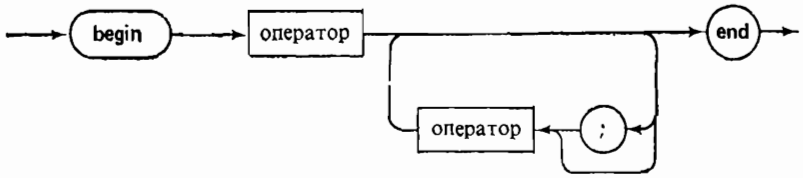


Рис. 5.6. Синтаксис с модифицированным составным оператором.

Любая схема восстановления, реализованная с разумными затратами, потерпит неудачу, т. е. не сможет адекватно обработать некоторые ошибочные конструкции. Однако хороший транслятор должен обладать такими важными свойствами:

1. Никакая входная последовательность не должна приводить к катастрофе.
2. Все конструкции, которые по определению языка являются незаконными, должны обнаруживаться и отмечаться.

```
program PL0 (input ,output);
{транслятор с ПЛ/0 с восстановлением при синтаксических ошибках}
label 99;
const norw = 11;        {количество зарезервированных слов}
    txmax = 100;        {длина таблицы имен}
    nmax = 14;          {максимальное количество цифр в числах}
    al = 10;            {длина имен}
type symbol =
    (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
    alfa = packed array [1 . . al] of char;
    object = (constant, variable, procedure);
    symset = set of symbol;
var ch: char;           {последний прочитанный входной символ}
    sym: symbol;        {последний прочитанный символ языка}
    id: alfa;           {последнее прочитанное имя}
    num: integer;       {последнее прочитанное число}
    cc: integer;        {счетчик символов}
    ll: integer;        {длина строки}
    kk: integer;
    line: array [1 . . 81] of char;
    a: alfa;
    word: array [1 . . norw] of alfa;
    wsym: array [1 . . norw] of symbol;
    ssym: array [char] of symbol;
    declbegsys, statbegsys, facbegsys: symset;
    table: array [0 . . txmax] of
          record name: alfa;
                 kind: object
          end ;
procedure error (n: integer);
begin writeln(' ':cc, '^', n:2);
end {error} ;
procedure test (s1,s2: symset; n: integer);
begin if ¬(sym in s1) then
       begin error(n); s1 := s1 + s2;
          while ¬(sym in s1) do getsym
       end
end {test} ;
```

```
procedure block (tx: integer; fsys: symset);
    procedure enter (k: object);
    begin {запись объекта в таблицу}
        tx := tx + 1;
        with table[tx] do
        begin name := id; kind := k;
        end
    end {enter} ;
    function position (id: alfa): integer;
        var i: integer;
    begin {поиск имени id в таблице}
        table[0] .name := id; i := tx;
        while table[i] .name ≠ id do i := i-1;
        position := i
    end {position} ;
    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym in [eql, becomes] then
            begin if sym = becomes then error (1);
                getsym;
                if sym = number then
                    begin enter (constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
        end {constdeclaration} ;
    procedure vardeclaration;
    begin if sym = ident then
            begin enter (variable); getsym
            end else error (4)
    end {vardeclaration} ;
procedure statement (fsys: symset);
var i: integer;
procedure expression (fsys: symset);
    procedure term (fsys: symset);
        procedure factor (fsys: symset);
            var i: integer;
        begin test (facbegsys, fsys, 24);
            while sym in facbegsys do
```

```
            begin
                if sym = ident then
                begin i := position (id);
                    if i = 0 then error (11) else
                    if table[i] .kind = procedure then error (21);
                    getsym
                end else
                if sym = number then
                begin getsym;
                end else
                if sym = lparen then
                begin getsym; expression ([rparen]+fsys);
                    if sym = rparen then getsym else error (22)
                end ;
                test(fsys, [lparen], 23)
            end
        end {factor} ;
    begin {term} factor (fsys+[times, slash]);
        while sym in [times, slash] do
            begin getsym; factor(fsys+[times, slash])
            end
    end {term} ;
begin {expression}
    if sym in [plus ,minus] then
        begin getsym; term(fsys+[plus, minus])
        end else term(fsys+[plus, minus]);
    while sym in [plus, minus] do
        begin getsym; term(fsys+[plus, minus])
        end
end {expression} ;
procedure condition(fsys: symset);
begin
    if sym = oddsym then
    begin getsym; expression(fsys);
    end else
    begin expression ([eql, neq, lss, gtr, leq, geq]+fsys);
        if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
            error (20) else
        begin getsym; expression (fsys)
        end
    end
end {condition} ;
```

```
begin {statement}
    if sym = ident then
    begin i := position(id);
        if i = 0 then error (11) else
        if table[i].kind ≠ variable then error (12);
        getsym; if sym = becomes then getsym else error (13);
        expression(fsys);
    end else
    if sym = callsym then
    begin getsym;
        if sym ≠ ident then error(14) else
            begin i := position(id);
                if i = 0 then error (11) else
                if table[i].kind ≠ procedure then error (15);
                getsym
            end
    end else
    if sym = ifsym then
    begin getsym; condition ([thensym, dosym]+fsys);
        if sym = thensym then getsym else error (16);
        statement(fsys)
    end else
    if sym = beginsym then
    begin getsym; statement([semicolon, endsym]+fsys);
        while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error (10);
            statement([semicolon ,endsym]+fsys)
        end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; condition([dosym]+fsys);
        if sym = dosym then getsym else error (18);
        statement(fsys);
    end ;
    test(fsys, [ ], 19)
end {statement} ;
begin {block}
    repeat
        if sym = constsym then
```

```
        begin getsym;
            repeat constdeclaration;
                while sym = comma do
                    begin getsym; constdeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident
        end ;
        if sym = varsym then
        begin getsym;
            repeat vardeclaration;
                while sym = comma do
                    begin getsym; vardeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident;
        end ;
        while sym = procsym do
        begin getsym;
            if sym = ident then
                begin enter (procedure); getsym
                end
            else error (4);
            if sym = semicolon then getsym else error (5);
            block (tx, [semicolon]+fsys);
            if sym = semicolon then
                begin getsym; test(statbegsys+[ident, procsym], fsys, 6)
                end
            else error (5)
        end ;
        test(statbegsys+[ident], declbegsys, 7)
    until ¬(sym in declbegsys);
    statement([semicolon, endsym]+fsys);
    test(fsys, [ ], 8);
end {block} ;
begin {основная программа}
    ... Инициация(см. программу 5.4)...
    cc := 0; ll := 0; ch := ' '; kk := al; getsym;
    block (0, [period]+declbegsys+statbegsys);
    if sym ≠ period then error (9);
99: writeln
end .
```

Программа 5.5. Грамматический разбор для ПЛ/0 с восстановлением при ошибках.

3. Ошибки, встречающиеся довольно часто и действительно являющиеся ошибками программиста (вызванными недосмотром или недопониманием), должны правильно диагностироваться и не вызывать каких-либо дальнейших отклонений в работе транслятора — сообщений о так называемых *наведенных* ошибках.

Предлагаемая схема восстановления работает удовлетворительно, хотя, как всегда, возможно ее дальнейшее усовершенствование. Ее преимущество в том, что она построена систематическим образом по нескольким основным правилам. Эти основные правила просто разработаны с помощью выбора параметров, основанного на эвристических соображениях и опыте практического использования языка.

5.10. ПРОЦЕССОР ПЛ/0

В самом деле, примечательно, что до сих пор транслятор ПЛ/0 разрабатывался в полном неведении, для какой машины он должен формировать рабочую программу. Да и с какой стати структура машины должна влиять на схему синтаксического анализа и восстановления при ошибках? Более того, она действительно *не должна* влиять. Вместо этого собственно схема формирования кода для любой вычислительной машины должна накладываться на алгоритм грамматического разбора методом поэтапного уточнения существующей программы. Поскольку теперь мы готовы к этому, нужно выбрать процессор, для которого производится трансляция.

Чтобы описание транслятора оставалось достаточно простым и свободным от посторонних соображений, связанных с конкретными особенностями какого-либо реального процессора, мы придумаем свою собственную вычислительную машину, специально приспособленную для ПЛ/0. Это некий гипотетический процессор, который не существует на самом деле (в аппаратном виде); мы назовем его *машиной ПЛ/0*.

В этом разделе мы не будем подробно объяснять, почему мы выбрали именно такую машинную архитектуру. Вместо этого данный раздел будет служить руководством по процессору, состоящим из вводного интуитивного описания, за которым будет следовать подробное определение процессора с помощью алгоритма. Эта формализация может служить примером аккуратного и подробного описания для реальных процессоров. Наш алгоритм последовательно интерпретирует команды ПЛ/0 и называется *интерпретатором*.

В машину ПЛ/0 входят две области памяти, регистр команды и три регистра адресов. *Память для программы*, называемой *рабочей программой*, загружается транслятором

и во время интерпретации программы не изменяется. Ее можно считать памятью, допускающей только считывание. *Область памяти для хранения данных* организована в виде *стека*, и все арифметические действия выполняются с двумя элементами на вершине стека, причем результат записывается на место операндов. Верхний элемент адресуется (индексируется) с помощью *регистра вершины стека T*. *Регистр команды I* содержит команду, которая интерпретируется в данный момент. *Регистр адреса команды P* указывает следующую команду, которую нужно будет интерпретировать.

Каждая процедура в машине ПЛ/0 может содержать локальные переменные. Поскольку процедуры могут вызываться рекурсивно, память для этих переменных нельзя выделить до действительного обращения к процедуре. Следовательно, сегменты данных для отдельных процедур последовательно помещаются в стек *S*. Так как вызовы процедур строго подчинены схеме «первым вошел — последним вышел», стек является подходящим способом размещения. Каждая процедура обладает своей собственной информацией: адресом команды ее вызова (так называемым *адресом возврата*), и адресом сегмента данных вызвавшей ее процедуры. Эти два адреса нужны для правильного возобновления работы программы после завершения работы процедуры. Их можно рассматривать как внутренние, или неявные локальные переменные, помещаемые в сегменте данных процедуры. Мы называем их *адресом возврата* RA и *динамической связкой* DL. Начало динамической цепочки, т. е. адрес размещенного последним сегмента данных, сохраняется в регистре базового адреса *B*.

Поскольку действительное выделение памяти происходит во время выполнения (интерпретации) программы, транслятор не может формировать рабочую программу с абсолютными адресами. Он может лишь задать расположение переменных внутри сегмента данных, поэтому способен выдавать только *относительные адреса*. Интерпретатор должен добавлять к этому так называемому *смещению* базовый адрес соответствующего сегмента данных. Если переменная локальна в процедуре, интерпретируемой в данный момент, то этот базовый адрес задается регистром *B*. В противном случае его можно получить, спускаясь по цепочке сегментов данных. Однако транслятору может быть известна только статическая глубина пути доступа к переменной, тогда как динамическая цепочка (цепочка динамических связок) отражает динамическую историю обращений к процедурам. К сожалению, эти два пути доступа не обязательно одинаковы.

Например, пусть процедура *A* обращается к процедуре *B*, описанной как локальная в *A*, процедура *B* вызывает *C*, опи-

санную как локальная в B, а C вызывает B рекурсивно. Мы говорим, что A описана на уровне 1, B — на уровне 2, C — на уровне 3 (см. рис. 5.7). Если в B имеется обращение к переменной a, описанной как локальная в A, то транслятору известно, что между A и B существует разница уровней, равная 1. Однако один шаг по динамической цепочке приводит к переменной, локальной в C!

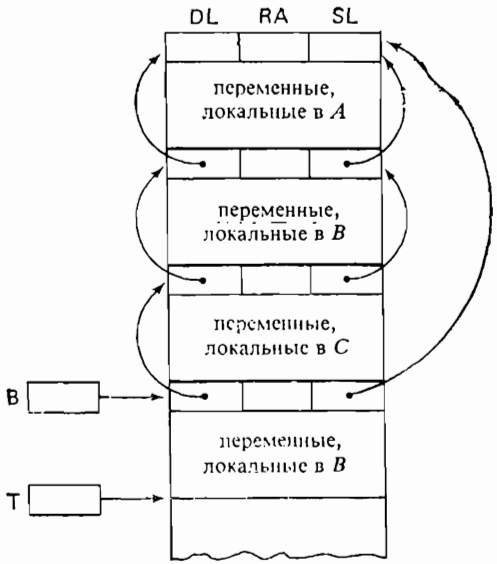


Рис. 5.7. Стек машины для ПЛ/0.

Поэтому ясно, что нужно иметь вторую цепочку связей, которая связывает сегменты данных таким способом, чтобы транслятор мог правильно воспринимать ситуацию. Мы назовем элементы этой цепочки *статической связкой* SL.

Итак, адреса формируются в виде пар чисел, указывающих статическую разность уровней и относительное смещение внутри сегмента данных. Мы считаем, что каждая ячейка памяти может содержать адрес или целое число.

Множество команд машины ПЛ/0 приспособлено к требованиям языка ПЛ/0. Оно содержит такие команды:

1. Засылки чисел (констант) в стек (LIT).
2. Считывания переменных в вершину стека (LOD).
3. Записи значения, находящегося в вершине стека (STO). (Соответствует оператору присваивания.)
4. Команда активации подпрограммы, соответствующая обращению к процедуре (CAL).

5. Выделение памяти для стека путем увеличения указателя стека T (INT).
6. Команды условной и безусловной передачи управления, используемые в условных операторах и циклах (JMP, JPC).
7. Команда, выполняющая арифметические действия и сравнения (OPR).

Так как в команду должны входить три компоненты, то она имеет следующий формат (см. рис. 5.8). Здесь присут-

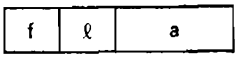


Рис. 5.8. Формат команды.

ствуют код операции (f) и параметр, состоящий из одной или двух частей (a и b либо только a)*). В случае команды OPR, выполняющей некоторое действие, параметр a задает это действие; в других случаях это либо число (LIT, INT), либо адрес в программе (JMP, JPC, CAL), либо адрес данных (LOD, STO).

Подробно работа машины ПЛ/0 определяется процедурой, называемой *interpret*, которая является частью программы 5.6, объединяющей законченный транслятор с интерпретатором в систему, которая транслирует, а затем выполняет программы на ПЛ/0. Мы предлагаем в качестве упражнения модифицировать эту программу, чтобы она формировала рабочую программу для какого-либо существующего процессора. Необходимое для этого увеличение программы можно считать мерой того, насколько выбранная вычислительная машина подходит для поставленной задачи.

Безусловно, представленную вычислительную машину ПЛ/0 можно было бы организовать более искусно, с тем чтобы некоторые операции выполнялись эффективнее. Например, можно было бы улучшить механизм адресации. Принятая схема была выбрана из-за ее простоты, а также потому, что все усовершенствования должны фактически основываться на ней и из нее выводиться.

5.11. ФОРМИРОВАНИЕ КОМАНД

Чтобы транслятор мог сформировать команду, ему должны быть известны ее код операции и параметр, который представляет собой либо число (саму константу), либо адрес. Эти значения транслятор связывает с соответствующими идента-

*) Поле b в команде (или параметре) используется для указания уровня (*level*). — *Прим. ред.*

фикаторами при обработке описаний констант, переменных и процедур. Для этой цели таблица идентификаторов дополнена атрибутами, присущими каждому идентификатору. Если идентификатор обозначает константу, то его атрибутом является значение этой константы, если идентификатор обозначает переменную, то его атрибутом является ее адрес, состоящий из смещения и уровня, а если он обозначает процедуру, то его атрибутами являются адрес входа в эту процедуру и ее уровень. Расширенное соответствующим образом описание переменной *table* («таблица») показано в программе 5.6. Это наглядный пример поэтапного уточнения (или дополнения) описания данных, происходящего одновременно с уточнением операторной части программы.

В то время как значения констант задаются в тексте программы, определять адреса транслятор должен самостоятельно. Язык ПЛ/0 достаточно прост, поэтому переменные и команды размещаются в памяти последовательно. Следовательно, каждое описание переменной сопровождается увеличением индекса размещения данных на 1 (так как каждая переменная по определению машины ПЛ/0 занимает ровно одну ячейку памяти). Индекс размещения данных *dx* должен инициироваться в начале трансляции любой процедуры, поскольку ее сегмент данных первоначально пуст. [В действительности *dx* получает начальное значение 3, так как каждый сегмент данных содержит по крайней мере три внутренние переменные RA, DL, SL (см. предыдущий раздел).] Соответствующие вычисления, позволяющие определить атрибуты идентификатора, включены в процедуру *enter*, которая добавляет в таблицу новые идентификаторы.

При наличии этой информации об операндах команды формируются довольно просто. Благодаря стековой организации машины ПЛ/0 существует практически однозначное соответствие между операндами и операциями исходного языка, с одной стороны, и командами рабочей программы, с другой стороны. Транслятор лишь должен выполнить необходимое преобразование в *постфиксную форму*. Постфиксная форма означает, что знаки операции всегда следуют за своими операндами, а не вставляются между ними, как в обычной *инфиксной форме*. Постфиксную форму иногда также называют польской записью (так как ее «изобрел» поляк Лукасевич) или *бесскобочной записью*, так как она делает скобки излишними. Примеры соответствия между инфиксной и постфиксной формами записи выражений приведены в табл. 5.4 (см. также разд. 4.4.2).

Очень простой способ выполнения такого преобразования описан в процедурах *expression* и *term* из программы 5.6. Он состоит в том, что передача знака арифметической операции

Таблица 5.4. Выражения в инфиксной и постфиксной записях

Инфиксная запись	Постфиксная запись
$x + y$	$xy+$
$(x - y) + z$	$xy - z+$
$x - (y + z)$	$xyz+-$
$x*(y + z)*w$	$xyz + *w*$

просто задерживается. В этот момент читатель должен убедиться, что взаимная связь процедур грамматического разбора учитывает соответствующую интерпретацию принятых правил приоритета различных операций.

Трансляция условных операторов и циклов несколько менее тривиальна. В этом случае нужно формировать команды перехода, для которых сам адрес перехода иногда еще неизвестен. Если обязательно, чтобы формируемые команды располагались строго последовательно в виде выходного файла, то необходима *двупроходная* схема транслятора. На втором проходе неполные команды перехода дополняются адресами. Другое решение, реализованное в данном трансляторе, — это помещение команды в массив, т. е. в память с непосредственным доступом, что позволяет вставлять недостающие адреса, как только они становятся известны. Такую операцию иногда называют *фиксацией* (*fixup*).

Единственное дополнительное действие, которое нужно выполнять при формировании такого перехода вперед, — это запоминание его местоположения, т. е. индекса в памяти для программы. Затем во время фиксации этот адрес используется для нахождения неполной команды. Детали опять можно видеть в программе 5.6 (см. процедуры, обрабатывающие операторы условия и цикла). Команды, соответствующие этим операторам, формируются по следующему шаблону (*L*1 и *L*2 означают адреса команд):

if *C* **then** *S*	**while** *C* **do** *S*
команды для условия *C*	L1: команды для *C*
JPC L1	JPC L2
команды оператора *S*	команды для *S*
L1: ...	JMP L1
	L2: ...

Для удобства вводится вспомогательная процедура, называемая *gen*. Ей задаются три параметра, из которых она формирует команду. При этом автоматически увеличивается индекс *cx*, указывающий место, куда помещается очередная команда.

Ниже в мнемонической форме приведена программа, полученная при трансляции процедуры умножения (5.14). Комментарии с правой стороны добавлены лишь для пояснения,

```
2   INT   0,5   выделить память для связок и локальных переменных
3   LOD   1,3   x
4   STO   0,3   a
5   LOD   1,4   y
6   STO   0,4   b
7   LIT   0,0   0
8   STO   1,5   z
9   LOD   0,4   b
10  LIT   0,0   0
11  OPR   0,12  >
12  JPC   0,29
13  LOD   0,4   b
14  OPR   0,7   нечетно
15  JPC   0,20
16  LOD   1,5   z
17  LOD   0,3   a
18  OPR   0,2   +
19  STO   1,5   z
20  LIT   0,2   2
21  LOD   0,3   a
22  OPR   0,4   *
23  STO   0,3   a
24  LOD   0,4   b
25  LIT   0,2   2
26  OPR   0,5   /
27  STO   0,4   b
28  JMP   0,9
29  OPR   0,0   возврат
```

Рабочая программа, соответствующая процедуре ПЛ/0 (5.14).

При трансляции с языков программирования обычно приходится решать значительно более сложные задачи, чем те, которые решал транслятор с языка ПЛ/0 для машины ПЛ/0 [5.4]. Большинство из них с гораздо большим трудом поддаются четкой организации. Если читатель попытается расширить данный транслятор, приспособив его либо для более мощного языка, либо для более привычной вычислительной машины, то он вскоре убедится в правоте этого утверждения. Тем не менее основной изложенный здесь подход к разработке сложных программ по-прежнему остается в силе, и его ценность даже возрастает в случае более тонких и сложных задач. Он действительно успешно применялся при построении крупных трансляторов [5.1, 5.9].

```
program PL0(input,output);
{транслятор с ПЛ/0 с формированием рабочей программы}
label 99;
const norw = 11;      {число зарезервированных слов}
   txmax = 100;       {длина таблицы имен}
   nmax = 14;         {максимальное количество цифр в числах}
   al = 10;           {длина имен}
   amax = 2047;       {максимальный адрес}
   levmax = 3;        {максимальная глубина вложенности блоков}
   cxmax = 200;       {размер массива кодов}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   object = (constant, variable, procedure);
   symset = set of symbol;
   fct = (lit, opr, lod, sto, cal, int, jmp, jpc);      {функции}
   instruction = packed record
                    f: fct;          {код функции}
                    l: 0 .. levmax;  {уровень}
                    a: 0 .. amax;    {смещение}
                 end ;
{  LIT 0,a  :  загрузка константы a
   OPR 0,a  :  выполнение операции a
   LOD l,a  :  загрузка переменной l,a
   STO l,a  :  запись переменной l,a
   CAL l,a  :  вызов процедуры a на уровне l
   INT 0,a  :  увеличение t-регистра на a
   JMP 0,a  :  переход на a
   JPC 0,a  :  условный переход на a   }
var ch: char;          {последний прочитанный входной символ}
   sym: symbol;        {последний прочитанный символ языка}
   id: alfa;           {последнее прочитанное имя}
   num: integer;       {последнее прочитанное число}
   cc: integer;        {счетчик символов}
   ll: integer;        {длина строки}
   kk, err: integer;
   cx: integer;        {индекс размещения команды}
   line: array [1 .. 81] of char;
   a: alfa;
   code: array [0 .. cxmax] of instruction;
```

```
    word: array [1 . . norw] of alfa;
    wsym: array [1 . . norw] of symbol;
    ssym: array [char] of symbol;
    mnemonic: array [fct] of
                  packed array [1 . . 5] of char;
    declbegsys, statbegsys, facbegsys: symset;
    table: array [0 . . txmax] of
           record name: alfa;
               case kind: object of
               constant: (val: integer);
               variable, procedure: (level, adr: integer)
           end ;
procedure error(n: integer);
begin writeln(' ****', ' ': cc-1, '↑', n:2); err := err+1
end {error} ;
procedure getsym;
    var i,j,k: integer;

    procedure getch;
    begin if cc = ll then
       begin if eof(input) then
                  begin write(' PROGRAM INCOMPLETE'); goto 99
                  end ;
          ll := 0; cc := 0; write(cx: 5, ' ');
          while ¬eoln(input) do
             begin ll := ll+1; read(ch); write(ch); line[ll] := ch
             end ;
          writeln; ll := ll+1; read(line[ll])
       end ;
       cc := cc+1; ch := line[cc]
    end {getch} ;

begin {getsym}
    while ch = ' ' do getch;
    if ch in ['A' . . 'Z'] then
    begin {имя или зарезервированное слово} k := 0;
       repeat if k < al then
          begin k := k+1; a[k] := ch
          end ;
          getch
       until ¬(ch in ['A' . . 'Z','0' . . '9']);
       if k ≥ kk then kk := k else
          repeat a[kk] := ' '; kk := kk-1
          until kk = k;
```

```
        id := a; i := 1; j := norw;
        repeat k := (i+j) div 2;
            if id ≤ word[k] then j := k-1;
            if id ≥ word[k] then i := k+1
        until i > j;
        if i-1 > j then sym := wsym[k] else sym := ident
    end else
    if ch in ['0' .. '9'] then
    begin {число} k := 0; num := 0; sym := number;
        repeat num := 10*num + (ord(ch)-ord('0'));
            k := k+1; getch
        until ¬(ch in ['0' .. '9']);
        if k > nmax then error (30)
    end else
    if ch = ':' then
    begin getch;
        if ch = '=' then
        begin sym := becomes; getch
        end else sym := nul;
    end else
    begin sym := ssym[ch]; getch
    end
end {getsym} ;

procedure gen(x: fct; y,z: integer);
begin if cx > cxmax then
            begin write(' PROGRAM TOO LONG'); goto 99
            end ;
    with code[cx] do
        begin f := x; l := y; a := z
        end ;
    cx := cx + 1
end {gen} ;

procedure test(s1,s2: symset; n: integer);
begin if ¬(sym in s1) then
        begin error(n); s1 := s1 + s2;
            while ¬(sym in s1) do getsym
        end
end {test} ;
procedure block(lev,tx: integer; fsys: symset);
    var dx: integer;      {индекс размещения данных}
        tx0: integer;     {начальный индекс таблицы}
        cx0: integer;     {начальный индекс программы}
```

```
    procedure enter(k: object);
    begin {запись объекта в таблицу}
        tx := tx + 1;
        with table[tx] do
        begin name := id; kind := k;
            case k of
            constant: begin if num > amax then
                                    begin error (30); num := 0 end ;
                        val := num
                      end ;
            variable: begin level := lev; adr := dx; dx := dx+1;
                      end ;
            procedure: level := lev
            end
        end
    end {enter} ;
    function position(id: alfa): integer;
        var i: integer;
    begin {поиск имени id в таблице}
        table[0].name := id; i := tx;
        while table[i].name ≠ id do i := i-1;
        position := i
    end {position} ;
    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym in [eql ,becomes] then
            begin if sym = becomes then error(1);
                getsym;
                if sym = number then
                    begin enter(constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
    end {constdeclaration} ;
procedure vardeclaration;
begin if sym = ident then
        begin enter(variable); getsym
        end else error (4)
end {vardeclaration} ;
```

```
procedure listcode;
    var i: integer;
begin
    for i := cx0 to cx-1 do
        with code[i] do
            writeln(i, mnemonic[f]:5, l:3, a:5)
end {listcode} ;

procedure statement(fsys: symset);
    var i,cx1,cx2: integer;
    procedure expression(fsys: symset);
        var addop: symbol;
        procedure term(fsys: symset);
            var mulop: symbol;
            procedure factor(fsys: symset);
                var i: integer;
            begin test(facbegsys, fsys, 24);
                while sym in facbegsys do
                begin
                    if sym = ident then
                    begin i := position(id);
                        if i = 0 then error (11) else
                        with table[i] do
                        case kind of
                          constant: gen(lit, 0, val);
                          variable: gen(lod, lev-level, adr);
                          procedure: error (21)
                        end ;
                        getsym
                    end else
                    if sym = number then
                    begin if num > amax then
                            begin error (30); num := 0
                            end ;
                        gen(lit, 0, num); getsym
                    end else
                    if sym = lparen then
                    begin getsym; expression([rparen]+fsys);
                        if sym = rparen then getsym else error (22)
                    end ;
                    test(fsys, [lparen], 23)
                end
            end {factor} ;
```

```
    begin {term} factor(fsys+[times, slash]);
        while sym in [times, slash] do
            begin mulop := sym; getsym; factor(fsys+[times, slash]);
                if mulop = times then gen(opr,0,4) else gen(opr,0,5)
            end
    end {term} ;
begin {expression}
    if sym in [plus, minus] then
        begin addop := sym; getsym; term(fsys+[plus, minus]);
            if addop = minus then gen(opr,0,1)
        end else term(fsys+[plus, minus]);
    while sym in [plus, minus] do
        begin addop := sym; getsym; term(fsys+[plus, minus]);
            if addop = plus then gen(opr,0,2) else gen(opr,0,3)
        end
end {expression} ;
procedure condition(fsys: symset);
    var relop: symbol;
begin
    if sym = oddsym then
    begin getsym; expression(fsys); gen(opr,0,6)
    end else
    begin expression([eql, neq, lss, gtr, leq, geq]+fsys);
        if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
            error (20) else
        begin relop := sym; getsym; expression(fsys);
            case relop of
              eql: gen(opr,0, 8);
              neq: gen(opr,0, 9);
              lss: gen(opr,0,10);
              geq: gen(opr,0,11);
              gtr: gen(opr,0,12);
              leq: gen(opr,0,13);
            end
        end
    end
end {condition} ;
begin {statement}
    if sym = ident then
    begin i := position(id);
        if i = 0 then error (11) else
        if table[i].kind ≠ variable then
```

```
            begin {присваивание переменной} error (12); i := 0
            end ;
        getsym; if sym = becomes then getsym else error (13);
        expression(fsys);
        if i ≠ 0 then
            with table[i] do gen(sto, lev-level, adr)
    end else
    if sym = callsym then
    begin getsym;
        if sym ≠ ident then error (14) else
            begin i := position(id);
                if i = 0 then error (11) else
                with table[i] do
                    if kind = procedure then gen (cal, lev-level, adr)
                    else error (15);
                getsym
            end
    end else
    if sym = ifsym then
    begin getsym; condition([thensym, dosym]+fsys);
        if sym = thensym then getsym else error (16);
        cx1 := cx; gen(jpc,0,0);
        statement(fsys); code[cx1].a := cx
    end else
    if sym = beginsym then
    begin getsym; statement([semicolon, endsym]+fsys);
        while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error (10);
            statement([semicolon, endsym]+fsys)
        end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin cx1 := cx; getsym; condition([dosym]+fsys);
        cx2 := cx; gen(jpc,0,0);
        if sym = dosym then getsym else error (18);
        statement(fsys); gen(jmp,0,cx1); code[cx2].a := cx
    end ;
    test(fsys, [ ], 19)
end {statement} ;
begin {block} dx := 3; tx0 := tx; table[tx] .adr := cx; gen(jmp,0,0);
    if lev > levmax then error (32);
    repeat
```

```
        if sym = constsym then
        begin getsym;
            repeat constdeclaration;
                while sym = comma do
                    begin getsym; constdeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident
        end ;
        if sym = varsym then
        begin getsym;
            repeat vardeclaration;
                while sym = comma do
                    begin getsym; vardeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident;
        end ;
        while sym = procsym do
        begin getsym;
            if sym = ident then
                begin enter(procedure); getsym
                end
            else error (4);
            if sym = semicolon then getsym else error (5);
            block(lev+1,tx,[semicolon]+fsys);
            if sym = semicolon then
                begin getsym; test(statbegsys+[ident, procsym], fsys, 6)
                end
            else error (5)
        end ;
        test(statbegsys+[ident], declbegsys, 7)
    until ¬(sym in declbegsys);
    code[table[tx0].adr].a := cx;
    with table[tx0] do
        begin adr := cx; {начальный адрес команд}
        end ;
    cx0 := cx; gen(int,0,dx);
    statement([semicolon, endsym]+fsys);
    gen(opr,0,0); {return}
    test(fsys, [ ], 8);
    listcode;
end {block} ;
```

```
procedure interpret;
   const stacksize = 500;
   var p,b,t: integer {регистр программы, регистр указателя
                       стека, базовый регистр}
      i: instruction; {регистр команды}
      s: array [1 .. stacksize] of integer; {память для данных}
   function base(l: integer): integer;
      var b1: integer;
   begin b1 := b; {поиск базы l уровнями ниже}
      while l > 0 do
         begin b1 := s[b1]; l := l-1
         end ;
      base := b1
   end {base} ;

begin writeln(' START PL/0');
   t := 0; b := 1; p := 0;
   s[1] := 0; s[2] := 0; s[3] := 0;
   repeat i := code[p]; p := p+1;
      with i do
      case f of
      lit: begin t := t+1; s[t] := a
         end ;
      opr: case a of      {операция}
         0: begin {возврат}
               t := b-1; p := s[t+3]; b := s[t+2];
            end ;
         1: s[t] := -s[t];
         2: begin t := t-1; s[t] := s[t] + s[t+1]
            end ;
         3: begin t := t-1; s[t] := s[t] - s[t+1]
            end ;
         4: begin t := t-1; s[t] := s[t] * s[t+1]
            end ;
         5: begin t := t-1; s[t] := s[t] div s[t+1]
            end ;
         6: s[t] := ord(odd(s[t]));
         8: begin t := t-1; s[t] := ord(s[t]=s[t+1])
            end ;
         9: begin t := t-1; s[t] := ord(s[t]≠s[t+1])
            end ;
        10: begin t := t-1; s[t] := ord(s[t]<s[t+1])
            end ;
```

```
            11: begin t := t-1; s[t] := ord(s[t]≥s[t+1])
                end ;
            12: begin t := t-1; s[t] := ord(s[t]>s[t+1])
                end ;
            13: begin t := t-1; s[t] := ord(s[t]≤s[t+1])
                end ;
          end ;
      lod: begin t := t+1; s[t] := s[base(l)+a]
          end ;
      sto: begin s[base(l)+a] := s[t]; writeln(s[t]); t := t-1
          end ;
      cal: begin {формирование отметки в новом блоке}
              s[t+1] := base(l); s[t+2] := b; s[t+3] := p;
              b := t+1; p := a
           end ;
      int: t := t+a;
      jmp: p := a;
      jpc: begin if s[t] = 0 then p := a; t := t-1
          end
        end {with, case}
    until p = 0;
    write(' END PL/0');
end {interpret} ;
begin {основная программа}
    for ch := 'A' to ';' do ssym[ch] := nul;
    word[ 1] := 'BEGIN ';   word[ 2] := 'CALL      ';
    word[ 3] := 'CONST';    word[ 4] := 'DO        ';
    word[ 5] := 'END   ';   word[ 6] := 'IF        ';
    word[ 7] := 'ODD   ';   word[ 8] := 'PROCEDURE';
    word[ 9] := 'THEN  ';   word[10] := 'VAR       ';
    word[11] := 'WHILE ';
    wsym[ 1] := beginsym; wsym[ 2] := callsym;
    wsym[ 3] := constsym; wsym[ 4] := dosym;
    wsym[ 5] := endsym;   wsym[ 6] := ifsym;
    wsym[ 7] := oddsym;   wsym[ 8] := procsym;
    wsym[ 9] := thensym;  wsym[10] := varsym;
    wsym[11] := whilesym;
    ssym['+'] := plus;    ssym['-'] := minus;
    ssym['*'] := times;   ssym['/'] := slash;
    ssym['('] := lparen;  ssym[')'] := rparen;
    ssym['='] := eql;     ssym[','] := comma;
    ssym['.'] := period;  ssym['≠'] := neq;
    ssym['<'] := lss;     ssym['>'] := gtr;
    ssym['≤'] := leq;     ssym['≥'] := geq;
```

```
    ssym[';'] := semicolon;
    mnemonic[lit] := 'LIT '; mnemonic[opr] := 'OPR';
    mnemonic[lod] := 'LOD'; mnemonic[sto] := 'STO';
    mnemonic[cal] := 'CAL'; mnemonic[int] := 'INT ';
    mnemonic[jmp] := 'JMP'; mnemonic[jpc] := 'JPC';
    declbegsys := [constsym, varsym, procsym];
    statbegsys := [beginsym, callsym, ifsym, whilesym];
    facbegsys := [ident, number, lparen];
    page(output); err := 0
    cc := 0; cx := 0; ll := 0; ch := ' '; kk := al; getsym;
    block(0,0,[period]+declbegsys+statbegsys);
    if sym ≠ period then error (9);
    if err = 0 then interpret else write(' ERRORS IN PL/0 PROGRAM');
99: writeln
end .
```

Программа 5.6. Транслятор для ПЛ/0

УПРАЖНЕНИЯ

5.1. Рассмотрим следующий синтаксис:

$$
\begin{aligned}
&S ::= A\\
&A ::= B \mid \textbf{if } A \textbf{ then } A \textbf{ else } A\\
&B ::= C \mid B + C \mid + C\\
&C ::= D \mid C * D \mid * D\\
&D ::= x \mid (A) \mid - D
\end{aligned}
$$

Каковы здесь терминальные и нетерминальные символы? Определите множества самых левых и внешних символов $L(X)$ и $F(X)$ для каждого нетерминального символа X. Постройте последовательность шагов грамматического разбора для следующих предложений:

$$
\begin{aligned}
&x + x\\
&(x + x) * (+ - x)\\
&(x * - + x)\\
&\textbf{if } x + x \textbf{ then } x * x \textbf{ else } - x\\
&\textbf{if } x \textbf{ then if } - x \textbf{ then } x \textbf{ else } x + x \textbf{ else } x * x\\
&\textbf{if } - x \textbf{ then } x \textbf{ else if } x \textbf{ then } x + x \textbf{ else } x
\end{aligned}
$$

5.2. Удовлетворяет ли грамматика упр. 5.1 ограничениям 1 и 2 для нисходящего грамматического разбора с просмотром вперед на один символ? Если нет, найдите эквивалентный синтаксис, который удовлетворяет этим ограничениям. Изобразите этот синтаксис в виде синтаксического графа и структуры данных, используемой в программе 5.3.

5.3. Выполните упр. 5.2 для следующего синтаксиса:

$S ::= A$
$A ::= B \mid$ **if** C **then** $A \mid$ **if** C **then** A **else** A
$B ::= D = C$
$C ::=$ **if** C **then** C **else** $C \mid D$

Примечание. Если это необходимо, вы можете удалить или заменить какую-либо конструкцию, чтобы сделать применимым односимвольный нисходящий грамматический разбор.

5.4. Рассмотрите нисходящий грамматический разбор для следующего синтаксиса:

$S ::= A$
$A ::= B + A \mid DC$
$B ::= D \mid D * B$
$D ::= x \mid (C)$
$C ::= +x \mid -x$

На сколько символов вперед нужно смотреть, чтобы анализировать предложения согласно этому синтаксису?

5.5. Преобразуйте описание ПЛ/0 (рис. 5.4) в эквивалентное множество порождающих правил БНФ.

5.6. Напишите программу, которая определяет множества начальных и внешних символов $L(S)$ и $F(S)$ для каждого нетерминального символа S в заданном множестве порождающих правил.

Примечание. Используйте часть программы 5.3 для построения внутреннего представления синтаксиса в виде структуры данных. Затем работайте с этой связанной структурой.

5.7. Расширьте язык ПЛ/0 и его транслятор, включив в него следующие операторы:

(a) Условный оператор вида
⟨*оператор*⟩ ::= **if** ⟨*условие*⟩ **then** ⟨*оператор*⟩ **else** ⟨*оператор*⟩

(b) Оператор цикла вида
⟨оператор⟩ ::= **repeat** ⟨*оператор*⟩ {; ⟨*оператор*⟩} **until** ⟨*условие*⟩

Есть ли какие-либо особые трудности, которые могли бы привести к изменению формы или интерпретации указанных операторов? Вводить какие-то дополнительные команды в систему машины ПЛ/0 вы не должны.

5.8. Расширьте язык ПЛ/0 и его транслятор, добавив в него параметры процедур. Рассмотрите два возможных решения и выберите одно из них для реализации:

(a) *Параметры-значения.* Фактические параметры обращения являются выражениями, значения которых присваиваются локальным переменным, представляющим формальные параметры, заданные в заголовке процедуры.

(b) *Параметры-переменные.* Фактические параметры являются переменными. При вызове они подставляются на место формальных

параметров. Параметры-переменные реализуются с помощью передачи адресов фактических параметров в места, выделенные для формальных параметров. Доступ к фактическим параметрам осуществляется косвенно через переданный адрес. Следовательно, параметры-переменные обеспечивают доступ к переменным, определенным вне процедур, и поэтому правила области определения можно изменить следующим образом: в каждой процедуре непосредственно доступны только локальные переменные; доступ к нелокальным переменным возможен только через параметры.

5.9. Расширьте язык ПЛ/0 и его транслятор, добавив переменные-массивы. Пусть диапазон индексов такой переменной *a* указывается в ее описании как

var *a*(*low*: *high*)

5.10. Модифицируйте транслятор ПЛ/0, чтобы он формировал рабочую программу для имеющейся у вас вычислительной машины.

Примечание. Формируйте программу на языке ассемблера, чтобы избежать проблем, связанных с загрузкой. На первом этапе не пытайтесь оптимизировать рабочую программу, например использовать регистры. Возможная оптимизация должна включаться лишь на четвертом этапе уточнения транслятора.

5.11. Расширьте программу 5.5 в программу, называемую *prettyprint* («красивая печать»). Задача этой программы — читать тексты ПЛ/0 и печатать их в форме, которая естественным образом отражает структуру текста при помощи соответствующего разделения на строки и абзацы. Вначале аккуратно определите правила деления на строки и абзацы, основанные на синтаксической структуре ПЛ/0; затем реализуйте их, вставив операторы печати в программу 5.5. (Разумеется, из сканера нужно убрать операторы печати.)

ЛИТЕРАТУРА

5.1. AMMANN U. The Method of Structured Programming Applied to the Development of a Compiler. — *International Computing Symposium 1973*, A Günther et al., eds., Amsterdam: North-Holland Publishing Co., 1974, 93—99.

5.2. COHEN D. J., GOTLIEB C. C. A List Structure Form of Grammars for Syntactic Analysis. — *Comp. Surveys*, **2**, No. 1, 1970, 65—82.

5.3. FLOYD R. W. The Syntax of Programming Languages — A Survey. — *IEEE Trans.*, EC-13, 1964, 346—353.

5.4. *GRIES D.* Compiler Construction for Digital Computers. — New-York: Wiley, 1971. [Имеется перевод: ГРИС Д. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.]

5.5. KNUTH D. E. Top-down Syntax Analysis. — *Acta Informatica*, **1**, No. 2, 1971, 79—110.

5.6. LEWIS P. M., STEARNS R. E. Syntax-directed Transduction, *J. ACM*, **15**, No. 3, 1968, 465—488.

5.7. NAUR P. Report on the Algorithmic Language ALGOL 60. — *ACM*, **6**, No. 1, 1963, 1—17.

5.8. SCHORRE D. V. META II, A Syntax-oriented Compiler Writing Language. — *Proc. ACM Natl. Conf.*, **19**, 1964, D 1 3. 1—11.

5.9. WIRTH N. The Design of a PASCAL Compiler. — *Software — Practice and Experience*, **1**. No. 4, 1971, 309—333.

ПРИЛОЖЕНИЕ А

МНОЖЕСТВО СИМВОЛОВ ASCII

y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	gs	-	=	M	]	m	}
14	so	rs	.	>	N	↑	n	~
15	si	us	/	?	O	_	o	del

Порядковый номер символа определяется по его координатам в таблице как

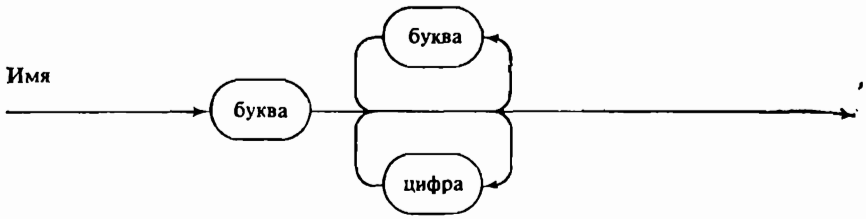
$$ord(ch) = 16 * x + y.$$

Символы с порядковыми номерами от 0 до 31 и 127 — так называемые *управляющие символы*, используемые для передачи данных и управления устройствами. Символ с порядковым номером 32 есть пробел.

ПРИЛОЖЕНИЕ В

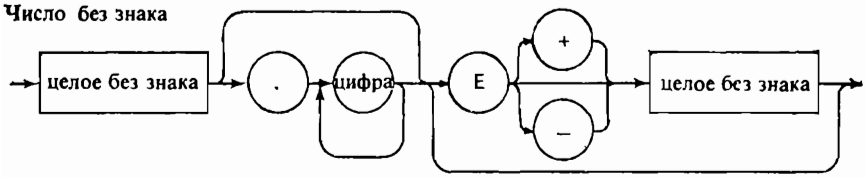
СИНТАКСИЧЕСКИЕ ДИАГРАММЫ ПАСКАЛЯ

Имя

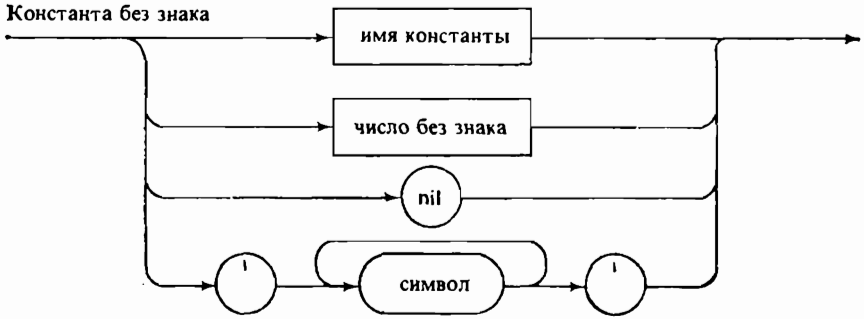


Целое без знака

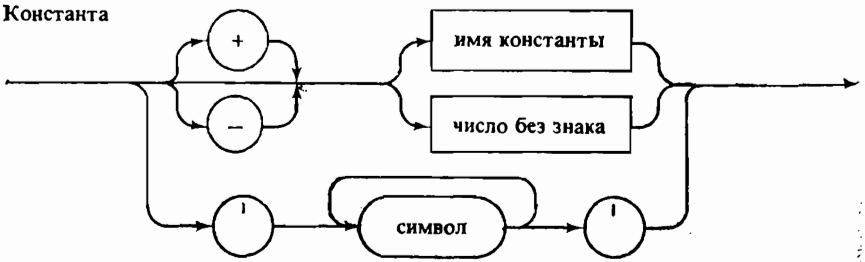
Число без знака



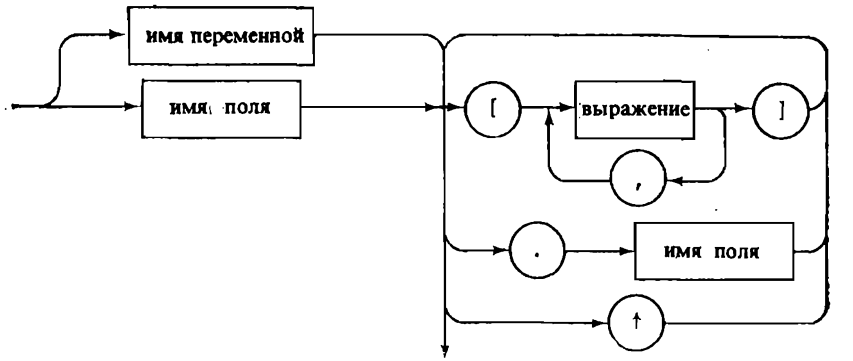
Константа без знака



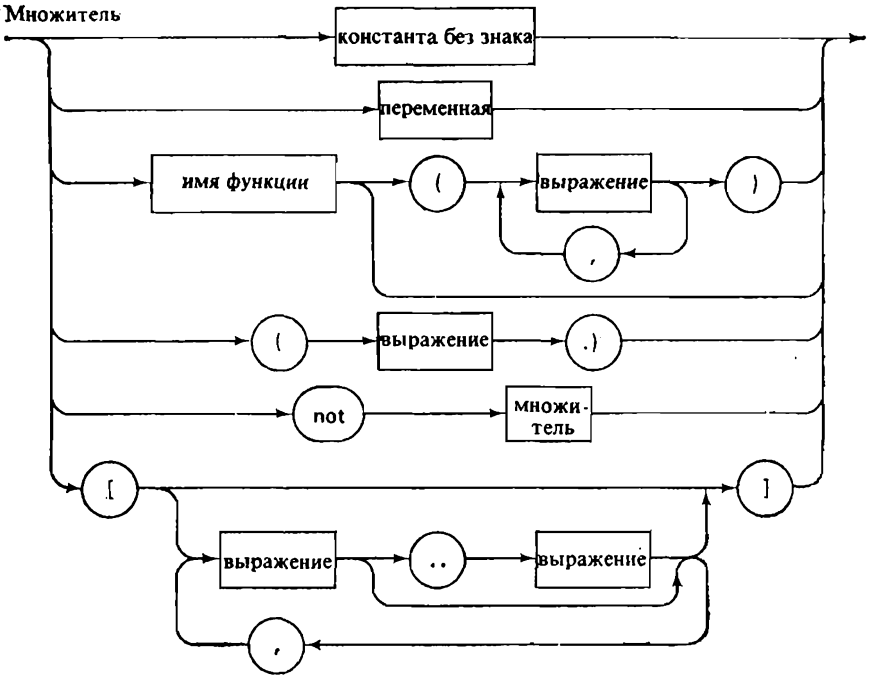
Константа



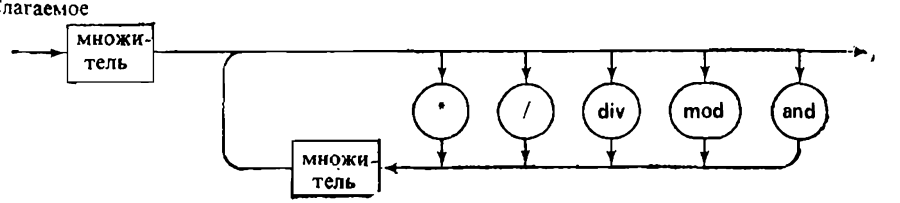
Переменная



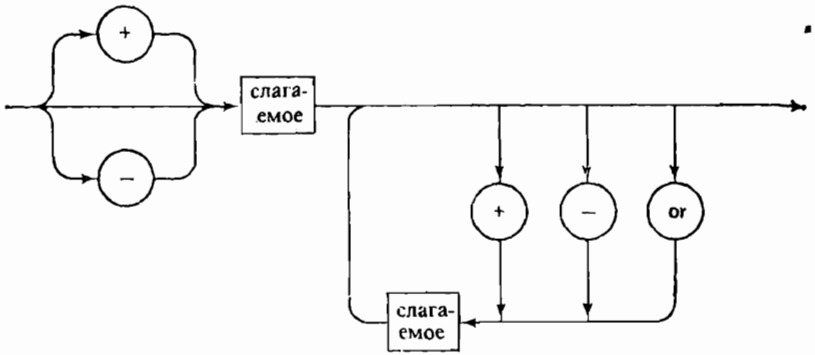
Множитель



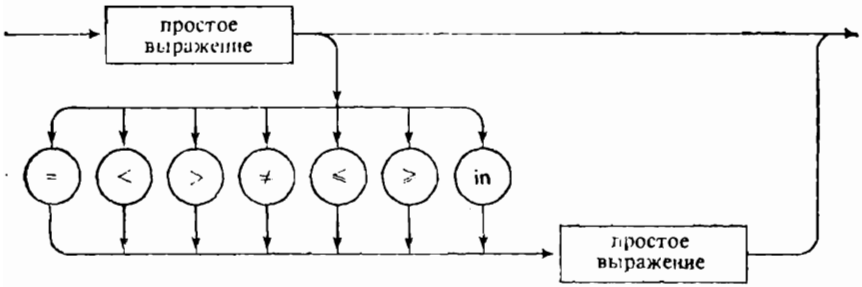
Слагаемое



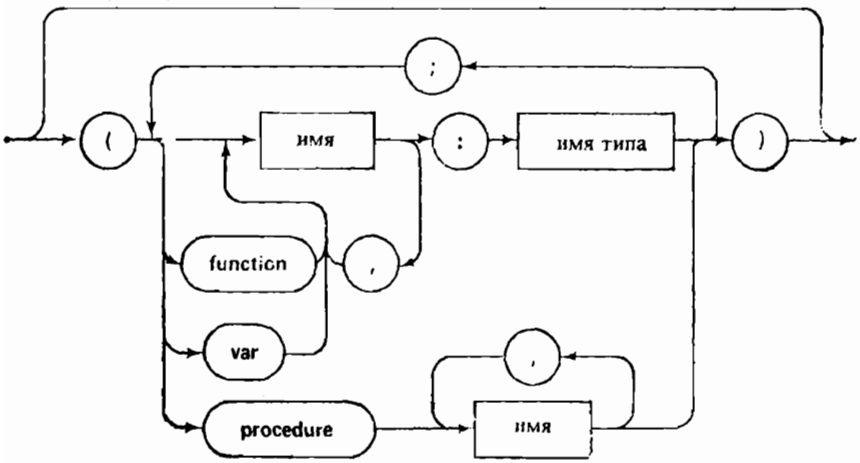
Простое выражение

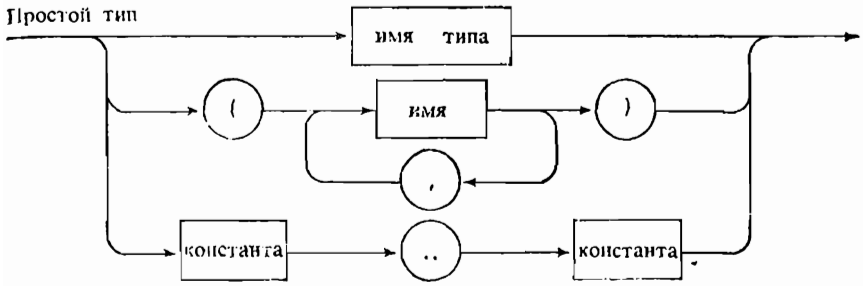


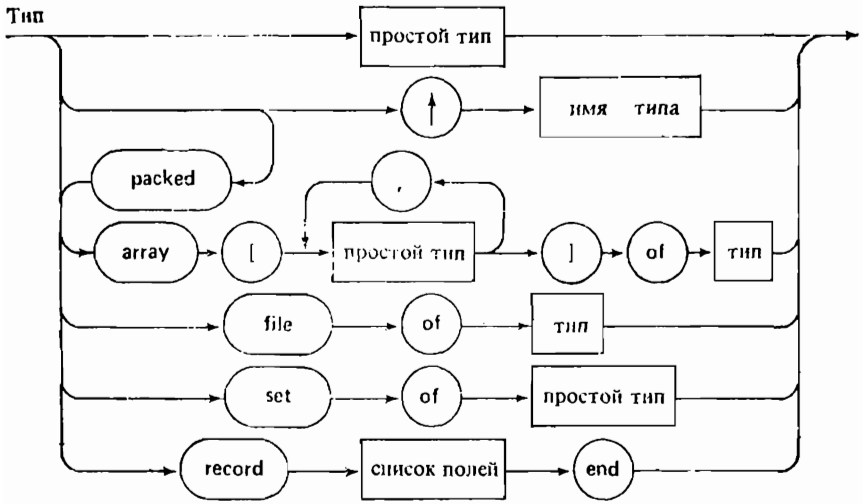
Выражение

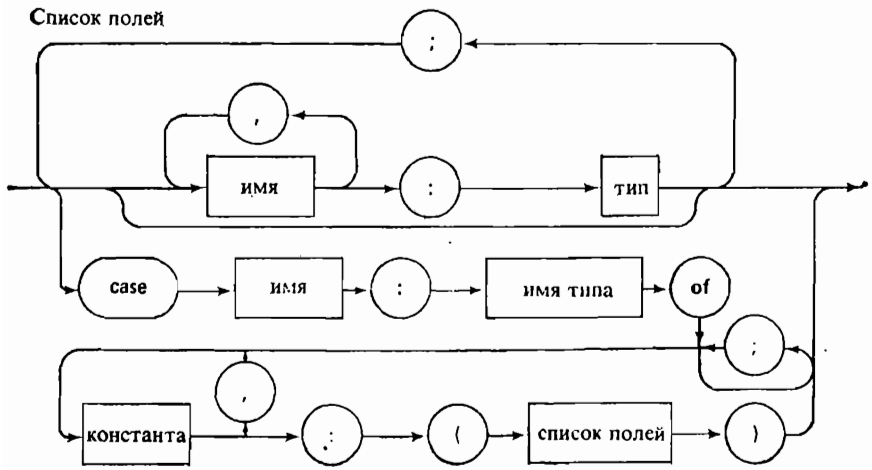


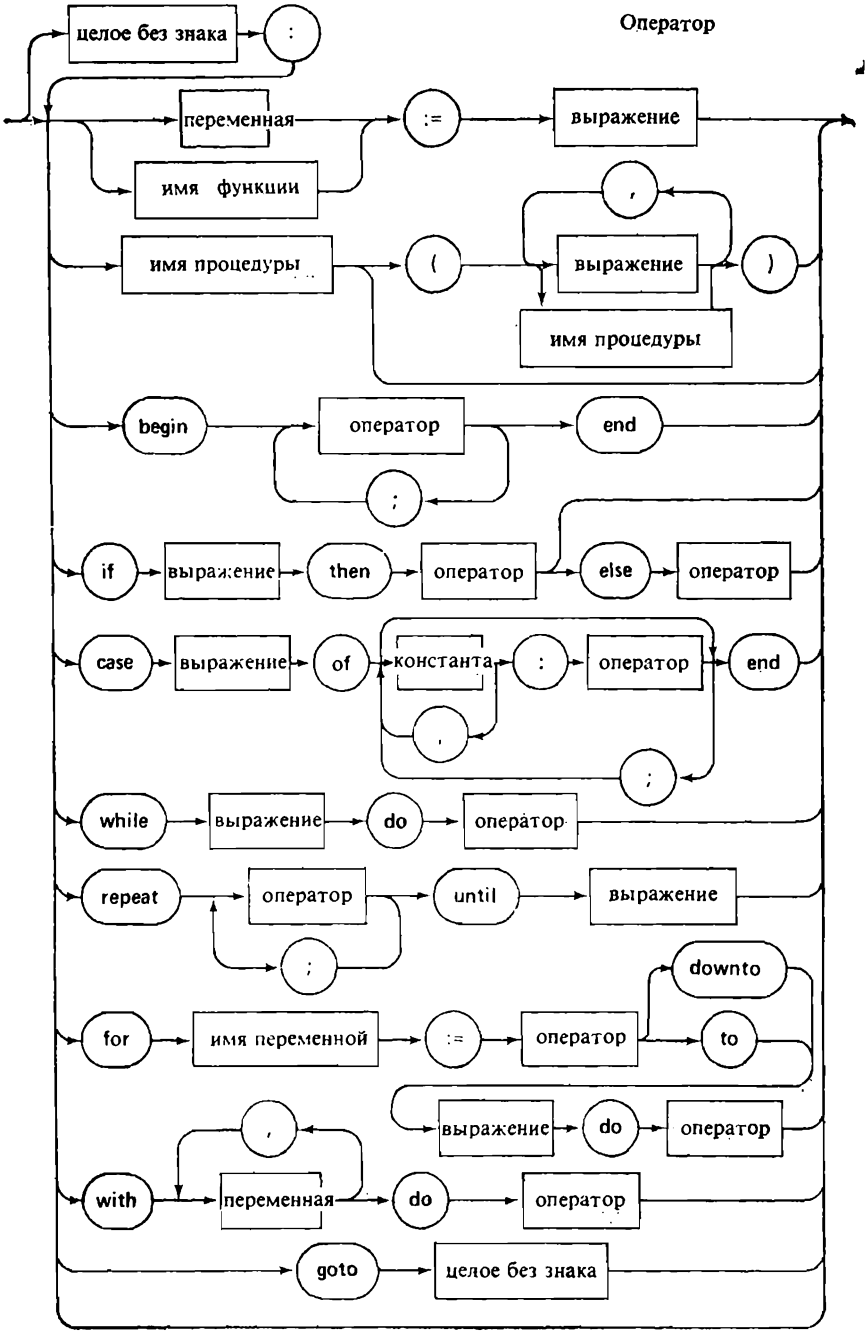
Список параметров



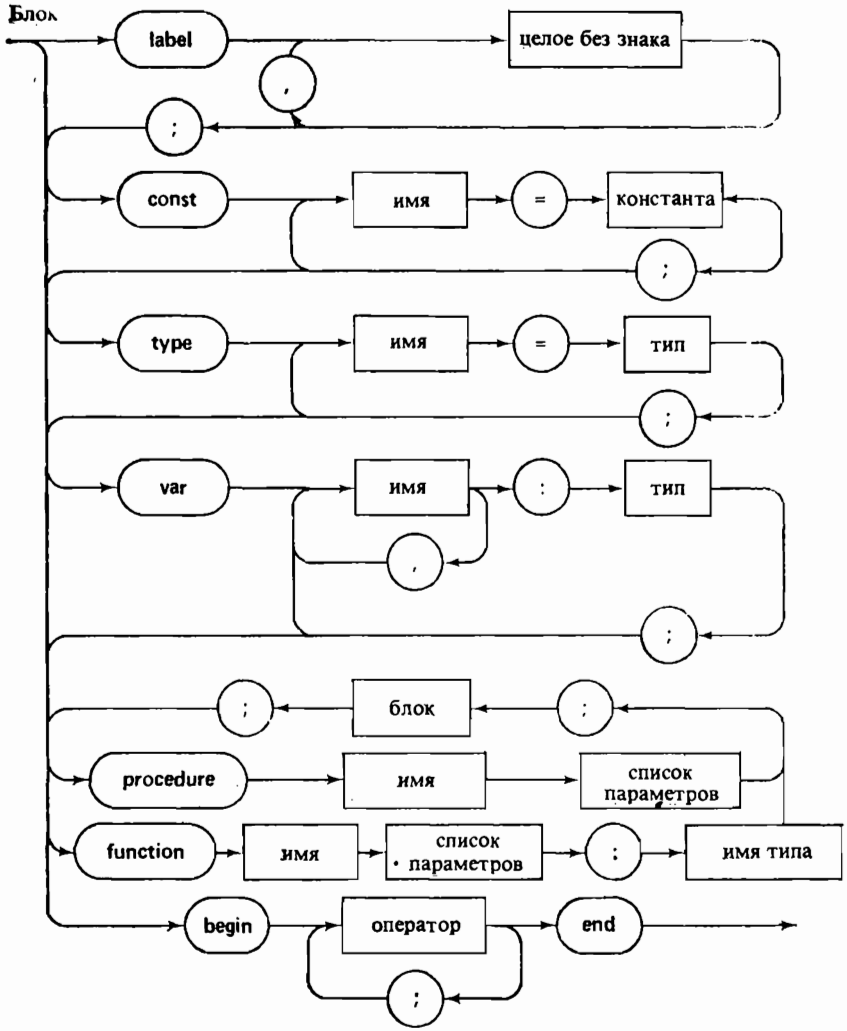

Простой тип
имя типа
(
имя
)
,
константа
..
константа


Тип
простой тип
↑
имя типа
packed
,
array
[
простой тип
]
of
тип
file
of
тип
set
of
простой тип
record
список полей
end

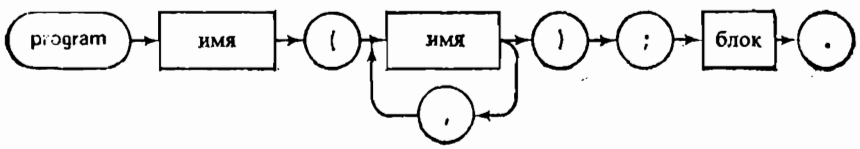

Список полей
;
,
имя
:
тип
case
имя
:
имя типа
of
;
,
константа
:
(
список полей
)


Оператор
целое без знака
:
переменная
:=
выражение
имя функции
,
имя процедуры
(
выражение
)
имя процедуры
begin
оператор
end
;
if
выражение
then
оператор
else
оператор
case
выражение
of
константа
:
оператор
end
,
;
while
выражение
do
оператор
repeat
оператор
until
выражение
;
downto
for
имя переменной
:=
оператор
to
выражение
do
оператор
,
with
переменная
do
оператор
goto
целое без знака

Блок



Программа



УКАЗАТЕЛЬ ПРОГРАММ

ОГЛАВЛЕНИЕ