

ПРОСТОЕ И СЛОЖНОЕ В ПРОГРАММИРОВАНИИ



АКАДЕМИЯ НАУК СССР

СЕРИЯ «КИБЕРНЕТИКА —
НЕОГРАНИЧЕННЫЕ ВОЗМОЖНОСТИ
И ВОЗМОЖНЫЕ ОГРАНИЧЕНИЯ»

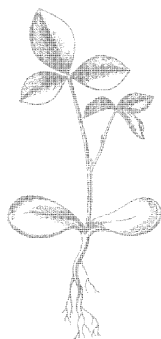
Основана в 1963 г.

ПРОСТОЕ И СЛОЖНОЕ В ПРОГРАМ- МИРОВАНИИ



МОСКВА «НАУКА»

1988



Scan AAW

ББК 32.973.2

П 82

УДК 519.6

Редакционная коллегия:

академик И. М. МАКАРОВ (председатель),

академик В. Г. АФАНАСЬЕВ,

доктор философских наук Б. В. БИРЮКОВ,

академик С. В. ЕМЕЛЬЯНОВ,

академик Н. Н. МОИСЕЕВ,

академик Б. Н. НАУМОВ,

В. Д. ПЕКЕЛИС,

доктор технических наук Д. А. ПОСПЕЛОВ (редактор-составитель),

доктор технических наук И. С. УКОЛОВ,

доктор физико-математических наук В. В. ЩЕННИКОВ

Ответственный секретарь редколлегии

кандидат философских наук С. Н. ГОНШОРЕК

Автор предисловия Е. П. ВЕЛИХОВ

Рецензенты:

академик А. П. ЕРШОВ,

доктор физико-математических наук В. А. УСПЕНСКИЙ

П82 Простое и сложное в программировании/Авт. предисл. Е. П. Велихов.— М.: Наука, 1988.— 176с., ил.— (Сер. «Кибернетика — неограниченные возможности и возможные ограничения»).

ISBN 5—02—006595—1

Книга является популярным введением в методологию программирования, помогающим понять специфику этой деятельности и раскрыть некоторые ее «секреты». Для широкого круга читателей.

П $\frac{1502000000-268}{054(02)-88}$ 44—48НП

ББК 32.973.2

Серия «Кибернетика — неограниченные возможности и возможные ограничения» удостоена диплома I степени на Всесоюзном конкурсе общества «Знание» в 1985 г.

ISBN 5—02—006595—1 © Издательство «Наука», 1988

Предисловие

Сейчас нет необходимости долго говорить о том, какую роль играет развитие индустрии информатики в решении стоящих перед нами задач. Как и всякая индустрия, индустрия информатики основана на определенной технологии (ее часто называют новой информационной технологией), а в основе любой технологии лежит наука — в данном случае информатика.

Предлагаемая вниманию читателей книжка представляет собой популярное изложение важнейших идей этой науки. Как показывает опыт, они довольно просты и вполне доступны подготовленным школьникам: многие главы сборника основаны на материалах занятий, проводившихся в классе с углубленным изучением математики (школа № 57 г. Москвы). Авторы адресуют свою книгу прежде всего программистам будущего — школьникам старших классов. Думаю, что она найдет намного более широкий круг читателей — от людей самых разных профессий, захотевших разобраться, «чем же эти программисты в конце концов занимаются», до профессионалов в области создания программного обеспечения и обучения программированию.

Остановимся более подробно на ее содержании. В главе 1 «Команды исполнителю» читатели знакомятся с тремя основными программными конструкциями (последовательное выполнение, выбор, повторение) на простом модельном примере — управление роботом,двигающимся в лабиринте. Одновременно демонстрируется использование спецификаций при разработке программ. Глава 2 «Коза, капуста и другие с точки зрения программиста» продолжает знакомство с этими конструкциями на других примерах. Глава 3 «Путник снова в лабиринте» возвращает к реалиям главы 1, но посвящена уже более трудной задаче. В ней читатели знакомятся с технологией построения программы сверху — вниз, называемой также методом последовательного уточнения. Она показывает, что уже в такой простой модельной ситуации возможно формулировать и решать содержательные программистские задачи.

Заметим здесь, что к идее использовать при обучении программированию алгоритмы, работающие в геометрическом мире, независимо пришли исследователи и педагоги в разных странах. Этот принцип широко используется в языке Лого, предложенном С. Пейпертом в США. В нашей стране такие идеи реализованы А. Г. Кушниренко в его лекциях на механико-математическом факультете МГУ, послуживших источником идей для многих глав настоящего сборника.

Глава 4 «Доказательства в программировании» содержит явные формулировки тех правил рассуждений о свойствах программ, которые в неявном виде уже применялись. Ключевым здесь является понятие инварианта цикла. Его иллюстрации на классическом примере алгоритма нахождения наибольшего общего делителя двух натуральных чисел посвящена глава 5 «Алгоритм Евклида».

Важную часть науки программирования составляет так называемая теория сложности вычислений. Грубо говоря, ее предметом является оценка ресурсов, необходимых для решения тех или иных задач. Простейшим вопросам этой теории посвящена глава 6 «Кто тяжелее, или Нижние и верхние оценки для задачи сортировки», в которой формулируется и решается задача о минимальном числе сравнений, необходимом для упорядочения заданного числа предметов.

Глава 7 «Сколько веревочке ни виться, или Почему программы заканчивают работу» посвящена математическим средствам, используемым при доказательстве утверждений о том, что данная программа рано или поздно закончит свою работу.

В главе 8 авторы возвращаются к задаче сортировки, уже рассматривавшейся в главе 6, и излагают некоторые конкретные алгоритмы решения такого рода задач (алгоритм сортировки слиянием и алгоритм быстрой сортировки).

Часто встречаются задачи, в которых нужно перебрать все варианты из некоторого заданного множества возможных вариантов. Общая схема решения таких задач рассматривается в главе 9 «Могут ли восемь ферзей не бить друг друга, или Обход дерева». Как видно из названия, применение этой схемы иллюстрируется на примере известной шахматной головоломки (в которой требуется расставить восемь ферзей на шахматной доске так, чтобы они не били друг друга).

Одно из важных средств современного программирования — так называемое рекурсивное описание алгоритма, в котором решение интересующей нас задачи описывается, так сказать, через «самого себя». Такие описания и возникающие в связи с ними проблемы рассматриваются в главе 10 «Можно ли поднять себя за волосы, или Рекурсия». Следующая глава — глава 11 «От рекурсивного определения к программе» показывает, что в некоторых случаях можно на основе рекурсивного описания построить обычную, «нерекурсивную» программу. Эта глава посвящена простейшим примерам такого рода; более сложные методы — «замена рекурсии использованием стека» — рассмотрены в главе 12 «Ханойские башни» на примере одноименной головоломки.

Хотя научно-технические расчеты в какой-то степени потеряли свою главенствующую роль среди всех применений ЭВМ, до сих пор они остаются одной из важнейших областей их использования. Некоторые возникающие при этом задачи рассматриваются на простом модельном примере извлечения квадратного корня в главе 13 «Вычисления и вычислительные машины».

Глава 14 снова возвращает нас к теории сложности вычислений. В ней рассказано о классе так называемых переборных (или NP-полных) задач и на простейшем примере проиллюстрирован метод доказательства NP-полноты конкретных задач. Кроме того, в этой главе рассказано о методе динамического программирования, позволяющем строить эффективные алгоритмы решения некоторых задач (на примере задачи о разрезании выпуклого многоугольника на треугольники).

Следующие две главы посвящены играм. В первой из них (глава 15 «Игры, игры, игры...») доказывается теорема о том, что в любой игре определенного класса у одного из игроков имеется выигрышная стратегия. Эта теорема, однако, не дает никакого явного описания такой стратегии. Оказывается, в некоторых случаях такое явное описание можно дать в терминах «сохранения инвариантного отношения». Такие примеры приводятся в главе 16 «Снова об играх».

В главе 17 «Гениальный режиссер и его жертва» рассказывается история, которая метафорически излагает доказательство одной из самых фундаментальных теорем теории алгоритмов — теоремы о неразрешимости так называемой проблемы остановки.

Следующие главы отражают мнения авторов о различных аспектах окружающей нас «компьютерной реальности». Зачастую мнения эти весьма спорны, но, как мне кажется, заслуживают обсуждения и будут интересны читателям.

Глава 18 «Редактор текстов, или Зачем компьютер грамотному» описывает одно из наиболее распространенных применений ЭВМ (особенно персональных) — обработку текстов. Глава 19 «Исполнитель-черепаха, или Язык Лого» рассказывает об одном из учебных языков программирования, нашедшем применение, в частности, в школах Болгарии. Глава 20 «Игра в животных, или Искусственный интеллект» рассказывает о том, как некоторые очень простые программы могут создать у неискушенного человека иллюзию «интеллектуального» поведения ЭВМ. Глава 21 «Программистские басни» воспроизводит два весьма поучительных эссе Э. Дейкстры. Социальные аспекты применения ЭВМ рассматриваются в главе 22 «Компьютеры и общество».

Книга завершается двумя главами, предназначенными главным образом для начинающих программистов. Первая из них (глава 23) содержит информацию о языке Паскаль. Вторая (глава 24) представляет собой список литературы по программированию, снабженный краткими рецензиями на каждую из упоминаемых книг.

То, что книга рассчитана в том числе и на школьников, не означает, что читать ее легко — не надо недооценивать возможности подготовленных школьников! Полагаю тем не менее, что многие читатели найдут в ней что-то для себя интересное.

В последнее время выпускается большое количество самой разной литературы по программированию, в том числе и популярной. Тем не менее, пожалуй, сейчас почти нет книг, которые, с одной стороны, не ограничивались бы рассказами типа «что умеют современные ЭВМ», и с другой — не перегружались бы техническими деталями устройства каких-то конкретных языков программирования, операционных систем и т. п. Мне кажется, что любую попытку заполнить возникающую «экологическую нишу» (будет эта попытка удачной или нет — судить читателям) можно только приветствовать.

Академик *Е. П. Велихов*

ГЛАВА 1

КОМАНДЫ ИСПОЛНИТЕЛЮ

Перед вами первая глава нашей книжки. Добраться до последней не так-то просто. Однако, если вам это удастся, вы познакомитесь с важными конструкциями программирования и решениями программистских задач, среди которых есть и довольно сложные.

Работа вычислительной машины состоит в выполнении команд. Ситуация, в которой имеется исполнитель, воспринимающий и выполняющий команды, возможна не только в вычислительной технике. На первых порах для иллюстрации важных идей программирования мы будем представлять себе исполнителя как человека. Пример — повар, получающий команды «добавить одно яйцо» или «жарить 15 минут». Помимо команд, исполнитель может осуществлять проверки. Например, повар может проверить, «сварилась ли капуста», и, если сварилась, «добавить одно яйцо». Для данного исполнителя фиксирован набор исходных команд и проверок, которые, как мы предполагаем, он умеет выполнять. Эти исходные команды и проверки могут использоваться как составные части при построении более сложных команд и проверок. Работа программиста — программирование — как раз и состоит в придумывании описаний (их называют программами) для этих более сложных команд и проверок.

Три способа образования команд

1. Последовательное выполнение. Команда «сделать_чай_сладким» требует последовательного выполнения команд

| положить_сахар
| размешать

Заметим, что порядок выполнения команд существен.

2. Выбор. В ходе работы исполнителю на основании результата проверок приходится выбирать для испол-

нения ту или иную команду. Так, например, команда «поставить оценку» описывается так:

выбор

. при ответ_отличный	делай	поставить_5
. при ответ_хороший	делай	поставить_4
. при ответ_средний	делай	поставить_3
. при ответ_плохой	делай	поставить_2

конец выбора

Вариантов может быть и больше и меньше четырех. Исполнителю разрешается выполнять любую из команд, стоящих после проверки, дающей ответ «да». Например, в команде «положить y равным модулю x », описываемой так:

выбор

. при $x \geq 0$	делай	положить y равным x
. при $x \leq 0$	делай	положить y равным $-x$

конец выбора

в случае $x = 0$ обе проверки дают ответ «да» и исполнитель может выбрать любое из действий.

При исполнении команды выбора требуется, чтобы хотя бы одна из проверок давала ответ «да». В противном случае исполнитель не сможет выполнить эту команду.

В команде определения четности числа

выбор

. при последняя цифра 0, 2, 4, 6, 8	делай
. сообщать, что число четно	
. при последняя цифра 1, 3, 5, 7, 9	делай
. сообщать, что число нечетно	

конец выбора

проверки являются взаимно исключающими: если одна дает ответ «да», то другая — «нет», и наоборот. Чтобы подчеркнуть это, команду выбора можно записать так:

если последняя цифра 0, 2, 4, 6, 8 **то**

. сообщать, что число четно

. **иначе**

. сообщать, что число нечетно

конец_если

3. Повторение. Часто нам бывает нужно, чтобы исполнитель выполнил какую-то команду не один, а несколько раз подряд. Выполнение команды должно повторяться до тех пор, пока не будет достигнут определенный результат. При этом перед каждым повторе-

нием выполняется проверка, дающая ответ «да» (пока результат не достигнут, надо продолжать повторение).

Например, команда «провести воспитательную работу» может быть описана так:

```
пока нарушитель_не_раскаялся_повторять
.   провести_беседу_с_нарушителем
конец_пока
```

Проверкой здесь является «нарушитель_не_раскаялся», а повторяемой командой «провести_беседу_с_нарушителем». Эта команда будет выполняться столько раз, сколько нужно, чтобы проверка перестала давать ответ «да», т. е. чтобы нарушитель раскаялся.

Специально отметим следующее. 1. Если нарушитель раскаялся с самого начала, беседа не проводится ни разу. 2. Выполнение команды «провести воспитательную работу» может не закончиться (как говорят, исполнитель «зациклился»). 3. Если эта команда кончит работу, то нарушитель окажется раскаявшимся. 4. Проверка «нарушитель не раскаялся» выполняется не в ходе беседы, а между беседами. (Нарушитель, раскаявшийся в середине беседы, выслушивает ее до конца.)

У п р а ж н е н и е. Пусть исполнитель имеет семилитровое ведро и может выполнять команды «вылить_воду_из_ведра», «долить_в_ведро_литр_воды» и проверку «ведро_не_полно». Что произойдет, если выполнить команды

```
вылить_воду_из_ведра
пока ведро_не_полно_повторять
.   долить_в_ведро_литр_воды
.   долить_в_ведро_литр_воды
конец_пока
```

(Ответ. Ведро наполнится, а один литр прольется на пол.)

Образовывая новые команды, мы наряду с уже имеющимися названиями команд и проверок использовали служебные слова «выбор», «конец_выбора», «пока», «повторять» и т. д. Эти слова будут использоваться и дальше.

Исполнитель «Путник»

В нескольких ближайших задачах мы будем использовать одного и того же исполнителя. Он называется Путник, и сейчас мы его опишем.

На листе бумаги в клетку изображен Прямоугольник. Его стороны проходят по линиям между клетками и называются (в соответствии с их расположением, как на географической карте) Северной Границей, Восточной Границей, Южной Границей и Западной Границей. Они образуют Границу Прямоугольника. Помимо сторон, в Прямоугольнике могут быть и другие стены — внутренние, также идущие по линиям между клетками. Кроме стен, мы выделяем в Прямоугольнике еще углы: Северо-Западный Угол, Северо-Восточный Угол и т. д. Путник в каждый момент времени находится в одной из клеток и смотрит на Север, Восток, Юг или Запад.

Перейдем к описанию команд и проверок Путника. При этом перед названием команды будем писать условия, в которых команда выполняется — что дано (они называются предусловием), а после названия команды будем указывать условия, которые имеют место после выполнения команды — что требуется (постусловие). Вот так:

предусловие (ДАНО): {непосредственно перед
Путником нет стены}

шаг_вперед

постусловие (ТРЕБУЕТСЯ): {Путник смотрит
в том же направлении, что и раньше, и переместился на одну клетку в этом направлении}

Как видите, пред- и постусловие мы заключаем в фигурные скобки. В дальнейшем слова «предусловие» и «постусловие», которые мы поместили перед фигурными скобками, писать не будем: и так ясно, что перед названием команды написано предусловие, а после команды — постусловие. Выписанная последовательность

{предусловие}

название команды

{постусловие}

называется спецификацией команды. Приведем спецификации других команд Путника. Следующая команда такова:

{ }

лицом_на_Север

{Путник в той же клетке, что и был, и смотрит на Север}

Мы видим, что в качестве предусловия (в фигурных скобках) не стоит ничего. Это значит лишь, что в условиях задачи, о которых мы договорились заранее, вы-

полнение команды всегда приводит к указанному результату. Вот еще одна команда:

```
{ }
```

```
направо
```

```
{Путник остался в той же клетке, повернувшись  
на 90 градусов по часовой стрелке}
```

Наши предусловия и постусловия ничего не говорят исполнителю — он может их вообще не замечать. Они нужны нам самим, и мы их пишем в произвольной форме.

Помимо команд, Путник умеет также осуществлять проверку «впереди_свободно». Ее смысл понятен.

Наконец, обратите внимание, что иногда мы склеиваем несколько слов вместе с помощью черточки: «шаг_вперед».

Новые команды

Итак, первоначально исполнитель может иметь весьма ограниченный набор команд. Смысл программирования состоит в том, чтобы «обучить» исполнителя выполнению нужных нам команд, расширив его «репертуар». При этом используются уже описанные способы образования команд. Чтобы «обучить» исполнителя новой команде, нужно сообщить ему описание этой команды.

Вот, например, описание команды «кругом» для Путника:

```
команду кругом понимать_как
```

```
. направо
```

```
. направо
```

```
конец_описания
```

(здесь нам понадобились новые служебные слова «команда», «понимать_как», «конец_описания»). Если мы сообщим это описание Путнику, команду «кругом» также можно будет использовать. Для нее предусловие и постусловие таковы:

```
{ }
```

```
кругом
```

```
{Путник остался в той же клетке, повернувшись  
на 180 градусов}
```

Напомним, что предусловие описывает то, в какой ситуации может использоваться команда, а постусловие — то, что будет после выполнения команды.

У п р а ж н е н и е. Придумать спецификацию и описание команды «палево».

Начиная определять новую команду, прежде всего надо иметь ее спецификацию. Попробуем написать спецификацию для команды, по которой Путник пойдет вперед, пока не упрется в ближайшую стенку и там остановится. Получится что-то вроде

```
{ }  
вперед_до_упора  
{сохраняя направление взгляда, Путник дошел  
до ближайшей стены}  
Как же выглядит описание команды? Вот оно:  
команду вперед_до_упора понимать_как  
. пока впереди_свободно повторять  
. . шаг_вперед  
. конец_пока  
конец_описания
```

Как видите, здесь описание команды не намного сложнее спецификации. В других случаях это не так.

У п р а ж н е н и е. Написать описание команды с такой спецификацией:

```
{ }  
на_Север_до_упора  
{Путник двигался только на Север, с Севера рядом  
стен}
```

Построение новой команды.

Процесс детализации

Пусть задана спецификация:

```
{все внутренние стены идут с Запада на Восток;  
они не доходят до Границы Прямоугольника}  
в_Северо-Западный_Угол  
{Путник в Северо-Западном Углу}
```

Будем строить описание команды. Представим себе, что мог бы делать Путник. Он, например, может идти на Запад до упора, при этом упрется он в Западную Границу, так как у внутренних стен нет участков, идущих с Севера на Юг. Затем он пойдет на Север до упора (тоже не встречая стен вплоть до Северной Границы). Выпишем описание нужной нам команды и спецификации двух других команд, которые мы сейчас придумали:

```
команду в_Северо-Западный_Угол понимать_как  
. на_Запад_до_упора  
. на_Север_до_упора  
конец_описания
```

При записи спецификаций будем опускать части условия, которые хотя и необходимы для правильного выполнения команды (например, «все внутренние стены идут с Запада на Восток»), но входят в условие задачи и не могут быть нарушены в результате действий Путника.

```
{ }
на_Запад_до_упора
{Путник у Западной Границы}
Еще одна спецификация:
{Путник у Западной Границы}
на_Север_до_упора
{Путник в Северо-Западном Углу}
```

Хотя на первый взгляд наше положение не улучшилось — пытаюсь объяснить исполнителю команду, мы получили пока еще непонятное ему описание и две новых спецификации для себя, но вы, конечно, понимаете, к чему идет дело. Теперь мы составим описания понадобившихся нам команд, используя еще более простые. Вот они:

```
команду на_Запад_до_упора понимать_как
. лицом_на_Запад
. вперед_до_упора
конец_описания
```

и

```
команду на_Север_до_упора понимать_как
. лицом_на_Север
. вперед_до_упора
конец_описания
```

Спецификация команды «лицом_на_Север» была задана с самого начала, а спецификация и описание команды «вперед_до_упора» были приведены в предыдущем разделе. Спецификация команды «лицом_на_Запад» ясна:

```
{ }
лицом_на_Запад
{Путник в той же клетке, где был, лицом на Запад}
```

Описание этой команды также несложно:

```
команду лицом_на_Запад понимать_как
. лицом_на_Север
. направо
. направо
. направо
конец_описания
```

Итак, построение команды закончено. Если сообщить Путнику все описания, он сумеет выполнить команду «в_Северо-Западный_Угол».

Процесс построения новой команды, которым мы пользовались, называется детализацией. Действительно, отправляясь от представления о работе команды в целом, мы разбивали программу на части, выписывая спецификации и описания для этих частей. Полученные детали, в свою очередь, разбивались на детали и т. д.

У п р а ж н е н и е. В Прямоугольнике имеется единственная стена, перегораживающая его в направлении Запад—Восток. В этой стене есть дверца шириной в одну клетку, расположенная не у Границы. Путник находится в Юго-Западном Углу и хочет попасть в Северо-Западный. Разработать описание соответствующей команды.

Семь раз отмерь — один раз отрежь, или Составные проверки

Обратите внимание на то, что мы много занимались командами и мало — проверками. Мы ввели средства для построения новых команд, но не ввели никаких средств для построения новых проверок, а такие средства, конечно, нужны. Сейчас мы рассмотрим некоторые из них.

Начнем с того, что исполнителю бывает полезно по ходу выполнения команды прикинуть, что было бы, если бы он поступил так-то и так-то, не делая самих действий, и на основании прикидки принимать решение.

Изменим временно мир Путника, предположив, что в некоторых клетках есть ямы, причем клетки с ямами и без них на вид одинаковы. Если Путник попадает в клетку с ямой, он проваливается в яму и выйти из ямы не может. В распоряжении Путника имеется проверка «в_яме», но пользы от нее мало: хотелось бы узнать о наличии ямы до того, как он в нее попал, а не после. Путнику очень пригодилась бы проверка «вперед_яма», которую можно описать так:

```
проверку_вперед_яма_понимать_как
.   после_шаг_вперед_было_бы
.   .   в_яме
.   конец_после
конец_описания
```

Такие составные проверки можно назвать сослагательными, поскольку команды и проверки, входящие в их состав, в действительности не выполняются. Они бывают полезны в ситуациях, в которых реальное выполнение невозможно или нежелательно. Простейший пример: нам хочется проверить, свободно ли на Север от Путника, но Путник не стоит лицом на Север и мы не хотим его поворачивать. Тогда можно написать:

```
| проверку на_Север_свободно_понимать_как  
| . после_лицом_на_Север_было_бы  
| . . впереди_свободно  
| . конец_после  
| конец_описания
```

Новые проверки можно образовывать, используя также союзы «и», «или», «не». Такие примеры нам еще встретятся.

У п р а ж н е н и я. (1) Описать проверку «сзади_свободно».

(2) В Прямоугольнике есть единственная стена, идущая с Востока на Запад и не примыкающая к Границе. Описать проверку «у Северной Границы». (Указание: не спутайте Границу со стеной.)

Пусть Путнику снова нужно попасть в Северо-Западный Угол, но при этом задача усложнилась:

```
| {внутренние стены не примыкают к Границе  
| и одна к другой}  
| в_Северо-Западный_Угол  
| {Путник в Северо-Западном Углу}
```

Здесь Путник уже не может, пойдя прямо на Запад или на Север, дойти до Границы. Можно, однако, заметить другое — есть ровно одна клетка Прямоугольника, из которой закрыт путь и на Запад, и на Север. Эта клетка — Северо-Западный Угол. В любом другом месте открыт путь или на Север, или на Запад (или оба): в противном случае две стены имели бы общую точку, что противоречит условию. Таким образом, можно записать:

```
| команду в_Северо-Западный_Угол_понимать_как  
| . пока_не_в_Северо-Западном_Углу_повторять  
| . . шаг_на_Север_или_на_Запад  
| . конец_пока  
| конец_описания
```

Спецификации новой проверки и новой команды очевидны. Правильность описания команды «в_Северо-

Западный Угол» (т. е. соответствие спецификации) вытекает из того, что

(1) команда повторения может закончить работу только тогда, когда ее условие не соблюдается, т. е. когда Путник окажется в Северо-Западном Углу;

(2) из каждой клетки, кроме Северо-Западного Угла, можно сдвинуться на Север или на Запад;

(3) такой сдвиг может произойти лишь конечное число раз.

Вот описание проверки:

проверку не_в_Северо-Западном_Углу

понимать_как

. на_Севере_свободно или на_Западе_свободно

конец_описания

А вот описание команды:

команду шаг_на_Север_или_на_Запад

понимать_как

. выбор

. . при на_Севере_свободно

. . делай шаг_на_Север

. . при на_Западе_свободно делай шаг_на_Запад

. конец_выбора

конец_описания

Все оставшиеся команды либо стандартны, либо очень просты, и их описания вы легко постройте сами.

У п р а ж н е н и я. (1) Одна из внутренних клеток Прямоугольника огорожена стенками со всех сторон, а больше стен нет. Описать проверку «впереди_огороженная_клетка».

(2) В условиях предыдущего упражнения Путник стоит у Южной Границы лицом на Север. Описать проверку, которая позволяла бы в этом случае определить, находится ли огороженная клетка на одной вертикали с ним. (Если он не стоит у Южной Границы лицом на Север, то проверка может давать любой ответ.)

(3) В условиях упражнения (1) описать команду

{Путник в Юго-Западном Углу}

ищи_клетку

{Путник в клетке, соседней с огороженной}

(4) В Прямоугольнике имеются лишь стены, идущие с Запада на Восток и не перегораживающие его. Описать команду

{ }

к_Северной_Границе

{Путник у Северной Границы}

(Указание: пока Путник не у Северной Границы, надо двигаться к Северу. Для этого надо искать дырку в стене, двигаясь с Запада на Восток).

Более сложные задачи, предназначенные для того же исполнителя, будут разобраны в главе 3 «Путник снова в лабиринте».

ГЛАВА 2

КОЗА, КАПУСТА И ДРУГИЕ С ТОЧКИ ЗРЕНИЯ ПРОГРАММИСТА

С задачами, которые можно уверенно отнести к ряду программистских, мы встречаемся не так уж редко. Ниже мы приведем примеры таких задач, обращая внимание на особенности подхода программиста к их решению.

Волк, коза, капуста

Условие этой старинной задачи таково. На левом берегу реки находятся лодочник с лодкой, волк, коза и капуста. Их нужно переправить на правый берег. Лодка, помимо лодочника, вмещает лишь одного из трех (волка, козу или капусту). При этом нельзя оставлять наедине волка с козой, а также козу с капустой.

Приведем решение этой задачи, т. е. программу действий лодочника (который в данном случае является исполнителем). Условимся, что «перевезти X » означает «переправиться вместе с X на другой берег», а «переплыть» означает «переправиться на другой берег без груза». Для начала попытайтесь разобраться в приводимом ниже решении без карандаша и бумаги, читая его по порядку и по возможности не возвращаясь к уже прочитанному. Вот что делает лодочник:

- перевезти козу
- переплыть
- перевезти волка
- перевезти козу
- перевезти капусту
- переплыть
- перевезти козу

Думаем, что предложенное задание потребовало от вас некоторых усилий (если вообще удалось его выпол-

нить: авторам это не удастся). Проверить правильность действий лодочника трудно — приходится несколько раз возвращаться к уже прочитанному, напрягать внимание и т. п. Нельзя ли как-то облегчить проверку?

Оказывается, можно, и очень просто. Нужно только добавить к последовательности команд утверждения, говорящие о том, какова ситуация в соответствующий момент выполнения программы:

{слева: волк, коза, капуста, лодочник; справа: никого}

перевезти козу

{слева: волк, капуста; справа: лодочник, коза}
переплыть

{слева: волк, капуста, лодочник; справа: коза}
перевезти волка

{слева: капуста; справа: коза, лодочник, волк}
перевезти козу

{слева: коза, лодочник, капуста; справа: волк}
перевезти капусту

{слева: коза; справа: волк, лодочник, капуста}
переплыть

{слева: коза, лодочник; справа: волк, капуста}
перевезти козу

{слева: никого; справа: волк, коза, капуста, лодочник}

Теперь проверка правильности стала тривиальной. Нужно убедиться лишь, что начальная и конечная ситуации совпадают с заданными условиями и что каждое действие в самом деле изменяет ситуацию ровно так, как это сказано. При этом — и в этом главное облегчение — при проверке очередной команды нам нужно знать только ее и окружающие ее утверждения, другие команды нам в этот момент безразличны.

У п р а ж н е н и е. Найдите другое решение этой задачи. Запишите его, указав все промежуточные утверждения.

Самый тяжелый

Задача. Есть несколько камней разного веса и чашечные весы без гирь, на каждую чашку которых помещается по одному камню. Найти самый тяжелый камень.

Попробуем решить эту задачу. Пусть вся куча кам-

ней лежит на левом краю стола. Выберем наугад один камень и переложим его на правый край. В каком случае можно утверждать наверняка, что он самый тяжелый? Такой случай, как ни странно, есть: когда камень единственный. Если камней несколько, то лежащий справа камень может оказаться и не самым тяжелым из всех камней. Зато он самый тяжелый из камней, лежащих справа (опять-таки потому, что он там один). Чтобы отметить это, изготовим табличку с надписью «самый тяжелый из тех, что справа» и повесим ее на выбранный камень.

Теперь перед нами две кучи камней — левая и правая. (Пусть вас не удивляет слово «куча»: мы считаем, что не только один камень, но и ноль камней — это куча.) В правой куче помечен самый тяжелый камень. Если бы при этом еще левая куча была пустой, задача была бы решена. Действительно, в этом случае

(1) самый тяжелый камень справа помечен;

(2) справа находятся все камни,

т. е. помечен самый тяжелый из всех.

К этой цели мы и будем стремиться. Поскольку цель (1) уже достигнута, попробуем достичь (2), не нарушая (1). Как? Перекладывая камни слева направо. Начнем.

Пусть мы переложили слева направо какой-нибудь камень. При этом мы явно приблизились к цели — утверждению (2). Но не нарушилось ли при этом утверждение (1), т. е. остался ли камень с табличкой самым тяжелым в правой куче? Да, остался, если только что переложенный камень легче камня с табличкой. А если новый камень тяжелее, то теперь самый тяжелый в правой куче — это он и табличку надо перевесить на него, восстановив тем самым истинность утверждения (1).

Мы видим, что возможна такая схема работы. Сперва кладем один камень на правый край стола, остальные на левый. Вешаем на камень справа табличку. Затем перекладываем камни по одному слева направо, каждый раз сравнивая только что переложенный камень с тем, на котором табличка, и добиваясь, чтобы табличка оставалась на самом тяжелом из правой кучи (т. е. всякий раз восстанавливая истинность утверждения (1), если она нарушилась после перекладывания). Так действуем до тех пор, пока левая куча не опустеет.

Запишем наши соображения более подробно. Будем считать, что исполнитель программы умеет выполнять

такие команды:

переложить_направо — по этой команде исполнитель переносит произвольный камень из левой кучи в правую; если переносить нечего, происходит авария;

пометить_переложенный — на последний из переложенных камней вешается табличка.

Кроме того, исполнитель способен выполнять проверки;

слева_есть_камни

помеченный_легче_переложенного

помеченный_тяжелее_переложенного

Вот что у нас получается:

{все камни слева}

переложить_направо

пометить_переложенный

{самый тяжелый в правой куче помечен}

пока слева_есть_камни повторять

. {слева есть камни, самый тяжелый в правой куче помечен}

. переложить_направо

. {самый тяжелый в правой куче — либо помеченный, либо переложенный}

. выбор

. . при помеченный_легче_переложенного делай

. . пометить_переложенный

. . при помеченный_тяжелее_переложенного

. . делай

. . ничего_не_делать

. конец_выбора

. {самый тяжелый в правой куче помечен}

конец_пока

{самый тяжелый в правой куче помечен; слева нет камней}

(Мы использовали команду «ничего_не_делать», не требующую никаких действий от исполнителя.)

У п р а ж н е н и е. Показать, что при решении данной задачи нельзя обойтись без команды повторения. (Указание: число камней может быть любым.)

Достаточно общий случай применения конструкции повторения (цикла) можно проиллюстрировать такой схемой. Работа всякой программы направлена на достижение какой-то цели. Выделим из этой цели то, чего можно достичь без использования цикла (повторения), назвав это подцелью № 1. Все остальное назовем под-

целью № 2. Тогда общий вид программы можно представить себе так:

достичь подцели № 1

пока подцель № 2 не достигнута **повторять**

. приблизиться к достижению подцели № 2, не

. нарушая уже достигнутого (подцели № 1)

конец_пока

Утверждение, истинность которого охраняется при повторении команд, входящих в цикл, называют инвариантом цикла. В нашей задаче инвариантом (подцелью № 1) было утверждение «самый тяжелый камень в правой куче помечен». Подробнее об инвариантах можно прочитать в главе 4 «Доказательства в программировании».

Не следует думать, что выбор подцелей однозначен — это как раз наименее очевидная (а значит, наиболее творческая) часть работы программиста. От выбора промежуточных целей существенно зависит получаемая в результате программа. В доказательство приведем другое решение задачи о самом тяжелом камне.

На этот раз запасемся коробкой и будем добиваться, чтобы в результате работы программы в коробке лежал самый тяжелый камень и только он. Таким образом, наша программа будет выглядеть так:

{есть по крайней мере один камень}

???

{самый тяжелый в коробке, других там нет}

Разобьем нашу цель на две, сформулировав ее так:

(1) самый тяжелый камень в коробке;

(2) в коробке ровно один камень.

Как достичь (1)? Очень просто — положив все камни в коробку! Приходим к такой программе:

{есть по крайней мере один камень}

положить_все_в_коробку

{самый тяжелый в коробке}

???

{самый тяжелый в коробке, в коробке один камень}

(Мы считаем, что исполнитель, для которого мы составляем программу, понимает все эти, а также все употребляемые далее команды и проверки.)

Теперь добиваемся истинности (2), не нарушая (1). Это можно сделать, лишь вынимая камни из коробки. Поскольку заранее неизвестно, сколько в коробке кам-

ней, придется воспользоваться циклом:

{самый тяжелый — в коробке}

пока в_коробке_больше_одного_камня повторять

. {самый тяжелый камень в коробке, там больше

. одного камня}

. уменьшить число камней в коробке, сохраняя истинность (1)

. {самый тяжелый — в коробке}

конец_пока

{самый тяжелый камень — в коробке, в коробке не больше одного камня}

Уменьшая число камней в коробке, нельзя вынуть самый тяжелый — иначе нарушим (1). То есть надо «вынуть не самый тяжелый камень». Что такое «не самый тяжелый»? Проще сначала сказать, что такое «самый тяжелый».

О п р е д е л е н и е. Камень, который тяжелее любого другого камня (из имеющихся у нас камней), называют самым тяжелым.

Не самый тяжелый камень этим свойством, очевидно, не обладает. Как сказать это без «не»? Вот как:

О п р е д е л е н и е. Камень, который легче хотя бы одного из камней, называют не самым тяжелым.

(Поскольку по условию все камни разного веса, понятия «тяжелее» и «легче» в данном случае взаимно исключающие.)

У п р а ж н е н и е. Куча — это нуль или более камней (снова считаем, что все они разного веса). Бывает ли куча, в которой нет самого тяжелого камня? Бывает ли куча, в которой все камни — самые тяжелые? Бывает ли куча, где все камни — не самые тяжелые?

Последнее определение нетрудно применить непосредственно. Выберем из коробки наугад два камня и сравним их веса. Тот, что легче, удовлетворяет последнему определению, т. е. не является самым тяжелым, и его можно вынуть из коробки. Окончательно программа выглядит так:

{есть не менее одного камня}

положить_все_в_коробку

{самый тяжелый — в коробке}

пока в_коробке_больше_одного_камня повторять

. {самый тяжелый в коробке; в коробке больше одного камня}

. взвесить_два_камня_и_вынуть_более_легкий

. {самый тяжелый — в коробке}

конец_пока

{самый тяжелый — в коробке, в коробке не более одного камня}

В заключение ответим на вопрос, который, возможно, уже возник у внимательного читателя. Как было сказано в главе «Команды исполнителю», выполнение команды повторения может никогда не закончиться. Почему этого не произойдет для наших программ поиска самого тяжелого камня? Ответ прост. В первой программе при каждом повторении число камней в левой куче уменьшается на 1, и потому число повторений равно общему числу камней минус 1. Аналогичным образом во второй программе при каждом повторении число камней в коробке уменьшается на 1 (подробнее об этом — в главе 7 «Сколько веревочке ни виться, или Почему программы кончают работу»).

ГЛАВА 3

ПУТНИК СНОВА В ЛАБИРИНТЕ

Одной из основных трудностей, стоящих перед программистом, является длина текстов программ. Мы поступили бы нечестно, если бы скрыли это и разбирали бы только короткие программы. Поэтому после некоторых колебаний мы решили включить в книгу пример решения трудной в этом отношении задачи, требующей разработки довольно объемной программы (реальные программы бывают еще во много раз длиннее). Советуем вам запастись терпением, читая эту главу (или, что проще, пропустить ее вовсе).

Итак, попробуем отправить Путника в Северо-Западный Угол в ситуации более сложной, чем ранее встречавшиеся.

{все внутренние стены идут с Севера на Юг или с Запада на Восток и не имеют общих точек друг с другом}

в_Северо-Западный_Угол

{Путник в Северо-Западном Углу}

Сразу же можно заметить, что требуемой команды не существует. Дело в том, что одна из внутренних стен может перегородивать весь Прямоугольник. Поэтому спецификации нужно изменить:

{все внутренние стены — отрезки, идущие с Севера на Юг или с Запада на Восток; никакая внутренняя стена не имеет общих точек с другими внутренними стенами и имеет не более одной общей точки с Границей}

в_Северо-Западный_Угол

{Путник в Северо-Западном Углу}

Мы начинаем детализацию команды «в_Северо-Западный_Угол», при этом, как и раньше, будем опускать часть предусловия, совпадающую с предусловием задачи в целом, которое мы только что выписали. (Напомним, оно не может быть нарушено Путником.) Естественный способ движения Путника состоит в постепенном приближении к Северо-Западному Углу:

команду в_Северо-Западный_Угол понимать как

. пока не_в_Северо-Западном_Углу повторять

. . приблизиться_к_Северо-Западному_Углу

. конец_пока

конец_описания

Команду «приблизиться_к_Северо-Западному_Углу» можно понимать по-разному. Единственное, что требуется, — чтобы повторение этой команды «в достаточном числе раз» приводило нас в этот Угол. Другими словами, нужно, чтобы команда повторения, которую мы сейчас выписали, заканчивала работу. Это можно обеспечить по-разному, например, с помощью следующей спецификации:

{Путник не в Северо-Западном Углу}

приблизиться_к_Северо-Западному_Углу

{Путник приблизился к Северной Границе, не удаляясь от Западной, или приблизился к Западной Границе, не удаляясь от Северной}

При такой спецификации наше описание команды «в_Северо-Западный_Угол» будет правильным. В самом деле, команда повторения заканчивает работу (при этом число повторений при выполнении этой команды будет не больше полупериметра Прямоугольника). После ее завершения Путник окажется в Северо-Западном Углу, так как условие команды повторения не соблюдается после ее завершения.

Приступим к детализации появившейся команды. Посмотрим, как должен вести себя Путник, чтобы удовлетворить ее спецификации. Первая и очевидная идея состоит в том, что в зависимости от ситуации Путник может или приблизиться к Северной Границе, не уда-

ляясь от Западной, или приблизиться к Западной Границе, не удаляясь от Северной. Другими словами, команду приблизиться_к_Северо-Западному_Углу понимать_как

• **выбор**

• . **при** можно_продвинуться_к_Северу **делай**

• . продвинуться_к_Северу

• . **при** можно_продвинуться_к_Западу **делай**

• . продвинуться_к_Западу

• **конец_выбора**

конец_описания

Мы видим, что понадобятся довольно сложные проверки. Конечно, их надо строить, выписывая сначала спецификации, а уже потом разрабатывая проверку, удовлетворяющую имеющейся спецификации. Как же задавать спецификацию проверки? В ней, как и в спецификации команды, будем указывать предусловие и постусловие:

{предусловие}

проверка

{постусловие}

Однако смысл этих условий будет несколько иной, чем для команд. Предусловие, как и в случае команды, описывает те ситуации, в которых проверка должна давать ясный для нас ответ. В частности, если предусловие выполнено, работа проверки должна обязательно заканчиваться, она не может работать бесконечно долго. Постусловие же выделяет среди всех ситуаций, удовлетворяющих предусловию, в точности те, для которых проверка дает ответ «да».

Возвращаясь к нашей задаче, нетрудно понять, что для правильности работы команды, которую мы хотим построить, достаточно, чтобы

(1) какая-то из проверок давала бы ответ «да», если Путник не в Северо-Западном Углу;

(2) в ситуации, где проверка «можно_продвинуться_к_Северу» дает ответ «да», команда «продвинуться_к_Северу» действительно дает продвижение к Северу (аналогично для Запада).

Продвижение к Северу заведомо возможно, если на Севере свободно (нет стены). Поэтому попытаемся дать проверкам такие спецификации:

{Путник не в Северо-Западном Углу}

можно_продвинуться_к_Северу

{на Севере от Путника свободно}

и

```
{Путник не в Северо-Западном Углу}
| можно_продвинуться_к_Западу
| {на Западе от Путника свободно}
```

Однако пара проверок, отвечающая этим спецификациям, не удовлетворяет выписанному условию (1) Действительно, Путник может оказаться в ситуации когда и с Севера и с Запада занято, но Путник находится не в Северо-Западном Углу (рис. 1).

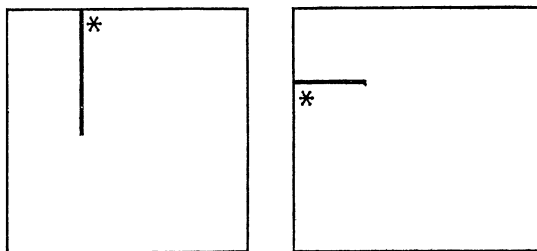


Рис. 1

Попытаемся изменить спецификации проверок. Конечно, продвижение на Запад уж точно невозможно, если Путник дошел до Западной Границы (аналогично для Севера). Попробуем выбрать спецификации так:

```
{Путник не в Северо-Западном Углу}
| можно_продвинуться_к_Северу
| {Путник не у Северной Границы}
```

и

```
{Путник не в Северо-Западном Углу}
| можно_продвинуться_к_Западу
| {Путник не у Западной Границы}
```

Теперь условие (1) выполняется. В самом деле, если Путник не в Северо-Западном Углу, то он не может оказаться одновременно у Северной и Западной границ, т. е. одна из проверок даст ответ «да».

Как же обстоит дело с условием (2)? Здесь нам придется подумать о том, как будут работать наши команды. По-видимому, в тех сложных случаях, которые были нарисованы, Путнику придется огибать стену с Юга или с Востока. Рассмотрим, скажем, случай, когда Путник у Северной Границы в клетке, с Запада примыкающей к стене. Тогда стену нужно огибать с Юга (рис. 2). Чтобы это оказалось возможным, нужно, чтобы, идя вдоль стены на Юг, Путник мог добраться до прохода на Запад. Тогда, дойдя до этого прохода, Путник сдвинется на Запад и потом пойдет на Север (рис. 3).

Это соответствует следующей спецификации:
 {с Запада от Путника свободно или стенка, кото-
 рую можно обогнуть с Юга}
 продвинуться_к_Западу
 {Путник приблизился к Западной Границе и не
 удалился от Северной}

Спецификация для команды «продвинуться_к_Северу» пишется аналогично. Мы получили спецификации новых команд, но теперь нам нужно доказать, что команда «в_Се-

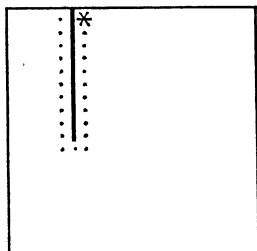


Рис. 2

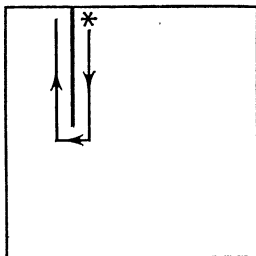


Рис. 3

веро-Западный Угол» соответствует своей спецификации, проверив условие (2). Приступив к этому, мы обнаруживаем, что доказательство не получается! Действительно, пусть проверка «можно_продвинуться_к_Западу» дала ответ «да». Это могло произойти, например, в ситуации, показанной на рис. 4.

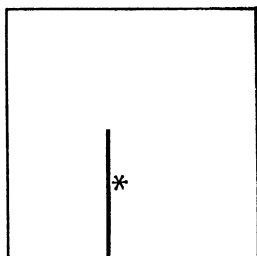


Рис. 4

(Путник ведь НЕ у Западной Границы!) Здесь, однако, стенку нельзя обогнуть с Юга. Конечно, это вроде бы не страшно, ведь здесь, очевидно, можно приблизиться к Северо-Западному Углу, просто пойдя на Север. Однако мы так устроили команду выбора, что Путник может попытаться сдвинуться на Запад и потерпит неудачу. Это означает, что мы неправильно выбрали спецификации проверок в команде выбора. Попытаемся в качестве постусловий для проверок взять просто предусловия для соответствующих команд. Тогда нам автоматически удастся добиться выполнения условия (2). Но зато снова возникнет вопрос об условии (1): почему в любой возможной ситуации одна из проверок дает ответ «да»? Другими словами, почему, если Путник не в Северо-Западном

Углу, то верно по крайней мере одно из четырех утверждений:

- на Севере свободно
- на Западе свободно
- с Запада от Путника стена и ее можно обогнуть с Юга
- с Севера от Путника стена и ее можно обогнуть с Востока

Попробуем это понять. Если на Севере и на Западе от Путника стены, то в северо-западном углу клетки, в которой он стоит, сходятся две стены. Одна из них — это Северная или Западная Граница, другая — внутренняя стена (рис. 5). Но в этих случаях внутреннюю

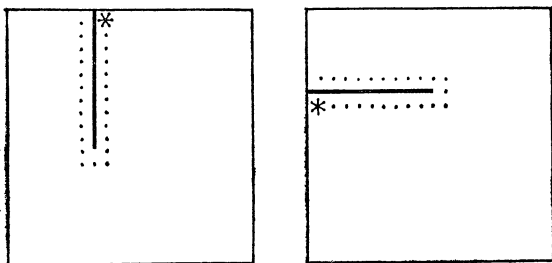


Рис. 5

стену можно обогнуть с Юга или с Востока. Итак, хотя бы одно из четырех утверждений действительно верно. Постусловия проверок и предусловия команд «состыковались»! Исправим спецификации для проверок:

```
{ }
можно_продвинуться_к_Западу
{с Запада свободно, или с Запада стена, которую
можно обойти с Юга}
```

и

```
{ }
можно_продвинуться_к_Северу
{с Севера свободно, или с Севера стена, которую
можно обойти с Востока}
```

Чего же мы достигли? Мы доказали соответствие команды «приблизиться_к_Северо-Западному_Углу» ее спецификации при условии, что проверки и команды, входящие в ее состав, удовлетворяют своим спецификациям. Изобразим все появившиеся команды и проверки на рисунке. На рис. 6 от команды или проверки идут вниз линии к тем командам или проверкам, которые она использовала. Для всех команд и проверок этого

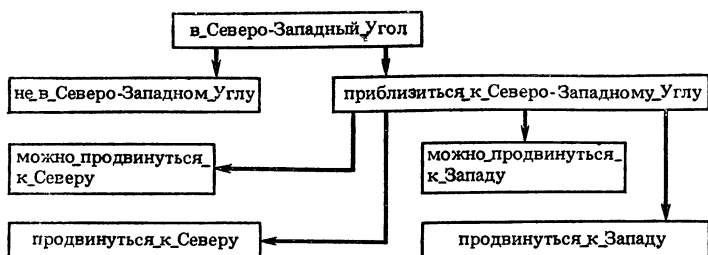


Рис. 6

рисунка мы построили спецификации, а все построенные нами описания правильны.

Однако наша задача еще не решена. Используемые нами команды и проверки нуждаются в детализации, они не являются исходными. Продолжим детализацию:

```

команду продвинуться_к_Западу понимать как
.  выбор
.  .  при на_Западе_свободно делай
.  .  шаг_на_Запад
.  .  при можно_обогнуть_стену_с_Юга делай
.  .  обогнуть_стену_с_Юга
.  конец_выбора
конец_описания
  
```

Что касается проверки «на_Западе_свободно» и команды «шаг_на_Запад», то с ними все просто, и мы о них больше говорить не будем. Оставшиеся команды и проверки имеют такие спецификации:

```

{с Запада свободно или стена, которую можно
 обогнуть с Юга} ← можно_обогнуть_стену_с_Юга
{с Запада стена, которую можно обогнуть с Юга}
  
```

и

```

{с Запада стена, которую можно обогнуть с Юга}
 обогнуть_стену_с_Юга
{Путник приблизился к Западной Границе, не
 удалившись от Северной}
  
```

Ясно, что при таких спецификациях наше описание команды «продвинуться_к_Западу» правильно. Попробуем детализировать новые проверки и команды:

```

проверку можно_обогнуть_стену_с_Юга понимать как
.  после на_Юг_до_упора_или_прохода_на_Запад
.  .  было бы на_Западе_свободно
  
```

Картинка здесь такова:

стена | * здесь Путник был
| * сюда Путник попал

ИЛИ

стена

* здесь Путник был

* сюда Путник попал

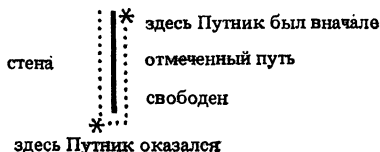
Если после выполнения команды проверка «на_Западе_свободно» дает ответ «да», то, значит, ситуация соответствует первой картинке и стену можно обогнуть с Юга, если нет, то второй и стену обогнуть нельзя. Итак, проверка детализирована правильно. Теперь детализируем команду:

конец_описания

Спецификации использованных команд понятны. Выпишем все же первую из этих спецификаций:

$\left\{ \begin{array}{l} \text{стена} \\ \vdots \\ \text{здесь Путьник} \end{array} \right\}$

Почему команда «обогнуть_стену_с_Юга» удовлетворяет своей спецификации (т. е. почему при ее выполнении Путник приближается к Западной стене и не удаляется от Северной)? В ходе выполнения команды Путник сдвигается на одну клетку к Западу. Что же касается движения на Север, то сначала дело обстояло неблагоприятно — Путник шел на Юг. Однако потом, дойдя до конца стены и сделав шаг на Запад, он оказался в такой ситуации:



Поэтому после команды «на_Север_до_упора» он заведомо окажется не южнее, чем был вначале. Итак, команда «обогнуть_стену_с_Юга» описана правильно. Используемая в ней команда «на_Юг_до_прохода_на_Запад» описывается так:

```
команду на_Юг_до_прохода_на_Запад понимать
как
. пока на_Западе_занято повторять
. . шаг_на_Юг
. конец_пока
конец_описания
```

Входящие в нее проверки и команды просты, и мы ими заниматься не будем.

Интересно посмотреть, что будет, если перед выполнением команды «обогнуть_стену_с_Юга» на Западе от Путника было свободно. Тогда команда «шаг_на_Запад» сдвинет его на Запад, и дальше он будет двигаться только на Север. Таким образом, команда «обогнуть_стену_с_Юга» удовлетворяет спецификации команды «продвинуться_к_Западу». Тем самым выбор в описании команды «продвинуться_к_Западу» становится ненужным, и мы получаем вот что:

```
команду продвинуться_к_Западу понимать как
. на_Юг_до_прохода_на_Запад
.. шаг_на_Запад
. на_Север_до_упора
конец_описания
```




Рис. 7

Теперь схема, изображающая наши команды и проверки, выглядит как показано на рис. 7. Что еще осталось трудного? Нужно детализировать проверки «можно_продвинуться_к_Северу» и «можно_продвинуться_к_Западу» и, наконец, проверку «не_в_Северо-Западном_Углу». При детализации первых двух проверок воспользуемся имеющейся детализацией (ненужной теперь) проверки «можно_обогнуть_стену_с_Юга». Получаем:

проверку можно_продвинуться_к_Западу
понимать_как
 . после на_Юг_до_упора_или_прохода_на_Запад
 было бы
 . . на_Западе_свободно
 . конец_после
конец_описания

(Проверьте правильность описания сами! Аналогично поступаем и с проверкой «можно_продвинуться_к_Северу».)

Последним крупным делом, которое осталось, является детализация проверки «не_в_Северо-Западном_Углу». Здесь нам помогут уже построенные проверки:

проверку не_в_Северо-Западном_Углу **понимать_как**
 . можно_продвинуться_к_Северу **или**
 . можно_продвинуться_к_Западу
конец_описания

Как мы уже знаем, если Путник не в Северо-Западном Углу, то по крайней мере одна из двух использованных проверок дает ответ «да». Значит, и построенная проверка будет давать ответ «да», что нас устраивает. Если же Путник находится в Северо-Западном Углу, то, как легко убедиться (и это обязательно надо сделать!) обе упомянутые проверки дадут ответ «нет». Значит, и проверка «не_в_Северо-Западном_Углу» даст ответ «нет», что тоже нас устраивает. Наша задача решена!

У п р а ж н е н и е. Построить команду со спецификацией

{все внутренние стены — отрезки, имеющие не более одной общей точки с другими стенами (внутренними или граничными)}

в_Северо-Западном_Углу

{Путник в Северо-Западном Углу}

Для тех читателей, которые добрались до этого места главы, сообщаем, что программы, управляющие движением Путника, можно запускать на реальных, даже небольших компьютерах. Быть может, воспользовавшись школьным или каким-нибудь еще компьютером, вы сможете увидеть, как Путник — в полном соответствии с придуманной вами программой — выбирается из различных головоломных ситуаций.

ГЛАВА 4

ДОКАЗАТЕЛЬСТВА В ПРОГРАММИРОВАНИИ

Как вы уже убедились, построение программы начинается с понимания ее назначения, т. е. того, что собственно эта программа должна делать. В ходе разработки программы мы стараемся добиться, чтобы программа действительно это делала. Соответствие между программой и ее спецификацией и называется правильностью программы и подлежит доказательству.

В школе вы встречались с доказательством правильности программ, решая геометрические задачи на построение. В них нужно не только привести искомое построение, но и доказать, что при его выполнении будет построено именно то, что требовалось. Во многих других случаях также мало только указать ответ, а необходимо и доказать его правильность.

Строя доказательства геометрических теорем, можно пользоваться интуитивными соображениями о том, что такое геометрическое доказательство. Однако довольно скоро возникает нужда в построении формальной аксиоматической системы для геометрии. В программировании такие формальные системы доказательства правильности программ также могут быть построены. Мы, однако, не будем стараться формально изложить систему такого рода. Вместо этого постараемся выделить некоторые принципы доказательства правильности программ, которыми мы уже пользовались. Выделение таких принципов можно рассматривать как первый шаг в указанном направлении.

Программы и спецификации

Вот фрагмент беседы Алисы с Чеширским Котом (из книги Л. Кэрролла «Алиса в Стране Чудес»):

— Скажите, пожалуйста, куда мне отсюда идти?

— А куда ты хочешь попасть? — ответил Кот.

— Мне все равно... — сказала Алиса.

— Тогда все равно, куда и идти, — заметил Кот.

Простое соображение, высказанное Чеширским Котом, достойно того, чтобы стать первой заповедью каждого программиста. Оно напоминает ему о том, что прежде чем писать программу, нужно сформулировать, чего он от нее ожидает, — то, что мы называли спецификацией. Спецификация, помимо желаемого результата (что нужно получить в результате работы программы), должна указывать, в каких условиях программа начинает свою работу (что дано). После того как спецификация составлена, можно начинать строить программу.

Вот пример спецификации:

 { дано: чайник с водой стоит на плите }

 X

 { получить: чайник вскипел }

Как вы думаете, в чем состоит искомая программа? Конечно, нужно всего лишь зажечь газ! Программа из этой единственной команды правильна в том смысле, что после ее выполнения в ситуации, когда чайник с водой стоит на плите, мы достигнем требуемого (чайник вскипит). В других ситуациях (например, если воды в чайнике нет) наша программа приведет к совсем другим результатам, но это не мешает ей быть правильной в описанном смысле.

Можно ставить задачу и иначе, считая неизвестным членом тройки

{предусловие — что дано}
программа
{постусловие — что нужно получить}

не программу, а что-нибудь другое. Вот пример такого рода:

{ $x = 1985$ }
увеличить x на 1
{???

Ответ ясен: в качестве постусловия можно взять условие { $x = 1986$ }.

Другой пример:

{???)
увеличить x на 1
{ x — четное целое число}

Здесь естественно предложить в качестве ответа условие { x — нечетное целое число}. Однако $x = 1985$ тоже годится, так как если x было равно 1985, то после прибавления единицы получится 1986, а это — четное число. Таким образом, возможны различные предусловия, гарантирующие истинность данного постусловия после выполнения данной программы. (Естественно стремиться выбрать среди них наиболее слабое и тем самым сделать множество случаев, в которых мы можем уверенно применять нашу программу, наибольшим.)

Обратимся теперь снова к трем способам образования новых команд и посмотрим, что происходит с пред- и постусловиями при построении программ с помощью этих конструкций.

Последовательное выполнение, или Не все сразу

Рассмотрим еще один пример с чайником.

{в чайник налита вода}
 Y
{чайник с водой стоит на газовой плите}

Здесь в качестве Y нужно, разумеется, взять команду «поставить чайник на плиту». Пусть теперь наша задача сложнее:

{в чайник налита вода}
 Z
{чайник закипел}

Ее решение может быть получено путем последовательного выполнения уже известных нам команд: поста-

вив чайник на плиту, мы добьемся выполнения условия {чайник с водой стоит на газовой плите}. После этого команда «зажечь газ» гарантирует требуемое.

Мы воспользовались таким правилом. Пусть имеются две команды и мы знаем, что они соответствуют спецификациям

{условие 1}	{условие 2}
команда 1	команда 2
{условие 2}	{условие 3}

Тогда для составной команды будем иметь

```
{условие 1}
команда 1
команда 2
{условие 3}
```

Чтобы не забывать о том, что между выполнением команд 1 и 2 соблюдается условие 2, пишут так:

```
{условие 1}
команда 1
{условие 2}
команда 2
{условие 3}
```

Здесь условие 2 является постусловием для команды 1 и предусловием для команды 2.

Разветвление, или Не мытьем, так катаньем

Часто бывает, что одну и ту же цель можно достигнуть различными способами. Пусть, например, из А в Б можно ехать на автобусе или троллейбусе:

{Мы в А, подошел автобус}	{Мы в А, подошел троллейбус}
ехать_на_автобусе	ехать_на_троллейбусе
{Мы в Б}	{Мы в Б}

С помощью команды выбора из этих двух команд можно составить одну:

```
{Мы в А, подошел троллейбус или автобус}
выбор
. при подошел_троллейбус делай
. ехать_на_троллейбусе
. при подошел_автобус делай ехать_на_автобусе
конец_выбора
{Мы в Б}
```

Напомним два обстоятельства, связанных с командой выбора: (1) если подошел и троллейбус и автобус, нашему исполнителю разрешено воспользоваться любым

из них; (2) если нет ни автобуса, ни троллейбуса, наша команда не позволяет попасть в пункт Б.

Вот еще один пример применения конструкции выбора:

$\{ \text{дано: } a, b \text{ — числа} \}$
 $\{ \text{получить: } c \text{ равно наибольшему из чисел } a, b \}$
 Мы знаем, что наибольшее из чисел a и b равно либо a , либо b . В каком случае оно равно a ? Если $a \geq b$! Аналогично оно равно b , если $b \geq a$. Другими словами, имеем

$\{ a, b \text{ — числа, } a \geq b \}$ положить c равным a $\{ c = \max(a, b) \}$	$\{ a, b \text{ — числа, } a \leq b \}$ положить c равным b $\{ c = \max(a, b) \}$
--	--

Соединяя эти команды в одну, получим

$\{ a, b \text{ — числа, } a \leq b \text{ или } a \geq b \}$
выбор
 . при $a \geq b$ делай положить c равным a
 . при $a \leq b$ делай положить c равным b
конец_выбора
 $\{ c = \max(a, b) \}$

Сформулируем теперь общее правило для команд, построенных с помощью конструкции выбора. Чтобы иметь право написать

$\{ A \}$
выбор
 . при проверка_1 делай команда_1

 . при проверка_n делай команда_n
конец_выбора
 $\{ B \}$

где A и B — некоторые условия, мы должны быть уверены в том, что

$\{ A \text{ и проверка}_1 \}$ команда_1 $\{ B \}$. . .	$\{ A \text{ проверка}_n \}$ команда_n $\{ B \}$
--	-------	--

И, кроме того, надо быть уверенными в том, что в любой ситуации, где выполнено условие A , одна из проверок дает ответ «да».

У п р а ж н е н и е. Сформулировать аналогичное правило для конструкции **если...то...иначе...конец_если**, рассматривая ее как частный случай конструкции выбора.

Команда повторения, или Терпение и труд все перетрут

Часто бывает так, что одна и та же команда или последовательность команд должна повторяться многократно, прежде чем требуемый результат не будет достигнут. Как мы помним, в такой ситуации используется конструкция цикла.

Прежде чем привести пример, напомним, что сумма цифр любого числа дает при делении на 9 тот же остаток, что и само это число (на этом основан известный признак делимости на 9). Пусть на доске написано некоторое число x . Будем применять такую команду:

- пока десятичная запись числа на доске содержит
 - более одной цифры
 - повторять
 - заменить число на доске суммой его цифр
- конец_пока

Какое число на доске будет написано после выполнения этой команды? Отвечая на этот вопрос, заметим следующее:

(1) Это число будет однозначным, так как, пока десятичная запись числа на доске содержит более одной цифры, команда продолжает работу.

(2) Это число будет положительным и будет давать тот же остаток при делении на 9, что и исходное число x . В самом деле, остаток при делении числа на 9 не меняется при замене числа на сумму его цифр. Значит, и после нескольких таких замен остаток не изменится. Кроме того, сумма цифр положительного числа положительна, так что число на доске будет оставаться положительным.

Таким образом, в конце работы программы на доске будет написано целое положительное число, дающее при делении на 9 тот же остаток, что и x . Другими словами, на доске будет написан остаток от деления на 9 числа x , если этот остаток не равен 0; в последнем случае на доске будет написано число 9.

Чтобы полностью убедиться в правильности нашей команды, осталось проверить, что она действительно заканчивает свою работу. Это легко: поскольку сумма цифр многозначного числа меньше самого числа, число на доске с каждым повторением становится все меньше. Таким образом, работа программы не может продолжаться бесконечно долго (подробнее об этом в главе 7).

У п р а ж н е н и е. Докажите, что сумма цифр числа, содержащего более одной цифры, меньше самого числа.

Построение циклических команд — одно из самых важных средств программирования. Важность объясняется тем, что только они позволяют описать в виде короткой программы большой объем вычислений. Обоснование свойств циклических команд играет центральную роль в доказательствах правильности программ. Чтобы привыкнуть к рассуждениям такого рода, рассмотрим еще один пример.

Пусть на доске написаны 57 единиц и 91 минус единица. Что останется на доске после выполнения такой команды:

пока на доске больше одного числа

. повторять

. взять какие-то два числа и заменить их на их

. произведение

конец_пока

Чтобы решить эту задачу, заметим, что при замене двух чисел на их произведение остается неизменным произведение всех чисел на доске. Вначале это произведение равняется минус единице, так как число минус единиц (91) нечетно. Таким образом, условие произведение чисел на доске равно -1

(назовем его И) продолжает соблюдаться и после любого числа повторений, или, как говорят, является инвариантом нашей команды повторения.

С другой стороны, команда повторения выполняется, пока на доске больше одного числа. Значит, после ее окончания на доске останется всего лишь одно число. Это число должно равняться -1 . Работа программы обязательно завершится, так как количество чисел на доске с каждым разом уменьшается.

Постараемся теперь явно сформулировать схему рассуждений, которой мы пользовались. Нам нужно было исследовать свойства команды

пока П

. повторять

. К

конец_пока,

где П — некоторая проверка, а К — команда. Для этого мы отыскивали свойство И, которое сохранялось при выполнении команды К. Такое свойство называют инвариантом цикла (это объясняет выбор буквы И

для его обозначения). В первом случае это было свойство «число на доске положительно и дает тот же остаток при делении на 9, что и x ». Во втором случае инвариантом было свойство «произведение всех чисел на доске равно -1 ». Найдя такое свойство, мы могли быть уверены в том, что если И соблюдалось до начала выполнения команды, то оно будет соблюдаться и после ее выполнения. Кроме того, мы знаем, что после выполнения команды повторения проверка П будет давать ответ «нет» (иначе команда бы не кончила работу). Все это можно изобразить так:

$\left \begin{array}{c} \{П \text{ и } И\} \\ K \\ \{И\} \end{array} \right $	$\left \begin{array}{c} \{И, \text{ выполнение команды в данной ситуации кончается} \} \\ \text{пока } П \text{ повторять } K \text{ конец_пока} \\ \{И, П \text{ дает ответ «нет»} \} \end{array} \right $

Слева написано то, что нужно проверить, чтобы быть уверенным в том, что написано справа. Как видно из написанного, вопрос о завершении работы программы (не «зациклится» ли она) должен быть рассмотрен отдельно (см. об этом в главе 7 «Сколько веревочке пи виться...»).

Не огорчайтесь, если вам пока не все понятно в сказанном: умение пользоваться изложенными правилами приходит лишь с опытом. Понадобилось около 10 лет (правила были изобретены в 1968 г. К. Хоаром), чтобы они превратились в рабочий инструмент программиста. Впрочем, эта фраза, быть может, слишком оптимистична: до сих пор некоторые программируют «по старинке». Мы надеемся, что приведенные в нашей книжке примеры помогут вам научиться пользоваться этим инструментом.

ГЛАВА 5

АЛГОРИТМ ЕВКЛИДА

Тема нашего рассказа — алгоритм Евклида нахождения наибольшего общего делителя. Это один из самых древних и популярных алгоритмов, поэтому редко кто из авторов книг о программировании не обсуждает его. Например, он имеется в школьном учебнике информатики (Основы информатики и вычислительной техники. Ч. 1. М.: Просвещение, 1985), и, кстати, его обоснование там содержит ошибку.

Начнем с определений. Говорят, что целое число x делится на целое число y , если $x = yz$ для некоторого целого z . В этом случае число y называют также делителем числа x . Наша задача состоит в отыскании наибольшего из общих делителей двух заданных целых чисел a, b — наибольшего числа, на которое делятся и a , и b (наибольший общий делитель a и b обозначается НОД (a, b)).

Прежде чем искать что-то, надо быть уверенным, что искомый объект существует. В данном случае наибольшего общего делителя может не быть, вообще говоря, по двум причинам:

- (1) нет ни одного общего делителя;
- (2) среди общих делителей нет самого большого: для каждого общего делителя есть еще больший.

Возможность (1), к счастью, отпадает, так как число 1 является делителем всех чисел и, следовательно, является общим делителем чисел a и b . Возможность (2) реализуется, если $a = b = 0$. В этом случае все целые числа будут общими делителями чисел a и b . Если же хотя бы одно из чисел (например, a) не равно 0, то все делители a (и тем более все общие делители) не превосходят $|a|$, их число конечно и среди них есть наибольший. Итак, условия разрешимости нашей задачи выяснены.

{ a, b — целые числа, не равные 0 одновременно}
 ???
 {найдено $x = \text{НОД}(a, b)$ }

Самый простой способ действий — буквальное следование определению. Что означает, что x — наибольший общий делитель? Это значит, что

- (1) x — общий делитель;
- (2) всякое число, большее x , не является общим делителем. Естественный способ поиска такого числа x — начать с x , заведомо большего требуемого, а затем уменьшать его, ища требуемое. Другими словами, схема поиска такова:

{ a, b — целые числа, не равные 0 одновременно}
 взять достаточно большое x
 {всякое число, большее x , не является общим делителем a, b }
 уменьшать x , пока не найдем требуемое
 { $x = \text{НОД}(a, b)$ }

Мы выбрали в качестве промежуточной цели утверждение (2) и тем самым разбили задачу на две. Начнем

с первой. По существу, первая задача (нахождение достаточно большого x) уже была решена нами, когда мы доказывали существование НОД. Именно, мы говорили, что при $a \neq 0$ никакое число, большее $|a|$, не является общим делителем. Аналогично, при $b \neq 0$ никакое число, большее $|b|$, не является общим делителем. Таким образом, мы можем достичь нашей цели при $a \neq 0$ (положив x равным $|a|$) и при $b \neq 0$ (положив x равным $|b|$). По условию, хотя бы одно из чисел a и b отлично от 0, так что первая задача решена:

```

{a, b — целые числа, не равные 0 одновременно}
  выбор
    . при a ≠ 0 делай x ← |a|
    . при b ≠ 0 делай x ← |b|
  конец_выбора
{всякое число, большее x, не является общим делителем a, b}

```

Здесь « $x \leftarrow |a|$ » означает «положить x равным $|a|$ » и т. п.

Перейдем ко второй задаче. В ней нам известно, что всякое число, большее x , не является общим делителем. Нужно, не нарушая этого свойства, добиться, чтобы x было общим делителем. Вот решение:

```

{всякое число, большее x, не является общим делителем a, b}
  пока x не является общим делителем a, b
    . повторять
    . уменьшить x на 1
  конец_пока
{x является общим делителем a, b, все бóльшие — нет}

```

Покажем, что эта программа действительно решает нашу задачу. Для этого покажем, что условие

{всякое число, большее x , не является общим делителем a, b },

которое мы будем обозначать для краткости через (*), не нарушается в ходе работы программы. В самом деле, мы знаем, что перед уменьшением x на 1

(1) x не является общим делителем;

(2) все бóльшие x числа не являются общими делителями.

Другими словами, все числа, бóльшие $(x - 1)$, не являются общими делителями, и мы можем смело уменьшить x на 1, не боясь нарушить условие (*). После завершения команды повторения x является

общим делителем (иначе она не завершилась бы) и условие (*) соблюдается, т. е. x — наибольший общий делитель.

Выполнение программы закончится, так как x не может стать отрицательным (тогда условие (*) нарушилось бы: число 1 было бы общим делителем, большим x), а каждый раз x уменьшается на 1. Приведем полностью получившуюся программу:

выбор $\{a, b$ — целые числа, не равные 0 одновременно}

. при $a \neq 0$ делай $x \leftarrow |a|$

. при $b \neq 0$ делай $x \leftarrow |b|$

конец_выбора

$\{$ все числа, большие x , не являются общими делителями a, b

пока x не является общим делителем a, b

. **повторять**

. $\{$ все числа, большие $(x - 1)$, не являются общими делителями a, b

. уменьшить x на 1

. $\{$ все числа, большие x , не являются общими делителями a, b

конец_пока

$\{x$ — общий делитель a, b , все большие — нет, т. е. $x = \text{НОД}(a, b)$

В этой программе мы предполагаем, что ее исполнитель умеет проверять, является ли одно число делителем другого. Ее недостатком является то, что цикл может выполняться очень много раз, прежде чем результат будет достигнут. Есть другой способ нахождения наибольшего общего делителя, который часто позволяет обойтись гораздо меньшим числом операций. Он приводится в знаменитых «Началах» Евклида и называется в его честь алгоритмом Евклида. К его изложению мы и перейдем. Отметим сразу же, что мы будем излагать упрощенный вариант этого алгоритма (в котором деление заменено вычитанием).

Построение алгоритма

Часто случается так, что определение какого-то понятия оказывается менее полезным, чем его свойства. Например, в большинстве геометрических задач про окружность используется не ее определение, а теорема о равенстве вписанных углов, опирающихся на одну

дугу. Подобная ситуация имеет место и здесь: пользуясь подходящими свойствами НОД, мы получаем другой, во многих отношениях лучший алгоритм его вычисления. Укажем эти свойства.

Л е м м а

$$(1) \text{НОД}(a, b) = \text{НОД}(-a, b) = \text{НОД}(a, -b)$$

$$(2) \text{НОД}(a - b, b) = \text{НОД}(a, b),$$

$$(3) \text{НОД}(a, b - a) = \text{НОД}(a, b),$$

$$(4) \text{НОД}(a, 0) = |a| \text{ при } a \neq 0.$$

Словами: НОД не меняется при изменении знаков одного из чисел и при уменьшении одного из чисел на величину, равную другому.

Д о к а з а т е л ь с т в о. Свойство (1) очевидно, так как делители у a и $-a$ одни и те же. (4) также очевидно: так как все числа являются делителями 0, то общие делители a и 0 — это попросту делители a и наибольший из них равен $|a|$. Центральными являются свойства (2) и (3). Они аналогичны друг другу; докажем, например (2).

Утверждение (2) будет доказано, если мы установим, что множество общих делителей чисел a и b равно множеству общих делителей чисел $a - b$ и b . Другими словами, нужно доказать, что всякий общий делитель x чисел a и b является общим делителем $a - b$ и b , и наоборот. Пусть x — делитель a и b . Тогда $a = kx$, $b = lx$ для некоторых целых k и l и $a - b = (k - l)x$. Таким образом, x является общим делителем $a - b$ и b . Наоборот, пусть $a - b = mx$ и $b = nx$ для некоторых m и n . Складывая, получаем $a = (m + n)x$; таким образом, x является общим делителем a и b . Лемма доказана.

Отметим, что утверждение леммы остается правильным, если мы доопределим функцию НОД, положив $\text{НОД}(0, 0) = 0$ (в этом случае оговорку «при $a \neq 0$ » можно исключить). Так мы и сделаем.

Перечисленные равенства подсказывают идею алгоритма нахождения НОД: нужно менять числа, не изменяя их наибольшего общего делителя до тех пор, пока одно из них не окажется равным 0. После этого свойство (4) позволит найти НОД. При такой схеме алгоритма (напоминаем: $A \leftarrow a$ — обозначение для «положить A равным a » и т. п.)

$$\left\{ \begin{array}{l} a, b \text{ — целые числа, одно из которых не равно } 0 \\ A \leftarrow a, B \leftarrow b \\ \text{НОД}(A, B) = \text{НОД}(a, b) \end{array} \right.$$

пока $A \neq 0$ и $B \neq 0$ повторять

. менять A и B , не меняя их НОД

конец_пока

{НОД (A , B) = НОД (a , b), $A = 0$ или $B = 0$ }

воспользоваться свойством (4)

{ $x = \text{НОД} (a, b)$ }

Воспользоваться свойством (4) легко, достаточно написать

выбор

. при $A = 0$ делай $x \leftarrow |B|$

. при $B = 0$ делай $x \leftarrow |A|$

конец_выбора

Теперь посмотрим, как мы можем менять A и B . Согласно (1)–(3), мы имеем три возможных действия: изменить знак у A или у B (запись: $A \leftarrow -A$, $B \leftarrow -B$);

уменьшить A на величину, равную B (запись: $A \leftarrow A - B$);

уменьшить B на величину, равную A (запись: $B \leftarrow B - A$).

Мы можем применять их сколько угодно, но нужно, чтобы они в каком-то смысле приближали нас к цели, т. е. чтобы рано или поздно оказалось, что $A = 0$ или $B = 0$. Для достижения этого применим такую стратегию: сначала сделаем A и B неотрицательными, пользуясь (1), а затем будем вычитать из большего меньшее (или равное). Наша схема видоизменится, приобретя вид

.....
{НОД (A , B) = НОД (a , b)}

сделать A и B неотрицательными, не меняя НОД, с помощью (1)

{НОД (A , B) = НОД (a , b), $A \geq 0$, $B \geq 0$ }

пока $A \neq 0$ и $B \neq 0$ повторять

. {НОД (A , B) = НОД (a , b), $A > 0$, $B > 0$ }

. применить (2) или (3)

конец_пока

{НОД (A , B) = НОД (a , b), $A = 0$ или $B = 0$ }

.....

Сделать A и B неотрицательными, не меняя НОД, легко:

{НОД (A , B) = НОД (a , b)}

выбор

. при $A \geq 0$ и $B \geq 0$ делай ничего не делать

. при $A \geq 0$ и $B \leq 0$ делай $B \leftarrow -B$

. при $A \leq 0$ и $B \geq 0$ делай $A \leftarrow -A$

. при $A \leq 0$ и $B \leq 0$ делай $A \leftarrow -A$, $B \leftarrow -B$
 конец выбора
 {НОД $(A, B) = \text{НОД}(a, b)$, $A, B \geq 0$ }

Сложнее понять, как применять (2) и (3). У нас есть две возможные команды:

$A \leftarrow A - B$ и $B \leftarrow B - A$.

Мы хотим, чтобы A и B оставались неотрицательными. Таким образом, мы можем выполнять $A \leftarrow A - B$ при $A \geq B$, а $B \leftarrow B - A$ при $B \geq A$. Получаем вот что:

{НОД $(A, B) = \text{НОД}(a, b)$, $A, B > 0$ }

выбор

. при $A \geq B$ делай $A \leftarrow A - B$

. при $B \geq A$ делай $B \leftarrow B - A$

конец_выбора

{НОД $(A, B) = \text{НОД}(a, b)$, $A, B \geq 0$ }

Убедимся, что указанное действие приближает нас к цели и что рано или поздно одно из чисел A и B станет равным 0. В самом деле, каждый раз одно из целых чисел A и B уменьшается, оставаясь неотрицательным, а другое остается неизменным, так что их сумма уменьшается, оставаясь неотрицательной. Это не может повторяться бесконечно много раз!

Итак, мы построили следующий алгоритм для отыскания НОД:

А л г о р и т м Е в к л и д а

{ a, b — целые числа, одно из которых не равно 0}

$A \leftarrow a$, $B \leftarrow b$

{НОД $(A, B) = \text{НОД}(a, b)$ }

выбор

. при $A \geq 0$ и $B \geq 0$ делай ничего_не_делать

. при $A \geq 0$ и $B \leq 0$ делай $B \leftarrow -B$

. при $A \leq 0$ и $B \geq 0$ делай $A \leftarrow -A$

. при $A \leq 0$ и $B \leq 0$ делай $A \leftarrow -A$, $B \leftarrow -B$

конец_выбора

{НОД $(A, B) = \text{НОД}(a, b)$, $A, B \geq 0$ }

пока $A \neq 0$ и $B \neq 0$ повторять

. {НОД $(A, B) = \text{НОД}(a, b)$, $A, B > 0$ }

. выбор

. . при $A \geq B$ делай $A \leftarrow A - B$

. . при $B \geq A$ делай $B \leftarrow B - A$

. . конец_выбора

. {НОД $(A, B) = \text{НОД}(a, b)$, $A, B \geq 0$ }

конец_пока

{НОД $(A, B) = \text{НОД}(a, b)$, $A, B \geq 0$, $A = 0$ }

```

или  $B = 0$ 
выбор
. при  $A = 0$  делай  $x \leftarrow B$ 
. при  $B = 0$  делай  $x \leftarrow A$ 
конец выбора
 $\{x = \text{НОД}(a, b)\}$ 

```

Замечание. В последней команде выбора мы опустили знаки модулей (см. выше аналогичный фрагмент), так как перед ее выполнением числа A и B неотрицательны.

Хороший алгоритм, как и хорошее доказательство, часто позволяет получить больше, чем ожидалось. В данном случае мы получаем доказательство такой теоремы из элементарной теории чисел:

Т е о р е м а. Если $x = \text{НОД}(a, b)$, то существуют такие целые p и q , что $x = pa + qb$.

Как говорят, $\text{НОД}(a, b)$ может быть выражен через a и b с целыми коэффициентами p и q .

С л е д с т в и е. Если d — любой общий делитель a и b , то $\text{НОД}(a, b)$ делится на d . (В самом деле, если a и b делятся на d , то и $pa + qb$ делится на d .)

Д о к а з а т е л ь с т в о т е о р е м ы. Докажем, что величины A и B в любой момент выполнения алгоритма выражаются через a и b , т. е. что

$$A = ka + lb, \quad B = ma + nb$$

при некоторых целых k, l, m, n . Проще всего сделать это, дополнив команды нашего алгоритма командами, вычисляющими k, l, m, n :

```

 $A \leftarrow a, B \leftarrow b$  [ $k \leftarrow 1, l \leftarrow 0, m \leftarrow 0, n \leftarrow 1$ ]
выбор
. при  $A \geq 0$  и  $B \geq 0$  делай ничего_не_делай
. при  $A \geq 0$  и  $B \leq 0$  делай  $B \leftarrow -B$ 
.   [ $m \leftarrow -m, n \leftarrow -n$ ]
. при  $A \leq 0$  и  $B \geq 0$  делай  $A \leftarrow -A$ 
.   [ $k \leftarrow -k, l \leftarrow -l$ ]
. при  $A \leq 0$  и  $B \leq 0$  делай  $B \leftarrow -B, A \leftarrow$ 
.    $-A$  [ $m \leftarrow -m, n \leftarrow -n, k \leftarrow -k, l \leftarrow -l$ ]
конец выбора
пока  $A \neq 0$  и  $B \neq 0$  повторять
. выбор
. . при  $A \geq B$  делай  $A \leftarrow A - B$ 
. .   [ $k \leftarrow k - m, l \leftarrow l - n$ ]
. . при  $A \leq B$  делай  $B \leftarrow B - A$ 
. .   [ $m \leftarrow m - k, n \leftarrow n - l$ ]

```


| . **конец_выбора**
 конец_пока

Вначале $A = a = 1a + 0b$, $B = b = 0a + 1b$, что соответствует значениям $k = 1$, $l = 0$, $m = 0$, $n = 1$. При изменении знака A для сохранения наших равенств надо изменить знак у k и l , при изменении знака B — у m и n . При замене A на $A - B$ мы пользуемся тем, что из $A = ka + lb$, $B = ma + nb$ путем вычитания получается

$$A - B = (k - m)a + (l - n)b,$$

аналогично для замены B на $B - A$. Итак, в конце выполнения приведенной части алгоритма наши равенства будут выполнены. Остается последняя команда:

| **выбор**
 . при $A = 0$ делай $x \leftarrow B$ [$p \leftarrow m$, $q \leftarrow n$]
 . при $B = 0$ делай $x \leftarrow A$ [$p \leftarrow k$, $q \leftarrow l$]
 конец_выбора

Добавив к ней то, что написано в квадратных скобках, видим, что после ее выполнения станет верным равенство $x = pa + qb$ (если $x = B$, то в качестве p и q надо взять m и n , а если $x = A$, то k и l).

Доказательство теоремы закончено.

В качестве бесплатного приложения мы получили алгоритм нахождения таких p и q , что $pa + qb = \text{НОД}(a, b)$. Это не так мало: попробуйте, например, решить уравнение $13x + 21y = 1$ без такого алгоритма!

У п р а ж н е н и е. Доказать, что можно уплатить кассиру любое целое число рублей, если у вас и у него имеется неограниченное количество 13-рублевых и 21-рублевых купюр. Указание: как уплатить 1 рубль?

ГЛАВА 6

КТО ТЯЖЕЛЕЕ, ИЛИ НИЖНИЕ И ВЕРХНИЕ ОЦЕНКИ ДЛЯ ЗАДАЧИ СОРТИРОВКИ

Пусть имеется n одинаковых с виду, но различных по весу камней. Необходимо расположить их в порядке возрастания весов, имея чашечные весы без гирь, на каждую чашку которых помещается по одному камню. При этом желательно сделать как можно меньше взвешиваний.

Первый приходящий в голову способ, вероятно, такой. Выберем самый легкий из камней. Затем выберем самый легкий из оставшихся и так далее. Подсчитаем, сколько взвешиваний будет сделано при этом. Покажем, что для выделения самого легкого из k камней достаточно $k - 1$ взвешивания. В самом деле, при $k = 1$ никаких взвешиваний делать не нужно, при $k = 2$ достаточно одного взвешивания. Поэтому при k , равном 1 или 2, наше утверждение верно. Покажем, что оно сохраняет истинность при увеличении k на 1 и тем самым верно при всех натуральных k (этот способ рассуждений называется рассуждением по индукции). Предположим, что для нахождения самого легкого из k камней достаточно $k - 1$ взвешивания, и покажем, что для нахождения самого легкого из $k + 1$ камня достаточно k взвешиваний. В самом деле, нужно выбрать самый легкий из k камней за $k - 1$ взвешивание и последним взвешиванием сравнить выбранный камень и оставшийся. Тем самым мы найдем самый легкий из $k + 1$ камня за k взвешиваний. Это рассуждение может показаться излишне сложным. Тем не менее мы привели его, так как в более сложных случаях подобные индуктивные рассуждения (переход от k к $k + 1$) нам пригодятся.

Итак, подсчитаем общее количество операций при упорядочении по весу (будем говорить кратко — сортировке) n камней нашим способом. Оно равно

$$\begin{array}{l} (n - 1) + (n - 2) + \dots + 1 \\ \left| \begin{array}{l} \text{нахождение} \quad \text{нахождение легчайшего} \\ \text{легчайшего} \quad \text{из оставшихся} \end{array} \right. \end{array}$$

Подсчитав эту сумму, находим, что она равна $n(n - 1)/2$. Таким образом, число взвешиваний при сортировке n камней по нашему способу примерно равно половине n^2 (при больших n слагаемым (-1) можно пренебречь). Скорость роста этого числа можно наглядно представить себе так: при увеличении числа камней в 2 раза число взвешиваний растет примерно в 4 раза.

Постараемся улучшить наш способ сортировки. Это можно сделать двояко. Можно либо изменить сам способ сортировки, либо научиться быстрее отыскивать самый легкий из камней, сделав меньшее число попарных сравнений. Второе на самом деле невозможно (попробуйте доказать!), так что придется менять способ сортировки.

Представим себе, что l камней уже рассортированы по весу и мы хотим добавить к ним еще один камень, поместив его на место, соответствующее его весу. Самый простой способ — сравнить новый камень со всеми старыми камнями — требует l взвешиваний. Если мы будем производить сортировку таким способом (добавляя все время по одному камню на надлежащее место после сравнения со всеми старыми), то число $T(n)$ взвешиваний, необходимых для сортировки n камней, можно найти по формулам

$$\left| \begin{array}{lll} T(n+1) & = & T(n) + n \\ \text{сортировка} & \text{сортировка} & \text{добавление} \\ n+1 \text{ камня} & n \text{ камней} & \text{еще одного} \end{array} \right.$$

Отсюда $T(n) = 1 + 2 + \dots + (n-1)$, как и раньше. Так что пока мы не получили никакого выигрыша.

Однако при таком способе сортировки мы не исчерпали всех резервов экономии. Именно, для помещения нового камня среди уже отсортированных не обязательно сравнивать его по весу со всеми старыми. Лучше выберем из старых камней средний по весу (или один из двух средних, если камней четное число) и сравним новый камень с ним. В зависимости от результата взвешивания нужно затем будет найти место нового камня среди первой или второй половины. Если обозначить через $U(k)$ число взвешиваний, необходимых для размещения камня среди k уже отсортированных при таком способе, то

$$U(1) = 1,$$

$$\left| \begin{array}{ll} U(k) \leq 1 & + \quad U(\lfloor k/2 \rfloor) \\ \text{сравнение} & \text{размещение} \\ \text{со средним} & \text{среди оставшихся} \end{array} \right.$$

(здесь $[x]$ обозначает целую часть x — наибольшее целое число, не превосходящее x .) Таким образом, при увеличении k вдвое величина $U(k)$ увеличивается на 1. Получаем

$$k = 1 \Rightarrow U(k) = 1,$$

$$k \leq 2 \Rightarrow U(k) \leq 1 + U(1) = 2,$$

$$k \leq 4 \Rightarrow U(k) \leq 1 + U(2) \leq 3$$

.....

Сравнивая U с двоичным логарифмом k ($\log_2 a =$ такое b , что $2^b = a$), который увеличивается на 1 при воз-

растании аргумента вдвое, видим, что $U(k) \leq \log_2 k + 2$. Остается подсчитать общее число взвешиваний при сортировке n камней новым способом. Оно равно

$$U(1) + \dots + U(n-1) \leq nU(n) \leq n(\log_2 n + 2).$$

(Мы заменили все $U(k)$ на большее значение $U(n)$; при этом мы ошибемся лишь в большую сторону n , как можно убедиться, не очень сильно.) При больших n можно пренебречь двойкой по сравнению с $\log_2 n$ и сказать, что число взвешиваний при сортировке n камней примерно равно $n \log_2 n$. Поскольку (при больших значениях n) $\log_2 n$ много меньше $n/2$, то $n \log_2 n$ много меньше $n^2/2$, так что новый способ значительно лучше старого при таких n .

А нельзя ли придумать еще лучший способ? Нельзя ли упорядочить n камней существенно меньше, чем за $n \log_2 n$ действий? В некоторых случаях можно. Может оказаться, например, что каждый новый взятый нами камень тяжелее предыдущего. Убедившись в этом, мы тем самым упорядочим все камни по весу, сделав $n - 1$ взвешивание. Ясно, однако, что это — случайная удача, связанная с расположением камней. Чтобы обсудить ситуацию подробнее, введем такое

О п р е д е л е н и е. Пусть A — способ сортировки n камней. Сложностью A назовем число взвешиваний, необходимых для упорядочения n камней в наихудшем случае (при таком соотношении весов, когда это число наибольшее).

Дав это определение, можно поставить вопрос: какова минимально возможная сложность способа упорядочения n камней? Мы знаем способ со сложностью порядка $n \log_2 n$; может ли она быть существенно меньше?

Чтобы ответить на этот вопрос, введем понятие «дерева вариантов» для данного способа упорядочения. Пронумеруем для удобства все камни от 1 до n . Эти номера выбираются произвольно и никак не связаны с весом камней. Посмотрим, какие случаи могут встретиться при упорядочении камней по нашему способу. Пусть i и j — номера камней, которые взвешиваются первыми. Возможны два результата взвешивания; равновесие невозможно, так как, по предположению, все камни разного веса (рис. 8). В зависимости от результата взвешивания мы решаем, какие камни нужно взвешивать дальше, результат этого взвешивания определяет, что будет взвешиваться на следующем шаге,

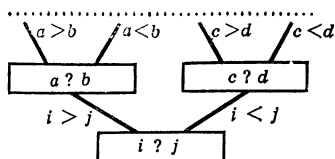


Рис. 8

и т. д. После нескольких взвешиваний получаем ответ (указание порядка камней). На рис. 9 изображено дерево вариантов для упорядочения трех камней (правую часть дорисуйте сами).

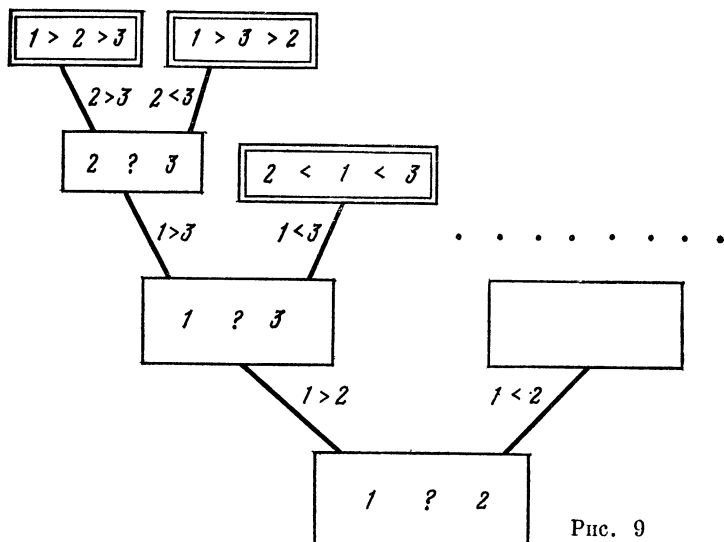


Рис. 9

Прямоугольники, нарисованные на картинке, называют вершинами дерева, нижнюю вершину — корнем, вершины, из которых больше ничего вверх не идет (прямоугольники, содержащие ответы), — листьями. В терминах дерева сложность способа сортировки соответствует высоте дерева вариантов — максимальному числу отрезков на пути, ведущем из корня в лист. Итак, нам нужно оценить высоту дерева вариантов для различных способов сортировки камней.

Прежде всего поймем, сколько имеется листьев в этом дереве. Их столько, сколько возможных ответов — возможных порядков расположения камней по

весу. Число возможных ответов для случая n камней равно $1 \cdot 2 \cdot \dots (n - 1) \cdot n$. Это число обозначают $n!$ (n -факториал). В самом деле, самым легким может оказаться любой из n камней. Каждый из этих n случаев делится на $n - 1$ подслучаев в зависимости от того, какой из оставшихся камней будет вторым по весу. Каждый из подслучаев делится на $n - 2$ подподслучаев и т. д. В итоге получаем $n!$ вариантов.

Покажем теперь, что дерево с большим числом листьев должно иметь не слишком маленькую высоту.

Л е м м а. Если дерево вариантов имеет высоту не больше h , то число листьев в нем не превосходит 2^h .

Д о к а з а т е л ь с т в о. Если бы все листья находились на высоте h , то их число было бы равно 2^h (на высоте 0 только корень, на высоте 1 есть две вершины... на высоте h есть 2^h вершин). Если же некоторые ветви дерева обрываются, не доходя до высоты h , от этого число листьев только уменьшается: заменив лист продолжением дерева до высоты h , мы увеличим число листьев. Лемма доказана.

Другими словами, если число листьев в дереве равно s , то его высота не может быть меньше $\log_2 s$. Применяя это к нашей задаче, где число листьев равно $n!$, находим, что высота дерева вариантов для сортировки n камней не может быть меньше $\log_2 (n!)$, или (мы пользуемся известным из курса алгебры свойством логарифма)

$$\log_2 (1 \cdot 2 \cdot \dots \cdot n) = \log_2 1 + \log_2 2 + \dots + \log_2 n.$$

Оставляя в этой сумме лишь правую половину слагаемых и заменяя их на $\log_2 (n/2)$, получаем, что высота дерева не меньше

$$(n/2) \cdot \log_2 (n/2) = (n/2) \cdot (\log_2 n - 1).$$

(Лучшая оценка получится, если воспользоваться формулой Стирлинга: $n!$ примерно равно $(n/2, 71828 \dots)$ в степени n). Итак, мы получили нижнюю оценку для сложности любого способа сортировки n камней, которая при больших n примерно в 2 раза меньше сложности нашего способа сортировки камней.

Разумеется, все сказанное можно применить не только к камням. Задача расположения слов в словаре или карточек в картотеке, задача составления списка лучших шахматистов в порядке убывания рейтинга (силы) и тому подобные — все это задачи того же типа. Построение алгоритмов сортировки, таким

образом, имеет большое значение. Мы еще вернемся к ним в главе 8 «Снова о сортировке».

У п р а ж н е н и я. (1) Придумать способ, как найти самый легкий и самый тяжелый из 60 камней не более чем за 100 взвешиваний.

(2) Эксперт хочет доказать суду, что самым легким из n камней является первый. Доказать, что ему потребуется не менее $n - 1$ взвешивания на чашечных весах без гирь (на каждую чашку помещается по камню). Веса камней неизвестны суду, но известны эксперту.

(3) Найти способ сортировки наименьшей сложности для 2, 3, 4, 5, 6, 7 камней.

ГЛАВА 7

СКОЛЬКО ВЕРЕВОЧКЕ НИ ВИТЬСЯ, ИЛИ ПОЧЕМУ ПРОГРАММЫ КОНЧАЮТ РАБОТУ

Некоторые слова, возникшие первоначально в связи с программированием, постепенно проникают в общетехнический жаргон и даже в разговорный язык. Одно из них — слово «зациклиться». В программировании оно употребляется по отношению к программе, не заканчивающей работу. Уже из этого видно, сколь важной задачей программиста является забота о том, чтобы написанные им программы заканчивали работу. В этой главе мы попытаемся продемонстрировать на простых примерах методы, используемые при доказательствах того, что работа программы завершается.

З а д а ч а. Имеется квадратная таблица, заполненная числами. Разрешается выполнять такие операции: (1) менять знаки всех чисел в одной строке; (2) менять знаки всех чисел в одном столбце. Доказать, что можно добиться того, чтобы сумма чисел в каждой из строк и в каждом из столбцов была бы неотрицательной.

Первая идея, приходящая в голову, совсем проста. Если в столбце сумма чисел отрицательна, то после перемены знака в этом столбце сумма чисел будет положительна. То же самое можно сказать и про строки. Таким образом, меняя знаки в строках и столбцах с отрицательной суммой, можно добиться желаемого. Сказанное можно условно записать так:

пока есть хоть один столбец или строка с отри-
 . цательной суммой
 . повторять
 . изменить знаки чисел в одном из таких столб-
 . цов или строк
 конец_пока

Ясно, что после завершения работы этой программы во всех строках и столбцах суммы будут неотрицательными. Однако не ясно, почему это завершение произойдет. Можно было бы, вообще говоря, опасаться такой ситуации: после изменения знака в строке становится отрицательной сумма чисел в каком-то столбце, после изменения знака в этом столбце становится отрицательной сумма в какой-то строке и т. д. Таким образом, нам осталось доказать, что наша программа не может работать неограниченно долго.

Ключевой идеей здесь является следующая: при замене знака в строке (в столбце) с отрицательной суммой сумма всех чисел таблицы возрастает (поскольку сумма в строке или столбце из отрицательной делается положительной, а остальные числа таблицы не меняются). С другой стороны, сумма является целым числом, не превосходящим суммы модулей всех чисел исходной таблицы. Таким образом, ее увеличение не может происходить неограниченно долго, и наша задача решена.

У п р а ж н е н и е. Решить ту же задачу, если числа в таблице — произвольные действительные (а не только целые).

В следующей задаче основная идея также состоит в выборе подходящей величины, которая может только возрасти или только убывать.

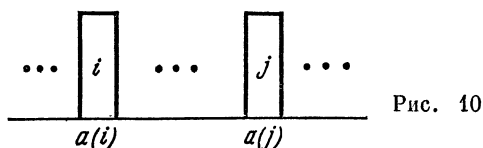
З а д а ч а. На полке в беспорядке стоят тома собрания сочинений. Хозяин, увидев два тома, стоящие в неправильном порядке (том с меньшим номером правее тома с большим номером), меняет их местами. Доказать, что рано или поздно собрание будет расставлено по порядку.

Р е ш е н и е. По аналогии с предыдущей задачей можно сформулировать доказываемое утверждение так: программа

пока есть два тома, стоящие в неправильном
 . порядке
 . повторять
 . переставить их
 конец_пока

заканчивает работу. Продолжая аналогию, введем величину, которая возрастает при каждой перестановке. Следуя голландскому программисту Э. Дейкстре, воспользуемся физической аналогией. Представим себе, что все тома имеют одинаковую толщину, но разный вес: чем больше номер тома, тем он тяжелее. В этом случае можно описать наши действия так: обнаружив более тяжелый том слева от более легкого, мы меняем их местами (не сдвигая с места остальные тома — все тома имеют одинаковую толщину). Легко видеть, что при этом центр тяжести всего собрания смещается вправо. Может ли такое происходить бесконечное число раз? Нет, так как число возможных положений центра тяжести конечно (оно не больше числа возможных расстановок томов) и центр тяжести не может все время сдвигаться вправо.

Если мы хотим превратить эту физическую аналогию в математическое доказательство, вспомним формулу для координат центра тяжести. Считаем, что вес i -го тома равен i . В этом случае в числителе выражения для координаты центра тяжести стоит величина $\sum ia(i)$, где $a(i)$ — положение i -го тома на полке. (Знаменатель — общая масса — константа, которая нас не волнует.) Посмотрим, что происходит с этой величиной при обмене i -го и j -го томов, стоящих на местах $a(i)$ и $a(j)$ соответственно (рис. 10).



Будем считать, что $i > j$, а $a(i) < a(j)$. Перестановка томов приведет к замене в нашей сумме слагаемых $ia(i) + ja(j)$

на

$$ja(i) + ia(j).$$

Покажем, что от этого сумма увеличится, т. е. что

$$ja(i) + ia(j) > ia(i) + ja(j)$$

или (переносим в другую часть и группируем)

$$(i - j)a(j) > (i - j)a(i).$$

А это так, потому что $i - j > 0$ и $a(j) > a(i)$.

Осталось заметить, что наша сумма является целым числом и не превосходит n^3 (слагаемых n , каждое не больше n^2). Следовательно, она не может возрасти неограниченно. Задача решена.

В двух рассматриваемых задачах нам удавалось построить монотонно меняющуюся величину с целочисленными значениями. Не всегда это возможно или удобно. Бывают задачи, где значения монотонно изменяющейся величины берутся из другого множества.

Задача (Э. В. Дейкстра). На сортировочной станции имеется несколько поездов. Разрешаются два типа действий: (1) расцепить поезд, состоящий из нескольких вагонов, на два поезда; (2) удалить поезд: это разрешается, если в нем всего один вагон. Доказать, что, выполняя эти действия в произвольном порядке, мы рано или поздно удалим все вагоны.

Соответствующая программа такова:

```
пока есть хоть один поезд повторять
.  выбрать какой-то поезд
.  выбор
.  .  при в нем один вагон делай
.  .  удалить его
.  .  при в нем больше одного вагона делай
.  .  расцепить его
.  конец_выбора
конец_пока
```

В этой задаче ни число поездов, ни число вагонов не годится на роль монотонно убывающей величины: число поездов может расти, если мы расцепляем поезд на два; число вагонов остается прежним, если мы расцепляем поезд.

Чтобы выйти из этого положения, введем такое определение. Будем говорить, что пара (a, b) натуральных чисел меньше пары (c, d) , если $a < c$ или $(a = c$ и $b < d)$. (Другими словами, мы сравниваем первые члены пар, а при равных первых — вторые). Пользуясь этим определением, можно сказать, что пара (число вагонов, число сцепок),

где число сцепок есть число межвагонных соединений, равное числу вагонов минус число поездов, убывает. При удалении поезда уменьшается число вагонов и пара становится меньше (в смысле нашего определения). При расцеплении поезда на два число вагонов не меняется, а число сцепок убывает и пара также уменьшается. Остается доказать лишь такую лемму.

Л е м м а. Не существует бесконечной убывающей последовательности пар натуральных чисел

$$(a_1, b_1) > (a_2, b_2) > \dots > (a_n, b_n) > \dots$$

Д о к а з а т е л ь с т в о. Последовательность a_1, a_2, \dots — невозрастающая (если первый член пары увеличивается, то и вся пара увеличивается). Поскольку члены этой последовательности — натуральные числа, то, начиная с некоторого места, все они равны (они не могут бесконечно число раз убывать). Начиная с этого места, вторые члены пар, согласно нашему определению, образуют (строго) убывающую последовательность натуральных чисел, а бесконечных строго убывающих последовательностей натуральных чисел не бывает. Лемма доказана.

Наша задача решена. Мы сознательно изложили решение задачи в такой абстрактной форме, чтобы общая схема его была возможно более ясной. Доказать лемму можно и сразу в ситуации с поездами. Пусть мы бесконечно много раз повторяем наши операции. Число вагонов при этом не возрастает, значит, в некоторый момент оно достигнет минимального значения. С этого момента вагоны со станции не удаляют, а только расцепляют поезда. Это, однако, тоже не может длиться бесконечно долго, так как число сцепок между вагонами конечно.

И последнее замечание об этой задаче. Ее можно было бы решить (хотя несколько искусственно) и старым способом, заметив, что величина

$$\text{число вагонов} + \text{число сцепок} =$$

удвоенное число вагонов — число поездов убывает при каждой операции, оставаясь натуральным числом. В следующей задаче такое уже не удастся.

З а д а ч а. Имеется конечная последовательность нулей и единиц, содержащая ровно две единицы. Разрешается заменять в ней группу символов 01 на группу 10...0 (число нулей любое). Доказать, что такие операции не могут выполняться бесконечно много раз.

Р е ш е н и е. Наши операции не меняют числа единиц, поэтому в любой момент в последовательности будет ровно 2 единицы. Рассмотрим пару

(число нулей перед первой единицей,

число нулей перед второй единицей).

При каждой замене группы 01 на 10...0 эта пара уменьшается. В самом деле, если заменяется группа, со-

держащая первую единицу, то уменьшается первый член пары (второй при этом может увеличиваться, но это не страшно); если заменяется группа со второй единицей, то первый член пары не меняется, а второй уменьшается. Тем самым, согласно доказанной лемме, этот процесс не может продолжаться бесконечно.

До сих пор мы исследовали вопрос о завершении работы программы, не обращая внимания на время, которое может пройти до этого завершения. На самом деле это время тоже существенно: программа, работа которой завершится через 10 тыс. лет, для нас так же неприемлема, как и программа, вовсе не завершающая работы.

Часто изложенные соображения позволяют не только доказать завершаемость, но и оценить число повторений и тем самым время работы. Именно, если при каждом повторении некоторая величина, оставаясь натуральным числом, убывает на 1, то число повторений не может быть больше первоначального значения этой величины. Такого рода соображения позволяют увидеть, что в задаче про таблицу число повторений не превосходит суммы модулей всех чисел таблицы, а в задаче про собрание сочинений — куба числа томов. На самом деле эти оценки сильно завышены, но более точные оценки можно получить аналогичными методами. О времени работы программы говорится также в главе 14 «Переборные задачи».

У п р а ж н е н и я.

1. По команде «напра-во» новобранцы, стоящие в шеренгу, поворачиваются кто куда. Далее каждый из них действует так: увидев перед собой лицо соседа, он через секунду поворачивается кругом. Доказать, что рано или поздно повороты прекратятся.

2. Как решить задачу о последовательностях нулей и единиц, если в первоначальной последовательности было произвольное число единиц?

3. Доказать, что программа

пока $n > 0$ повторять

. **выбор**

. . **при n четно делай** разделить n пополам

. . **при n нечетно делай** уменьшить n на 1

конец_выбора

конец_пока

при любом натуральном n заканчивает работу. Сколько повторений может потребоваться, если n не превосхо-

дит миллиарда? (Указание. Рассмотрите две величины: (1) n ; (2) $2 \lfloor \log_2 n \rfloor +$ (остаток от деления n на 2). Квадратные скобки обозначают целую часть.)

4. Доказать, что в задаче о собрании сочинений число перестановок не превосходит квадрата числа томов. (Указание: рассмотреть величину, равную числу пар томов, стоящих в неправильном порядке).

5. Скупой рыцарь заключил с чертом следующее соглашение. Каждый день он обязуется отдавать черту по монете. Взамен нее он может получить любое число монет меньшего достоинства. Доказать, что рано или поздно у него все монеты кончатся (если он не может получать или менять деньги в каком-либо другом месте).

6. Вы помещаете на отрезок точку, затем вторую — так, чтобы получившиеся две точки были в разных половинах отрезка, затем третью — так, чтобы получившиеся три точки были в разных третях отрезка и так далее. Может ли это продолжаться бесконечно долго? (Эту задачу было бы интересно попытаться решить с помощью компьютера.)

ГЛАВА 8

СНОВА О СОРТИРОВКЕ

В главе 6 «Кто тяжелее, или Нижние и верхние оценки для задачи сортировки» мы занимались вопросом о том, сколько взвешиваний на чашечных весах без гирь необходимо, чтобы упорядочить по возрастанию весов n предметов. Было установлено, что необходимое число взвешиваний пропорционально $n \log_2 n$ и предложен соответствующий способ. В этом разделе мы предложим еще несколько способов сортировки, которые в некоторых отношениях удобнее.

Для разнообразия вместо камней будем сортировать библиотечные карточки (рис. 11), стремясь расположить их в алфавитном порядке. Будем при этом считать, что сравнение двух карточек и выяснение того, какая из них идет раньше в алфавитном порядке, является элементарной операцией. В отличие от взвешивания камней эта задача вполне практическая и встречается довольно часто; неумелое ее решение может стоить многих часов напрасного труда.

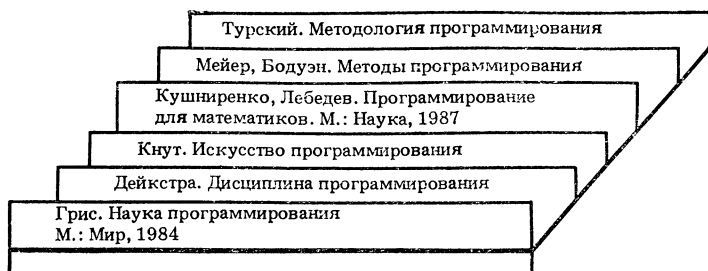


Рис. 11

Рассмотренный нами в главе 6 метод сортировки можно изложить так. Нужно добавлять карточки по одной, вставляя каждую на нужное место. Важно только правильно отыскивать это место. Если мы будем просматривать каждый раз подряд все карточки, начиная с самого начала, пока не отыщем нужное место, то это займет много времени. Нетрудно сообразить, что число действий в худшем случае будет приблизительно пропорционально квадрату числа карточек: для картотеки из 1000 карточек (а это совсем небольшой ящик) число операций будет порядка сотен тысяч. Если мы не хотим заниматься одним ящиком целую неделю, нужно придумать лучший способ. Мы видели, что для нахождения нужного места в группе из n карточек достаточно около $\log_2 n$ действий: сравнив новую карточку со средней, определим, в какой она половине, затем сравним ее со средней карточкой этой половины и т. д.

При этом способе число сравнений пропорционально $n \log_2 n$, что близко к теоретическому пределу. Однако сам способ довольно неудобен, поскольку, помимо сравнений, он требует большого числа других действий (нужно отыскивать среднюю карточку в куче, затем среднюю карточку в одной из половинок и т. д.). Расскажем о некоторых других, во многих отношениях более удобных способах сортировки.

Изложим два таких способа. Оба они используют некоторую общую идею, оказывающуюся полезной при построении алгоритмов. За ней закрепилось условное название «разделяй и властвуй» («divide et impera»), а смысл ее состоит в том, что задача сводится к нескольким задачам того же типа, но меньшего раз-

мера. (Некоторые применения этой идеи встретятся и в других главах книги.) В нашем случае мы сведем сортировку большого набора карточек к сортировке двух меньших.

При этом возможны два подхода. При первом из них делим кучу подлежащих сортировке карточек пополам, не глядя на то, что написано на карточках. После этого сортируем по отдельности каждую из половин. Остается лишь соединить две отсортированные половин. Это сделать просто. Положим перед собой две кучи карточек и будем брать сверху по одной карточке — той, которая идет раньше в алфавитном порядке. Из взятых карточек образуется стопка (очередную карточку мы добавляем в ее конец) расположенных в алфавитном порядке карточек, в которой в конце концов и окажутся все карточки.

Опишем этот процесс слияния более подробно:

```
пока одна из (старых) куч не пуста повторять
.  выбор
.  .  при обе кучи не пусты делай
.  .  взять ту из верхних карточек, которая идет
.  .  раньше в алфавитном порядке
.  .  при одна из куч пуста делай
.  .  взять верхнюю карточку из другой кучи
.  конец_выбора
.  добавить взятую карточку в конец
конец_пока
```

После каждого выполнения команд, входящих в команду `пока... конец_пока`, справедливо следующее свойство:

И н в а р и а н т : (1) все три кучи упорядочены по алфавиту, причем (2) любая карточка новой кучи предшествует в алфавитном порядке любой карточке из старых куч.

Убедимся в этом. Вначале это свойство очевидно выполняется (обе старые кучи упорядочены по алфавиту, а новая куча пуста). Проверим, что оно не нарушается после каждого цикла команды повторения. В самом деле, взятая карточка, будучи первой в алфавитном порядке среди первых карточек каждой из куч, предшествует всем остальным карточкам этих куч, так что ее перемещение в новую кучу не нарушает свойства (2). С другой стороны, она, как и всякая карточка из двух старых куч, следовала по алфавиту за всеми карточками новой кучи, поэтому не нарушится

и свойство (1). (В случае, когда одна из куч пуста, рассуждения аналогичны.)

Таким образом, инвариант сохраняется и после окончания работы программы. (Рано или поздно конец наступит, так как общее количество карточек в старых кучах неуклонно убывает.) После этого все карточки будут находиться в новой куче в алфавитном порядке, что и требовалось.

Оценим число сравнений, необходимых для сортировки этим способом кучи из n карточек. Будем считать для простоты, что n есть степень числа 2 и каждый раз мы делим кучу на две равные части. При слиянии двух куч из m карточек в одну кучу из $2m$ карточек необходимо не более $2m$ сравнений (каждое выполнение команд, входящих в команду повторения, требует не более одного сравнения и добавляет одну карточку к новой куче). Поэтому число сравнений, необходимых для сортировки кучи из n карточек, удовлетворяет неравенству

$$T(n) \leq 2T(n/2) + n$$

(сортировка n карточек = 2 раза · (сортировка $n/2$ карточек) + соединение).

При этом, очевидно, $T(1) = 0$ (сортировка одной карточки не требует сравнений). Получаем

$$T(2) \leq 2T(1) + 2 = 2,$$

$$T(4) \leq 2T(2) + 4 \leq 4 + 4 = 8,$$

$$T(8) \leq 2T(4) + 8 \leq 16 + 8 = 24,$$

$$T(16) \leq 2T(8) + 16 \leq 48 + 16 = 64$$

.....

и т. д. Покажем, что при всех m

$$T(2^m) \leq m \cdot 2^m.$$

В самом деле, при $m = 1$ это верно, поэтому достаточно доказать, что это свойство не нарушается (программисты сказали бы: остается инвариантным) при переходе от $m - 1$ к m . Если

$$T(2^{m-1}) \leq (m - 1) \cdot 2^{m-1},$$

то

$$\begin{aligned} T(2^m) &\leq 2T(2^{m-1}) + 2^m \leq 2 \cdot (m - 1) \cdot 2^{m-1} + \\ &+ 2^m = (m - 1) \cdot 2^m + 2^m = m \cdot 2^m. \end{aligned}$$

Таким образом, при $n = 2^m$ имеем

$$T(n) \leq m \cdot 2^m = n \log_2 n.$$

При n , не являющемся степенью двойки, можно применить, по существу, тот же способ, деля кучу на две «почти равные» (отличающиеся по числу карточек не более чем на 1). А можно добавить в кучу несколько «фиктивных» карточек, написав на них какое-нибудь слово, которое идет в алфавитном порядке после всех других слов, написанных на карточках, с тем чтобы общее число карточек стало степенью двойки. И в том, и в другом случае мы получим близкую к $n \log_2 n$ оценку для числа сравнений.

Мы не будем сейчас приводить полную и довольно сложную программу для сортировки карточек изложенным способом. Вместо этого приведем эскизное описание еще одного способа сортировки.

Идея его очень проста: мы можем ликвидировать необходимость в слиянии двух отсортированных куч, если будем знать, что все карточки в первой куче идут раньше в алфавитном порядке, чем все карточки во второй. Тем самым мы добьемся экономии, поставившись разделить карточки на две кучи так, чтобы указанное свойство выполнялось. Сделать это довольно просто: нужно выбрать одну из карточек и использовать ее как границу: все предшествующие ей карточки отнести в одну кучу, а все следующие за ней — в другую (ее саму можно отнести к любой из куч).

В этом способе есть, однако, некоторая опасность. При неудачном стечении обстоятельств может оказаться так, что кучи, на которые мы поделим исходную, будут сильно различаться по величине (крайний случай: карточка, выбранная в качестве границы, окажется первой или последней по алфавиту). Можно надеяться, что это будет случаться редко и что «в среднем» предлагаемый способ будет достаточно эффективен (утверждение об эффективности метода «в среднем» можно уточнить, превратив его в теорему: см. книгу Ахо, Хопкрофта, Ульмана «Построение и анализ вычислительных алгоритмов», о которой идет речь в последней главе). Если оценка «в среднем» нас не устраивает, можно попытаться улучшить сам метод сортировки.

Не будем, однако, гнаться за такими уточнениями;

нашей целью было продемонстрировать, каким образом принцип «разделяй и властвуй» может служить источником идей при разработке алгоритмов.

Отметим в заключение, что достоинством описанных способов сортировки с помощью принципа «разделяй и властвуй» (особенно при «ручном» выполнении) является то, что каждую из куч, на которые разбивается исходная куча, можно сортировать независимо. Благодаря этому большую стопку карточек можно отсортировать быстро, если у вас достаточно друзей, готовых вам помочь. К сожалению, современное состояние науки программирования не позволяет использовать это достоинство в полной мере — мы не владеем как следует языком, на котором можно было бы писать программы для нескольких исполнителей сразу. Однако в будущем указанное достоинство рассмотренных методов сортировки может оказаться важным.

ГЛАВА 9

МОГУТ ЛИ ВОСЕМЬ ФЕРЗЕЙ НЕ БИТЬ ДРУГ ДРУГА, ИЛИ ОБХОД ДЕРЕВА

Шахматная головоломка

Исходным пунктом нашего обсуждения будет такая головоломка: могут ли восемь ферзей (на шахматной доске 8×8) не бить друг друга? Более точно — требуется найти все расстановки восьми ферзей, в которых никакие два ферзя не стоят на одной вертикали, горизонтали или диагонали (если такие позиции вообще существуют).

Прежде чем обсуждать вопрос о написании программы, решающей эту задачу, расскажем (следуя книжке Е. Я. Гика «Математика на шахматной доске») об истории ее решения без машины. Задача была поставлена в 1848 г. немецким шахматистом М. Беццелем. В июне 1850 г. Ф. Наук опубликовал 60 решений. Великому математику К. Гауссу удалось найти 72 решения. Однако вскоре его результат перекрыл тот же Наук, нашедший 92 решения. В 1874 г. английский математик Д. Глэшер доказал, что больше решений не существует.

Вернемся к нашей головоломке. Легко заметить, что на каждой вертикали должен стоять один ферзь, поскольку вертикалей столько же, сколько ферзей, а никакие два ферзя не должны стоять на одной вертикали. Аналогичным образом на каждой горизонтали должен стоять один ферзь. Если бы у нас были не ферзи, а ладьи, то эти условия были бы достаточными, чтобы ладьи не били друг друга. Знакомые с комбинаторикой легко сообразят, что в случае ладьей существует $8! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 8$ решений. Действительно, на каждой горизонтали должно стоять ровно по одной ладье. У ладьи, стоящей на первой горизонтали, есть выбор из восьми полей; у ладьи, стоящей на второй горизонтали, есть выбор из семи полей — одно из полей уже пробито ладьей, стоящей на первой горизонтали, и т. д. Когда мы доходим до восьмой горизонтали, остается лишь одно разрешенное («непробитое») поле. Получаем всего $8 \cdot 7 \cdot \dots \cdot 2 \cdot 1$ расстановок.

В нашем случае (когда расставляют не ладей, а ферзей) существенны и диагонали, так что задача, видимо, сложнее. Попытаемся перебрать все варианты и найти среди них искомые. Приняв такое решение, сталкиваемся со следующими вопросами: нам нужно быть уверенными, что в ходе перебора мы не пропустим ни одного решения головоломки. Кроме того, было бы желательно не рассматривать одну и ту же позицию многократно, чтобы по возможности избежать лишней работы. Таким образом, нам нужно составить алгоритм (план) перебора, позволяющий достигнуть этих целей. Честно говоря, это и составляет основную тему данной главы, а задача о ферзях является лишь поводом. Поэтому, составив интересующий нас алгоритм, мы не будем пытаться его выполнить ни вручную, ни с помощью компьютера.

Итак, мы хотим перебрать и испробовать все варианты, чтобы быть уверенными, что ни одного решения головоломки мы не пропустили. Однако выражение «все варианты» не ясно: хотим ли мы рассматривать все расстановки ферзей на доске? или все расстановки восьми ферзей? или все расстановки, где на каждой горизонтали стоит по ферзю? или еще что-нибудь?

Мы должны, таким образом, определить, какие конфигурации будут рассматриваться в качестве «кандидатов» в решении нашей головоломки. Здесь

есть две опасности. Первая состоит в том, что кандидатов окажется много. А ведь чем их больше, тем более трудоемким окажется перебор. Этой опасности можно было бы избежать, считая кандидатами расстановки восемь ферзей, в которых никакие два ферзя не бьют друг друга. Тогда количество кандидатов, разумеется, минимально, но не ясно, как их перебирать: уметь их пере-

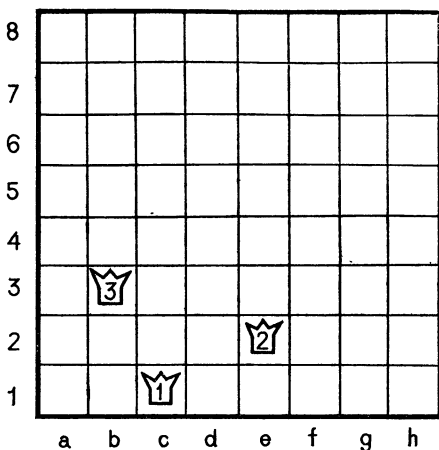


Рис. 12

ребирать означает уметь решать исходную головоломку.

Мы будем представлять себе позицию на доске как результат появления на доске ферзей друг за другом. Поскольку порядок появления ферзей безразличен, и на каждой горизонтали должно стоять по ферзю, можно представлять себе, что ферзей ставят снизу вверх — сначала ставится ферзь на первую горизонталь, затем на вторую и т. д. (рис. 12). Может оказаться, что ферзь, поставленный последним, попадает под бой уже стоящих на доске. В этом случае наша позиция бесперспективна и ставить ферзей дальше смысла нет.

Возникает «дерево вариантов», изображенное на рис. 13 (см. стр. 68). Внизу, в его корне, находится пустая позиция — позиция, в которой нет ни одного ферзя. Выше нее нарисованы все позиции, которые могут получиться, когда мы поставим ферзя на первую горизонталь. Из каждой такой позиции можно получить восемь позиций, поставив следующего ферзя на одну из клеток второй горизонтали. Например, из позиции ($a1$), как показано на рисунке, можно получить позиции ($a1, a2$), ($a1, b2$), . . . , ($a1, h2$). В некоторых из них второй ферзь попадает под бой первого: в позиции ($a1, a2$) он оказывается на той же вертикали, а в позиции ($a1, b2$) — на той же диагонали, что и первый. Эти позиции являются бесперспек-

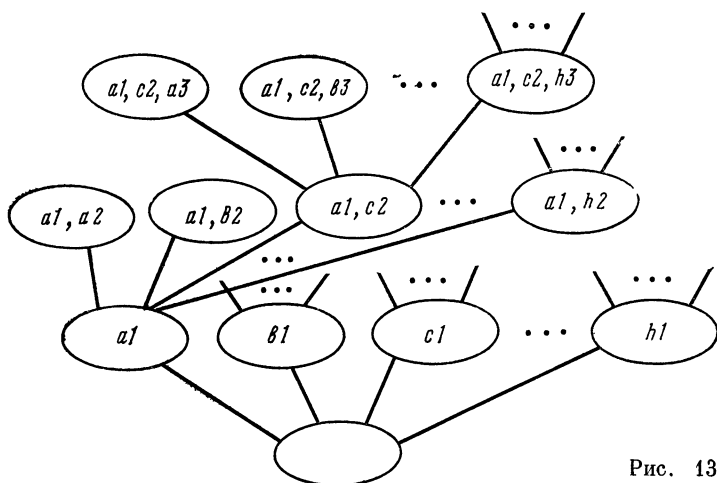


Рис. 13

тивными кандидатами — ставить еще одного ферзя нет смысла. В остальных позициях мы можем поставить третьего ферзя на одно из полей третьей горизонтали. Например, из позиции $(a1, c2)$ получатся позиции $(a1, c2, a3)$, $(a1, c2, b3)$, ..., $(a1, c2, h3)$. Из них четыре первые являются бесперспективными в том смысле, что новый ферзь попадает под бой старых. Из остальных четырех можно получить по восемь новых позиций, поставив ферзя на четвертую горизонталь, и т. д.

Другими словами, дерево вариантов строится следующим образом. Внизу рисуем пустую позицию. Далее применяется такое правило: над каждой уже имеющейся в дереве позицией, в которой ферзи не бьют друг друга и есть свободная горизонталь, рисуем (если это еще не сделано) восемь других, соответствующих восьми возможным положениям ферзя на нижней из свободных горизонталей, и соединяем их с ней линиями. Таким образом, «листьями» нашего дерева (позициями, из которых не выходит линий вверх) будут, во-первых, бесперспективные позиции (в которых ферзи бьют друг друга) и, во-вторых, решения нашей головоломки, т. е. позиции, в которых все горизонтали заполнены ферзями и они друг друга не бьют.

Теперь нашу задачу можно сформулировать так: нужно «обойти» все дерево и отобрать среди входящих

в него позиций те, которые являются решениями нашей головоломки. На время оставим нашу головоломку и займемся более общей (и более простой) задачей — задачей обхода произвольного дерева.

Обход дерева

Представим себе, что на плоскости нарисована точка — «корень дерева». Из нее выходит вверх несколько отрезков, соединяющих ее с несколькими другими точками. Из каждой из этих точек выходит вверх несколько отрезков и т. д. (рис. 14). Общее число точек конечно.

Договоримся о терминологии. Точки будем называть вершинами. Среди вершин выделяется корень (самая нижняя точка) и листья (те вершины, из которых не идут отрезки вверх). Верхним соседом вершины x называется вершина, находящаяся выше x и соединенная с x отрезком. Если вершина y является верхним соседом вершины x , то вершину x называют нижним соседом вершины y . Таким образом, у всех вершин, кроме корня, есть ровно один нижний сосед, а у корня нижнего соседа нет. Верхние соседи есть у всех вершин, кроме листьев. Для вершин, являющихся верхними соседями одной вершины x , введем отношение «быть правым соседом»: у каждой из них, кроме самой правой, есть ровно один правый сосед — ближайшая справа вершина. Подчеркнем, что у любой вершины и ее правого соседа (если он есть) один и тот же нижний сосед. На рис. 15 в соответствии с определениями отрезки соединяют вершину с ее

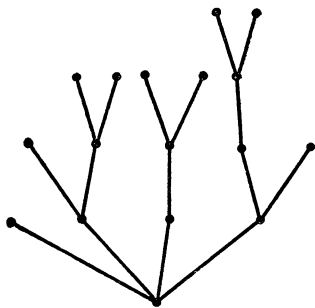


Рис. 14

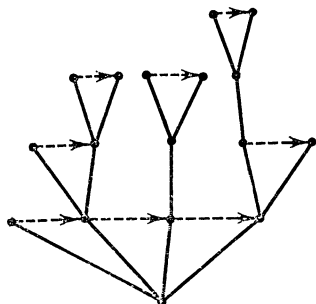


Рис. 15

верхними и нижними соседями; пунктирные стрелки проведены от вершины к правому соседу (если он есть).

Исполнитель, для управления которым мы будем писать программу, умеет выполнять следующие проверки и команды:

проверки

есть_сверху вершина, в которой находится исполнитель, имеет верхних соседей, т. е. не является листом;

есть_справа есть правый сосед;

есть_снизу есть нижний сосед, т. е. исполнитель не находится в корне;

команды

в_корень исполнитель попадает в корень дерева;

вверх_налево перемещается в вершину, являющуюся самым левым из верхних соседей;

вправо перемещается в правого соседа;

вниз перемещается в соседа снизу;

обработать исполнитель красит вершину, в которой находится, в зеленый цвет.

Отметим, что команды выполнимы не всегда. Проверки «есть_сверху», «есть_справа» и «есть_снизу» соответствуют условиям выполнимости команд «вверх_налево», «направо» и «вниз». На рис. 16 все возмож-

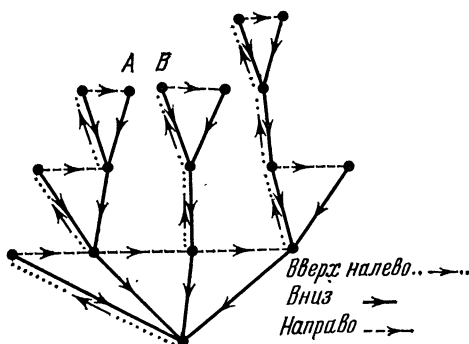


Рис. 16

ные движения по командам указаны линиями соответствующего типа.

Описав возможности нашего исполнителя, формулируем задачу: требуется придумать алгоритм,

заставляющий исполнителя обработать все листья дерева по одному разу.

Поскольку мы хотим красить листья (вершины, не имеющие соседей сверху) по одному разу, нам неизбежно придется делать это в каком-то порядке. В каком? По-видимому, проще всего это делать слева направо. Тем более что и система команд исполнителя предусматривает команду вправо, но не предусматривает команды влево. Таким образом, мы должны отправиться в самый левый лист и, начав обработку с него, двигаться направо. Мы не всегда можем, однако, двигаться только по листьям: чтобы попасть из одного листа в другой, может оказаться необходимым спуститься вниз и затем подняться вверх.

Чтобы прояснить ситуацию, введем некоторые обозначения. Пусть x — произвольная вершина дерева. Все листья дерева разобьем на три категории: находящиеся левее x , находящиеся над x и находящиеся правее x . Соответствующие множества листьев обозначим Лев (x), Над (x), Прав (x) (рис. 17). К типу

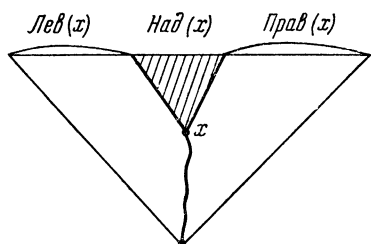


Рис. 17

«над x » мы отнесем те листья, в которые можно попасть из вершины x , идя вверх по стрелкам. К типу «левее x » отнесем листья, находящиеся левее листьев типа «над», а к типу «правее x » — листья, находящиеся правее. Другими словами, для определения типа листа нужно провести путь по дереву, ведущий в него из корня. Если путь пройдет через вершину x , то лист относится к типу «над». Если путь отклонится в левую сторону от пути, ведущего в x , то лист относится к типу «левее», если в правую — то к типу «правее». Разумеется, деление листьев на типы зависит от того, какую вершину x мы рассматриваем, что подчеркивается обозначениями Лев (x), Над (x), Прав (x). Например, если x — корень, то Лев (x) и Прав (x)

пусты, а Над (x) — множество всех листьев. Если же x — лист, то Над (x) = { x }.

Введенные обозначения позволяют сделать такое наблюдение. Раз мы решили обрабатывать листья слева направо, будет такой промежуток времени, когда мы уже обработали все листья из Лев (x), но не обработали ни одного листа из Над (x). В течение этого промежутка времени мы должны побывать в точке x , так как все пути из Лев (x)-листьев в Над(x)-листья ведут через вершину x . Аналогичным образом найдется промежуток времени, когда мы обрабатываем все Лев (x)- и Над (x)-листья, но не обработаем ни одного Прав (x)-листа. На этом промежутке также будет момент прохождения через вершину x . Таким образом, мы должны побывать в вершине x дважды. В первый раз будет выполнено такое условие:

 | обработаны все листья левее
 | текущей вершины и только они (ОбрЛев)
а во второй раз такое:
 | обработаны все листья левее
 | текущей вершины, все листья над (ОбрЛевНад)
 | текущей вершиной и только они

Отметим следующие простые факты. Напомним, что запись {предусловие}действие{постусловие} означает, что если выполнено предусловие, то действие выполнимо и после его совершения будет выполнено постусловие.

Факт 1. {ОбрЛев и есть_сверху}
 верх_налево
 {ОбрЛев}

В самом деле, если вершина x имеет t верхних соседей и y — самый левый из них, то при переходе по команде верх_налево из x в y множество листьев типа «левее» не изменится: Лев (x) = Лев (y). Поэтому при переходе от x к y условие {ОбрЛев} не нарушится (рис. 18).

Факт 2. {ОбрЛев, неверно, что есть_сверху}
 обработать
 {ОбрЛевНад}

В самом деле, если исполнитель находится в листе x , то Над (x) = x . Поэтому для перехода от {ОбрЛев} к {ОбрЛевНад} достаточно обработать лист x .

Факт 3. {ОбрЛевНад, есть_справа} вправо {ОбрЛев}

В самом деле, если y — правый сосед x , то Лев (y) есть в точности объединение Лев (x) и Над (x) (рис. 19).

Факт 4. {ОбрЛевНад, неверно, что есть_справа,
 есть_снизу}
 .вниз
 {ОбрЛевНад}

В самом деле, если y — самый правый из верхних соседей x , то Прав(y) = Прав(x) и, следовательно, Лев(x) \cup Над(x) = Лев(y) \cup Над(y), так что условие {ОбрЛевНад} не нарушается при переходе из y в x (рис. 20).

Выполнив команду «в_корень», мы делаем истинным условие {ОбрЛев}, так как листьев левее корня нет. Нам нужно, чтобы в конце исполнения программ все листья были бы обработаны. Это будет гарантировано, если исполнитель будет в корне и будет выполнено условие {ОбрЛевНад}, так как листьев правее корня нет. Итак, получаем общую схему алгоритма:

| в_корень
 | {ОбрЛев}
 | X
 | {в_корне, ОбрЛевНад}

Здесь через X обозначено неизвестное пока действие, которое позволит перейти от состояния, где верно {ОбрЛев}, к состоянию, где верно {ОбрЛевНад}.

Лев(x) = Лев(y)

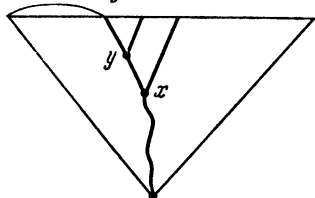


Рис. 18

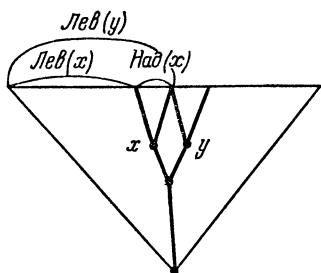


Рис. 19

Лев(x) \cup Над(x) = Лев(y) \cup Над(y)

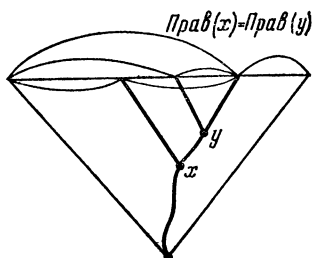


Рис. 20

Разобьем задачу на две части таким образом:

```
в_корень
{ОбрЛев}
Y
{ОбрЛевНад}
Z
{в_корне, ОбрЛевНад}
```

т. е. сначала добьемся {ОбрЛевНад}, а затем, не нарушая этого условия, добьемся еще и попадания в **корень**. Второе действие будем искать в виде цикла. Получаем такую схему для Z:

```
{ОбрЛевНад}
пока есть_снизу повторять
. {ОбрЛевНад и есть_снизу}
. U
. {ОбрЛевНад}
конец_пока
{в_корне, ОбрЛевНад}
```

Как всегда, после выполнения команды повторения входящая в нее проверка — в данном случае «есть_снизу» — ложная, т. е. исполнитель в **корне**.

Внутри команды повторения нам необходимо какое-то действие, не нарушающее {ОбрЛевНад} и приближающее нас к **корню**. Что это за действие? Наверное, команда вниз. Действительно, если мы не в **корне** и нет правого соседа, то команда вниз не нарушает {ОбрЛевНад} и приближает нас к **корню**. Остается научиться сохранять {ОбрЛевНад}, если правый сосед есть. Получаем такую схему для V:

```
{ОбрЛевНад, есть_снизу}
выбор
. при есть_справа делай
  {ОбрЛевНад, есть_справа, есть_снизу}
. V
. {ОбрЛевНад}
. при неверно, что есть_справа делай
  {ОбрЛевНад, есть_снизу, нет справа}
. вниз
. {ОбрЛевНад}
конец_выбора
{ОбрЛевНад}
```

Что же делать, если {ОбрЛевНад}, есть снизу и есть справа (первый вариант в команде выбора)? Если мы выполним команду вправо, то будет выполнено условие {ОбрЛев} (см. выше) и останется перейти

от него к {ОбрЛевНад}. А это все равно нам необходимо (см. выше спецификации для Y). Вот что теперь получается для V :

```
{ОбрЛевНад, есть_справа, есть_снизу}  
вправо  
{ОбрЛев}  
 $Y$   
{ОбрЛевНад}
```

Итак, нам осталось (сразу для двух мест) найти команду, позволяющую от условия {ОбрЛев} перейти к {ОбрЛевНад}. Мы видели, что это возможно, если нет верхних соседей: в этом случае достаточно обработать текущую вершину. Остается

```
{ОбрЛев}  
 $W$   
{ОбрЛев, неверно, что есть_сверху}  
обработать  
{ОбрЛевНад}
```

Оставшаяся неизвестной команда W должна сохранять {ОбрЛев} и добиваться отсутствия верхних соседей. Мы знаем, что если верхние соседи есть, команда «вверх_налево» не нарушает {ОбрЛев}. Получаем такое описание для W :

```
{ОбрЛев}  
пока есть_сверху повторять  
 . вверх_налево  
конец_пока  
{ОбрЛев, нет_сверху}
```

Все, задача решена. Посмотрим на получившуюся программу. Для удобства дадим команде V более подходящее название «вверх_до_упора_и_обработать». Получим вот что:

```
в_корень  
вверх_до_упора_и_обработать  
пока есть_снизу повторять  
 . выбор  
 . . при есть_справа делай  
 . . вправо  
 . . вверх_до_упора_и_обработать  
 . . при неверно, что есть_справа делай  
 . . вниз  
 . конец_выбора  
конец_пока
```

где команда «вверх_до_упора_и_обработать» означает

пока есть_сверху повторять
 . вверх_налево
конец_пока
обработать

Остается доказать, что команды повторения заканчивают свою работу. Для команды повторения, входящей в описание команды «вверх_до_упора_и_обработать», это очевидно: с каждым повторением мы движемся ввѣрх по дереву, а дерево конечно. Чуть менее это очевидно для команды повторения в основной программе. Будем следить за парой (число необработанных вершин, высота исполнителя в дереве). При каждом повторении либо уменьшается первый член пары (если есть_справа), либо при неизменном первом уменьшается второй (исполнитель спускается вниз). Как мы видели в главе 7 «Сколько веревочке ни виться...», это не может продолжаться бесконечно долго.

Снова о ферзях

Чтобы применить нашу программу к головоломке о ферзях, нужен исполнитель, который бы умел выполнять команды движения по дереву позиций и соответствующие проверки, а по команде «обработать» проверял бы, не является ли позиция дерева, в которой он находится, решением головоломки, и если да, то печатал бы ее. (Наша программа обрабатывает только листья; это не мешает найти все решения головоломки, так как решения всегда являются вершинами дерева позиций.) Разумеется, мы не требуем, чтобы исполнитель — будь он человеком или компьютером — действительно рисовал бы на бумаге дерево позиций и передвигался бы по нему. Вполне достаточно, чтобы он имитировал это поведение, двигаясь по воображаемому дереву. Вначале дадим инструкции для исполнителя-человека.

Инструкция для исполнителя-человека. Для ее исполнения необходимы шахматная доска и восемь ферзей.

Предписания	Действия
в_корень	убрать всех ферзей с доски
есть_сверху	если ферзи не бьют друг друга
	и есть свободная горизонталь,
	ответ — «да», иначе — «нет».

вверх_налево	поставить нового ферзя на самую левую клетку самой нижней из свободных горизонталей
есть_справа	если самый верхний ферзь стоит у правого края доски, то ответ — «нет», иначе — «да».
направо	продвинуть самого верхнего ферзя на одну клетку вправо
есть_снизу	если на доске есть хоть один ферзь, ответ — «да», иначе — «нет».
вниз	убрать верхнего ферзя с доски
обработать	посмотреть, является ли позиция решением; если да, добавить ее запись к списку решений.

Если исполнитель аккуратно следует этим инструкциям, то выполнение приведенной выше программы приведет к созданию списка всех решений головоломки, что и требовалось.

Разумеется, нам хотелось бы поручить эти действия компьютеру. При этом манипуляции с доской и ферзями нужно заменить какими-то действиями с более абстрактными объектами.

Представление позиций. Будем представлять позиции с помощью переменной «число_ферзей», принимающей значения от 0 до 8, и таблицы «координаты [1:8]», заполненной целыми числами от 1 до 8. При этом число «координаты [i]» будет номером вертикали, в которой стоит ферзь, расположенный на *i*-й горизонтали. Так, если число_ферзей = 2, координаты [1] = 3, координаты [2] = 7, то такие значения переменных будут представлять позицию (*c*1, *g*2). Отметим, что значения координаты [i] при *i* > число_ферзей не влияют на представляемую позицию. Наш исполнитель должен менять значения переменных число_ферзей и координаты так, чтобы они представляли позицию в дереве позиций, меняющуюся в соответствии с полученными командами. Выпишем соответствующие действия, предположив, что умеем проверять, бьют ли ферзи друг друга.

Предписания Действия
 в_корень число_ферзей ← 0
 есть_сверху ответ: (число_ферзей < 8) и
 ферзи не бьют друг друга

вверх_налево	число ферзей увеличить на 1
есть_справа	координаты [число ферзей] $\leftarrow 1$
	ответ: число_ферзей > 0 и коор-
	динаты [число_ферзей] < 8
направо	координаты [число_ферзей]
	увеличить на 1
есть_снизу	ответ: число_ферзей > 0
вниз	число_ферзей уменьшить на 1
обработать	если ферзи не бьют друг друга то
	. напечатать позицию
	. иначе
	. ничего не делать
	конец_если

Осталось объяснить, как напечатать позицию и как проверить, бьют ли ферзи друг друга. Первое зависит от того, в каком виде мы хотим печатать позиции и какими средствами для этого располагаем; подробно описывать это не будем. Проверка, бьют ли ферзи друг друга, упрощается таким замечанием: в позициях, входящих в дерево позиций, все ферзи, кроме последнего, друг друга не бьют (если бы они били друг друга, мы не ставили бы новых!). Поэтому достаточно проверять, бьет ли верхний ферзь кого-либо из предыдущих. Будем проверять их по очереди.

Введем переменные « k » и «бьет_до_ k » и будем стремиться к тому, чтобы

(*) бьет_до_ k = да \Leftrightarrow новый ферзь бьет одного из ферзей, стоящих ниже k -й горизонтали.

Условие (*) можно сделать истинным, положив $k \leftarrow 1$, бьет_до_ k \leftarrow нет.

Получаем такую схему:

```

|  $k \leftarrow 1$ , бьет_до_ $k$   $\leftarrow$  нет
| пока  $k \neq$  число_ферзей повторять
| . увеличить  $k$ , сохраняя истинность (*)
| конец_пока
| {(*)},  $k =$  число_ферзей
| ответ: бьет_до_ $k$ 

```

Увеличение k с сохранением (*) выполняется так: если k -й ферзь бьет последнего то

```

| . увеличить  $k$  на 1
| . бьет_до_ $k$   $\leftarrow$  да
| . иначе
| . увеличить  $k$  на 1
| конец_если

```

Здесь « k -й ферзь бьет последнего» означает, что координаты $[k]$ = координаты [число_ферзей] (ферзи на одной вертикали) или $|k - \text{число_ферзей}| =$ = | координаты $[k]$ — координаты [число_ферзей] | (ферзи на одной диагонали).

Дальнейшую конкретизацию алгоритма (а при необходимости и запись его в виде программы на каком-то реальном языке программирования) мы оставляем читателю: когда (и если) это понадобится, вы это легко сделаете.

У п р а ж н е н и е. Проверьте правильность следующей программы обхода дерева (здесь состояние — переменная, которая может принимать два значения: ОбрЛев и ОбрЛевНад).

```

в_корень
состояние ← ОбрЛев
{значение переменной состояние правильно отра-
жает действительность}
пока есть_снизу или не (состояние = ОбрЛевНад)
. повторять
. выбор
. . при состояние = ОбрЛев и есть_сверху делай
. . вверх_налево, состояние ← ОбрЛев
. . при состояние = ОбрЛев и не есть_сверху
. . делай
. . обработать, состояние ← ОбрЛевНад
. . при состояние = ОбрЛевНад и есть_справа
. . делай
. . вправо, состояние ← ОбрЛев
. . при состояние = ОбрЛевНад и
. . (не есть_справа) и есть_снизу
. . делай
. . вниз, состояние ← ОбрЛевНад
. конец_выбора
конец_пока
{значение переменной состояние правильно отра-
жает действительность, исполнитель в корне
и состояние = ОбрЛевНад, т. е. все обработано}

```

Отметим, что эта (пожалуй, более простая, чем приведенная выше) программа обхода дерева в точности соответствует утверждениям 1 — 4 в разделе «Обход дерева». Свойства «ОбрЛев» и «ОбрЛевНад», которые в предыдущей программе фигурировали лишь в утверждениях, здесь становятся неотъемлемой частью самой программы.

МОЖНО ЛИ ПОДНЯТЬ СЕБЯ ЗА ВОЛОСЫ, ИЛИ РЕКУРСИЯ

Всякое математическое определение, вводящее какое-то новое понятие или термин, объясняет его значение с помощью других понятий, предполагающихся известными. Например, определение тангенса с помощью формулы

$$\operatorname{tg}(x) = \sin(x)/\cos(x)$$

предполагает известными понятия синуса и косинуса (а если быть педантичными, то и деления). Двигаясь от определяемых понятий к используемым в определениях, мы в конце концов приходим к первичным (не определяемым в рамках данной математической теории) понятиям. Так, в геометрии понятия «точка», «прямая», «лежать на» являются неопределяемыми.

Похожая ситуация возникает и в программировании. Каждую программу можно рассматривать как набор определений. Каждое из них определяет какую-то команду через другие, предполагающиеся известными. Двигаясь от определяемой команды к используемым, мы в конце концов должны прийти к неопределяемым командам, т. е. к командам, входящим в систему команд исполнителя, для которого пишется наша программа.

И в математике, и в программировании иногда встречаются «определения», не укладывающиеся в эту схему. Рассмотрим в качестве примера такое «определение» факториала натурального числа n :

$$\operatorname{fact}(n) = 1, \quad \text{если } n = 0;$$

$$\operatorname{fact}(n) = n \operatorname{fact}(n - 1), \quad \text{если } n > 0.$$

На первый взгляд оно кажется бессмысленным: если мы знаем, что такое факториал, то нам не нужно никакого определения; если же мы не знаем, что такое факториал, то запись $\operatorname{fact}(n - 1)$ в правой части столь же непонятна, как и запись $\operatorname{fact}(n)$ в левой, и это «определение» ничего не определяет. Тем не менее чувствуется, что какой-то смысл в нем есть. В самом деле, с его помощью можно найти

$$\text{fact}(0) = 1, \quad \text{fact}(1) = 1 \cdot \text{fact}(1) = 1,$$

$$\text{fact}(2) = 2 \cdot \text{fact}(1) = 2, \quad \text{fact}(3) = 3 \cdot \text{fact}(2) = 6$$

и т. д.

Это отличает приведенное определение от такого:
 $\text{fact}(n) = \text{fact}(n+2)/(n+1)(n+2),$

которое кажется бесполезным, хотя и «верным» (в том смысле, что «настоящий» факториал ему удовлетворяет) или от еще более бесполезного (хотя и «верного»)

$$\text{fact}(n) = \text{fact}(n).$$

В чем же разница между приведенными «определениями»? Почему первое кажется нам более осмысленным, чем второе (и тем более третье)? Чтобы понять это, нужно назвать эти «определения» их настоящим именем — это не определения, а требования к функции fact , требования, которым она может удовлетворять или не удовлетворять. Третье требование является самым слабым — ему удовлетворяет любая функция (мы всюду рассматриваем функции с натуральными аргументами и значениями). Второе — более сильное, но по-прежнему удовлетворяющих ему функций много: если fact — такая функция, то функция

$$\text{fact1}(n) = c \cdot \text{fact}(n)$$

также удовлетворяет ему при любом натуральном c . Наконец, первому требованию удовлетворяет лишь одна функция. Докажем это. Пусть fact1 и fact2 — две функции с натуральными аргументами и значениями, ему удовлетворяющие. Это значит, что

$$\text{fact1}(0) = \text{fact2}(0) = 1,$$

$$\text{fact1}(n) = n \text{ fact1}(n-1),$$

$$\text{fact2}(n) = n \text{ fact2}(n-1).$$

Докажем, что $\text{fact1}(k) = \text{fact2}(k)$ при любом натуральном k . Рассуждаем по индукции. Базис индукции у нас есть:

$$\text{fact1}(0) = \text{fact2}(0) = 1.$$

Шаг индукции. Предполагая, что

$$\text{fact1}(k-1) = \text{fact2}(k-1),$$

докажем

$$\text{fact1}(k) = \text{fact2}(k).$$

В самом деле,

$$\begin{aligned}\text{fact1}(k) &= k \cdot \text{fact1}(k-1) = k \cdot \text{fact2}(k-1) = \\ &= \text{fact2}(k).\end{aligned}$$

Доказанное позволяет рассматривать равенства

$$\text{fact}(n) = 1, \quad \text{если } n = 0,$$

$$\text{fact}(n) = n \cdot \text{fact}(n-1), \quad \text{если } n > 0$$

как определение — определение той единственной функции, которая этим требованиям удовлетворяет. Такие определения часто называют рекурсивными.

Заметим, что наше доказательство, по существу, без изменений переносится и на другие случаи. В частности, условия

$$f(0) = a, \quad f(n) = F(n, f(n-1)) \quad \text{при } n > 0$$

однозначно определяют функцию $f: \mathbb{N} \rightarrow \mathbb{N}$, каковы бы ни были натуральное число a и функция двух натуральных аргументов F . Например, при $a = 0$, $F(n, x) = x + 3$ получаем равенства

$$f(0) = 0, \quad f(n) = f(n-1) + 3,$$

которые, очевидно, определяют функцию

$$f(n) = 3n.$$

У п р а ж н е н и е. Какие функции получатся, если

$$(1) \quad a = 1, \quad F(n, x) = (n+1)x;$$

$$(2) \quad a = 1, \quad F(n, x) = 2x;$$

$$(3) \quad a = 0, \quad F(n, x) = x + 2n - 1?$$

Ответ к (1): $f(n) = (n+1)!$

Существуют и многие другие ситуации, когда функция, удовлетворяющая данным требованиям, единственна, и, следовательно, эти требования могут рассматриваться как ее определение. Мы не будем формулировать каких-либо общих теорем по этому поводу, а приведем несколько примеров.

Равенства

$$f(x, 0) = x, \quad f(x, y+1) = f(x, y) + 1$$

однозначно определяют функцию сложения $f(x, y) = x + y$. В самом деле, если $f(x, y) = x + y$ верно

при каком-то y , то из второго равенства следует, что это верно и при y , большем на 1:

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

Первое равенство гарантирует, что

$$f(x, y) = x + y$$

верно при $y = 0$. Ссылаясь вновь на принцип математической индукции, получаем требуемое. Аналогичным образом равенства

$$f(x, 0) = 0, \quad f(x, y + 1) = f(x, y) + x$$

задают функцию умножения $f(x, y) = xy$.

У п р а ж н е н и е. Доказать, что требованиям

$$f(x, 0) = 1, \quad f(x, y + 1) = f(x, y) \cdot x$$

удовлетворяет единственная функция. Какая?

В приведенных выше примерах значение f в любой точке определяется через значение f в «предыдущей» точке ($f(n)$ через $f(n - 1)$, $f(x, y + 1)$ через $f(x, y)$). Бывают и более сложные случаи.

Определим функцию $f: (x, y) \mapsto f(x, y)$ с натуральными аргументами и значениями так:

$$f(x, 0) = x, \quad f(0, y) = y,$$

$$f(x, y) = f(x - y, y) \quad \text{при } x \geq y,$$

$$f(x, y) = f(x, y - x) \quad \text{при } x \leq y.$$

Докажем, что существует не более одной функции f , обладающей такими свойствами. В самом деле, пусть f_1 и f_2 — две такие функции, причем разные. Будем называть пару (x, y) плохой, если в ней значения функций f_1 и f_2 различаются, т. е.

$$f_1(x, y) \neq f_2(x, y).$$

Выберем среди всех плохих пар (x, y) (а такие существуют по нашему предположению) пару с наименьшим значением $x + y$. Обозначим эту пару (a, b) . Заметим, что a не равно 0, иначе $f_1(a, b)$ и $f_2(a, b)$ равнялись бы b в соответствии с нашим требованием. Аналогично и b не равно 0. Не ограничивая общности, можно считать, что $a \geq b$ (в случае $a \leq b$ рассуждения аналогичны). Согласно нашим требованиям,

$$f_1(a, b) = f_1(a - b, b), \quad f_2(a, b) = f_2(a - b, b).$$

Поскольку левые части по предположению не равны, то не равны и правые. Значит, пара $(a - b, b)$ плохая. Но у нее сумма меньше:

$$(a - b) + b = b < a + b.$$

Это противоречит нашему предположению о том, что (a, b) — плохая пара с наименьшей суммой. Полученное противоречие показывает, что $f_1 \neq f_2$ невозможно.

У п р а ж н е н и е. Доказать, что функция f , удовлетворяющая указанным требованиям, существует, и найти ее. (Указание: см. главу 5 «Алгоритм Евклида».)

В следующем примере «определение» выражает значение функции f в одной точке через значения в двух других:

$$f(0) = f(1) = 1, \quad f(n) = f(n-2) + f(n-1)$$

при $n > 1$.

Последовательность $f(0), f(1), f(2) \dots$ называют последовательностью Фибоначчи.

Рекурсивные определения оказываются полезными не только (и не столько) для функций натурального аргумента, сколько для функций, определенных на других множествах. Пусть A — конечное множество, называемое алфавитом, элементы которого мы называем буквами. Будем называть словом в алфавите A произвольную конечную последовательность букв этого алфавита. Будем обозначать через Λ пустое слово (слово, не содержащее ни одной буквы), а через xP и Px — слова, получающиеся приписыванием буквы x слева и справа к слову P . Рассмотрим теперь такое рекурсивное определение:

$$\text{Rev}(\Lambda) = \Lambda, \quad \text{Rev}(Px) = x \text{Rev}(P),$$

выражающее значение функции Rev на любом непустом слове через значение Rev на том же слове без последней буквы.

У п р а ж н е н и е. Проверьте, что этим требованиям удовлетворяет единственная функция. Что это за функция? (Ответ: переворачивание слова.)

Пусть наш алфавит состоит из цифр $0, 1, \dots, 9$. Определим функцию Val , сопоставляющую со словами в этом алфавите некоторые натуральные числа,

таким рекурсивным определением:

$$\text{Val}(\Lambda) = 0, \quad \text{Val}(Px) = 10 \text{Val}(P) + x$$

(в последнем равенстве цифра x рассматривается как натуральное число и прибавляется к числу $10 \text{Val}(P)$). Это определение однозначно определяет $\text{Val}(P)$ как натуральное число, имеющее P своей десятичной записью.

У п р а ж н е н и е. Проверьте это.

Особенно важны рекурсивные определения функций на деревьях. Мы уже знаем, что такое дерево: из некоторой точки (называемой корнем) проведены идущие вверх отрезки, соединяющие ее с несколькими другими точками («вершинами»), из некоторых из них проведены еще отрезки и т. д. (см. рис. 14). Другими словами, деревом является конфигурация, которую можно получить по следующим правилам (рис. 21):

- (1) конфигурация из одной точки — дерево;
- (2) если T_1, T_2, \dots, T_k — уже построенные деревья, а x — точка, лежащая ниже них, то, соединив x с нижними точками T_1, T_2, \dots, T_k , получим дерево.

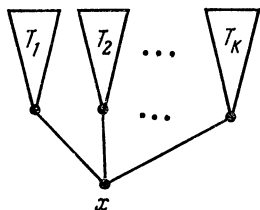


Рис. 21

По существу, это определение можно также называть рекурсивным.

Если x — произвольная вершина дерева T , то можно рассмотреть дерево $T(x)$, которое получится, если оставить от T лишь точку x и все, что над ней находится.

Введенные обозначения позволяют дать рекурсивные определения некоторых функций на деревьях. Вот пример:

$$\text{высота}(T) = \begin{cases} 0, & \text{если } T \text{ состоит из одного корня;} \\ 1 + \max(\text{высота}(T(x_1)), \dots, \text{высота}(T(x_k))), & \text{где } x_1, \dots, x_k \text{ — все вершины,} \\ & \text{соединенные с корнем отрезками} \end{cases}$$

У п р а ж н е н и е. Дайте аналогичное определение для функции, сопоставляющей с каждым деревом (1) число его вершин; (2) число его листьев (вершин, из которых не выходит отрезков вверх).

Мы говорили о рекурсивных определениях функций. Подобная ситуация может возникнуть и в программировании, когда мы разрешаем в описании команды использовать саму описываемую команду. Вот возможный пример (см. главу 1 «Путник в лабиринте»):

команду вперед_до_упора понимать_как

. если впереди свободно то

. . шаг вперед

. . вперед_до_упора

. . иначе

. . ничего_не_делать

. конец_если

конец_описания

Хотя интуитивно кажется правдоподобным, что это описание в каком-то смысле описывает нужные действия, но точный смысл придать ему не так просто. Современное состояние науки программирования недостаточно удовлетворительно для того, чтобы дать краткое, точное и понятное объяснение смысла таких «рекурсивных программ» (показательный пример: в книге Мейера и Бодуэна «Методы программирования» в главе про рекурсию первая же рекурсивная программа неверна). Тем не менее будем надеяться, что со временем это важное понятие найдет свое законное место.

Закончим раздел загадкой. Золотая рыбка обещала выполнить два любых желания. Как заставить ее выполнить бесконечно много желаний? (Указание: второе желание должно быть таким: «хочу, чтобы рыбка выполнила два следующих желания».)

ГЛАВА 11

ОТ РЕКУРСИВНОГО ОПРЕДЕЛЕНИЯ К ПРОГРАММЕ

В главе 10 «Можно ли поднять себя за волосы...» мы дали рекурсивные «определения» некоторых функций. В этой главе мы покажем, как можно написать

программы, соответствующие этим определениям. Приведем две программы — для вычисления факториала и для вычисления членов последовательности Фибоначчи, постаравшись сделать их построения возможно более похожими,

Факториал

Напишем программу, вычисляющую факториал любого натурального числа x (обозначается $x!$). Для этого вспомним, что такое факториал. По определению,

$$0! = 1, \quad (x + 1)! = x! \cdot (x + 1)$$

(знающие комбинаторику вспомнят, что $x!$ — это число различных перестановок последовательности из x элементов). Как видно из этих формул, вычисление $(x + 1)!$ требует знания $x!$, поэтому будем вычислять все факториалы по очереди, пока не дойдем до нужного нам $x!$. Поскольку программа должна вычислять $x!$ для всякого натурального x , заранее мы не знаем количество промежуточных факториалов, которые надо будет вычислять. Таким образом, следует воспользоваться командой повторения. Введем две целочисленные переменные k и f и сделаем так, чтобы соотношения

$$(*) \begin{cases} 0 \leq k \leq x, \\ f = k! \end{cases}$$

были верны после каждого повторения. В качестве проверки возьмем $k \neq x$; тогда после выполнения команды повторения будет $k = x$, $f = k!$ и, следовательно, $f = x!$, т. е. f — искомый ответ. Итак, наша команда описывается так:

пока $k \neq x$ повторять
 . сделать что-нибудь, не нарушая $(*)$
 конец пока

Необходимо сделать так, чтобы соотношения $(*)$ были верны к началу цикла. Для этого присвоим переменной k значение 0, а переменной f значение 1. Тогда $k! = f$, так как $0! = 1$.

Теперь нужно детализировать команду дальше. Задача состоит в том, чтобы приближаться к цели (в данном случае к равенству $k = x$), не нарушая ус-

ловия (*). Будем увеличивать k на 1. При этом условие $0 \leq k \leq x$ сохранится, так как до прибавления единицы было $0 \leq k \leq x$ и $k \neq x$ (раз мы не завершили выполнение цикла). Условие же $k! = f$ будет нарушено, вместо него станет верным $(k-1)! = f$. Чтобы восстановить необходимое условие, умножим f на k . Посмотрим теперь, что же мы получили (текст, заключенный в фигурные скобки, — утверждения, справедливые в этом месте программы):

```

{ $x \geq 0$ }
 $k \leftarrow 0, \quad f \leftarrow 1$ 
{ $0 \leq k \leq x, \quad f = k!$ }
пока  $k \neq x$  повторять
. { $0 \leq k < x, \quad f = k!$ }
.  $k \leftarrow k + 1$ 
. { $0 < k \leq x, \quad f = (k-1)!$ }
.  $f \leftarrow f \cdot k$ 
. { $0 < k \leq x, \quad f = k!$ }
конец пока
{ $f = k!, \quad k = x$ , и поэтому  $f = x!$ }

```

Докажем теперь, что программа заканчивает работу. В самом деле, величина $x - k$ при каждом повторении уменьшается на 1, оставаясь неотрицательной, поэтому программа кончит работу. Таким образом, наша программа вычисляет факториал числа x .

На языке Паскаль (см. главу 24) соответствующий фрагмент программы выглядит так:

```

k:= 0;
f:= 1;
while k <> x do begin
    k:= k + 1;
    f:= f * k;
end

```

Последовательность Фибоначчи

Напишем программу, вычисляющую x -й член последовательности Фибоначчи (будем обозначать его $\text{fib}(x)$). Для этого вспомним, что такое последовательность Фибоначчи. По определению,

$$\text{fib}(0) = 1, \quad \text{fib}(1) = 1,$$

$$\text{fib}(x) = \text{fib}(x-2) + \text{fib}(x-1) \quad \text{при } x > 1.$$

Как видно из этих формул, вычисление $\text{fib}(x)$ требует знания $\text{fib}(x - 2)$ и $\text{fib}(x - 1)$, поэтому мы будем вычислять все члены последовательности Фибоначчи по очереди, пока не дойдем до нужного нам $\text{fib}(x)$. Поскольку программа должна вычислять $\text{fib}(x)$ для всякого натурального x , то мы заранее не знаем количество промежуточных членов последовательности Фибоначчи, которые надо будет вычислять. Таким образом, следует воспользоваться командой повторения. Введем три целочисленные переменные k , f и $f1$ и сделаем так, чтобы соотношения

$$(*) \quad \begin{cases} 0 \leq k \leq x, \\ f = \text{fib}(k), \\ f1 = \text{fib}(k + 1) \end{cases}$$

были верны после каждого повторения. В качестве проверки возьмем $k \neq x$; тогда после выполнения команды повторения будет $k = x$, $f = \text{fib}(k)$ и, следовательно, $f = \text{fib}(x)$, т. е. f — искомый ответ. Итак, наша команда описывается так:

пока $k \neq x$ повторять
 . сделать что-нибудь, не нарушая (*)
 конец_пока

Теперь необходимо сделать так, чтобы соотношения (*) были верны к началу цикла. Для этого присвоим переменной k значение 0, а переменным f и $f1$ значение 1. Тогда $\text{fib}(k) = f$, $\text{fib}(k + 1) = f1$, так как $\text{fib}(0) = \text{fib}(1) = 1$. Детализируем команду дальше. Задача состоит в том, чтобы приближаться к цели (равенству $k = x$), не нарушая условий (*). Будем увеличивать k на 1. При этом условие $0 \leq k \leq x$ сохранится, так как до прибавления единицы было верно $0 \leq k \leq x$ и $k \neq x$ (раз мы не завершили выполнение цикла). Два других условия нарушатся: вместо них станут верными $f = \text{fib}(k - 1)$, $f1 = \text{fib}(k)$. Нам нужно перейти от $f = \text{fib}(k - 1)$ к $f = \text{fib}(k)$. Для этого достаточно положить f равным $f1$. Однако в этом случае будет безвозвратно утеряно значение $\text{fib}(k - 1)$, необходимое для вычисления $\text{fib}(k + 1)$, поэтому мы поступим несколько иначе: сперва запомним его, положив t равным f (t — вспомогательная переменная, используемая лишь для указанной цели), затем положим f равным $f1$ и, наконец, положим $f1$ равным сумме предыдущего значения $f1$ и t . Посмотрим, что же мы получили.

```

{ $x \geq 0$ }
 $k \leftarrow 0, f \leftarrow 1, f1 \leftarrow 1$ 
{ $0 \leq k \leq x, f = \text{fib}(k), f1 = \text{fib}(k+1)$ }
пока  $k \neq x$  повторять
. { $0 \leq k < x, f = \text{fib}(k), f1 = \text{fib}(k+1)$ }
.  $k \leftarrow k+1$ 
. { $0 < k \leq x, f = \text{fib}(k-1), f1 = \text{fib}(k)$ }
.  $t \leftarrow f$ 
. { $0 < k \leq x, t = f = \text{fib}(k-1), f1 = \text{fib}(k)$ }
.  $f \leftarrow f1$ 
. { $0 < k \leq x, t = \text{fib}(k-1), f = f1 = \text{fib}(k)$ }
.  $f \leftarrow f+t$ 
. { $0 < k \leq x, f = \text{fib}(k), f1 = \text{fib}(k+1)$ }

```

конец пока

{ $f = \text{fib}(k), k = x$ и поэтому $f = \text{fib}(x)$ }

Докажем теперь, что программа заканчивает работу. В самом деле, величина $x - k$ при каждом повторении уменьшается на 1, оставаясь неотрицательной, поэтому программа кончит работу. Таким образом, наша программа действительно вычислит x -й член последовательности Фибоначчи.

На языке Паскаля соответствующий фрагмент программы выглядит так:

```

 $k := 0; f := 1; f1 := 1;$ 
while  $k < x$  do begin
     $k := k + 1; t := f; f := f1; f1 := f + t;$ 
end;

```

З а м е ч а н и е. Мы вовсе не утверждаем, что приведенная программа представляет собой наилучший способ вычисления чисел Фибоначчи — существуют способы, позволяющие сделать это гораздо быстрее.

ГЛАВА 12

ХАНОЙСКИЕ БАШНИ

Головоломка, давшая название этой главе, состоит в следующем. Имеются три палочки, на которые можно надевать кольца, и n колец различных размеров (рис. 22). Вначале все кольца находятся на одной палочке (в порядке убывания размеров); их нужно переложить на другую палочку. При этом не допускается, чтобы большее кольцо лежало поверх меньшего, и на



Рис. 22,

каждом шаге разрешается перекладывать только одно кольцо. Нужно доказать, что это можно сделать, и придумать как. Другими словами, нужно составить программу, выполнение которой привело бы к перемещению всех колец с одной палочки на другую.

Занумеруем наши палочки цифрами 1, 2, 3. После этого возможные действия можно обозначить так:

$$1 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 1, 2 \rightarrow 3, 3 \rightarrow 2,$$

где $i \rightarrow j$ означает «переложить кольцо с палочки i на палочку j ». Нет необходимости указывать, какое именно кольцо нужно перекладывать, так как мы можем по условию взять только одно верхнее кольцо. С помощью введенных обозначений можно записать программу для перекладывания двух колец с первой палочки на вторую (рис. 23). Эта программа подсказы-

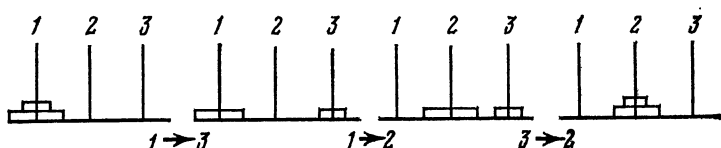


Рис. 23

вает, что мы могли бы решить головоломку, если бы каким-то образом сумели перекладывать пирамиду из всех колец, кроме самого большого, как одно кольцо (рис. 24). Рисунок показывает, что перекладывание пирамиды из n колец с одной палочки на другую сводится

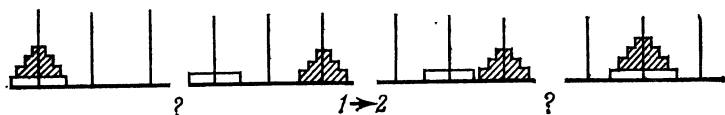


Рис. 24

к двукратному перекладыванию пирамиды из $n - 1$ кольца и к одному перекладыванию самого большого кольца. Отметим, что при перекладывании пирамиды из $n - 1$ кольца оставшееся кольцо помешать не может, так как оно больше всех перекладываемых и на него можно класть что угодно. Получаем такой алгоритм перекладывания n колец ($n > 1$) с первой палочки на вторую:

```

| переложить  $n - 1$  кольцо с 1 на 3
|  $1 \rightarrow 2$ 
| переложить  $n - 1$  кольцо с 3 на 2

```

Аналогично можно записать более общий алгоритм перекладывания n колец ($n > 1$) с i -й палочки на j -ю:

```

| ( $k$  — номер оставшейся палочки:  $k \neq i, k \neq j$ )
| переложить  $n - 1$  кольцо с  $i$  на  $k$ 
|  $i \rightarrow j$ 
| переложить  $n - 1$  кольцо с  $k$  на  $j$ 

```

Наконец, если мы хотим допустить и случай $n = 1$, получаем такой алгоритм перекладывания n колец ($n > 0$) с i -й палочки на j -ю:

```

| выбор
| . при  $n = 1$  делай
| .  $i \rightarrow j$ 
| . при  $n > 1$  делай
| . ( $k$  — номер оставшейся палочки:  $k \neq i, k \neq j$ )
| . переложить  $n - 1$  кольцо с  $i$  на  $k$ 
| .  $i \rightarrow j$ 
| . переложить  $n - 1$  кольцо с  $k$  на  $j$ 
| конец выбора

```

Этот алгоритм, как говорят, является рекурсивным: команда «переложить... колец с ...-й палочки на ...-ю» определяется через себя самое (но с другим, меньшим на 1 значением числа колец). К сожалению, придать таким конструкциям смысл иногда бывает нелегко (см. главу 10 о рекурсии), хотя и возможно. Чтобы убедиться в том, что рекурсивные конструкции могут быть источником трудностей, попробуйте по этому алгоритму решить головоломку для четырех колец, не делая никаких вспомогательных записей. Возможно, вас постигнет та же участь, что и авторов, — вы запутаетесь в том, что уже сделано, что осталось сделать и вообще в каком месте алгоритма вы находитесь.

Интересно поэтому придумать другой вариант алгоритма, который бы описывал ту же последователь-

ность переключений, но не использовал бы того, что мы назвали рекурсией (т. е. использования самой команды в ее описании). Предыдущее обсуждение подсказывает, как это можно сделать. Нужно в каждый момент хранить сведения о том, какие задачи нам еще осталось решить.

С этой целью изготовим большое количество одинаковых карточек, разделенных на три графы: «сколько», «откуда», «куда». Заполнив эти графы числами, мы превратим карточку в задание на переключивание указанного числа колец с указанной палочки на указанную. В таких карточках первое число должно быть больше 0, а два других — от 1 до 3 и не совпадать между собой.

Будем складывать карточки в стопку, лежащую на столе, надписями кверху, подразумевая при этом, что лежащая выше карточка должна выполняться раньше. Будем стараться действовать так, чтобы было справедливо такое свойство:

(*) для завершения решения головоломки осталось выполнить действия в соответствии с лежащими на столе карточками.

Пусть наша исходная задача такова: нужно переложить n колец с i -й палочки на j -ю. Заполним карточку этими числами и положим ее на стол. После этого свойство (*) станет (тривиально) истинным: лежащая на столе единственная карточка отражает наше задание. Далее программа будет манипулировать с кольцами и карточками, сохраняя истинность утверждения (*), пока стопка карточек на столе не станет пустой: в этот момент утверждение (*) гарантирует нам, что все нужные действия уже сделаны и больше ничего делать не надо. Приходим к такой схеме алгоритма переключения n колец с i -й палочки на j -ю:

положить на стол карточку

n	i	j
-----	-----	-----

{осталось выполнить действия, соответствующие карточкам на столе}

пока стопка карточек на столе не пуста повторять
· манипулировать с карточками и кольцами,
· сохраняя истинность утверждения (*)

конец пока

{осталось выполнить действия, соответствующие карточкам на столе, а их нет, значит, задача решена}

Осталось понять, каким именно образом мы можем манипулировать с кольцами и карточками. Первой карточкой, подлежащей исполнению, является, согласно нашему соглашению, верхняя. Пусть на ней написаны числа m, p, q , т. е. она требует переложить m колец с палочки p на палочку q . Если $m = 1$, то мы можем выполнить требуемое действие ($p \rightarrow q$) и убрать карточку из стопки. Если же $m > 1$, то непосредственно выполнить требуемое действие нельзя. Но мы знаем, что оно может быть разбито на три «подзадачи» (через r обозначим палочку, отличную от p и q):

- перенести $m - 1$ кольцо с p на r
- перенести 1 кольцо с p на q
- перенести $m - 1$ кольцо с r на q

Таким образом, можно убрать карточку

m	p	q
-----	-----	-----

 из стопки, положив вместо нее три карточки

$m-1$	p	r
1	p	q
$m-1$	r	q

(в указанном порядке: первую наверх, вторую за ней, третью за второй, далее идут оставшиеся в стопке карточки в том порядке, в котором они были раньше). При такой замене мы не нарушим требуемого условия «после выполнения указанных на карточках действий задача будет решена».

Получаем следующий алгоритм перекладывания n колец с i -й палочки на j -ю:

положить на стол карточку

n	i	j
-----	-----	-----

{осталось выполнить действия, соответствующие карточкам на столе}

пока стопка карточек не пуста повторять

- . (m, p, q — числа на верхней карточке)
- . **выбор**
- . . при $m = 1$ делай
- . . $p \rightarrow q$
- . . убрать верхнюю карточку из стопки
- . . при $m > 1$ делай
- . . (r — номер оставшейся палочки: $r \neq p$,
- . . $r \neq q$)
- . . убрать верхнюю карточку из стопки

- . . . добавить наверх карточку
- . . . добавить наверх карточку
- . . . добавить наверх карточку
- . . .

$m-1$	r	q
-------	-----	-----

1	ρ	q
---	--------	-----

$m-1$	ρ	r
-------	--------	-----

. **конец_выбора**

конец_пока

{осталось выполнить действия, соответствующие карточкам на столе, а их нет, значит, задача решена}

Обратите внимание, что добавляемые карточки указаны в обратном порядке: мы должны вначале добавить ту карточку, которая в стопке будет ниже всех.

На рис. 25 в качестве примера показано, как будет меняться стопка карточек при перекладывании трех колец с первой палочки на вторую (у стрелок указаны перекладываемые кольца).

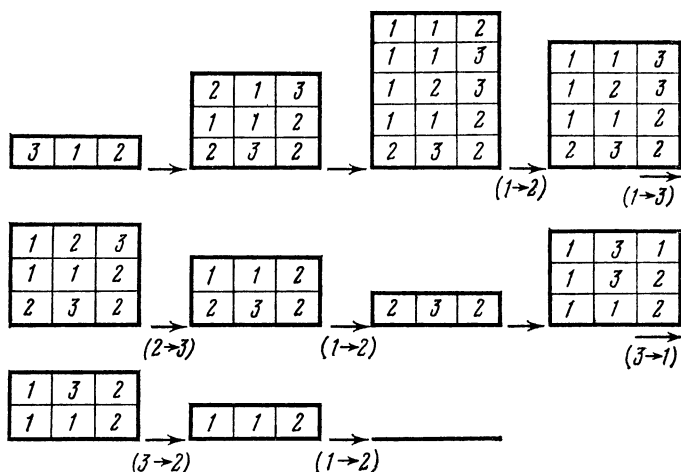


Рис. 25

Нам нужно убедиться, что составленная программа при любых значениях n , i , j заканчивает работу, т. е. что манипуляции с карточками рано или поздно приведут к опустошению стопки. Это следует из того, что каждый раз мы заменяем карточку на другие, у которых на первом месте стоит меньшее число (сравнить

упражнение про скупого рыцаря и черта в главе 7 «Сколько веревочке ни виться...»).

У п р а ж н е н и я. (1) Найдите количество перекладываний при работе нашей программы. (2) Докажите, что никакой другой способ перекладывания не позволяет обойтись меньшим числом действий. (Указание: чтобы переложить самое нижнее кольцо, нужно, чтобы все остальные кольца были сосредоточены на одной из двух других палочек; воспользуйтесь математической индукцией.)

З а м е ч а н и е о т е р м и н о л о г и и. Для стопки карточек, в которую можно добавлять и из которой можно брать карточки с одного конца (в нашем случае сверху), программисты придумали специальное название — «стек» (английское «stack» — стопка, куча). Иногда такую стопку называют «магазином» по аналогии с магазином автомата, изучаемого на уроках начальной военной подготовки. А пределанное нами преобразование рекурсивной программы в программу без рекурсии называют «устранением рекурсии с помощью стека».

ГЛАВА 13

ВЫЧИСЛЕНИЯ И ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ

Изначально, как показывает само название «вычислительные машины» (английское «computer» — вычислитель), вычислительные машины рассматривались прежде всего как очень быстрые арифмометры, заменяющие работу огромного количества людей. Со временем стало ясно, что сфера применения ЭВМ гораздо шире. Однако и теперь научно-технические расчеты являются одной из важных областей применения ЭВМ.

Главной особенностью этой области применения ЭВМ является необходимость иметь дело с действительными числами. Действительных чисел, как известно из математики, очень много — континуум, что больше числа возможных состояний любой ЭВМ. Это обстоятельство является причиной трудностей, природу которых можно пояснить таким примером: нетрудно написать программы, вычисляющие с любой заданной точностью

числа

$$A = 1 + 1/2^2 + 1/3^2 + \dots + 1/n^2 + \dots$$

и

$$B = \pi^2/6,$$

но для проверки того, что $A = B$, понадобился гений Леонарда Эйлера.

В настоящее время наиболее распространенный подход состоит в том, чтобы выделить из множества всех действительных чисел некоторое конечное подмножество «представимых в компьютере» действительных чисел и иметь дело только с ними. Легко понять, однако, что арифметические операции в применении к представимым числам могут давать непредставимые результаты (конечное множество действительных чисел, не состоящее только из нуля, не может быть замкнуто относительно арифметических операций), поэтому наряду с настоящими арифметическими операциями появляются их «тени» в мире представимых чисел, заменяющие непредставимый результат его представимым приближением. Как и всякие тени, они зачастую оказываются уродливыми, не обладая красотой своих прообразов. (Так, например, операция сложения в мире представимых чисел может не быть ассоциативной.) Из-за этого все наши знания о действительных числах, накопленные математикой за многие века, оказываются лишенными оснований.

Эти теоретические трудности находят свое практическое воплощение: каждый, кто пытался выполнить с помощью ЭВМ сколько-нибудь сложный расчет, знаком с подстерегающими его на каждом шагу «арифметическими переполнениями», «машинными нулями» и прочими опасностями. Мы не будем больше говорить о них, тем более что проблема корректного обоснования проводимых на компьютере вычислений с действительными числами далека от своего решения. Все сказанное — лишь предупреждение о том, что при современном состоянии дел любые вычисления на компьютере с действительными числами — хождение по краю пропасти, и помешать падению в нее могут лишь здравый смысл, математическая интуиция и (быть может, в первую очередь) удача.

Предупредив о возможных сложностях, мы рассмотрим простую задачу, в которой они практически не

возникают — задачу нахождения квадратного корня из заданного числа. Впрочем, одна сложность есть и здесь — еще пифагорейцы открыли, что квадратные корни бывают иррациональными, даже если они извлекаются из целых чисел. Так что не вполне ясно, например, какого ответа мы ждем, попросив компьютер вычислить нам $\sqrt{2}$. По-видимому, одно из разумных толкований задачи таково: нужно уметь вычислять приближения к искомому корню с любой наперед заданной точностью. Другими словами, уметь вычислить \sqrt{a} означает: по любому $\varepsilon > 0$ указывать число, отстоящее от \sqrt{a} не больше чем на ε . Мы будем считать, что числа a и ε рациональны и что с ними наша программа может оперировать свободно (как и с другими рациональными числами). Числа, предлагаемые программой в качестве приближений, также будут рациональными. Эти допущения не вполне реалистичны, зато упрощают наши рассуждения. Мы приходим к следующей задаче:

{дано: $a \geq 0, \varepsilon > 0$??? {получить: ответ — \sqrt{a} $\leq \varepsilon$ }

Идея решения проста: найдем отрезок, в котором искомый корень содержится, а затем будем этот отрезок сужать, пока его длина не станет достаточно малой (не выпуская из него корня). Приходим к такому наброску программы (концы отрезка, в котором содержится корень, обозначаем l и r ; по ходу процесса l и r сближаются):

{ $a \geq 0, \varepsilon > 0$ найти отрезок, содержащий \sqrt{a} { \sqrt{a} содержится в отрезке $[l, r]$ пока $r - l > 2\varepsilon$ повторять . уменьшать отрезок, не выпуская из него \sqrt{a} конец_пока { $r - l < 2\varepsilon, \sqrt{a}$ содержится в $[l, r]$ ответ: $(r + l)/2$

В качестве ответа мы взяли середину отрезка: после окончания исполнения команды повторения условие $r - l > 2\varepsilon$ не выполнено, значит, длина отрезка $[l, r]$ не превышает 2ε и его середина отстоит от любой его точки, в том числе и от \sqrt{a} , не более чем на ε ,

Теперь надо научиться двум вещам: (1) находить отрезок, содержащий \sqrt{a} ; (2) уменьшать его, не нарушая этого свойства. Задача (1) проста: мы знаем, что $\sqrt{a} \geq 0$ (корень всегда неотрицателен), так что левым концом отрезка можно считать 0. Если $a \geq 1$, то $\sqrt{a} \leq a$ и правым концом можно считать a ; если же $a \leq 1$, то $\sqrt{a} \leq 1$ и правым концом можно считать 1. Получаем

$\{a \geq 0, \varepsilon > 0\}$

$l \leftarrow 0$

выбор

. при $a \geq 1$ делай $r \leftarrow a$

. при $a \leq 1$ делай $r \leftarrow 1$

конец_выбора

$\{\sqrt{a} \in [l, r]\}$

(напомним: $l \leftarrow 0$ означает: положить l равным 0 и т. д.).

Перейдем ко второй задаче. Здесь ключевая идея такова: если \sqrt{a} содержится в каком-то отрезке, то он попадает в одну из его половин. Эту половину и можно взять в качестве нового, уменьшенного отрезка. В следующей программе мы вычисляем середину отрезка, а затем решаем, какая половина нам подходит, и в зависимости от этого сдвигаем левую или правую границу:

$\{\sqrt{a} \in [l, r]\}$

$m \leftarrow (l + r)/2$

выбор

. при $\sqrt{a} \in [l, m]$ делай $r \leftarrow m$

. при $\sqrt{a} \in [m, r]$ делай $l \leftarrow m$

конец_выбора

$\{\sqrt{a} \in [l, r]\}$

В этом месте читатель, вероятно, обвинит нас в жульничестве: как мы можем узнать, находится ли \sqrt{a} в $[l, m]$, если мы не знаем, чему равно \sqrt{a} — отыскание \sqrt{a} и является нашей главной целью! К счастью, можно проверить принадлежность числа \sqrt{a} к произвольному отрезку $[p, q]$ и не вычисляя \sqrt{a} : при $p, q \geq 0$

$$\sqrt{a} \in [p, q] \Leftrightarrow p^2 \leq a \leq q^2.$$

Это соотношение позволяет заменить проверки в команде выбора на равносильные, но понятные исполните-

лю (мы предполагаем, что наш исполнитель умеет складывать, вычитать, умножать, делить и сравнивать рациональные числа).

Примененный нами метод (его называют половинным делением, или бисекцией) допускает обобщение. Мы, по существу, решали уравнение $x^2 - a = 0$. Аналогично можно решать и другие уравнения вида $f(x) = 0$. В этом случае можно искать «отрезок перемены знака для f », т. е. такой отрезок $[l, r]$, что $f(l) \geq 0$, $f(r) \leq 0$ или $f(l) \leq 0$, $f(r) \geq 0$. Если функция f достаточно хороша (например, непрерывна), то всякий отрезок перемены знака содержит корень. В этом случае мы можем, начав с некоторого отрезка перемены знака, найти приближение с заданной точностью к (некоторому) корню:

```

{[l, r] — отрезок перемены знака} .
пока  $r - l > 2\varepsilon$  повторять
   $m \leftarrow (l + r)/2$ 
  . выбор
  . . при  $[l, m]$  — отрезок перемены знака делай
  . .    $r \leftarrow m$ 
  . . при  $[m, r]$  — отрезок перемены знака делай
  . .    $l \leftarrow m$ 
  . конец_выбора
конец_пока
{[l, r] — отрезок перемены знака,  $r - l \leq 2\varepsilon$ }
```

Правильность программы вытекает из того, что если $[l, r]$ — отрезок перемены знака, а m — любое число, то хотя бы один из отрезков $[l, m]$ и $[m, r]$ является отрезком перемены знака.

Отметим, что этот способ позволяет найти лишь приближение к одному из корней (которых на отрезке перемены знака может быть много).

Мы не сказали до сих пор, почему наши программы кончают работу, т. е. почему длина отрезка рано или поздно станет меньше 2ε . На этот счет есть специальная теорема: если каждый член последовательности вдвое меньше предыдущего, то рано или поздно члены этой последовательности станут меньше любого наперед заданного положительного числа, в том числе и 2ε . Как говорят, последовательность сходится к 0. Можно оценить число повторений, заметив, что при каждом повторении длина отрезка уменьшается в 2 раза. Отсюда следует, что после k повторений длина отрезка уменьшится в 2^k раз. Так как $2^3 < 10 < 2^4$,

видно, что для уменьшения длины отрезка в 10 раз нужно сделать от трех до четырех повторений и тем самым получение каждой следующей цифры десятичной искомого корня (соответствующее сужению отрезка в 10 раз) требует от трех до четырех повторений.

Возникает вопрос о том, нельзя ли придумать более экономный алгоритм нахождения квадратного корня. Оказывается, можно. Идея его такова. Пусть мы уже нашли какое-то приближение к \sqrt{a} . Обозначим его через x . Истинное значение корня отличается от x на какую-то величину; обозначим ее через h : $\sqrt{a} = x + h$. Тогда

$$a = (\sqrt{a})^2 = (x + h)^2 = x^2 + 2xh + h^2.$$

Можно надеяться, что h мало (т. е. исходное приближение достаточно точно). В этом случае h^2 еще меньше. Выкидывая его, получаем приближенное равенство $a \approx x^2 + 2xh$, откуда $h \approx (a - x^2)/2x$. Положив h равным $(a - x^2)/2x$, можно надеяться, что число $x + h$ будет новым, более точным приближением к \sqrt{a} . Подытоживая сказанное, мы получаем способ уточнения приближений к корню: можно надеяться, что если x близко к \sqrt{a} , то

$$x + h = x + \frac{a - x^2}{2x} = \frac{2x^2 + a - x^2}{2x} = \frac{x^2 + a}{2x}$$

будет еще ближе. Отметим, кстати, что $\frac{x^2 + a}{2x}$ можно переписать как $(x + (a/x))/2$, т. е. как среднее арифметическое чисел x и a/x . Если $x = \sqrt{a}$, то x и a/x равны и новое приближение к \sqrt{a} , как и следовало ожидать, совпадает с \sqrt{a} .

Сказанное наводит на мысль о таком способе отыскания \sqrt{a} : начав с какого-то приближения x_0 , строить все лучшие приближения по формулам

$$x_1 = \frac{x_0^2 + a}{2x_0}, \quad x_2 = \frac{x_1^2 + a}{2x_1} \dots x_{n+1} = \frac{x_n^2 + a}{2x_n}.$$

Строго говоря, все наши рассуждения исходили из того, что h было достаточно малым, так что этот способ пока выглядит совершенно необоснованным. Тем не менее попробуем его применить.

Пр и м е р. Ищем $\sqrt{2}$, положив $x_0 = 1$:

$$x_1 = \frac{1+2}{2} = 1,5, \quad x_2 = \frac{(1,5)^2 + 2}{2 \cdot 1,5} = 1,4166 \dots,$$

$$x_3 = 1,414215 \dots \text{ и т. д. } (\sqrt{2} = 1,414213 \dots)$$

Чтобы получить алгоритм отыскания корня, нам осталось ответить на три вопроса: (1) как выбирать начальное приближение; (2) когда останавливаться; (3) почему этот метод действительно дает нужный результат. Ответим последовательно на эти вопросы. В качестве начального приближения будем брать любое положительное число, например 1. Если x — любое положительное число, то x и a/x лежат по разные стороны от \sqrt{a} . Поэтому отклонение x от \sqrt{a} не больше отклонения x от a/x . Таким образом, достаточно продолжать вычисления до тех пор, пока $|x - a/x|$ не станет меньше заданного ε :

$\{a \geq 0, \varepsilon > 0\}$ $x \leftarrow 1$ пока $ x - a/x \geq \varepsilon$ повторять $x \leftarrow (x^2 + a)/2x$ конец_пока

Осталось ответить на третий, самый трудный вопрос: почему эта программа кончит работу? Чтобы ответить на него, нужно доказать, что разница между очередными приближениями к \sqrt{a} и самим \sqrt{a} становится сколь угодно малой. Это выражают словами: последовательность x_n приближений сходится к \sqrt{a} . Если мы это докажем, то по одной из стандартных теорем математического анализа (о предельном частном) можно будет утверждать, что и разница между a/x_n и \sqrt{a} становится сколь угодно малой. Тем самым и разница между x_n и a/x_n становится сколь угодно малой, так как x_n и a/x_n становятся сколь угодно близкими к \sqrt{a} .

Приступая к доказательству, обозначим последовательные приближения через x_n : $x_0 = 1$, $x_{n+1} = (x_n^2 + a)/2x_n$.

Л е м м а 1. Все x_n положительны.

В самом деле, свойство «быть положительным» является инвариантом: если $x_n > 0$, то и $x_{n+1} > 0$, так как и числитель $x_n^2 + a$, и знаменатель $2x_n$ положительны. Лемма доказана.

Обозначим через h_n отклонение x_n от \sqrt{a} : $x_n = \sqrt{a} + h_n$. Тогда

$$h_{n+1} = x_{n+1} - \sqrt{a} = \frac{x_n^2 + a - 2x_n\sqrt{a}}{2x_n} = \\ = \frac{(x_n - \sqrt{a})^2}{2x_n} = \frac{h_n^2}{2x_n}.$$

Отсюда вытекает

Л е м м а 2. $h_n \geq 0$ при $n \geq 1$. Другими словами, $x_n \geq \sqrt{a}$ при $n \geq 1$.

Так как $h_n = x_n - \sqrt{a} \leq x_n$, то при $n \geq 1$ имеем

$$h_{n+1} = h_n^2 / 2x_n \leq h_n x_n / 2x_n \leq h_n / 2.$$

Отсюда следует, как уже говорилось, что с ростом n число h_n становится меньше любого наперед заданного положительного числа и тем самым x_n становится сколь угодно близким к \sqrt{a} , что и требовалось доказать.

Отметим, что из формулы $h_{n+1} = h_n^2 / 2x_n$ вытекает, что, став малым, h_n убывает с ростом n чрезвычайно быстро, так как квадрат малой величины еще гораздо меньше. Говоря совсем грубо, с каждой итерацией число верных десятичных цифр возрастает примерно вдвое.

У п р а ж н е н и е. Придумать аналогичный способ для отыскания кубического корня из заданного числа.

Мы привели два алгоритма вычисления квадратного корня с заданной точностью. Поучительна разница между ними. Первый алгоритм не использует почти никакой специфики задачи: по существу, уравнение $x^2 - a = 0$ решается общим методом, пригодным и для других уравнений вида $f(x) = 0$. Доказательство его правильности очень просто. Второй алгоритм сложнее: не сразу ясно, можно ли его обобщить на другие уравнения вида $f(x) = 0$. Оказывается, что можно, но для этого по крайней мере нужно уметь вычислять производную функцию f . Доказательство его правильности сложнее и потребовало определенной математической изобретательности. Зато он и эффективнее, притом намного.

ПЕРЕБОРНЫЕ ЗАДАЧИ

Мы уже видели, каким образом можно доказывать, что программа кончает работу. Однако, честно говоря, для нас нет особой разницы между программой, не заканчивающей своей работы никогда, и программой, работающей 1000 лет.

Существует целый класс задач, которые попадают в категорию теоретически разрешимых, но практически неосуществимых. Этот класс — класс «переборных задач». Прежде чем дать его описание, приведем пример одной задачи такого рода.

Имеется набор грузов известного веса и машина известной грузоподъемности (веса грузов и грузоподъемность машины — целые числа). Мы должны определить, можно ли полностью загрузить машину, поместив в нее некоторые из имеющихся грузов (т. е. существует ли набор грузов с суммарным весом, равным грузоподъемности машины).

Теоретическая разрешимость этой задачи очевидна. Если у нас имеется n грузов, то возможных комбинаций этих грузов будет 2^n . Перепробовав их все, можно узнать, есть ли среди них комбинация, полностью загружающая машину. Оценим, однако, число операций, необходимых для выполнения этого алгоритма. Если $n = 100$, то $2^n = 2^{100} = (2^{10})^{10} \approx 1000^{10} = 10^{30}$, так что нужно перебрать около 10^{30} вариантов. Сколько же времени будет работать машина? Пусть мы располагаем компьютером даже не сегодняшнего, а завтрашнего дня, который выполняет миллиард (10^9) операций в секунду. На проверку одного набора уходит никак не меньше одной операции. Значит, понадобится не менее $10^{30}/10^9 = 10^{21}$ секунд, что больше 10^{13} лет. Видно, что с любой практической точки зрения этот алгоритм непригоден.

Не спасет дело и рост скорости вычислительной техники: если быстродействие компьютера возрастет в 10 раз, то за такое же время мы сможем решать задачу с тремя — четырьмя лишними грузами. (Возрастание 2^n в 10 раз соответствует увеличению n на три — четыре единицы). Единственный выход — это изменить алгоритм, избавившись от необходимости перебирать огромное число вариантов. В настоящее время

этого никто (для описанной задачи) делать не умеет. Более того, есть причины полагать, что это в принципе невозможно. Постараемся объяснить, в чем тут дело.

Опишем — по необходимости неформально — некоторый класс задач, называемых переборными. В этих задачах по заданным исходным данным (в нашей задаче это были набор весов предметов и грузоподъемность) и требуется найти некоторый ответ (у нас — загружающий машину полную набор грузов) или хотя бы узнать, существует ли таковой. При этом про каждого кандидата в ответы можно легко выяснить, годится ли он (в нашем случае нужно просто сложить веса грузов и сравнить сумму с грузоподъемностью), зато таких кандидатов очень много. Мы не будем описывать класс переборных задач более формально и ограничимся уже сказанным; заинтересованные читатели могут обратиться к книге Ахо, Хопкрофта, Ульмана «Построение и анализ вычислительных алгоритмов», о которой говорится в последней главе нашей книги.

Так вот, было доказано, что задача о загрузке машины является «полной переборной задачей». Это значит, что если бы ее можно было решать быстро (точнее, если бы число шагов, необходимых для ее решения, было ограничено многочленом от числа грузов), то любую (!) переборную задачу можно было бы также решать быстро. Это было бы очень хорошо, потому что среди переборных задач есть множество практически важных, — слишком хорошо, чтобы быть правдой. Поэтому большинство современных специалистов по «теории сложности вычислений» полагают, что это не так. Хотя доказать невозможность быстрого решения переборных задач пока никто не может.

Поясним, каким образом из быстрой разрешимости одной задачи может вытекать быстрая разрешимость другой. Мы проиллюстрируем это на примере. Рассмотрим две такие задачи.

Задача 1 (о раскраске графа). Дан граф — набор из нескольких точек (вершин), некоторые из которых соединены линиями (дугами), и число h . Требуется определить, можно ли так раскрасить вершины в h цветов, чтобы любые две точки, соединенные дугой, были раскрашены по-разному.

Задача 2 (о разбиении). Дано множество S и некоторое семейство его подмножеств. Требуется вы-

яснить, можно ли из этого семейства выбрать несколько попарно непересекающихся множеств, объединение которых равно S .

Покажем, что если вторая задача может быть быстро решена, то и первая — тоже. Пусть есть некоторый алгоритм, быстро решающий вторую задачу, а нам нужно быстро решить первую и узнать, можно ли раскрасить граф G в h цветов с соблюдением требований. Построим множество S , элементами которого будут все вершины графа, а также все пары \langle дуга графа, цвет \rangle (эти пары естественно называть покрашенными дугами). Обратите внимание, что в задаче о раскраске нужно красить вершины графа, а в множество S входят покрашенные дуги — каждая дуга h раз, поскольку цветов h . Опишем теперь семейство подмножеств множества S . Во-вторых, отнесем к нему все одноэлементные подмножества с элементами, являющимися покрашенными дугами. Далее, для каждой вершины и каждого цвета образуем множество, содержащее эту вершину и все выходящие из нее дуги, покрашенные в этот цвет. (Таких множеств будет столько, сколько пар \langle вершина, цвет \rangle .)

Мы утверждаем, что граф можно раскрасить в h цветов с соблюдением требования «концы любой дуги окрашены по-разному» тогда и только тогда, когда из указанных множеств можно построить разбиение множества S на непересекающиеся множества. Таким образом мы свели задачу о раскраске к задаче о разбиении: для последней есть быстрый алгоритм, то его можно применить и для быстрого решения первой.

Осталось доказать сформулированное утверждение. Посмотрим, как может выглядеть разбиение множества S . Всякая вершина графа входит в множество S и, следовательно, должна входить в некоторое множество разбиения. Она не может туда входить одна: по построению она входит в него вместе со всеми дугами какого-то цвета, примыкающими к этой вершине. Эти цвета для вершин, соединенных дугой, должны быть разными, иначе одна и та же дуга, покрашенная в один и тот же цвет, входила бы в два множества разбиения и они бы пересеклись. Таким образом, если разбиение существует, то существует и раскраска графа. Обратное: если существует раскраска графа, при которой концы любой дуги раскрашены по-разному, то, взяв каждую вершину графа вместе со всеми примыкающими к

ней дугами, покрашенными в цвет этой вершины, мы получим непересекающиеся подмножества S . Они не будут образовывать разбиения, так как не все покрашенные дуги в них входят. Но у нас в семействе есть одноэлементные множества, соответствующие всем покрашенным дугам, так что мы можем нужные из них добавить и получить разбиение. Таким образом, из существования раскраски вытекает существование разбиения. Наше утверждение доказано.

В этом месте читатель может спросить нас: ну хорошо, вы продемонстрировали, как одна конкретная задача сводится к другой. Но каким образом можно доказать, что всякая переборная задача сводится к данной, ведь их же бесконечно много? К сожалению, ответ на этот вопрос предполагает знакомство с математической логикой и теорией алгоритмов и выходит за рамки нашей книжки. Его можно найти в упоминавшейся книге Ахо, Хопкрофта, Ульмана. Заметим, что после того, как полнота какой-то одной переборной задачи доказана, можно доказать полноту другой задачи, сведя к ней первую. Например, известно, что задача о раскраске является полной, т. е. любая переборная задача к ней сводится. Поэтому приведенное выше рассуждение показывает, что и задача о разбиении является полной.

Однако, к счастью, не все переборные задачи являются полными. Для некоторых переборных задач удастся придумать алгоритм, позволяющий избавиться от перебора, хотя часто это требует большой изобретательности. Мы приведем одну такую задачу вместе с ее решением.

Задача о раскрое многоугольника. Дан выпуклый n -угольник. Его раскроем назовем разбиение его на треугольники с помощью $n - 3$ непересекающихся диагоналей. Стоимостью раскроя называется суммарная длина проведенных диагоналей. Требуется найти наименьшую возможную стоимость раскроя.

Казалось бы, для ее решения нужно перебрать все способы раскроя и сравнить суммарную длину разрезов (диагоналей) в каждом из них. Для четырехугольника есть 2 раскроя (рис. 26), для пятиугольника — 5 раскроев (рис. 27), для шестиугольника — 14 раскроев (рис. 28) и т. д. Можно убедиться, что с ростом n число раскроев растет очень быстро и их перебор

Рис. 26



Рис. 27

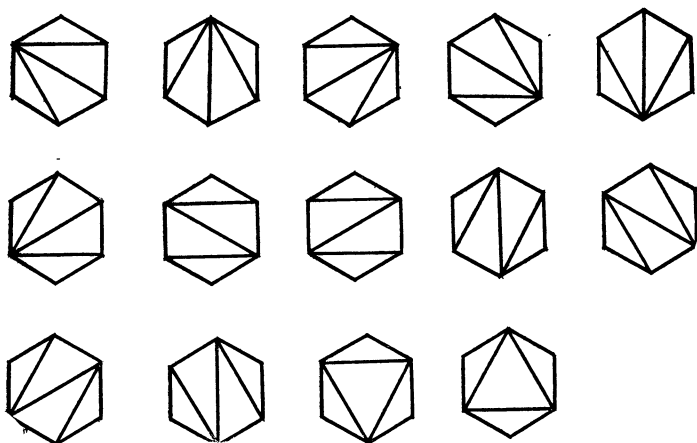


Рис. 28

скоро становится практически невозможным. Тем не менее задача может быть решена сравнительно просто.

Общая идея, которая оказывается здесь применимой, — «разделяй и властвуй» — сведение задачи к нескольким подзадачам того же типа, но меньшего размера. Проиллюстрируем экономию, достигаемую таким способом, на модельном примере. Зафиксируем произвольную диагональ многоугольника и будем временно рассматривать только раскрои, включающие данную диагональ. Чтобы найти среди них раскрой наименьшей стоимости, нужно соединить наилучшие раскрои отдельно для левой и правой частей много-

угольника, на которые он разбивается этой диагональю. Выгода, доставляемая этим простым приемом, видна из такой простой оценки: если одна часть многоугольника допускает k раскроев, а другая — l , то всего раскроев (включающих эту диагональ) kl . Используя сказанное, мы ограничиваемся рассмотрением отдельно k и l раскроев, всего $k + l$, что может быть существенно меньше kl .

Таким способом можно найти наилучший раскрой, включающий данную диагональ. Сделав это для каждой из диагоналей многоугольника, можно найти наилучший его раскрой. Это соображение уже существенно помогает, если в нашем распоряжении есть много одинаковых исполнителей. Возьмем столько пар исполнителей, сколько диагоналей в нашем многоугольнике. Каждой паре дадим части многоугольника, на которые он разбивается одной из диагоналей. Когда они представят нам наилучшие раскрои частей, соединим их в раскрой всего многоугольника и найдем среди полученных раскроев наилучший. При этом каждый из исполнителей может решать поставленную перед ним задачу тем же способом, разбив ее на меньшие части и поручив их решение своим подчиненным и т. д. Такая организация работ позволит получить ответ довольно быстро, особенно если нам удастся добиться, чтобы число вершин многоугольников при каждом разбиении существенно уменьшалось (например, на одну треть). Попробуйте понять, можно ли этого достичь.

Однако если исполнитель у нас один, указанная тактика не дает хорошего результата. Нам придется существенно изменить ее, рассматривая треугольники вместо диагоналей. Начнем с нескольких очевидных замечаний.

Всякая сторона многоугольника, разрезанного на треугольники непересекающимися диагоналями, является стороной некоторого треугольника раскроя (рис. 29). Стоимость раскроя, содержащего указанный треугольник, есть сумма стоимостей раскроев двух полученных многоугольников плюс длина двух проведенных диагоналей. Таким образом, чтобы найти стоимость самого дешевого раскроя многоугольника среди всех раскроев, содержащих этот

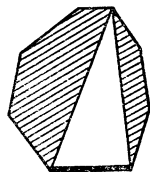


Рис. 29

треугольник, надо сложить стоимости наилучших раскроев полученных многоугольников и прибавить к ним длины диагоналей. А стоимость раскроя, являющегося наилучшим среди всех, получится, если взять минимум по всем треугольникам с данной стороной. (Возможен и случай, когда один из двух получающихся многоугольников вырождается в отрезок.)

Заметим, что при таком сведении раскроя многоугольника к раскрою двух других в этих двух многоугольниках одна сторона является диагональю исходного, а остальные — его сторонами, выделенная же первоначально сторона вовсе не входит в новые многоугольники. Это замечание, на первый взгляд выглядящее как не очень относящееся к делу, в действительности является решающим для построения эффективного алгоритма.

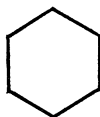
Объясняется это следующим. Мы указали, как найти стоимость наилучшего раскроя многоугольника, если известны стоимости наилучшего раскроя других, меньших. Чтобы извлечь максимальную выгоду из этого способа, нам понадобится таблица, содержащая сведения о стоимостях раскроя. Такая таблица будет приемлемой, лишь если ее объем (количество многоугольников, для которых мы находим и запоминаем стоимость их наилучшего раскроя) не слишком велик. На первый взгляд кажется, что необходимо иметь в таблице все многоугольники, вершинами которых являются некоторые из вершин исходного. А их много (порядка 2^n). Оказывается, однако, что можно ограничиться хранением сведений лишь о некоторых многоугольниках, которые мы будем называть специальными.

Многоугольник будем называть специальным, если его вершины являются вершинами исходного многоугольника, а все стороны, кроме, быть может, одной, — сторонами исходного. Будем решать задачу о раскрое специального многоугольника указанным способом, выбрав в качестве выделенной стороны диагональ исходного многоугольника. В этом случае получающиеся два многоугольника снова будут специальными и будут иметь меньшее число сторон. Таким образом, нас будут интересовать лишь стоимости наилучшего раскроя специальных многоугольников, а их не так много.

Сколько? Подсчитаем: каждая диагональ исходного многоугольника делит его на два специальных многоугольника. Так может быть получено $n(n-3)$ специальных многоугольников (число диагоналей равно $n(n-3)/2$), и остается учесть исходный многоугольник. Всего получается, следовательно, не более n^2 специальных многоугольников.

Построив таблицу, содержащую графу для стоимости наилучшего раскроя каждого специального многоугольника, начнем ее постепенно заполнять. Заполнение будем производить в порядке возрастания числа сторон многоугольников. Для треугольников стоимость раскроя равна 0 (резать нечего).

Допустим, мы уже заполнили таблицу для всех специальных многоугольников с не более чем k вершинами. Теперь наша задача состоит в том, чтобы найти наименьшую стоимость раскроя для специальных $(k+1)$ -угольников. Нарисуем специальный $(k+1)$ -угольник; выделим (жирной линией) сторону, не являющуюся стороной исходного многоугольника. Нам надо найти стоимость самого дешевого его раскроя. Все раскрои делятся на классы в зависимости от того, в какой треугольник раскроя входит выделенная сторона (рис. 30). Чтобы найти наименьшую стоимость



Все раскрои делятся на такие типы:

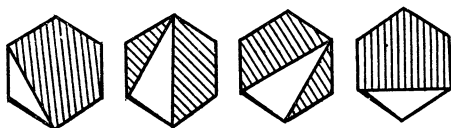
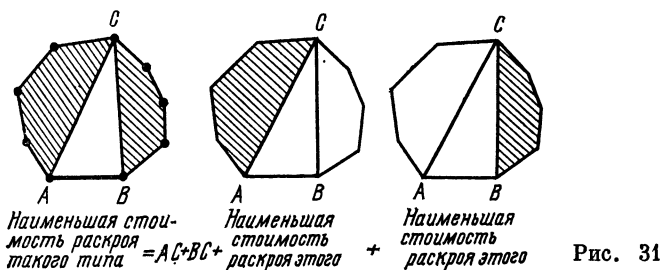


Рис. 30

раскроя, достаточно найти наименьшую стоимость раскроя в каждом из классов и затем выбрать минимальное среди этих чисел. Чтобы найти наименьшую стоимость раскроя в каждом классе, нужно сложить длины двух изображенных диагоналей со стоимостью раскроя двух возникающих многоугольников с мень-

шим числом сторон (рис. 31). Замечательным образом оказывается, что эти многоугольники также являются специальными, и стоимость их раскроя можно найти в таблице, так как число вершин в них меньше. Случай, когда один из них вырождается в отрезок, рассматривается аналогично.

Посмотрим, сколько действий нужно сделать для заполнения одной клетки таблицы. Нахождение мини-



мальной стоимости раскроя каждого типа (для данного треугольника), как мы видели, требует ограниченного числа действий (надо сложить два числа из таблицы и длины двух диагоналей). Типов раскроев не больше чем n (число вершин исходного многоугольника), поэтому общее число действий, необходимых для заполнения одной клетки таблицы, не превосходит $\text{const} \cdot n$. А для заполнения всей таблицы необходимо не более $\text{const} \cdot n^3$ действий. Это не так много, и задача остается практически разрешимой и для n порядка нескольких сотен.

ГЛАВА 15

ИГРЫ, ИГРЫ, ИГРЫ...

Эффектной демонстрацией «интеллектуальных» возможностей компьютеров являются программы, играющие в шахматы, шашки, калáх и т. д. Поэтому особенно интересно понять, как такие программы устроены «внутри». К сожалению, объяснение этого выходит за рамки нашей книжки, и мы ограничимся изложением свойств игр, лежащих в основе таких программ.

Прежде всего скажем о том, какие игры мы будем рассматривать. Мы предполагаем, что в игре участвуют двое, что они ходят по очереди и что каждый игрок рас-

полагает полной информацией о ходе игры. Тем самым, например, большинство карточных игр, где игроки не знают карт партнеров и карт, оставшихся в колоде, исключается. Мы предполагаем также, что в игре не может быть бесконечных партий (тем самым исключаются, например, крестики-нолики на бесконечной доске) и что всякая игра кончается победой одного из игроков (и проигрышем другого). Последнее требование (отсутствие ничьих) на самом деле несущественно, и в конце мы скажем, что будет, если от него отказаться.

Основной результат, который мы собираемся доказать, состоит в том, что всякая игра описанного типа «предопределена» — один из игроков может гарантировать себе выигрыш, если будет играть правильно. Разумеется, предварительно мы должны более точно описать рассматриваемый класс игр.

Говоря о позиции игры, мы будем включать в нее всю необходимую для продолжения игры информацию. В частности, позиция несет в себе информацию о том, кому ходить. Есть и менее очевидные сведения: например, позиция в шахматах включает в себя сведения о том, ходил ли король, — иначе нельзя определить, возможна ли рокировка; еще более сложная информация нужна для правила троекратного повторения ходов.

Правила игры определяют, какие ходы возможны в данной позиции, т. е. какие позиции могут из нее получиться. (Говоря формально, можно отождествить правила игры с множеством допустимых ходов — множеством пар, состоящим из позиций до и после хода). Требование очередности ходов предполагает, что во всех позициях, получающихся из данной, очередь хода — за другим игроком. Заключительными будем называть те позиции, в которых продолжение игры невозможно. Все заключительные позиции делятся на выигрышные и проигрышные (для того, чья очередь хода).

Наконец, мы предполагаем, что среди позиций выделена некоторая начальная позиция — та, с которой начинается партия.

Принципиальная особенность рассматриваемых нами игр состоит в следующем условии завершения: не существует бесконечных последовательностей ходов.

В качестве иллюстрации нашей терминологии рассмотрим как пример игру из школьного учебника информатики. Она названа там игрой Баше. Имеется

15 предметов и каждый из игроков (как обычно, будем называть участников Белыми и Черными) за один ход может взять один, два или три предмета. Тот, кому достался последний предмет, проигрывает. В наших терминах игра описывается так. Позиция задается указанием числа оставшихся предметов и очередности хода. Правила игры разрешают переход из позиции $\langle n, \text{Белые} \rangle$ в позицию $\langle k, \text{Черные} \rangle$, если n больше 1, k больше 0 и k равно $n - 1$, $n - 2$ или $n - 3$. (Аналогично для переходов от позиции $\langle n, \text{Черные} \rangle$ к позиции $\langle k, \text{Белые} \rangle$.) Заключительными являются позиции $\langle 1, \text{Белые} \rangle$ и $\langle 1, \text{Черные} \rangle$, являющиеся проигрышными для Белых и Черных соответственно. Начальной позицией является $\langle 15, \text{Белые} \rangle$.

Теперь можно сформулировать наше утверждение о «предопределенности» точнее. Назовем партией последовательность позиций, в которой:

- (1) первая позиция — начальная позиция игры;
- (2) каждая следующая позиция получается из предыдущей по правилам игры (т. е. с помощью допустимого хода);

(3) последняя позиция — заключительная позиция игры. Результат партии (кто выиграл) определяется тем, является ли заключительная позиция выигрышной или проигрышной и кто из игроков в ней оказался.

Пример партии в игре Баше: $\langle 15, \text{Белые} \rangle$, $\langle 12, \text{Черные} \rangle$, $\langle 11, \text{Белые} \rangle$, $\langle 9, \text{Черные} \rangle$, $\langle 6, \text{Белые} \rangle$, $\langle 3, \text{Черные} \rangle$, $\langle 1, \text{Белые} \rangle$. В этой партии Белые проиграли.

Стратегией мы будем называть правило, определяющее, в какой позиции как ходить. (Формально стратегия игрока заключается в сопоставлении с каждой позицией, в которой ход за ним и которая не является заключительной, некоторой позиции, в которую можно попасть из этой.) Говорят, что в данной партии игрок следовал данной стратегии, если все его ходы сделаны в соответствии с ней. Говорят, что стратегия является выигрышной, если все партии, в которых игрок ей следует, оканчиваются его победой. Другими словами, выигрышная стратегия — это правило, гарантирующее выигрыш при любой игре противника.

На этом все нужные нам определения заканчиваются. Чтобы разобраться с ними, попытаемся ответить на такой вопрос: могут ли в одной игре оба противника иметь выигрышную стратегию?

Как и следовало ожидать, такого быть не может.

В самом деле, пусть оба противника имеют выигрышную стратегию. Предоставим им возможность сыграть друг с другом, руководствуясь этими стратегиями. В силу условия завершения партия должна оборваться на заключительной позиции. В ней выиграл один из игроков. Следовательно, стратегия, которой руководствовался проигравший, не выигрышная.

Более сложным оказывается такой вопрос: бывают ли игры нашего класса, в которых ни один из противников не имеет выигрышной стратегии? Как мы увидим, такого быть не может, но доказательство этого потребует от нас существенно больших усилий.

Начнем с такого замечания. Наше определение игры предполагает, что все партии игры начинаются с одной и той же начальной позиции. Тем не менее можно (по примеру шахматных композиций) рассматривать и партии, начинающиеся с произвольно заданной позиции. Здесь есть четыре возможности:

(1) существует выигрышная стратегия для начинающего (того, чья очередь ходить в данной позиции) и нет выигрышной стратегии для его противника;

(2) не существует выигрышной стратегии для начинающего, но существует выигрышная стратегия для его противника;

(3) ни у того, ни у другого нет выигрышной стратегии;

(4) и у того и у другого есть выигрышная стратегия. Невозможность случая (4) доказывается, как и раньше, легко. Мы же хотим доказать, что невозможен и случай (3).

Будем называть позицию выигрышной, если имеет место случай (1); проигрышной, если (2); и нейтральной, если (3). Эта классификация, очевидно, продолжает заданную правилами игры классификацию заключительных позиций на выигрышные и проигрышные, позволяя говорить о выигрышных или проигрышных незаключительных позициях. Мы хотим доказать, что нейтральных позиций нет вовсе. При этом нам понадобится такая лемма.

Л е м м а. Пусть S — не заключительная позиция. (1) Если любая позиция T , которая может быть получена из S ходом игры, является выигрышной, то S — проигрышная позиция. (2) Если среди позиций T , которые могут быть получены из S ходом игры, имеется проигрышная, то S — выигрышная позиция.

Доказательство. (1) Если игра начинается с позиции S , то после первого хода начинающего возникнет позиция T , являющаяся выигрышной, т. е. для нее существует выигрышная стратегия. Одной из таких стратегий (в зависимости от хода начинающего) и должен руководствоваться второй игрок.

(2) Начинаящий должен сделать ход, переводящий позицию S в проигрышную позицию T . После этого он должен следовать выигрышной стратегии для второго игрока, которая по предположению имеется в позиции T . Лемма доказана.

Следствие. Пусть S — незаклывительная позиция, являющаяся нейтральной. Тогда существует нейтральная позиция T , которая может быть получена из S ходом игры.

Доказательство. Пусть среди позиций T , получаемых из S ходом игры, нет нейтральных. Тогда или все они выигрышные (а S , по лемме, проигрышная), либо среди них есть проигрышная (и тогда S выигрышная).

Теперь легко доказать отсутствие нейтральных позиций. В самом деле, если нейтральная позиция существует, то, согласно следствию, из нее ходом игры может быть получена другая нейтральная, из этой другой — третья и так далее до бесконечности. А это противоречит условию завершения.

Итак, сформулированное нами утверждение о «предопределенности» любой игры рассматриваемого класса доказано. Скажем несколько слов о том, что будет, если допустить ничьи. При этом все позиции разделятся на три категории — выигрышные (в которых начинающий имеет выигрышную стратегию), проигрышные (в которых его противник имеет выигрышную стратегию) и ничейные (в которых оба игрока имеют стратегии, гарантирующие им, по крайней мере, ничью).

Упражнение. Докажите это. (Указание: наряду с игрой, в которой возможны ничьи, рассмотрите две ее модификации, в которых ничья считается проигрышем одного из игроков. Примените к ним доказанное нами утверждение.)

Почему же игры описанного класса по-прежнему остаются привлекательными? Почему мы с увлечением играем, например, в шахматы — в игру, результат которой предопределен? Дело в том, что это предопределение нам неизвестно. В большинстве встречающихся

игр не придумано никакого способа выяснить проигрышность или выигрышность позиции, который не требовал бы полного или почти полного просмотра всех возможных позиций. А число всех возможных позиций, как правило, необозримо велико. Из-за этого не удастся написать программу для компьютера, реализующую оптимальную стратегию игры, например, в шахматы (и работающую приемлемое время). Поэтому создатели программ, «играющих в шахматы», вынуждены довольствоваться программами, которые хотя и не являются оптимальными и могут проигрывать в выигрышных позициях, но проявляют это свойство лишь в игре с достаточно сильными партнерами.

Тем не менее существуют примеры игр, в которых выигрышная стратегия может быть описана явно. Такие игры представляют собой излюбленную тему для занятий математических кружков. Несколько примеров таких игр будет разобрано в главе 16 «Снова об играх».

З а м е ч а н и е д л я з н а т о к о в. Наше условие завершения не запрещает игр, в которых из одной и той же позиции возможно сделать бесконечное число различных ходов (и, следовательно, количество возможных позиций бесконечно).

ГЛАВА 16

СНОВА ОБ ИГРАХ

В главе «Игры, игры, игры...» мы показали, что с теоретической точки зрения исход всякой игры, в которой оба противника располагают полной информацией о действиях друг друга, нет элемента случайности и нет бесконечных партий, предрешен: для одного из игроков существует выигрышная стратегия или для обоих игроков существует стратегия, гарантирующая, по крайней мере, ничью. Тем не менее наличие такой стратегии не мешает играм оставаться увлекательными. Популярности шахмат не мешает то обстоятельство, что всезнающий мудрец мог бы снабдить изображение начальной позиции шахмат одной из трех надписей: «Белые начинают и выигрывают», «Черные выигрывают» или «Ничья». В чем же дело? Ответ очевиден — дело в том, что выигрышная (или гарантирующая

ничью) стратегия нам неизвестна! Доказанная нами в главе «Игры, игры, игры...» теорема является, как говорят, чистой теоремой существования. Она не позволяет определить, кто из партнеров имеет выигрышную стратегию и тем более не позволяет найти эту стратегию, если дерево игры, как это обычно бывает, необозримо велико.

В некоторых играх удастся доказать, что выигрышная или гарантирующая ничью стратегия имеется у данного игрока. Например, докажем, что в игре в крестики-нолики начинающий имеет стратегию, гарантирующую ему ничью. В самом деле, пусть это не так. Согласно доказанной нами теореме, это означало бы, что второй игрок имеет стратегию, гарантирующую ему выигрыш. Но в этом случае первый игрок мог бы также воспользоваться этой стратегией, пойдя на первом ходу как угодно и в дальнейшем играя так, как если бы этого хода не было. Поскольку лишний крестик не может помешать выигрышу, то следование в дальнейшем стратегии для второго игрока гарантирует выигрыш первому. А этого не может быть (у обоих игроков не может быть выигрышной стратегии одновременно). Полученное противоречие показывает, что у первого игрока всегда существует стратегия, гарантирующая ему по крайней мере ничью.

Для крестиков-ноликов на поле 3×3 такая стратегия хорошо известна. Однако наше рассуждение может быть применено и ко многим другим играм — ко всем тем, в которых можно «пропустить ход, не ухудшив своего положения». Оно, как и доказательство теоремы о существовании выигрышной стратегии для одного из игроков, не конструктивно и не дает никакого способа найти требуемую стратегию.

Во многих играх оказывается возможным указать такую стратегию в терминах «инварианта» — соотношения, выполнение которого нужно поддерживать, чтобы выиграть. Приведем несколько примеров.

М о н е т ы н а с т о л е. Двое по очереди кладут пятаки на прямоугольный стол. Проигрывает тот, кому некуда положить следующий пятак. Кто выигрывает при правильной игре?

Выигрывает первый, если он будет играть так. Первым ходом нужно положить пятак в центр стола. Затем нужно делать ходы, симметричные относительно центра стола ходам противника. Другими словами,

нужно поддерживать такое свойство: позиция на столе симметрична относительно центра стола.

О д н о с т о р о н н я я л а д ь я. Ладью, стоящую в левом нижнем углу шахматной доски, можно двигать либо на несколько клеток вправо, либо на несколько клеток вверх (но не влево и не вниз). Двое ходят по очереди; проигрывает тот, кто не может сделать ход (это бывает, когда ладья находится в верхнем правом углу). Кто выигрывает при правильной игре?

Выигрывает второй, если он будет ходить так, чтобы после каждого его хода ладья находилась на диагонали, ведущей из левого нижнего угла в правый верхний. В этом случае любой ход первого игрока неизбежно уводит ладью с этой диагонали и дает возможность второму сделать ход, возвращающий ладью на диагональ.

У п р а ж н е н и е. На столе лежат две кучи по 100 монет. Игроки ходят по очереди. За один ход игрок может взять любое число монет, но только из одной кучи. Проигрывает тот, кому не остается монет. Кто выигрывает при правильной игре? Как он должен играть?

Идея инварианта оказывается полезной и во многих других играх. Вот простейшее ее применение в шахматах: если белый король стоит на поле $a1$, а черный на поле $c1$ и сейчас ход белых, то черные могут не выпустить белого короля с вертикали a ; для этого им достаточно поддерживать такое свойство: белый и черный короли находятся на одной и той же горизонтали в 1-й и 3-й вертикалях. Гораздо более изощренное применение той же идеи позволяет белым достичь ничьей в шахматном этюде (рис. 32, см. стр. 120), заимствованном из книги Г. Штейнгауза «Математический калейдоскоп» (М.: Наука, 1981). «Эндшпиль доктора Эберса носит чисто математический характер. Можно точно доказать, что белые не позволят черному королю взять какую-либо из своих пешек, если только всегда будут ходить своим королем на поле, обозначенное той же буквой, что и поле, на котором стоит черный король. В соответствии с этим первый ход белых есть ход $B - F$. Если белые ни разу не отступят от этого правила, то результат будет ничейным. Если они нарушат его хотя бы однажды ... то черные добьются победы» (с. 12). Точнее было бы сказать, что эндшпиль носит «чисто программистский характер».

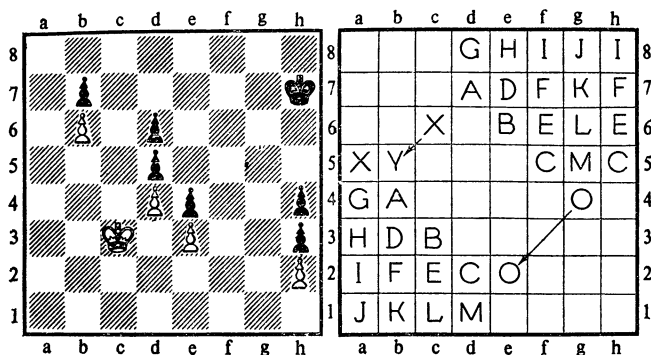


Рис. 32

Мы закончим эту главу задачей, имеющей интересную историю. Она приведена в книге Д. Гриса «Наука программирования» со ссылкой на Дж. Уильямса, который задал ее Грису в 1973 г. Нам она показалась знакомой, и после некоторых поисков мы нашли ее в сборнике задач московских математических олимпиад (М.: Просвещение, 1965), составленном А. А. Леманом. Вот она. На клетчатой бумаге играют двое: А и Б. Одним ходом можно соединить две соседние вершины, находящиеся на расстоянии 1. Доказать, что если первый ход сделал А, то Б может помешать А образовать из своих линий какой-нибудь замкнутый многоугольник (с. 116 сборника Лемана, с. 171 книги Гриса).

Решение просто, хотя додуматься до него едва ли легко: Б должен мешать А построить левый нижний угол! Если А проводит вертикальную линию, то Б должен провести горизонтальную линию из ее нижнего конца вправо; если А проводит горизонтальную линию, то Б должен провести вертикальную линию из ее левого конца вверх (если только соответствующие линии не были проведены им раньше). Другими словами, Б должен поддерживать истинность такого утверждения: если одна из сторон юго-западного угла проведена А, то другая сторона того же угла проведена Б.

ГЕНИАЛЬНЫЙ РЕЖИССЕР И ЕГО ЖЕРТВА

На киностудии «...фильм» работает Гениальный Режиссер. Естественно, его сотрудникам, как и руководству студии, с ним не сладко. Но что поделаешь — Гений.

Одна из гениальных причуд Гениального Режиссера состоит в том, что он общается со своими сотрудниками с помощью отснятых кинолент: на них он запечатлевает для исполнителя (и заодно для истории) свои гениальные указания, озарения, задумки, ходы и т. п. Расход пленки жуткий, но руководству студии приходится выкручиваться — Гений.

Достается от Гениального Режиссера (ГР) всем, но главная его жертва — безответный Монтажер (М). Дело в том, что ГР считает себя королем монтажа. И вот М получает от ГР указания одно нелепее другого. Почти каждое утро ему приносят указания ГР и рабочую пленку для монтажа, и он с ужасом садится за монтажный стол и начинает просмотр указаний, пытаясь понять, что же от него требуется. Однажды М для выполнения очередного указания нужно было выяснить, существует ли бесконечное число таких простых чисел p , что $p + 2$ тоже простое. Иногда же указания бывают просто невыполнимыми. Например, однажды М получил указание, суть которого сводилась к следующему: «взять рабочую ленту, отрезать у нее первый кадр, приклеить в конец, с полученной пленкой поступить снова так же и т. д.». Через некоторое время М понял, что указание невыполнимо: работа в соответствии с ним продолжалась бы бесконечно долго.

М пришел и сказал об этом ГР. Тот даже не извинился, а только сказал: «Ах, да, действительно, тут надо что-то подправить, впрочем, меня сейчас интересует совсем другое».

У М есть друг — заместитель директора (ЗД) студии, они живут неподалеку друг от друга и по вечерам вместе гуляют с собаками (у М — лайка, а у ЗД — пудель). М рассказал ЗД о бесчинствах ГР, но в ответ услышал: «Надо терпеть, но я постараюсь как-нибудь тебе помочь».

Однако время шло, указания ГР становились все более изощренными и запутанными. Выполнение неко-

торых из них требовало от М фантастической изобретательности. Другие оказывались невыполнимыми, но и это обнаружить было не просто.

И вот однажды весь «...фильм» содрогнулся от ужаса. Любимый всеми М был найден в монтажной мертвым, запутавшимся в ленте и указаниях ГР.

Можно представить себе состояние ЗД. Но что он мог сделать? Никто не осмеливался ограничить свободу творчества ГР. ЗД удалось добиться лишь создания комиссии, в задачу которой отныне входила оценка принципиальной выполнимости указаний ГР. Каждая пара выдаваемых им лент (указание и рабочая лента) поступали на комиссию. В задачу комиссии входило решить, можно ли выполнить указание (в применении к рабочей ленте). При этом комиссию не интересовали затраты времени и пленки, существенной была только выполнимость указания в принципе.

Комиссия начала работать, и своеволие ГР было хоть как-то ограничено. Но вот однажды на стол комиссии легла пара лент. Просмотр первой показал, что она содержит следующее указание: «Взять две копии рабочей ленты и отнести в комиссию в качестве ленты с указаниями и рабочей ленты. Если комиссия примет решение о невыполнимости указания, смыть рабочую ленту и кончить работу. Если комиссия примет решение о выполнимости указания, переставить первый кадр рабочей ленты в конец, потом опять и т. д.»

Комиссия просмотрела вторую ленту — рабочую. Она не отличалась от первой.

Комиссия начала заседание. Стемнело, похолодало, пошел снег, ... наступила удушающая жара, такой уже не было лет двести, кино перестали снимать на пленку, потом на магнитную ленту, потом вообще. Комиссия заседала...

Что же она решила?

Наш рассказ, по существу, представляет собой изложение в иносказательной форме доказательства одной из важнейших теорем теории алгоритмов. Эта теорема утверждает, что не существует программы, позволяющей определить, применима ли заданная программа к заданному исходному данному. Более подробно об этом можно узнать, прочитав статью В. А. Успенского, А. Л. Семенова «Решимые и нерешимые алгоритмические проблемы» (Квант. 1985. № 7).

РЕДАКТОР ТЕКСТОВ, ИЛИ ЗАЧЕМ КОМПЬЮТЕР ГРАМОТНОМУ

Разговоры о компьютерной грамотности могут создать впечатление, что в недалеком будущем каждый, или почти каждый, будет постоянно сталкиваться с компьютером. Пожалуй, это так и будет. Однако важно понимать, что использование компьютера вовсе не предполагает какого-либо программирования и тем более «общения с компьютером». К сожалению, об этом часто забывают, отождествляя компьютерную грамотность с умением кое-как программировать на каком-либо конкретном языке, чаще всего Бейсике. Между тем Э. Дейкстра считает, что «практически невозможно научить хорошо программировать студентов, первоначально ориентированных на Бейсик: как потенциальные программисты они умственно оболванены без надежды на исцеление». Даже коммерческие журналы по компьютерам печатают посреди многочисленных реклам предупреждения вроде такого:

ОСТОРОЖНО!!!

Занятие программированием может лишить Вас будущего.
Не думайте, что научившись программировать на Бейсике,
Вы чего-то добьетесь в жизни

Мы не вполне согласны с этим предостережением: программирование — вполне уважаемая профессия, чтение популярной книги по программированию (вроде нашей) и решение программистских задач ничем не опаснее чтения популярной книги по математике и решения математических задач. Не повредят и компьютерные игры в небольших дозах. Важно лишь не забывать о мудром правиле:

Когда в делах — я от веселий прячусь,
Когда дурачиться — дурачусь,
А смешивать два этих ремесла
Есть тьма искусников — я не из их числа.

Наш призыв к людям, не являющимся профессиональными программистами высокого класса:

Если Вы хотите получить от компьютера пользу,
НЕ ПРОГРАММИРУЙТЕ!!!

А зачем же тогда компьютер? — спросите вы. Действительно, сам по себе компьютер не нужен. Нужен компьютер, снабженный специальными программами, позволяющими применить его в профессиональной деятельности (увы, сейчас такие программы практически отсутствуют). Возможно, правильнее было расставить акценты иначе, сказав, что в первую очередь нужны такие программы и уж затем компьютер, на котором их можно выполнять, поскольку стоимость программ обычно выше стоимости самого компьютера.

Что же это за программы? В популярных книгах по кибернетике иногда можно прочесть, что нужны программы, превращающие компьютер в «интеллектуального», «дружественного» помощника, вступающего с вами в «диалог», понимающего ваш язык, много «знающего», приспособляющегося к вашим требованиям и т. п. Однако практика показывает, что это не так. Действительно удобные программы — те, при работе которых компьютер вообще перестает быть заметным, поскольку он делает то, что надо, и тогда, когда надо.

Расскажем об одном из видов весьма полезных компьютерных программ, называемых редакторами текстов, — они приходят на смену пишущим машинкам и могут быть полезны всякому, кто говорит тексты.

Итак, понаблюдаем за работой человека, пользующегося текстовым редактором. На первый взгляд все происходит, как в пишущей машинке. Человек нажимает на клавиши, и на экране появляются соответствующие буквы (в отличие от пишущей машинки — бесшумно). Как и в машинке, имеется клавиша «верхний регистр»: нажимая на нее, можно напечатать большую букву. Есть также специальная клавиша «возврат каретки». Нажав ее, можно перейти с одной строки на другую.

Стоп! Опечатка! Что же делать? Как убрать неверную букву? Для этого предусмотрена специальная клавиша «удалить». При нажатии на нее последний набранный символ бесследно исчезает и на его месте можно напечатать новый, правильный.

Но вот абзац закончен, и сочинитель смотрит на него критическим взглядом. Вот здесь опечатка, которую он сразу не заметил. А это слово неудачно и его надо бы заменить другим. Неужели весь абзац придется набирать заново? Нет. Можно вернуться к уже

набранной части текста и внести исправления. Как это сделать? Мы не сказали еще, что на экране, помимо набранного текста, имеется маленькая светящаяся точка, которую называют (не знаем, почему) курсором. Она находится в том месте, где будет напечатан следующий символ. Когда символ печатается, курсор автоматически сдвигается вправо, на следующую позицию. Помимо букв, на клавиатуре имеются специальные клавиши со стрелками →, ↓, ←, ↑. Нажатие на эти клавиши перемещает курсор на одну клетку в соответствующем направлении. С их помощью можно подогнать курсор к любому месту текста. После этого, нажав клавишу с буквой, напечатаем эту букву (вместо той, которая была на этом месте раньше).

Бывают, однако, и более сложные исправления. Вот это слово не на месте, и его надо убрать. Подводим курсор к началу слова, несколько раз нажимаем на специальную клавишу и слово исчезает буква за буквой, а остаток строки сдвигается влево на освободившееся место. А вот возникла обратная задача — нужно вставить слово в текст. Это столь же просто: есть специальная клавиша, при нажатии на которую строка раздвигается. Нажав на нее нужное число раз, мы освободим место для нового слова. Аналогично добавлению и удалению символов можно добавлять и удалять строки.

Приближается волнующий момент: уже почти весь экран заполнен текстом и скоро не останется места. Что же будет? Ничего страшного: текст на экране автоматически сдвигается и освобождается место для новой строки. Экран дисплея превращается в окно, через которое виден кусок текста. Когда мы двигаем курсор по тексту, окно само смещается вслед за курсором так, чтобы тот был виден. Благодаря этому можно редактировать длинные тексты, не уместяющиеся на экране.

Описанные действия не исчерпывают возможностей программы-редактора. Можно нажимать клавишу, и строки одна за другой пропадут с экрана. Вот уже исчез целый абзац. Через некоторое время нажмем еще одну клавишу, и весь «стертый» абзац появится в другом месте. Оказывается, его не стерли, а «запомнили».

Вот, наконец, все изменения внесены. Текст, однако, имеет «нетоварный» вид — одни строки длинные, другие короткие. Не беда: нажмем специальную кла-

вишу, и все строки будут выровнены до заданной длины, как это обычно делается в книге. Теперь осталось лишь напечатать его на печатающем устройстве (еще одно нажатие клавиши) и отнести в редакцию или выкинуть в корзину. (Увы, последнее приходится делать столь же часто, как и в старое доброе время: качество ваших сочинений целиком зависит от вас, и компьютер тут не помощник!)

ГЛАВА 19

ИСПОЛНИТЕЛЬ-ЧЕРЕПАХА, ИЛИ ЯЗЫК ЛОГО

Каждому программисту, а особенно начинающему, хочется испытать написанную им программу и убедиться, что она работает правильно. Это особенно приятно, если результатом работы программы является красивая картинка. Язык Лого, разработанный в Массачусетском технологическом институте под руководством С. Пейперта, предоставляет такую возможность уже при первых сеансах работы с компьютером. Он получил распространение как средство начального знакомства с возможностями ЭВМ (в частности, в Болгарии многие школы снабжены компьютерами, воспринимающими Лого). Мы расскажем о его возможностях (хотя не будем стремиться познакомить вас с деталями языка).

Главный исполнитель в языке Лого называется черепахой. Ее нужно представлять себе ползающей по плоскости и оставляющей на ней следы. Эти следы, как и положение черепахи, видны на экране. Тем самым, управляя движением черепахи, можно нарисовать на экране картинку.

Движения, повороты

В каждый момент времени черепаха находится в какой-то точке плоскости и смотрит в каком-то направлении. (В отличие от Путника, который мог смотреть лишь на Север, Запад, Восток и Юг, черепаха может смотреть в любую сторону.)

Команда FORWARD — вперед, не меняя направления взгляда черепахи, сдвигает ее в этом направлении, а команда BACK, т. е. назад, — в противополож-

ном. Каждая из этих команд требует указания расстояния, на которое должна сдвинуться черепаха. Например, после выполнения команд

```
| FORWARD 100  
| BACK 40  
| BACK 60
```

черепаха вернется в исходное положение.

Легко видеть, что в отсутствие других команд все движение черепахи происходило бы по прямой. Сойти с этой прямой позволяют команды поворота LEFT и RIGHT (влево, вправо). Каждая из них требует указания угла поворота: для LEFT — против часовой стрелки, для RIGHT — по часовой стрелке. Например,

```
| LEFT 140  
| LEFT 140
```

производит то же действие, что и

```
RIGHT 80
```

($140 + 140 + 80 = 360$). Умея поворачиваться, мы в принципе могли бы обойтись без команды BACK. Например,

```
| LEFT 180  
| FORWARD 3  
| LEFT 180
```

дает тот же результат, что и

```
BACK 3
```

У п р а ж н е н и е. Какой след останется от черепахи после выполнения такой программы:

```
| FORWARD 40  
| LEFT 45  
| BACK 20  
| RIGHT 90  
| FORWARD 20  
| LEFT 45  
| BACK 40
```

(Ответ: в форме буквы «М».)

Повторение команд

В языке Лого имеется конструкция, позволяющая выполнить команду или серию команд заданное число раз:

REPEAT число раз [повторяемая серия команд]
Например, команда

```
REPEAT 4 [FORWARD 10 RIGHT 90]
```

нарисует квадрат со стороной в 10 единиц.

У п р а ж н е н и е. Какие команды нужны, чтобы нарисовать (1) правильный пятиугольник? (2) пятиконечную звезду?

Новые команды

Мы знаем, что важным средством программирования является определение новых команд. Такая возможность есть и в Лого и выглядит это так:

```
TO имя_новой_команды
    команды, составляющие описание новой ко-
    манды
END
```

Например, можно определить команду рисования квадрата со стороной 10:

```
TO SQUARE
    REPEAT 4 [
        FORWARD 10
        RIGHT 90
    ]
END
```

(Как обычно, команды, входящие в состав команды повторения, выделяем с помощью отступов.)

Разумеется, хотелось бы определить команду, позволяющую рисовать квадраты произвольных размеров. Это делается так:

```
TO SQUARE :SIZE
    REPEAT 4 [
        FORWARD :SIZE
        RIGHT 90
    ]
END
```

После такого описания команда

```
SQUARE 50
```

нарисует квадрат со стороной в 50 единиц. В описаниях команд разрешается употреблять арифметические операции. Так, описание

```
TO POLIGON :N :SIZE
    REPEAT :N [
        FORWARD :SIZE
        RIGHT (360/:N)
    ]
END
```

определяет команду рисования правильного многоугольника с заданным числом сторон и с заданной длиной стороны.

Рекурсивные команды

Мы уже говорили, что в некоторых языках программирования в описании команды допускается использовать саму описываемую команду. Вот пример такого рода в Лого:

```
TO WHATISIT :SIZE
  FORWARD :SIZE
  RIGHT 90
  WHATISIT(:SIZE + 3)
END
```

Посмотрев на описание, можно предположить, что эта команда не заканчивает работы. Так и есть: запустив ее, мы видим, что на экране постепенно появляется спираль (рис. 33).

WHATISIT10

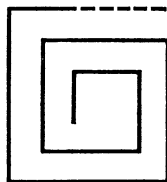


Рис. 33

Другие возможности языка

Мы описали лишь небольшую часть всех возможностей языка Лого — в основном те, которые связаны с рисованием. Помимо этого, в Лого имеются средства для выполнения арифметических действий, ввода и вывода, цветного рисования, исполнения простейших мелодий и другие. В целом Лого представляет собой вполне разумный язык для начального знакомства с возможностями компьютера. Не следует только углубляться в него слишком далеко и пытаться использовать его для того, для чего он не предназначен (например, для написания больших программ).

ГЛАВА 20

ИГРА В ЖИВОТНЫХ, ИЛИ ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Часто компьютеры называют «думающими машинами», говорят о «диалоге с компьютером» и даже об «искусственном интеллекте». Действительно, иногда компьютеры производят «разумное» впечатление. Но это всего лишь иллюзия, хотя и весьма убедительная. Чтобы продемонстрировать, как такая иллюзия может возникнуть, приведем сначала протокол сеанса работы с программой «Отгадывание животных», а затем объяс-

ним, как эта программа «устроена изнутри» (тексты, выведенные машиной, напечатаны прописными буквами).

ЗАДУМАЙ ЖИВОТНОЕ, А Я ПОСТАРАЮСЬ ЕГО УГАДАТЬ. КОГДА ЗАДУМАЕШЬ, НАПИШИ «ГОТОВО».

Готово.

У НЕГО ЧЕТЫРЕ НОГИ? (ОТВЕЧАЙ «ДА» ИЛИ «НЕТ»).

Да.

ГОВОРIT ЛИ ОНО «МУ...»?

Нет.

ЭТО СВИНЬЯ?

Нет.

УВЫ, Я ОШИБСЯ. КТО ЖЕ ЭТО?

Кошка.

СФОРМУЛИРУЙ, ПОЖАЛУЙСТА, ВОПРОС О ЖИВОТНОМ, ОТВЕТ НА КОТОРЫЙ БУДЕТ «ДА», ЕСЛИ ЭТО КОШКА, И «НЕТ», ЕСЛИ ЭТО СВИНЬЯ.

Ловит ли оно мышей?

СПАСИБО, Я ЭТО ЗАПОМНЮ. ХОЧЕШЬ СЫГРАТЬ ЕЩЕ?

Да.

ЗАДУМАЙ ЖИВОТНОЕ, А Я ПОСТАРАЮСЬ ЕГО УГАДАТЬ. КОГДА ЗАДУМАЕШЬ, НАПИШИ «ГОТОВО».

Готово. (Игрок снова задумал кошку.)

У НЕГО ЧЕТЫРЕ НОГИ? (ОТВЕЧАЙ «ДА» ИЛИ «НЕТ»).

Да.

ГОВОРIT ЛИ ОНО «МУ...»?

Нет.

ЛОВИТ ЛИ ОНО МЫШЕЙ?

Да.

ЭТО КОШКА?

Да.

ВОТ ВИДИШЬ, Я УГАДАЛ. ХОЧЕШЬ СЫГРАТЬ ЕЩЕ?

Нет.

ДО СВИДАНИЯ. ПРИХОДИ ЕЩЕ.

Не правда ли, машина кажется вполне разумной? Между тем программа чрезвычайно проста. Она хранит дерево вариантов, которое в нашем случае могло бы выглядеть так (рис. 34):

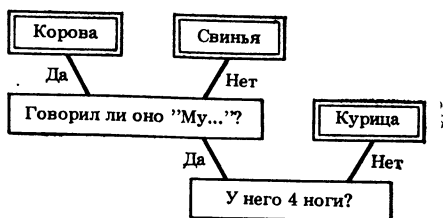


Рис. 34

В прямоугольниках записаны тексты вопросов, задаваемых играющему. Ответ игрока определяет, какой вопрос будет задан следующим и когда будет названо предполагаемое животное. В первой части протокола машина, задав два вопроса, дает ответ «свинья». Узнав, что была задумана кошка, машина дополняет дерево новым вопросом, который она узнает у игрока. После этого дерево приобретает такой вид (рис. 35):

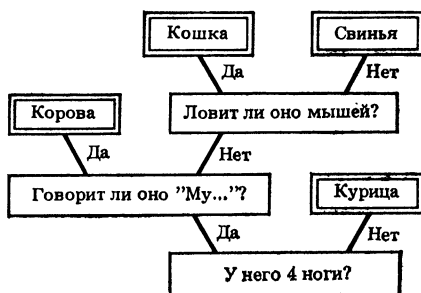


Рис. 35

Мы видим, что никакой «разумности» в программе нет — она попросту повторяет вопросы, сообщенные ей ранее, а также некоторые стандартные предложения (вроде «хочешь ли играть еще?»), заранее вписанные в программу. Ее кажущаяся интеллектуальность — это лишь отражение интеллекта игрока в абсолютно неинтеллектуальном зеркале очень простой программы.

Разумеется, «искусственные интеллигенты» (так называет Дж. Вейценбаум людей, занимающихся «разумом» машины) создали и более сложные программы. Более подробно об этом можно прочесть в книге Дж. Вейценбаума «Возможности вычислительных машин и человеческий разум» (см. главу 22).

ПРОГРАММИСТСКИЕ БАСНИ

Вначале несколько слов об их авторе. Эдстер Дейкстра — один из тех людей, с именем которых связано превращение программирования из шаманства в науку. Более подробно о программировании как шаманстве можно прочесть в статье Ч. Хоара «Программирование как инженерная профессия», напечатанной в журнале «Микропроцессорные средства и системы» (1984. № 4). Работы Э. Дейкстры уже сейчас можно назвать классическими.

Одной из форм научной деятельности Дейкстры являются письма, которые он время от времени посылает своим корреспондентам (а также нанимателям: живя в Голландии в г. Эйндховене, он работал в фирме «Burroughs», находящейся в США), призывая распространять их дальше. Сборник, содержащий некоторые из этих писем, был опубликован в 1982 г. Жанр их весьма разнообразен: наряду со статьями среди них есть отчеты о научных командировках, рассказы о встречах с коллегами, а также эссе, демонстрирующие незаурядные литературные таланты автора. Когда взгляды Дейкстры стали известны широкому кругу программистов, они вызвали сильную (и далеко не всегда положительную) реакцию. Знакомство с точкой зрения крупного ученого в этой области будет безусловно полезно и нашему читателю. Поэтому мы воспроизводим с некоторыми сокращениями два характерных эссе Дейкстры «Притча» и «Как быть, если правда колет глаза?».

Притча

В незапамятные времена была организована железнодорожная компания. Один из ее руководителей (вероятно, коммерческий директор) обнаружил, что можно сэкономить много денег, если снабжать туалетом не каждый железнодорожный вагон, а лишь половину из них. Так и решено было сделать.

Однако вскоре после начала пассажирских перевозок начались неприятности с туалетами. Причина их была крайне проста: хотя компания была только что создана, неразберихи уже хватало, и о распоряжении

коммерческого директора ничего не знали на сортировочных станциях, где все вагоны считали одинаковыми. В результате в некоторых поездах туалетов почти совсем не было.

Чтобы исправить положение, каждый вагон снабдили надписью, говорящей, есть ли в нем туалет, и сцепщикам было велено составлять поезда так, чтобы около половины вагонов имели туалеты. Хотя это и осложнило работу сцепщиков, вскоре они с гордостью сообщили, что тщательно выполняют новую инструкцию.

Тем не менее неприятности с туалетами продолжались. Новое расследование их причин показало, что хотя действительно половина вагонов в поезде снабжена туалетами, иногда выходит так, что все они оказываются в одной половине поезда. Чтобы спасти дело, были выпущены новые инструкции, предписывающие чередовать вагоны с туалетами и без них. Это добавило работы сцепщикам, однако, поворчав, они и с этим справились.

Но проблема на этом не кончилась. Поскольку туалет располагается в одном из концов вагона, расстояние между двумя соседними туалетами в поезде могло достигать трех длин вагонов и для пассажиров с детьми — особенно если коридоры были заставлены багажом — это было слишком далеко. Тогда вагоны с туалетами были снабжены стрелкой, и были изданы новые инструкции, предписывавшие, чтобы все стрелки были направлены в одну сторону. Нельзя сказать, чтобы эти инструкции были встречены на сортировочных станциях с энтузиазмом — количество поворотных кругов было недостаточным, но, напрягшись, сцепщики научились делать и это.

Теперь, когда все туалеты находились на равных расстояниях, компания была уверена в успехе, однако пассажиры продолжали беспокоиться: хотя до ближайшего туалета было не больше одного вагона, но не было ясно, с какой стороны он находится. Чтобы решить и эту проблему, внутри вагонов были нарисованы стрелки с надписью «ТУАЛЕТ», сделавшие необходимым правильно ориентировать и вагоны без туалетов.

На сортировочных станциях новая инструкция вызвала шок: сделать требуемое вовремя было просто невозможным. В критический момент кто-то, чье имя сейчас невозможно установить, заметил следующее.

Если мы сцепим вагон с туалетом с вагоном без одного так, чтобы туалет был посередине, и никогда их не будем расцеплять, то сортировочная станция будет иметь дело вместо N ориентированных объектов с $N/2$ объектами, которые можно во всех отношениях и со всех точек зрения считать симметричными. Это наблюдение решило проблему ценой двух уступок. Во-первых, поезда могли теперь состоять лишь из четного числа вагонов — недостающие вагоны могли быть оплачены за счет экономии от сокращения числа туалетов, и, во-вторых, туалеты были расположены на чуть-чуть неравных расстояниях. Но кого беспокоит лишний метр?

Хотя во времена, к которым относится наша история, человечество не знало ЭВМ, неизвестный, нашедший это решение, был первым в мире компетентным программистом.

Я рассказывал эту историю разным людям. Программистам, как правило, она нравилась, а их начальники обычно сердились все больше и больше по мере ее развития. Настоящие математики, однако, не могли понять, в чем ее соль.

Как быть, если правда колет глаза?

Иногда мы обнаруживаем неприятные истины. И когда это происходит, попадаем в затруднительное положение, поскольку утаить их — научная нечестность, сказать же правду — значит, вызвать огонь на себя. Если эти истины достаточно неприятны, то ваши слушатели психологически неспособны принять их и вы будете ославлены как абсолютно лишенный здравого смысла, опасно революционный, глупый, коварный или какой-то еще там человек. (Не говоря уже о том, что, настаивая на таких истинах, вы обеспечите себе непопулярность во многих кругах и вообще не обойдетесь без персонального риска. Вспомните Галилея...)

Информатика (computer science) выглядит тяжело больной от такого противоречия. В целом она хранит молчание и пытается избежать конфликта, переключая внимание на другое (например, в отношении Кобола вы можете либо бороться с болезнью, либо делать вид, что ее не существует: многие факультеты информатики

выбирают второй, более простой путь). Но, собратья, я спрашиваю вас: разве это честно? Не подрывает ли наше затянувшееся молчание единства информатики? Прилично ли сохранять молчание? Если нет, то как об этом говорить?

Чтобы подбросить вам идей, касающихся этой проблемы, перечислю некоторые из таких истин. (Почти все знакомые мне ученые-программисты без колебаний согласятся в основном со всеми из них. Однако мы вынуждены позволить миру действовать так, как будто мы их не знаем.)

Программирование — одна из наиболее трудных отраслей прикладной математики: слабым (poor) математикам лучше оставаться чистыми (pure) математиками.

Научно-технические расчеты — простейшее применение вычислительной техники.

Средства, которые мы применяем, оказывают глубокое (и тонкое) влияние на наши способы мышления и, следовательно, на нашу способность мыслить.

Фортран — «младенческое расстройство» с двадцатилетним стажем — безнадежно неадекватен какому бы то ни было применению ЭВМ сегодня: он слишком неуклюж, слишком опасен и слишком дорог, чтобы его применять.

ПЛ/1 — «роковая болезнь» — принадлежит скорее к области проблем, чем к области решений.

Практически невозможно научить хорошо программировать студентов, ориентированных первоначально на Бейсик: как потенциальные программисты они умственно оболванены без надежды на исцеление.

Использование Кобола калечит ум. Его преподавание, следовательно, должно рассматриваться как уголовное преступление.

АПЛ — ошибка, доведенная до совершенства. Это язык будущего для программистской техники прошлого: он открывает новую эру для любителей кодирования.

Проблемы управления в целом и управления базами данных в частности чрезвычайно сложны для людей, которые думают на смеси «ЕС-овского с нижегородским» (буквальный перевод: в терминах фирмы IBM, смешанных с неряшливым английским).

Об использовании языка: невозможно заточить карандаш тупым топором. Столь же тщетно пытаться сделать это десятком тупых топоров.

Помимо математических способностей, жизненно важным качеством программиста является исключительно хорошее владение родным языком.

Многие компании, которые поставили себя в зависимость от оборудования фирмы IBM (и, поступив так, продали душу дьяволу), потерпят крах под весом неуправляемой сложности своих систем обработки данных.

Ни научная дисциплина, ни крепкая профессия не могут быть основаны на технических ошибках министерства обороны и производителей ЭВМ.

Использование антропоморфной терминологии в отношении вычислительных систем — признак профессиональной незрелости.

Провозглашая себя работающими в области программного обеспечения (software), слабые (soft) ученые делают себя еще более смешными (но не менее опасными). Вопреки своему названию, software (буквально: мягкое оборудование) требует (жесточайше) твердой научной дисциплины для своей поддержки.

В старые добрые времена физики повторяли опыты друг друга, чтобы быть уверенными в результатах. Сегодня они придерживаются Фортрана, перенимая друг у друга программы с ошибками.

Проекты, предлагающие программирование на естественном языке, гибельны по своей сути.

Разве не достаточен этот список, чтобы дать повод для беспокойства? Но что мы собираемся делать? Вероятно, заняться повседневными делами...

Если предположение о том, что «вы предпочли бы, чтобы я не волновал вас пустяками, посылая вам это», справедливо, то можете добавить его к списку неприятных истин.

ГЛАВА 22

КОМПЬЮТЕРЫ И ОБЩЕСТВО

Мы уже говорили, что основная задача нашей книги — дать «почувствовать», какие проблемы приходится решать программисту в его профессиональной деятельности. Однако проблемы, о которых шла речь, были «внутренними» — считалось совершенно ясным,

что нужно запрограммировать, при этом не возникали вопросы, зачем это нужно программировать, почему именно так надо ставить задачу и т. д. В жизни именно эти вопросы оказываются наиболее серьезными, а часто и роковыми для судьбы той или иной программы (или тех, кто ее использует).

Десять лет назад в США вышла книга известного программиста Дж. Вейценбаума, переведенная в 1982 г. на русский язык издательством «Радио и связь» под названием «Возможности машин и человеческий разум. От суждений к вычислениям». Книга эта вызвала бурную дискуссию в США; многие ее положения были слишком заострены, некоторые — даже ошибочны. Тем не менее она сыграла положительную роль в прояснении вопроса о месте компьютера в обществе. Многие проблемы, поднятые Вейценбаумом, актуальны и для нашей страны, его книга до сих пор остается у нас одной из самых читаемых работ о программировании. Попытаемся изложить некоторые из положений этой книги, кажущиеся нам наиболее важными.

Важнейший вопрос, с которым сталкивается человек в своей деятельности, — это вопрос о стоящих перед ним целях и средствах их достижения. Часто общие цели оказываются ясными, но их конкретизация уже может вызвать сомнения. Пусть мы хотим обеспечить наилучшие условия труда для некоторой категории людей. Должны ли мы обеспечить им возможность как можно проще добраться до места работы или возможность удобной работы дома (что, кстати, во многих отраслях осуществимо с помощью компьютеров)? Если мы выбрали первый вариант, возникает следующая проблема: следует ли сделать путь наиболее приятным для пешехода и велосипедиста, расширить парки, покрыть их живописными дорожками, прекратить движение большинства видов транспорта и т. д. или, напротив, расширить проезжую часть, увеличить число подземных переходов, снести старые дома и деревья, мешающие движению? Должны ли мы, приняв определенное решение, пытаться убедить граждан следовать ему, терпеливо разъясняя им наши задачи, подавая пример, или вводить материальные премии для тех, кто ему следует, а нарушителей штрафовать либо бороться с ними еще более жестко? Должны ли мы принимать решения исходя из какой-либо научной теории или руководствоваться соображениями морали, тради-

циями, произведениями литературы и искусства, опросами общественного мнения и т. д.?

Как видите, список вопросов очень велик и большинство из них таково, что компьютер в принципе не может помочь при их решении. Предположим все же, что, тщательно проанализировав ситуацию, вы смогли поставить перед собой некоторые цели - и выбрали научные методы описания этих целей и средств их достижения. Это опять-таки не означает, что вам удастся использовать компьютер. Совсем не обязательно, что научный язык, которым вы пользуетесь, удастся перевести на язык математики, не потеряв полностью смысла того, что вы хотели сказать. Сам перевод может оказаться проблемой неразрешимой или более сложной, чем первоначальная. Сложность ее тем более велика, что за перевод можно принять видимость перевода. Предположим, что перевод осуществляет коллектив из биологов и математиков. Кто должен гарантировать правильность перевода? (Может ли переводить с японского на русский человек, знающий только японский? или только русский? или они вдвоем?)

Но пусть (это еще далеко не последнее «но пусть») вам удалось построить соответствующую (адекватную, выражаясь научно) модель. Надо ее «обсчитать». Не торопитесь с этим! Прямое программирование скорее всего приведет к программе, которая будет работать невообразимо долго уже для очень простых исходных данных. И это не потому, что вы что-то не так запрограммировали, а в силу сложности вычислений, неизбежно связанных даже с очень простыми вычислительными моделями. Таким образом, вам придется еще «огрубить» свою модель, заменив ее более простой, приводящей к разумным объемам вычислений, но все еще отражающей реальность. Здесь опять возникает вопрос: как проверить, что вы с водой не выплеснули и ребенка?

Итак, пусть вам все же удалось выбрать удовлетворительную с разных точек зрения модель. Вы вступаете непосредственно в область программирования. Вам удалось написать программу, которая (по вашему мнению) соответствует принятой модели. Вы запускаете ее на простейших примерах и... Ах да, вы забыли то-то и то-то... Но вот ошибки, мелкие и крупные, постепенно иссякают; программа, по всей видимости, правильно реагирует на тестовые примеры. Вы начинаете ее применять для больших задач и вдруг через

год обнаруживаете, что результаты как-то не согласуются друг с другом и с данными, полученными другим путем. Программа где-то в глубине содержала ошибку, которая не проявлялась на простейших примерах, где вы могли сверить результаты работы программы с заранее известным ответом, а «срабатывала» там, где ответ вы предсказать не могли. И тут вы задумываетесь, на чем же была основана ваша первоначальная уверенность в том, что ваша программа (и используемые в ней программы других людей) работает правильно. В этом месте почва окончательно ускользает из-под ваших ног и...

Впрочем, на последнем этапе — получении программы по заданной спецификации — какой-то выход есть. Вместо расплывчатой «уверенности» в правильности программы нужно располагать «доказательством» ее правильности. (Доказательство, однако, само может содержать ошибку...)

Но пусть ваша программа действительно соответствует своей спецификации. Пусть, далее, транслятор, которым вы пользуетесь, отвечает той же самой спецификации языка программирования, которую вы использовали при написании программы. Программу можно реально использовать. А это значит, что вы должны обеспечить для себя условия работы, включающие возможность доступа к ЭВМ, ее безошибочную работу, необходимые ресурсы (например, запас бумаги) и т. д.

Наконец, получив результаты, нужно понять, что они означают для вашей вычислительной модели, перевести математические выводы на язык другой науки, наконец, начать действовать в реальном мире. Слабость любого звена в той цепи, которая сейчас развернулась перед вами, сделает вашу деятельность бессмысленной и, как правило, вредной.

Вейценбаум, как он рассказывает, впервые столкнулся с проблемами и парадоксами использования компьютеров на примере собственной программы под названием ЭЛИЗА. Эта программа — игрушка, предназначенная для демонстрации того, как можно за счет нескольких очень простых приемов поддерживать беседу, абсолютно не вдаваясь в содержание реплик собеседника. При этом стиль «беседы» программы с человеком напоминает стиль беседы врача-психиатра с пациентом. Вот взятый из книги Вейценбаума пример диалога:

Компьютер. Расскажите мне о Вашей семье.

Человек. Моя мать заботится обо мне.

К. Кто еще из Вашей семьи заботится о Вас?

Ч. Отец.

К. Ваш отец.

Ч. Вы чем-то напоминаете мне отца.

К. В чем Вы видите сходство?

Ч. Вы не очень настойчивы, но, по-моему, Вам бы не хотелось, чтобы я это замечал.

К. Почему Вы думаете, что я не очень настойчив?

Ч. Вы не спорите со мной.

К. Как Вы думаете, почему я не спорю с Вами?

.....
Поразительным для автора программы — Вейценбаума — явилось то, что большое число людей, «беседовавших» с программой, наделяло ее «разумом». Люди воспринимали ее как человека, иногда даже стремились беседовать с ней без свидетелей и т. д. В то же время программа вовсе не так уж принципиально отличалась от программы для «игры в животных», о которой рассказывается в одной из глав нашей книги.

Попытавшись разобраться в причинах такого отношения к его программе, Вейценбаум обнаружил, что его случай — далеко не исключительный: об «искусственном интеллекте» говорят в ситуациях еще менее содержательных. Более того, такое мнение часто поддерживается самими авторами программ и другими программистами. Часто речь идет даже не о самих программах, а об их возможных, с точки зрения автора этой программы, усовершенствованиях, иногда нет еще программы, а есть только математическая модель (или то, что программисту кажется математической моделью), иногда есть работающие программы, но нет никаких доказательств того, что они делают именно то, на что претендуют их авторы.

Книга Вейценбаума — крик души ученого, осознавшего свою ответственность за результаты использования научных достижений (в том числе и за злоупотребления ими) и обеспокоенного прежде всего этическими проблемами, связанными с применением ЭВМ. Он пишет: «Наука и техника, созданные человеком... привели к тому, что деятельность современного человека может повлиять не только на всю планету — его естественную среду обитания, но и предопределить будущее человека как вида в целом... Это означает,

что ответственность человека, а особенно ученого и инженера, выходит за пределы сиюминутной ситуации» (с. 334).

Вейценбаум призывает, прежде чем браться за какую-либо работу в области информатики, тщательно анализировать ее последствия. Например, говоря о попытках создать устройство для распознавания устной речи, он пишет: «Почему мы должны хотеть заниматься этой задачей? Я задавал этот вопрос многим энтузиастам проекта... Если задать эти вопросы основному заказчику работ — управлению перспективного планирования НИР министерства обороны США ... то получим следующий ответ: «Военно-морской флот надеется управлять кораблями и другими видами своего оружия с помощью устных команд»... Я не вижу основания советовать своим студентам посвящать свои таланты этой цели... Я предлагал своим студентам и коллегам... еще один вопрос... Для чего бы она [машина, распознающая речь] могла быть в принципе использована? У меня нет сомнений в том, что у человека нет таких насущных проблем, разрешение которых облегчило бы наличие такой машины. Но подобные... машины сделали бы слежение за речевой связью намного легче, чем оно представляется сейчас. Может быть, единственной причиной ограниченности слежки за телефонными разговорами, практикуемой правительствами многих стран, является то обстоятельство, что подобная слежка требует участия очень большого числа людей. Каждый разговор по подслушиваемому телефону должен быть когда-то прослушан каким-то человеком. Машина же, распознающая речь, может отсеивать все «неинтересные» разговоры и предоставлять своим хозяевам расшифровки только оставленных... Я спрашиваю, почему талантливые специалисты в области вычислительной техники должны оказывать поддержку подобному проекту?» (с. 346—347).

Удивительно, что при всем различии исходных позиций Вейценбаума и Дейкстры, который провозгласил в качестве правила научного исследования, что «мы все хотим, чтобы наша работа была общественно полезна и научно значима; если мы смогли найти тему, отвечающую обоим требованиям, все хорошо, если же эти две цели приходят в противоречие, научная значимость должна преобладать», их отношение к конкретным вопросам информатики оказывается очень близким.

Вейценбаум считает крайне опасной «безрассудную антропоморфизацию вычислительной машины, столь сейчас распространенную, особенно в среде „искусственной интеллигенции“» (с. 365); Дейкстра также указывает на это обстоятельство как на одно из главных препятствий на пути развития информатики и называет использование антропоморфной терминологии признаком профессиональной незрелости. Вейценбаум подчеркивает, что многие — и, возможно, самые существенные аспекты деятельности человека — в принципе нельзя поручать вычислительной машине: недопустимы ситуации, когда «ни один человек не несет ответственности за результаты, предлагаемые вычислительной машиной» (с. 305), и считается, что компьютеры «в значительной мере отвечают (!) за поддержание в мире мира и стабильности и в то же время способны привести в действие силы, разрушительный эффект которых для человека почти непостижим». Подобно этому Дейкстра говорит о понимании как о специфически человеческой деятельности, которую в принципе нельзя передать компьютерам. И Вейценбаум, и Дейкстра считают бессмысленными утопии о программировании на естественном языке. «Раздаются настойчивые требования создать системы программирования на естественном, например на английском, языке. Программисты, придерживающиеся такой точки зрения, вероятно, никогда не имели дела с действительно трудной задачей», — пишет Вейценбаум. Дейкстра называет проекты программирования на естественном языке «гибельными по своей сути».

Если общее кредо Вейценбаума и Дейкстры (как мы его поняли) выразить одной фразой, то можно сказать так:

ПРИМЕНЕНИЕ КОМПЬЮТЕРОВ ДОПУСТИМО И ОПРАВДАНО В ТЕХ СЛУЧАЯХ, КОГДА ИМЕЕТСЯ ТОЧНО СФОРМУЛИРОВАННОЕ И ДОКАЗАННОЕ УТВЕРЖДЕНИЕ ОБ ИХ РАБОТЕ, ПОЛНОСТЬЮ ЯСНОЕ ТОМУ, КТО БЕРЕТ НА СЕБЯ ОТВЕТСТВЕННОСТЬ ЗА ИХ ИСПОЛЬЗОВАНИЕ.

В заключение главы дадим портрет, ставший уже классическим, «одержимого программиста» — «хакера», нарисованный Вейценбаумом. «Где бы ни организовывались вычислительные центры... можно наблюдать... молодых людей, всклоченных, часто с запавшими, но сияющими глазами, которые сидят за пультами уп-

равления вычислительных машин, сжав в напряжении руки... Если они не находятся в таком трансе, то часто сидят за... машинными распечатками... подобно людям, одержимым постижением каббалистического текста. Они работают чуть ли не до полного изнеможения, по 20—30 часов подряд... Если возможно, они спят около вычислительной машины на раскладушках, но всего несколько часов, а затем — снова за пульт управления... Они — машинные наркоманы, одержимые программисты. Это явление наблюдается во всем мире...» (с. 162—163).

Хочется верить, что нашим читателям такая участь не угрожает.

ГЛАВА 23

ПАСКАЛЬ ДЛЯ НАЧИНАЮЩИХ

Язык Паскаль, изобретенный в начале 70-х годов Н. Виртом и названный в честь французского математика и философа Блеза Паскаля, является одним из наиболее распространенных языков программирования. От других распространенных языков он выгодно отличается возможностью более ясно и логично записывать программы, что делает его подходящим языком как для начинающих, так и для опытных программистов.

Изложение языка начнем с минимума сведений, необходимых для написания простых программ. При этом опускаются многие конструкции и средства языка. Тем не менее оставшегося для начала вполне достаточно. Затем постепенно вводятся некоторые более сложные конструкции.

Общая структура программы

Программа на языке Паскаль состоит из двух частей: описания действий, которые должны быть выполнены, и описания данных, над которыми они выполняются. В тексте программы описание данных предшествует описанию действий. В этом выражается общее правило языка — каждый встречающийся в программе объект должен быть предварительно описан.

Описание данных состоит из описаний переменных, действия над которыми называют операторами. Программа имеет вид

заголовок программы
раздел описания переменных
раздел операторов

Программа может быть как угодно разбита на строки — смысл ее от этого не меняется (нельзя только переносить слова с одной строки на другую). Поэтому стоит стараться записывать возможно более наглядно.

Заголовок. Идентификаторы

Заголовок программы имеет вид:

`program имя_программы (input, output);`

Слова `program`, `input`, `output` — это так называемые ключевые слова, они должны записываться буквально. Имя_программы — это любая последовательность букв и цифр, начинающаяся с буквы. Такие последовательности называют идентификаторами. Идентификатор не должен совпадать ни с одним из ключевых слов (их список приведен в конце главы).

Переменные. Стандартные типы

Каждая переменная имеет имя и тип. Имя переменной — это произвольный идентификатор. В дальнейшем мы будем говорить «переменная *X*» вместо «переменная с именем *X*».

Тип переменной определяет множество ее возможных значений. В языке Паскаль существуют четыре типа переменных: `integer` (целый), `real` (вещественный), `boolean` (логический) и `char` (символьный).

Значениями переменных целого типа являются целые числа. Операции над целыми числами таковы: «+» (сложение), «—» (вычитание), «*» — (умножение), «div» (деление нацело положительных чисел), «mod» (остаток от деления для положительных чисел). Примеры записи целых чисел:

`5 +1500 —2`

Значениями переменных вещественного типа являются вещественные числа. Определены операции сложения, вычитания, умножения (обозначения те же) и деления «/». Запись вещественных чисел похожа на общепринятую, только вместо десятичной запятой

используется точка и знак «Е» в записи числа означает «умножить на десять в степени...». Примеры:

1.0—0.0000072—4.1E41E7

два последних числа равны — 41 000 и 10 000 000.

Значениями переменных логического типа являются истина (true) и ложь (false). Операции: not (не), and (и), or (или). Допустимо использовать константы true и false.

Значения переменных символьного типа — имеющиеся в машине символы (их набор может быть различным). Примеры записи констант этого типа:

'1' 'A' ' ' (пробел)

Операции отношения

Существуют операции, определенные на парах целых чисел и дающие в результате логическое значение (истину или ложь). Эти операции таковы:

= равно <> не равно

< меньше > больше

<= меньше или равно >= больше или равно

Операции с теми же обозначениями и аналогичным смыслом определены и на множестве вещественных чисел. Над символьными значениями определены операции = и <>.

Раздел описания переменных

Этот раздел имеет вид

var описание; описание; описание;...

где слово «var» — ключевое, а каждое описание представляет собой один или несколько идентификаторов, разделенных запятыми, за которыми следует двоеточие, а затем название типа. Описание сообщает исполнителю программы, какие в ней есть переменные и какого они типа. Все имеющиеся в программе переменные должны быть описаны (но только однажды!).

Пример раздела описаний переменных:

```
var a, b, c : real;  
    n, q1, следующий : integer;  
    упрсимв : char;  
    поракончить : boolean;
```

Выражения. Приоритет операций

Из констант и значений переменных можно составлять выражения, используя перечисленные операции. Примеры выражений:

$$(a + b)/c$$

(следующий $*n + q1$) $\text{div } (n + q1)$

(поракончать $\text{or not } (a = b))$ and $(n < > q1)$

Порядок выполнения операций в выражениях определяется скобками. Кроме того, действуют обычные правила, вроде того, что умножение и деление выполняются раньше сложения и т. п.

Операторы

Основная часть программы на Паскале — раздел операторов. Он начинается ключевым словом `begin` и заканчивается ключевым словом `end`, за которым следует точка. В этом разделе описываются действия, которые исполнитель должен выполнить.

О п е р а т о р ы п р и с в а и в а н и я. Элементарное действие над переменной — изменение ее значения. Для этого применяется оператор присваивания, имеющий вид

имя_переменной := выражение

В нем переменная и выражение должны быть одного типа. Пример (X — имя целой переменной):

$$X := X + 1$$

Обратите внимание, что в левой части идентификатор X обозначает переменную, а в правой части — число, являющееся ее текущим значением. Выполнение этого оператора приводит к увеличению значения переменной X на единицу. Другие примеры:

$$a := b + c$$

поракончать := поракончать $\text{or } (a > b)$

О п е р а т о р ы в в о д а и в ы в о д а. Во время исполнения программа обменивается информацией с «внешним миром». Например, она может выдавать информацию на экран дисплея и получать ее с клавиатуры. Для этого используются операторы ввода и вывода.

Оператор вывода имеет вид

write (выражение)

В результате выполнения этого оператора значение соответствующего выражения будет напечатано. Выражение может быть любого из четырех указанных выше типов. Примеры:

```
write (2 + 2) write (X = Y)
```

```
write ('4')   write (5E8)
```

Кроме того, можно указывать в операторе вывода несколько выражений, разделяя их запятыми, а также последовательности символов в одинарных кавычках — апострофах, которые выводятся буквально так, как записано. Пример: пусть в некоторый момент значение переменной *A* равно 5. Тогда при выполнении оператора

```
write ('Значение A равно ', A)
```

будет напечатано

Значение *A* равно 5

Разновидностью оператора вывода является оператор вывода с переходом на новую строку `writeln`. Он отличается от обычного тем, что после его выполнения происходит переход на новую строку.

Оператор ввода имеет вид:

```
read (имя_переменной)
```

В результате его выполнения переменной присваивается считанное с устройства ввода (клавиатуры) значение. Вводимое значение записывается так, как показано выше (за исключением того, что символ не должен заключаться в апострофы). При употреблении оператора ввода надо помнить, что если в программе стоит этот оператор, то, дойдя до него, программа будет ждать, пока вы не введете необходимые данные. Так как на внешнем виде компьютера это никак не отражается, то понять, что программа чего-то ждет (и уж тем паче, чего именно), бывает трудно. Поэтому целесообразно перед вводом данных печатать сообщение, используя, например, оператор `write ('X = ?')` перед оператором `read (X)`.

Программные конструкции

П о с л е д о в а т е л ь н о е в ы п о л н е н и е .
Выполняемые друг за другом операторы разделяются точкой с запятой:

оператор; оператор; ... оператор

Ветвление. Команда ветвления

```
если условие то  
  . A1  
  . A2  
  . ....  
  . иначе  
  . B1  
  . B2  
  . ....  
конец_если
```

записывается в Паскале как

```
if условие then begin  
  A1;  
  A2;  
  ....  
end else begin  
  B1;  
  B2;  
  ....  
end
```

(после последнего оператора каждой из серий точек с запятой можно не ставить).

Более общая конструкция типа

```
выбор  
  . при условиеА делай A1 A2 A3 ...  
  . при условиеБ делай B1 B2 B3 ...  
  . при условиеВ делай V1 V2 V3 ...  
конец_выбора
```

записывается так:

```
if условиеА then begin  
  A1; A2; A3; ...  
end else if условиеБ then begin  
  B1; B2; B3; ...  
end else if условиеВ then begin  
  V1; V2; V3; ...  
end else begin  
  writeln ('Ошибка в выборе; все условия ложны'  
end
```

Повторение. Команда

```
пока условие повторять  
  . действие1  
  . действие2  
  . ....  
конец_пока
```

записывается в Паскале как

```
while условие do begin
    действие1;
    действие2;
    ....
end
```

Прежде чем привести пример простой программы, скажем, что среди текста программы могут встречаться комментарии — любые последовательности символов, заключенные в фигурные скобки {...} или в скобки (*...*), если среди символов нет фигурных скобок. Комментарии удобно использовать для включения в программу утверждений.

Пример программы

```
program НОД (input, output);
var M, N, A, B, X: integer;
begin
    writeln ('Введите два числа');
    read (M); read (N);
    A := abs (M); B := abs (N);
    {НОД (M, N) = НОД (A, B), A, B >= 0}.
    while (A < > 0) and (B < > 0) do begin
        {НОД (M, N) = НОД (A, B), A, B > 0}
        if A > B then begin
            A := A - B;
        end else begin
            B := B - A;
        end;
    end;
    {НОД (M, N) = НОД (A, B), A, B >= 0}
end;
{НОД (M, N) = НОД (A, B), A, B >= 0,
A = 0 or B = 0}
if A = 0 then begin
    X := B;
end else begin
    X := A;
end;
{X = НОД (M, N)}
writeln;
write ('НОД (' , M , ' , ' , N , ') = ');
writeln (X);
end.
```

З а м е ч а н и е. Оператор `writeln` без аргументов вызывает переход на следующую строку. Функция `abs` — стандартная функция языка Паскаль и соответствует модулю (абсолютной величине).

Константы

В Паскале есть возможность дать константе любого из четырех стандартных типов (`integer`, `real`, `boolean`, `char`) имя, при этом в последующем тексте программы всюду вместо этой константы можно употреблять данное ей имя. Все определения констант перечисляются в специальном разделе — разделе описания констант, имеющем вид

```
const имя1 = значение1;  
      имя2 = значение2;  
      .....  
      имяN = значениеN;
```

(здесь `имя1`, `имя2`, ... — произвольные идентификаторы, а `значение1`, `значение2`, ... — числа, записанные по приведенным выше правилам, символы в апострофах или константы `true`, `false`).

Примеры описания констант:

- (1) `const g = 981E-2; {метров в секунду за секунду}`
 `атмосфера = 0.76; {метров ртутного столба}`
- (2) `const максчислоитераций = 1000;`
 `погрешность = 1E-7;`
- (3) `const pi = 3.1415926;`
 `печатаемаЦПУ = false;`
 `управляющийсимвол = '$';`

Есть по крайней мере два довода в пользу описаний констант. Во-первых, использование буквенных обозначений для констант — традиция в физике и математике. Поддерживая эту традицию, мы делаем программы более понятными. Осмысленные названия констант — один из способов прокомментировать программу: запись «длинастроки» гораздо информативнее, чем «60». Во-вторых, использование описаний констант облегчает изменение программы. Например, при желании изменить длину печатаемых строк с 60 на 40 достаточно заменить запись «длинастроки = 60» на «длинастроки = 40» в разделе описания констант. Без такого описания пришлось бы отыскивать каждое число 60 в программе, выяснять, имеет ли оно отношение к длине строки и если да, то заменять его на 40. Потра-

тив массу труда на это, особенно досадно обнаружить, что и теперь программа не работает — не работает потому, что вы пропустили число 59, которое надо было заменить на 39.

Определение новых типов данных

Помимо четырех стандартных типов, программист имеет возможность определять новые типы данных и давать им названия. После этого новый тип можно использовать наряду со стандартными. Раздел определения типов имеет вид

```
type имя1 = описание1;  
      имя2 = описание2;  
      .....  
      имяN = описаниеN;
```

где имя1, имя2, ... — идентификаторы, а описание1, описание2, ... — описания типов. Допустимы следующие описания типов.

О т р е з о к ц е л ы х ч и с л. Описание имеет вид `min .. max`, где `min` и `max` — две целочисленные константы. Значениями этого типа являются целые числа от `min` до `max` включительно. Переменные этого типа могут употребляться наравне с переменными целого типа, однако при присваивании `X := A`, где `X` — переменная типа `min .. max`, `A` — выражение, значение которого меньше `min` или больше `max`, может произойти ошибка.

П р и м е р.

```
type возраст = 0..200;  
var  возрастМиши, возрастМаши : возраст;  
      .....  
if    возрастМиши = возрастМаши +  
      + 1 then begin  
      .....  
end
```

М а с с и в ы. Если типиндекса — тип, являющийся отрезком целых чисел, а типкомпоненты — произвольный тип, допустимо описание

`аггау[типиндекса] of типкомпоненты;`

Оно определяет тип, значениями которого являются отображения, сопоставляющие с каждым значением типа «типиндекса» некоторое значение типа «типкомпоненты». Например, если описание выглядит так:

```
type час = 0..23;  
      таблица = array [час] of integer;  
var  температура: таблица;
```


то значениями переменной температура являются таблицы из 24 целых чисел, пронумерованных от 0 до 23. В этом случае записи температура [0], . . . , температура [23] могут фигурировать в программе наряду с именами переменных целого типа. Вообще описания

```
| type индекс = min..max;  
| массив = array [индекс] of типкомпоненты;  
| var t : массив;
```

позволяют использовать записи вида $t[A]$, где A — переменная типа индекс, наряду с переменными типа типкомпоненты. (Допустимо использование и записи $t[E]$, где E — выражение целого типа, но в этом случае при выполнении программы может произойти авария, если значение выражения E не попадает в границы типа индекс.) Использование записи $t[E]$ позволяет узнать (или изменить, если эта запись встречается в левой части оператора присваивания) значение переменной t (точнее, его компоненту, соответствующую значению E).

П р и м е р. Оператор

температура [9] = 13

устанавливает одну из компонент таблицы «температура» равной 13. Оператор

температура [7] := температура [7] + 1

увеличивает на 1 другую компоненту той же таблицы.

З а м е ч а н и е: допустимо совмещать определение типа индекса с определением типа массива и писать

```
| type массив = array [min..max] of  
| типкомпоненты;
```

Процедуры

Возможность определить новую команду реализуется в Паскале с помощью процедур. В простейшем случае (процедуры без параметров) определение процедуры связывает идентификатор (имя процедуры) с некоторым действием. Это действие может изменять значения переменных программы и включать в себя ввод и вывод информации. Описания процедур размещаются между описаниями переменных и словом `begin`, начинающим собственно программу. Каждое из них имеет вид:

```
| procedure имяпроцедуры;  
| begin  
| .....  
| операторы
```

| end;

.....

Операторы, входящие в процедуру, могут использовать определенные в программе константы, типы, переменные, а также процедуры, описанные до этой. Кроме того, в процедуре допускается между заголовком и списком операторов описывать константы, типы и переменные. Эти описания действительны лишь внутри процедуры (и в этом случае отменяют внешние, если относятся к тому же идентификатору). Описанные в процедуре переменные при каждом выполнении процедуры получают произвольные значения, а не сохраняют их от одного раза до другого.

П р и м е р.

```
program пример;
  var A, B : real;
  procedure обменAB;
    var x: real;
    begin
      x := A; A := B; B := x;
    end;
  begin
    A := 3; B := 5;
    {здесь A = 3, B = 5}
    обменAB;
    {здесь A = 5, B = 3}
    .....
  end.
```

Параметры

Описанные выше процедуры при каждом выполнении производят одно и то же действие. Часто необходимо описать не какое-то одно действие, а семейство действий, зависящих от параметра. Это можно сделать с помощью описания процедуры, начинающегося так:

```
procedure имяпроцедуры (параметр1 : тип1;
                        параметр2: тип2;
                        .....
                        параметрN: типN);
```

В операторах процедуры имена параметров можно использовать как выражения соответствующих типов.

Пример.

```
procedure дваждыпечатать (x: integer);  
begin  
    writeln (x);  
    writeln (x);  
end;
```

После этого описания оператор
 дваждыпечатать ($x + y + 1$)
в программе приведет к двукратной печати значения
выражения $x + y + 1$.

Функции

Наряду с процедурами в Паскале можно определять новые функции, которые можно использовать наряду со стандартными (sin, cos, ...). Описания функций помещаются там же, где и описания процедур, и выглядят так:

```
function имяфункции (параметр1 : тип1;  
    параметр2 : тип2;  
    .....  
    параметрN: типN): типзначений;
```

Это описание определяет функцию из $(тип1) \times (тип2) \times \dots \times (тип N)$ в (типзначений). Типзначений должен быть одним из типов real, integer, boolean, char. В описании функции не допускаются операторы, изменяющие значения переменных программы (хотя можно изменять значения локальных переменных этой функции); последним оператором должен быть оператор

 имяфункции := выражение соответствующего типа,
который и определяет значение функции.

Пример.

```
function большеиздвух(x : integer;  
    y : integer) : integer;  
var max;  
begin  
    if (x >= y) then begin  
        max := x  
    end, else begin  
        max := y  
    end;  
    большеиздвух := max  
end; ;
```

Процедуры с параметрами-переменными

Иногда необходимо определить команду, изменяющую заданным образом значение переменной данного типа. Например, это может быть команда сортировки массива, и мы хотим, чтобы с ее помощью можно было сортировать разные массивы. В Паскале разрешается, описывая процедуры, писать перед некоторыми из параметров слово `var`. В таком случае при использовании этой процедуры на соответствующем месте должна стоять переменная соответствующего типа, которая и передается процедуре для обработки. Следующий пример (сортировка массива простейшим способом) иллюстрирует применение функций и процедур с параметрами-переменными.

```
program sort (input, output);
  const длина = 30;
  type массив = array [1..длина] of integer;
  var a, b, c : массив;
  function номерминимального (x : массив; l : integer; r : integer) : integer;
    {дает номер минимального элемента среди x [l]..
     ..x [r]}
    var граница : integer;
        номер : integer;
  begin
    граница := l;
    номер := l;
    {x [номер] — минимальное среди x [l]..x [граница]}
    while граница < r do begin
      граница := граница + 1;
      {x [номер] — минимальное среди x [l]..x [граница - 1]}
      if x [граница] < x [номер] then begin
        номер := граница
      end else begin
        {ничего не делать}
      end;
    end;
    номерминимального := номер;
  end;
  procedure обмен (var x : массив; k : integer; l : integer);
```

```

    {меняет местами  $x[k]$  и  $x[l]$ }
    var  $z$  : integer;
    begin
         $z := x[k]; \quad x[k] := x[l]; \quad x[l] := z;$ 
    end;
    procedure сортировка (var  $m$ : массив);
        {располагает массив в порядке возрастания}
        var числорасставленных:integer; номер : inger;
    begin
        числорасставленных := 0;
        {числорасставленных наименьших элементов
        массива  $m$  установлены на свои места}
        while числорасставленных < > длина do begin
            номер := номернаименьшего ( $m$ ,
            числорасставленных + 1, длина);
            обмен ( $m$ , номер, числорасставленных + 1);
            числорасставленных := числорасставленных
            + 1;
        end;
    end;
end;
begin
    ... ввод массивов  $a, b, c...$ 
    сортировка ( $a$ ); сортировка ( $b$ ); сортировка ( $c$ );
    ... вывод массивов  $a, b, c...$ 
end

```

Ниже приводится список всех идентификаторов языка Паскаль, имеющих специальный смысл, и их перевод или краткое пояснение. Список этот нужно иметь в виду, чтобы случайно не употребить какой-либо из входящих в него идентификаторов, не подозревая об его истинном смысле.

Идентификатор Перевод, краткое пояснение

abs	abs (x) — абсолютная величина x
and	и
arctan	arctan (x) — арктангенс x
array	массив
begin	начало
boolean	булевский (= логический) тип
case	случай
char	символьный тип
chr	chr (x) — символ с кодом x
const	константа
cos	cos (x) — косинус x

div	$a \text{ div } b$ — частное при делении a на b с остатком
do	делать
downto	вниз до
else	иначе
end	конец
eof	конец файла
eoln	конец строки
exp	$\text{exp}(x)$ — экспонента x
false	ложь
file	файл
for	для
function	функция
get	получить
goto	идти на
if	если
in	в
input	стандартный ввод
integer	целый тип
label	метка
ln	$\ln(x)$ — натуральный логарифм x
maxint	наибольшее представимое целое число
mod	$a \text{ mod } b$ — остаток при делении a на b
new	новый
nil	нил (указатель, не указывающий никуда)
not	не
odd	$\text{odd}(x)$ — число x нечетно
of	из
or	или
ord	$\text{ord}(x)$ — код символа x
output	стандартный вывод
pack	упаковать
packed	упакованный
page	переход на новую страницу
pred	$\text{pred}(x)$ — предшественник x
procedure	процедура
program	программа
put	поместить
read	читать
readln	пропустить остаток строки
real	вещественный тип
record	запись
repeat	повторять
reset	начать чтение с начала
rewrite	начать запись с начала

round	round (x) — целое число, получающееся при округлении x
set	множество
sin	sin (x) — синус x
sqr	sqr (x) — квадрат x
sqrt	sqrt (x) — квадратный корень из x
succ	succ (x) — следующий за x
text	текст
then	то
to	до
true	истина
trunc	trunc (x) — целая часть x
type	тип
unpack	распаковать
until	до
var	переменная
while	пока
with	с
write	писать
writeln	писать с переходом на новую строку

ГЛАВА 24

РЕКОМЕНДАЦИИ ПО ДАЛЬНЕЙШЕМУ ЧТЕНИЮ

Книг по программированию довольно много, и с каждым годом их становится все больше. При таком обилии легко не заметить хорошую книгу или, что еще хуже — ибо эта ошибка непоправима, — прочесть плохую. Некоторые советы по выбору литературы для дальнейшего чтения (отражающие личный вкус авторов) приведены в этой главе.

Как правило, следует опасаться книг, в названиях которых фигурируют упоминания конкретных языков или систем — вроде «ПЛ/1 для программистов», «IBM /360» или «Язык Фортран». Обычно авторы таких книг смешивают программирование с изучением (зачастую нелепых) особенностей какого-либо языка или операционной системы.

Следует также на первых порах избегать книг, в которых объясняются подробности внутреннего устройства машины, — на начальном этапе изучения программирования эти подробности могут только помешать.

Вот примерный список слов, наличие которых в книге может иногда служить признаком того, что ее лучше не читать: Фортран, Бейсик, PL/1, ЕС ЭВМ, Кобол, Ассемблер, ячейка памяти, язык управления заданиями, методы доступа к файлам, бит, байт, двоичная система счисления, объектный код, загрузчик, редактор связей, микрокалькуляторы, перфокарты...

Мы приведем список книг по программированию, которые кажутся нам примечательными в том или ином отношении, сопроводив его краткой рецензией на каждую из них.

Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 536 с.

Книга представляет собой одно из лучших изложений наиболее важных быстрых алгоритмов. За немногими исключениями, алгоритмы и доказательства их правильности изложены довольно ясно. Последние главы книги посвящены теоретическим результатам о невозможности построения эффективных алгоритмов для некоторых задач. Названия глав: 1. Модели вычислений. 2. Разработка эффективных алгоритмов. 3. Сортировка и порядковые статистики. 4. Структуры данных для задач, касающихся работы с множествами. 5. Алгоритмы на графах. 6. Умножение матриц и связанные с ним операции. 7. Быстрое преобразование Фурье и его приложения. 8. Арифметические операции над целыми числами и полиномами. 9. Алгоритмы идентификации. 10. NP-полные задачи. 11. Некоторые доказуемо трудно разрешимые задачи. 12. Нижние оценки числа арифметических операций.

Брукс Ф. Как проектируются и создаются программные комплексы. М.: Наука, 1979. 151 с.

Автор книги, имевший отношение к разработке первых версий злополучных операционных систем компьютеров серии IBM/360 (ЕС ЭВМ), рассказывает о непреодолимых трудностях, возникающих при попытках управлять «производством программного продукта» как обычным материальным производством. Становится ясным, как коммерческие соображения, принятые в расчет на ранней стадии программистского проекта, губят его. Автор подчеркивает, что правильная организация работ и правильное руководство проектом играют решающую роль. Однако некоторые приводимые им факты (например, он чуть ли не с гордостью

сообщает, что толщина рабочей документации по операционной системе OS/360 достигла полутора метров и что проблема была решена (!) переходом на микрофиши), как и результаты подобных проектов, заставляют усомниться в эффективности такого подхода. Трудно себе представить, что математическую проблему можно решить, пригласив для этого тысячу математиков (в основном средней квалификации), придав им менеджера блестящих способностей и снабдив твердым календарным планом работ, — а автор предлагает нечто подобное в программировании.

Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975. 245 с.

Книга состоит из трех статей:

1. Э. Дейкстра. Заметки по структурному программированию. Одна из первых статей Дейкстры, ознаменовавших начало новой эпохи в программировании. Содержит много глубоких философских замечаний. Приводится подробное описание процесса построения программ для нескольких примеров (составление таблицы простых чисел, печатание графиков, поиск последовательности без повторяющихся фрагментов, обработка текста, расстановка ферзей). Сейчас эти примеры, по-видимому, можно было бы изложить лучше, но и текст Дейкстры интересен не только с исторической точки зрения.

2. К. Хоор. О структурной организации данных. Обсуждается система обозначений, понятие типа данных и некоторые полезные способы образования новых типов данных (перечисляемый тип, прямое произведение, размеченное объединение, массивы, множество подмножеств, рекурсивные структуры, разреженные структуры). Обсуждается возможность реализации определяемых типов в терминах исходных типов. Применение типов иллюстрируется на примере разработки программы, составляющей расписание экзаменов. Это одна из первых (и одна из наиболее толковых) статей о типах данных. Многие ее предложения в урезанном виде вошли в язык Паскаль.

3. У. Дал, К. Хоор. Иерархические структуры программ. Как пишут авторы в своем предисловии к статье, в ней рассматриваются некоторые методы структурной организации программ и подчеркивается их связь с конструкциями в области моделирования. В качестве

языка программирования используется язык СИМУЛА-67. Статья представляет собой попытку объяснить механизмы языка СИМУЛА и их применение. Объяснения мало понятны, большое количество терминов используется без сколько-нибудь вразумительного определения.

Дейкстра Э. Дисциплина программирования. М.: Мир, 1978. 275 с.

Одна из лучших книг по программированию. Составит из большого числа отдельных разделов. Большинство разделов содержит постановку и решение какой-либо программистской задачи. Есть также несколько (по большей части весьма интересных) разделов, посвященных общим вопросам. Книгу отличают содержательность и оригинальность, не часто встречающиеся среди книг по программированию. Несколько менее удачных разделов не снижают общего впечатления.

Фокс Дж. Программное обеспечение и его разработка. М.: Мир. 1985. 368 с.

Автор этой поучительной книги «имел счастье руководить большим программистским предприятием — отделением фирмы IBM». Под его руководством выполнялись проекты в несколько тысяч человеколет (например, разработка программного обеспечения для полетов «Аполлон», «Скайлаб», «Шаттл» заняла 7 лет работы 700 человек). «Моя работа, — пишет он, — была похожа на тушение пожара в нефтяной скважине или на что-то в этом роде. Не успею я взять под контроль один проект, как другой внезапно взрывается и горит ярким пламенем» (с. 9).

На основе своего опыта автор пытается объяснить, «каково общее положение дел, какие наиболее важные уроки извлечены за последнее время в мире программирования» (с. 9). В целом как и положено ему по должности, Дж. Фокс расценивает ситуацию с программным обеспечением как удовлетворительную: «За последнее десятилетие в деле разработки программного обеспечения произошел разительный прогресс... Вступили в действие огромные системы — причем работают они хорошо и надежно» (с. 357). Он считает, что эти успехи — как и дальнейший прогресс — связаны не с научными, а с технологическими достижениями, прежде всего с совершенствованием организации производства программного обеспечения.

Говоря о претензиях к нему со стороны руководства фирмы IBM (с. 9), федерального агентства США (с. 96) и др., утверждавших, что разрабатываемые программы не готовы в срок, содержат ошибки и т. п., Фокс пишет: «если кто-то говорит, что он создал программу, в которой нет ни одной ошибки, значит, он говорит об очень маленькой программе, либо вообще не знает, о чем говорит» (с. 97); «если мы еще можем быть уверены в правильности программы из 200 команд, то 100%-ная уверенность в правильности программы из 1000 или 10 000 команд нам недоступна» (с. 112). В доказательство он приводит следующее рассуждение. Проанализировав программу диспетчеризации воздушного транспорта, он насчитал там 39 203 условных переходов. «Сколько же вариантов возможно при выполнении программы?... Это просто астрономическое число... Мы не можем создать... тестирующую систему, которая могла бы проверить нам все варианты...» (с. 97).

Продолжая мысль автора, заметим, что по аналогичным причинам невозможно построить сложную математическую теорию без ошибок: чтобы убедиться, достаточно подсчитать количество слов «если» в любом учебнике математики! А уж о количестве различных непрерывных функций, необходимом для полного тестирования, например теоремы о промежуточных значениях, не приходится и говорить.

Чтобы лучше оценить книгу Фокса, полезно иметь в виду и высказывания других авторов по затронутым в ней вопросам. Вот что пишет Ч. Хоар (Программирование как инженерная профессия // Микропроцессорные средства и системы. 1984. № 4): «Мосты, которые строят инженеры, как правило, не падают в реку. Пока все программисты не приобретут такой же уровень точности и надежности, наше право на статус профессионалов будет вызывать сомнения... Каждый раз, когда поставщик программного обеспечения официально снимает с себя ответственность за ущерб, наносимый непосредственными ошибками в его товаре, он порочит нашу профессию».

Э. Дейкстра, обсуждая вопрос о том, что может дать применение техники формального доказательства свойств программ на практике, пишет: «Ответ менеджера на этот вопрос прост: ничего. Он скажет, что сложные проблемы требуют для своего решения больших программ, что большие программы могут быть

написаны только большими коллективами, которые по необходимости состоят из людей низкой квалификации — достаточно низкой для того, чтобы сделать применение формальной техники полностью невозможным.

... Я не согласен с этим ответом, поскольку он опирается на два неявно сделанных предположения. Первое из них состоит в том, что решение трудных проблем требует больших программ, второе — что с помощью «легионов» второсортных специалистов можно решить трудную проблему. Оба предположения вызывают сомнения.

О втором из них не стоит долго говорить. Попытки использования легионов малокомпетентных программистов неоднократно предпринимались — и всегда с катастрофическими результатами. Наиболее известным примером такого рода является, конечно, OS/360; не думайте, однако, что успех полетов на Луну показывает, что в других случаях этот подход сработал. Имеется множество свидетельств того, что соответствующее программное обеспечение было полно ошибок, и лишь в результате общей избыточности они не имели тяжелых последствий. Короче говоря, экспериментально установлено, что большие коллективы второсортных программистов бесполезны, а совершенствование искусства их предводителей — напрасный труд».

Грис Д. Наука программирования. М.: Мир, 1984. 416 с.

Из предисловия редактора перевода А. П. Ершова: «Если считать переломную книгу Э. Дейкстры «Дисциплина программирования» интеллектуальным открытием, то книга Д. Гриса — это апостольское деяние, направленное на то, чтобы превратить учение одиночки в мировую религию».

Из вступительной статьи Э. Дейкстры: «В течение последнего десятилетия значение слова «программа» претерпело глубокое изменение: программы, которые мы писали десять лет назад, и программы, которые мы пишем сегодня, выполняются на машине, и это единственное, что их объединяет. За исключением этого поверхностного сходства, они настолько разительно отличаются друг от друга, что употребление для них общего термина может привести к путанице. Разница между «старой» программой и «новой» программой столь же глубока, сколь глубока разница между гипотезой и доказанной теоремой или между математическим на-

блюдением и утверждением, строго выведенным из свода постулатов...

Новые формализмы всегда страшат, и требуются значительные и тщательные педагогические усилия, чтобы убедить новичка в том, что формализм не только полезен, но и необходим. Сделать материал доступным может лишь ученый, который объединяет научную причастность к предмету с даром прирожденного педагога. Нам повезло — профессор Дэвид Грис принял этот вызов».

Названия частей книги А. Гриса: 0. Зачем нужно использовать логику и доказывать правильность программ. I. Высказывания и предикаты. II. Семантика простого языка программирования. III. Построение программ (эта часть содержит такие главы: 13. Введение. 14. Программирование как целенаправленная деятельность. 15. Построение циклов, исходя из инвариантов и ограничений. 16. Построение инвариантов. 17. Замечания об ограничивающих функциях. 18. Использование циклов вместо рекурсии. 19. Соображения эффективности. 20. Два больших примера построения программ. 21. Обращение программ. 22. Замечания о документации. 23. Исторические замечания).

Мы очень советуем прочесть эту книгу, причем рекомендуем делать это в таком порядке: начав с части 0, перейти сразу к части III, возвращаясь к частям I и II по мере необходимости. Дело в том, что формальный язык и обозначения, принятые в книге, разъясняются в частях I и II длинно и довольно утомительно, и при отсутствии интересных примеров (которые появятся в части III) это может надоесть. Предлагаемый порядок чтения, быть может, позволит избежать этого. Достоинством книги является большое число интересных задач; все они снабжены указаниями и ответами.

Йодан Э. Структурное проектирование и конструирование программ. М.: Мир, 1979. 415 с.

После опубликования первых работ по «структурному программированию» появилось большое число лиц, начавших торговлю модным товаром вразнос, «применительно к реальности». При этом слово «структурное» превратилось в рекламный ярлык, который навешивался на что угодно. В рассматриваемой книге, например, фигурируют словосочетания «структурное программирование», «структурное проектирование», «структурное тестирование», «структурная отладка».

Как утверждает автор, излагаемые в книге методы «помогают обратить неструктурное мышление в структурное» (!) Подобные заявления настолько дискредитировали сам термин, что один из его изобретателей — Э. Дейкстра — перестал употреблять термин «структурное программирование» с тех пор, как он «был украден фирмой IBM».

Книга Йодана представляет собой одну из наиболее известных (и заслуженно) книг, предназначенных для популяризации «идей структурного программирования» в массах работающих программистов. Она содержит большое количество благих пожеланий и полезных советов типа «ничто не заслуживает большего порицания, чем программа без комментариев» или «программа должна быть проста в тестировании и отладке». Большинство из них действительно могут оказаться полезными при практической работе. Некоторые из них рассчитаны исключительно на «опытных» программистов: «Новичок, еще не успевший приобрести привычку запутывать свои программы многочисленными передачами управления по оператору goto, вероятно, совсем не нуждается в методах, излагаемых в разделе 4.3.3, знакомство с ними могло бы оказаться для него даже вредным» (с. 10). Мы надеемся, что вы, наш читатель, попадаете в эту категорию «новичков», так что берегитесь!

Названия глав: 1. Отличительные особенности «хорошей» программы для ЭВМ. 2. Нисходящее проектирование программ. 3. Модульное программирование. 4. Структурное программирование. 5. Стил в программировании: простота и ясность. 6. Программирование с защитой от ошибок. 7. Принципы тестирования программ. 8. Принципы и способы отладки. Приложение. Упражнения и задачи для классных занятий.

Керниган Б., Ритчи Д. Язык программирования Си; Фьюэр А. Задачи по языку Си. М.: Финансы и статистика, 1985. 280 с.

Сочинение Кернигана и Ритчи является каноническим описанием языка Си (латинская буква «С»). Этот язык был разработан в связи с операционной системой UNIX и в настоящее время вместе с ней приобретает растущую популярность. На общем фоне язык Си выглядит неплохо (а в сравнении с каким-нибудь Фортраном и тем более Коболом — так просто отлично), но тенденция его повсеместного внедрения, как и по-

всеместного внедрения системы UNIX, настораживает: став фактическим стандартом, они могут принести в следующие 20 лет не меньше вреда, чем Фортран и OS/360 в предыдущие.

Книга представляет собой описание языка с помощью примеров, которые должны демонстрировать одновременно возможности языка Си и «хороший стиль программирования». Рекомендовать их как образцы стиля мы бы не осмелились, но научиться использовать язык по ним можно.

Вторая часть книги («Задачи по языку Си») весьма своеобразна. В языке Си, как это часто бывает, встречаются малопривлекательные области, куда каждый сколь-нибудь разумный программист остережется попадать. Автор не обходит ни одного из них: его девиз: «то, что делает программу плохой, может сделать головоломку интересной. Это, скажем, двусмысленность выражений, требующая для их интерпретации обращения к описанию языка; сложность структуры: структуру данных и структуру программы нелегко удержать в голое; неочевидность конструкций, возникающая при использовании их нестандартным способом». Если вы верите, что преодоление искусственно созданных трудностей делает программирование интересным, сочинение А. Фьюэра доставит вам массу удовольствия.

Качество перевода этой книги оставляет желать лучшего. Например, английская идиома «from scratch» (с самого начала) переведена как «с бесформенных, случайных соображений», фраза «ее трудно найти с помощью текстового редактора» переводится «для ее отыскания нужен текстовый редактор», «modest familiarity with external characteristics» (буквально: умеренное знакомство с внешними характеристиками) переведено как «хорошее знакомство с внутренними характеристиками» и т. д. Часто небрежность перевода делает непонятным смысл некоторых фраз, и лишь посмотрев в английский оригинал, можно догадаться, что имели в виду авторы.

Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. М.: Мир, 1976, 735 с.; Т. 2. Получисленные алгоритмы. М.: Мир, 1977. 724 с.; Т. 3. Сортировка и поиск. М.: Мир, 1978.

Книги представляют собой начало серии, которая, как сказал Ч. Уэзерелл, «имеет все шансы стать Библией программистов, если Кнут когда-нибудь ее за-

кончит» (быть может, сравнение с Ветхим заветом было бы точнее). Содержит описание и глубокий математический анализ большого количества алгоритмов, а также обсуждение различных математических проблем, с ними связанных. Единственный — но, возможно, роковой — недостаток книги состоит в принятом в ней способе записи алгоритмов. Их словесные описания, и тем более программы на языке гипотетической машины MIX, являющиеся в книге единственным средством для формальной записи алгоритмов, неудобочитаемы. Обращение к книгам Кнута, таким образом, становится последним средством, которое приходится применять, если в более доступных источниках нужно-го вам материала не оказалось.

Мейер Б., Бодуэн К. Методы программирования. М.: Мир, 1982, Т. 1. 356 с.; Т. 2, 368 с.

Если неумолимый рок бросит вас в пучину производственного программирования на языках Фортран или ПЛ/1, эта книга станет той соломинкой, за которую хватает утопающий: как говорит в своем предисловии редактор перевода (А. П. Ершов), «авторы сумели . . . внушить читателю оптимизм в его стремлении научиться хорошо составлять программы, не дожидаясь создания «идеальных» языков программирования».

Книга знакомит с основными конструкциями программирования и показывает, как эти конструкции выражаются на любом из трех языков: Фортране, Алголе W, ПЛ/1. Подробно описываются подводные камни, заготовленные разработчиками этих языков. Неизбежным последствием такого подхода является многословие: небольшой по объему материал превращается при таком «параллельном переводе» в объемистый двухтомник. Авторы излагают правила Хоара, однако используют их лишь для проверки готовых программ (и то изредка), но не как средство написания программ.

Названия глав: I. Общие сведения о программировании. II. Введение в языки программирования Фортран, Алгол W, ПЛ/1. III. Управляющие структуры. IV. Подпрограммы. V. Структуры данных (т. 1). VI. Рекурсия. VII. Алгоритмы. Глава посвящена описанию наиболее важных алгоритмов сортировки и поиска. VIII. На путях к методологии (т. 2).

Турский В. Методология программирования. М.: Мир, 1981. 265 с.

Книга содержит много интересных замечаний, от-

носящихся как к программированию «в малом» (разработке и исследованию небольших программ), так и к программированию «в большом», т. е. более крупных задач. Помимо собственных соображений автора, приводится содержание большого числа статей. К сожалению, обычная невразумительность статей по программированию сказалась и на их изложении. Так что не удивляйтесь, если какие-то части книги покажутся вам малопонятными. Несмотря на это, книга отличается (в лучшую сторону) от многих других книг, рассматривающих вопросы программирования «в большом».

Названия разделов: 1. Введение. 2. Основные программные конструкторы. 3. Взаимодействие модулей. 4. Проектирование программ.

Уэзерелл Ч. Этюды для программистов. М.: Мир, 1982. 287 с.

Книга содержит 27 тем для учебных программистских проектов. Среди них раскрашивание карты методом исчерпывающего поиска; сжатие файла; моделирование машины Тьюринга; программа, играющая в калáх, и многое другое. Частью задач является разработка точных спецификаций по неформальным описаниям назначения программ.

Книга приятно выделяется среди стандартных задачников по программированию (где, как правило, нужно искать незаконные идентификаторы, пропущенные запятые и т. п.). Однако программы, приведенные в ней в качестве решений (ими снабжены две из 27 задач), никак нельзя назвать образцовыми. Они, как и общий стиль книги, позволяют почувствовать предпочтение, отдаваемое автором внешнему украшательству (красивой форме ввода-вывода и т. п.) перед изяществом программ. Особенно это видно по запутанному решению задачи о раскрашивании карты, представляющей собой на самом деле простое упражнение по обходу дерева.

Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. М.: Финансы и статистика, 1982. 152 с.

Книга состоит из двух частей: «Руководства для пользователя» (К. Йенсен, Н. Вирт) и «Сообщения» (Н. Вирт). Сообщение представляет собой описание синтаксиса и (частично) семантики Паскаля. Является, по-видимому, лучшим справочным материалом по

Паскалю. К сожалению, это означает лишь, что остальные руководства еще хуже; «Сообщение» Вирта страдает невразумительностью, характерной практически для всех современных описаний языков программирования. Руководство для пользователя содержит примеры применения конструкций языка.

Вирт Н. Систематическое программирование. М.: Мир, 1977. 181 с.

Из предисловия автора: «Мой главный замысел — представить программирование как искусство или технику конструирования... алгоритмов, причем представить как систематическую дисциплину ...Студентов нужно обучать конструированию алгоритмов... знакомя их с задачами и приемами, типичными для программирования и не зависящими от конкретных приложений... По этой причине никакая специальная область... не выделена как самоцель. По той же причине особо не выделяется нотация или язык программирования: язык — наш инструмент, но не цель... Я попытался включить в книгу описание основных идей и методов проверки правильности программ... Основные понятия, которые отобраны для этой книги (...идеи утверждений и инвариантов), имеют такое фундаментальное значение, что их нельзя отнести ни к каким другим учебным курсам, кроме как к самым начальным ...Эта книга написана для тех, кто рассматривает курс систематического конструирования алгоритмов как часть своего общего математического образования, а не для тех, кому изредка приходится составлять программу...»

Намерения автора достаточно ясно выражены в предисловии. Следует отметить, однако, что далеко не все программы книги снабжены доказательствами их правильности. Кроме того, эти доказательства иногда даются *post factum*, как дополнение к уже готовой программе, а не строятся одновременно с ее написанием (это и не удивительно: английский оригинал книги вышел сравнительно давно, в 1973 г.).

Сказанное не мешает книге Вирта быть одним из лучших имеющихся в настоящее время учебников программирования. Изложение достаточно сжатое, но ясное. Используемая при записи алгоритмов нотация близка к языку Паскаль, разработанному тем же Н. Виртом.

Названия разделов: 1. Введение. 2. Основные по-

нения. 3. Структура вычислительных машин. 4. Средства и системы программирования. 5. Некоторые примеры простых программ. 6. Конечность программ. 7. Последовательная нотация и языки программирования. 8. Типы данных. 9. Программы, основанные на рекуррентных соотношениях. 10. Файловая структура данных. 11. Массив как структура данных. 12. Подпрограммы, процедуры и функции. 13. Преобразование представлений чисел. 14. Обработка текстов с использованием массивов и файлов. 15. Пошаговая разработка программ.

Как отмечается в предисловии Е. П. Велихова, книга эта необычная — в существенной части она представляет собой переработанные материалы занятий по информатике, проводившихся в математических классах школы № 57 г. Москвы. Мы пользуемся возможностью поблагодарить учеников математических классов этой школы, принимавших участие в занятиях, особенно Л. Астрина, А. Катанову, Ю. Махлина, Т. Мисирпашаева, А. Погосянц, А. Сиваченко, А. Хачатуряна, С. Штейнгольда, М. Юдашкина. Каждый когда-нибудь работавший со школьниками знает, что книга такого типа как наша, всегда — результаты совместного труда преподавателей и учеников. Мы благодарны также Л. М. Дубинскому, А. Кириллову, А. Г. Когану, А. Г. Кушниренко, Г. В. Лебедеву, М. А. Макаревскому, А. Г. Микерину, М. А. Ройтбергу, Е. Т. Семеновой, Р. Л. Татузову, В. В. Фоку, А. Шеню, Ф. Н. Шерстюку и всем остальным, участвовавшим в обсуждении задач и программ, вошедших в книгу.

На заключительном этапе обработку материалов провели: А. Л. Семенов (главы 1, 3, 4, 9, 14, 15, 17—19, 20, 22), А. Шень (главы 1, 4—16, 18—20, 23, 24); А. Г. Коган (главы 1, 2, 24), А. А. Кириллов (глава 23), М. А. Ройтберг (глава 1), М. Юдашкин (глава 14), С. Штейнгольд (глава 11), Л. Астрин (глава 18).

При подготовке рукописи были использованы программы экранного редактирования и распечатки текстов, составленные Л. Астриным, А. Сиваченко, А. Хачатуряном, М. Юдашкиным (школа № 57), Р. Л. Татузовым и А. Шенем (ИППИ АН СССР).

Вишняков Юрий Саввич, кандидат физико-математических наук, заместитель председателя Научного совета АН СССР по комплексной проблеме «Кибернетика»

Грюнталь Андрей Игоревич, кандидат физико-математических наук, заведующий сектором Института автоматизации проектирования (ИАП) АН СССР

Кольцова Анастасия Адриановна, кандидат технических наук, заведующая сектором ИАП АН СССР

Пачков Степан Александрович, старший научный сотрудник Рабочей консультативной группы при Президенте АН СССР

Семенов Алексей Львович, доктор физико-математических наук, заведующий лабораторией Научного совета АН СССР по комплексной проблеме «Кибернетика»

Оглавление

Предисловие	3
Глава 1. Команды исполнителю	7
Глава 2. Коза, капуста и другие с точки зрения программиста	17
Глава 3. Путник снова в лабиринте	23
Глава 4. Доказательства в программировании	33
Глава 5. Алгоритм Евклида	40
Глава 6. Кто тяжелее, или Нижние и верхние оценки для задачи сортировки	48
Глава 7. Сколько веревочке ни виться, или Почему программы кончают работу	54
Глава 8. Снова о сортировке	60
Глава 9. Могут ли восемь ферзей не бить друг друга, или Обход дерева	65
Глава 10. Можно ли поднять себя за волосы, или Рекурсия	80
Глава 11. От рекурсивного определения к программе	86
Глава 12. Ханойские башни	90
Глава 13. Вычисления и вычислительные машины	96
Глава 14. Переборные задачи	104
Глава 15. Игры, игры, игры	112
Глава 16. Снова об играх	117
Глава 17. Гениальный режиссер и его жертва	121
Глава 18. Редактор текстов, или Зачем компьютер грамотному	123
Глава 19. Исполнитель-черепаха, или Язык Лого	126
Глава 20. Игра в животных, или Искусственный интеллект	129
Глава 21. Программистские басни	132
Глава 22. Компьютеры и общество	136
Глава 23. Паскаль для начинающих	143
Глава 24. Рекомендации по дальнейшему чтению	153
От авторов	171
Об авторах	172

Научно-популярное издание

ПРОСТОЕ И СЛОЖНОЕ В ПРОГРАММИРОВАНИИ

Утверждено к печати
редколлегией серии
«Научно-популярная литература
Академии наук СССР»

ИБ № 37159

Редактор
Н. Б. Прокофьева

Художник
А. М. Драговой

Художественный редактор
В. С. Филатович

Технический редактор
А. М. Сатарова

Корректоры
В. А. Алешкина,
Р. З. Землянская

Сдано в набор 11.12.87
Подписано к печати 4.07.88
Т-00189 Формат 84×108¹/₃₂
Бумага кн.-журнальная
Гарнитура обыкновенная
Печать высокая
Усл. печ. л. 9,24.
Усл. кр. отт. 9,45.
Уч.-изд. л. 9,3
Тираж 100 000 экз. Тип. зак. 1199
Цена 35 коп.

1-й завод (1—50 000 экз.)
Ордена Трудового Красного
Знамени
издательство «Наука»
117864, ГСП-7, Москва, В-485,
Профсоюзная ул., 90

2-я типография издательства
«Наука»
121099, Москва, Г-99,
Шубинский пер., 6

ИЗДАТЕЛЬСТВО «НАУКА»
ВЫШЛИ ИЗ ПЕЧАТИ КНИГИ:

**ПЕРСОНАЛЬНЫЙ КОМПЬЮТЕР
В ИГРАХ И ЗАДАЧАХ**

Впервые в отечественной научно-популярной литературе дается обширная систематизированная подборка занимательных игр для персонального компьютера. Осваивая эти игры, читатель одновременно получает полезные навыки обращения с персональным компьютером и составления простых программ на языке Бейсик. Книга содержит тексты программ на Бейсике различного характера: вычислительные, программы-тренажеры, игровые программы.

**КОМПЬЮТЕРЫ, МОДЕЛИ,
ВЫЧИСЛИТЕЛЬНЫЙ ЭКСПЕРИМЕНТ**

Работа представляет собой популярное введение в новую научную дисциплину — информатику. Объясняется изменение стиля мышления в связи с появлением и широким распространением компьютеров. Ведущие ученые — авторы книги — раскрывают ключевую роль математического моделирования и вычислительного эксперимента в информатике с помощью ряда ярких и наглядных примеров. Обсуждаются важнейшие перспективы применения вычислительного эксперимента в науке и технике.

Для получения книг почтой заказы просим направлять по адресу:

Адреса магазинов «Академкнига»:

- 480091 Алма-Ата, ул. Фурманова, 91/97 («Книга — почтой»);
370001 Баку, ул. Коммунистическая, 51 («Книга — почтой»);
232600 Вильнюс, ул. Университето, 4;
690088 Владивосток, Океанский проспект, 140 («Книга — почтой»);
320093 Днепропетровск, проспект Гагарина, 24 («Книга — почтой»);
734001 Душанбе, проспект Ленина, 95 («Книга — почтой»);
375002 Ереван, ул. Туманяна, 31;
664033 Иркутск, ул. Лермонтова, 289 («Книга — почтой»);
420043 Казань, ул. Достоевского, 53 («Книга — почтой»);
252030 Киев, ул. Ленина, 42;
252142 Киев, проспект Вернадского, 79;
252030 Киев, ул. Пирогова, 2;
252030 Киев, ул. Пирогова, 4 («Книга — почтой»);
277012 Кишинев, проспект Ленина, 148 («Книга — почтой»);
343900 Краматорск, Донецкой обл., ул. Марата, 1 («Книга — почтой»);
660049 Красноярск, проспект Мира, 84;
443002 Куйбышев, проспект Ленина, 2 («Книга — почтой»);
191104 Ленинград, Литейный проспект, 57;
199164 Ленинград, Таможенный пер., 2;
196034 Ленинград, В/О, 9 линия, 16;
197345 Ленинград, Петрозаводская ул., 7 («Книга — почтой»);
194064 Ленинград, Тихорецкий проспект, 4;
220012 Минск, Ленинский проспект, 72 («Книга — почтой»);
103009 Москва, ул. Горького, 19а;
117312 Москва, ул. Вавилова, 55/7;
117192 Москва, Мичуринский проспект, 12 («Книга — почтой»);
630076 Новосибирск, Красный проспект, 51;
630090 Новосибирск, Морской проспект, 22 («Книга — почтой»);
142284 Протвино Московской обл., ул. Победы, 8;
142292 Пущино Московской обл., МР, «В», 1 («Книга — почтой»);
620151 Свердловск, ул. Мамина-Сибиряка, 137 («Книга — почтой»);
700000 Ташкент, ул. Ю. Фучика, 1;
700029 Ташкент, ул. Ленина, 73;
700070 Ташкент, ул. Шота Руставели, 43;
700185 Ташкент, ул. Дружбы народов, 6 («Книга — почтой»);
634050 Томск, наб. реки Ушайки, 18;
634050 Томск, Академический проспект, 5;
450059 Уфа, ул. Р. Зорге, 10 («Книга — почтой»);
450025 Уфа, ул. Коммунистическая, 49;
720000 Фрунзе, бульвар Дзержинского, 42 («Книга — почтой»);
310078 Харьков, ул. Чернышевского, 87 («Книга — почтой»).

35 коп.