Б. МЕЙЕР, К. БОДУЭН

МЕТОДЫ ПРОГРАММИРОВАНИЯ 2

Перевод с французского
Ю. А. ПЕРВИНА

под редакцией
А. П. ЕРШОВА

Издательство «Мир» Москва 1982

ББК 32.973
М 45
УДК 681.142.2

М45 Мейер Б., Бодуэн К.

Методы программирования: В 2–х томах. Т.2. Пер. с франц. Ю.А. Первина. Под ред. А.П. Ершова.–М.: Мир, 1982 368 с.

Второй том монографии французских ученых, посвященной основным понятиям информатики и трудным проблемам методологии программирования.

В гл. VI–VIII рассматриваются понятие рекурсии и эффективные алгоритмы. Последняя глава посвящена общим аспектам методологии программирования.

Книга рассчитана на профессиональных программистов, желающих овладеть современными методами программирования. Может служить учебным пособием по языкам программирования и алгоритмам.

Редакция литературы по математическим наукам

ОГЛАВЛЕНИЕ

ГЛАВА VI.
РЕКУРСИЯ

У попа была собака.
Он ее любил.
Она съела к;усок мяса –
Он ее убил.
Убил и закопал.
И надпись написал:
«У попа была собака.
Он ее любил ...[1]
(Из русского фольклора)

РЕКУРСИЯ

VI.1. В защиту рекурсии
VI.2. Несколько рекурсивных программ
VI.3. Практическая реализация рекурсии
VI.4. Восходящее рекурсивное вычисление
VI.5. Применение: алгоритмы последовательных испытаний
УПРАЖНЕНИЯ

Рекурсивные программы, т.е. программы, которые могут обращаться к самим себе, естественным образом вводятся во всех областях программирования. Тем не менее у многих «практиков» наблюдается отрицательное отношение к рекурсии, потому что она создает видимость порочного круга (что на самом деле не так), поскольку понимание рекурсивных алгоритмов у тех, кто не привык работать с ними, требует порой высокого уровня абстракции; и, потому что два наиболее распространенных языка ФОРТРАН и КОБОЛ, как правило, не разрешают писать непосредственно рекурсивные программы, – это, однако, не означает, что в этих языках невозможно создавать рекурсивные алгоритмы.

Цель данной главы состоит в том, чтобы показать полезность, важность и необходимость этого понятия. Мы приведем ряд задач, которые решаются с помощью рекурсии просто и удобно (в частности, класс алгоритмов «последовательных испытаний» или же алгоритмов «с возвратом», в которых сложная на вид управляющая структура изящно выражается с помощью рекурсии); покажем, как программировать рекурсивный алгоритм на таком языке, как ФОРТРАН, который не «разрешает рекурсии», и исследуем те упрощения, которые позволяют повысить эффективность при выполнении рекурсивного алгоритма.

[1] Авторы выбрали в качестве эпиграфа слова героя одной из пьес Самуэля Беккета, являющиеся перефразировкой этих «стихов».– *Прим. перев.*

VI.1. В защиту рекурсии

VI.1.1. Введение

Рекурсия, т.е. возможность ввести в определение объекта ссылку на сам объект, часто возникает в программировании.

Рекурсия является одним из фундаментальных «концептуальных инструментов», имеющихся в распоряжении программиста. К сожалению, она освоена далеко не достаточно; многие программисты избегают простых рекурсивных решений; другие относятся к рекурсии, как господин Журден к прозе[1]: часто встречаются программы на ФОРТРАНе – некоторые из них даже публикуются, – в которых легко обнаруживается, что сложная обработка массивов (являющихся, например, представлениями стеков) с иерархической структурой управления сводится к незамеченному программистом рекурсивному алгоритму, например к обходу дерева.

Рекурсивные механизмы и их конкретная реализация в языках заслуживают более отчетливого понимания.

VI.1.2. Рекурсивные определения и рекурсивные программы

Примечание: В этой главе любое непосредственно воспринимаемое появление рекурсии, т.е. любой элемент в определении объекта., обращающийся к самому этому объекту, отмечается *курсивом.*

В V гл. мы исследовали рекурсивные структуры данных. Так, двоичное дерево типа T может быть определено:

- либо как пустое;
- либо как совокупность элементов типа T и двух непересекающихся *двоичных деревьев.*

Как было отмечено;

тип ДВОДЕР = (корень T; слева: *ДВОДЕР;* справа: *ДВОДЕР*)

Рекурсивные определения удобны также для описания языков программирования. Так, в АЛГОЛе W блок есть заключенная в скобки операторами *BEGIN* и *END* последовательность объявлений и операторов, разделенных точками с запятой; оператор же определяется:

- либо как «простой» оператор (присваивание, чтение, запись и т.д.);
- либо как блок.

Система обозначений, применяемая, в частности, для описания такого типа рекурсивных «синтаксических уравнений», известных под названием «форма Бэкуса–Наура», широко используется для синтаксического описания языков программирования [Наур 63], начиная с АЛГОЛа 60.

Рекурсивные определения могут также применяться в повседневной жизни. Например, самый простой способ точно определить родственное отношение между двумя людьми–это сказать, что x и y являются родственниками:

- либо если y–отец, мать, сын или дочь x (или супруг, если –включить степень родства по браку),
- либо если существует некий z, такой, что x является *родственником* z, а z является родственником y.

(*Вопрос:* Почему отпадает необходимость включать случай, когда y является

[1] Имеется в виду главный герой пьесы Мольера «Мещанин во дворянстве».– *Прим. перев.*

братом или сестрой x?)

Для выражения того же самого определения нерекурсивным способом потребовалось бы прибегнуть к понятию «цепочка родственных отношений» и ввести в определение длину такой цепочки.

Особым случаем рекурсивного определения является математическое понятие рекуррентного определения. Так, коэффициенты C_n^m в треугольнике Паскаля (сочетания) могут быть определены двойной рекуррентностью:

$C_n^0 = 1$ для всякого $n \geq 0$;

$C_n^m = 0$ для $m > n \geq 0$;

$C_n^m = C_{n-1}^{m-1} + C_{n-1}^m$ для $n \geq m \geq 0$.

В этой главе нас интересуют *рекурсивные подпрограммы,* т.е. такие подпрограммы, текст которых содержит один или несколько вызовов самой подпрограммы (*прямая* рекурсивность), или, шире, подпрограммы, выполнение которых может привести к одному или нескольким вызовам самой подпрограммы. Такие подпрограммы могут непосредственно использоваться для установления, соответствует ли объект рекурсивному определению, как мы только что видели. Так, предположим существование типа ЛИЧНОСТЬ ; если для всякого x этого типа функция семья(x) дает список элементов его ближайших родственников (отец, мать, дети), то для определения, являются ли два человека родственниками, имеет программу:

```
программа родственники: ЛОГ (аргументы x, y: ЛИЧНОСТИ)
    родственники ← ложь;
    для z из семья (x) пока ~ родственники повторять
        родственники ← (z = y. или родственники (z, y))
```

Отметим, что в последней строке имя «родственники» слева от стрелки означает результат программы, а справа от стрелки означает, наоборот, результат рекурсивного вызова этой подпрограммы (оно отмечается курсивом). Эта строка означает, чтох и y являются родственниками тогда и только тогда, когда существует член z семьи x**,** такой, что либо z = y**,** либо z и y являются *родственниками***.**

Далее в этой главе мы приведем сначала несколько примеров рекурсивных алгоритмов, заимствованных из различных областей информатики; затем укажем, каким образом рекурсивные подпрограммы могут быть реализованы на машине: трансляторами, если речь идет о развитых языках, и программистами при использовании языков более низкого уровня; введем понятие восходящего рекурсивного вычисления; наконец, изучим особо важный класс рекурсивных алгоритмов–алгоритмы последовательных испытаний.

VI.1.3. Свойства рекурсивных алгоритмов

Все рекурсивные алгоритмы в целом имеют ряд свойств, которые объединяют их с «рекурсивными структурами данных» и «рекурсивными определениями», рассмотренными выше. Прежде всего рекурсия может быть **косвенной**:

```
программа a (аргумент x: ...)
    ...
    b(f(x)) {f(x) – выражение, зависящее от x};
    ...
программа b (аргумент y: ...)
    ...
    a(g(y)) {g(y) – выражение, зависящее от y};
    ...
```

Предоставим читателю поиски строгого определения понятия прямо или

косвенно рекурсивной подпрограммы (*подсказка:* это определение может быть рекурсивным).

Другое важное условие, касающееся использования рекурсии, состоит в том, что объекты, порожденные рекурсивным определением (будь то информационные структуры или вычисления), должны быть *конечными.* Следовательно, вся совокупность рекурсивных подпрограмм должна включать такое положение, при котором в определенных случаях вычисление могло бы делаться *непосредственно,* без рекурсивного вызова. Одна из подпрограмм по крайней мере должна будет, следовательно, иметь общий вид:

если с **то**
| $A_{ПРЯМ}$ {действие или выражение, выполняемое непосредственно}
иначе
| $B_{РЕК}$ {часть, включающая при необходимости рекурсивные вызовы}

или

пока ~ с **повторять**
| $B_{РЕК}$
$A_{ПРЯМ}$

или некоторую эквивалентную форму. Для того чтобы выполнение соответствующей подпрограммы могло быть завершено, необходимо, следовательно, чтобы существовала некоторая связанная с этой программой **управляющая величина** m, т.е. целое неотрицательное число, относительно которого можно было бы с уверенностью сказать, что оно строго убывает при каждом рекурсивном вызове. В простейших случаях роль такой управляющей величины может играть один из параметров подпрограммы. Так, любая подпрограмма, отвечающая следующей схеме:

программа пп (**аргументы** n: ЦЕЛ, x, y, z: ...)
| **если** n = 0 **то**
| | $A_{ПРЯМ}$(n > x, y, z, ...) {без рекурсивного вызова}
| **иначе**
| | $B_{РЕК}$ {где все рекурсивные вызовы имеют вид}
| | пп(n – 1, f(x), g(y), h(z), ...)}

после конечного числа рекурсивных вызовов выдаст определенный результат для каждого неотрицательного аргумента д

Примером такого рекурсивного вычисления, не связанного с проблемой конечности (для $m \geq 0$, $n \geq 0$), является вычисление коэффициентов треугольника Паскаля C™, рассмотренное выше. Очевидно, что управляющей величиной в этом случае является n.

Однако в общем случае ситуация не всегда столь удобна и управляющая величина не всегда так просто выражается. В качестве самостоятельного упражнения мы предлагаем читателю пример функции, определенной на совокупности положительных целых чисел следующим образом:

странность (n) = **если** n = 1 **то** 1

иначе если n четное то *странность* $\left(\frac{n}{2}\right)$

иначе *странность* $\left(\frac{3n+1}{2}\right)$

Еще более важной в теории вычислений является теорема, утверждающая, что нет **никакого общего метода**, определяющего, даёт ли результат произвольная рекурсивная подпрограмма p, примененная к произвольному данному q, после конечного числа этапов (при желании доказательство этой теоремы можно получить из доказательства упражнения III.10, дающего подобный результат для цикла **«пока»**.

Близость этих двух результатов станет более ясной в разд. VI.3.5.2, где будет показано, что цикл **«пока»** является частным случаем рекурсивной программы).

Исследование частичных свойств завершимости рекурсивных вычислений представляет собой важную область теории вычислений. Показано, например (см. упражнение VI.9), что способ передачи параметров влияет на завершимость программ: передача по имени является, например, более «надежной», чем передача значением. Действительно, вычисление параметра, передаваемого значением, может «расходиться», тогда как вызываемая подпрограмма не использует этот параметр. Для изучения этих вопросов, которые далеко выходят за рамки нашей работы, хотя они и упомянуты в упражнениях, мы отсылаем читателя к [Манна 74], [Кадью 72] или [Вийемен 73].

Во всех примерах рекурсивных алгоритмов, которые будут далее приведены, имеется некоторая убывающая неотрицательная *управляющая величина* (указанная в комментарии[1]), обеспечивающая завершимость программы, когда данные принадлежат соответствующим областям.

Поиск рекурсивных алгоритмов увлекателен: он имеет собственную технику, развитую, в частности, сторонниками языка ЛИСП [Мак–Карти 62], [Сиклосси 76], [Грессей], который дал с 1959 г. методику рекурсивного программирования. Простота и изящество этого языка особенно поразительны в связи с использованием рекурсивных структур данных («списки», ср. V.8).

Задача особенно удобна для рекурсивного анализа, если она может быть разложена на совокупность подзадач того же типа, но меньшей размерности. В таком случае общая методика такого анализа содержит три этапа:

- **параметризация задачи**, заключающаяся в выделении различных элементов, от которых зависит решение, и в частности размерности решаемой задачи, причем размерность должна (в благоприятных случаях) убывать после каждого рекурсивного вызова;
- **поиск тривиального случая и его решение**. Зачастую это ключевой этап алгоритма, который может быть разрешен непосредственно без рекурсивного вызова. При этом часто размерность задачи нулевая или равна 1 и т.п.;
- **декомпозиция общего случая**, имеющая целью привести его

к одной или нескольким задачам, в основном более простым

(меньшей размерности). На протяжении всей этой главы мы будем следовать трем этим этапам–для первых примеров явным образом, а затем неявно. Тот порядок, в котором эти этапы описаны, представляется нам наиболее эффективным в общем случае.

[1] *Бесконечная* рекурсия, которую мы несколько поспешно отклоняем, заслуживает со всей очевидностью серьезного внимания, как только мы выходим за пределы чисто практических рассмотрений. Это справедливо хотя бы потому, что она позволяет самым непосредственным образом выразить понятие бесконечности. Не говоря даже о некоторых представлениях из повседневной жизни, например рекламе (или известный пример с двумя зеркалами.-Ред.), можно утверждать, что графика Мориса Эшера или в литературе новеллы Жоржа Боргеса являются в значительной степени вариациями темы рекурсии.

VI.2. Несколько рекурсивных программ

VI.2.1. Игра «Ханойская башня»

Задача, о которой пойдет речь не столь легкомысленна, как может показаться: в действительности она приводит к простой программе, которую можно рассматривать в качестве прототипа важного класса рекурсивных алгоритмов, многочисленные примеры которых мы рассмотрим далее (обход двоичного дерева, быстрая сортировка сегментацией).

Священники некой секты должны решить следующую задачу (Рис. VI.1): 64 кольца (из которых на рисунке приведены только 4), лежащие одно на другом в порядке убывания размеров на основа нии A, должны быть перемещены на основание B при использовании «промежуточного» основания C.

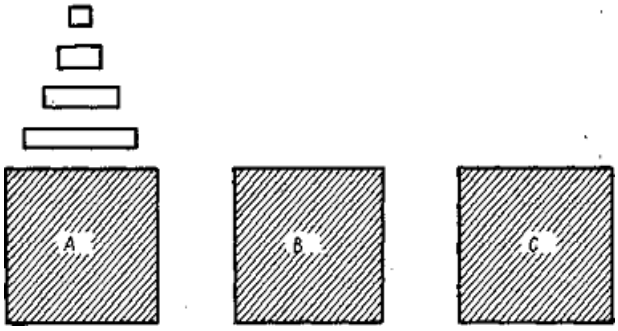


Рис. VI.1 Ханойская башня (начальное положение).

Единственными разрешенными перемещениями являются такие, при которых кольцо, взятое с вершины одной из пирамид, помещается на большее кольцо либо на пустое основание. Согласно легенде, конец мира совпадет с концом перенесения 64 колец[1]). Рассмотрим более прозаическое решение: выдачу на печать последовательности перемещений колец (см. Рис. VI.3, для решения случая с 4 кольцами). Положим, что если x и y представляют собой названия двух каких–либо оснований, то оператор

переместить (x, y)

порождает печать сообщения

переместить (x, y)

порождает печать сообщения

ПЕРЕМЕСТИТЬ КОЛЬЦО С ОСНОВАНИЯ x НА ОСНОВАНИЕ y

Читателю, которому эта задача не встречалась, советуем попытаться решить ее самому, не читая дальнейшего (в качестве колец возьмите монеты!).

В поисках рекурсивного решения этой задачи мы пройдем тремя этапами:

а) **параметризация**. Здесь естественным параметром является n – число колец. После некоторого размышления (и при некоторой интуиции) в число параметров можно включить также и основания: x (исходное основание), y (конечное основание), z (промежуточное основание). Напишем подпрограмму:

```
программа Ханой (аргументы n: ЦЕЛОЕ, x, y, z: ОСНОВАНИЯ)
    {решение задачи о Ханойской башне с n кольцами, исходное x, конечное
    y, промежуточное z}
    ...
```

Условимся, что ОСНОВАНИЕ может быть представлено ЛИТЕРОЙ (основания

[1] Задача о Ханойской башне и связанная с ней легенда принадлежат математику Эдуарду Люка [Люка 83].

обозначены "A", "B", "C") или ЦЕЛЫМ (1, 2, 3) и т.д. Решение задачи дается с помощью вызова

Ханой (64, "A". "B", "C")

b) **поиск тривиальных случаев**. Здесь тривиальным случаем будет, очевидно, такой, при котором n = 0; в этом случае просто нечего делать. Подпрограмма примет вид

```
если n > 0 то
    | ... {обработка нетривиального случая}
{иначе ничего не делать}
```

В качестве тривиального случая можно было бы также принять n = 1, тогда

```
если n = 1 то
    | переместить (x, y)
иначе
    | ... {обработка нетривиального случая}
```

c) **редукция общего случая к более простому**. Заметим здесь, что n колец могут быть перемещены с x на y путем:

- *переноса* (рекурсивно) n – 1 колец с вершины x (исходное основание) на z («промежуточное» основание) с учетом правил игры: y (конечная цель) используется как промежуточное основание;
- перемещения на y кольца (наибольшего), остающегося на x;
- *переноса* (рекурсивно) n – 1 других колец с z на y при соблюдении правил игры с x в качестве промежуточного основания[1].

Это записывается совсем просто с помощью рекурсии:

Ханой (n – 1, x, z, y);
переместить (x, y);
Ханой (n – 1, z, y, x)

Следует заметить, что правомерность этого алгоритма связана с тем, что условия, в которых находятся две подпрограммы (перенести n – 1 колец с x на z в одном случае, с z на y в другом, оставляя n–е на месте), легко отождествляются с условиями задач

Ханой (n – 1, x, z, y)

и Ханой (n – 1, z, y, z) {соответственно}

так как основание, имеющее единственное кольцо, которое больше любого из n – 1 других, приравнивается с точки зрения переноса последних (Рис. VI.2) к пустой позиции.

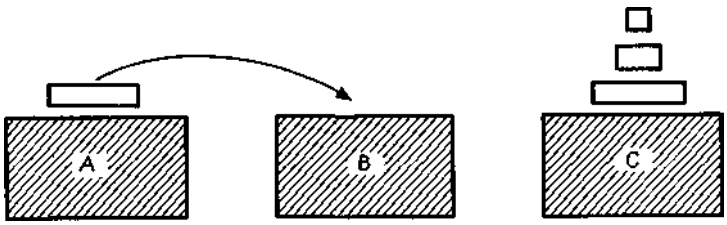


Рис. VI.2 Ханойская башня (промежуточное положение).

[1] Обратите внимание на разницу между *«перенести* p колец с x на y». что указывает на рекурсивное применение алгоритма к задаче в целом, и «переместить кольцо с x на y*. что указывает на непосредственно выполняемое действие и является к тому же единственно элементарным действием, разрешаемым правилами игры, заключающимися в перемещении одного кольца с одного основания на другое.

Подпрограмма записывается просто:

```
программа Ханой (аргументы n: ЦЕЛ, x, y, z: ЛИТЕРЫ {или нечто другое})
{Решение задачи о Ханойской башне с n кольцами; исходное основание: x,
конечное основание: y. Управляющая величина: n }
если n > 0 то
    Ханой (n – 1, x, z, y);
    переместить (x, y);
    Ханой (n – 1, z, y, x)
```

Отметим, что в этой подпрограмме единственное реальное «действие» выполняется строкой переместить (x, y) Остальное служит только для описания последовательности рекурсивных вызовов и соответствующих модификаций параметров. Однако (как бы это ни было удивительно для тех, кто не привык к рекурсии) программа АЛГОЛа W на Рис. VI.3 печатает строки, приведенные на этом же рисунке.

АЛГОЛ W

```
BEGIN
PROCEDURE HANOI (INTEGER VALUE N; STRING (1)
                 VALUE DEPART, BUT, INTERMEDIAIRE);
COMMENT: DEPART–ИСХОДНОЕ, BUT–КОНЕЧНОЕ,
INTERMEDIAIRE–ПРОМЕЖУТОЧНОЕ, HANOI–ХАНОЙ;
IF N > 0 THEN
    BEGIN
    HANOI (N–1, DEPART, INTERMEDIAIRE, BUT);
    WRITE ("ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ",
           DEPART, "В ТАБЛИЦУ", BUT);
    HANOI (N –1, INTERMEDIAIRE, BUT, DEPART)
    END
HANOI (4, "A", "B", "C")
END.
```

```
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ C
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ B
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ C В ТАБЛИЦУ B
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ C
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ B В ТАБЛИЦУ A
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ B В ТАБЛИЦУ C
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ C
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ B
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ C В ТАБЛИЦУ B
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ C В ТАБЛИЦУ A
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ B В ТАБЛИЦУ A
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ C В ТАБЛИЦУ B
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ C
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ A В ТАБЛИЦУ B
ПЕРЕМЕСТИТЬ КОЛЬЦО ИЗ ТАБЛИЦЫ C В ТАБЛИЦУ B
ВРЕМЯ ВЫПОЛНЕНИЯ 000.01 СЕКУНД
```

Рис. VI.3 Ханойская башня для n = 4 (Программа АЛГОЛ W и результаты).

Каково число ND(n) элементарных перемещений, выполняемых вышеприведенной программой? Из структуры программы легко увидеть, что

$$\text{и} \quad \begin{cases} ND(0) = 0 \\ ND(n) = 2ND(n-1) + 1 \text{ для } n > 0 \end{cases}$$

следовательно, $ND(n) = 2^n - 1$. Можно ли сделать лучше? Простое рассуждение показывает, что нет. Пусть nd(n)–минимальное число перемещений. В определенный момент переноса следует переместить наибольшее кольцо с x на y, что требует, чтобы y было пустым и чтобы третье основание z имело n – 1 других колец, уложенных обязательно в порядке возрастания диаметров, согласно правилам игры. Это требует, чтобы было выполнено по меньшей мере nd(n – 1) перемещений. Затем будет выполнено одно (перемещение большого кольца), а затем еще по меньшей мере nd(n - 1). Итак,

$nd(n) \geq 2nd(n-1) + 1 \geq 2^n - 1$ (так как $nd(0) = 0$).

Особый интерес такого *рекуррентного* рассуждения состоит в тесном сходстве, которое оно имеет с самой конструкцией *рекурсивного* алгоритма. Фактически в случае алгоритмов, оперирующих с натуральными целыми, поиск **тривиального случая** очень схож с «инициализацией» доказательства с помощью индукции; **декомпозиция** общего случая соответствует доказательству P_{n+1}, исходя из P_n. Рекурсия оказывается, таким образом, простым обобщением метода математической индукции на информационные процессы.

Для решения задачи с 64 кольцами потребуется число перемещений, равное $2^{64} - 1$, или около 10^{20}. Необходимое время – десять миллионов лет на сверхбыстродействующей ЭВМ. Что касается «конца света», то он произойдет по истечении более пяти миллиардов веков, если считать, что одно кольцо перемещается за одну секунду.

В разд. VI.3 мы исследуем нерекурсивные варианты этой программы.

VI.2.2. Численная задача

Пусть непрерывная функция f определена на отрезке [a, b] (Рис. VI.4). Требуется найти корень функции f т.е. вещественное $x \in [a, b]$ такое, что $f(x) = Q$.

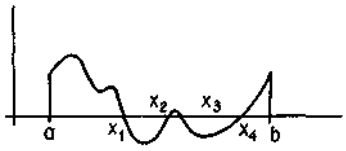


Рис. VI.4

Если функция имеет несколько корней на [a, b], то речь может идти о любом из них. Если нет ни одного корня, то выдается специальное значение, обозначенное $+\infty$

Численный анализ предлагает несколько классических алгоритмов для случая, когда f(a) и f(b) имеют разные знаки, что приводит к тому, что [a, b] содержит корень функции (в силу ее непрерывности). Исследуемый алгоритм относится к наиболее общему случаю, когда априори ничего неизвестно о знаках f на концах отрезка. Он может быть использован для определения подотрезка $[\alpha, \beta]$ из [a, b], такого, что $f(\alpha) \cdot f(\beta) \leq 0$, на котором можно будет использовать один из классических алгоритмов.

Разумеется, нельзя ожидать от программы, чтобы она дала точное значение корня или даже значение x, такое, чтобы f(x), аппроксимированная при помощи ЭВМ, равнялась нулю. Мы ищем решение с заданной точностью g, т.е. два вещественных x и y, таких, что

$$\begin{cases} x < y \le x + \varepsilon \\ f(x) \cdot f(y) \le 0 \end{cases}$$

Положим ε достаточно малым для того, чтобы

$$\begin{cases} x < y \le x + \varepsilon \\ f(x) \cdot f(y) > 0 \end{cases}$$

и можно было бы утверждать, что f не имеет корня на отрезке [x, y] (технически выбор такого ε возможен тогда и только тогда, когда функция имеет конечное число корней на отрезке [a, b]).

Если f(a) и f(b) имеют противоположные знаки, то известно, что отрезок [a, b] содержит корень, и существуют различные методики его быстрого нахождения (алгоритм Ньютона, метод ложного положения –«regula falsi»). Напротив, если f(a) и f(b) имеют одинаковый знак, как на Рис. VI.5 и Рис. VI.6, то нельзя ничего утверждать: отрезок [a, b] может как содержать, так и не содержать корень.

Идея алгоритма состоит в процедуре «дихотомии», т.е. в разделении отрезка на каждом этапе на две половины. Если один из двух вновь полученных отрезков, например [α, β], таков, что $f(\alpha) \cdot f(\beta) \le 0$ (Рис. VI.5), то, поскольку f непрерывна, [α, β] содержит корень; тогда дальнейший поиск продолжается на этом отрезке.

Если, наоборот (Рис. VI.6), два полуотрезка таковы, что f имеет одинаковый знак на обоих концах, то решение, если оно есть, может быть как в одном, так и в другом из них (на рисунке–отрезок [m, b]). В этом случае для продолжения поисков произвольно выбирается любой из двух отрезков; в случае неудачи берется отрезок, который временно игнорировали.

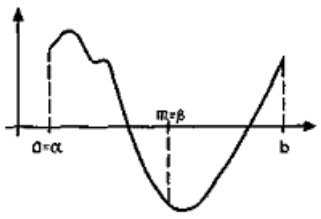


Рис. VI.5

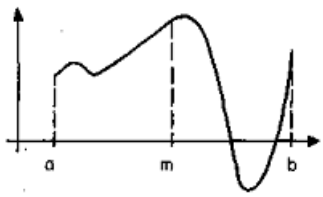


Рис. VI.6

Итак, «решением» является отрезок [α, β], такой, что

$$\begin{cases} \alpha \le \beta \le \alpha + \varepsilon \\ f(\alpha) \cdot f(\beta) \le 0 \end{cases}$$

Возможная неудача «фиксируется», если отрезок [α, β] оказывается таким, что

$$\begin{cases} \alpha \le \beta \le \alpha + \varepsilon \\ f(\alpha) \cdot f(\beta) > 0 \end{cases}$$

В этом случае поиск возобновляется на последнем из нерассмотренных отрезков.

Все это выражается проще посредством рекурсивного алгоритма: *параметризуем* задачу с помощью a и b (границы отрезка, на котором разыскивается корень и которые меняются при каждом вызове); *тривиальный случай* $|b - a| < \varepsilon$ (в этом случае известно сразу, содержит ли [a, b] решение или нет); наконец, общий случай приводится ниже:

```
программа корень : ВЕЩ (аргументы a, b: ВЕЩ, f:(ВЕЩ → ВЕЩ))
    {поиск корня f с точностью e в [a, b] при a < b; результат = ∞, если корня
    нет. Метод–дихотомия.
    Управляющая величина: ⌊(b – a)/ε⌋}
  переменная середина: ВЕЩ,
  если b – a ≤ ε то {тривиальный случай}
      если f(a) – f(b) ≤ 0 то {успех}
          корень ← a
      иначе {неудача}
          корень ← ∞
  иначе {общий случай}
      середина ← ⌊(b – a)/2⌋;
      если f(a) · f(середина) ≤ 0 то {решение слева}
          корень ← корень (a, середина)
      иначе если f(середина) · f(b) ≤ 0 то {решение справа}
          корень ← корень (середина, b)
      иначе   {попытаться с левой половиной, а потом, при неудаче, – на правой
              половине}
          корень ← корень (a, середина);
          если корень = ∞ то
                  корень ← корень (середина, b)
```

Заметим, что в случае, когда неизвестно, на левом или на правом полуотрезке следует искать решение (операторы, следующие за последним иначе), нет никаких оснований для того, чтобы всегда начинать слева, как это сделано в данной программе. Здесь можно было бы использовать «недетерминистскую конструкцию» выбрать, рассмотренную в III.3.2.2:

```
иначе
    выбрать
        корень(a, середина) ≠ +∞: корень ← корень(a, середина)
        корень(середина, b) ≠ +∞: корень ← корень(середина, b)
    иначе
        корень ← ∞
```

Вариант этой программы на ПЛ/1 приводится ниже. Процедура *ZERO (КОРЕНЬ)* включает процедуру *VALZERO (ЗНАЧЕНИЕ КОРНЯ),* которая позволяет избежать ненужной передачи параметров f и ε при каждом рекурсивном вызове. Для того чтобы не вычислять f чаще, чем это необходимо, в *VALZERO,* кроме значений a и b, передают два параметра, имеющие в качестве значения f(a) и f(b).

Использованная здесь методика (дихотомия) корректна с точки зрения численных методов. Заметим, однако, что с момента, когда найден отрезок [a, b], такой, что $f(a) \cdot f(b) < 0$, существуют алгоритмы, сходящиеся гораздо быстрее; эти алгоритмы в качестве новой рассматриваемой точки выбирают на каждом шаге не середину [a, b], а

- пересечение с 0x касательной к кривой в a и b ($a - \frac{f(a)}{f'(a)}$ или $b - \frac{f(b)}{f'(b)}$) по *методу Ньютона* (Рис. VI.7).

- пересечение с 0x линейной интерполяции f между a и b по методу *ложного положения* («regula falsi») (Рис. VI.8).

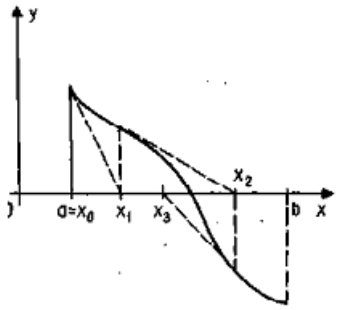


Рис. VI.7 Метод Ньютона.

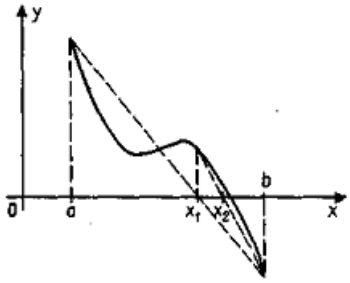


Рис. VI.8 Regula falsi.

ПЛ/1

```
ZERO: PROCEDURE (A, B, F, EPSILON) RETURNS (BINARY FLOAT);
      DECLARE (A, B, EPSILON) BINARY FLOAT;
      DECLARE F ENTRY (BINARY FLOAT)
      RETURNS (BINARY FLOAT);
VALZERO : PROCEDURE (X, Y, FX, FY) RECURSIVE RETURNS (BINARY FLOAT);
      DECLARE (X, Y, FX, FY) BINARY FLOAT;
      IF Y – X < = EPSILON THEN
                  DO;
                  IF SIGN .(FX) 1=SIGN (FY) THEN
                                          RETURN (FX);
                  ELSE RETURN (INFINI);
                  /* INFINI–БЕСКОНЕЧНОСТЬ */
                  END;
      ELSE
                  BEGIN;
                  DECLARE (MILIEU, FMILIEU) BINARY FLOAT;
                  /* MILIEU СЕРЕДИНА */
                  MILIEU = (X + Y)/2.E0;
                  /* ВНИМАНИЕ ПРИ ДЕЛЕНИИ НА КОНСТАНТУ! СМ. АБЗАЦ
                     II.5.2.4.D */
                  FMILIEU = F(MILIEU);
                  IF SIGN (FX) ¬= SIGN (FMILIEU) THEN
                        RETURN (VALZERO (X, MILIEU, FX, FMILIEU));
                  ELSE IF SIGN (FMILIEU) ¬= SIGN(FY) THEN
                        RETURN (VALZERO (MILIEU, Y, FMILIEU, FY));
                  ELSE
                        BEGIN;
                        DECLARE VALGAUCHE BINARY FLOAT;
                        /* VALGAUCHE – ЗНАЧЕНИЕ == СЛЕВА*/
                        VALGAUCHE = VALZERO(X, MILIEU, FX, FMILIEU);
                        IF VALGAUCHE ¬= INFINI
                        THEN RETURN (VALGAUCHE);
                        ELSE RETURN (VALZERO (MILIEU, Y, FMILIEU, FY));
                        END;
                  END;
      END VALZERO;
/* ТЕЛО ПРОЦЕДУРЫ ZERO */
RETURN (VALZERO (A, B, F(A), F(B));
END ZERO;
```

VI.2.3. Вычисление стоимости

Нижеследующая задача была взята из реальной практики программирования; сначала она была предназначена для решения с помощью ФОРТРАНа.

Требуется оценить стоимость функционирования *системы производства электроэнергии,* предназначенной удовлетворить *потребление* к. Эта «система» образована совокупностью n *электростанций,* каждая из которых имеет *«стоимость функционирования»* c_i ($i = 1, 2, ..., n$), и предполагается, что электростанции пронумерованы в порядке возрастания стоимостей:

$$c_1 \le c_2 \le ... \le c_n$$

Каждая электростанция состоит из некоторого числа агрегатов, которые могут работать независимо друг от друга; на этих агрегатах могут, однако, с известной вероятностью случаться аварии. В зависимости от состояния различных агрегатов (исправное или аварийное) каждая электростанция может, следовательно, находиться в одной из некоторого числа *конфигураций.* Условимся, что i–я электростанция (для $1 \le i \le n$) может находиться в t_j возможных конфигурациях; в своей j-й конфигурации ($1 < j < t_i$) она находится с вероятностью P_{ij} , и эта конфигурация обеспечивает производство электроэнергии K_{ij} (стоимость функционирования составляет тогда c_iK_{ij})

Для обеспечения полного потребления K электростанции «запускают» последовательно в порядке их нумерации, который вместе с тем является порядком возрастания стоимостей. Если на каких–либо агрегатах происходит авария, то в работу включаются исправные. Каждая конфигурация системы в целом должна обеспечивать суммарное потребление

$$\left(\sum_{\substack{\text{рабочие} \\ \text{конфигурации}}} K_{ij} \right) \ge K$$

со стоимостью

$$\sum_{\substack{\text{рабочие} \\ \text{конфигурации}}} c_i K_{ij}$$

Совокупность возможных состояний «системы» может быть представлена в виде дерева, где каждому уровню соответствует одна электростанция. Ветвям дерева соответствует выбор конфигураций работающих агрегатов для электростанции предыдущего уровня; ветви обозначены в соответствии с производимой мощностью и с вероятностью выбора соответствующей конфигурации, т.е. P_{ij} и K_{ij}.

Так, Рис. VI.9 соответствует простому частному случаю трех электростанций, каждая из которых состоит из одного агрегата с соответствующей вероятностью функционирования P_1, P_2, P_3, (следовательно, вероятности аварии $1–P_1$, $1–P_2$, $1–P_3$), производствами энергии $K_1 = 2$, $K_2 = 1$, $K_3 = 2$ и стоимостями функционирования C_1, C_2 и C_3; Полный объем электроэнергии, производство которой надо обеспечить, есть $K = 3$. На Рис. VI.9 узлы, соответствующие возможным конфигурациям, обозначены квадратами; средние стоимости функционирования составляют

$$P_1P_2(2C_1 + C_2) + P_1(1 – P_2)P_3(2C_1 + 2C_3) + (1 – P_1)P_2\ P_3(C_2 + 2C_3)$$

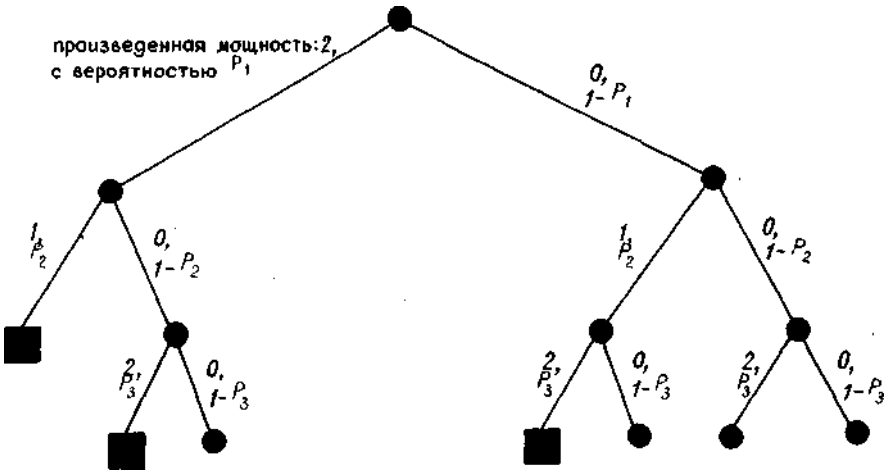


Рис. VI.9 Конфигурация электростанций.

Для общего случая задача допускает чрезвычайно простое рекурсивное решение. При *параметризации задачи* учитываются естественно параметры n (число электростанций) и K (обеспечиваемое потребление). *Тривиальным* является случай отсутствия электростанций и случай, когда при данном состоянии электростанций производство K_{ij} выше обеспечиваемого потребления. Наконец, при *декомпозиции общего случая* необходимо учитывать различные конфигурации, в которых могут оказаться каждая электростанция i (с заданными вероятностями P_{ij}).

Вычисляемая стоимость задается в виде

C = стоимость(1, K)

где функция стоимость определяется как

стоимость (i, x) = **если** i > n **то** 0

иначе $\sum_{j=1}^{t_i} P_{ij}$ *подстоимость* (i, j, x)

функция подстоимость задается в виде

подстоимость (i, j, x) = **если** $x < K_{ij}$ **то** $c_i K_{ij}$

иначе если *стоимость* (i +. 1, $x - K_{ij}$) > 0

то $c_i K_{ij}$ + *стоимость* (i + 1, $x - K_{ij}$)

иначе 0

Эти функции интерпретируются следующим образом: стоимость (i, x)–средние стоимости функционирования системы, обеспечивающей потребление x при работе лишь электростанций, пронумерованных i, i + 1, .., n; подстоимость(i, j, x) – элемент стоимостей, включенных в стоимость(i, x) как результат выбора j–й конфигурации для i–й станции (выбор делается с вероятностью P_{ij}).

Эта интерпретация достаточна для доказательства вышеприведенной формулы, которую можно также понять в терминах обхода деревьев.

Эти формулы сразу же переводятся в рекурсивную программу: вершина заменяется на цикл **для** в определении функции стоимость; в функции подстоимость используется переменная v, которой присваивается при $x > K_{ij}$ величина стоимость(i + 1, $x - K_{ij}$). Это делается во избежание двукратного рекурсивного вызова при вычислении данной величины.

Эта задача типична в классе задач, в которых речь идет о кратных суммах

$$\sum_{c(x)} f(x)$$

на множестве n–ок x вида $[i_1, i_2, ..., i_{kp}]$, отвечающем некоторому условию c(x); при этом необязательно, чтобы все $_x$ имели то же самое число k_p элементов. В подобных случаях зачастую удобной является рекурсивная формулировка.

VI.2.4. Сортировка

В гл. V мы видели, каким образом **двоичное дерево поиска** позволяет управлять множеством данных, на которых установлено отношение порядка. Например,

```
тип ЛИЧНОСТЬ = (имя: ТЕКСТ; характеристики ИНФО);
тип ИНФО = (...) {характеристики, принадлежащие "ЛИЧНОСТЬ";
тип ДВОДЕР = (корень: ЛИЧНОСТЬ; слева: ДВОДЕР; справа: ДВОДЕР)
```

Подпрограммы «включения» и «проверки принадлежности» имеют простые рекурсивные варианты, эквивалентные нерекурсивным вариантам разд. V.8:

```
программа включение (аргумент р: ЛИЧНОСТЬ,
                      модифицируемый параметр а: ДВОДЕР)
    если а = ПУСТО то
        а ← ДВОДЕР (р, ПУСТО, ПУСТО)
    иначе если имя (р) < имя (корень (а)) то
        включение (р, слева (а))
    иначе включение (р, справа (а))
программа наличие: ЛОГ
                  (аргументы р: ЛИЧНОСТЬ, а: ДВОДЕР)
    если а = ПУСТО то
        наличие ← ЛОЖЬ
    иначе если имя (р) = имя (корень (а)) то
        наличие ← ИСТИНА
    иначе если имя (р) < имя (корень (а)) то
        наличие ← наличие (р, слева (а))
    иначе наличие ← наличие (р, справа (а))
```

Благодаря самому определению двоичного дерева поиска более интересной является возможность (потому что соответствующая нерекурсивная программа оказывается далеко не простой) выполнить рекурсивную сортировку, т.е. напечатать множество данных, содержащихся в дереве в алфавитном порядке имен:

```
программа печатьальфа (аргумент а: ДВОДЕР)
    {печатать в алфавитном порядке данные, принадлежащие узлам дерева}
    если а ≠ ПУСТО то
        печатьальфа (слева (а));
        печатать имя (корень (а)), характеристики (корень (а))
        печатьальфа (справа (а))
```

Речь идет просто об обходе ЛКП дерева (V.7.5) (напомним, что ЛКП означает: *«обойти»* Левое поддерево; пройти Корень; *«обойти»* Правое поддерево).

Очевидный алгоритм сортировки для чтения, например, объектов типа ЛИЧНОСТЬ и печати характеристик в алфавитном порядке имен имеет вид

```
переменные р:ЛИЧНОСТЬ, а:ДЕРЕВО, ф:ВНЕШНИЙ;
а ←ПУСТО;
пока ~ конец файла (ф) повторять
    читать (ф) р;
    включение (р, а);
печатьальфа (а)
```

В гл. VII мы рассмотрим другие алгоритмы сортировки, часть которых непосредственно использует рекурсию.

Между печатьальфа и подпрограммой Ханой из VI.2.1 заметно большое сходство. В этом нет ничего удивительного: обе подпрограммы являются не чем иным, как обходом двоичного дерева ЛКП (V.7.5). Двоичное дерево, соответствующее

Ханойской башне, представлено на Рис. VI.10 при n = 3; во время обхода ЛКП это го дерева пересечение узла, обозначенного [a, b, c] приводит к выполнению

переместить (a, b)

а ветвь, отходящая от этого узла, обозначена 1 или 2 в зависимости от того, переставляет ли соответствующий рекурсивный вызов третий элемент (c) с первым (a) или со вторым (b).

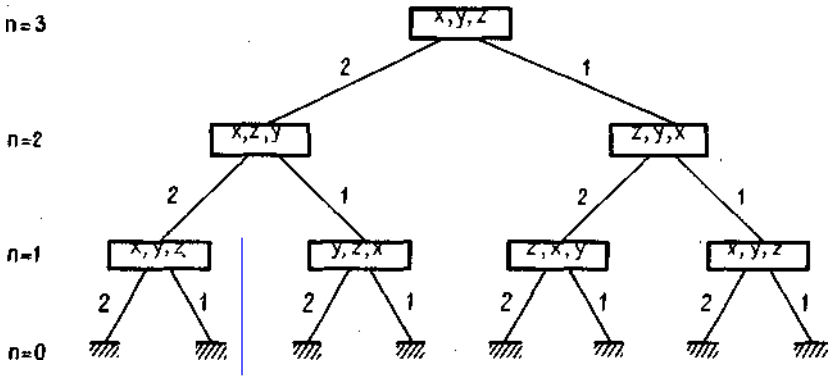


Рис. VI.10 Двоичное дерево Ханойской башни.

Программы печатьальфа и Ханой (а также Быстрая Сортировка: VII.3) являются частными случаями программы общего вида

программа p (аргумент x: ...)
| **если** x не пусто **то**
| | A_0
| | $p(x_1)$
| | A_1
| | $p(x_2)$

где A_0 и A_1 есть непосредственно выполняемые действия (без рекурсивного вызова p), а x_1 и x_2 – отдельные части данных. Такие подпрограммы представляют собой рекурсивный вариант принципа «разделяй (на два) и властвуй» (см. VII.1.2).

VI.3. Практическая реализация рекурсии

VI.3.1. Рекурсия и языки программирования

Во многих языках программирования подпрограммы могут содержать рекурсивные вызовы. Так, в ПЛ/1 подпрограмма должна иметь специальный атрибут, обозначаемый *RECURSIVE,* если она способна вызывать саму себя с помощью прямой или косвенной рекурсии. Пишут

TRUC: PROCEDURE (A, B, ...) RECURSIVE;

.../ тело процедуры TRUC*/...;*

В АЛГОЛе W, так же как и во всей серии алголоподобных языков (АЛГОЛ 60, ПАСКАЛЬ, СИМУЛА 67, АЛГОЛ 68 и др.), в ЛИСПе и т.д., любая подпрограмма может вызваться рекурсивно без необходимости специальных объявлений.

В некоторых из старых языков, среди которых наиболее известными являются ФОРТРАН и КОБОЛ, наоборот, рекурсивные вызовы запрещены. Следует заметить, что большинство трансляторов не «протестуют» против подпрограммы, содержащей прямые или косвенные рекурсивные вызовы, но транслируют некорректно, вызывая обычно зацикливание при выполнении.

Техника практической реализации рекурсии, о которой пойдет речь, в этом типе языков входит в обязанности программистов. Она дает ему возможность построить

алгоритм в рекурсивном виде для последующей адаптации алгоритма к имеющемуся в его распоряжении языку. В языках, допускающих рекурсию, соответствующие преобразования выполняются компиляторами или интерпретаторами. Знание механизмов реализации помогает, однако, программисту эффективно использовать рекурсию.

VI.3.2. Задача

Следующая программа печатает решение задачи о Ханойской башне на 7 кольцах:

```
Ханой (7, "A", "B", "C")
```

при

```
программа Ханой (аргументы n : ЦЕЛ, x, y, z : ЛИТЕРЫ)
(1)  | если n > 0 то
(2)  |       | Ханой (n – 1, x, z, y);
(3)  |       | печатать ("перенести кольцо с: ", x, " на " y);
(4)  |       | Ханой (n – 1, z, y, x)
```

Что происходит, когда программа выполняет рекурсивный вызов, например первый такой вызов строки (2)? Выполнение подпрограммы прерывается с тем, чтобы тотчас возобновить ее выполнение сначала, но *с новыми параметрами:* n заменяется на n – 1, x остается неизменным, y и z меняются местами. Можно сказать, что одно **поколение** Ханоя породило новое поколение той же подпрограммы с другими параметрами и впало затем в «зимнюю спячку».

Созданное таким образом поколение само может порождать новые поколения, и процесс будет повторяться, но не до бесконечности, а в зависимости от управляющей величины (в данном случае n), которая убывает при переходе к каждому новому поколению. Когда поколение, порожденное строкой (2), заканчивается или «умирает», породившее его материнское поколение возрождается для выполнения строки (3) при тех значениях параметров, какие оно имело в момент прерывания. Затем оно вновь порождает поколение (4) с n – 1, z, y и x в качестве новых параметров. По окончании этих дочерних поколений исходное поколение также заканчивается, так как (4) является Последней строкой подпрограммы. И только в том случае, когда она выполняется старшим поколением, соответствующим n = 7, строка (4) порождает действительный *возврат* к программе, вызвавшей Ханой.

Как видно, задача управления рекурсией заключается в том, чтобы *сохранить* при каждом рекурсивном вызове информацию, необходимую для последующего поколения, инициировавшего этот вызов, и правильно *отыскать* эту информацию, когда предыдущему поколению вновь будет передано управление.

Какая же информация должна быть сохранена? В разд. III.2.1 мы ввели понятие «состояние программы», которое в каждый момент определяется значениями переменных и содержимым «указателя выполнения», т.е. указанием активного в этот момент оператора. Именно это «состояние» и следует сохранить перед каждым вызовом, имея в виду, что в число «переменных» здесь надо включать аргументы подпрограммы (в данном случае n, x, y, z), к которым добавляются локальные переменные, если они имеются, и возможный результат подпрограммы.

Каким образом обеспечить эти последовательные запоминания и восстановления? Основное наблюдение заключается в следующем: ни одно из поколений не может «умереть» до тех пор, пока хотя бы одно из поколений, порожденных им, остается «в живых». Это же замечание может быть выражено тем фактом, что для «дерева последовательных поколений» принят обход ЛКП (V.7.5), т.е. следуя за стрелкой вдоль Рис. VI.11.

Равным образом можно сказать, что информация должна восстанавливаться в порядке, *обратном* тому, в котором она *запоминалась*. Это замечание сразу же подсказывает два решения:

а) Поведение информации, которое мы только что выявили, точно совпадает с поведением информации, управляемой в стеке (V.4), функциональная специфика которого уточняет, что данные извлекаются из стека в порядке, обратном их *засылке* в стек (V.4.2). Рекурсия реализуется, таким образом, путем установления соответствия стека указателю выполнения, каждому параметру (в частности, тому, который представляет результат подпрограммы, если он имеет место) и каждой локальной переменной. Запоминание выполняется с помощью операции засылки в стек; восстановление–с помощью выборки из стека. По завершении поколения остается завершить предшествующее поколение в том и только в том случае, когда стек не пуст.

б) Стек, однако, не является необходимым в случае, когда рекурсивный вызов заменяет значение параметра или локальную переменную, например x, на

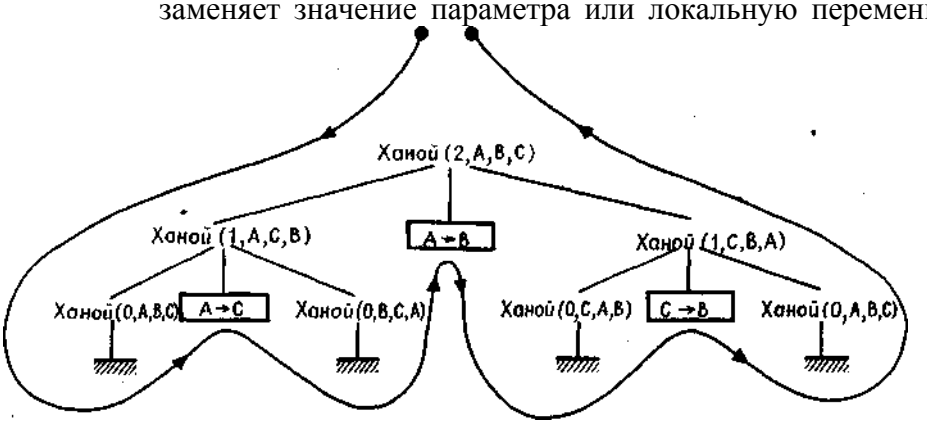


Рис. VI.11 Обход «дерева Ханоя».

новое значение $x' = f(x)$, такое, что x может быть вновь вычислено в зависимости от x', т.е. что функция f имеет обратную f^{-1}. В этом случае нет необходимости хранить x в стеке; достаточно применить при рекурсивном вызове присваивание

$x \leftarrow f(x)$

а при соответствующем возврате

$x \leftarrow f^{-1}(x)$

Например, в случае с Ханойской башней при каждом рекурсивном вызове n заменяется на $n - 1$. Поэтому отпадает надобность связывать стек с n: достаточно, чтобы каждому рекурсивному вызову предшествовало

$n \leftarrow n - 1$

и чтобы при каждом рекурсивном возврате выполнялось обратное присваивание

$n \leftarrow n + 1$

Однако при этом способе возникает проблема, так как, кроме операций «сохранение» и «восстановление», нам нужно дать средство проверки, пуст ли «стек», т.е. существует ли предшествующее поколение у рассматриваемого (это третья операция функциональной спецификации стека–операция «стекпуст»). Здесь, например, n будет соответствовать поколению, которое является предком всех других в том и только в том случае, когда $n = n_0$, где n_0 есть начальное значение n при первом вызове Ханоя. Следует, таким образом, сначала запомнить n_0.

В общем случае функция f связанная с параметром, не обязательно является той же самой для всех рекурсивных вызовов, связанных с этим же поколением; при

возврате, однако^анализ указателя выполнения позволяет найти функцию f , обратную к примененной. Таким образом, для Ханойской башни констатируем, что три последних параметра рекурсивного вызова строки (2), [x, y, z] выводятся из [x, y, z] путем перестановки y и z, а параметры рекурсивного вызова строки (4), [z, y, x] выводятся из [x, y, z] путем перестановки x и z Тогда для операции

переставить (a, b)

обратной операцией является она сама, поскольку

переставить (a, b); переставить (a, b)

равноценно пустому действию (конкретно переставить(a, b) можно записать c←a; a ← b; b ← c).

К тому же нет необходимости связывать стеки с параметрами x, y, z: просто перед каждым вызовом строки (2) ставится:

переставить (y, z)

а перед каждым вызовом строки (4)

переставить (x, y)

При рекуррентном возврате анализ указателя выполнения, взятого из стека, позволит установить, какая из операций

переставить (y, z)

или

переставить (x, z)

является обратной по отношению к применяемой.

В случае с Ханойской башней, следовательно, только указатель выполнения реально нуждается в стеке.

Теперь мы обладаем базовыми элементами для представления рекурсии: использование **стеков** и отыскание **обратных преобразований.** Несколько ниже мы приведем примеры, в которых представление рекурсии может быть значительно упрощено; пока же нам известно достаточно для того, чтобы записывать правила реализации, необязательно оптимальные, но всегда приводящие к правильным результатам.

VI.3.3. Практические правила реализации

Для определенности предположим, что мы хотим перевести рекурсивный алгоритм на ФОРТРАН. Методы остаются теми же в любом языке программирования, в ассемблере или в машинном языке.

VI.3.3.1. Предварительная обработка

а) Первый этап состоит в написании программы на «псевдо–ФОРТРАНе», допускающем рекурсивные вызовы (позаимствуем у лингвистов обозначение в виде звездочки для изображения строк, не принятых формальными правилами). Например, представляя имена колец Ханойской башни в виде целых, получают

```
      SUBROUTINE HANOI (N, X, Y, Z)
      INTEGER N, X, Y, Z
      IF (N. EQ. 0) GOTO 1000
          * CALL HANOI (N – 1, X, Z, Y)
            CALL DEPLAC (X, Y)
          * CALL HANOI (N – 1, Z, Y, X)
1000  RETURN
      END
```

б) Затем сделаем так, чтобы программа имела только один оператор *RETURN* и что бы он был последним. Это, впрочем, соответствует рекомендации, которая была дана в гл. IV; мы придерживаемся ее и в нашем примере.

в) Первому оператору программы присваивается метка ℓ_0 (возьмем $\ell_0 = 100$), а оператору, следующему за каждым рекурсивным вызовом, присвоим $\ell_1, \ldots \ell_n$ (здесь возьмем $\ell_1 = 200$; ℓ_2 есть метка оператора *RETURN*, например**,** равная 1000):

```
      SUBROUTINE HANOI (N, X, Y, Z)
      INTEGER N, X, Y, Z
100   IF (N. EQ. 0). GOTO 1000
            *  CALL HANOI (N – 1, X, Z, Y)
200         CALL DEPLAC (X, Y)
            *  CALL HANOI (N – 1, Z, Y, X)
1000  RETURN
      END
```

г) Преобразуем циклы со счетчиком, содержащие рекурсивные вызовы, в циклы **пока**.

VI.3.3.2. Реализация рекурсивных вызовов

Подготовив таким образом программу, займемся теперь реализацией рекурсивных вызовов. Для этого каждый рекурсивный вызов заменяется на последовательность операторов:

а) запоминающих переменные и аргументы, которые должны быть сохранены. В данном случае мы видели, что нет надобности запоминать *N*, *X*, *Y* или *Z*, поскольку выполняемые преобразования обратимы, в общем случае некоторые значения должны были бы засылаться в стек;

б) запоминающих указатель выполнения. При программировании на машинном языке запоминаемая величина (в стеке) является адресом. На ФОРТРАНе это будет номер рекурсивного вызова (1–для первого, 2–для второго), который мы поместим в стек целых;

в) (затем) присваивающих аргументам значения, которые они должны получить при новом рекурсивном вызове (в нашем случае при первом рекурсивном вызове надо поменять местами *Y* и *Z*, при втором – *X* и *Z* и уменьшить *N* на 1 в обоих случаях). Следует сделать так, чтобы эти модификации не воздействовали ни на какие программы, кроме преобразованной программы, обеспечивая вызов по значению[1];

г) наконец, выполняющих ветвление в начале программы с помощью оператора *GOTO* ℓ_0, где, напомним, ℓ_0 есть метка первого оператора.

Таким образом, для нашего примера имеем:

```
      *  SUBROUTINE HANOI (N, A, B, C)
         INTEGER N, A, B, C C
         ***  ВНИМАНИЕ ЭТО РЕЗУЛЬТАТ НЕПОЛНОГО С ПЕРЕВОДА***
         INTEGER X, Y, Z, U
C        ПЕРЕДАЧА ЗНАЧЕНИЕМ
         X = A
         Y=B
         Z = C
100      IF (N. EQ. 0) GOTO 1000
```

[1] Теоретически эта предосторожность бесполезна, поскольку при помощи совокупности последовательных преобразований и восстановлений параметры в конце концов должны обрести свои начальные значения. Практически же она необходима из–за возможных ошибок в ходе выполнения подпрограммы при возможном обобществлении данных и в различных сортировках параметров, встречающихся в этом разделе.

```
                CALL EMPIL (1)
C               EMPIL–ПОДПРОГРАММА ЗАСЫЛКИ В СТЕК
                U = Z
                Z= Y
                Y= U
                N = N–1
                GOTO 100
200             CALL DEPLAC (X, Y)
                CALL EMPIL(2)
                U = Z
                Z = X
                X = U
                N = N – 1
                GOTO 100
1000        RETURN
            END
```

VI.3.3.3. Перевод возвратов

Последний этап заключается в замене *RETURN* (единственного в конструкции оператора) на сложный оператор, который:

- выполняет фактически *RETURN* если (и только если) стек пуст;
- иначе, восстанавливает указатель выполнения, т.е. в ФОРТРАНе вновь находит (обычно с помощью выборки из стека) номер последнего выполненного рекурсивного вызова, который будет присвоен переменной *NUMAPP*; восстанавливает переменные последнего приостановленного поколения либо при помощи выборки из стека, либо путем обратного преобразования (в случае необходимости) функции указателя выполнения, который только что извлечен из стека (таким образом, в нашем случае выполняется:

  если *NUMAPP* = 1 **то** переставить (*Y*, *Z*)
  иначе {т.е. если *NUMAPP* = 2} переставить (*X*, *Y*);

  наконец, выполняет переход к оператору, следующему за последним рекурсивным вызовом. В ФОРТРАНе, где эти операторы предварительно получили метки ℓ_1, ℓ_2, ..., ℓ_n мы обычно используем «индексированное ветвление» (III.5.3.2)

  GOTO (ℓ_1, ℓ_2, ..., ℓ_n) *NUMAPP*

Для Ханойской башни это дает вариант, приведенный ниже, где добавлен начальный оператор, засылающий в стек значение –1, которое не может быть помещено в стек алгоритмом позднее: благодаря этой уловке можно извлекать из стека *до* его проверки на пустоту; тогда проверка состоит в сравнении извлеченного элемента с – 1.

ФОРТРАН

```
        SUBROUTINE HANOI (N, A, B, C)
        INTEGER N, X, Y, Z
C       ЗАДАЧА ХАНОЙСКОЙ БАШНИ
        INTEGER X, Y, Z. U, NUMAPP
C
C       --- ПЕРЕДАЧА ЗНАЧЕНИЕМ ---
        X = A
```

```
        Y=B
        Z = C
        CALL EMPIL (–1)
100     IF (N. EQ. 0) GOTO 1000
                CALL EMPIL (1)
                U = Z
                Z=Y
                Y= U
                N = N–1
                GOTO 100
200             CALL DEPLAC (X, Y)
                CALL EMPIL (2)
                U = Z
                Z = X
                X = U
                N = N – 1
                GOTO 100
1000    CALL DEPIL (NUMAPP)
        IF (NUMAPP. EQ–1) GOTO 2000
                    N = N + 1
                    IF (NUMAPP. EQ. 2) GOTO 1100
C       ЗДЕСЬ NUMAPP = 1 : ПОЛУЧЕНО ПРИ ПЕРВОМ ВЫЗОВЕ
                              U = Z
                              Z=Y
                              Y= U
                              GOTO 200
C       ДАЛЕЕ NUMAPP = 2 : ПОЛУЧЕНО ПРИ ВТОРОМ ВЫЗОВЕ
1100            U = Z
                Z = X
                X = U
                GOTO 1000
2000    RETURN
        END
```

VI.3.3.4. Случай параметров–результатов

До сих пор мы работали главным образом с **параметрами–аргументами.** Параметры как **результаты** подпрограммы (в частности, результат подпрограммы «выражения») являются предметом специальной обработки. Пусть x – такой параметр, а y – соответствующий фактический параметр при рекурсивном вызове:

```
программа p (...; результат x: ...; ...)
        |
        |   ...
        |   p(..., y, ...)
```

y должно быть элементом, способным принимать значение (переменная, элемент массива, формальный параметр и т.д.). Существуют два случая:

- если y есть x, то можно не запоминать значение x: рекурсивный вызов должен вычислить его новое значение;
- наоборот, если y не есть x, то значение x должно быть «сохранено» до рекурсивного вызова; и при возврате последнее значение x должно присваиваться y перед «восстановлением» x.

Это правило будет проиллюстрировано на примере с алгоритмом, который подробно

рассматривается в упражнении VII.2 («минимум и максимум одновременно»). Речь идет об одновременном определении минимума и максимума массива:

массив t [1 : N] : **ВЕЩ**

{или любой другой тип с упорядоченными значениями}

Для их раздельного определения потребовалось бы 2(N – 1) сравнений. Можно ограничиться примерно 3N/2 сравнениями, записав по меньшей мере в случае $N = 2^n$ (общий случай разбирается в упражнении VII.2):

переменные мин, макс: **ВЕЩ;**

минимакс (1, N, мин, макс)

где

программа минимакс (**аргументы** i, y. ЦЕЛЫЕ; **результаты** m, M: ВЕЩ)
| {вычисление минимума и максимума t}
| {неявный доступ к массиву t (обобществление)}
| **переменные** m', M': ВЕЩ;
| **если** j = i + 1 **то**
| | m ← минимум (t[i], t[j]);
| | M ← максимум (t [i], t [j])
| **иначе**
| | *минимакс* $\left(i, \left\lfloor \frac{i+j}{2} \right\rfloor, m, M\right)$;
| | *минимакс* $\left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1, j, m', M'\right)$;
| | m ← минимум (m, m')
| | M ← максимум (M, M')

([x] обозначает целую часть вещественного x).

Получаем нерекурсивный вариант (с дополнительной переменной номер):

(1) **если** j = i + 1 **то**
m ← минимум (t[i], t[j]);
M ← максимум (t[i], t[j])
на (5);
(2) засылка (i, j, 3); {в этом порядке}
$j \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
на (1);
(3) засылка (m, M, 4); {в этом порядке}
$i \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
на (1);
(4) m ← минимум (m, m')
M ← максимум (M, M');
(5) **если** стек не пуст **то**
| выборка (номер);
| **если** номер = 3 **то**
| | выборка (i. j) {в этом порядке};
| | **на** (3)
| **иначе** {номер = 4}
| | выборка (M', m') {в этом порядке}
| | **на (4)**

Предвосхищая последующие разделы, заметим, что засылка в стек i и j в (3) и соответствующая выборка из стека в (5) при номер = 4 излишни.

VI.3.3.5. Дополнение: косвенная рекурсия–исключение аргументов

В целях упрощения изложения мы пренебрегли преобразованиями, которые могут потребоваться перед вышеописанной обработкой.

Первое из этих преобразований касается рекурсивных подпрограмм,

вызываемых теперь не прямо, а *косвенно*. Прежде чем применять вышеописанный способ, связное множество взаимно рекурсивных подпрограмм должно быть перегруппировано в единый программный модуль; при трансляции рекурсивного (косвенного) вызова появляется переход в начало рабочей программы, соответствующей вызываемой подпрограмме (а не в начало программного модуля, как это было для прямых рекурсивных вызовов).

Второе преобразование является упрощением, заключающимся в том, что при переводе рекурсии обрабатываются только параметры, которые могут измениться в ходе по меньшей мере одного вызова. Такое упрощение стоит выполнять, даже если рекурсивная форма программы допускается, в частности в таких языках, как АЛГОЛ W и ПЛ/1. Именно так было сделано в VI.2.2, когда говорилось о подпрограмме *ZERO(A, B, F, EPSILON)* на языке ПЛ/1. Поскольку два параметра *F* и *EPSILON* не подлежат изменению, мы записали *ZERO* в виде нерекурсивной процедуры, которая вызывает, однако, локальную рекурсивную процедуру *VALZERO:* при каждом вызове *VALZERO* передаются только изменяемые параметры.

VI.3.4. Представление стеков

Нам остается указать на общие способы представления стеков и соответствующих подпрограмм в ФОРТРАНе (они были рассмотрены в гл. V), способы, которые наилучшим образом применяются для конкретного случая управления рекурсией.

VI.3.4.1. Частный случай

Частным случаем является стек **двоичных** значений; таков, например, случай указателя выполнения, когда в программе имеется только два рекурсивных вызова, как в нашем примере с Ханойской башней. Вместо стека целых можно довольствоваться стеком ЛОГИЧЕСКИХ значений; легко заметить также, что такой стек можно достаточно удобно представить (при условии что его размер остается всегда достаточно малым) целым P, которое инициализируется единицей при создании стека:

- стек пуст тогда и только тогда, когда $P = 1$;
- засылка в стек значения 0 представляется операцией $P \leftarrow 2 \times P$; засылка в стек значения 1 – с помощью $P \leftarrow 2 \times P + 1$;
- выборка из стека некоторого элемента состоит в выполнении $P \leftarrow P/2$ (целое деление); взятый из стека элемент является остатком этого деления (0 или 1).

Условие применимости этого способа состоит, очевидно, в том, что максимальный при выполнении размер m стека определяется неравенством

$2^m \leq M$

где M – наибольшее целое, представимое в машине, а m –максимальный уровень вложенности рекурсивных вызовов. Если целые представлены по меньшей мере 16 битами, например, то этот способ можно использовать для Ханоя при $n \leq 16$ (это является границей, гораздо более широкой, чем все, что можно разумно вообразить в данной конкретной задаче).

Благодаря этому свойству стеков логических значений задачу с Ханойской башней можно переписать. Заметим, кстати, что можно обойтись без параметров X, Y и Z, если пронумеровать кольца 1, 2 и 3; это позволит записать:

$y \leftarrow 6 - x - y$ для поменять (y, z)

и

x ← 6 – x – y для поменять (x, z)

(параметризовав задачу для получения рекурсивного алгоритма, мы убираем из нее параметры, чтобы сделать программу!). Получившаяся программа показана ниже.

Разработка этой программы (которая может быть дополнена другими упрощениями) интересна тем, что она типична. Советуем читателю самому переделать ее, дополнив следующими упражнениями:

– объяснить, почему при возврате из рекурсивного вызова (операторы, следующие за оператором с меткой *1000*) оператор *N = N + 1* выполняется только в случае, когда *ELMPIL* равен 1 (оператор с меткой *1100* и следующие);

ФОРТРАН

```
      SUBROUTINE HANOI (M)
      INTEGER M
C     ЗАДАЧА ХАНОЙСКОЙ БАШНИ С M КОЛЬЦАМИ
C     ПРОНУМЕРОВАННЫЕ СТЕКИ 1,2, 5
      INTEGER N, X, Y
      INTEGER PILE, ELMPIL
C     --- ПЕРЕДАЧА ЗНАЧЕНИЕМ ---
      N = M
C
C
      PILE = 1
      X = 1
      Y=2
100   IF (N. EQ. 0) GOTO 1000
C     ПЕРВЫЙ РЕКУРСИВНЫЙ ВЫЗОВ
      PILE = 2 *PILE
      Y=6– X – Y
      N = N – 1
      GOTO 100
1000  IF (PILE. EQ. 1) GOTO 5000
C     ВОЗВРАТ РЕКУРСИВНОГО ВЫЗОВА
      ELMPIL = MOD (PILE, 2)
      PILE = PILE/2
      IF (ELMPIL. EQ. 1) GOTO 1100
C     ЗДЕСЬ ELMPIL= 0:
C     РЕЗУЛЬТАТ ПЕРВОГО ВЫЗОВА
      Y=6– X – Y
      CALL DEPLAC (X, Y)
C     ВТОРОЙ РЕКУРСИВНЫЙ ВЫЗОВ
      PILE = 2 * PILE + 1
      X=6–X–Y
      GOTO 100
C     ДАЛЕЕ ELMPIL=1:
C     РЕЗУЛЬТАТ ВТОРОГО РЕКУРСИВНОГО ВЫЗОВА
1100  X = 6– X –Y
      N = N + 1
      GOTO 1000
C     КОНЕЦ
5000  RETURN
END
```

- преобразовать в программу на ФОРТРАНе модифицированную программу, из которой устранены ненужные рекурсивные вызовы:

```
если n = 1 то
    | переместить (x, y)
иначе
    | Ханой (n – 1, x, z, y);
    | переместить (x, y);
    | Ханой (n – 1, z, y, x)
```

- убедиться в применимости алгоритма, интерпретируя его в терминах обхода двоичного дерева игры (Рис. VI.11).

VI.3.4.2. Внутренние стеки подпрограмм

В общем случае, разумеется, невозможно обойтись без «хитростей» в представлении стека (битов) целыми числами. Можно использовать классическое представление стеков с помощью массивов и соответствующих указателей («сплошное представление»), которые могут быть объявлены как внутренние к подпрограмме. Конкретно:

а) возьмем целую переменную, например урстек, которая будет указывать *уровень вложенности* рекурсивных вызовов;

б) заменим на массив каждую локальную переменную в подпрограмме и каждьгё скалярный аргумент, который должен быть сохранен, а каждый массив–на массив, обладающий дополнительной размерностью. Новая размерность представляет глубину рекурсии. Ее границами являются 1 и m, где m – верхняя граница глубины рекурсии, достигаемая при выполнении (тем самым предполагается, что такая граница известна);

в) в заголовок подпрограммы введем оператор *инициализации,* присваивающий, переменной урстек значение 1;

г) засылка, предшествующая переходу в начало программы и представляющая собой рекурсивный вызов $P(y_1, y_2, ..., y_n)$ где $y_1 ..., y_n$ – новые значения параметров для рекурсивного вызова, выражается с помощью:

- операторов

хстек [урстек] ← x

включаемых для каждого аргумента, указателя выполнения или локальной переменной x, где хстек есть массив, соответствующий x (см. пункт а) выше), и операторов

астек [i_1, i_2, ..., i_n, урстек] ← a [i_1, i_2, ..., i_n]

включаемых для каждого элемента каждого массива a, которому соответствует массив астек;

- затем операторов

$x_i \leftarrow y_i$

(для каждого аргумента x_i; $1 \le i \le n$); • наконец, увеличения счетчика

урстек ← урстек + 1

д) *проверка «пустой стек?»,* определяющая, надо ли выполнить настоящий «возврат» или же простую выборку из стека, записывается просто:

урстек = 1

е) *выборка из стека,* следующая за неокончательным рекурсивным возвратом, записывается (предыдущая проверка обязательг но привела к результату ложь) как

урстек <– урстек – 1;

x ← хстек [урстек]; {для каждого параметра, указателя выполнения или локальной переменной}
a[i,..., i_n] ← астек [i_1,..., i_n, урстек]
{для каждого элемента массива}

Эти операторы соответствуют обычному кодированию операций засылки и выборки при сплошном представлении стека. Это представление было рассмотрено в разд. V.4.4, где сказано, что следует добавить две проверки, диагностирующие ошибку, если урстек при выборке становится отрицательным или же если урстек при засылке становится больше m. В данном случае задача ставится несколько по особому:

- первый тип ошибок (переполнение «снизу») не должен возникать, если перевод рекурсии выполнен верно. На этой проверке можно, таким образом, сэкономить при выполнении на свой страх и риск при условии, разумеется, что вы уверены в этом факте;
- ошибки второго типа (переполнение «сверху») не являются, как указывалось в гл. V, логическими ошибками; это просто «технические» ошибки, возникающие вследствие плохого выбора m, максимально предусматриваемого размера. Обычно при каждой засылке в стек невозможно обойтись без соответствующей проверки; в некоторых случаях, однако, верхняя граница глубины рекурсии известна априори; так, в задаче с Ханойской башней эта граница есть n; стек длиной 20 охватывает любую трудность для практически встречающихся случаев. Читателю предлагается изучить все рекурсивные программы этой главы и найти для каждой из них такую границу.

VI.3.4.3. Глобальный стек

Только что рассмотренное использование внутреннего стека в подпрограмме оправдано единственно в случае, когда в языке, не допускающем рекурсии, пытаются представить только одну рекурсивную подпрограмму (или небольшое число рекурсивных подпрограмм) и когда априори известна максимальная глубина рекурсивных вызовов. В ином случае следует отделять стек от программ, используя для рекурсии единый для всех рекурсивных подпрограмм общий стек. Если вызовы подпрограммы соответствуют классической иерархической модели (IV.7), т.е. если любое поколение подпрограммы заканчивается строго раньше создавшего его поколения подпрограммы (той же самой или другой), то структура стека действительно оказывается приспособленной к представлению структуры вызовов (рекурсивных или нерекурсивных), рассматриваемых теперь уже не для единственной подпрограммы, а для совокупности подпрограмм.

В том, что касается использования памяти, этот способ оказывается наиболее эффективным: как показано в V.4.4, невозможно простое сосуществование нескольких стеков без сохранения максимально необходимого размера для *каждого* из них при сплошном представлении. Наоборот, использование единого стека позволяет сохранять только максимальный размер *множества* стеков, который, как правило, гораздо меньше суммарного размера. Этот вопрос был детально рассмотрен в гл. IV и V. Другое преимущество глобального стека состоит в отделении данных от программ, что всегда полезно.

Данный способ обычно используется при реализации рекурсии с помощью транслятора. Например, в АЛГОЛе W память, доступная программе, при выполнении разделена обычно на две части (Рис. VI.12):

а) часть, предназначенную для постоянных элементов, которая может быть сохранена с этапа трансляции: сама программа и данные фиксированного (или «статического») размера (ср. IV.6);

б) часть, управляемую как стек, предназначенную для переменных элементов.

Заметим, что в АЛГОЛе эти постоянные элементы включают, кроме локальных данных рекурсивной подпрограммы, все элементы, управляемые «динамическим распределением» (IV.6.2), в частности массивы, размер которых может меняться при выполнении (ср. объекты *AUTOMATIC* в ПЛ/1: IV.6.5 и V.3). Как мы видели в V.2.2, управление этими объектами также отвечает структуре стека. Базовая схема управления памятью в языках типа АЛГОЛ 60, таким образом, очень проста: имеется фиксированная область для «статических» объектов и единственный стек, предназначенный для всех объектов, управляемых динамическим распределением. Напомним, что в АЛГОЛе 60 эта категория включает не только массивы с фиксированными при трансляции границами, но фактически все объекты программы (переменные, аргументы, массивы), создающиеся только при каждом выполнении блока, к которому они принадлежат (отсюда следует невозможность иметь в АЛГОЛе 60 остаточные[1] объекты). В ПЛ/1 (и фактически в некоторых алголо–подобных системах) различают фиксированные объекты (*STATIC* – в ПЛ/1) и «стековые».

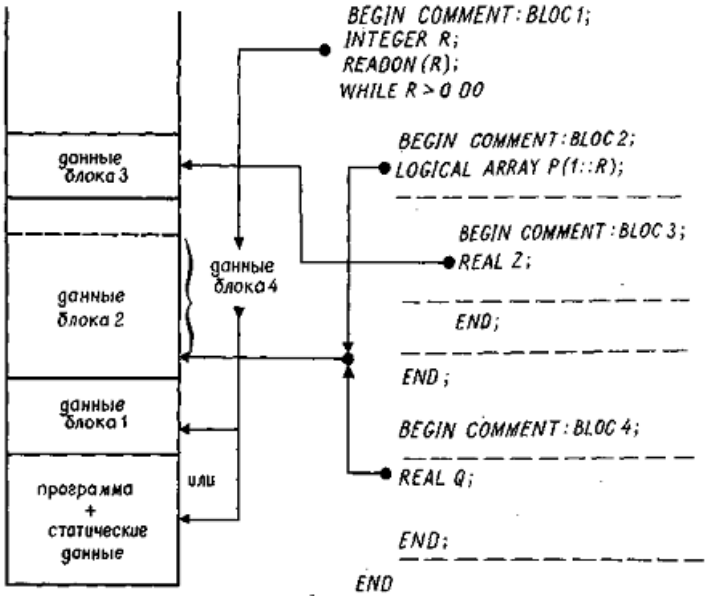


Рис. VI.12 Управление памятью в сплошном стеке

[1] Такие, которые существуют после завершения программы.– *Прим. перев.*

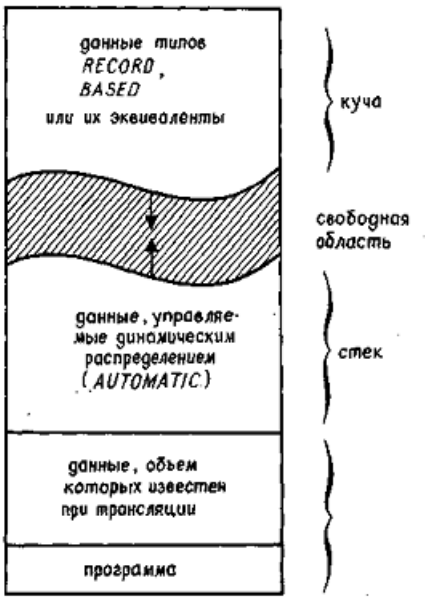


Рис. VI.13 Управление памятью с фиксированной областью, стеком и кучей.

В языках, требующих использования «кучи» (V.1.4) для структур данных, способ расширения которых непредвидим (данные типа *RECORD* в АЛГОЛе W, объекты, снабжаемые указателями в ПЛ/1 и их эквиваленты в ПАСКАЛе, АЛГОЛе 68, СИМУЛе 67 и всех других языках этого класса), «кучу» рассматривают обычно как стек, по меньшей мере в первом приближении: действительно в V.4.4 мы видели, что представление двух «перевернутых» стеков исключительно просто (Рис. VI.13). Когда два стека соединяются – это означает, что память исчерпана: поскольку «куча» не является все–таки «настоящим» стеком, – можно вызвать программу «сборщик мусора», чтобы найти свободные места (V.1.4), а если их нет, то переупаковать «кучу», чтобы снова управлять ею как стеком до очередного переполнения памяти.

По–видимому, здесь небесполезно напомнить, что объект, объявляемый как

REFERENCE (ENGER) X в АЛГОЛе W

или

DECLARE X POINTER в ПЛ/1

т.е. указатель, принадлежит блоку программы и будет, следовательно, заводиться в стеке при каждом новом поколении этого блока; наоборот, объект, который он указывает в текущий момент (запись типа *ENREG* в АЛГОЛе W или произвольное данное в ПЛ/1), является частью кучи; он может в свою очередь содержать указатель, который отмечает некоторый другой элемент, принадлежащий куче, но не стеку (за исключением языка ПЛ/1, в котором допускаются некоторые смешения сомнительного свойства: упражнение VI.3).

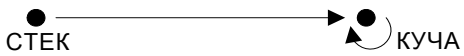


Рис. VI.14 Возможные отношения между указателями и данными, принадлежащими стеку и куче (на АЛГОЛе W, АЛГОЛе 68, ПАСКАЛе и т.д.).

VI.3.4.4. Цепное представление стека

До сих пор мы использовали стеки в сплошном представлении. Другим возможным представлением является *цепное* представление (V.4.4). Оно особенно естественно в системе, где широкое применение рекурсии сочетается с использованием *списков* как базовых структур данных (V.8). Таким является программирование на языке ЛИСП. Концептуально управление памятью в этом случае просто: все структуры данных представлены цепочками. В любой момент работают главным образом два списка:

- список «активных» объектов, включающий фактически несколько подсписков: программы (структура которых на ЛИСПе является списковой) и данные, доступные этим программам;

- список свободных мест, сцепленных друг с другом и управляемых как стек: получение нового места для программ или для данных (например, при рекурсивном вызове) соответствует выборке из стека, а высвобождение места (выполняемое, например, сборщиком мусора) соответствует засылке в стек свободных мест. Об этом способе было упомянуто в разд. V.6.4 в связи с линейными списками. Рис. VI.15 может рассматриваться как изображение структуры данных, которая используется в такой системе, или как представление абстрактной «машины ЛИСП», которую система «моделирует» на реальной машине.

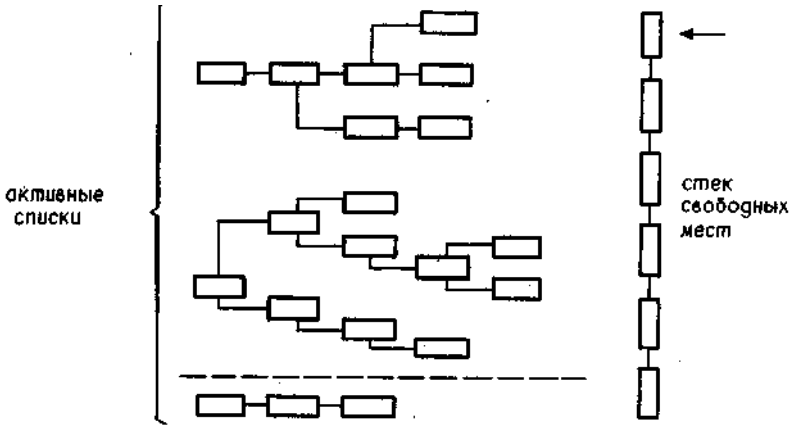


Рис. VI.15 Управление «списковой памятью».

VI.3.5. Упрощения. Исключение рекурсии

Способ реализации рекурсии, рассмотренный в предыдущих разделах, применим ко всем случаям. Однако не всегда его следует применять слепо и огульно: во многих практических случаях возможны упрощения, которые позволяют освободиться от бремени, налагаемого использованием рекурсии. В противоположность базовым методам, рассмотренным выше, эти упрощения требуют анализа «высокого уровня» и относятся к компетенции программиста или очень совершенного транслятора. Мы исследуем три случая:

- как осуществить рекурсию, не разрушая структуры программы;
- как сократить применение стеков, используя свойства обратных функций;
- как исключить частично или полностью рекурсивные вызовы, сохранив при этом, разумеется, эквивалентность программы.

Относительно последнего пункта заметим, что часто используемое выражение «исключение рекурсии» несколько двусмысленно: с помощью использования стеков рекурсию всегда можно исключить; это то, что выполняет транслятор в конечном итоге. Мы рассматриваем только такие случаи исключения или по меньшей мере ограничения рекурсивного характера всей совокупности подпрограмм, когда найден способ использования, позволяющий сэкономить один или несколько стеков.

> Результаты этого раздела основываются больше на интуиции, чем на строгом доказательстве. Теория «эквивалентности программ» фактически требует аксиоматизации, которая выходит за рамки этой работы (см., например, [Хоар 71c], [Вейон 76], [Вийемен 73], [Манна 74], [Берстол 76], [Арсак 77]).

VI.3.5.1. Попытка «структурирования»

В свете гл. III может показаться неудобным представление рекурсии с помощью несколько анархических операторов ветвления, о чем мы упоминали в III.6, так как они затуманивают чтение программы. Такая проблема не возникает, если рекурсия реализуется транслятором (поскольку во всяком случае ветвления в конечном итоге являются неизбежным представлением в машинном языке); сложнее, если программист сам стремится исключить рекурсию в программе, написанной на языке высокого уровня.

Рассмотрим рекурсивную подпрограмму P. Предположим, что P имеет только один параметр x; в действительности нижеприведенные результаты легко распространяются на произвольное число параметров: достаточно рассмотреть x как набор, представляющий собой совокупность параметров P, возможно, ее результат и т.д.

Особенно часто встречается случай, когда P имеет общий вид:

```
программа P (аргументы x: ...)
    если c(x) то
        A0(x)
    иначе
        A1(x);
        P(f1(x));
        A2(x);
        P(f2(x));
        ...
        Am(x);
        P(Ux));
        Am+1(x);
```

где $c(x)$ есть некоторое условие; $A_0(x)$, $A_2(x)$,..., $A_{m+1}(x)$ – действия, возможно, сложные, но не приводящие к рекурсивному вызову P, а $f_1(x)$, $f_2(x)$, ..., $f_m(x)$ – выражения, зависящие от x.

В этом частном случае реализация рекурсии вписывается в очень простую схему. Целая переменная, называемая номвоз («номер возврата»), используется для обозначения номера следующего выполняемого рекурсивного вызова, т.е. сохраняемого указателя выполнения. Используем два оператора:

сохранить (x, номвоз)

и восстановить (x, номвоз)

которые сохраняют и восстанавливают состояние программы, представленной аргументом x и «номером возврата»; в простейшем случае операторы «сохранение» и «восстановление» будут означать соответственно засылку в стек и выборку из стека, представление которых известно из предыдущего раздела.

Прежде чем ввести оператор, позволяющий проверить, является ли рассматриваемое поколение последним активным, т.е. при представлении с помощью стека определить, пуст ли стек, воспользуемся «уловкой», которая заключается в том, что в начале выполняется оператор

сохранить (x, 1)

запоминающий специальный «номер возврата» 1, который не будет использоваться в каком–либо другом месте. Операция восстановить станет тогда возможной без предварительной проверки: рассматриваемое поколение будет заключительным в том и только в том случае, когда номвоз = 1.

При этих условиях P можно записать в нерекурсивном виде:

```
сохранить (x, 1);
повторять
    | пока ~ c(x) повторять {спуск слева}
    |     | A1(x);
    |     | сохранить (x, 2); x ← f1(x);
    | {здесь проверяется c(x)}
    | A0(x);
    | восстановить (x, номвоз);
    | пока номвоз = m + 1 повторять {подъем}
    |     | Am+1(x)
    |     | восстановить (x, номвоз);
    | если номвоз ≠ 1 то {прохождение вершины}
    |     | {2 ≤ номвоз ≤ m}
    |     | Aномвоз(x);
    |     | сохранить (x, номвоз + 1); x ← fномвоз (x)
до номвоз = 1
```

Последний оператор внешнего цикла выполняемый, если номвоз ≠ 1:

$A_{номвоз}(x)$;
сохранить (x, номвоз + 1); $x \leftarrow f_{номвоз}(x)$

должен пониматься как сокращение от

```
если номвоз = 2 то
    | A2(x);
    | сохранить (x, 3); x ← f2(x)
иначе если номвоз = 3 то
    | A3(x);
    | сохранить (x, 4); x ← f3(x)
иначе если ...
иначе если номвоз = m то
    | Am(x);
    | сохранить (x, m + 1); x ← fm(x)
```

представляемое структурой **выбрать** ... или **вариант**... (III.3.2.2).

Вышеприведенный алгоритм может быть интерпретирован и объяснен как обход дерева.

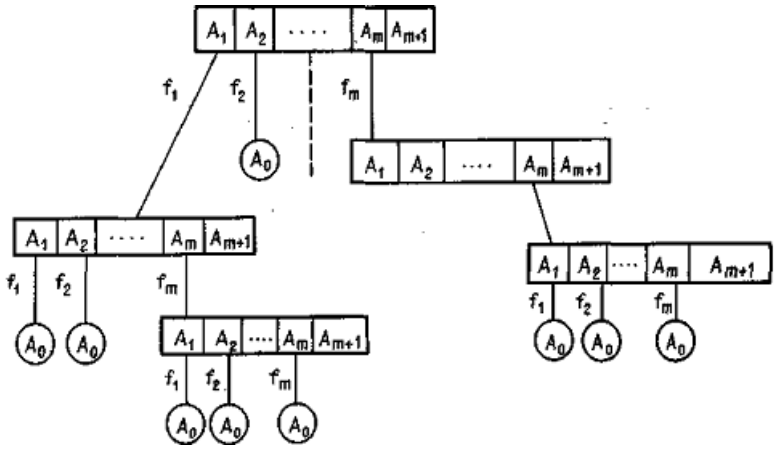


Рис. VI.16

В самом деле, P есть обход некоторого упорядоченного дерева (Рис. VI.16), в котором внутренние вершины имеют вид

A_1	A_2	.....	A_m	A_{m+1}

а листья, соответствующие c(x) – **истина,** имеют вид

Обход дерева, образованного одним листом, заключается в выполнении $A_0(x)$; обойти дерево, корень которого есть вершина первого вида, значит выполнить $A_1(x)$ и обойти (рекурсивно) его первое поддерево; затем выполнить $A_2(x)$ и обойти (рекурсивно) его второе поддерево; ...; выполнить $A_m(x)$ и обойти (рекурсивно) его m–е поддерево; наконец, выполнить $A_{m+1}(x)$.

При переходе от вершины к ее i–му поддереву ($1 \le i \le m$) x преобразуется в $f_i(x)$ и соответствующей ветви присваивается метка f_i.

Приведенный выше нерекурсивный алгоритм становится, тогда легко понимаемым: оператор

пока ~ c(x) **повторять**
| $A_1(x)$;
| сохранить (x, 2); $x \leftarrow f_1(x)$

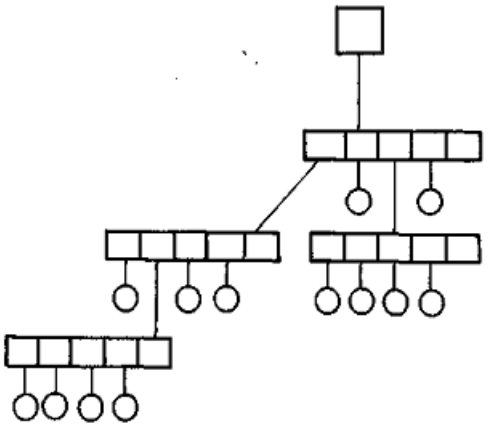
заключается в том, что спускаются «все время слева» до листа, соответствующего c(x) истина, сохраняя последовательные значения x. Со всяким листом связано действие $A_0(x)$. Затем «поднимаются» по дереву путем:

восстановить (x, номвоз)
пока номвоз = m + 1 **повторять**
| $A_{m+1}(x)$;
| восстановить (x, номвоз)

что приводит к последней, частично, но не полностью исследованной вершине; при переходе выполняется действие A_{m+1} в полностью исследованных вершинах, т.е. в вершинах, все поддеревья которых были обойдены. На достигнутой в конечном счете вершине выполняют $A_{номвоз}$, где номвоз – номер первого, еще неисследованного поддерева этой вершины, и входят в поддерево, сохранив номвоз + 1 для следующего прохождения вершины.

Такое соответствие показывает особенно тесные отношения, существующие между деревьями и рекурсией.

Заметим, что начальный оператор
сохранить (x, 1)
который позволяет нам узаконить все операции «восстановить» в программе, может быть понят как присоединение конкретного искусственного «отца» к корню дерева:



Примечание: можно заметить, что цикл нерекурсивной программы имеет вид:

```
повторять
    спуститься;
    A0;
    подняться;
    если номвоз # 1 то
        пройти
до номвоз = 1
```

Повторение проверки «номвоз = 1?» при каждом выполнении цикла может оказаться неудобным. Для получения эквивалентной программы (которую можно будет интерпретировать в терминах обхода дерева) такое повторение можно убрать, открыв цикл

```
спуститься; A0; подняться;
пока номвоз ≠ 1 повторять
    пройти;
    спуститься; A0; подняться
```

Эта программа заменяет двойную проверку при выполнении на удвоение части текста программы. Можно убедиться, что в действительности ни одна из двух конструкций не удовлетворительна; мы имеем здесь случай, когда цикл выполняется «n + 1/2 раз» (ср. упражнение III.4) или, скорее, 1/2 + n раз. Фактически для получения удовлетворительного решения нужно выйти за строгие рамки управляющих структур, которыми мы ограничивались, вводя, например, операторы выхода из цикла. Задача, которую мы оставляем читателю для самостоятельного решения, состоит в выборе между небольшим изобразительным неудобством, введенным здесь с помощью достаточно жестких ограничений, и дополнительными сложностями, касающимися читаемости и т.д., связанными с более богатыми структурами.

Приведенная выше нерекурсивная версия допускает обобщение на случай программ, в которых число рекурсивных вызовов на поколение не фиксированно, а переменно. Например, если i–й «блок» имел вид

```
пока ri(x) повторять
    Ai(x);
    P(fi(x)),
```

а не просто

```
Ai(x);
P(fi(x))
```

то соответствующее «пройти» в рекурсивном варианте, которое раньше было:

```
Ai(x);
сохранять (x, i + 1); x ← fi(x)
```

запишется теперь как

```
если ri(x) то
    Ai(x);
    сохранять (x, i + 1); x ← fj(x)
иначе
    Ai+1(x);
    сохранять (x, i + 2); x ← fi+1(x)
```

Этот вид мы будем использовать в VI.5.4. Пока мы можем дать простую нерекурсивную форму для различных рассмотренных выше рекурсивных программ. Наш старый друг Ханойская башня

```
если n > 0 то
    Ханой (n – 1, x, z, y);
    переместить (x, y);
    Ханой (n – 1, z, y, x)
```

записывается с помощью прямого использования вышеприведенного преобразования с учетом того, что параметр n не нуждается в явном сохранении:

```
переменная номвоз: ЦЕЛОЕ;
сохранить (x, y, z, 1);
повторять
    пока n > 0 повторять
        сохранить (x, y, z, 2);
        поменять (y, z); n ← n – 1;
    восстановить (x, y, z, номвоз); n ← n + 1;
    пока номвоз = 3 повторять
        восстановить (x, y, z, номвоз); n ← n + 1;
    если номвоз ≠ 1 то
        переместить (x, y);
        сохранить (x, y, z, 3);
        поменять (x, z); n ← n – 1
до номвоз = 1
```

Если использовать идеи разд. VI.3.4.1 и продолжать сохранять значения 2 и 3 номвоз с помощью двоичного счетчика (бит 1 для значения 2, бит 0 для значения 3), то получаем

(A)

```
программа Ханой (аргументы n : ЦЕЛОЕ, x, y, z: КОЛЬЦА)
переменная счетчик : ЦЕЛОЕ;
счетчик ← 1;
повторять
    пока n > 0 повторять
        счетчик ← 2 × счетчик + 1;
        поменять (y, z); n ← n – 1;
    пока счетчик четный повторять
        поменять (x, z); n ← n + 1;
        счетчик ← счетчик/2;
    поменять (y, z); n ← n + 1;
    счетчик ← ⌊ счетчик / 2 ⌋
    если счетчик ≠ 0 то
        переместить (x, y);
        счетчик ← 2 × счетчик;
        поменять (x, z); n ← n – 1;
до счетчик = 0
```

В данном случае условие четности счетчика в (A) *до* деления на 2 эквивалентно условию номвоз = 3 *после* восстанавливаемого вызова; из этого следует изменение порядка операторов в центральной части цикла.

Более эффективный вариант мог бы быть получен, исходя из программы, не имеющей бесполезных рекурсивных вызовов; оставим этот вариант читателю в качестве упражнения.

VI.3.5.2. Построение обратных функций

Предыдущее рассуждение показывает, насколько интересным свойством является существование обратных функций преобразования аргументов; оно позволяет избежать многочисленных засылок в стек и выборок из стека и экономить место (разумеется, такая «оптимизация» не всегда необходима; в частности, в ней нет нужды, когда вычисление обратной функции занимает много времени и время более ценно, чем пространство. Проблемы соотношения «пространства–времени» будут рассматриваться в VII.1.2.4, а задача эффективности и оптимизации программ – в VIII.4.7).

Если преобразование аргумента при рекурсивном вызове не кажется

необратимым, то можно иногда, используя свойства структур данных, «построить» механизм вычисления Обратной функции. Проиллюстрируем этот способ на примере.

Мы уже обращали внимание на большое сходство между программой Ханой и обходом ЛКП двоичного дерева, который, как мы видели, дает метод сортировки:

```
программа сортировка (аргумент а: ДВОДЕР)
    если а ≠ ПУСТО то
        сортировка (слева (а));
        пройти корень (а);
        сортировка (справа (а))
```

где пройти означает некоторую операцию. Эта программа может, в частности обслуживать программу *«сборщик мусора»* (V.1.4): речь тогда идет о просмотре структуры данных, представляющих собой объекты, доступные программе, а операция **пройти** означает в этом случае «пометить» определенные элементы как использованные и невосстанавливаемые; последовательный просмотр области памяти, доступной программе, позволяет после сортировки восстановить все непомеченные объекты.

Поскольку схема рекурсии в этой программе полностью совпадает со схемой рекурсии в программе Ханой, мы сразу получаем нерекурсивный вариант:

```
переменная намвоз: ЦЕЛОЕ;
сохранить (а, 1);
повторять
    пока а ≠ ПУСТО повторять {спуститься слева}
        сохранить (а, 2);
        а ← слева (а);
    восстановить (а, номвоз);
    пока номвоз = 3 повторять {подняться справа}
        восстановить (а, номвоз);
    если номвоз ≠ 1 то {пройти и выйти справа}
        пройти корень (а);
        сохранить (а, 3);
        а ← справа (а)
до номвоз = 1
```

Здесь, однако, функции преобразования аргумента (а ← слева (а) и а ←с права(а)) оказываются сложнее простых перестановок, и кажется, что использовать обратные функции, а не стек двоичных деревьев (т.е., конкретно, указателей), невозможно.

Тем не менее оказывается возможным и в этом случае *построить* надлежащую обратную функцию. Предполагая для простоты, что представление цепное, можно использовать поле «сын левый» или «сын правый» для обозначения родительской вершины при спуске соответственно слева или справа. Операция сохранить а записывается тогда в предположении, что переменная отец обозначает отца а:

```
{при спуске слева}
сын ← слева (а);
слева (а) ← отец;
отец ← а;
а ← сын
```

{здесь отец означает снова отца а: это свойство инвариантно} {при спуске справа: то же самое с заменой «левое» на «правое»}

Таким же образом, если подниматься слева, то операция восстановить а запишется как

```
сын ← а;
а ← отец; отец ← слева (а);
слева (а) ← сын
{здесь отец снова означает отца а: это свойство инвариантно}
```

и симметричным образом, если подниматься справа.

Стек величин номвоз будет обозначать, как и прежде, последовательность перемещений, позволяющих достичь вершины, начиная с корня (0 = правая; 1 = левая).

Нерекурсивный вариант записывается тогда в виде

```
программа сортировка (аргумент а: ДВОДЕР)
    переменные счетчик: ЦЕЛОЕ {битовый стек},
                сын, отец: ДВОДЕР;
    отец ← пусто; счетчик ← 1;
    повторять
        пока а ≠ ПУСТО повторять {спуститься слева}
            счетчик ← 2 × счетчик + 1 {засылка 1 в стек};
            сын ← слева (а); слева (а) ← отец;
            отец ← а; а ← сын;
        пока счетчик четный повторять {подняться справа}
            сын ← а;
            а ← отец; отец ← справа (а);
            справа (а) ← сын; счетчик ← счетчик/2;
        {подняться на один уровень слева}
        если отец ≠ ПУСТО то
            сын ← а; а ← отец;
            отец ← слева (а); слева (а) ← сын;
        счетчик ← счетчик/2;
        если счетчик ≠ 0 то {спуститься на один уровень справа}
            счетчик ← 2 × счетчик {засылка 0};
            сын ← справа (а); справа (а) ← отец;
            отец ← а; а ← сын
    до счетчик = О
```

Как и в VI.3.5.1, на порядок операторов здесь несколько повлиял тот факт, что четность счетчика до деления на 2 соответствует иомвоз = 3 после восстановления.

Заметим все же, что здесь, **вероятно,** представление последовательности битов с помощью целого неадекватно: когда двоичное дерево «полно» (V.7.6), как то, которое соответствует Ханойской башне, трудно обеспечить, чтобы размер целого (от 16 до 60 битов) был удовлетворителен. Мы придерживались обозначения с помощью целого, поскольку оно удобно, но практически следовало бы говорить о битовом стеке: массив значений ЛОГ или битовая строка (*BITS* в АЛГОЛе W, *BIT* в ПЛ/1).

Способ «возвращения сына», предложенный Шорром и Вейтом, является классическим; он изложен в [Кнут 69, 2.3.5] применительно к алгоритму сборщика мусора.

Тем не менее в том виде, в каком он обычно излагается (в частности, у Кнута), рекурсия управляется не битовым стеком, а дополнительным битом, который присваивается каждой вершине и значение которого указывает, какое поле вершины–правое или левое–означает отца, а не сына. Кажется предпочтительным не связывать себя с модификацией структуры данных для каждой выполненной операции (в данном случае – обхрд ЛКП)

Отметим, что полученный результат имеет особое значение в случае сборщика мусора: как мы упоминали в V.3, сборщик мусора вызывается в момент, когда не хватает места в памяти: рекурсивное решение, использующее стеки непредусмотренного размера, следовательно, запрещается. В этом случае, исходя из разумных гипотез относительно максимальной глубины структур данных, применяется,

наоборот, только битовый стек, требующий памяти ограниченного и относительно малого размера.

Как и раньше, предоставим читателю возможность самостоятельно разработать нерекурсивную версию программы (более эффективную, но несколько более длинную), из которой исключены ненужные рекурсивные вызовы:

```
если а ≠ ПУСТО то
| если слева (а) ≠ ПУСТО то
|     | сортировка (слева (а));
| пройти корень (а);
| если справа (а) ≠ ПУСТО то
|     | сортировка (справа (а))
```

VI.3.5.3. Исключение рекурсии

В некоторых случаях при рекурсивном вызове сохранять ничего не требуется.

В частности, когда последний выполняемый программой оператор представляет собой рекурсивный вызов (как для Ханоя и сортировки), можно в принципе обойтись без того, чтобы после соответствующего ветвления ставился бы оператор запоминания. В самом деле, в рассматриваемом поколении локальные данные никогда больше не понадобятся.

В качестве примера читатель сможет проверить, что в программе корень (численный пример из VI.2.2) только один их всех возможных рекурсивных вызовов (предпоследний) действительно требует сохранения локальных объектов в подпрограмме, остальные же вызовы всегда выполняются последними в этой программе.

Это правило очевидно для частного случая, рассмотренного нами выше, а именно конструкция

```
| если c(x) то A0(x)
| иначе
|     | A1(x); P(f1(x));
|     | A2(x); P(f2(x));
|     | .....
|     | Am(x); P(fm(x));
|     | Am+1(x);
```

которая реализуется в виде

```
| сохранить (x, 1);
| повторять
|     | пока ~ c(x) повторять
|     |     | A1(x); сохранить (x, 2); x ← f1(x);
|     | A0(x); восстановить (x, номвоз);
(1)   | пока номвоз = m + 1 повторять
|     |     | Am+1(x); восстановить (x, номвоз);
|     | если номвоз ≠ 1 то
(2)   |     | Aномвоз(x); сохранить (x, номвоз +1);
|     |     | x ← fномвоз(x)
| до номвоз = 1
```

В том случае, когда A_{m+1} есть пустое действие, нет необходимости хранить «номер–возврата», равный $m + 1$, и соответствующие аргументы x, поскольку в цикле (1) не нужны эти ранее запоминавшиеся элементы.

В этом случае, следовательно, получают более простую рекурсивную форму, никогда не сохраняя номвоз = m + 1; достаточно убрать цикл (1) и заменить условный оператор (2) на

```
если номвоз ≠ 1 то
    | Aномвоз(x)
    | если номвоз < m то
    |     | сохранить (x, номвоз + 1);
    | x ← fномвоз(x)
```

Можно было бы попытаться применить это упрощение к последним версиям, полученным для Ханоя и обхода двоичного дерева. Читатель, однако, убедится, что оно несовместимо с приемом, использующим обратные функции; в самом деле, выполняемые преобразования $x \leftarrow f_m(x)$ (в одном случае переставить (x, z), в другом x ← справа (x)) изменяют аргументы так, что нельзя продолжать алгоритм, если не выполнить соответствующие обратные преобразования (так, переставить (y, z) является своим обратным только для одинаковых значений y и z). Напротив, только что описанное упрощение применимо к этим двум алгоритмам, если запоминание осуществляется с помощью стека, а не обратными преобразованиями.

Важным случаем, в котором возможно упрощение, является случай подпрограммы с единственным рекурсивным вызовом, после которого не выполняется никакой оператор:

```
программа P (аргументы x:...)
    | если c(x) то
(1) |     | A0(x) {нерекурсивное действие}
    | иначе
    |     | A1(x); {нерекурсивное действие}
    |     | P(f(x))
```

где f(x)– есть выражение, зависящее от x. Эта программа–частный случай программы, рассмотренной выше в VI.3.5.1. Таким образом, благодаря только что описанному упрощению получается нерекурсивная форма:

```
пока ~ c(x) повторять
    | A1 (x); x ← f(x);
A0(x)
```

Здесь сохранять ничего не требуется. Нам уже приходилось сталкиваться с тем, что некоторые рекурсивные алгоритмы могут записываться просто с помощью циклов; так, включение в линейный список:

```
тип ЛИНЕЙНЫЙСПИСОКт = ПУСТО | НЕПУСТОЙЛИНСПИСОКт;
тип НЕПУСТОЙЛИНСПИСОКт = (голова: T; хвост: ЛИНЕЙНЫЙСПИСОКт);
программа включение (аргумент x: T;
                      модифицируемый параметр ℓℓ; ЛИНЕЙНЫЙСПИСОКг)
    | выбрать
    |     | ℓℓ есть ПУСТО;
    |     | ℓℓ ← НЕПУСТОЙЛИНСПИСОКт (x, ПУСТО),
    |     | ℓℓ есть НЕПУСТОЙЛИНСПИСОКт:
    |     |     включение (x, хвост (ℓℓ))
```

может быть записано в нерекурсивном виде:

```
переменная y: ЛИНЕЙНЫЙСПИСОКт;
    выбрать
        | ℓℓ есть ПУСТО:
        |     | ℓℓ ← НЕПУСТОЙЛИНСПИСОКт (x, ПУСТО),
        | ℓℓ есть НЕПУСТОЙЛИНСПИСОКт:
        |     | y ← x
        |     | пока хвост (y) есть НЕПУСТОЙЛИНСПИСОКт
```

```
повторять
    | y ← хвост (y)
хвост (y) ← НЕПУСТОЙЛИНСПИСОК_т (x, ПУСТО)
```

Это дает немного более сложную программу по сравнению со схемой (1), приведенной выше, потому что здесь участвуют структуры данных, передаваемые как модифицируемые параметры (а не просто аргументы).

Часто, опираясь на предыдущий результат, утверждают, что программа, отвечающая приведенной выше схеме (1), не является «по–настоящему» рекурсивной, поскольку рекурсия может здесь выражаться простым циклом **пока.** Не обсуждая, что такое «настоящая» рекурсия, отметим, что рекурсивная формулировка может быть удобнее в первом приближении, особенно когда она придает программе наиболее ясный и простой вид, как, например, в случае с включением в линейный список. В гл. VIII мы вернемся к способу программирования, использующему *последовательные преобразования* для улучшения программ, исходя из простых и корректных, но необязательно очень эффективных начальных вариантов. Преобразование только что рассмотренного типа, исключающее рекурсию в простой программе, можно выполнить с помощью умеренно «интеллектуального» транслятора.

Важное следствие приведенного выше результата состоит в том, что цикл пока можно в свою очередь использовать как частный случай рекурсивной программы:

```
пока c повторять
    | A
```

можно понимать как вызов

```
циклпока (c, A)
```

при определении подпрограммы:

```
программа циклпока (аргументы: c : УСЛОВИЕ, A: ПРОГРАММА)
    если c то
        | A
        | циклпока (c, A)
```

Именно это объясняет то, что к этим двум структурам программы применяются одни я те же понятия (завершимость, инварианты, управляющая величина).

Существует много других случаев рекурсии, которая может быть исключена полностью или частично; к сожалению, нет еще теории, которая позволила бы свести в систему отдельные конкретные рекомендации. Наиболее интересные из этих правил приведены в недавно вышедшей статье Берсталла и Дарлингтона [Дарлингтон 76] (см. также [Берсталл 77], [Грифитс 75], [Арсак 77] и [Вейон 76]). Здесь мы ограничимся демонстрацией одного случая, часто встречающегося на практике. Это программа вида:

```
программа p: T (аргумент x : T')
    если c(x) то
        | p ← f(x)
    иначе
        | p ← u(g(x), p(h(x)))
```

где T и T' – произвольные типы; f(x), g(x) и h(x) – выражения, зависящие от x; u – некоторая *ассоциативная* операция. Тогда p можно вычислить с помощью эквивалентной нерекурсивной программы:

```
программа p1; T (аргумент x : T')
    если c(x) то
        | p1 ← f(x)
    иначе
        | p1 ← g(x); x ← h(x);
        | пока ~ c(x) повторять
        |     p1 ← u(p1, g(x)); x ← h(x)
        | P1 ← u(p1, f(x))
```

Отметим, что, как и в общем случае, нерекурсивная программа гораздо сложнее.

Простым примером использования этого преобразования является вычисление функции факториала для натуральных аргументов :

$$\text{факториал}\ (n) = \begin{cases} 1 & \text{если } n = 0 \\ n \times \textit{факториал}(n-1) & \text{если } n > 0 \end{cases}$$

что сразу же дает ($c(x) = (n = 0)$; $f(x) = 1$; $u(x,y) = x \times y$; $g(x) = x$; $h(x) = x - 1$);

```
если n = 0 то
    | факториал ← 1
иначе
    | факториал ← n; n ← n – 1;
    | пока n ≠ 0 повторять
    |       факториал ← факториал × n;
    |       n ← n – 1
    | факториал ← факториал × 1 {оператор, очевидно, излишний}
```

Оставим читателю использование этого правила применительно к программе, обеспечивающей объединение двух списков (V.8).

Докажем это правило, рассматривая последнюю строку рекурсивной программы как формальное равенство. С помощью последовательных подстановок получаем:

$$\begin{aligned} p(x &= u(g(x), p(h(x))) \\ &= u(g(x), u(g(h(x)), p(h(h(x))))) \\ &= u(g(x), u(g(h(x)), u(g(h^2(x)), p(h^3(x))))) \\ &= \ldots \\ &= u(g(x), u(gh(x), u(gh^2(x), \ldots, u(h^{m-1}(x), fh^m(x))\ldots))) \end{aligned}$$

где $gh(x)$ означает $g(h(x))$, $gh^2(x)$ означает $g(h(h(x)))$ и т.д., а m – наименьшее натуральное целое, такое, что $c(h^m(x))$ есть **истина** (если такого m нет, то $p(x)$ не определено). В силу ассоциативности u имеем $p(x) = u(u(\ldots(u(g(x), gh(x)), gh^2(x)), \ldots, gh^{m-1}(x))$

откуда следует способ вычисления, который последовательно дает (при $c(x) =$ **истина** – тривиальный случай):

$u(g(x), gh(x))$
$u(u(g(x), gh(x)), gh^2(x))$
$u(u(u(g(x), gh(x)), gh^2(x)), gh^3(x))$

и т.д. Когда $h^m(x)$ удовлетворяет условию c, то f применяется к последнему терму.

Правила, подобные этому и опирающиеся на свойства используемых функций (ассоциативность, коммутативность и т.п.), применяются некоторыми оптимизирующими ЛИСП–системами для автоматического исключения рекурсии в некоторых случаях: системы Интерлисп [Тейтельман 73], REMREC [Рисх 73], система Берсталла и Дарлингтона [Берсталл 76].

VI.4. Восходящее рекурсивное вычисление

Во всех рассмотренных рекурсивных программах предлагавшиеся способы вычисления могли быть представлены как *нисходящие;* для каждого значения аргументов исходили из общего случая, чтобы с помощью последовательных преобразований прийтк к тривиальному случаю. Так, для Ханойской башни выполнялась декомпозиция общей задачи путем последовательного уменьшения n и спуска по соответствующему двоичному дереву, чтобы прийти к тривиальному случаю $n = 0$ (или $n = 1$, т.е. к листьям дерева). Часто представляется интересным выполнить обратную операцию: отправляясь от тривиальных ситуаций, «подняться» к случаю,

требующему решения. Нужно быть, конечно, уверенным в возможности реализовать эту ситуацию. Когда этот способ применим, он позволяет получить более эффективные программы.

Понятие «восходящие вычисления рекурсивных программ» было предложено в статье Берри [Берри 76]. Его строгое изложение требует дальнейшей формализации, и в частности использования теории вычислений Скотта [Теннент 76]; здесь мы ограничимся несколькими интуитивными понятиями и примерами, показывающими достоинства этого способа.

Предположим, что функция f определяется рекурсивной формулой вида

$f(x) = u(f,x)$

где $u(f, x)$ есть выражение, включающее функцию f (поскольку формула рекурсивна), применяемую к аргументам, зависящим от x (а не только к самому x; без этого замечания определение содержало бы логическую ошибку). Вообще говоря, $u(f,x)$ включает условное выражение; например,

факториал (x) = **если** x = 0 то 1
иначе x × *факториал* (x – 1)

или

комбинация (n, m) = **если** m > n **то** 0
иначе если m = 0 **то** 1
иначе *комбинация* (n – 1, m)
+ *комбинация* (n – 1, m – 1)
{коэффициенты треугольника Паскаля}

Рассмотрим график G функции, т.е. множество пар [x, f(x)] (наборов $[x_1, x_2, ..., x_n, f(x_1, ..., x_n)]$, если f имеет n аргументов). Для всякого x, такого, что функция f(x) определена, т.е. такого, что $\alpha = [x, f(x)]$ принадлежит графику, имеем

$[x, f(x)] = [x, u(f,x)]$

Это уравнение можно рассматривать как уравнение на графике G вида

$\alpha = F(\beta, \gamma, ...)$

где α, β, γ, ... – элементы графика, а F – «производящая функция», которая позволяет выявить новые элементы, исходя из известных элементов графика. Именно эта функция F будет использовать восходящий способ: отправляясь от «тривиальных» элементов графика, т.е. пар [x, f(x)], таких, что f(x) вычислима без рекурсивного вызова (например, x = 0 для факториала), G пополняется с помощью последовательности итераций до получения пары (или набора), где первый элемент есть аргумент x_0, для которого определяется значение f.

Функция F выводится из определения f, однако в общем случае переход достаточно сложен. На данных примерах он выполняется непосредственно. Согласно определению функции факториал, таким образом, ясно, что [x, y] принадлежит G тогда и только тогда, когда y = факториал(x), т.е. когда

x = 0 **и** y = 1

или $x \neq 0$ и y = x × факториал (x – 1)

или же

[x, y] = [0, 1]

или [x, y] = [a + 1, (a + 1) × b]

где [a, b] есть элемент графика. Другими словами, функция F преобразует график G в график G', равный G, увеличенному на все пары [a + 1, (a + 1) × b], где [a, b] – элемент G. Это подсказывает восходящий способ вычисления факториал(n), исходя из графика, содержащего только тривиальный элемент [0, 1], и пополняет его на

каждом этапе на $[a + 1, (a + 1) \times b]$, где $[a, b]$ – последний полученный элемент.

Этот итеративный метод, надлежащим образом формализованный, сходен с решением математических уравнений методом «неподвижной точки».

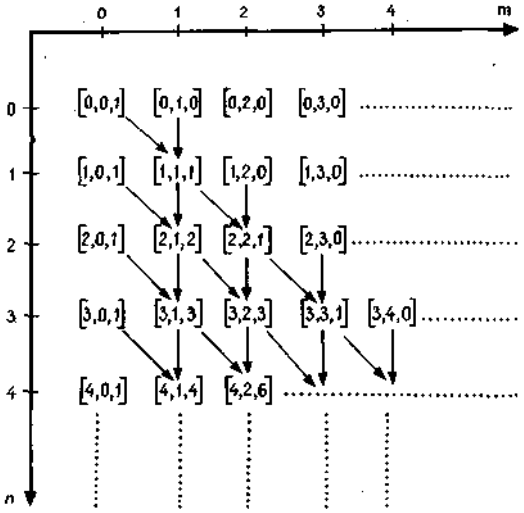
Что касается функции комбинация, из определения сразу же вытекает, что $\alpha = [n, m, c]$ принадлежит графику тогда и только тогда, когда:

(а) $\alpha = [n, m, 0]$ при $m > n$;

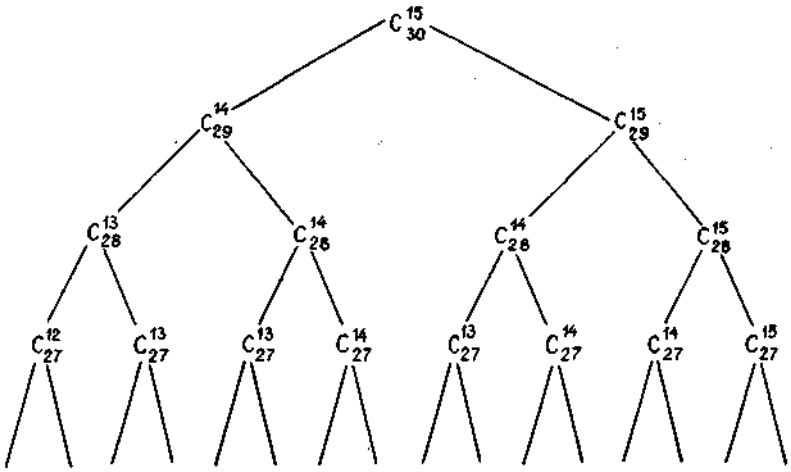
(б) **или** $\alpha = [n, 0, 1]$ (n произвольное);

(в) **или** $\alpha = [n, m, c_1 + c_2]$ при $[n - 1, m, cj, [n - 1, m - 1, c_2]$, принадлежащих графику.

Здесь восходящий метод состоит в том, что, исходя из графика (в принципе бесконечного), содержащего элементы типа (а) и (б), итеративно комбинируют элементы, смежные с m, до получения новых элементов типа (в):



Разумеется, этот метод пока ничего нового не внес в наши два примера: Блэзу Паскалю не нужно было знать восходящего вычисления рекурсивных программ, чтобы изобрести вышеприведенный треугольник. Не в столь классических случаях, однако, восходящее вычисление может позволить значительно повысить эффективность программ. Важно отметить, что в нашем последнем примере классический нисходящий метод не осуществим при n и m, достаточно больших; например, для $n = 30$, и $m = 15$ он приводит к двоичному дереву



где различные элементы будут вычисляться многократно.

Читатель сможет применить эти рассуждения к другой классической задаче–числам Фибоначчи:

$x_0 = 0, x_1 = 1, x_n = x_{n-1} + x_{n-2}$ для $n > 1$

«Восходящий» метод, близкий к только что описанному, позволяет получить изящное решение

задачи Ханойской башни (опять она!). Рассмотрим решение задачи размерности n как последовательность H_n образованную следующими элементами:

- D (для переместить (x, y))
- X (для перестановки x ⇔ z)
- Y (для перестановки y ⇔ z)

В соответствии с самой структурой алгоритма сразу же имеем

H_0 = пусто

H_1 = D

а для $n \geq 1$

$H_n = YH_{n-1},YDXH_{n-1}X$

где последний X обеспечивает сохранение аргументов в их начальном состоянии. Удобнее выразить H_n в зависимости от H_{n-2}:

$H_n = Y(YH_{n-2}YDXH_n{-}_2X)\ YDX(YH_{n-2}YDXH_{n-2}X)\ X = H_{n-2}YDXH_n{-}_2XYDXYH_{n-2}YDXH_{n-2}$

(поскольку YY и XX – пустые операции). Полагая A = YDX, получаем

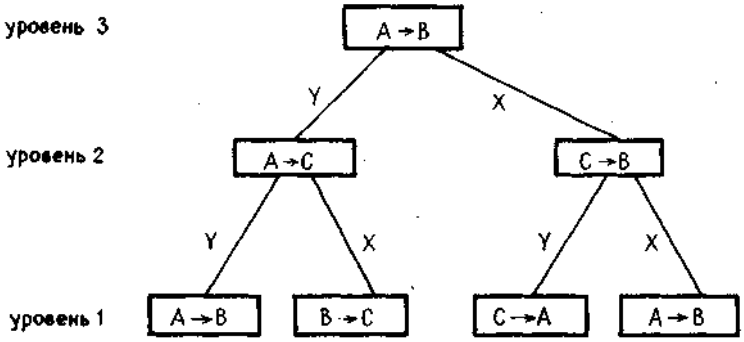
$H_n = H_{n-2}AH_{n-2}XAYH_{n-2}AH_{n-2}$

Это подсказывает способ решения задачи: построить последовательность H_n в виде массива, элементы которого могут равняться A, X, Y или D, исходя из H_0 = пусто или H_1 = D в зависимости от четности или нечетности n и, двигаясь с шагом 2 по вышеприведенной формуле; затем заполненный таким образом массив интерпретировать как последовательность перестановок и перемещений.

Последнее решение Ханойской башни

В этой главе мы слишком часто использовали пример с Ханойской башней, оставляя читателя в неведении о другом существующем решении, по–видимому, самом эффективном из всех и которое можно считать восходящим.

Рассмотрим двоичное дерево, соответствующее игре (без лишних рекурсивных вызовов); в данном случае для n = 3.



Вершины соответствуют элементарным перемещениям и последовательно отмечены. Ветви помечены X или Y в зависимости от того, соответствуют ли они перестановке X или перестановке Y (т.е. x ↔ z или y ↔ z).

Условимся, что листья имеют «уровень 1» и что (рекурсивно) внутренняя вершина имеет уровень на 1 больше уровня своих сыновей. Корень имеет, следовательно, уровень n, его сыновья – уровень n – 1 и т.д.

Если разыскивается последовательность S_n уровней вершин, участвующих в алгоритме обхода ЛПК, то непосредственно из самой рекурсивной структуры алгоритма видно, что S_n удовлетворяет следующим соотношениям:

S_0 = пусто

$S_1 = 1$

$S_n = S_{n-1},nS_{n-1}$, для $n \geq 1$

таким образом, $S_2 = 121$, $S_3 = 1213121$, $S_4 = 121312141213121$ и т.д.

Ясно также, что пройденные вершины поочередно являются листьями уровня 1 и внутренними вершинами уровня, большего на 1.

Если уровень вершины четный, то это внутренняя вершина, а предыдущая, следовательно, –

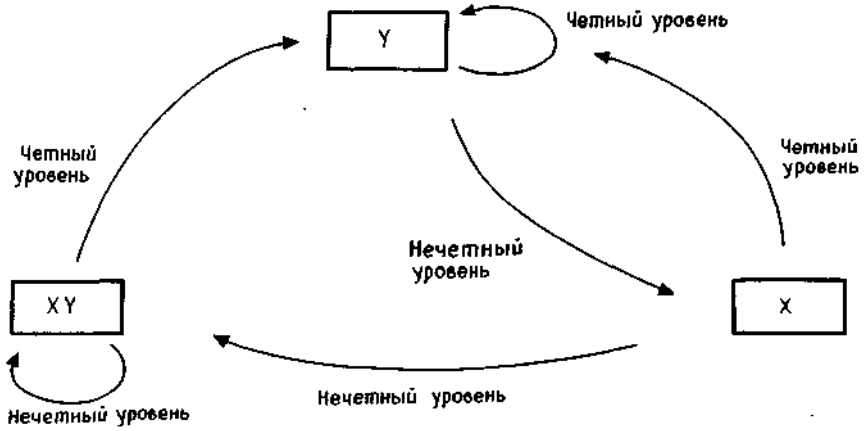
лист. В этом случае:

- либо предыдущая вершина была левым сыном, а новая вершина уровня 2 – его отцом. Тогда выполняемая перестановка есть Y;
- либо предыдущая вершина была правым сыном и для перехода к новой вершине выполняется x перестановок X и единственная перестановка Y. Тогда число уровней восхождения, которое является полным числом перестановок, оказывается нечетным и, следовательно, x – четное: перестановки X аннулируются, и остается выполнить еще только перестановку Y.

Оставим в качестве упражнения доказательство того, что точно так же при нечетном уровне пройденной вершины выполняемая перестановка представляет собой:

- X, если уровень предыдущей вершины был четным;
- X, за которым следует Y (обозначение XY), если уровень предыдущей вершины был нечетным.

Достаточно, таким образом, следовать «диаграмме перехода»:



Таким образом, зная последовательность S_n, непосредственно получают алгоритм. Начальное состояние известно: тривиально, что первое выполняемое перемещение есть A → B (исходная точка–конечная точка), если n нечетно; A → C (исходная точка → промежуточная точка), если n четно.

Для построения последовательности S_n можно заметить в качестве математического курьеза, что она подобна последовательности C_n мест первой справа единицы в двоичном представлении чисел между 1 и $2^n - 1$. Действительно,

$C_1 = 1$

и

$C_n = C_{n-1} n C_{n-1}$

поскольку упорядоченные двоичные числа между 1 и $2^n - 1$ представляют собой числа между 1 и $2^{n-1} - 1$, за которыми следует

$\underbrace{100...0\ (2^{n-1})}_{n-1 \text{ нулей}}$

а далее предыдущие числа, к которым слева добавлена 1. Отсюда наш последний алгоритм, который – с этим можно согласиться – не выводится тривиально из формулировки задачи, приведенной в VI.2.1:

```
программа Ханой (аргумент n : ЦЕЛОЕ)
    {задача Ханой с n кольцами; исходное: A, конечное: B, промежуточное: C}
    переменные x, y, z: КОЛЬЦА,
                четное, предыдущее–четное: ЛОГ
    x ← C; y ← B; z ← A;
    предыдущее–четное ← "n нечетно"; {инициализация}
    для i от 1 до 2^n–1 повторять
        четное ← "первая 1, начиная справа в двоичном представлении i, имеет
                  четный номер";
        если четное то переставить (y, z)
        иначе если предыдущее–четное то переставить (x, z)
        иначе
            переставить (x, z)
            переставить (y, z);
        переместить (x, y);
        предыдущее–четное ← четное
```

VI.5. Применение: алгоритмы последовательных испытаний

VI.5.1. Введение и определения

Алгоритмы последовательных испытаний, или же «возвратные» алгоритмы (*backtrack*), появляются естественным образом в различных областях, таких, как искусственный интеллект или исследование операций. Они соответствуют задачам, в которых поиск решения не подчиняется твердым правилам, а выполняется путем последовательных испытаний; если испытание не приводит к результату, то «возвращаются назад» для других попыток.

В таких алгоритмах каждый этап соответствует одной из трех следующих ситуаций;

а) решение найдено;

б) имеется некоторое число возможных действий, которые, можно надеяться, приблизят решение;

в) невозможно никакое действие этого типа, и решение не найдено. Эту ситуацию называют *тупиковой.*

В случае *а)* алгоритм, очевидно, завершен (если только не разыскиваются все возможные решения, а не одно из них). В случае *б)* алгоритм «выбирает» согласно некоторому критерию одно из возможных действий для выполнения. Наконец, в случае *в)* надо «вернуться назад» к последнему этапу, на котором еще не были исчерпаны все возможные альтернативы, и пробовать снова, если только все возможные альтернативы не были уже испытаны на всех предыдущих этапах. В этом последнем случае следует остановиться.

Мы назовем *конечной* ситуацию, соответствующую *а)* или *в)*, т.е. ситуацию, когда можно сразу же обнаружить успех или неудачу.

Алгоритм последовательных испытаний можно представить себе как обход лабиринта (Рис. VI.17). На каждом перекрестке приходит ся выбирать новый возможный путь; при попадании в тупик или на перекресток, через который проходит выбранный в данный момент путь, надо следовать за нитью Ариадны в обратном направлении до первого перекрестка, от которого отходит еще не испытанная дорога.

Рис. VI.17 Попытка обхода лабиринта (стрелками отмечены начала уже безуспешно испробованных путей).

Метафора лабиринта выявляет два основных противоречия алгоритмов возврата: с одной стороны, необходимо уметь определять порядок выбора различных возможных альтернатив на каждом этапе, чтобы не испытывать один и тот же путь дважды, с другой стороны, сложность алгоритма резко возрастает, если существует, как на Рис. VI.17, возможность *зацикливания,* т.е. возврата к уже встречавшейся ситуации; в последнем случае придется хранить в памяти все этапы испытываемого в настоящий момент пути и сравнивать всякий новый переход с каждым из них. Попытаемся сформулировать алгоритм таким образом, чтобы лабиринт соответствовал, например, «дереву» (Рис. VI.18); другими словами, всякая новая альтернатива приводит к уменьшению некоторой неотрицательной «управляющей величины», которую можно считать «расстоянием» между рассматриваемой и конечной ситуациями. Очевидно, что эта управляющая величина играет ту же роль, что и управляющая величина, соответствующая рекурсивной программе. Мы снова увидим ее в рекурсивной формулировке алгоритмов последовательных испытаний. Однако она не всегда присутствует в этих алгоритмах.

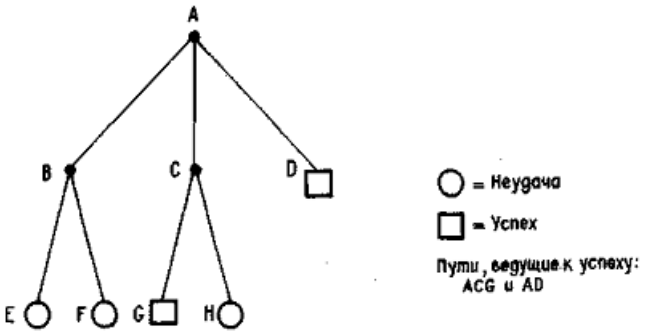


Рис. VI.18

VI.5.2. Алгоритм последовательных испытаний и искусственный интеллект

Алгоритмы последовательного выбора особенно часто встречаются в искусственном интеллекте. В этой области действительно путь, ведущий к решению изучаемой задачи, обычно заранее не известен; надо, следовательно, идти путем проб и ошибок, выбирая возможные альтернативы и заново пересматривая их в тупиковой ситуации.

Часть программ искусственного интеллекта посвящается, таким образом, исследованию деревьев, представляющих собой последовательные альтернативы. Вся трудность состоит в том, что деревья «растут» очень быстро; их густота, известная под названием «комбинаторный взрыв», делает обычно практически невозможным исчерпывающий обход рассматриваемого дерева. Одна из главных задач искусственного интеллекта состоит, таким образом, в применении априорных знаний к изучаемой области для получения *эвристик,* которые позволят исследовать только небольшую часть полных деревьев.

VI.5.2.1. Дендрал, УРЗ

В области искусственного интеллекта одним из успешных проектов, использующих алгоритм последовательных испытаний, является проект ДЕНДРАЛ [Фейгенбаум 71], разработавший программу, применяемую многими химиками. Эта программа на базе данных относительно неизвестной молекулы, полученных с помощью масс–спектрографа (прибор, который дает распределение различных элементов, входящих в состав одной молекулы), дает полную пространственную формулу молекулы.

> ДЕНДРАЛ – весьма специализированная программа, обладающая глубокими «знаниями» в химии по некоторым классам веществ. Эта программа использует алгоритм последовательных испытаний. Она включает «генератор», который предлагает возможные в принципе формулы, и «предсказатель», который исследует, соответствуют ли эти формулы физическим теориям. Генератор связан с «планировщиком», предназначенным для использования узко специализированных химических сведений с целью существенного сокращения числа вариантов, «порожденных» генератором, которые должны проверяться «предсказателем». Наличие «планировщика» совершенно необходимо для некоторых категорий молекул, число априорных возможных конфигураций которых превосходит 10^7. Так, для ди–*н*–децила ($C_{20}H_{42}O$) имеется больше 11 млн. возможных вариантов; «планировщик» позволил проверить только 22 366 вариантов и определить правильную пространственную конфигурацию.

ДЕНДРАЛ повседневно используется многими химиками; применительно к некоторым категориям химических веществ (простые эфиры, спирты, аминокислоты, тиоэфиры, тиолы и т.д.) он позволил за несколько секунд или минут вычислений получить результаты, которые требовали у химиков дней и даже недель исследований.

ДЕНДРАЛ, разработанный совместной группой квалифицированных химиков и специалистов по искусственному интеллекту, является типичным образцом подхода искусственного интеллекта к узкоспециализированным программам, связывающим в единое целое накопившиеся за годы технические сведения. В противоположность ДЕНДРАЛу такая программа, как УРЗ (GPS: *General Problem Solver* –универсальный решатель задач) [Эрнст 69], имеет целью обеспечить универсальность применения к различным задачам. Задача в принципе может быть решена с помощью УРЗ при условии, что она сводится к формализму УРЗ (т.е. в грубом приближении можно определять начальное положение, критерий успешности и преобразования состояния, используемые для последовательных испытаний). УРЗ может, таким образом, интегрировать функции, подобно SAINT (ср. ниже), или решать головоломки, как, например, «семь кенигсбергских мостов» Эйлера, «миссионеры и людоеды» (которые должны пересечь реку)... не говоря уже о задаче, о которой читатель, возможно, слышал в этом разделе, – Ханойская башня!

> Между специалистами «узкого профиля» и «универсалистами» ведется дискуссия по искусственному интеллекту. На современном этапе примечательно, что ДЕНДРАЛ является программой, эффективно использующейся для решения «настоящих» задач, тогда как УРЗ – более общая в принципе программа–решает главным образом школьные задачи.

По всем вопросам, связанным с искусственным интеллектом, мы отсылаем читателя к двум работам: [Слэгл 71] – приятная в чтении, но несколько поверхностная, и [Нильсон 71] – менее общая, но более систематическая.

VI.5.2.2. Деревья и/или

Важная категория алгоритмов последовательных испытаний, используемая в задачах искусственного интеллекта, занимается деревьями особого типа, называемыми «деревьями и/или» в соответствии с двумя типами присутствующих в них вершин.

Предположим, надо решить задачу P. Попытаемся свести ее к более простым задачам. Для этого существуют два способа:

а) найти более простые подзадачи $P_ь$ P_2, ..., $P_т$, такие, чтобы было достаточно решить P_1 **или** P_2 **или** ... **или** P_m для решения P;

б) найти более простые подзадачи P_1, P_2, ..., P_m, такие, чтобы было достаточно решить P_1 **и** P_2 **и** ... **и** P_m для решения P.

P_1, P_2 и т.д. могут ив том, и в другом случае оказаться либо задачами, решение которых следует непосредственно, либо задачами, которые сами сводятся к подзадачам в соответствии со случаями *а)* и *б)*, приведенными выше. Следует, таким образом, построить конкретное дерево (Рис. VI.19), у которого имеются внутренние вершины, двух видов:

- одни, помеченные как

  соответствуют задачам, решение которых требует решения всех задач–сыновей (назовем их «вершинами и»);
- другие, обозначенные как

  представляют собой задачи, решение которых треоует решения по меньшей мере одной из задач последующего поколения (скажем, что это «вершины **или**»).

Листья деревьев помечены белым квадратиком, если они соответствуют задачам, решение которых следует непосредственно, и черным квадратиком, если они соответствуют задачам, не имеющим решения. Видно, что задача на Рис. VI.19 имеет решение (а на самом деле их два: одно «проходит» через BHOPU, а другое–через CIJR).

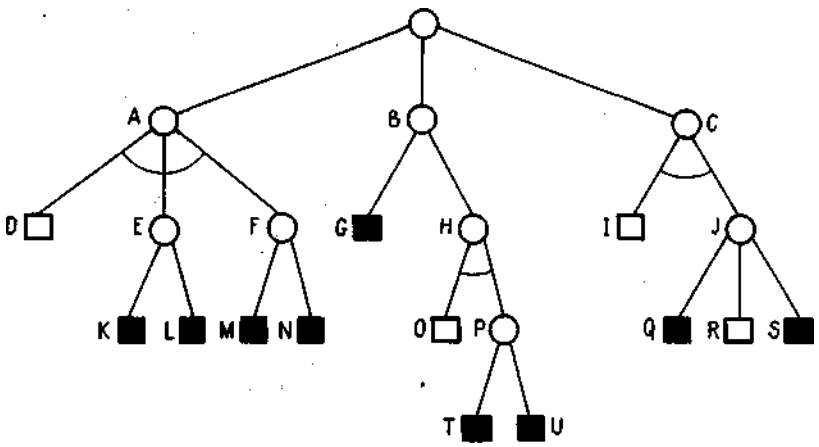


Рис. VI.19 Дерево и/или.

На первый взгляд только вершины **или** в действительности соответствует

поиску с помощью последовательных испытаний. На самом деле ситуация совершенно симметрична:

- если показано, что один из сыновей вершины **или** соответствует решаемой задаче, то и сама эта вершина соответствует решаемой задаче;
- если показано, что один из сыновей вершины **и** соответствует неразрешимой задаче, то и сама эта вершина соответствует неразрешимой задаче.

Эти правила следует понимать рекурсивно: решаемая или неразрешимая задача является либо элементарной, о которой можно непосредственно сказать, что она решаема (или неразрешима), либо задачей, которая в результате рассмотрения ее сыновей оказывается разрешимой (или неразрешима) путем повторяющегося применения тех же правил.

В качестве первого примера дерева и/или рассмотрим программу, играющую в **шахматы**, и попытаемся определить оптимальный ход из любой игровой ситуации. Программа попытается «вообразить» возможное развертывание партии для каждого допустимого хода:

- если нет ни одного допустимого хода (или в случае шаха и мата и т.д.), то партия проиграна;
- если возможны n ходов, то исследуются новые ситуации S_1 ..., S_n, вытекающие из этих ходов.

Для каждой из этих ситуаций программа проверяет возможные ответы воображаемого противника:

- если у него нет возможного хода (или в случае шаха и мата и т.д.), то рассматриваемая ситуация *выигрышная* или ничейная;
- иначе следует рассмотреть различные возможные ходы для противника.

Это приводит к рассмотрению дерева и/или (Рис. VI.20). Вершина **или** соответствует ситуации, в которой играет программа; она *выигрывает,* если *выигрывает* по меньшей мере один из ее сыновей. Вершина **и** – это ситуации, в которых играющим предполагается противник; он *выигрывает* (у программы), только если все его сыновья выигрывают. Листья соответствуют конечным ситуациям, представляющим выигрышные или проигрышные для программы партии.

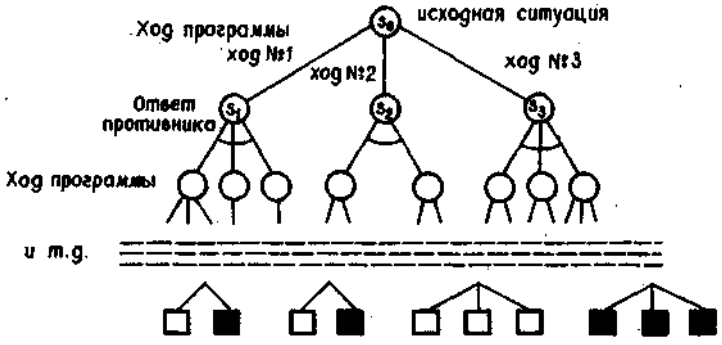


Рис. VI.20

Построенное таким образом дерево является *конечным* для шахмат и всех обычных игр, если ввести по крайней мере некоторое число правил, чтобы избежать бесконечного зацикливания (и которые в шахматах по крайней мере явно присутствуют). Таким образом, теоретически возможно определить, является ли начальная ситуация выигрышной или проигрышной. Другими словами, черные **или** белые (какие точно, мы не можем утверждать) могут *наверняка* выиграть при условии, что применяется некоторая стратегия.

> Поспешим успокоить читателя–страстного поклонника шахмат в том, что число вершин в шахматном дереве составляет 10^{120}; (число 10^{120} соответствует шахматному дереву, из которого уже выброшено большое число вершин, не приводящих ни к какому решению) т.е. что, если бы 10^{75} частиц Вселенной были бы воедино сопряженными вычислительными машинами,

просматривающими за секунду в 1000 раз больше вершин, чем число команд, выполняемых современными ЭВМ, то потребовалось бы еще примерно 10^{30} тысячелетий для определения выигрышной стратегии по дереву игры. Так что шахматисты имеют в перспективе еще несколько безмятежных дней. В VI.5.5 мы увидим, как работают существующие играющие программы, чтобы найти, если не выигрышные, то по меньшей мере «не слишком» плохие ходы. Однако заметим, что деревья и/или позволяют разработать выигрышную стратегию для некоторых простых игр, как, например, класс игр Ним (среди которых наиболее известным примером является игра Мариенбад[1], пришедшая из известного фильма).

VI.5.2.3. SAINT

Вторым примером программы, использующей деревья и/или, является уже давняя программа искусственного интеллекта SAINT [Слэгл 71]. Эта программа вычисляет символические (неопределенные) интегралы; например, исходя из функции $\frac{x^4}{(1-x^2)^{5/2}}$, она вычислит ее интеграл arcsin x + ⅓tg^3 arcsin x – tg arcsin x. Следует подчеркнуть, что речь идет не о численном вычислении конечного интеграла, например $\int_0^{\pi/2} \sin^2 x dx = \frac{\pi}{4}$, а о нахождении символической формулы, дающей неопределенный интеграл, например $\int \sin^2 x dx = -\frac{\sin 2x}{4} + \frac{x}{2} + \text{const}$. (О символьном дифференцировании см. в V.8.6.)

Эта программа основывается на тех же идеях, которые преподносятся студентам при обучении интегральному исчислению. С самого начала известно некоторое число базовых функций, интегрируемых непосредственно,, например $\int \sin x dx = -\cos x$ и т.д. Эти функции соответствуют листьям дерева (конечные ситуации).

С другой стороны, можно попробовать некоторые считающиеся плодотворными способы. Например, это замена переменных; так, если интегрируемая функция имеет член вида $(1 + x^2)^n$, то можно написать x = tg t; если же она имеет одновременно члены sinx и cosx, то можно написать $t = \text{tg}\frac{x}{2}$ и т.д. Эти выбираемые ситуации будут определять вершины или дерева.

Наконец, программе известны правила, позволяющие свести вычисление интеграла к вычислению нескольких, в принципе более простых интегралов (а не одного произвольного из этих интегралов, как по предыдущим правилам). Например, правило

$$\int [f_1(x) + f_2(x) + ... + f_n(x)]dx = \int f_1(x)dx + \int f_2(x)dx + ... + \int f_n(x)dx$$

позволяет свести задачу вычисления интеграла от $f_1 + f_2 + ... + f_n$ к n подзадачам вычисления интегралов от f_1, f_2, ..., f_n. Этот тип правил даст нам вершины и дерева.

Поведение программы SAINT при вычислении интеграла от $\frac{x^4}{(1-x^2)^{5/2}}$ показано на Рис. VI.21.

[1] «Лето в Мариенбаде»–фильм Аллэна Рене, 1961 *г – Прим. перев.*

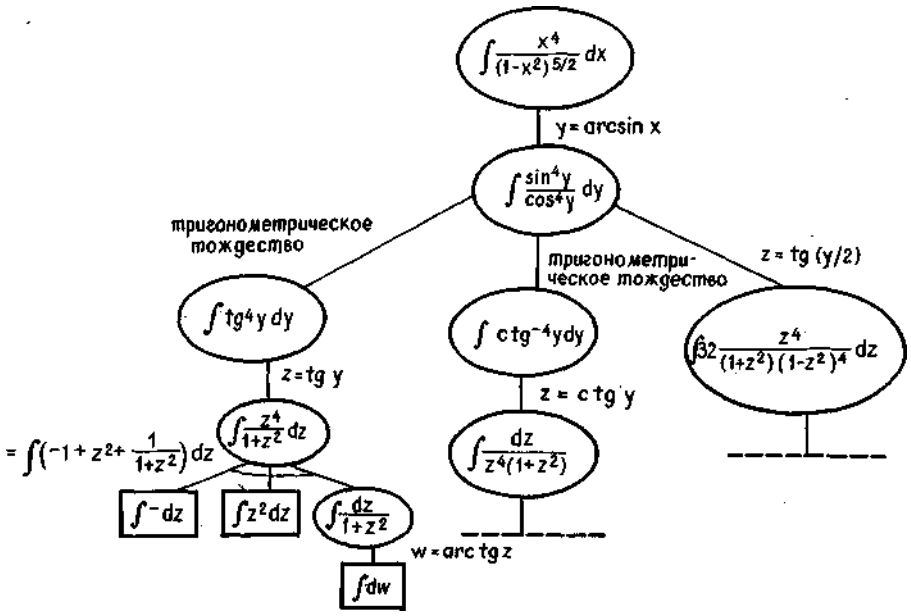


Рис. VI.21 Пример выполнения SAINT.

В этом примере нет «возвратов».

Заметьте, насколько важно для программы упорядочить возможные попытки на каждом уровне. Некоторые попытки приводят к тупику или к бесконечному дереву; кроме того, они могут привести к результату, но слишком длинным путем. Необходимость априорной оценки «стоимости» различных возможных попыток имеет место во всех алгоритмах по искусственному интеллекту; соответствующие методы, которые основываются на предварительных гипотезах об изучаемой области, называются *эвристическими.*

Многочисленные задачи «автоматического доказательства» теорем допускают ту же формализацию. В этих задачах, в самом деле, известны гипотеза H, заключение C, которое следует доказать, и аксиомы, заданные в одной из двух «нормальных» форм:

а) P_1 **или** P_2 **или** ... **или** $P_n \Rightarrow P$

б) Q_1 **и** Q_2 **и** ... **и** $Q_n \Rightarrow Q$

Речь идет о том, чтобы от H перейти к C путем применения аксиом. Аксиомы типа *а)* будут соответствовать вершинам **или**, а аксиомы типа *б)* – вершинам **и**. Одной из наиболее трудных проблем является, очевидно, проблема сопоставления образцов (*pattern matching*), заключающаяся в проверке, действительно ли гипотеза имеет вид, требуемый в посылке одной из аксиом.

VI.5.3. Рекурсивная форма алгоритмов последовательных испытаний

Алгоритмы последовательных испытаний обычно могут быть выражены просто в рекурсивном виде; в этом нет ничего удивительного, поскольку это в основном обходы деревьев[1].

Алгоритм последовательных испытаний включает:

- заданную начальную ситуацию s_0, которая соответствует корню дерева;
- критерий, который соответствует произвольной ситуации s и который мы обозначим успех (s); успех (s) – это логическое значение, которое есть **истина** тогда и только тогда, когда s –ситуация, в которой достигается искомая цель (достижение выигрышной позиции в шашках или шахматах, удовлетворительный изомер в химии, непосредственно интегрируемая

1 Другой, эффективно применяемый формализм – это формализм недетерминированных алгоритмов [Флойд 67].

формула в математике и т.д.);

- функцию списоквыбор, которая сопоставляет любой ситуации s множество списоквыбор (s) действий, возможных в ситуации s. Если списоквыбор (s) оказывается пустым списком, то s – это тупик;
- функцию преемник, которая сопоставляет всякой ситуации s и всякому действию a, принадлежащему данному списоквыбор (s), новую ситуацию s' = преемник (s, a), полученную путем применения действия a в ситуации s.

Дадим имя путь последовательности действий a_1, a_2,..., a_n. Цель программы состоит, очевидно, в нахождении пути, ведущего от начальной ситуации s_0 к ситуации s, такой, при которой успех(s) был бы истиной. Обозначим ПУСТО путь, не содержащий никакого действия; поскольку c – путь, а a – действие, обозначим a || c путь, начинающийся с действия a и продолжаемый путем c.

При таких обозначениях общая формула алгоритма последовательных испытаний (апи), имеющего два результата – логическое значение, которое мы назовем возможно, указывающее, существует ли решение, и путь путьрешения, дающий в случае успеха путь, которым надо следовать, – записывается в виде

```
программа апи {алгоритм последовательных испытаний}
| (аргумент s: СИТУАЦИЯ {начальная ситуация};
| результаты  возможно: ЛОГ,
|             путьрешения: ПУТЬ)
| переменная c : ПУТЬ;
| путьрешения ← ПУСТО;
| если успех (s) то
|   | возможно ← истина
| иначе
|   | возможно ← ложь; {порождение преемников s и их испытание}
|   | для a из списоквыбор (x) пока ~ возможно повторять
|   |   | апи (преемник (s, a), возможно, c)
|   | если возможно то путьрешения ← a || c
```

Эта программа будет вызываться с помощью

```
| переменные p :ЛОГ, c :ПУТЬ;
| апи (s0, p, c)
```

где s_0 – начальная ситуация; решение существует тогда и только тогда, когда P – **истина** после этого вызова, и оно тогда получается с помощью c.

Разумеется, возможны многочисленные вариации на эту основную тему. Один из обычных практических приемов – это принятие мер предосторожности, позволяющих ограничить глубину исследуемого дерева; мы видели, что это почти всегда необходимо на практике. Кроме случая успех (s) и случая, в котором s – тупик (задаваемый с помощью списоквыбор(s) = ПУСТО), рассмотрение может заканчиваться, когда s будет оценено как конечная ситуация по некоторому критерию, называемому конечное (s). Тогда программа апи начинается с

```
если успех (s) то
  | возможно ← истина
иначе если конечное (s) то
  | возможно ← ложь
иначе
  | {исследование преемников s, как раньше}
```

Критерий «конечность» может просто означать, что дерево превосходит некоторую глубину n. В этом случае программа включает дополнительный параметр, например i, который равен 0 (или 1 в зависимости от принятого допущения) при

начальном вызове и i + 1 при каждом рекурсивном вызове; конечное (s) записывается просто i = n.

VI.5.4. Обработка простого примера

Следующий пример является классической задачей, которая интересна тем, что она демонстрирует один из простейших случаев алгоритма последовательных испытаний, в котором имеется фиксированное число возможных выборов, всегда одних и тех же в каждой вершине дерева.

Задача состоит в следующем: найти текст длиной в 100 литер, образованный исключительно литерами "A", "B" и "C" и не имеющий двух смежных и идентичных подтекстов. Например, если бы длина была 6, а не 100, то решение имело бы вид

"ABCBAB"

а не

"ABBCAB" (который имеет две последовательные "B")

"CBABAC" (который имеет две последовательные "BA")

или "ABCABC" (который имеет две последовательные "ABC").

Условимся называть «правильным» текст произвольной длины, не содержащий смежных идентичных подтекстов. Задача состоит в построении из букв "A", "B", "C" текста, отвечающего двойному условию:

- его длина – 100;
- он правилен.

Кажется естественным изучить все тексты, отвечающие одному из двух условий (которое будет приниматься программой как «инвариантное»), до нахождения среди них одного, удовлетворяющего также и второму условию. Это дает две возможные схемы программ:

```
t ← «текст длиной 100»;
пока ~ правильно (t) повторять
    | {инвариант: t имеет длину 100}
    | t ← следующий (t)  {текст, который следует за t в полном перечислении
                          текстов длиной 100}
```

и

```
t ← «правильный текст»
пока длина (t) ≠ 100 повторять
    | {инвариант: t правильно}
    | t ← следующий (t)  {текст, который следует за t в полном перечислении
                          правильных текстов}
```

Инициализация проста в обоих случаях: "AAA...A" есть текст длиной 100 и пустой текст есть правильный текст. На первый взгляд кажется, что проще перечислить все тексты длиной 100, чем все правильные тексты (тем более что нам неизвестно, конечна ли их последовательность): достаточно, действительно, взять в качестве примера алфавитный порядок. С другой стороны, существует 3^{100} текстов длиной 100, т.е. около 10^{47}, кроме того, оказывается, что нелегко использовать для проверки правильности (n – 1)–го текста информацию, накопленную при проверке предыдущего текста: все, по–видимому, следует начинать сначала. Первый подход поэтому кажется неосуществимым.

Наоборот, при заданном правильном тексте t есть простое средство увеличить его длину путем добавления к нему одной из литер–"A", "B", "C". Следует проверить, остается ли правильным полученный текст, но поскольку t правильный, достаточно проверить подтексты, содержащие добавленную литеру, что значительно упрощает

проверку на «правильность». Возможно, что ни одно расширение текста с помощью "A", "B" или "C" не ведет к правильному тексту, но в этом случае можно поставить под сомнение предыдущие расширения текста, что наталкивает на применение алгоритма последовательных испытаний.

Заметим мимоходом, что, поскольку последовательность правильных текстов может быть бесконечной («может быть» означает просто, что никакое очевидное свойство не позволяет априори показать, что эта последовательность конечна), следует

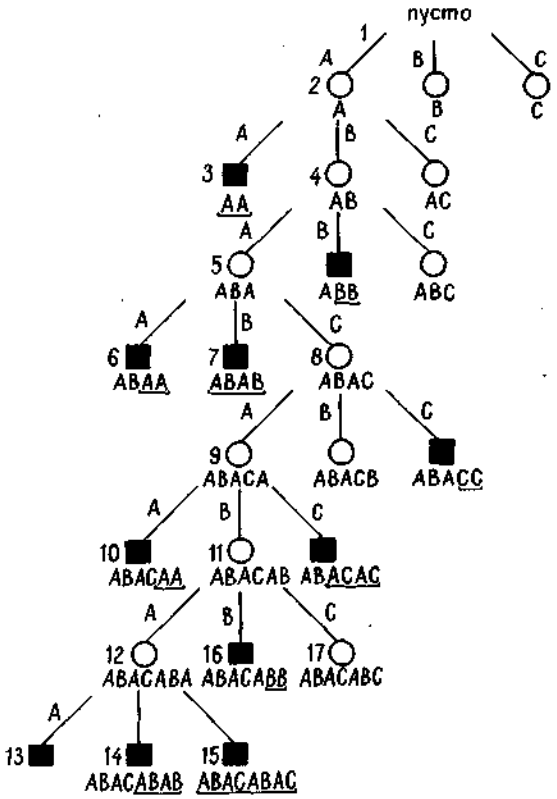


Рис. VI.22 Начало просмотра (■ = неправильная, вершина).

(чтобы быть уверенным в нахождении решения длиной 100, если оно существует, или чтобы уметь доказывать путем исчерпывающего перечисления, что оно не существует) принять алгоритм, гарантирующий для произвольного n, что по истечении конечного времени рассмотрены все подходящие тексты длиной, не превосходящей n.

Все это приводит к алгоритму последовательных испытаний, который на каждом этапе стремится «расширить» текст с помощью "A", "B" или "C" и который ставит под сомнение предыдущий выбор, если невозможно никакое расширение, дающее правильный текст. На Рис. VI.22 показан пример, иллюстрирующий начало поиска. Вершины пронумерованы в том порядке, в котором они рассматриваются. Заметим, что существует возврат назад между вершинами 15 и 16; обнаруживается, что вершина 12, соответствующая правильному тексту, не имеет правильного преемника. Следовательно, надо подняться до вершины 11, где не все еще правильные преемники были исследованы.

Назовем правильным–расширением (x, t) программу, которая выдает значение **истина**, если получается правильный текст при добавлении x (равного "A", "B" или "C") к тексту t, считающемуся правильным, и **ложь** – в противном случае. Запись этой программы очевидна, и мы оставляем ее читателю.

Если t – текст, обозначим его длину через |t|; для x, равных "A", "B" или "C", обозначим через t||x текст, получаемый прибавлением x в конце t (аналогично мы могли бы построить решение, удлиняя тексты с начала). Программа, которая, исходя из

правильного текста t, пытается построить правильный текст длиной 100, начинающийся с t, записывается так:

```
программа возможно: ЛОГ (аргумент t: СТРОКА)
| {первый вариант программы}
| {гипотеза: t правильно (инвариант вызова)}
| если |t| = 100 то {решение найдено}
| | возможно ← истина
| иначе
| | {построение возможно тогда и только тогда, когда расширение с
| | помощью "A", "B" или "C" ведет к решению}
| | возможно ←
| | | (правильное–расширение (t, "A") и
| | | возможно (t || "A"))
| | | или (правильное–расширение (t, "B") и
| | | возможно (t || "B"))
| | | или (правильное–расширение (t, "C") и
| | | возможно (t || "C"))
```

Эта программа, вызванная с помощью возможно(ПУСТО) даст результат **истина**, если существует правильный текст длиной 100, и **ложь** в противном случае. С точки зрения логики эта программа всегда правильна; конкретный результат ее выполнения зависит от практической реализации условий **и** и **или**:

- если это условие подчиняется соглашениям АЛГОЛа W, т.е. если
  - при вычислении a **и** b b не вычисляется при a = **ложь** (результат **ложь**),
  - при вычислении a **или** b b не вычисляется, если a = **истина** (результат **истина**),

  то программа строит только часть дерева, строго необходимую для нахождения первого решения в алфавитном порядке;
- если это не так, т.е. если все вычисляется, то выполнение программы приводит к полному построению дерева до глубины 100 с рассмотрением многочисленных ненужных априори вершин.

Так или иначе, вышеприведенная программа не пригодна в том виде, в каком она дана: она ограничивается определением существования решения, не давая самого решения, если оно есть! Если нужно, чтобы программа определяла не логическое значение, а текст решения, принимая, что результат пуст, когда решение отсутствует, то получают следующую программу, инициируемую значением ПУСТО в качестве фактического параметра:

```
программа текст100 : СТРОКА (аргумент t: СТРОКА)
| {второй вариант}
| {t предполагается правильным (инвариант вызова)}
| если |t| = 100 то
| | текст100 ←t
| иначе
| | текст100 ← ПУСТО;
| | для x из ["A", "B", "C"]
| | пока текст100 = ПУСТО повторять
| | | если правильное–решение (t, x) то
| | | | текст100 ← текст100 (t || x)
| | | | {поскольку t|| x по построению является правильным
| | | | текстом, инвариант вызова не нарушается}
```

Переход от этого алгоритма к нерекурсивному алгоритму прост. Сначала

заметим, что нет необходимости в использовании стека текстов: поскольку фактически алгоритм всегда только дополняет текст буквами A, B и C, то ранее рассмотренные и неотвергнутые тексты всегда остаются начальными подтекстами текста t («приставки» t). Кроме того, выполненный ранее рекурсивный вызов известен благодаря простому данному – последней букве текста t; выборка из стека заключается просто в изъятии последней буквы. Если считать, что функция следующий такова, что следующий ("A") = "B", следующий ("B") = "C", а следующий ("C") = "D", то получаем:

```
программа текст100 : СТРОКА {нет необходимого аргумента}
    {нерекурсивный вариант}
    переменная x : ЛИТЕРА;
    текст100 ← ПУСТО; x ← "A";
    повторять {инвариант: здесь текст 100 является правильным текстом}
        пока x ≠ "D" и ~ правильное–расширение (текст100, x)
        повторять
            x ← следующий (x);
        если x ≠ "D" то {удлинить текст}
            текст100 ← текст100 || x; x ←"A"
        иначе {вернуться к вершине, где возможно новое испытание}
            повторять
                x ← последняя буква текста100;
                убрать из текста100 его последнюю букву
            до x ≠ "C" или | текст100 | = 0;
            x ← следующий (x)
    до | текст100 | = 0 или | текст100 | = 100
```

Примечание: в условии последнего до предусмотрен случай, когда алгоритм завершается неудачно, давая тогда на выходе пустой текст. Правомерность такой проверки завершения вытекает из симметричной роли литер "A", "B" и "C": если на некотором этапе необходимо убрать из текста100 все его литеры, то это означает, что ни один допустимый текст длиной 100 не начинается с буквы "A", и в этом случае бесполезно брать за начало "B" или "C".

VI.5.5. Играющая программа

В заключение этой главы мы приведем черновой набросок программы, более претенциозной, нежели программа, «играющая» в игру двух лиц, например шашки или шахматы. Эту программу можно было бы использовать по–разному:

- для решения «задачи» типа *«белые играют и выигрывают в пять ходов»,* начиная с данной ситуации s_0 (взятой, например, из учебника);
- для игры против другой программы или против самой себя, исходя из обычной начальной ситуации;
- для игры против противника, которым может быть человек, находящийся перед диалоговым терминалом.

Во всех случаях исходят из начальной ситуации, обозначенной s_0, которая может быть (но необязательно) обычной ситуацией начала партии. Задача программы–найти наилучший возможный ход, исходя из этой ситуации; она будет, следовательно, иметь вид.

```
программа лучшийход : ХОД
        (аргументы  s0 : СИТУАЦИЯ,
                    j ; ИГРОК)
    {найти наилучший возможный ход для игрока j, исходя из ситуации s0}
```

СИТУАЦИЯ – это описание состояния игры (клетки заняты различными фигурами и т.д.). Тип ИГРОК имеет два элемента, называемые *«программа»* и *«противник»;* практически они могли бы быть представлены данными типа ЛОГ.

Задача программы состоит в построении воображаемого дерева различных возможных ходов, возможных ответов противника на каждый из них и т.д. Как мы видели в VI.5.2, практически не ставится вопрос о полном исследовании дерева, следует только соблюдать некоторый критерий, обозначенный:

конечное (s)

позволяющий остановить построение дерева в вершинах, соответствующих ситуациям s, которые рассматриваются как «конечные».

В простейшем случае критерий конечное(s) представляет собой глуб(s) = n, где глуб(s) есть глубина вершины, относящейся к s, а n – константа, определяемая в зависимости от ограничений времени и пространства.

Конечная вершина только изредка будет «выигрывающей» или «проигрывающей». Для того чтобы классифицировать различные возможные ходы, следует каждой конечной вершине присвоить положительное или отрицательное значение; это значение будет тем больше, чем более благоприятна для программы рассматриваемая ситуация, и тем меньше, чем ситуация более благоприятна для противника. Если принять симметричные критерии оценки для одного и другого, то значения, близкие к нулю, будут соответствовать, следовательно, «близким» партиям.

Значение, присваиваемое конечной позиции s, обозначается с помощью

оценка (s)

где оценка есть *оценочная функция,* в которой участвуют параметры, характеризующие рассматриваемую игру; при игре в шашки, например, это преимущество в шашках, преимущество в дамках или некоторые более технические параметры, такие, как контроль наиболее важных линий шашечной доски и т.д. Некоторые из лучших существующих играющих программ [Сэмюэл 72] являются *обучающимися* программами: оценочная функция, вычисляемая с помощью коэффициентов, которые присвоены каждому из этих параметров, постоянно улучшается в зависимости от результатов каждой сыгранной партии.

Поскольку значения присвоены конечным вершинам, процедура выбора хода на каждом уровне выполняется сразу же (Рис. VI.23): программа выбирает ходы, которые дают ситуацию максимального значения; в вариантах, соответствующих ситуациям, в которых «ее очередь» играть, программа обеспечивает максимум значений, со ответствующих сыновьям. Противник, наоборот, стремится обеспечить в каждой из «своих» вершин минимум из значений сыновей. Эта процедура называется **минимакс**. Один из игроков *максимизирует* ее значение, другой – *минимизирует.* Начиная с конечных вершин, последовательно обеспечивается минимум или максимум на высшем уровне, так что процесс заканчивается присваиванием значения корню дерева; ход, гарантирующий это значение, считается «наилучшим», и он выбирается программой. Так, игра на Рис. VI.23 имеет в качестве «значения» – 3, и программа выберет ход № 3 или № 4 (в данном случае – ничья), чтобы прийти к этой ситуации.

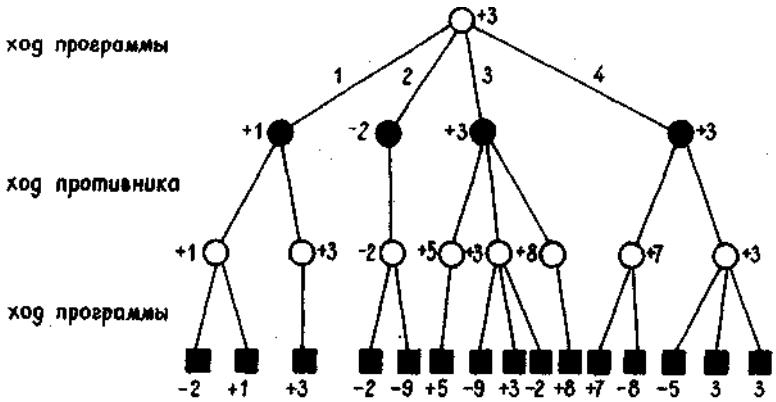


Рис. VI.23 Минимакс.

Для написания соответствующей программы нам необходима функция

списокходов (s, j)

которая дает список возможных ходов для игрока j, исходя из ситуации s, и функция

результат (s, c)

которая является результатом хода c, выполненного, исходя из ситуации s (предполагая, что c принадлежит списокходов(s, j)). Функции списокходов и результат представляют собой правила рассматриваемой игры; их программирование не должно вызвать никаких особых трудностей. Во избежание многочисленных повторов длинных логических выражений используем функцию ЛОГ:

лучший (v_1, v_2, j)

которая имеет значение **истина** тогда и только тогда, когда значение v_1 ситуации оказывается лучшим для игрока j, чем значение v_2 другой ситуации (т.е. $v_1 > v_2$ или $v_2 > v_1$.в соответствии с тем, представляет ли j программу или противника), и таким же образом функцию

максимум (j)

дающую наилучшее возможное значение ситуации для игрока j.

Наконец, условимся, что если j – игрок, то противник (j) обозначает другого игрока.

Программа поиска лучшего возможного хода может тогда быть записана (в первом варианте) как

(a)

```
программа лучший–ход : ХОД (аргументы s0 : СИТУАЦИЯ, первый : ИГРОК)
    переменные v, v1 : ЦЕЛЫЕ;
    v ← максимум (первый); лучший–ход ← ПУСТО;
    для c из списокходов (s0, первый) повторять
        v1 ← значение ((результат s0, c), противник (первый));
        если лучший (v1, v, первый) то
            лучший–ход ← c;
            v ← v1
```

где

```
программа значение : ЦЕЛОЕ
            (аргументы s : СИТУАЦИЯ, j: ИГРОК)
    переменная v : ЦЕЛОЕ;
    если конечное (s) то
                значение ← оценка (s)
    иначе
                значение ← максимум (j);
                для c из списокходов (s, j) повторять
                    v ← значение (результат (s, c), противник (j));
                    если лучший (v, значение, j) то
                        значение ← v
```

Программа лучший–ход реализует, таким образом, метод минимакса. Возможно существенное улучшение, известное под названием (α, β)–процедуры; она восходит к первым играющим программам, и на авторство претендуют многие ученые. Принцип состоит в следующем: двигаясь «сначала вглубь», предположим, что на произвольном уровне дерева получают ситуацию, подобную представленной на Рис. VI.24. Поскольку

противник – минимизирующий игрок, обеспечиваемое вершине B значение есть –5.

Затем исследуют вершины F, G, H, ... Вершина G позволяет обеспечить в F предварительный минимум 6; вершина H позволяет заменить минимум, соответствующий F, на –8. Но рассмотрим вершину A, которую надо максимизировать.

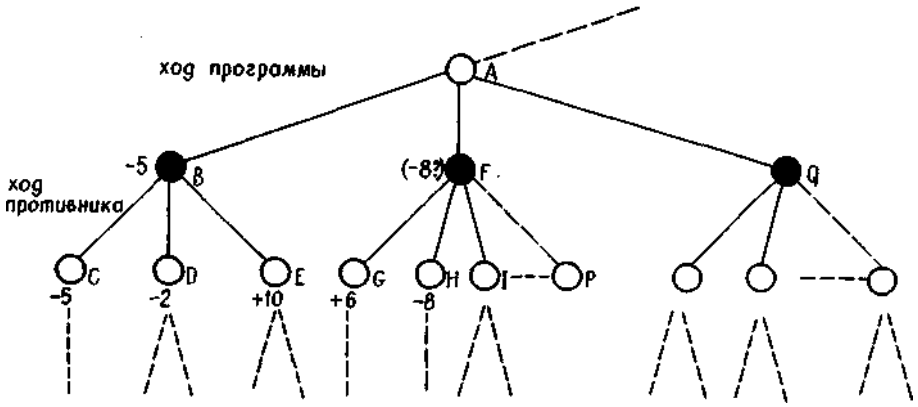


Рис. VI.24 Процедура альфа–бета.

Предварительный максимум есть значение вершины B, например –5; другими словами, максимизирующий игрок в вершине A, наверняка, будет иметь ход, обеспечивающий значение, большее или равное –5; итак, если выбран ход, ведущий в F, то противник мог бы дать этой вершине значение, меньшее или равное –8. Никогда максимизирующий игрок не выберет такого хода: отпадает, таким образом, надобность в исследодовании вершин I, J, ..., P и тем более в построении поддеревьев, имеющих эти вершины в качестве корней. Выигрыш в эффективности может быть, следовательно, значительным.

В нашей программе (α, β)–процедуру можно учесть очень просто. Достаточно к процедуре значение добавить параметр, означающий порог снизу (или сверху), при котором нет необходимости продолжать исследование дерева. В результате получим

```
программа значение : ЦЕЛОЕ
          (аргументы s : СИТУАЦИЯ,
          j : ИГРОК,
          порог : ЦЕЛОЕ)
переменная v : ЦЕЛОЕ;
если конечное (s) то
    значение ← оценка (s)
иначе
    значение ←максимум (s);
    для c из списокходов (s, j) пока ~ лучший (значение, порог j) повторять
        v ← значение (результат (s, c), противник (j), значение);
        если лучший (v, значение, j) то
            значение ← v
```

Достаточно, чтобы вызов значение, имеющийся в программе лучший–ход (строка, отмеченная выше (а)), инициализировал порог значением ±∞: теперь этот вызов запишется в виде

$v_1 \leftarrow$ значение (результат (s_0, c), противник (первый), v)

УПРАЖНЕНИЯ

VI.1. Бухгалтерская задача

Исходя из последовательного файла F, содержащего расходы по различным счетам, требуется напечатать бухгалтерский отчет с частичным подведением итогов.

Файл F представляет собой последовательность пар [счет, расход], где каждый счет представлен номером, содержащим от 1 до 9 цифр. Он упорядочен в лексикографическом порядке номеров счетов. Типичным фрагментом файла мог бы быть:

счета	расходы
4516	10000
4517	20000
452932	5000
5542	30000
66	1000

и т.д.

Корень счета – это номер, образованный одной или несколькими первыми цифрами; например корни от 78455 – 7, 78, 784 и 7845. Корни имеют бухгалтерский смысл: они позволяют сгруппировать подсчета одной категории. Например, корень 94 (служебные покупки) мог бы объединять 945 (служебные покупки во Франции), 947 (служебные покупки за границей в Европе), 948 (покупки за пределами Европы). Корень 947 в свою очередь мог бы быть подразделен на 9475, 9478 и 9479 и т.д. В последовательном файле никогда одновременно не встречаются счет и один из его корней, например 9472 и 94.

При выдаче ведомости расходы будут воспроизводиться такими, какими они записаны в файле, так же как и частичные итоги по «обрезанным» m первым цифрам корня (m – константа, задаваемая на входе). Например, для m = 3 и приведенного фрагмента файла будет напечатано:

4516 10000 (воспроизведение данного)

4517 20000 (воспроизведение данного)

451 30000 (итог по 451)

и т.д.

Однако во избежание неинформативной выдачи учитываются два следующих правила:

а) не печатать итог для корней, соответствующих только одному счету. В данном случае, например, после предыдущих строк будет напечатано:

452932 5000 (воспроизведение входа)

без суммирования по 45293, 4529 и 452.

б) не печатать итог при урезании последовательности на i–й цифре ($1 \le i < m$), если оно также соответствует урезанию последовательности на цифре более высокого ранга j. Последнее урезание учитывается отдельно ($1 \le i < j \le m$). Например,

451 30000(итог по 451)

452932 5000 (воспроизведение входа)

45 35000(итог по 45)

5542 30000(воспроизведение входа: нет итога по 4, который не нужен

при наличии итога по 45)

Считая m заданным, требуется составить программу печати ведомости.

VI.2. Поиск корня

На любом языке программирования составить нерекурсивный вариант подпрограммы поиска корня численной функции (VI.2.2). Показать, что можно ограничиться стеком двоичных значений.

VI.3. Стек и куча

Почему предлагаемая языком ПЛ/1 возможность сделать так, чтобы указатель, принадлежащий к «куче», обозначал данные, принадлежащие «стеку», квалифицируется в VI.3.4.3 как «смесь сомнительного свойства»?

VI.4. Головоломка

В магазине можно купить головоломку, образованную из n колец, закрепленных на n стержнях, которые скользят внутри деревянной дощечки и вытянутой рамки (Рис. VI.25). В начале каждое кольцо проходит через стержень следующего кольца, а вытянутая рамка проходит вокруг n стержней, но внутри всех колец, как показано на рисунке.

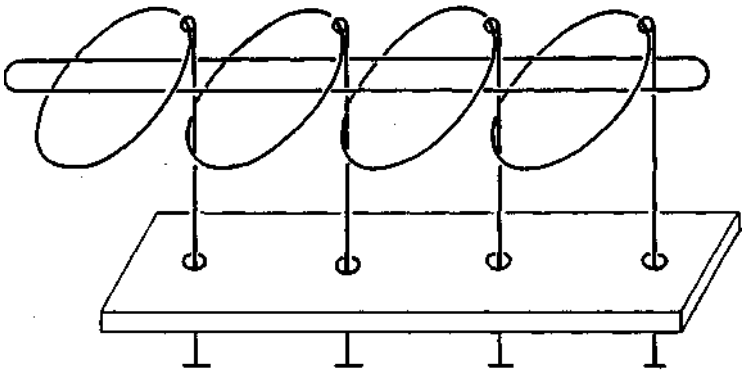


Рис. VI.25 Головоломка.

Требуется вытащить рамку.

VI.5. Язык ПЛ/–1

Рассмотрим язык программирования, называемый ПЛ/–1, в котором:

- единственными базовыми объектами, обрабатываемыми непосредственно, являются целые натуральные и логические константы **истина** и **ложь**;
- единственные базовые операции даются функцией преемник (такой, что преемник (x) обозначает x + 1 для всякого целого натурального x) и предикатом равно (такой, что равно(x, y) имеет значение **истина** тогда и только тогда, когда целые натуральные x и y равны);
- единственными средствами построения программ являются условные выражения вида

если предикат **то** выр1 **иначе** выр2

Возможны определения программ с одним или несколькими параметрами аргументы типа ЦЕЛОЕ или ЛОГ и одним результатом ЦЕЛОЕ или ЛОГ (определения, возможно, рекурсивные) и вызовы таких подпрограмм.

Показать, что на ПЛ/– 1 можно написать подпрограммы, вычисляющие сумму двух целых натуральных, их разность, если она определена, их произведение, их целое

частное и остаток и т.д., а также подпрограммы с результатом ЛОГ, соответствующие операторам обычных отношений (< , ≤ , >, ≥) и логическим операторам (**и**, **или**, **не**). Написать на ПЛ/–1 подпрограмму, определяющую, является ли заданное число простым.

Показать, что с теоретической точки зрения ПЛ/–1 обладает теми же возможностями и той же общностью, что и самые «развитые» языки программирования.

VI.6. Функция Аккермана

Классическая функция в теории рекурсии задается следующей рекурсивной формулой (для целых m и n):

Аккерман (m, n) = **если** m = 0 то n + 1
иначе если n = 0 то *Аккерман* (m – 1, 1)
иначе *Аккерман* (m – 1, *Аккерман* (m, n – 1))

Найти значение функции для некоторых (малых!) целых неотрицательных значений тип. Рассмотреть возможную реализацию вычисления.

VI.7. Факториал и К°

[Кадью 72] Пусть для целого n рекурсивно определены три функции:

$f_1(n)$ = **если** n = 0 **то** 1 **иначе** n × f_1(n–1)
$f_2(n)$ = **если** n ≤ 0 **то** 1 **иначе** n × f_2(n – 1)
$f_3(n)$ = F(n, 0, 1)
при F (x, y, z) = **если** x = y **то** z **иначе**
F(x, y + 1, (y + 1) × z)

Показать, что для n ≥ 0

$f_1(n) = f_2(n) = f_3(n)$ = факториал (n)

Идентичны ли эти три функции?

VI.8. Функция 91

Чему равна для целых n функция, определенная с помощью

маккарти (n) = **если** n > 100 то n – 10
иначе *маккарти (маккарти* (n + 11))?

VI.9. Рекурсия и передача аргументов

[Кадью 72] Пусть целая функция с двумя аргументами определена с помощью подпрограммы

программа кадью : ЦЕЛОЕ (**аргументы** x, y : ЦЕЛЫЕ)
кадью ← **если** x = 0 **то** 1
иначе x × *кадью (*x – 1, *кадью* (x, y + 1))

Попытайтесь вычислить кадью (27, 42). Всегда ли заканчивается выполнение кадью для неотрицательных x и y? Как влияет на результат передача параметров по **имени** или по **значению?**

(*Подсказка:* подпрограмма использует значение своего второго формального параметра только для образования нового фактического параметра рекурсивного вызова.)

VI.10. Восемь ферзей

Задача «о восьми ферзях» принадлежит Гауссу. Требуется разместить восемь ферзей на шахматной доске 8×8 так, чтобы они не били друг друга, т.е. чтобы ни один из них не находился на одной и той же горизонтали, вертикали или диагонали, что и любой другой:

а) найти некоторое решение;

б) найти все решения задачи, т.е. все правильные конфигурации восьми ферзей, не бьющих друг друга на шахматной доске;

в) найти все решения, взяв только по одному экземпляру из каждого класса решений, которые сводятся одно к другому путем простых геометрических преобразований (симметрии, поворотов).

VI.11. Коды Грэя

В теории используются логические цепи *кодов Грэя*. Код Грэя порядка n есть перестановка 2^n слов длиной n битов в таком порядке, что i–е слово отличается от $(i + 1)$–го в единственной позиции, и i берется по модулю n. Пример кода Грэя порядка 3:

000 001 011 010 110 111 101 100

Рассматриваются коды Грэя порядка 4 (т.е. содержащие 16 слов длиной 4 бита). Два кода считаются эквивалентными, если они сводятся один к другому путем циклической перестановки слов кода, путем циклической перестановки позиций битов, выполняемой над каждым словом кода, или путем замены каждого слова кода p_i на $p_i \oplus x$, где x – произвольное слово длиной 4 бита, $\oplus$ – «исключающее или».

Требуется перечислить коды Грэя порядка 4, беря для каждого класса эквивалентности только первый из кодов в лексикографическом порядке. (*Замечание:* решение содержит 11 различных неэквивалентных кодов. *Рекомендация:* попробуйте сначала «вручную» случай порядка 3, для которого решение содержит единственный код, указанный выше.)

VI.12. Беседы

Арсена, Урсула, Селимена, Адельфа, Гермогена, Аделаида, Раиса и Эмелина каждый день вместе пьют чай и каждый раз ведут разные беседы. В ходе каждой беседы Арсена семь раз говорит: «Ну и что». Урсула пять раз говорит: «Почему бы и нет?» Селимена восемь раз говорит: «Может быть...» Адельфа четыре раза говорит: «Я очень сомневаюсь!» Гермогена шесть раз говорит: «Надо, так надо». Аделаида семь раз говорит: «Все это политика!» Раиса шесть раз говорит: «Нет, а вы?» Эмелина семь раз говорит: «Да, но это не надолго». Никто никогда не высказывается два раза подряд.

Через сколько дней наши героини исчерпают все возможные беседы?

ГЛАВА VII.
АЛГОРИТМЫ

Чтобы играть в эту игру, нужны две игральные кости–кубики с нанесенным на каждую грань числом очков от 1 до 6. Каждый игрок бросает кости по очереди и передвигает свою фишку на число клеток, равное сумме очков, выпавшей на двух костях. В начале игры особая роль отведена сумме очков, равной 9. Дело в том, что отмеченные фигуркой гуська клетки, на которых не разрешено останавливаться, следуют через девять клеток. Другое правило гласит, что при попадании фишки на гуська игрок удваивает выпавшую на костях сумму. Тогда, если в начале игры выпадает 9, это может привести игрока сразу на заключительную клетку 63. Поэтому при начальной девятке, составленной из 3 и 6, игрок переставляет свою фишку на 26, где нарисованы две игральные кости, а при девятке, составленной из 4 и 5–на 53, где нарисована другая пара костей. Игрок, попавший на 6, где нарисован мост, перемещается на клетку 12 и «тонет» под другим мостом–выбывает из игры. Попадая на клетку 19, на которой изображена гостиница, игрок «отдыхает»–пропускает один ход. Тот, кто приходит на 31, где нарисован колодец, остается там до тех пор, пока какой–нибудь другой участник игры не окажется на этой оке клетке. Тогда вновь прибывший игрок «вытаскивает» первого из колодца, но сам остается на его месте; «вылезший» из колодца игрок отправляется на клетку, с которой попал в колодец его спаситель. Прибывший на 42 попадает в лабиринт и вынужден вернуться на 30. На клетке 52 расположена тюрьма, где игрок «сидит», пока кто–нибудь другой его оттуда не освободит (история, похожая на колодец – Перев.). В клетке 58 нарисован череп: прибывший сюда игрок должен начинать игру сначала. Если один игрок догнал другого на любой из клеток, то первый прибывший отправляется назад на ту клетку, с которой прибыл последний игрок. Если, подходя к номеру 63, игрок получает лишнее количество очков, то он должен отступить назад на величину, равную излишку: он может достичь 63, только точно набрав необходимое число очков.

Правила игры «Гусек»

АЛГОРИТМЫ

Алгоритм – это точное и строгое описание последовательности операций, позволяющей за конечное число шагов получить решение задачи.

Эта глава знакомит с несколькими фундаментальными алгоритмами, которые должны присутствовать в арсенале всякого программиста. Таким образом, это «техническая» глава, как и гл. V, в которой рассматривались тесно связанные понятия, относящиеся к структурам данных. Изложение направлено на то, чтобы дать не только конкретно используемые *орудия,* но и *методы* построения правильных, надежных и эффективных алгоритмов, применимых во всех областях информатики, а не только в трех важных приложениях, рассмотренных здесь подробно.

VII.1. Общие сведения. Методы

Эта глава посвящена представлению и обсуждению некоторого числа фундаментальных алгоритмов. Нам предстояло сделать выбор из массы опубликованных и просто известных важных алгоритмов; мы остановились на нескольких примерах, выбранных из соображений их практического интереса и методического значения. Мы ограничиваемся тремя областями:

- **управлением таблицами** в памяти;
- **сортировкой**, или **упорядочением**, массивов;
- **обработкой текстов**.

Читатель сможет найти другие многочисленные примеры важных алгоритмов в энциклопедическом труде Кнута (вышли три тома из объявленных семи: [Кнут 68], [Кнут 69], [Кнут 73]) и более краткой работе Ахо, Хопкрофта и Ульмана [Ахо 74].

Прежде чем представить собственно алгоритмы, определим важнейшее понятие *сложности алгоритмов* и несколько методов, нужных программисту при поиске эффективных алгоритмов.

VII.1.1. Сложность алгоритмов

Понятие «сложность алгоритмов» позволяет придать точный смысл интуитивной идее, по которой алгоритм характеризуется «стоимостью». Под стоимостью понимают, вообще говоря, время выполнения и количество требуемой памяти.

Важно различать **практическую сложность,** которая является точной мерой времени вычислений и объема памяти для конкретной модели вычислительной машины, и **теоретическую сложность,** которая более независима от практических условий выполнения алгоритма и дает порядок величины стоимости.

Мы введем эти понятия сначала на простом примере. Пусть написана программа, определяющая для любого целого n > 1 его **наибольший делитель,** отличный от самого n (результатом, следовательно, будет единица, если n простое). Такая программа имеет вид

```
программа наибольший делитель : ЦЕЛОЕ ПОЛОЖИТЕЛЬНОЕ
        (аргумент i : ЦЕЛОЕ ПОЛОЖИТЕЛЬНОЕ)
    переменная i: ЦЕЛОЕ ПОЛОЖИТЕЛЬНОЕ;
    i ← n – 1;
(A) пока n мод i ≠ 0 повторять
        {инвариант цикла: наибольший делитель n меньше или равен i}
        i ← i + l;
    наибольший делитель ← i
```

(Цикл завершается, так как n мод 1 = 0.)

Другой возможный метод основан на том, что разыскиваемый результат есть n/i, где i – наименьший делитель n, превосходящий единицу. Если n простое, этот «наименьший делитель» равен n; но если n непростое, то его наименыший делитель» с необходимостью содержится между 2 и $\sqrt{n}$. Можно, таким образом, написать другой вариант программы:

```
программа наибольший делитель : ЦЕЛОЕ ПОЛОЖИТЕЛЬНОЕ
        (аргумент n : ЦЕЛОЕ ПОЛОЖИТЕЛЬНОЕ)
    переменная i: ЦЕЛОЕ ПОЛОЖИТЕЛЬНОЕ;
    i ← 2;
(B) пока i < √n и n мод i ≠ 0 повторять
        {инвариант цикла: наименьший делитель n, превосходящий 1, больше
        или равен i}
        i ← i + 1;
    наименьший делитель ← (если n мод i = 0 то n/i иначе 1)
```

Чтобы сравнить время выполнения вариантов (A) и (B), можно отметить, что оба они имеют вид

```
I1
пока c повторять
    I2
I3
```

Пусть t_1, t_2 и t_3 – времена выполнения операторов I_1, I_2 и I_3 соответственно. Предположим, что цикл **пока** представлен в машине обычным образом (III.5.2.2) с помощью одного условного и одного безусловного перехода, требующих соответственно времен t_y и $t_б$. Тогда время выполнения того или другого варианта определяется как

$$t = t_1 + t_3 + m(t_y + t_2 + t_б) + t_y$$

где m – число повторений цикла. Следовательно, время выполнения имеет вид

$$t = P + Qm$$

где P и Q – константы (различные в этих двух вариантах программы).

В варианте (A) программы максимальное число выполнений цикла равно

$$m_A = n - 2$$

В варианте (B), напротив, цикл выполняется по меньшей мере

$$m_B = \lceil \sqrt{n} \rceil - 2 \text{ раз}^1$$

[1] Напомним, что для вещественных x $\lfloor x \rfloor$ обозначает его целую часть, т.е. целое n, такое, что n ≤ x < n + l, а $\lfloor x \rfloor$

Тогда *максимальная временная сложность* варианта (A) определяется в виде

$a + bn$

где a и b – константы, а для варианта (B) эта величина равна

$a' + b'\sqrt{n}$

где a' и b' – другие константы.

Детализация *практической сложности* этих двух вариантов привела бы к вычислению констант a, b, a', b', определяемых конкретной ЭВМ и зависящих от времени выполнения ее команд. Гораздо более интересна в этом случае *теоретическая сложность,* сравнивающая два алгоритма. Когда n велико, она определяется членом, пропорциональным величине n в варианте (A), и членом, пропорциональным величине $\sqrt{n}$ в варианте (B). При n порядка 10^{10} цикл варианта (A) потребовал бы недопустимо большого на современных ЭВМ времени, тогда как число выполнений цикла варианта (B) имеет порядок 100000, что уже вполне реализуемо. Можно говорить о *максимальных временных сложностях* (A) и (B) соответственно:

$O(n)$ и $O(\sqrt{n})$

где O означает «величина порядка»[1].

Рассмотрим в общем случае некоторый алгоритм A. Почти всегда существует параметр, обозначаемый n и характеризующий объем данных, к которым в каждом случае применяется A. Если, например, A – это алгоритм сортировки, то n – число сортируемых данных. Пусть $T(n)$ – время выполнения алгоритма A, выраженное как функция от n, а f – некоторая функция; говорят, что алгоритм A имеет теоретическую сложность $O(f(n))$, если отношение $T(n)/f(n)$ ограничено при $n \to \infty$ (в частности, речь может идти о случае, когда $T(n)/f(n)$ стремится к константе k, т.е. – в математических терминах– если T эквивалентно kf).

Например, алгоритм может иметь теоретическую сложность $O(n)$, $O(n^2)$, $O(2^n)$, $O(n\log n)$ (не уточняя основание логарифмов) и т.д.

Если операция выполняется за фиксированное время, не зависящее от размера задачи, говорят, что ее сложность O(1).

Это определение теоретической сложности легко обобщается на случай, когда время выполнения зависит от нескольких параметров. Например, алгоритм, определяющий, входит ли множество m элементов в множество n элементов, может иметь в зависимости от используемых структур данных сложность $O(m\times n)$ или $O(m + n)$.

Практически время выполнения алгоритма зависит не только от объема множества данных, но также и от их значений, например, время выполнения некоторых алгоритмов сортировки существенно сокращается, если первоначально эти данные частично упорядочены (тогда как другие алгоритмы могут оказаться нечувствительными к этому свойству). Чтобы учитывать этот факт, полностью сохраняя при этом возможность анализировать алгоритмы независимо от их данных, различают:

- **максимальную сложность**, или сложность наиболее неблагоприятного случая, являющуюся сложностью функции T_{max}, где $T_{max}(n)$ – время выполнения алгоритма, когда выбранные n данных порождают наиболее долгое выполнение алгоритма. Именно ее мы рассматривали в двух вариантах программы «наибольший делитель»;
- **среднюю сложность**, которая есть сложность функции $T_{ср}$ – среднее время выполнения алгоритма, примененного к n произвольным данным. Следует

обозначает целое n', такое, что $n' - 1 < x \le n'$.

[1] Не путать с обозначением $o(f(n))$, используемом в математике, которое читается: «пренебрежимо по сравнению с ...». В частности, для всякого α и $\beta > 0$ $o(n^{\alpha}) = o(n^{\alpha+\beta})$. См. упражнение VII.1.

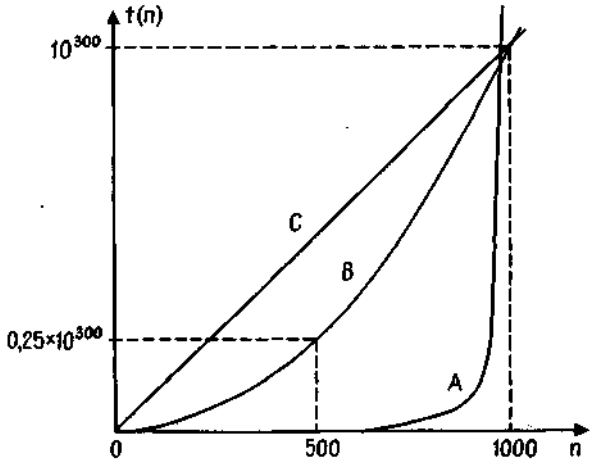
отметить, что $T_{ср}(n)$ не совпадает с понятием, выводимым из вероятностных гипотез о распределении данных.

Эти понятия без труда распространяются на измерение стоимости в единицах объема памяти: можно говорить о средней и максимальной **пространственной сложности.**

Теоретический подход к понятию сложности, который мы здесь используем, может вызвать критические замечания, поскольку он игнорирует коэффициенты пропорциональности и учитывает только асимптотические отношения; так, считается, что все функции n^2,

$10^{75}n^2$, $1000n^2 + 10^{75}n + 10^{150}$, $\frac{n^2}{10^{75}} + \log n$ имеют сложность $O(n^2)$.

Действительно, функция сложности $O(2^n)$ может оказаться меньше функции сложности $O(n^2)$ для всех практически рассматриваемых значений:



Функции $A(n) = 2^n - 1$ (увеличено), $B(n) = 10^{294}n^2$, $C(n) = 10^{297}$ (масштаб по оси ординат в 10^{297} раз превосходит масштаб по оси абсцисс).

Тем не менее два обстоятельства могут оправдать понятие теоретической сложности:

а) [Axo 74] Пусть n_0, n_1 n_2 и n_3 – максимальные размеры задач, соответствующих четырем алгоритмам A, B, C, D теоретической сложности $O(2^n)$, $O(n^2)$, $O(n)$ и $O(\log n)$. Допустим, что конструктор X предлагает новую машину, в тысячу раз более быстродействующую, чем предыдущая. Теперь можно работать с задачами размерности:

$n'_0 \simeq n_0 + 10$	для A	$n'_2 \simeq 1000n_2$	для C
$n'_1 \simeq 32n_1$	для B	$n'_3 \simeq n_3^{1000}$	для D

Значит, вопреки тому, что можно было бы предполагать, прогресс в технологии делает еще более важным поиск эффективных с точки зрения теоретической сложности алгоритмов (напротив, коэффициенты, участвующие в практической сложности, становятся менее существенными).

б) В каждом отдельном случае трудно дать точную величину теоретической сложности алгоритма вне связи с представлением в конкретной машине. И все же видно, что все научные школы, следуя Кнуту, развивают теорию *анализа алгоритмов,* дающую подробные результаты в практической сложности алгоритмов, выраженные в количестве базовых операций. В этой главе мы будем интересоваться в первую очередь теоретической сложностью представляемых алгоритмов. Тем не менее будут даны несколько оценок точных результатов в числе выполняемых операций.

VII.1.2. Методы поиска эффективных алгоритмов

Построения эффективных алгоритмов, т.е. алгоритмов с невысокой пространственной и временной сложностью, безусловно, составляют наиболее трудные задачи программирования. Не существует волшебного правила, гарантирующего получение хорошего алгоритма; однако некоторые общие принципы поиска таких алгоритмов представляются важными.

VII.1.2.1. Разделяй и властвуй

Одна из самых конструктивных идей состоит в разложении задачи «размерности» n на одну операцию некоторой сложности $O(n)$ или $O(1)$, например, и похожую задачу размерности m, меньшей, чем n, или в общем случае на k задач с размерностями m_1, m_2, ..., m_k, таких, что $m_1 + m_2 + ... + m_k < n$. Это не что иное, как идея *рекурсии* в том виде, в каком она была представлена в VI.1 (параметризация, решение тривиального случая, рассмотрение общего случая, сведение его к более простому). Эта идея открывает путь к решению многих задач. Из нее исходят, в частности, почти все методы *сортировки* (VII.3).

VII.1.2.2. Уравновешивание

Все методы «разделения» не всегда, однако, приводят к одинаковому с точки зрения теоретической сложности результату. Пусть метод, сводящий задачу размерности n к задаче размерности $n - 1$, использует операции сложности $O(n)$, а $T(n)$ – временная сложность алгоритма. По определению

$T(n) \le cn + T(n - 1)$, где c – константа.

Раскрывая, получаем

$$T(n) \le cn + c(n - 1) + c(n - 2) + ... + c + T(0) \le c\frac{n(n+1)}{2} + T(0)$$

и, следовательно,

$T(n) = O(n^2)$

Пусть наряду с этим существует метод, позволяющий свести задачу размерности n к двум подзадачам размерности $\left\lfloor \frac{n}{2} \right\rfloor$ тоже посредством операции, имеющей сложность $O(n)$. Временная сложность $T'(n)$ в этом случае будет

$$T'(n) \le cn + 2T'\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

и, значит,

$$T'(n) \le cn + 2c\left\lfloor \frac{n}{2} \right\rfloor + 2c\left\lfloor \frac{n}{4} \right\rfloor + ... + 2^{[\log_2 n]}\, T'(1)$$

т.е.

$$T'(n) \le \underbrace{cn + cn + cn + ... + cn}_{\lfloor \log_2 n \rfloor \text{ членов}} + nT'(1)$$

Следовательно,

$T'(n) = O(n\log n)$

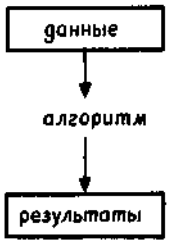
что, несравненно, лучше, чем $0(n^2)$. Таким образом, недостаточно разделять задачу для получения эффективных алгоритмов: нужно *уравновешивать* разделяемые части. Легко показать, что среди возможных способов разбиения деление на равные части дает в

общем случае оптимальный результат. Алгоритмы дихотомического поиска (VII.2.3) или Быстрой сортировки (VII.3.6) являются прямыми приложениями этого принципа.

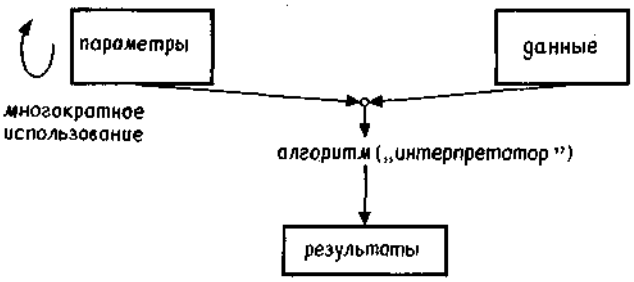
Следует, однако, отметить, что точный оптимум в некоторых случаях достигается не строго равновесным разделением (ср. упражнение VII.2).

VII.1.2.3. Компиляция или интерпретация

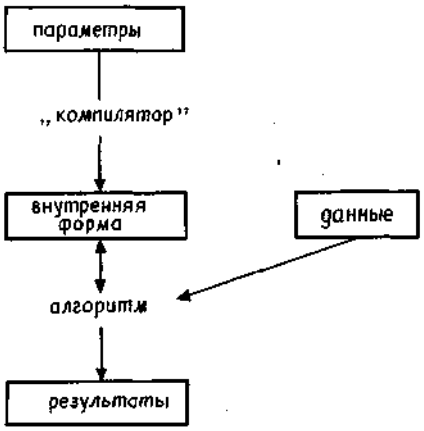
Первый набросок алгоритма приводит зачастую к обработке всех данных следующим образом:



Однако данные бывают двух типов: одни являются параметрами алгоритма и используются многократно, другие служат для описания точных характеристик каждой задачи, решаемой в текущий момент. Вышеприведенную схему лучше переписать так:



По аналогии со структурой систем, которые позволяют выполнять написанные на языках программирования программы, и рассматривая «параметры» как «программу», написанную на некотором языке, можно, таким образом, сказать, что такой алгоритм есть *интерпретатор* (1.5). Зачастую на упрощении алгоритма решающим образом сказывается предварительное приведение параметров к некоторой внутренней форме, позволяющей более эффективное использование, чем исходная «человеческая» форма. В таком случае задача сводится, следовательно, к *компиляции* параметров:



Внутренняя форма могла бы быть структурой данных быстрого доступа, таблицей и т.д. В VII.4 мы увидим алгоритм обработки текстов, где внутренней формой является «диаграмма переходов», которая есть результат «компиляции» некоторого числа текстов.

VII.1.2.4. Соотношение пространство–время

Важной особенностью программирования является тот факт, что часто приходится выбирать между уменьшением пространственной сложности алгоритма и уменьшением его временной сложности, так что одно делается в ущерб другому[1]. В качестве примера предположим, что три величины X, Y, Z связаны уравнением типа

$X = f(Y, Z)$

Можно представить, что X, Y и Z являются физическими величинами (примеры: ток, напряжение, сопротивление); такая ситуация часто имеет место в задачах управления предприятием (примеры: кредит, дебит, остаток). При большом числе триплетов вида [X, Y, Z], например 10000 или миллион, имеет практический смысл вопрос, надо ли сохранять все триплеты или лучше хранить только пары [Y, Z], а соответствующие X каждый раз вычислять с помощью формулы $X_i = f(Y_i, Z_i)$, когда в них возникает необходимость. На этот вопрос, очевидно, нет общего ответа; выбор зависит от способа и частоты использования различных данных, а также от относительной важности ограничений во времени или в пространстве. Можно, однако, руководствоваться двумя общими правилами:

- осознавать важность отношений пространство–время: часто приходится решать задачи, которые требуют, казалось бы, огромных объемов памяти, систематическим перевычислением некоторых величин (на первый взгляд такие перевычисления могли бы показаться абсурдными);
- считать всегда, что имя данного представляет не данное само по себе, а механизм доступа к этому данному. Тогда программам, использующим X из нашего приведенного выше примера, не обязательно «знать», получен ли X непосредственно или вычислен: нет необходимости менять эти программы при изменении внутреннего представления триплетов, например, если в результате статистических проверок решено, что более экономично вычислять X путем вызова подпрограммы, нежели хранить их в памяти.

Мы говорили уже подробно об этом принципе в гл. V, обосновывая понятие *функциональной спецификации* структуры данных.

VII.2. Управление таблицами

VII.2.1. Определение и общие сведения

Управление таблицами–задача, часто встречающаяся в программировании: решая задачи информатики, обрабатывают некоторое число объектов, принадлежащих некоторому *множеству* и обладающих некоторыми *свойствами.* Важно хранить эти объекты с тем, чтобы можно было ответить на вопрос типа «встречался ли раньше объект x?, если «да», то каковы его свойства?»

Наиболее классический пример такого типа проблем–это управление «таблицей символов» в трансляторе: в ходе анализа программы транслятор распознает индентификаторы переменных, которые он должен хранить в таблице, чтобы позднее проверять, были ли они объявлены, и обеспечивать доступ к таким их свойствам, как *тип,* или *адрес,* который им, возможно, был назначен, или *длина,* если речь идет о переменных типа СТРОКА (в АЛГОЛе W или ПЛ/1, например), и т.д. Аналогичная проблема встречается в большинстве программ для задач управления производством, где необходимо обрабатывать «записи», соответствующие изделиям, служащим и т.д.,

[1] Это относится только к алгоритмам, которые уже являются достаточно эффективными. Многие из существующих программ могут быть на порядок улучшены как по времени, так и по объему памяти одновременно с помощью элементарных приемов, например улучшением используемых структур данных.

сохраняя соответствующие сведения.

В этом разделе мы будем полагать, что обрабатываемые объекты принадлежат к некоторому типу, который мы будем обозначать ЭЛЕМЕНТ, и что каждый из этих объектов обладает некоторым числом свойств, одно из которых играет особую роль и будет называться **ключом** этого объекта. Если p – элемент, его ключ обозначается как

ключ (p)

и мы предполагаем, что он относится к типу, обозначаемому КЛЮЧ.

Задача управления таблицами состоит, по существу, в реализации двух функций:
включение : ЭЛЕМЕНТ × ТАБЛИЦА → ТАБЛИЦА
поиск : КЛЮЧ × ТАБЛИЦА → (ЭЛЕМЕНТ | ПУСТО)

Функция «включение» элемента в таблицу состоит в размещении его способом, обеспечивающим последующее обнаружение элемента, а «поиск» позволяет с помощью ключа получить предварительно размещенный элемент, обладающий этим значением ключа; результатом служит ПУСТО, если такого элемента нет. Например, объекты типа ЭЛЕМЕНТ могут быть представлениями «личностей», характеризуемых фамилией, именем, возрастом, адресом и т.д.; «ключом» может служить фамилия. Тогда функция поиска состоит в обнаружении, встречалась ли ранее личность с заданной фамилией (т.е. информационным образом личности). Точно так же в примере с транслятором «ключом» объекта, представляющего переменную, очевидно, является ее идентификатор.

Представим эти операции подпрограммами:

```
программа включение (аргумент x : ЭЛЕМЕНТ;
                      модифицируемое данное t: ТАБЛИЦА)
    ....
программа поиск : ЭЛЕМЕНТ(аргумент c : КЛЮЧ, t: ТАБЛИЦА)
    ...
```

приняв соглашение, что ЭЛЕМЕНТ может быть величиной ПУСТО.

Сразу же видно, что в зависимости от ситуации можно предусмотреть однозначность ключей в таблице или отказаться от нее: в некоторых случаях для операции «включение» безразлично, существует ли уже объект с тем же значением ключа, что и включаемый элемент; в других, наоборот, важно сохранять все элементы, даже если некоторые ключи встречались многократно. Далее мы будем предполагать, что однозначность ключей не предусматривается; если же в ней возникает необходимость, достаточно будет заменить встречающиеся далее алгоритмы включения на их соответствующие видоизмененные версии типа

```
если поиск (ключ (x), t) = ПУСТО то
    включение (x, t) {где «включение» – это алгоритм включения, не
    предусматривающий однозначность ключей}
```

В некоторых случаях можно обеспечить однозначность ключей также путем внутренней модификации подпрограмм включения; так обстоит дело, например, в методе, использующем двоичное дерево (VII.2.4). Необходимые изменения тривиальны, и мы оставляем их читателю.

Заметьте также, что на практике бывает необходимым комбинировать операции поиска и включения; например, когда транслятор обрабатывает объявления, он проверяет одновременно, что не существует идентификатора с тем же именем, ранее объявленного в том же блоке, и, если это условие выполняется, включает новый идентификатор в таблицу символов. В этом случае, безусловно, рекомендуется сохранять логическое разделение этих двух операций; но безрезультатный поиск может часто давать вызывающей программе, кроме результата ПУСТО, еще и указание места, куда может быть включен новый элемент; тем самым отпадает необходимость

повторять ту же работу для операции включение– Среди приводимых ниже алгоритмов отмечены те, которые легко поддаются такой модификации, предоставляемой, впрочем, читателю. Другая очевидная модификация состоит в том, чтобы результатом поиска был не элемент, а средство доступа к этому элементу (указатель, индекс таблицы и т.д.).

Понятие ТАБЛИЦЫ, рассматриваемое в этом разделе, это то же самое, что структура данных МНОЖЕСТВО, функциональная спецификация которой была дана в V.3. Здесь нас будут интересовать наиболее эффективные (по времени и объему памяти) физические представления хранимых данных и операций поиска и включения. Как это было показано в V.3, иногда оказываются полезными другие операции–удаление элемента (заданного своим ключом) или слияние двух таблиц; будут отмечены их представления, позволяющие их удобное выполнение.

Общая проблема хранения и поиска информации, конечно, более сложна, чем она показана здесь, хотя основными методами являются методы, изложенные в этом разделе. Большие размеры некоторых информационных множеств приводят, действительно, к особым трудностям, вызываемым необходимостью ограничить число вводов и выводов, когда данные размещаются во внешней памяти. Другим обобщением является наличие нескольких ключей: если обрабатывается таблица объектов, представляющих личности, например телефонный справочник, может возникать необходимость обращения то по ключу «фамилия», то по ключу «адрес». Некоторые приложения требуют также, чтобы система могла отвечать на более сложные «вопросы», чем поиск по ключу; например, информационная библиотечная система должна уметь обрабатывать запросы вида: «Назовите авторов всех работ по географии Океании, опубликованных во Франции до 1975 г.» Подобные проблемы, лежащие за рамками этого раздела, составляют предмет исследования *баз данных* [ACM 76в], [Кодд 70], [Абриал 74].

Методы, с которыми мы будем знакомиться, эффективны, по крайней мере с точки зрения временных оценок, в такой же степени, в какой они располагают информацией о ключах. Мы будем предполагать прежде всего, что о ключах ничего не известно. Это даст алгоритмы поиска сложности O(n) и алгоритмы включения сложности O(l) для *неупорядоченной таблицы*. Если существует порядок по ключам, можно использовать *упорядоченную таблицу* (поиск сложности O(n) или O(logn) в зависимости от представления, включение сложности O(n)) или *двоичное дерево* (поиск и включение сложности O(logn) при некоторых предосторожностях). Наконец, если можно использовать ключи как «псевдоадреса», получают весьма эффективный метод *ассоциативной адресации* (поиск и включение сложности O(1) в большинстве ситуаций и O(n) в самых неблагоприятных случаях).

Для каждого метода будет дана оценка практической сложности; к сожалению, трудно указать точные правила типа: «для таблиц от 150 до 250 элементов выбирайте такой–то метод». Практическая сложность зависит от многочисленных факторов: в некоторых случаях, например, существенна стоимость сравнений ключей; в других, напротив, она пренебрежима по сравнению со стоимостью операторов управления циклами или рекурсией и т.д. Приводимые результаты относятся к «разумным» случаям, когда сравнение ключей требует примерно времени от 5 до 100 базовых операций.

VII.2.2. Последовательная неупорядоченная таблица

VII.2.2.1. Представления

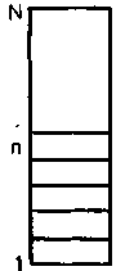
Если не высказано никакой конкретной гипотезы о ключах, самый простейший метод состоит в использовании представления с помощью линейного списка (V.6):

тип ТАБЛИЦА = ЛИНЕЙНЫЙСПИСОК$_{\text{элемент}}$

Как было показано в V.6, существуют в основном два возможных представления для таких линейных списков: сплошное и цепное представления. Первое использует массив t и индикатор n, инициализируемый нулем при создании таблицы и представляющий число имеющихся элементов:

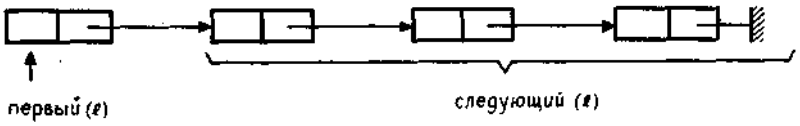
```
переменная n : ЦЕЛОЕ;
массив t[1 : N] : ЭЛЕМЕНТ
```

где N константа, превосходящая максимальное число элементов, содержащихся (или предполагаемых) в таблице



Второе представление, **цепное** имеет вид

```
тип ЛИНЕЙНЫЙ СПИСОК_элемент = (ПУСТО | НЕПУСТОЙСПИСОК);
тип НЕПУСТОЙСПИСОК = (начало : ЭЛЕМЕНТ;
                      следующий : ЛИНЕЙНЫЙСПИСОК_элемент);
переменная ℓ : ЛИНЕЙНЫЙСПИСОК_элемент
```



Алгоритмы, оперирующие с обоими представлениями (которые были частично рассмотрены в V.6), похожи в случае последовательного просмотра: *инициализация* в зависимости от представления записывается в виде

n← 0 или ℓ ← ПУСТО;

Список *пуст* тогда и только тогда, когда

n = 0 или ℓ = ПУСТО;

Переход к *следующему элементу* (возможно, пустому) записывается как

если i ≤ n **то** i ← i + 1 или **если** x ≠ пусто **то** x ← следующий (x);

Элемент является *последним* в списке, если

i = n или следующий (x) = ПУСТО

Эти «эквивалентности» позволят рассматривать только один вариант для алгоритмов, которые записываются так, что обеспечивается формально перенос на оба представления.

VII.2.2.2. Алгоритмы поиска

Алгоритм поиска требует одного просмотра списка и записывается в нерекурсивной форме (ср. V.6.4)

```
программа поиск : ЭЛЕМЕНТ
  (аргументы  c: КЛЮЧ,
              ℓ : ЛИНЕЙНЫЙСПИСОК_элемент)
  {поиск в последовательной неупорядоченной таблице}
  переменная x : ЛИНЕЙНЫЙСПИСОК_элемент
  x ← ℓ;
  пока x ≠ ПУСТО и ключ (начало (x)) ≠ c повторять
      x ← следующий (x);
  поиск ← ( если x = ПУСТО то ПУСТО
            иначе начало (x))
```

Обратите внимание, что этот алгоритм, как и другие, излагаемые ниже, предполагает, что вторая компонента отношения и не вычисляется, если первая равна значению ложь, как в АЛГОЛе W (III.5.3.1). Заметьте также, что если разрешена многозначность ключей, то алгоритм будет «находить» всегда первый элемент из списка элементов, обладающих требуемым ключом.

Какова эффективность этого алгоритма поиска? Пусть n – число имеющихся элементов. Цикл выполняется n раз (а условие цикла n + 1 раз), если разыскиваемого ключа в списке не существует. Если он есть и находится с равной вероятностью в позициях 1, 2, ..., n, то цикл выполняется в среднем n + 1/2 раз. Максимальная сложность и средняя сложность поиска равны, таким образом, O(n) с коэффициентом, вдвое меньшим для последней. Минимальная сложность равна, очевидно, O(1) (случай, когда разыскиваемый элемент–первый в таблице).

VII.2.2.3. Алгоритм включения. Обсуждение

Включение может записываться (в сплошном представлении) в виде

```
программа включение (аргумент x : ЭЛЕМЕНТ;
                      модифицируемые данные n : ЦЕЛОЕ,
                                  массив t [1 : N] : ЭЛЕМЕНТ)
{включение в последовательную неупорядоченную таблицу без
обязательной однозначности ключей}
если n < N то
    n ← n + 1
    t[n] ← x
  {иначе ошибка: таблица полна}
```

Сложность этого включения равна O(1). Соответствующий алгоритм можно реализовать в цепном представлении при условии сохранения указателя, отмечающего конец списка; однако наиболее естественно при таком представлении, по–видимому, производить операции включения в *голову* списка, получая, таким образом, отношение *стека,* а не *файла* (ср. V.4 и V.5).

Однако если необходимо предусмотреть однозначность ключей, то в обоих случаях надо, чтобы включение предшествовало поиску, при этом сложность алгоритма становится O(n).

Пространственная теоретическая сложность обоих представлений равна O(n); практическая сложность больше для цепного представления, поскольку необходимо сохранять указатель элемента. Цепное представление предпочтительнее в двух следующих случаях (IV.6.2 и V.1.4):

а) верхнюю границу N числа элементов трудно фиксировать, и имеется несколько структур данных, по поводу которых можно предполагать, что они не все станут «полными» одновременно;

б) помимо включений и поисков, необходимо выполнять удаления: сплошное представление мало удобно для этой операции.

Использование неупорядоченной таблицы представляет собой самый простой метод управления данными (можно сказать, что данные практически не управляются); основной его характеристикой является то, что поиск требует времени, пропорционального числу имеющихся элементов; для включения требуется такое же время, если имеет место однозначность ключей.

Этот метод совершенно удовлетворителен, если число n ключей невелико (например, около сотни), если каждый элемент, будучи один раз включенным, не становится объектом слишком большого числа поисков (например, не более 10 в среднем) и если выигрыш времени не является решающим. Если же одно из этих

условий не выполнено, следует переходить к более совершенным методам.

VII.2.3. Упорядоченная последовательная таблица; дихотомический поиск

VII.2.3.1. Последовательный поиск в упорядоченной таблице

Можно несколько улучшить предыдущие алгоритмы, если предположить существование отношения порядка в типе КЛЮЧ, т.е. возможность сравнения двух ключей операторами отношений > , ≥ , < , ≤ (которые практически могут быть представлены подпрограммами, выдающими результат типа ЛОГ). В этом случае можно так управлять таблицей, чтобы сохранять возрастающий порядок ключей в линейном списке (точно так же можно выбрать убывающий порядок); всегда, когда элемент x предшествует элементу y в линейном списке, имеет место ключ(x) ≤ ключ(y).

Использование отношения порядка добавляет некоторое *количество информации,* что совершенно естественно облегчает поиск, но усложняет включение, поскольку необходимо соблюдать имеющийся порядок.

Пусть имеется последовательная таблица, представленная, как и раньше, списком или массивом, но на этот раз упорядоченным. При *поиске* нет необходимости систематически добираться до конца таблицы:

```
программа поиск : ЭЛЕМЕНТ
  (аргументы  c : КЛЮЧ,
              n : ЦЕЛОЕ,
  массив      t [1 : n] : ЭЛЕМЕНТ)
  {последовательный поиск в упорядоченной таблице}
  переменная i : ЦЕЛОЕ;
  i ← 1;
  пока i ≤ n и c > ключ (t[i]) повторять
      i ← i + 1;
  поиск ← (если i ≤ n и ключ (t[i]) = c то t [i]
           иначе ПУСТО)
```

Если ключ c имеется в таблице и если с равной вероятностью он может находиться на 1–м, 2–м, ..., n–м местах, то среднее число сравнении ключей равно $\frac{n+1}{2}$.

При безрезультатном поиске, если принято одно и то же соглашение о вероятностях для случаев: c < ключ (t[l]), что порождает одно сравнение ключей, ключ (t [1]) < c < ключ (t [2]) что порождает два сравнения, ..., ключ (t[n – 1]) < c < ключ(t[n]), что порождает n сравнений, и ключ(t[n]) < c, что также требует n сравнений, получают среднее число проверок цикла, равное

$$\frac{1+2+...+n-1+n+n}{n+1}=\frac{n(n+3)}{2(n+1)}\approx\frac{n}{2}+1$$

Средняя сложность остается, таким образом, равной O(n), но с коэффициентом, вдвое меньшим (по сравнению с неупорядоченной таблицей) при *безрезультатном* поиске. Максимальная сложность остается той же.

VII.2.3.2. Последовательное включение

В противоположность случаю неупорядоченной таблицы *включение* здесь не может выполняться без предварительного поиска; при цепном представлении сохранение указателя, отмечающего конец списка, становится бесполезным. В любом

случае включение будет иметь сложность O(n).

При сплошном представлении первый приходящий на ум метод состоит в том, что прежде всего выполняется поиск для определения позиции i, в которую предстоит сделать включение, а затем n – i последних элементов сдвигаются вправо:

```
программа включение (аргумент x : ЭЛЕМЕНТ;
                      модифицируемое данное n : ЦЕЛОЕ,
                      массив t[l : N] : ЭЛЕМЕНТ)
  {включение в упорядоченный линейный список – не окончательная версия}
  переменные i, j : ЦЕЛЫЕ;
  i ← 1;
  {поиск}
  пока i < n и ключ (x) > ключ (t [i]) повторять
      | i ← i + 1;
  если n < N то
      | n ← n + 1;
      | {сдвиг вправо} для j от n до i + 1 шаг – 1
      | повторять
      |     | t[j] ← t[j – i];
      | t[i] ← x
    {в противном случае ошибка: таблица была полна}
```

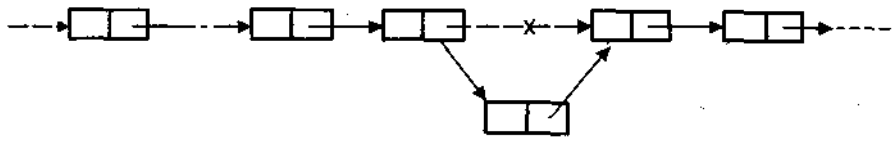
И этой версии программы выполняются не только i сравнений ключей, где i, как и раньше, в среднем равняется примерно n/2 + 1, но, кроме того, еще n – i «сдвигов», т.е. всегда n операций и один полный просмотр n значащих элементов массива. Гораздо интереснее комбинировать сравнения и сдвиги, двигаясь справа, поскольку все элементы с ключами, превосходящими x, могут сдвинуться сами собой:

```
программа включение (аргумент x : ЭЛЕМЕНТ;
                                модифицируемое данное n : ЦЕЛОЕ,
                                массив t[l : N] : ЭЛЕМЕНТ)
  {включение в упорядоченный линейный список}
  если n < N то
      | i ← n;
      | пока i ≥ 1 и ключ(x) < ключ(t[i]) повторять
      |     | t[i + 1] ← t[i];
      |     | i ← i – 1;
      | t[i + l] ← x;
      | n ← n + 1
    {в противном случае ошибка: таблица была полна}
```

Улучшение, достигаемое этим методом, вызвано перегруппировкой двух циклов по i в один, в результате чего просматривается в среднем только половина значащей части таблицы, а также тем, что *минимальная* сложность, получаемая, когда x имеет ключ, превосходящий ключи всех других элементов, равна теперь O(l), а не O(n). Мы используем это свойство для сортировки включением (VII.3.5).

С другой стороны, эта форма программы не позволяет выполнять включение, когда должна быть предусмотрена однозначность ключей. В этом случае всегда требуется полное разделение фаз поиска и модификации таблицы.

Первая версия включения легко переносится на цепное представление: удается избежать сдвигов, потому что добавление нового элемента осуществляется простой манипуляцией указателей:



Вторая версия, при которой просматривается список справа налево, может быть реализована в цепном представлении, только если сохраняются указатели назад (ср. файлы с двойным доступом в V.5.5).

Во всех этих версиях алгоритма средняя сложность включения в упорядоченную таблицу равна O(n) примерно с n/2 сравнениями ключей в среднем для предварительного поиска; собственно включение выполняется с O(l), кроме случая первой версии в сплошном представлении, при котором выполняется около n/2 «сдвигов».

> Заметьте, что в случае, когда допускается многозначность ключей, обе версии рассматривают список либо как стек (включение перед элементами, имеющими одинаковый ключ), либо как файл (включение в хвост). Это может модифицироваться, если необходима другая организация. Алгоритм можно сократить в первой версии при сплошном представлении и в обеих версиях при цепном представлении, если непосредственно предшествующий «поиск» выдает значение i или ℓ (соответственно). Стоимость собственно включения в таком случае остается O(n) при сплошном представлении (из–за сдвигов), но становится O(l) при цепном представлении.

На первый взгляд упорядоченная таблица улучшает вдвое безрезультатный поиск, а также включение при цепном представлении, если предусмотрена однозначность ключей. Напротив, сложность включения становится равной O(n) во всех случаях, и включение может оказаться очень неэффективным, если перемещения элементов дороже сравнений.

Мы не добились еще молниеносного улучшения алгоритма. Тем не менее при сплошном представлении можно получить весьма эффективный алгоритм *поиска* в упорядоченной таблице; метод состоит в использовании (наконец!) свойств прямого доступа к элементам массива, а не в их последовательном просмотре. Этот метод, называемый *дихотомическим поиском,* дает алгоритм поиска со сложностью O(logn), но не решает проблему включения.

VII.2.3.3. Дихотомический поиск

Этот принцип прост. В основе этого алгоритма лежит рекурсия. Сравнивая разыскиваемый ключ с ключом середины массива, узнают, следует ли его искать в левой половине или в правой. После этого алгоритм рекурсивно применяется к левому или правому полумассиву до получения массива из одного элемента. Интуитивно ясно, что на каждом этапе размер рассматриваемого массива уменьшается примерно вдвое, и, следовательно, число этапов будет равно приблизительно $\log_2 n$.

Алгоритм (который читатель сможет записать в рекурсивной форме) допускает несколько версий. Одна из них имеет следующий вид:

```
программа поиск: ЭЛЕМЕНТ
    (аргументы с: КЛЮЧ,
               n: ЦЕЛОЕ,
               массив t[1 :N] : ЭЛЕМЕНТ)
  {дихотомический поиск в сплошной упорядоченной таблице. Версия А}
  переменные слева, справа, середина : ЦЕЛЫЕ;
  если n < 0 то {замечание: n < 0 это случай ошибки}
      поиск ← ПУСТО
  иначе {n > 0}
      слева ← 1; справа ← n;
      пока слева < справа повторять
          {инвариант цикла:
          а) 1 ≤ слева ≤ справа < n;
          б) для всякого i, такого, что 1 ≤ i < слева, для всякого j, такого, что
             справа ≤ j < n ключ(t [i]) < с ≤ ключ (t [j])}
          середина = ⌊(слева + справа)/2⌋;
          если с ≤ ключ (t[середина]) то
              {двигаться влево}
              справа ← середина
          иначе {с > ключ (t[середина])}
              {двигаться вправо}
              слева ← середина + 1;
      поиск ← (если ключ (t [слева]) ≠ с то ПУСТО
                иначе t [слева])
```

Эту форму алгоритма мы будем называть версией А. Часто пробуют написать и версию В, в которой цикл завершается, если ключ(t([середина]) = с, уходя таким образом от необходимости проводить до конца все сравнения. Мы покажем немного ниже, что эта версия В на самом деле не интересна.

Прежде чем детально анализировать причины, по которым этот алгоритм корректен, а также его точную эффективность, заметим, что его можно было бы модифицировать сразу же после постановки вопроса: Между какими индексами упорядоченной таблицы t располагается ключ с? (Это уже **интерполяция**, а не только поиск.) В таком приложении, однако, надо инициализировать переменную справа значением n + 1, а инвариант слегка изменить, чтобы учитывать существование n + 1 возможных интервалов. Интервал решения задается на выходе из цикла в виде

$$]\text{слева}-1,\ \text{слева}]$$

с условием $1 \le \text{слева} \le n + 1$ и очевидными интерпретациями: $\text{слева} = 1$ – с меньше всех ключей таблицы и $\text{слева} = n + 1$ – с больше всех ключей таблицы.

Попытка упрощения версии А – достаточно тонкая задача, и ошибки в деталях могут сделать алгоритм неправильным. Поэтому небесполезно поискать доказательство правильности алгоритма, тем более что оно дает прекрасную иллюстрацию аксиоматике Хоара (III.4).

Случай $n = 0$ очевиден; для установления правильности алгоритма достаточно показать, что цикл завершается, что предлагаемый «инвариант» изначально верен и что выполнениетшкла оставляет его инвариантом. Если это так, то, действительно, на выходе из цикла имеют в силу самого определения инварианта

$$\text{слева} \ge \text{справа}$$

и

$$1 \le \text{слева} \le \text{справа} \le n$$

для всякого i, такого, что 1 ≤ i < слева,

для всякого j, такого, что справа ≤ j < n,

ключ (t [i]) < c ≤ ключ (t [j])

поэтому слева = справа, и, поскольку таблица упорядочена, ключ существует в таблице тогда и только тогда, когда c = ключ (t [слева]) = ключ (t [справа]), как это видно отдельно в случаях слева < n и слева = n.

Инвариант тривиально верен изначально, поскольку предполагается, что n > 0 и что множества {1 ≤ i< 1} и {n ≤ j < n} пусты (напомним, что предложение типа “для всякого x, принадлежащего E, свойство P(x) верно” всегда имеет значение истина, если E – пустое множество). Заметьте, что минимальная модификация инварианта, например замена < на ≤, делает это свойство неверным.

Часть 1 ≤ слева ≤ справа ≤ n инварианта остается всегда верной при выполнении тела цикла, так как

1 ≤ слева ≤ справа ≤ n

⇒ 1 ≤ [(слева + справа)/2] < [(слева + справа)/2] + 1 ≤ n

Поэтому достаточно показать, что оставшаяся часть инварианта, которую мы обозначим (I), является инвариантом, если слева < справа:

(I) для всякого i, такого, что 1 ^ i < слева,
J для всякого j, такого, что справа ^ j < n,
ключ (t[i]) < c ≤ ключ (t[j])

В силу характеристических свойств альтернативы (III.4.3) достаточно доказать

а) {I и c ≤ ключ (t[середина])} справа ← середина{I}

и б) {I и c > ключ (t[середина])} слева ← середина + 1{I}

В силу характеристических свойств присваивания (III.4.1) доказательство а) и б) сводится к доказательству

а') {I и c ≤ ключ (t[середина])} ⇒ I [середина → справа]

и б') {I и c > ключ (t[середина])} ⇒ I [середина + 1 → слева]

Напомним, что нотация P[e → x] означает «свойство P при условии, что все вхождения переменной x заменены выражением e».

Принимая во внимание обозначение (I) и воспользовавшись тем фактом, что середина сама по себе является объектом присваивания, достаточно доказать а") и б"):

а") *если* для всякого i, такого, что 1 ≤ i < слева,
и для всякого j, такого, что справа ≤ j < n, (*начальное условие* (I))
ключ (t[i]) < c ≤ ключ (t[j])

и если

$$c \le \text{ключ}\left(t\left[\left\lfloor \frac{\text{слева}+\text{справа}}{2} \right\rfloor\right]\right)$$

то

для всякого i, такого, что 1 ≤ i < слева,

и для всякою j. такого, что $\left\lfloor \frac{\text{слева} + \text{справа}}{2} \right\rfloor \le j < n$

ключ (t[i]) < c ≤ ключ (t[j])

Это вытекает из упорядоченности таблицы; так как для рассматриваемых j

$$1 \le j < \text{слева} \le \left\lfloor \frac{\text{слева} + \text{справа}}{2} \right\rfloor < \text{справа} \le j < n$$

то получают

$$\text{ключ}(t[i]) < \text{ключ}\left(t\left\lfloor\left\lfloor\frac{\text{слева} + \text{справа}}{2}\right\rfloor\right\rfloor\right) \le \text{ключ}(t[j])$$

б") *если* I (то же начальное свойство)

и если c > ключ (t[середина])

то для всякого i, такого, что $1 \le i < \text{середина} + 1$,

и для всякого j, такого, что $\text{справа} \le j \le n$,

ключ (t[i]) < c ^ ключ (t[j])

Правое неравенство сохраняется из начального условия (I). Левое неравенство вытекает из того, что $i < \text{середина} + 1$ можно записать как $i \le \text{середина}$. Так как таблица упорядочена, то для рассматриваемых i получают

$$\text{ключ}(t[i]) \le \text{ключ}(t[\text{середина}]) < c$$

Чтобы доказать, что цикл завершается, мы докажем, что величина

$$\text{расст} = \text{справа} - \text{слева}$$

есть «управляющая величина» цикла (III.4.4), т.е. что $\text{расст} \ge 0$ является инвариантом цикла (расст остается неотрицательным), и что расст строго возрастает при каждом выполнении цикла. $\text{расст} \ge 0$ переписывается в виде $\text{слева} \le \text{справа}$, что образует часть инварианта цикла; с другой стороны, строгое возрастание расст обеспечивается отношениями

$$1 \le \text{слева} < \text{справа} \Rightarrow \begin{cases} \text{справа} - \left\lfloor\frac{\text{слева} + \text{справа}}{2}\right\rfloor \ge 1 \\ \left(\left\lfloor\frac{\text{слева} + \text{справа}}{2}\right\rfloor + 1\right) - \text{слева} \ge 1 \end{cases}$$

Действительно, можно утверждать, что при каждом выполнении цикла значение d величины расст заменяется на d', такое, что

$$\begin{cases} d' = \left\lfloor\frac{d}{2}\right\rfloor \\ \text{или} \\ d' = \left\lfloor\frac{d}{2}\right\rfloor - 1 \end{cases} \qquad (1)$$

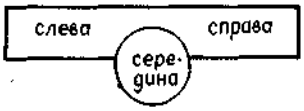
Это завершает доказательство правильности алгоритма.

Интересно отметить здесь, в какой мере определяющая роль некоторых деталей по–настоящему становится ясной, только когда полностью описано доказательство. Уже отмечалась важность выбора символов $<$, $>$, $\le$, $\ge$ в инварианте. Если бы присваивание $\text{слева} \leftarrow \text{середина} + 1$ было заменено на $\text{слева} \leftarrow \text{середина}$ (случай, при котором $c < \text{ключ}(t[\text{середина}])$), было бы невозможно доказать, что расст – это управляющая величина, и в результате программа зацикливалась бы. Если бы мы желали «симметризовать» алгоритм, заменяя $\text{справа} \leftarrow \text{середина}$ на $\text{справа} \leftarrow \text{середина} - 1$ (обратный случай), I перестало бы быть инвариантом, и тогда программа дала бы в некоторых случаях результат ПУСТО, хотя ключ и существует. Эти детали обнаруживаются именно при доказательстве. Это объясняет, почему для многих первая попытка написать программу дихотомического поиска оказывается неудачной.

Мы уже отмечали (и последнее отношение (1) это подтверждает), что сложность дихотомического поиска равна $O(\log n)$ при числе сравнений ключей, равном примерно $\log_2 n$. Интересно продолжить его анализ немного дальше и поискать точное число сравнений; это исследование позволит лучше проникнуть в алгоритм и углубить понимание двоичных деревьев.

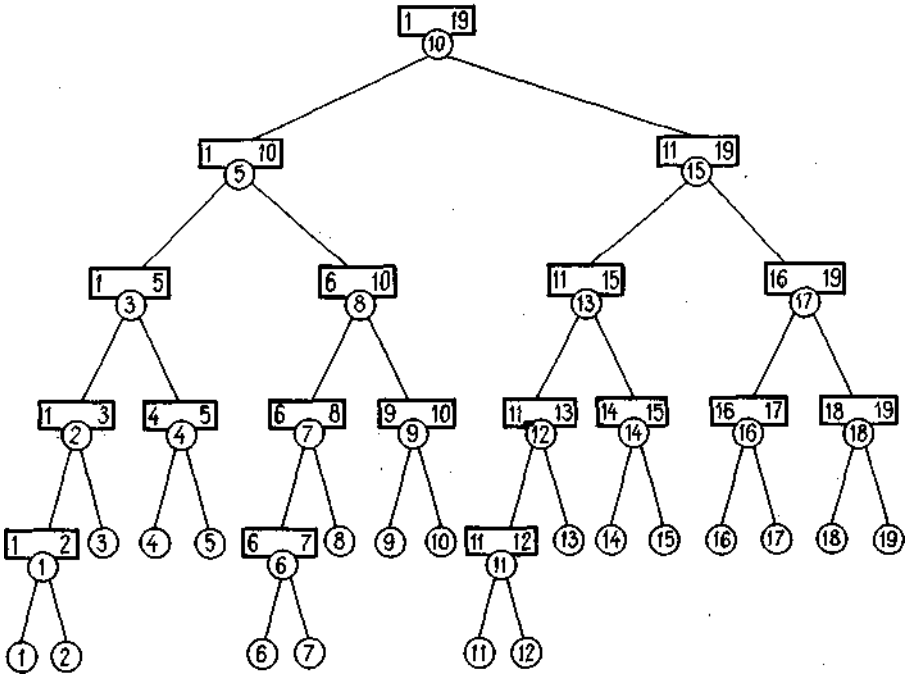
VII.2.3.4. Практическая сложность дихотомического поиска

Дихотомический поиск может быть естественно интерпретирован как движение по двоичному дереву; взятое целиком, двоичное дерево представляет все возможные поиски. Внутренняя вершина двоичного дерева соответствует интервалу [слева, справа], и мы будем ее представлять с помощью рисунка



Два сына этой вершины соответствуют интервалам [слева, середина] и [середина + 1, справа]. Листья получаются при слева = справа и представляются с помощью

Так, ниже изображено двоичное дерево дихотомических поисков для n = 19. Заметим только, что единственная информация, связанная с вершинами, представляет индексы; значения элементов и их ключи не изображены.



Выполнение алгоритма–это прохождение пути, ведущего от корня к некоторому листу; каждый индекс i в кружочке указывает, что надо выполнять сравнение c < ключ(t[i]).

Всегда можно положить

$$n = 2^k + \ell$$

при

$k = [\log_2 n]$ (k = 4 в нашем примере)

и

$0 \le \ell \le 2^k - 1$ (ℓ = 3 нашем примере)

Глубина[1] двоичного дерева (V.7.2) равна k + 1, если $\ell = 0$, и k + 2 в противном случае. Ясно, что здесь дерево полно до глубины k + 1 включительно. Существует, таким образом, некоторое число листьев, например f_{k+1}, на глубине k + 1 и, возможно, другая часть листьев, например f_{k+2}, на глубине k + 2. Видно, впрочем, что с каждым

[1] Авторы постоянно смешивают термины «глубина» (profondeur) и «высота» (hauteur) дерева, говоря об одном и том же понятии. Не имея оснований переносить это смешение в перевод, мы остановились на термине «глубина», используемом в этом смысле всюду ниже. – *Прим. перев.*

листом связан один и только один индекс, заключенный между 1 и n; следовательно,

$$f_{k+1} + f_{k+2} = n \qquad (1)$$

Полное двоичное дерево глубины k + 1 имеет 2^k листьев, как это непосредственно видно из рекуррентости по k. Среди 2^k вершин глубины k + 1 в нашем дереве существует f_{k+1} листьев, а каждая из $2^k - f_{k+1}$ остальных вершин имеет в точности двоих сыновей, которые образуют f_{k+2} листьев глубины k + 2. Имеют, следовательно,

$$2^k - f_{k+1} = \frac{f_{k+2}}{2} \qquad (2)$$

Из (1) и (2) в силу $n = 2^k + \ell$ выводят, что

$$\begin{cases} f_{k+1} = 2^k - \ell = n - 2\ell \\ f_{k+1} = 2\ell \end{cases}$$

Если предположить равновероятное распределение ключей (результативный поиск) или равновероятное распределение интервалов между ключами (безрезультатный поиск), то алгоритм выполняет k + 1 сравнений с вероятностью f_{k+1}/n и k + 2 сравнений с вероятностью f_{k+2}/n. Таким образом, среднее число сравнений равно

$$C = ((k + 1)(n - 2\ell) + (k + 2) \times 2\ell)/2 = k + 1 + \frac{2\ell}{n}$$

где $k = \lfloor \log_2 n \rfloor$ и $\ell = n - 2^k$.

Итак, алгоритм выполняет примерно $\log_2 n + 1$ сравнений; член + 1 соответствует последнему сравнению, которое определяет результат программы поиск.

Выше упоминалась версия B алгоритма, позволяющая уменьшить в некоторых случаях число выполнений цикла. Она записывается в виде

```
переменные слева, справа, середина: ЦЕЛЫЕ;
    переменные есть : ЛОГ;
слева ← 1; справа ← n + 1, есть ← ложь;
пока ~ есть и слева ≤ справа повторять
        середина ← ⌊(слева + справа)/2⌋;
        если ключ (t[середина]) > c то
            справа ← середина – 1
        иначе если (ключ t[середина]) < c то
            слева ← середина + 1
        иначе есть ← истина;
поиск ← (если есть то t[середина]
        иначе ПУСТО)
```

(*Упражнение:* Напишите инвариант цикла для этой версии и докажите ее правильность.)

Двоичное дерево, соответствующее этой версии, слегка отличается от предыдущего: интервалы, соответствующие левым сыновьям, меньше. Используя, однако, предыдущее дерево в качестве приблизительной модели, можно сказать, что:

- в случае безрезультатного поиска проходится тот же путь; но здесь в среднем одно выполнение цикла из двух осуществляет два сравнения ключей; число сравнений, таким образом, не превосходит 50%[1];

[1] На практике второе сравнение может оказаться более дорогим, чем первое, если используется, например, подпрограмма, выдающая 3 возможных значения в зависимости от того, что один ключ меньше, равен или больше другого.

- в случае результативного поиска при предыдущей гипотезе равновероятности путь будет иметь длину 1 с вероятностью 1/n (корень), длины 2 с вероятностью 2/n (вершины глубины 1), длины 3 с вероятностью 4/n и т.д.

Средняя длина вычисляется в виде

$$\frac{1 + 2 \times 2^1 + 3 \times 2^2 + \ldots + k \times 2^{k-1} + (k + 1) \times f_{k+2}}{n}$$

(здесь нет сравнений, выполняемых на листьях). Это дает

$$k - 1 + \frac{2\ell + k}{n}$$

Но не надо забывать, что здесь имеют место в среднем полтора сравнения на каждую вершину. Таким образом, получают в среднем примерно $1{,}5(\log_2 n - 1)$ сравнений, т.е. почти на 50% больше, чем раньше; верно, однако, и то, что в среднем экономится от 1 до двух выполнений цикла. Итак, «улучшения», получаемые версией B, сомнительны, если только стоимость сравнений ключей не является действительно пренебрежимо малой.

VII.2.3.5. Обсуждение и заключение

Дихотомический поиск (версия A) благодаря своему логарифмическому характеру имеет хорошую эффективность для достаточно больших таблиц:

n	Минимальное число сравнений ключей	Среднее число сравнений ключей	Максимальное число сравнений ключей
10	4	4,4	5
100	7	7,72	8
1000	10	10,97	11
10000	14	14,36	15

Важно, однако, еще раз отметить, что метод **не обобщается на включение.** После того как предварительным поиском обнаружено подходящее место, для выполнения включения нужно физически подвинуть n/2 элементов. Таким образом, включение требует сложности $O(n/2) + O(\log_2 n)$, т.е. $O(n)$ (для малых значений n ситуация может быть улучшена с помощью операторов группового перемещения).

Итак, вопреки распространенному мнению *дихотомия в последовательной упорядоченной таблице не дает, вообще говоря, хорошего алгоритма управления таблицей.* Однако она очень интересна в одном частном случае, когда «жизнь» таблицы состоит из двух фаз: фазы заполнения, когда выполняются только включения без поиска (или с малым числом поисков), и фазы эксплуатации, когда выполняются только поиски с возможной корректировкой свойств, не относящихся к ключам, без новых включений (или с малым числом их). Типичным примером может служить файл персонала предприятия с малой текучестью кадров. В этом случае возможна такая процедура:

- в ходе первой фазы обрабатываются данные в сплошной не– упорядочной таблице;
- сортировка таблицы; в VII.3 мы познакомимся с алгоритмами,

позволяющими сделать это за O(nlogn);

- на второй фазе обрабатывается упорядоченная таблица с помощью дихотомических поисков и, если необходимо, небольшого числа включений общей сложности O(n).

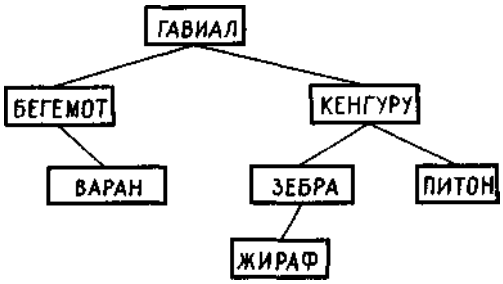
Чтобы получить метод, позволяющий также легко выполнять включение, как выполняется поиск при дихотомии, надо перешагнуть некоторый «концептуальный порог» и отказаться от такой структуры данных, как «массив».

VII.2.4. Двоичное дерево поиска

VII.2.4.1. Метод

Структура данных, называемая «двоичным деревом поиска», которую мы видели в V.7.5, позволяет, отказавшись от свойства прямого доступа, распространить на включение сложность $O(\log_2 n)$, которая была целью дихотомического поиска. Мы видели к тому же, что дихотомический поиск состоит в рассмотрении упорядоченного массива как двоичного дерева.

Напомним, что двоичное дерево поиска–это такое двоичное дерево, с каждой вершиной которого связан элемент (типа ЭЛЕМЕНТ хранимых данных) так, что *ключ* элемента, соответствующего некоторой вершине, не меньше ключей элементов, принадлежащих левому поддереву этой вершины, и меньше ключей элементов правого поддерева. Пример такого двоичного дерева, элементы которого совпадают с их ключами и являются строками ("БЕГЕМОТ", "ГАВИАЛ" и т.д.), а рассматриваемый порядок алфавитный, показан на рисунке:



Конечно, для одного и того же множества элементов существует не единственное двоичное дерево поиска. Используем определение ,типа

тип ДВОДЕР = (ПУСТО | ДВОДЕРНП)
{«НП»–сокращение слов «не пустое»}
тип ДВОДЕРНП = (корень: ЭЛЕМЕНТ; слева: ДВОДЕР; справа: ДВОДЕР)

и дальнейшее будет в равной степени применимо как для цепного, так и для сплошного представления. Напомним, что сплошное представление, близкое к относительно полным деревьям, размещает сыновей элемента с индексом i по местам, соответствующим индексам 2i и 2i + 1.

Процедуры включения и поиска, сохраняющие характеристическое свойство двоичного дерева поиска, были даны в рекурсивной форме в гл. V. В нерекурсивной форме они записываются так:

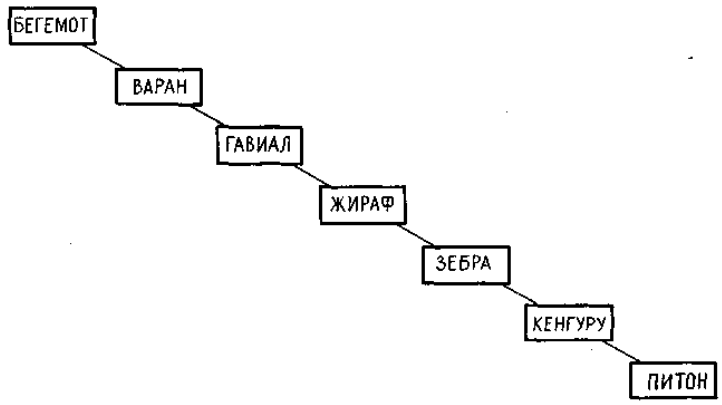
```
программа включение (аргумент x: ЭЛЕМЕНТ;
                      модифицируемое данное a: ДВОДЕР)
{включение в двоичное дерево поиска, нерекурсивная версия}
переменные отец, a': ДВОДЕР;
если a = ПУСТО то
    a ← ДВОДЕРНП (x, ПУСТО, ПУСТО)
иначе a'<~a;
    повторять
        отец ← a';
        если ключ (x) ≤ ключ (корень (a')) то
            a' ← слева (a')
        иначе
            a' ← справа (a)
    до a' = ПУСТО
    если ключ (x) < ключ (корень (отец)) то
        слева (отец) ← ДВОДЕРНП (x, ПУСТО, ПУСТО)
    иначе
        справа (отец) ← ДВОДЕР (x, ПУСТО, ПУСТО)
программа поиск: ЭЛЕМЕНТ
          (аргумент c: КЛЮЧ, a: ДВОДЕР)
{поиск в двоичном дереве поиска, нерекурсивная версия}
пока a ≠ ПУСТО и c ≠ ключ (корень (a)) повторять
    если c ≤ ключ (корень (a)) то
        a ← слева (a)
    иначе
        a ← справа (a)
поиск ← (если a = ПУСТО то ПУСТО иначе корень (a))
```

Поскольку однозначность ключей не учитывается, включение выполняется всегда слева от элементов с ключом, равным ключу вершины.

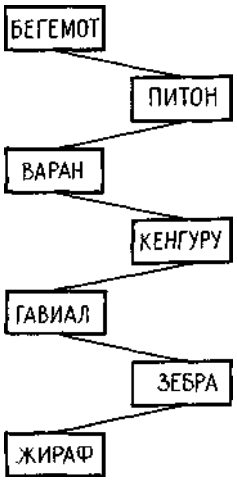
Заметим, что включение легко упростить, если предварительный безрезультатный поиск дает механизм доступа к вершине отца создаваемого элемента: тогда нет необходимости проходить все дерево снова.

Двоичные деревья достаточно хорошо соответствуют операции «удаления»; следует, однако, обращать внимание на то, чтобы не слишком нарушалось «равновесие» дерева по причинам, изучаемым ниже.

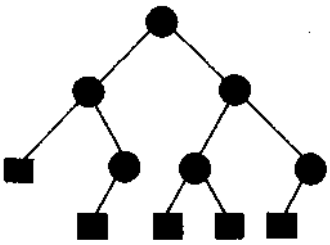
Какова эффективность этих алгоритмов? Она существенно зависит от формы дерева, которая в свою очередь определяется порядком последовательных включений. Если включения выполнены в порядке, заданном ключами



или в обратном порядке, или в порядке «первый–последний–второй–предпоследний–третий и т.д.»:



то ясно, что двоичное дерево ведет себя как линейный список – O(n) с потерей пространства. Двоичное дерево представляет интерес, только если оно относительно «полно», т.е. если оба поддерева всякой вершины почти *равновесны* в смысле, который мы намерены уточнить, но который интуитивно ясен:



(Заметим, что этот критерий близок к тому, который делает интересным сплошное представление.) В таком случае каждый этап алгоритма поиска или включения делит примерно надвое размер двоичного дерева, которое еще остается просмотреть. Более точно, если n – число имеющихся элементов, а T(n) – время выполнения одного поиска или одного включения, то

$$T(n) \approx O(1) + T\left(\frac{n}{2}\right)$$

Член O(l) соответствует выполнению цикла и сравнению ключа с ключ (корень (a)). Эта формула приводит к

$$T(n) = O(\log_2 n)$$

в равной мере как для включения, так и для поиска. Это представляет интерес, как было показано в VII.1: «Разделяй и властвуй» Как можно обеспечить эту благоприятную ситуацию? Достаточно легко показать, что катастрофические ситуации, о которых мы говорили, имеют малую вероятность среди всех возможных двоичных деревьев поиска на множестве данных элементов, если все n! возможных включений рассматриваются равновероятными. Более того, Кнут [Кнут 73] доказал, что при этой гипотезе равновероятности средняя глубина двоичного дерева поиска, т.е. число

этапов поиска, равняется $\sqrt{n}$. Если порядок включений случайный, то нужно в среднем $1{,}386\log_2 n$ сравнений на один поиск.

Это свойство может быть обеспечено, если необходимо работать с не слишком большой таблицей, например в несколько сот элементов. Однако может быть опасным слишком доверять этому свойству при очень больших объемах, потому что на практике множества обрабатываемых ключей, например в АСУ, имеют часто особые свойства,

которые делают их совсем не случайными. Так, ключи могут быть частично отсортированными.

VII.2.4.2. Равновесные двоичные деревья

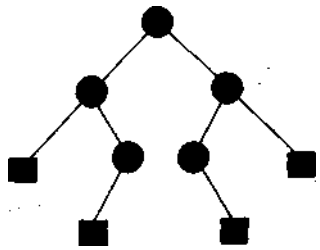
Для такой ситуации представляются плодотворными идеи очень интересного метода, предложенного Адельсоном–Вельским и Ландисом. Он состоит в таком управлении всеми включениями (и возможными удалениями), которое гарантирует, что дерево все время остается в некотором особом классе, называемом *равновесными двоичными деревьями* или *деревьями АВЛ,* обеспечивающем сложность, несколько превосходящую O(logn). Интерес этого метода в том, что само по себе управление никогда не требует сложности больше, чем O(logn) на включение; без этого улучшение было бы неочевидным.

Ниже будет представлена упрощенная рекурсивная версия метода АВЛ. Затем мы улучшим его эффективность одновременно с исключением рекурсии.

Напомним, что **глубина** двоичного дерева–это число вершин на самом длинном пути от корня дерева к листьям. Ее также можно определить рекурсивно:

- глубина пустого дерева есть 0;
- глубина непустого дерева равна h' + 1, где h' – максимум *глубин* двух его поддеревьев.

Например, глубина такого дерева.

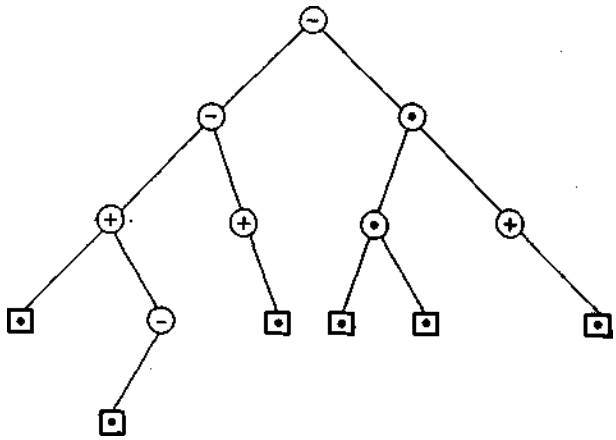


равна 4.

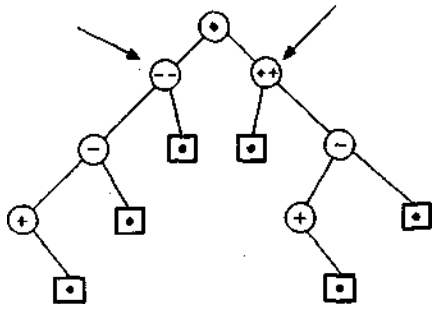
Заметьте, что глубину двоичного дерева можно вычислить с помощью программы

```
программа глубина : ЦЕЛ (данное a : ДВОДЕР)
    переменные hл, hп: ЦЕЛЫЕ;
    если a = ПУСТО то глубина ← 0
    иначе
        hл ← глубина (слева [a]);
        hп ← глубина (справа [a]);
        глубина  ← 1 + max (hл, hп)
```

Будем называть *равновесным двоичным деревом* или *деревом АВЛ* такое двоичное дерево, у которого глубины поддеревьев для каждой вершины отличаются не более чем на 1. Пример равновесного двоичного дерева:



и пример неравновесного дерева



О вершине равновесного дерева можно говорить, что она «устойчива», «нагружена слева», «нагружена справа». Это условно отмечается на схемах значками •, – или + соответственно. В ходе включения вершина может быть временно «выведена из равновесия» (+ + или --).

Прежде чем показать, как в двоичном дереве сохраняется его «равновесие», посмотрим, чего можно добиться этим методом. Равновесные двоичные деревья почти «полны» и должны обеспечивать поведение, «близкое» к $O(\log_2 n)$. Для уточнения этих понятий надо знать максимальную глубину дерева АВЛ, содержащего n элементов; тогда сложность поиска, например, равна O(h max(n)) максимальная сложность).

Проще ответить на обратный вопрос: каково минимальное число элементов чисмин (h) в дереве АВЛ глубиной h? Действительно, ясно, что

чисмин (0) = 0

чисмин (1) = 1

чисмин (2) = 2

а для h > 1:

чисмин (h) = 1 + чисмин (h – 1) + чисмин (h – 2)

На самом деле, дерево АВЛ глубины h должно иметь по крайней мере одно поддерево глубины h – 1, минимальная глубина другого h – 2.

Легко видеть, что

чисмин (h) = $фиб_{h+3} - 1$

где $фиб_h$ есть элемент последовательности, определяемой

$фиб_1 = 0$, $фиб_2 = 1$

для m > 2

$фиб_m = фиб_{m-1} + фиб_{m-2}$

Речь идет о хорошо известной последовательности чисел Фибоначчи, относительно которой доказано, что

$$\text{фиб}_m = \left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^m \right\rfloor$$

следовательно,

$$\text{чисмин}(h) = \left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} \right\rfloor - 1$$

Соответственно максимальная глубина h max (n) дерева АВЛ, содержащего n элементов, есть наибольшее h, такое, что

$$\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \le n$$

т.е.

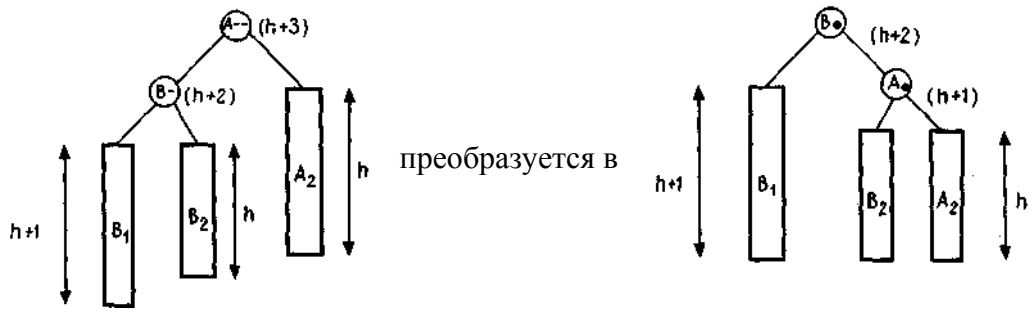
$$h \le \log_{\frac{1+\sqrt{5}}{2}} \left(\sqrt{5} \times (n+1) \right) - 3$$

что дает h max(n) меньше $1{,}441\log_2(n+1) - 1{,}33$. Следовательно, поиск по равновесному двоичному дереву имеет *максимальную* теоретическую сложность O(logn), что и требовалось доказать. Кроме того, максимальная практическая сложность превосходит меньше чем на 50% сложность поиска в оптимальном двоичном дереве. Так как двоичное дерево глубины h при сплошном представлении требует объема памяти, пропорционального $2^h - 1$, мы доказали попутно, что для дерева АВЛ из n элементов потребуется при сплошном представлении максимальный объем $O(n^{1{,}44})$.

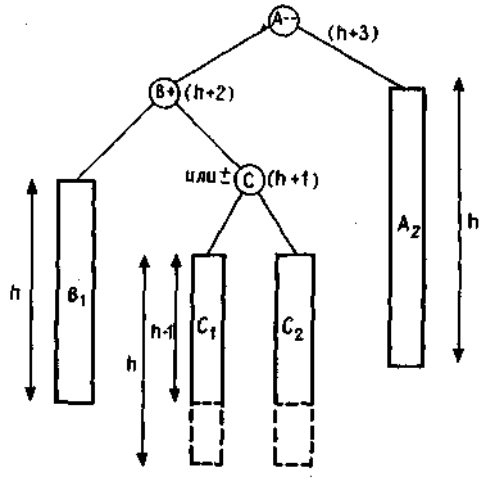
Посмотрим теперь, как удается «поддерживать равновесие» при включении, не превосходя $O(\log_2 n)$.

Проблема возникает только в случае, когда включение приводит к нарушению равновесия. Как это установить? Рассмотрим вершину A, которая перед включением была «равновесной», т.е. «устойчивой» или «нагруженной» слева или справа. Пусть включение приводит к нарушению равновесия в этой вершине: одно из ее поддеревьев имеет глубину h + 2, тогда как другое–глубину h. Однако эти два дерева сами по себе остаются равновесными.

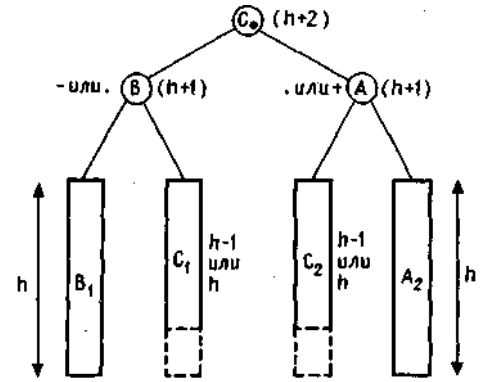
В этом случае можно восстановить равновесие дерева с корнем A путем локальных перестановок. Предположим, что более «тяжелое» поддерево дерева A есть его левое поддерево с корнем B; противоположный случай симметричен. Существует две возможности: если нарушение равновесия вызвано левым поддеревом левого поддерева A, то



Если же нарушение равновесия вызвано правым поддеревом левого поддерева A, то



где одно из двух поддеревьев C_1 и C_2 имеет глубину h, а другое – h – 1[1], преобразуется в



Важной характеристикой этих преобразований является то, что во всех случаях результирующее дерево имеет ту же глубину h + 2, что и до включения. Отсюда вытекает свойство:

(P) Если включение изменяет глубину дерева АВЛ, то оно не требует восстановления равновесия; и, наоборот, если необходимо восстановление, то включение не меняет глубину дерева

Метод, в общих чертах описанный выше, дает алгоритм

```
программа выравнивание модифицируемое данное a : ДВОДЕР)
    {восстановление двоичного дерева a, равновесие которого могло быть
    нарушено включением; дерево a предполагается не пустым}
    если глубина (слева (a)) > глубина (справа (a)) + 1 то
        если глубина (слева (слева (a))) > глубина (справа (слева (a))) то
            {первый из рассмотренных выше случаев}
            a ← ДВОДЕР (корень (слева (a)), слева (слева (a)),
                        ДВОДЕРНП (корень (a), справа (слева (a)), справа (a))
        иначе {необходимо глубина (справа (слева (a))) >
                глубина (слева (слева (a)))}
            {второй из рассматриваемых выше случаев}
            a ← ДВОДЕРНП (корень справа (слева (a))),
                        ДВОДЕРНП (корень (слева (a)),
                                  слева (слева (a)),
```

[1] Оба они не могут иметь глубину h, поскольку дерево с корнем A было бы тогда равновесным еще до включения.

```
                                        слева (справа (слева (а)))),
                              ДВОДЕРНП (корень (а),
                                        справа (справа (слева (а))),
                                        справа (а)
                                        )
                    )
    иначе если глубина (справа (а)) > глубина (слева (а)) + 1 то
    {симметричная ситуация}
      если глубина (справа (справа (а))) > глубина (слева (справа (а))) то
          а ← ДВОДЕРНП (корень (справа (а)),
                              ДВОДЕРНП (корень (а), слева (а),
                                        слева (справа (а))),
                                        справа (справа (а))
                                        ),
      иначе {необходимо глубина (слева (справа (а))) >
                                 глубина (справа (справа (а)))}
          а ← ДВОДЕРНП (корень (слева (справа (а))),
                              ДВОДЕРНП (корень (а),
                                        слева (а),
                                        слева (слева (справа (а)))
                                        ),
                              ДВОДЕРНП (корень (справа (а)),
                                        справа (слева (справа (а))),
                                        справа (справа (а))
                                        )
                    )
    {в противном случае дерево равновесно}
```

Мы описали локальные модификации структуры путем неявного создания новых вершин в форме

а ← ДВОДЕРНП (корень, слева, справа)

Ясно, что на уровне непосредственного выполнения тот же результат может быть достигнут путем работы с указателями (цепное представление) или индексами (сплошное представление), используя локальные переменные.

Включение можно теперь записывать в рекурсивной форме:

```
программа включение (аргумент х: ЭЛЕМЕНТ;
                      модифицируемое данное а : ДВОДЕР)
    {включение, предусматривающее равновесие в двоичном дереве поиска; а
    предполагается равновесным}
    {рекурсивная версия}
    если а = ПУСТО то
        а ← ДВОДЕРНП (х, ПУСТО, ПУСТО)
    иначе
        если ключ (х) < ключ (корень (а)) то
            включение (х, слева (а))
        иначе
            включение (х, справа (а));
            {здесь благодаря рекурсивному применению инварианта слева (а)
            и справа (а) равновесны}
            выравнивание (а) {выравнивание на уровне корня}
      {а равновесно}
```

В этой программе достаточно легко исключить рекурсию. Пусть F – лист, слева или справа от которого делается включение; дадим имя первый–неустойчивый–предок **ближайшему предку** F, «нагруженному» со стороны F, но такому, что никакая

вершина на пути от первого–неустойчивого–предка до F не нагружена со стороны, противоположной F. (Если никакой из предков F не обладает таким свойством, то в качестве первого–неустойчивого–предка выбирается сама вершина F.) В силу отмеченного выше свойства (Р), если необходимо восстановление равновесия, то это может быть только в единственной вершине, которая и есть первый–неустойчивый–предок. Действительно, все вершины между этой вершиной и F были бы с необходимостью устойчивыми: включение F не может, таким образом, вывести их из равновесия. В силу свойства (Р) между корнем и первым–неустойчивым–предком равновесие не нарушается.

Таким образом, в ходе спуска по дереву необходимо сохранять средство доступа к вершине первый–неустойчивый–предок.

Для улучшения подпрограммы включение заметим также, что глубина поддерева не обязательно должна каждый раз перевычисляться; она может быть значением поля, связанного с корневой вершиной этого поддерева. Если принимается такое решение, то можно заменить объявление ДВОДЕРНП:

```
тип ДВОДЕРНП = (корень: ЭЛЕМЕНТ; слева: ДВОДЕР; справа: ДВОДЕР;
                глубина: ЦЕЛОЕ)
```

В таком случае нужно, чтобы выравнивание правильно инициализировало поле глубина различных обрабатываемых вершин; эта простая модификация (в которой следует использовать свойство (Р)) оставлена читателю.

То же самое свойство (Р) позволяет осуществлять простую корректировку полей глубина в программе включение без использования стека в нерекурсивной версии; меняется только, увеличиваясь на 1, глубина вершин, принадлежащих пути, ведущему от вершины первый–неустойчивый–предок (не включая ее) к листу F, в котором выполяется включение (эта вершина становится корнем «нагруженного» поддерева глубины 2).

Таким образом, в нерекурсивной форме программа включения записывается в виде

```
программа включение (аргумент x: ЭЛЕМЕНТ;
                      модифицируемое данное a:ДВОДЕР
                      {новая модель})
  {включение, предусматривающее равновесие в двоичном дереве поиска; a
  предполагается равновесным}
  {нерекурсивная версия}
  переменные первый–неустойчивый–предок, отец, b : ДВОДЕР;
  если a = ПУСТО то
      a ← ДВОДЕРНП (x, ПУСТО, ПУСТО, 1)
  иначе
      b ← a; первый–неустойчивый–предок ← ПУСТО;
      повторять
          отец ← b;
          если ключ (x) ≤ ключ (корень) то
          {движение влево}
              если глубина (слева (b)) > глубина(справа (b)) то
                  первый–неустойчивый–предок ← b
              иначе если глубина (справа (b)) > глубина (слева (b)) то
                  первый–неустойчивый–предок ← ПУСТО;
              b ← слева (b)
      до b = ПУСТО;
      если ключ (x) < ключ (корень (отец)) то
          слева (отец) ← ДВОДЕРНП (x, ПУСТО, ПУСТО, 1)
```

```
        иначе
            справа (отец) ← ДВОДЕРНП (x, ПУСТО, ПУСТО, 1)
        если первый–неустойчивый–предок = ПУСТО то
            первый–неустойчивый–предок ← отец;
        {скорректировать глубины, соответствующие вершинам между
        «первым–неустойчивым–предком» и новым листом}
        b ← первый–неустойчивый–предок;
        повторять
            если b ≠ ПУСТО то
                глубина (b) ← глубина (b) + 1;
            b ← (если ключ (x) ≤ ключ (корень b)) то слева (b)
                иначе справа (b))
        до b = ПУСТО;
        выравнивание (первый–неустойчивый–предок)
        {a равновесно}
```

Ясно, что эта процедура включения требует более двух проходов пути, ведущего от корня до созданного листа; ее сложность равна, следовательно, $O(\log_2 n)$. Если экономия места в памяти является решающим критерием, можно заменить поле глубина полем равновесие, которое принимает значение 0, –1 или + 1 в зависимости от того, является ли вершина устойчивой, нагруженной слева или справа. Для представления этого значения достаточно двух битов. Легко получаются соответствующие модификации программ выравнивание и включение. Доказывается, что на равновесных деревьях операцию удаление также можно реализовать со сложностью $O(\log_2 n)$.

Равновесные деревья интересны для достаточно больших n (например, $n \geq 50$), когда есть основания считать случайным порядок включения ключей и когда отношение числа поисков к числу элементов достаточно велико (например, больше или равно 5 в среднем), чтобы оправдать дополнительную работу, порождаемую операциями восстановления равновесия.

> Интересным обобщением представления с помощью двоичных деревьев поиска является использование «B–деревьев», где число сыновей в вершинах не фиксировано, а содержится между граничными значениями m и 2m. Можно сделать так, чтобы все листья такого дерева имели одинаковую глубину. Эти деревья полезны, в частности, когда данные не размещаются целиком в оперативной памяти и надо минимизировать число внешних доступов (см. [Кнут 73]).

Таким образом, двоичные деревья дают гибкий и эффективный метод для управления таблицами. Они особенно удобны в задачах, требующих многочисленных включений и поисков, чередующихся заранее неизвестным образом, а также удалений, для которых мы не предложили алгоритма. Эти деревья требуют некоторой предосторожности, если ключи не случайны, – предосторожности, требующей вдвое или втрое большего времени по сравнению с оптимальным выполнением алгоритма, но позволяющей обеспечить характеристику «$O(\log_2 n)$» метода.

VII.2.5. Ассоциативная адресация

VII.2.5.1. Определения и общие сведения

В предыдущих методах выполнялись последовательные включения элементов в определенную позицию по отношению к ранее включенным элементам. В результате время поиска и вообще время включения были функциями n, числа имеющихся элементов.

Идея **ассоциативной адресации**[1] отличается коренным образом. Речь идет об

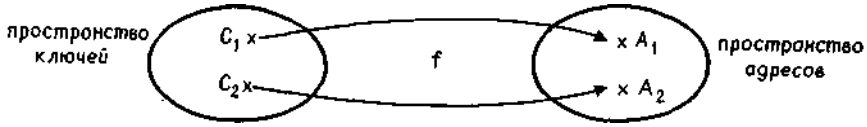
[1] Ассоциативная адресация называется также прямой адресацией, адресацией по значению, адресацией по функции

имитации прямого доступа, который позволяет при обычном представлении получить с помощью индекса элемент массива за некоторое постоянное время. В задачах управления таблицами роль индекса играет ключ, связанный с элементом.

Таким образом, в идеальном случае ассоциативная адресация предполагает, что таблица представлена с помощью массива N элементов, которые более удобно нумеровать от 0 до N – 1:

массив t [0: N – 1] : ЭЛЕМЕНТ

где N – общее число возможных ключей. Предполагается, кроме того, что существует функция f, ставящая в соответствие всякому ключу с некоторое целое i, различное для разных c и такое, что $0 < i \le N - 1$; тогда достаточно разместить элемент, обладающий этим ключом в t [i], и время поиска становится постоянным (O(l)).



К сожалению, эта идеальная схема практически нереализуема. Даже оставляя в стороне проблему различных элементов с одинаковыми ключами, рассмотрение типичного случая показывает, что число N возможных ключей настолько велико, что нельзя и помышлять об использовании массива размером N. Пусть, например, транслятор ФОРТРАНа должен расположить в «таблице символов» все идентификаторы программы. Число возможных идентификаторов ФОРТРАНа, ограниченных 6 литерами, равно

$$26 + (26 \times 36) + (26 \times 36^2) + \ldots + (26 \times 36^5)$$

т.е. более полутора миллиардов, из которых, очевидно, только незначительная часть встретится в данной программе (в АЛГОЛе множество возможных идентификаторов практически бесконечно.). Существует, следовательно, ограничение на практически пред–ставимое «пространство адресов», например величиной N, равной самое большее нескольким тысячам, если таблица должна оставаться в оперативной памяти. Значит, функция f должна давать самое большее N различных значений, где N много меньше числа теоретически возможных ключей. Такая функция, следовательно, будет неизбежно *неиньектиной,* т.е. два различных ключа могут иметь одинаковые значения. Будем говорить, что два ключа C_1 и C_2, таких, что

$$C_1 \neq C_2 \text{ и } f(C_1) = f(C_2)$$

вызывают **коллизию.** Наличие коллизий обязывает сдержанно относиться к использованию «прямого доступа» методами, сходными с теми, которые мы рассматривали раньше. Заметим, что очень близкая ситуация обсуждалась в V.9.4 в связи с представлением *ненасыщенных массивов:* там тоже предпринималась попытка применить прямой доступ к данным, занимающим весьма большое «виртуальное», но очень малое «фактическое» пространство; мы пришли к некоторому компромиссу.

Феномен коллизии приводит к представлению таблицы в виде массива, состоящего не из элементов, как в рассмотренных выше сплошных представлениях, а из линейных списков элементов массив $t[0 : N - 1]$ ЛИНЕЙНЫЙСПИСОК$_{элемент}$

Границы 0 и N – 1 более удобны, чем 1 и N; разумеется, показанные ниже алгоритмы могут быть непосредственно перенесены и в такой язык, как ФОРТРАН, где индексы массивов должны быть положительными.

> В свете того, что было рассказано в VII.2.3 об упорядоченных последовательных таблицах, естественной идеей представляется сохранение *упорядоченности* линейных списков, если существует отношение порядка для ключей: мы видели, действительно, что время безрезультатного поиска, также как и время включения при цепном представлении с однозначностью ключей, делилось примерно пополам. Далее будет рассмотрено применение этой идеи.

распределения, вычисляемой адресацией, H–кодированием, а по–английски hash–coding.

При инициализации таблицы все линейные списки t[i](0 < i < < N – 1) заполняются константой ПУСТО. Алгоритмы поиска и включения схематически записываются как

```
программа поиск: ЭЛЕМЕНТ
        | (аргумент с : КЛЮЧ,
        |   массив t[0:N– 1]: ЛИНЕЙНЫЙСПИСОК элемент)
        |   {поиск в таблице, управляемой ассоциативной адресацией}
        |   переменная индекс: ЦЕЛ;
        |   индекс ← f(c);
        |   поиск ← искать (c, 1[индекс])
программа включение (аргумент x : ЭЛЕМЕНТ;
    модифицируемое данное массив t[0 : N – 1]: ЛИНЕЙНЫЙ–СПИСОК элемент
        |   {включение при ассоциативной адресации}
        |   переменная индекс: ЦЕЛ;
        |   индекс ←f (ключ (x));
        |   включить (x, 1[индекс])
```

Подпрограммы, выполняющие поиск по ключу и включение в линейных списках, названы здесь искать и включить с тем, чтобы не наводить на мысль о рекурсии там, где ее нет. Ясно, что в этом методе включение упрощается благодаря предварительному поиску, определяющему свободное место. Подпрограммы поиск и включение будут использоваться, если списки упорядочены.

Возникают две задачи представления:

а) как лучше выбрать функцию f, называемую функцией нахождения адреса или просто функцией расстановки, чтобы уменьшить долю коллизий?

б) как представить линейный список–в самом массиве или вне его? К этим двум вопросам мы сейчас и обратимся.

VII.2.5.2. Выбор функции расстановки

Функция расстановки f должна иметь значения, заключенные между 0 и N – 1. По этой причине ее часто определяют в форме

$f(c) = \varphi(c)$ **mod** N

где φ – функция, принимающая целые значения. Существует поэтому два вида коллизий: **непосредственные коллизии,** соответствующие ключам C_1 и C_2, таким, что $\varphi(C_1) = \varphi(C_2)$, и **коллизии по** модулю, порождаемые ключами C_1 и C_2, такими, что $\varphi(C_1) \neq \varphi(C_2)$, но $\varphi(C_1) = \varphi(C_2)$(**mod** N).

Мы увидим далее, как можно решить проблему коллизий «по модулю», а сейчас отметим следующее правило [Люм 71]:

Доля коллизий «по модулю» уменьшается, если размером таблицы выбрать целое N, не имеющее делителей, меньших 20; N можно взять, например (но не обязательно), простым.

Чтобы ограничить количество «начальных» коллизий, вызываемых функцией φ, нужно построить функцию, обеспечивающую хорошее начальное распределение адресов. Для этого нет универсального рецепта; можно отбросить, однако, сразу некоторые *плохие* идеи. Например, в случае транслятора недостаточно брать в качестве функции φ значение, зависящее только от первой буквы идентификатора; анализ существующих программ показывает, что 26 букв[1] представлены очень неравномерно (даже если не учитывать влияние ФОРТРАНа, который выделяет *I, J, K, L, M, N* среди

[1] Число 26 фигурирует здесь потому, что авторы рассматривают идентификаторы состоящими только из 26 букв латинского алфавита. – *Прим. перев.*

других букв). Точно так же не всегда удачным является выбор функции, равномерно распределенной на множестве всех теоретически возможных ключей, поскольку ключи, фактически используемые на практике, часто далеки от равномерного распределения на этом множестве; в случае транслятора, например, известно, что короткие идентификаторы встречаются чаще.

Не следует пренебрегать ресурсами, предлагаемыми моделированием для сравнения некоторого числа функций, являющихся кандидатами на роль функции φ. Если размер таблицы и частота ее использования это оправдывают, можно провести одно или несколько испытаний, которые могут рассматриваться «типичными» (этот термин нуждается в уточнении для каждого приложения), и изучить распределение ключей с помощью программы, печатающей результаты вида

ФУНКЦИЯ РАССТАНОВКИ НОМЕР 2: СУММА ЛИТЕР
ИСПЫТАНИЕ НОМЕР 3
3227 КЛЮЧЕЙ ВКЛЮЧЕНЫ ЗА 1 ОБРАЩЕНИЕ
1426 КЛЮЧЕЙ ВКЛЮЧЕНЫ ЗА 2 ОБРАЩЕНИЯ
352 КЛЮЧА ВКЛЮЧЕНЫ ЗА 3 ОБРАЩЕНИЯ
46 КЛЮЧЕЙ ВКЛЮЧЕНЫ ЗА 4 ОБРАЩЕНИЯ
3 КЛЮЧА ВКЛЮЧЕНЫ ЗА 5 ОБРАЩЕНИЙ
СРЕДНЕЕ ЧИСЛО ОБРАЩЕНИЙ = 1,451

что можно сравнивать с оптимальными результатами (см. ниже). Понятие «обращения» зависит от способа решения коллизий, рассмотренного ниже.

Часто рассматриваемый тип функции φ полагает ключ образованным из m смежных «полей» – например, для идентификатора, или, более широко, для объекта типа СТРОКА, из полей, составляющих этот объект, – и ставит в соответствие некоторый числовой код значениям этих полей, например от 1 до 26 для букв и от 27 до 36 для цифр. Значение ср получается применением к этим кодам таких операций, как сложение или «исключающее или». Применим, например, функцию φ = «сумма значений числовых кодов букв» к множеству ключей

БЕГЕМОТ ВАРАН ГАВИАЛ ЖИРАФ ЗЕБРА КЕНГУРУ ПИТОН

и выберем N = 20 (случай, по–видимому, не реальный); тогда таблица будет иметь 20 элементов, пронумерованных от 0 до 19. функция расстановки φ по модулю 20 дает[1]

f("БЕГЕМОТ") = (2 + 6 + 4 + 6 + 13 + 15 + 19) **mod** 20 = 5
f("ГАВИАЛ") = 10
f("ВАРАН") = 16
f("ЖИРАФ") = 15
f("ЗЕВРА") = 14
f("КЕНГУРУ") = 12
f("ПИТОН") = 15

Только последнее включение в этом примере приводит к коллизии.

VII.2.5.3. Внешнее разрешение коллизий

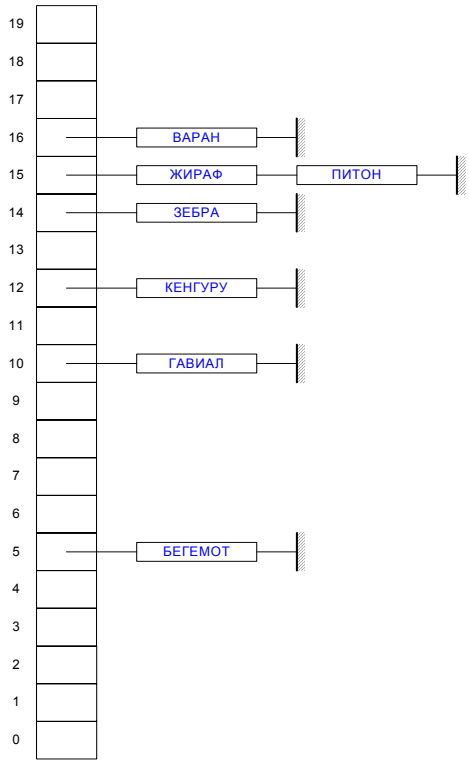
Простое решение для обработки коллизий состоит в буквальном понимании объявления

массив t[0 : N – 1]:ЛИНЕЙНЫЙ СПИСОК$_{\text{элемент}}$

и представлении t в виде таблицы указателей, инициализируемых при создании таблицы константой ПУСТО. Каждый из них указывает на упорядоченный линейный

[1] В этом примере буквы русского алфавита имеют, естественно, нумерацию от 1 до 32. – *Прим. перев.*

список вне массива t, либо цепной (например, в «куче»), либо состоящий из смежных блоков (например, во внешней памяти). В таком случае отделяют распределение места массиву t, осуществляемое статически и независимо от числа n элементов, которые могут быть включены, от управления самими элементами, выполняемого динамически в зависимости от потребностей. Ясно, что при таком методе n вполне может превосходить размер N массива t



Алгоритмы поиска и включения записываются в этом случае непосредственно, исходя из соответствующих им. алгоритмов, которые служили для управления таблицей как цепным линейным списком (VII.2.2 и V.2.3). Если для ключей установлено отношение порядка, то при таком представлении, несомненно, интересно использование упорядоченных линейных списков, по крайней мере если проверка этого отношения порядка не слишком дорога. Среднее число сравнений ключей в том же списке делится примерно на 2 для безрезультатного поиска (и для включения, если предусматривается однозначность ключей; напротив, если однозначность ключей не предполагается, то включение становится более долгим; ср. VII.2.3).

Этот метод по–своему прост, обладая свойством полного отделения содержащихся в массиве t «точек входа» таблицы от самих элементов, содержащихся в линейных списках. Однако он приводит к некоторой потере места в памяти, вызванной необходимостью представления указателей. Он может быть интересен для таблиц большого размера; в этом случае можно держать массив t в оперативной памяти так, что каждое t[i] обозначает линейный список, имеющий сплошное представление в некотором числе «блоков» или «страниц», смежно расположенных во внешней памяти. Тем самым удается избежать чрезмерно большого числа внешних обращений; каждый доступ к данным означает простое вычисление t[i] и последующий поиск одной или нескольких смежных страниц.

VII.2.5.4. Разрешение коллизий в самом массиве

Вместо того чтобы рассматривать t как массив указателей, можно с тем же успехом разместить в нем сами элементы:

массив t[0 :N – 1] : ЭЛЕМЕНТ

Массив t инициализируется специальным значением, также обозначенным

ПУСТО.

Ясно, что в этом случае число n включенных элементов никогда не должно превышать N (в действительности мы будем ниже ограничиваться $n < N - 1$).

Здесь задача состоит в разрешении коллизий в самом массиве: чтобы включить объект x с ключом c, выбирается позиция с индексом $i = f(c)$, если $t[i] =$ ПУСТО; если же это не так, нужно испытать последовательную *замену адресов,* задаваемых **функциями вторичной расстановки,** зависящими, вообще говоря, от ключа: $f_1(c)$, $f_2(c)$, ..., $f_n(c)$ до тех пор, пока не обнаружится свободная позиция или будет отмечено, что свободных позиций нет–таблица заполнена. Те же самые значения $f_1(c)$, $f_2(c)$ и т.д. будут последовательно проверяться и в ходе поиска.

Чтобы оценить эффективность методов этого типа, недостаточно искать среднее число обращений в зависимости от числа n имеющихся элементов, потому что важную роль играет N, фиксированных размер таблицы. Удобно брать в качестве параметра **коэффициент заполнения**

$$\alpha = \frac{n}{N}$$

который выражается, вообще говоря, в процентах.

Как выбрать функции вторичной расстановки? Метод, кажущийся наиболее простым, состоит при «взятой» позиции $i = f(c)$ в последовательном испытании позиций $i + 1$, $i + 2$, $i + 3$ и т.д.; эти значения вычисляются по модулю N. Массив, таким образом, рассматривается «циклическим» (ср. циклическое представление файлов в V.5.4.2). В таком случае говорят о **линейной вторичной расстановке.** Этот метод интересен только тогда, когда массив остается мало заполненным ($\alpha \leq 75\%$). Попробуем посмотреть, что происходит в обсуждавшемся уже примере. Включение ключей БЕГЕМОТ, ГА–ВИАЛ, ВАРАН, ЖИРАФ, ЗЕБРА, КЕНГУРУ происходит без коллизий. Включение ключа ПИТОН дает значение адресной функции, равное 15. Оно равно значению адресной функции для ЖИРАФ; тогда надо попробовать следующую позицию – 16, но она занята ключом ВАРАН; поэтому ПИТОН будет включен в позицию 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
					БЕГЕМОТ					ГАВИАЛ		КЕНГУРУ		ЗЕБРА	ЖИРАФ	ВАРАН	ПИТОН		

Для включения ПИТОН потребовались три сравнения ключей; столько же будет нужно для поиска этого элемента.

А элемент с адресной функцией, имеющей значение 8, как, например, БАБУИН, включается за одно сравнение в позицию 8:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
					БЕГЕМОТ			БАБУИН		ГАВИАЛ		КЕНГУРУ		ЗЕБРА	ЖИРАФ	ВАРАН	ПИТОН		

Заметьте, что на то же самое место были бы включены элементы со значением адресной функции 14, 15 или 16. Предположим теперь, что выполняется включение элемента с ключом, равным 14, но отличным от ЗЕБРА; в этом случае потребуется выполнить 5 обращений к таблице. Хорошо виден главный недостаток этого метода: он имеет тенденцию к созданию смежных «пакетов» элементов и слить соседние пакеты в ходе последовательных включений, образуя таким образом все более крупные пакеты. Этот метод, таким образом, допустим лишь при небольшом коэффициенте заполнения

массива.

Можно улучшить метод, выбирая фиксированное приращение, большее 1. Наилучшая идея состоит в том, чтобы избегать коллизии «по модулю», беря в качестве последовательно заменяемых адресов

f(c) + g(c) f(c) + 2g(c) f(c) + 3g(c),...

Все эти значения вычислены, разумеется, по модулю N, а функция g похожа на функцию расстановки f. В самом деле, если f имеет вид

f(c) = φ(c) **mod** N

то особенно интересно взять

q(c) = φ(c) **mod** M

где M – число взаимно простое с N; проще всего взять M = N – 2 (но не N – 1, которое было бы четным при нечетном N). В этом случае говорят о **методе двойного совпадения;** из него можно извлечь два преимущества:

- последовательность адресов f(c) + kg (c) (k = 0, 1, 2, ..., N – 1) включает все значения от 0 до N – 1;
- на уровне вторичных адресов практически исключаются ситуации, которые мы назвали «коллизиями по модулю», т.е. случаи, где

c ≠ c', φ(c) ≠ φ(c'), но φ(c) = φ(c') **mod** N

Действительно, в этом случае получают

$$\begin{cases} \varphi(c) = \varphi(c) \bmod N \\ \varphi(c) = \varphi(c) \bmod (N-2) \end{cases}$$

только если φ(c) = φ(c') **mod** (N – 2) что маловероятно.

Метод дает описываемые ниже алгоритмы. Для того чтобы уметь выполнить поиск, не подсчитывая число испытаний, удобно всегда оставлять свободное место **в** таблице и диагностировать ошибку N–й попытки включения (а не (N + 1)–й).

```
программа включение (аргумент x : ЭЛЕМЕНТ;
        модифицируемое данное массив t[0 : N – 1] :ЭЛЕМЕНТ)
        {включение при ассоциативной адерсации, коллизии в массиве}
        переменные a, индекс, приращение, число–испытаний: ЦЕЛЫЕ;
        a ← φ(ключ (x));
        индекс ← a mod N; приращение ← a mod (N – 2);
        число–испытаний ← 0;
        пока t[индекс] ≠ ПУСТО повторять
            | число–испытаний ← число–испытаний + 1;
            | индекс ← (индекс + приращение) mod N;
        если число–испытаний < N – 1 то
            | t[индекс] ← x
        {в противном случае ошибка: отказано в последней свободной
        позиции}
программа поиск : ЭЛЕМЕНТ
        (аргументы c: КЛЮЧ,
        массив t[0 : N – 1] : ЭЛЭМЕНТ)
        {поиск при ассоциативной адресации, коллизии в массиве}
        переменные a, индекс, приращение: ЦЕЛЫЕ;
        a ← φ(c);
        индекс ← a mod N; приращение ← a mod (N – 2);
```

```
        пока t[индекс] ≠ x и t[индекс] ≠ ПУСТО повторять
            | индекс ← (индекс + приращение) mod N;
        поиск ← t[индекс]
```

Чтобы оценить эффективность этих алгоритмов, нужно обладать некоторыми сведениями относительно функций расстановки f и g, т.е. относительно φ. Если предположить, что получаемые значения существенно случайны, т.е. что f и g обеспечивают хорошее рассеяние, то можно доказать, что среднее число сравнений для достаточно больших n примерно равняется

$\frac{1}{1-\alpha}$ для безрезультатного поиска или включения

и

$\frac{1}{\alpha}\log_e\left(\frac{1}{1-\alpha}\right)$ для результативного поиска (натуральный логарифм).

Напомним, что $\alpha = \frac{n}{N}$ – это коэффициент заполнения.

Конкретно, это дает численные результаты:

Коэффициент заполнения α	Среднее число обращений при включении или безрезультатном поиске	Среднее число орбращений при результативном поиске
25%	1,33	1,15
50%	2	1,39
75%	4	1,85
90%	10	2,56
95%	20	3,15

Изучая эти результаты, можно заметить, что они зависят только от степени заполнения таблицы, а не от ее размера: в таблице, содержащей тысячи элементов, можно разыскать любой из них в среднем за два с половиной обращения, если только таблица заполнена не больше чем на 90%.

Более пессимистическая модель метода рассматривает все возможные последовательности значений функции расстановки, включая в числе равновозможных и самые катастрофические. Интересно отметить, что даже при этой гипотезе метод ассоциативной адресации остается достаточно хорошим. В случае когда вторичная функция расстановки «линейна» (предыдущий метод, использующий в качестве величины приращения единицу), можно доказать, что значения равны приблизительно

$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$ для безрезультатного поиска или включения для результативного поиска

При коэффициенте заполнения 50% получают соответственно 2,5 и 1,5 обращения; для 75% – соответственно 8,5 и 2,5. Эти результаты подтверждают то, что нам подсказал пример: метод линейной вторичной адресации быстро теряет эффективность для коэффициентов заполнения, превышающих 75%.

VII.2.5.5. Использование упорядоченности ключей в предыдущем методе

Анализируя снижение эффективности предыдущего метода, связанное главным образом с безрезультатным поиском, естественно поставить вопрос, можно ли эту задачу решить, сохраняя упорядоченными цепочки элементов, являющихся результатами коллизий, если на ключах существует отношение порядка

(обозначаемое <). Модификация алгоритмов здесь немного более сложна, чем в случае внешнего разрешения коллизий.

Чтобы получить новую версию алгоритма поиска, достаточно заменить условие в цикле **пока** на

t[индекс] < x **и** t[индекс] ≠ ПУСТО

Зато алгоритм включения немного усложняется, так как он может быть сведен к перемещению элементов при просмотре цепочки, образованной из элементов, участвующих в коллизии (ср. включение в отсортированный линейный список в сплошном представлении, VII.2.3):

```
программа включение (аргумент x : ЭЛЕМЕНТ;
                      модифицируемое данное массив t[0:N – 1]: ЭЛЕМЕНТ)
    {включение при ассоциативной адресации, коллизии в массиве}
    {использование и сохранение порядка ключей в списках}
    переменные: a, индекс, приращение, число–испытаний:ЦЕЛЫЕ,
                y: ЭЛЕМЕНТ;
    a ← φ(ключ (x));
    индекс ← a mod N; приращение ← a mod (N – 2);
    число–испытаний ← 0;
    пока t[индекс] ≠ ПУСТО повторять
        число–испытаний ← число–испытаний + 1
        если ключ (x) > ключ (([индекс]) то
            y ← ([индекс];
            t[индекс] ← x;
            x ← y;
            приращение ← φ(ключ (x)) mod N – 2
        индекс ← (индекс + приращение) mod N
    если число–испытаний < N – 1 то
        индекс ← x
    {в противном случае ошибка: отказано в выделении последнего свободного
    места}
```

Правильность этого алгоритма совсем не очевидна: он оперирует со списками, не являющимися непересекающимися. Мы, однако, предоставим читателю убедиться, что включенные элементы будут правильно отысканы. Надо проверить, в частности, что множество заданных элементов может быть включено в таблицу *некоторым способом,* и при том *только единственным* (доказать, что порядок двух последовательных включений не влияет на конечный результат независимо от того, имеет ли место коллизия).

Существен ли выигрыш в эффективности? Что касается сравнений ключей, это бесспорно. Доказано (ср. [Амбль 73]), что этот алгоритм сводит среднее число сравнений в случае безрезультатного поиска к значениям, полученным ранее для *результативного* поиска; как следует из предыдущей таблицы, примерно за три обращения можно определить, присутствует ли ключ в таблице произвольного размера, заполненной на 95%, что представляет замечательный результат. Этот результат является очень серьезным побуждением к использованию упорядоченности ключей, когда она существует (ср. 20 сравнений, необходимых в предыдущем случае).

Следует, однако, заметить, что теперь алгоритмы требуют повторяющихся проверок для сравнения порядка ключей, и преимущество метода становится иллюзорным, если эти проверки трудоемки. Кроме того, включение не только не обходится меньшим числом сравнений, чем ранее (в некоторых «вырожденных» маловероятных случаях их требуется много больше), оно может требовать многократного и, возможно, «дорогого» вычисления ср. Поэтому следует внимательно

изучить относительную стоимость сравнений ключей, проверок отношения порядка и вычислений φ, перед тем как будет принято решение о реализации алгоритма.

Другое возможное улучшение, позволяющее уменьшить вдвое размер обрабатываемых цепочек, подсказано в упражнении VII.3.

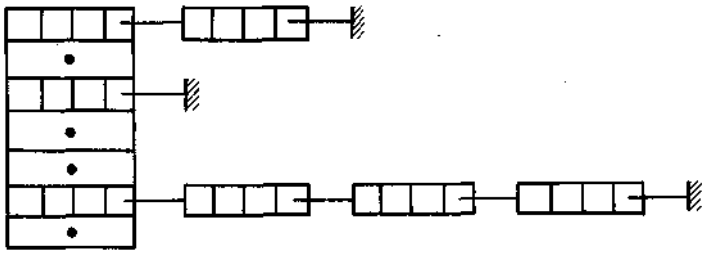
VII.2.5.6. Вариации и заключение

Важно отметить, что методы, практически используемые в больших программах, часто являются компромиссом между «внутренним» и «внешним» разрешением коллизий. Так, в случае переполнения таблицы при внутреннем разрешении ($n > N - 1$) не очень разумно диагностировать ошибку и прекращать обработку, как это предлагают данные алгоритмы. Предпочтительнее динамически назначать «вторичную таблицу», соединенную с предыдущей по методу, используемому обычно при управлении внешними файлами на дисках («прямой доступ с областью переполнения»).

Точная реализация алгоритма ассоциативной адресации зависит, в частности, от ограничения, налагаемого отношением между доступными объектами оперативной и внешней памяти. Часто применяемое усовершенствование состоит в группировке элементов в «блоки», содержащие b элементов и один указатель:

```
тип БЛОК = (прямо :массив [1 : b] :ЭЛЕМЕНТ;
            цепь:(БЛОК | ПУСТО))
```

В этом случае с тем же успехом, что и массив t, списки формируются из «блоков»:



Как это часто бывает при реализации алгоритма, представленные здесь базовые методы можно почти до бесконечности улучшать. Практическое решение есть результат, учитывающий физические ограничения (время, объем памяти), важность, с которой оцениваются гибкость и эффективность алгоритма, и время, которое можно (или хочется) посвятить программированию этого приложения.

В различных своих версиях метод ассоциативной адресации является, несомненно, самым основательным и самым «промышленным» среди методов управления таблицами, которые применяются в задачах большой разномерности. При хорошей функции расстановки и соответствующем методе разрешения коллизий ассоциативная адресация может дать наилучшее решение управления таблицей в ситуации, хорошо известной с самого начала.

VII.3. Сортировка

VII.3.1. Задача

Сортировкой называют операцию, упорядочивающую множество элементов по *ключам,* на которых определено отношение порядка[1].

[1] Здесь при переводе из оригинала исключена фраза, обосновывающая выбор французского термина для понятия сортировки. В связи с этим можно заметить, что, хотя «упорядочение» было бы, несомненно, более точным термином, мы всюду далее используем ставший уже привычным термин в информатике «сортировка».– *Прим. перев.*

Алгоритмы сортировки имеют большую практическую важность; они являются фундаментальными в некоторых областях, таких, как информатика АСУ, где данные перед обработкой практически регулярно сортируют по некоторым критериям, что не всегда, впрочем, оправдано. Изучение сортировок в равной степени интересно и само по себе, потому что речь идет об одной из наиболее глубоко исследованных областей информатики, где получены замечательные результаты по построению алгоритмов и изучению их сложности.

Интересным моментом в исследовании этой задачи является то, что переход от тривиальных алгоритмов (имеющих в наших примерах временную сложность $O(n^2)$) к основывающимся на тех же основных идеях алгоритмам повышенной эффективности (которые практически будут иметь сложность $O(n\log(n))$ требует значительного «концептуального прыжка». В таких случаях мало интересен поиск «микроскопических» улучшений; нужно опираться на основные задачи и использовать такие идеи, как «разделяй и властвуй», «уравновешивание» и т.д. Проводимые далее алгоритмы сортировки получили благодаря их достоинствам определенную известность. Поэтому не следует удивляться тому, что мы персонифицируем их, записывая с большой буквы и обращаясь к ним по их собственным именам: Пузырьковая сортировка, Древесная сортировка, Быстрая сортировка, Сортировка Слиянием...

VII.3.2. Определения

Традиционно различают *внутреннюю сортировку,* обрабатывающую хранимые в оперативной памяти данные, и *внешнюю сортировку,* оперирующую с данными, которые принадлежат к внешним файлам. Проблемы оптимизации различаются в обоих случаях: во внутренней сортировке стремятся сократить число сравнений и других внутренних операций, во внешней сортировке тоже интересуются, сравнениями, но решающим фактором эффективности алгоритма становится количество необходимых вводов и выводов.

Мы ограничиваемся здесь проблемами внутренней сортировки по двум соображениям: с одной стороны, они лучше формализованы, их легче исследовать, не связывая себя какими–либо гипотезами о конкретном окружении (вычислительная машина, внешние устройства, операционная система); с другой стороны, большинство разработчиков снабжают операционную систему общей программой внешней сортировки, учитывающей особенности машины; программисту легко ознакомиться с возможностями этой сортировки. Зато зачастую эффективный алгоритм внутренней сортировки бывает неизвестен; поэтому часто можно видеть программы, которые для сортировки 1000 данных используют алгоритм сложности $O(n^2)$ (миллион сравнений!) или копирование на внешний файл для использования полученной от разработчика машины внешней сортировки и последующее переписывание данных в оперативную память! Из этих соображений мы включаем в эту главу полные программы сортировки, написанные на ФОРТРАНе, АЛГОЛе W и ПЛ/1 или легко переводимые на эти языки.

Кроме того, мы ограничимся данными, представляемыми с помощью *массивов.* Это серьезное ограничение, поскольку многие интересные алгоритмы оперируют с другими структурами данных (списки, файлы, деревья и т. д); единственным оправданием может служить богатство сюжетов, которое обязывает нас ограничиться. Заметим, что в фундаментальном труде Кнута [Кнут 73] сортировке посвящено 338 плотно заполненных страниц, из которых 246 рассказывают о внутренней сортировке.

Таким образом, далее мы будем сортировать

массив a[1 : n]: ЭЛЕМЕНТ

где ЭЛЕМЕНТ – производительный тип, который остается неопределенным. С каждым элементом, имеющим индекс i, связан ключ, обозначаемый

ключ (i)

принадлежащий к типу КЛЮЧ, произвольному, но допускающему сравнения с помощью проверок вида

ключ (i) < ключ (j)

или ключ (i) < ключ (j) и т.д.

Фактически достаточно уметь сравнивать два ключа путем вызова подпрограммы

меньше (i, j)

выдающей значение ЛОГ, однако обозначение ключ (i) < ключ (j) более привычно.

Практически ключ(i) часто бывает компонентой a[i] массива или значением, выводимым из a[i] применением подпрограммы–«выражение».

В примерах подпрограмм, написанных на ФОРТРАНе, АЛГОЛе W и ПЛ/1, подпрограммам сортировки передаются в качестве аргументов имена двух подпрограмм *ОБМЕН* и *СРКЛЮЧ,* полагая, что вызов подпрограммы *ОБМЕН(I, J)* переставляет элементы *I* и *J*, а вызов *СРКЛЮЧ(I, J)* («сравнение ключей») выдает целое, которое равно –1, если ключ (*I*) ≤ ключ *(J);* нулю, если ключ (*I*) = ключ(*J*), и 1, если ключ(*I*) > ключ(*J*) .

Базовая операция перестановки для реорганизации массива обозначается

поменять элементы i и j

После каждого ее выполнения значения элементов a[i] и a[j], так же как и значение их ключей, меняются местами. В некоторых методах удобно в такой же степени использовать «полуобмены», обозначаемые

a[i] ← e

и e ← a[i]

где e имеет тип ЭЛЕМЕНТ. Здесь тоже одновременно выполняются присваивания модифицируемому элементу и соответствующему ключу.

Многократно использован пример массива, содержащий элементы БЕГЕМОТ, ГАВИАЛ, ВАРАН, ЖИРАФ, ЗЕБРА, КЕНГУРУ, ПИТОН в произвольном начальном порядке; ключами являются сами элементы; сравниваются они в алфавитном порядке.

Наконец, три определения:

- **инверсия** в массиве a – это пара индексов i и j, такая, что i < j, а ключ (i) > ключ (j);
- массив называется **отсортированным,** если он не содержит ни одной инверсии. Заметим, что так определяется сортировка в *возрастающем порядке.* Разумеется, алгоритмы сортировки в убывающем порядке выводятся непосредственно из соображений симметрии;
- алгоритм сортировки называется **устойчивым,** если он никогда не меняет относительный порядок в массиве двух элементов с равными ключами. Это может служить важным критерием, в частности, если происходит сортировка по некоторому ключу элементов, уже отсортированных по другому ключу. (*Пример:* исходя из телефонного справочника, отсортированного по фамилиям в алфавитном порядке, надо получить список абонентов по улицам и номерам домов; для каждого дома фамилии абонентов должны оставаться в алфавитном порядке.)

VII.3.3. Замечания о сложности алгоритмов сортировки

Алгоритм сортировки n данных, оперирующий сравнениями, имеет *минимальную сложность* O(n) или выше. Действительно, для того чтобы сортировка была правильной, надо, чтобы симметричный граф, определенный на множестве этих n данных с помощью отношения «элемент i связан с элементом j тогда и только тогда, когда они непосредственно сравниваются в ходе сортировки», был связным, откуда

следует, что он содержит по крайней мере n – 1 дуг (как это легко видеть из рекуррентности); следовательно, должны быть выполнены по крайней мере n – 1 сравнений.

Максимальная сложность всякого алгоритма сортировки, оперирующего сравнениями, не меньше O(nlogn). Если, действительно, рассмотреть двоичное дерево всех последовательных сравнений, которые могут быть выполнены алгоритмом, можно увидеть, что это дерево должно иметь по крайней мере n! листьев. С помощью рекуррентности легко доказывается, что двоичное дерево с m листьями имеет глубину, превосходящую $\log_2 m$. Глубина дерева возможных сравнений и, следовательно, число сравнений, выполняемых в наиболее неблагоприятном случае, имеют, таким образом, порядок

O(log(n!))

По формуле Стерлинга O(log(n!)) = O(nlogn).

Аналогичным образом доказывается (связывая вероятности с листьями дерева), что *средняя сложность* алгоритма сортировки сравнениями равна по крайней мере O(nlogn), если все перестановки данных равновероятны.

VII.3.4. Главные базовые идеи

Некоторые базовые методы основываются непосредственно на интуиции; каждый из них может дать более или менее эффективные алгоритмы. Мы их представим в общих чертах; легко заметить применение принципа «разделяй и властвуй». Читатель сможет обнаружить близость к ситуациям повседневной жизни: карточные игры, поддержание некоторого порядка в своем кабинете (задача, которая тоже является «управлением таблицей»), классификация картотек и т.д.

VII.3.4.1. Сортировка включением

Основная идея **сортировки включением** проста; выбирают некоторый элемент, сортируют другие элементы, «включают» выбранный элемент, т.е. устанавливают его на свое место среди других элементов (ср. включение в упорядоченную таблицу в VII.2.3)[1]

```
программа Сортировка Включением
        (модифицируемое данное массив a[1 : n] : ЭЛЕМЕНТ)
    если n > 1 то
        Сортировка Включением (a[1 : n – 1]);
        включение (a[n], a[1 : n – 1])
```

Усовершенствования приведут нас к алгоритму, известному под именем «Сортировка Шелла».

VII.3.4.2. Сортировка слиянием

Сортировка слиянием возвращается к идее сортировки включением, применяя к ней рассмотренный выше принцип *уравновешивания* ; вместо разделения массива на один элемент a[n] и подмассив a[1 : n – 1] его расчленяют на две примерно равные части

[1] Напомним, что запись a[i : j], где 1 ≤ i ≤ j ≤ n, означает подмассив, содержащий элементы a[i], a[i + 1], ..., a[j – 1], a[j].

```
программа Сортировка Слиянием
    (модифицируемое данное массив a [1 : n] : ЭЛЕМЕНТ)
    переменная середина : ЦЕЛ;
    если n > 1 то
        середина ← ⌊n/2⌋ {то или другое значение, заключенное между 1 и n};
        Сортировка Слиянием (a[1 : середина});
        Сортировка Слиянием (a[середина + 1 : n]);
        Слияние (a[1 : середина], a[середина + 1 : n])
```

Операция Слияние, которая соединяет в единый отсортированный массив два отсортированных массива p и q, может выполняться за время O(max(p, q)). Таким образом, Сортировка Слиянием приводит к алгоритму сложности O(nlogn). Однако пространственная сложность для операции Слияние может оцениваться в O(n). Этот вид сортировки особенно интересен для структур данных, отличных от массивов, и мы не будем развивать его подробнее.

VII.3.4.3. Сортировка обменами

Сортировка обменами выполняется за несколько «просмотров», каждый из которых уменьшает число инверсий:

```
программа Сортировка Обменами
    (модифицируемое данное массив a[1 : n] : ЭЛЕМЕНТ)
    пока существуют два индекса i и j
          таких, что i < j и ключ (i) > ключ (j)
    повторять
        поменять элементы i и j
```

Вся проблема кроется в выборе переставляемых элементов i и j. Один простой метод приводит нас к Пузырьковой Сортировке; другой, гораздо более развитый дает Быструю Сортировку, представляющую собой алгоритм, основанный на сегментации с применением принципа уравновешивания.

VII.3.4.4. Сортировка извлечением

В своей простейшей форме сортировка извлечением соответствует тривиальной идее:

```
программа Сортировка Извлечением
    (модифицируемое данное массив a[1 : n] : ЭЛЕМЕНТ)
    пусть i–индекс элемента с минимальным ключом в a;
    поменять элементы 1 и i;
    Сортировка Извлечением (a[2 : n])
```

Более эффективная версия, использующая понятие двоичного дерева, будет рассмотрена под именем Древесная Сортировка.

VII.3.4.5. Сортировка распределением

Метод, совершенно отличный от предыдущих, не использует сравнений. Он предполагает, что множество, к которому принадлежат ключи, конечно и имеет относительно небольшой размер. В частности, предполагается, что каждый ключ имеет вид

$a_1\ a_2\ \ldots\ a_k$

где каждая компонента a_j есть элемент конечного множества E. Например, ключи могли бы быть числами, выраженными последовательностями цифр, или строками, в которых роль a_j играют литеры. Тогда можно использовать следующий алгоритм:

```
программа Сортировка Распределением
    (модифицируемое данное массив a[1 : n]: ЭЛЕМЕНТ)
    для i от K до 1 шаг –1 повторять
        сортировать по i–й компоненте ключей с помощью алгоритма
        устойчивой сортировки
```

Заметим, что порядок, в котором рассматриваются компоненты–справа налево в предположении, что на ключах определен «лексикографический» порядок компонент

$a_1 a_2 \ldots a_{j-1} a_j a_{j+1} a_m < a'_1 a'_2 \ldots a'_{j-1} a'_j a'_{j+1} \ldots a'_m$

если $a_j < a'_j$

Далее будет показано, как некоторые из бегло очерченных здесь пяти базовых методов дают практически эффективные алгоритмы.

VII.3.5. Сортировка включением. Метод Шелла

VII.3.5.1. Сортировка простым включением

В нерекурсивной форме сортировка включением записывается в виде

```
программа Сортировки Включением
    (модифицируемое данное массив a[1 : n] : ЭЛЕМЕНТ)
    для i от 2 до n повторять
        {здесь отсортирован a[1 : i – 1] (инвариант цикла)}
        включение a[i] на свое место в a[1 : i – 1]
        {здесь отсортирован a[1 : i]}
```

Операция включения в упорядоченный список, имеющий сплошное представление с помощью массива, была рассмотрена в VII.2.3. Напомним, что ее максимальная и средняя сложности равны O(n), а время ее выполнения улучшается, если сравнения ключей комбинируются с перемещениями элементов, имеющих ключи, не меньшие, чем ключ a[i]:

```
{включение a[i] на свое место в a[1 :i – 1]}
переменная j: ЦЕЛ;
j ← i – 1
пока j > 0 и ключ (j) > ключ (i) повторять
    поменять элементы j и j + 1;
```

Легко убедиться, что этот метод сортировки *устойчив*.

Если перестановки более трудоемки, чем полуобмены, можно заметить, что в процессе включения включаемый элемент участвует во всех последовательных перестановках и что можно слегка улучшить эффективность этого процесса, записав:

```
{включение a[i] на свое место в a[1 :i – 1]}
переменные x: ЭЛЕМЕНТ,
           c: КЛЮЧ,
           j: ЦЕЛОЕ;
x ← a[i]; c ← ключ (i); j ← i – 1;
пока j > 0 и ключ (j) > c повторять
    a[j + l] ← a[j];
    j ← j – 1
{новым местом a[i] является j + 1}
a[j + 1] ← x
```

Средняя и максимальная сложности алгоритма сортировки включением равны

$O(n^2)$, если все перестановки предполагаются равновероятными. Точное число выполнении цикла равно $\frac{n(n-1)}{4}$ для среднего и $\frac{n(n-1)}{2}$ для максимального значения (полученного для массива, исходно отсортированного в обратном направлении); метод, состоящий в комбинации проверок и перемещений, позволяет просматривать в среднем только половину подмассива a[1 : i – 1]; именно это и дает делитель 4 в средней сложности.

Сортировка включением имеет, таким образом, алгоритм сложности $O(n^2)$, т.е. весьма плохой алгоритм, так как отмеченный нами оптимум равен $O(n\log n)$. Вопреки установившемуся мнению использование *дихотомического поиска* не улучшает его эффективности; дихотомический поиск, вообще говоря, даже ухудшает ее, поскольку для этого требуется примерно $\log_2 n$ проверок и, кроме того, в среднем $n/2$ перемещений на включение.

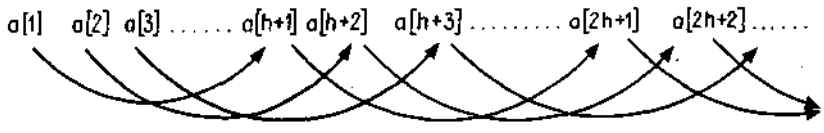
Отметим, однако, одно важное свойство сортировки включением: в противоположность другим методам она имеет наилучшую эффективность, если в начальном массиве установлен некоторый порядок. В случае «идеального», исходно отсортированного массива сложность равна $O(n)$; в общем случае алгоритм использует любой частичный порядок, имеющийся в массиве. Соединенное с простотой алгоритма, это его свойство естественно предопределяет его назначение для «завершения работы» более претенциозных методов, которые, подобно Быстрой Сортировке (VII.3.6), быстро разделяют массив на «почти» рассортированные части, но требуют достаточно тяжелой машинной обработки и задыхаются в окончательной сортировке малых подмассивов.

В VII.3.6 приводится фортрановская версия Сортировки Включением, которая использована для реализации Быстрой Сортировки.

VII.3.5.2. Метод Шелла

Основной недостаток простой сортировки включениями состоит в том, что каждый элемент перемещается за один раз только на одну позицию. Так, если m–й элемент таблицы должен быть перемещен в позицию 1, необходимо переместить на одну позицию вправо m – 1 предшествующих элементов. Каждое из таких перемещений уничтожает в точности *одну инверсию* в списке, т.е. оно уменьшает на единицу число пар [t[i], t[j]], таких, что i < j и ключ (i) > ключ (j). В результате общее число перемещений данных по крайней мере равно числу начальных инверсий, которое в среднем равняется $\frac{n(n-1)}{4}$.

Чтобы переступить этот нижний предел, не переделывая метод полностью (не совершая слишком большого «концептуального прыжка»), можно предусмотреть следующее решение, предложенное в 1959 г. Д. Л. Шеллом: вместо систематического включения элемента с индексом i в подмассив предшествующих ему элементов – способ, который, очевидно, слишком противоречит принципу «уравновешивания», чтобы вести к эффективному алгоритму, – включают этот элемент в подсписок, содержащий элементы с индексами i – h, i – 2h, i – 3h и т.д., где h – положительная константа. Таким образом формируется массив, в котором h–«серии» элементов, отстоящих друг от друга на расстояние h, сортируются отдельно:



Попутно заметим, что естественно ожидать эффекта от этого метода, который использует свойства «прямого доступа» к элементам массива, несмотря на то что простое включение как при сплошном, так и при цепном представлении является серией последовательных просмотров.

Разумеется, для получения отсортированного массива недостаточно отсортировать отдельно непересекающиеся h–серии: процесс возобновляется с новым значением $h' < h$. Но в силу характеристического свойства сортировки включением задача упрощается по сравнению с ситуацией, в которой исходный массив является произвольным: предварительная сортировка «серий с расстоянием h» ускоряет сортировку «серий с расстоянием h'». Алгоритм Шелла существенно использует это магическое свойство, сортируя сначала «серии с расстоянием h_i», затем h_{i-1}, ..., затем h_1, где h_i, h_{i-1},..., h_1 – это убывающая последовательность «приращений», в которой, разумеется, $h_1 = 1$ для того, чтобы последний переход обеспечивал окончательную сортировку массива.

Какой должна быть последовательность значений h_i, h_{i-1},..., h_1 выбираемых в качестве приращений? На этот вопрос нет абсолютного ответа, так как оптимальные значения зависят от размера сортируемого массива. Кроме нескольких частных случаев последовательностей приращений, математический анализ алгоритма исключительно сложен: сегодня не известны последовательности, оптимальность которых доказана (см. [Кнут 73]). Для достаточно больших массивов результаты тестов показывают, что рекомендуемой можно считать последовательность таких h_i, что $h_{i+1} = 3h_i + 1$; значит, последовательность включает по порядку приращения: ..., 364, 121, 40, 13, 4, 1. Начинается процесс с h_{m-2}, где m – наименьшее целое, такое, что $h_m > n$; другими словами, h_{m-2} есть первый превосходящий или равный $\left\lfloor \frac{n}{9} \right\rfloor$ элемент последовательности.

Тогда алгоритм записывается просто (сравните его внутренние циклы с циклами рассмотренной выше сортировки простым включением):

```
программа Сортировка Шелла
    (модифицируемое данное массив a[1 : n] : ЭЛЕМЕНТ)
    {Сортировка массива a методом Шелла, приращения h_{i+1} = 3h, + 1}
    переменные приращение, j: ЦЕЛЫЕ,
                              x: ЭЛЕМЕНТ,
                              c: КЛЮЧ;
    {определение исходного приращения}
    приращение ← 1;
    пока приращение < ⌊n/9⌋ повторять
        приращение ← 3 × приращение + 1;
    {здесь приращение равно max(l, h_{m–2})}
    {последовательные сортировки сериями}
    повторять
        {сортировки сериями с расстоянием «приращение»}
        для k от 1 до приращение повторять
            {сортировка k–й серии включением}
            для i от приращение +k до n шаг приращение повторять
                {включение a[i] на свое место среди предшествующих в
                серии}
                x ← a[i], c ← ключ(i); j ← i – приращение;
                пока j ≥ 1 и ключ (j) > c повторять
                    a[j + приращение] ←a[j];
                    j ← j – приращение;
                {новое место a[i]определяется суммой j + приращение}
                a[j + приращение] ← x;
        {здесь массив отсортирован сериями с расстоянием «приращение»}
        {переход от приращения = h_i к приращению = h_{i–1}}
        приращение приращение ← ⌊приращение/3⌋
    до приращение = 0
```

Вместо того чтобы перевычислять последовательность h_i, можно было бы получать ее из таблицы, предварительно вычисленной и размещенной в памяти. Можно также слить циклы по k и циклы по i при условии, что полуобмены заменяются на полные обмены.

Видно, что сортировка методом Шелла *неустойчива;* вообще говоря, это может оказаться неудобным. Простых средств для исправления этого недостатка нет.

ФОРТРАН

```
      SUBROUTINE SHELL(N, ECHANG, COMPCL)
      INTEGER N
      EXTERNAL ECHANG, COMPCL
      INTEGER COMPCL
C
C     ****************************************************************
C     * * *                                                      * * *
C     * * *     ВОСХОДЯЩАЯ СОРТИРОВКА МЕТОДОМ ШЕЛЛА             * * *
C     * * *     МАССИВА, К КОТОРОМУ ОБРАЩАЮТСЯ                   * * *
C     * * *     С ПОМОЩЬЮ ПОДПРОГРАММ                            * * *
C     * * *             ECHANG (I, J) (ПЕРЕСТАНОВКА ДВУХ         * * *
C     * * *             ЭЛЕМЕНТОВ)                               * * *
C     * * *         И   COMPCL (I, J) (СРАВНЕНИЕ ДВУХ            * * *
C     * * *             КЛЮЧЕЙ)                                  * * *
C     * * *     (РЕЗУЛЬТАТ –1, 0 ИЛИ 1 В ЗАВИСИМОСТИ ОТ          * * *
C     * * *      СООТНОШЕНИЯ КЛЮЧА(1) И КЛЮЧА(3) –               * * *
C     * * *     СООТВЕТСТВЕННО МЕНЬШЕ, РАВНО, БОЛЬШЕ)            * * *
C     * * *                                                      * * *
C     ****************************************************************
C
      INTEGER INCR, INCRP1, I, J
C     INCR – ПРИРАЩЕНИЕ, INCRP1 ПРИРАЩЕНИЕ P1
      IF(N. LE. 1) GOTO 1000
C     --- ИНИЦИАЛИЗАЦИЯ ПОСЛЕДОВАТЕЛЬНОСТИ ПРИРАЩЕНИИ   ---
      INCR = 1
C     /ПОКА INCR <N/9 ПОВТОРЯТЬ/
10    IF (INCR. GE. N/9) GOTO 20
          INCR = 3* INCR + 1
          GOTO 10
C
C     --- СОРТИРОВКА ---
C     /ПОВТОРЯТЬ ДО INCR=1/
20    CONTINUE
C         --- СОРТИРОВКА СЕРИЯМИ "РАССТОЯНИЕ INCR" ---
            INCRP1 = INCR + 1
            DO 50 J = INCRP1, N
C           --- ПОСТАВИТЬ ЭЛЕМЕНТ НА СВОЕ МЕСТОВ СЕРИИ ---
                L = J– INCR
                IF(L. LT. 1) GOTO 50
30              IF (COMPCL (L, L + INCR).LE. 0) GOTO 50
                    CALL ECHANG(L, L + INCR)
                    L=–L – INCR
                    GOTO 30
50              CONTINUE
C         --- ПЕРЕХОД К СЛЕДУЮЩЕМУ ПРИРАЩЕНИЮ ---
          INCR = INCR/3
          IF(INCR.GE.1) GOTO 20
C
1000  RETURN
      END
```

В такой реализации метод Шелла значительно лучше, чем простое включение, потому что число необходимых перемещений в среднем имеет порядок $1.66n^{1,25}$ при достаточно больших n. Значения (приближенные) из следующей ниже таблицы показывают, что метод выдерживает сравнение с методами O(nlogn) до n, равных нескольким тысячам. Свыше $n = 10^5$, однако, значения данных не имеют большого смысла при современном состоянии устройств оперативной памяти; можно заметить, что Сортировка Шелла, которая «перекрывает» массив широкими интервалами, особенно плохо приспособлена к системам с виртуальной памятью.

n	$1{,}66n^{1,25}$	$n\log_2 n$
100	525	664
1000	9 335	9966
10000	166000	132877
10^5	$2{,}95\cdot10^6$	$1{,}66\cdot10^6$
10^6	$5{,}25\cdot10^7$	$1{,}99\cdot10^7$

VII.3.6. Сортировка обменами; «Быстрая Сортировка»

VII.3.6.1. Пузырьковая Сортировка

Самый простой алгоритм сортировки обменами можно записать в виде

```
программа Пузырьковая Сортировка (аргумент массив a [1 : n]: ЭЛЕМЕНТ)
    переменная инверсия: ЛОГ;
    {инверсия имеет значение истина, если предшествующий просмотр
    убеждает, что в массиве осталась еще по крайней мере одна инверсия}
    повторять
        инверсия ← ложь;
        для i от 1 до n – 1 повторять
            если ключ (i) > ключ (i + 1) то
                {обнаружена инверсия: перестановка}
                инверсия ← истина;
                поменять элементы i и i + 1
    до ~ инверсия
```

Название «Пузырьковая Сортировка» происходит от образной интерпретации, по которой алгоритм заставляет «легкие» элементы мало–помалу всплывать на «поверхность».

Этот алгоритм имеет минимальную сложность O(n): если массив первоначально отсортирован, переменная инверсия никогда не примет значение истина. Напротив, максимальная сложность равна $O((n-1)^2) = O(n^2)$: в самом деле, если минимальный элемент имеет первоначально индекс n, потребуется n – 1 выполнений внешнего цикла, чтобы дать ему индекс 1. Итак, внутренний цикл всегда выполняется n – 1 раз при каждом выполнении внешнего цикла. Средняя сложность также равна $O(n^2)$, если минимальный элемент исходно распределяется случайно между индексами 1, 2,..., n.

Возможны различные усовершенствования; наиболее очевидное состоит в сохранении некоторой целой переменной, например последний, инициализируемой значением n – 1 и указывающей последний индекс, при котором была выполнена перестановка при последнем просмотре; тогда можно заменить n – 1 на последний в качестве верхней границы цикла для. Можно также просматривать массив во встречных направлениях, слева направо и справа налево.

Все это, однако, не меняет существенным образом сложность Пузырьковой Сортировки, которая в основе порочна своим «микроскопическим» подходом к массиву, когда каждый элемент всегда сравнивается только с непосредственно

соседними. Пузырьковая Сортировка – это **один из наихудших известных алгоритмов сортировки** (и в то же время один из наиболее употребительных среди программистов). Заметим, например, что, если a[n] имеет минимальный ключ, а оставшаяся часть массива a[1 : n – 1] исходно отсортирована, Пузырьковая Сортировка достигает своей максимальной сложности $(n - 1)^2$! Когда необходим алгоритм простой сортировки, легко программируемый за несколько минут, следует обращаться к *Сортировке Включением, которая всегда предпочтительнее Пузырьковой Сортировки.*

Если желательно получить эффективный метод, оперирующий обменами, нужно решительно отказаться от наивной идеи Пузырьковой Сортировки и обратиться к алгоритму Хоара [Хоар 62], [Хоар 71d], который дает один из лучших известных методов (если не наилучший). Его обычно называют Быстрой Сортировкой (Quicksort); этому алгоритму в той же мере хорошо подошло бы имя «Сортировки сегментацией».

VII.3.6.2. Принцип Быстрой Сортировки

Быстрая Сортировка является действительно систематическим применением принципов «разделяй и властвуй» и «уравновешивание» к сортировке обменами. Идея состоит в том, чтобы избежать рассеивания, порождаемого предыдущей программой, разделяя множество сортируемых элементов на две одинаковые части.

Для этого предположим, что подпрограмма, которую мы назовем Деление, может разыскать «главный» элемент с ключом c и реорганизовать массив таким образом, что главный элемент получает некоторый индекс s, все элементы с ключами, меньшими или равными c, размещаются слева (т.е. имеют индексы < s), а все элементы с ключами, большими или равными c, – справа:

ключ ≤ c	ключ c	ключ ≥ c
1 … s – 1	s	s + 1 … n

Тогда для полной сортировки массива достаточно:

а) больше не трогать главный элемент, который находится на своем месте;

б) *отсортировать* (рекурсивно) элементы с ключами, меньшими или равными c, т.е. подмассив a[1 : s – 1];

в) *отсортировать* (рекурсивно) элементы с ключами, большими или равными c, т.е. подмассив a[s + 1 : n].

Мы имеем здесь типичный рекурсивный подход:

- параметризация: рассматривают подмассив a[i : j]; для целого массива i = 1, а j = n;
- тривиальный случай: это, например, i = j – случай, в котором нечего делать;
- переход к общему случаю от более простого: он осуществляется благодаря подпрограмме Деление:

```
программа Деление: ЦЕЛ
    (модифицируемое данное a[i : j] ЭЛЕМЕНТ)
    {выбор главного элемента в a[i:j] и распределение a[i : j] вокруг этого
    элемента. Результат, выдаваемый подпрограммой, – окончательный индекс,
    назначенный главному элементу}.
```

Если такая подпрограмма существует, Быстрая Сортировка сразу же получается в рекурсивной форме:

```
программа Быстрая Сортировка
  | (модифицируемое данное массив a [i : j] : ЭЛЕМЕНТ)
  | переменная s : ЦЕЛ;
  | если j > i то
  |       | s ← Деление (a[i :j]);
  |       | Быстрая Сортировка (a[i : s – 1]);
  |       | Быстрая Сортировка (a[s + 1 : j])
```

Заметьте, что алгоритм правомерен при любом порядке двух рекурсивных вызовов; это свойство будет использовано для усовершенствования Быстрой Сортировки. А пока рассмотрим программу Деление. Можно «делить» массив с помощью «главного» элемента за время O(n), рассматривая каждый элемент только один или два раза, если «зажигать свечу с двух концов». Схематически метод состоит в обработке массива и слева, и справа до тех пор, пока слева не будет обнаружен элемент с ключом, превосходящим ключ главного элемента, а справа–элемент с ключом, меньшим, чем ключ главного:

	u			v		
→	↓			↓		←
ГАВИАЛ	ПИТОН	БЕГЕМОТ	ЖИРАФ	ВАРАН	КЕНГУРУ	ЗЕБРА

После этого можно поменять местами эти элементы (и тем самым уничтожить инверсию). Затем такая двойная обработка слева и справа продолжается с уже достигнутых позиций. Массив считается разделенным, когда левая и правая позиции соединяются; их общее значение и есть s.

Мы вернемся еще к подробностям этого алгоритма. Пока достаточно, чтобы читатель убедился, что алгоритм можно реализовать со сложностью O(n), или, вообще говоря, O(j – i), когда он применяется к подмассиву a[i : j]. Чтобы написать программу Деление оптимальным образом, нужно прежде всего вернуться к общему обзору алгоритма Быстрой Сортировки.

VII.3.6.3. Построение эффективной Быстрой Сортировки

В Быстрой Сортировке написано

```
| если j > i то
|       | s ← Деление (a[i : j]);
|       | Быстрая Сортировка (a[i : s – 1]);
|       | Быстрая Сортировка (a[s + 1 : j])
```

и мы отмечали, что относительный порядок двух рекурсивных вызовов может быть в принципе произвольным. Рассмотрим, однако, практическую реализацию рекурсии: незавершившиеся вызовы будут занесены в стек. В самом неблагоприятном случае последовательные «деления» могут давать систематически s = i или s = j, т.е. делить массив на два существенно не одинаковых подмассива, поскольку один будет пустой, а другой–иметь размер j – i. Глубина вложенности рекурсивных вызовов может достичь величины n – размера исходного массива. Другими словами, надо предусматривать стек длиной n (пространственная сложность O(n)), что неприемлемо.

Существует простое средство в этом случае–начинать всегда с подмассива меньшего размера. Тогда этот размер будет меньше половины размера предыдущего подмассива; иначе говоря, максимальное число P(n) одновременно записываемых в стек элементов, которое является также максимальной глубиной рекурсивных вызовов, удовлетворяет отношению

$$P(n) < 1 + P\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

Учитывая, что P(0) = 0, получаем $P(n) < \log_2 n + 1$. Этот метод позволяет свести пространственную сложность Быстрой Сортировки к O(logn). Учитывая, что $\log_2 10^6 = 20$ (примерно), можно считать, что для всех практических приложений на нынешних машинах (напомним, что речь идет о внутренней сортировке) для представления стека достаточно фиксированной области памяти из 20 элементов. Непосредственно получается модификация алгоритма, обеспечивающая это свойство:

```
программа Быстрая Сортировка
    (модифицируемое данное массив a [i : j] : ЭЛЕМЕНТ)
    переменная s : ЦЕЛ;
    если i < j то
        s ← Деление (a[i : j]);
        если s – i ≤ j – s то {начинать слева}
                Быстрая Сортировка (a[i : s – 1]);
                Быстрая Сортировка (a[s + 1 : j])
        иначе {j – s < s – i: начинать справа}
                Быстрая Сортировка (a[s + 1 : j]);
                Быстрая Сортировка (a[i : s – 1])
```

Пространственная сложность сводится таким образом к величине, не превосходящей O(logn). Какова же временная сложность T(n)? Так как Деление имеет сложность O(j – i), получаем

$$T(n) = O(n) + T(s - 1) + T(n - s)$$

Все зависит, таким образом, от s, т.е. от сегментации, выполняемой Делением. Идеальная ситуация, которой можно было достичь применением принципа уравновешивания, состоит в сегментировании массива на две примерно равные части, т.е. $s = \frac{n}{2}$ (приблизительно). Тогда имеем

$$T(n) \le O(n) + 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

Следовательно, T(0) = 0, $T(n) = O(n\log_2 n)$, при этом коэффициент такой же, как и коэффициент при n в сложности Деления. Таким образом, метод достигает O(nlogn), что, как мы видели, является нижней границей для сложности алгоритмов сортировки. Отлично! Но принятая гипотеза была благоприятной. Предположим, напротив, что деление всегда выполняется около первого или последнего элемента рассматриваемых подмассивов (s = i или s = j). Тогда остается сортировать подмассив, в котором число элементов на единицу меньше, и имеем:

$$\begin{aligned} T(n) &= O(n) + O(1) + T(n - 1) \\ &= O(n) + T(n - 1) \\ &= O(n) + O(n - 1) + \ldots + O(1) \\ &= O(n^2) \end{aligned}$$

В этом случае Быстрая Сортировка имеет теоретическую сложность, сходную со сложностями наихудших из рассмотренных ранее алгоритмов, а практическую сложность, вероятно, даже большую из–за времени, требуемого для реализации

рекурсий (управление стеком).

Таким образом, Быстрая Сортировка имеет максимальную сложность $O(n^2)$ и минимальную сложность $O(n\log n)$. Показано (см. [Кнут 73] или [Седжуик 75]), что средняя сложность, оцениваемая для равных вероятностей всех перестановок, также равна $O(n\log n)$ с примерно $2n\log_2 n$ сравнениями ключей; показано также, что вероятность самого неблагоприятного случая со сложностью $O(n^2)$ достаточно мала.

Этот результат, однако, не очень утешителен, потому что, как легко утверждать, наиболее благоприятный случай достигается, в частности, когда данные *уже исходно отсортированы* (возможно, в обратном порядке). В этом **парадокс** Быстрой Сортировки: в противоположность Сортировке Включением или даже Пузырьковой Сортировке она теряет свои качества на частично упорядоченных массивах! Это серьезное неудобство: как уже отмечалось, сортировки данных, предварительно «почти» упорядоченных, нередки в различных прикладных задачах.

Чтобы использовать Быструю Сортировку практически, нужно обязательно требовать от Деления ограничений на выбор главного элемента, уменьшающих вероятность того, что два «сегмента» смогли бы оказаться слишком неравными. Возможны две стратегии:

а) «Случайная выборка» значения индекса главного элемента среди $\{i, i + 1, ..., j\}$ при каждом выполнении деления. Для этого используется подпрограмма датчика псевдослучайных чисел, существующая в большинстве систем (для написания такой подпрограммы можно обратиться к книге [Кнут 69]). Основной недостаток – время, необходимое для этой операции.

б) Уход от наиболее катастрофических ситуаций путем назначения главным элементом в Делении среднего элемента небольшой выборки элементов из массива, т.е. такого элемента, что рассматриваемая выборка содержит примерно одинаковое число ключей, меньших и больших, чем ключ этого элемента. Тогда, если выборка из массива представительна с точки зрения значений ключей, можно рассчитывать на получение главного элемента, который находится приблизительно в середине массива. Наиболее простой и, несомненно, лучшей в общем случае является выборка, содержащая три элемента с индексами i, j и $\left\lfloor \frac{i+j}{2} \right\rfloor$. Мы будем применять в Делении именно этот метод (заметим, что обобщение задачи поиска среднего элемента на определение k–го (по порядку ключей) элемента массива, содержащего n элементов, может рассматриваться как вариант Быстрой Сортировки; задача VII.4).

Использование того или другого из этих методов существенно уменьшает вероятность «катастрофического» случая $O(n^2)$. Важно, однако, отметить, что полностью такая возможность не исключается: Быстрая Сортировка *всегда может вырождаться.* Парадоксально, что от Быстрой Сортировки, являющейся лучшим известным алгоритмом внутренней сортировки, приходится отказываться в задачах, где установление верхней границы (вида $kn\log_2 n$) времени, необходимого для сортировки, является критическим. Такими являются, например, задачи, решаемые в «реальном времени» (представьте себе программу управления воздушным движением, которая должна выполнить сортировку на последних секундах посадки самолета...). Этот недостаток Быстрой Сортировки объясняет, почему далее включено обсуждение другого важного алгоритма – Древесной Сортировки – вообще говоря, более «тяжелого», чем Быстрая Сортировка (его константа k примерно удваивается), но имеющего среднюю и *максимальную* сложность $O(n\log n)$.

Чтобы стать действительно эффективным алгоритмом, Быстрая Сортировка требует еще одного улучшения. В рассмотренной выше версии очевидно, что машинная реализация алгоритма (управление рекурсией и выбор главного элемента)

становится слишком тяжелой для небольших подмассивов. Быстрая Сортировка неприменима к малым массивам; в большом массиве следует «останавливать» рекурсию, когда размеры подмассивов становятся меньше некоторой константы, называемой порогом: после этого используется метод, эффективность которого может улучшаться на частично отсортированных данных, например сортировка простым включением. Теоретическое исследование Кнута приводит к оптимальному значению порог = 9, вычисляемому для машины, достаточно типичной среди обычных ЭВМ. В .практических тестах, с которыми мы работали на ИБМ 370/168 (VII.3.9), значения порога от 8 до 20 давали хорошие результаты, а оптимум располагался между 14 и 16. Интересно в связи с этим отметить, что при пороге, равном 60, когда «почти каждая работа» должна поручаться Сортировке Включением, время вычислений только на 25% превосходит оптимум (т.е. для 1000 элементов–она более чем в 15 раз меньше времени сортировки простым включением!)

В этой новой модификации алгоритм принимает вид

```
программа Быстрая Сортировка
    (модифицируемое данное массив a [i : j] : ЭЛЕМЕНТ)
    {полностью рекурсивная версия}
    константа порог = ...{значение, несомненно, близкое к 15}
    переменная s .ЦЕЛ;
    если j – i > порог то
        s ← Деление (a[i : j]);
        если s – i < j – s то
            Быстрая Сортировка (a[i : s – 1]);
            Быстрая Сортировка (a[s + 1 : j])
        иначе
            Быстрая Сортировка (a[s + 1 : j]);
            Быстрая Сортировка (a[i : s – 1])
    иначе
        Сортировка Включением (a[i : j])
```

В практической реализации алгоритма при каждом рекурсивном вызове передаются в качестве аргументов индексы i и j, а не сам подмассив: массив a обобществлен, таким образом, между всеми рекурсивными поколениями. Кроме того, чтобы облегчить управление стеком (оставаясь по–прежнему в рамках рекурсивной версии), можно, применяя рассмотренные в гл. VI упрощения, заменить второй рекурсивный вызов присваиванием и возвратом в начало программы:

```
пока j – i < порог повторять
        s ←Деление (a[i : j]);
        если s – i ≤ j – s то
            Быстрая Сортировка (a[i : s – 1]);
            i ← s + 1
        иначе
            Быстрая Сортировка (a[s + 1 : j]);
            j ← s – 1;
Сортировка Включением (a[i : j])
```

Эти соображения использованы в показанной ниже рекурсивной версии программы на АЛГОЛе W. Программа включает процедуру *PARTITION* (*ДЕЛЕНИЕ*), которая обсуждается в следующем разделе. В соответствии с рекомендациями VI.3 выделяется собственно рекурсивная часть подпрограммы путем объявления рекурсивной процедуры *TRIREC* (*РЕКУРСИВНАЯ СОРТИРОВКА*) с аргументами *I* и *J*. Эта процедура – внутренняя по отношению к *TRI_RAPIDE* – *БЫСТРОЙ СОРТИРОВКЕ*, которая сама не является рекурсивной; аргумент *TRI_RAPIDE* – массив *A*. Для примера

в качестве *A* рассматривается массив вещественных; ключом элемента *A(I)* служит его абсолютное значение. Более общая подпрограмма могла бы в качестве аргумента иметь не массив, а две подпрограммы, одна из которых сравнивает ключи, а другая меняет местами элементы:

```
PROCEDURE TRI_RAPIDE (LOGICAL PROCEDURE
                COMPARAISON_DE_CLES: PROCEDURE ECHANGER);
COMMENT  COMPARAISON_DE_CLES – СРАВНЕННЕ КЛЮЧЕЙ,
         ECHANGER – ОБМЕН
COMMENT  СРАВНЕНИЕ КЛЮЧЕЙ (I, J) РАВНО ИСТИНЕ ТОГДА И ТОЛЬКО
         ТОГДА, КОГДА КЛЮЧ ЭЛЕМЕНТА I БОЛЬШЕ КЛЮЧА
         ЭЛЕМЕНТА J, ОБМЕН (I, J) ПЕРЕСТАВЛЯЕТ ЭЛЕМЕНТЫ 1 И J;
...
```

Такое решение принимается в VII.3.6.5, где рассмотрена нерекурсивная версия этой программы на ФОРТРАНе.

Рекурсивная версия ПЛ/1 сходна с версией АЛГОЛа W в синтаксических подробностях с той разницей, что в ПЛ/1 необходимо явно указывать, что *TRIREC* может вызываться рекурсивно:

```
TR1REC: PROCEDURE(I, J )
          RECURSIVE RETURNS (BINARY FIXED);
```

Прежде чем говорить о нерекурсивной версии, надо уточнить, как пишется программа Деление.

VII.3.6.4. Хорошая программа Деление

Существует много вариантов программы Деление, однако они далеко не одинаковы с точки зрения простоты и эффективности. Для удовлетворения этому критерию ставятся две цели: попытаться ускорить внутренние циклы и предусмотреть «случайный» характер массива, т.е. не ввести по оплошности некоторый порядок, который мог бы оказаться «убыточным» с позиций общей производительности Быстрой Сортировки: мы должны воздержаться от попыток сортировать при Делении!

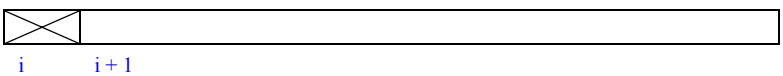
АЛГОЛ W

```
PROCEDURE TRI_RAPIDE (REAL ARRAY A(*); INTEGER VALUE N);
    COMMENT:  СОРТИРОВКА МАССИВА A(1 :: N) МЕТОДОМ ХОАРА:
              КЛЮЧАМИ ЭЛЕМЕНТОВ ЯВЛЯЮТСЯ ИХ АБСОЛЮТНЫЕ
              ЗНАЧЕНИЯ;
    BEGIN
    INTEGER PROCEDURE PARTITION (INTEGER VALUE I, J);
    COMMENT:  ДЕЛИТ ПОДМАССИВ A(I :: J) ЭЛЕМЕНТОМ, ИНДЕКС
              КОТОРОГО ВЫДАЕТСЯ В КАЧЕСТВЕ РЕЗУЛЬТАТА
              ПРОЦЕДУРЫ;
    IF J <=I THEN I
    ELSE
        BEGIN
        INTEGER MILIEU, INDICE_PIVOT, U, V;
        COMMENT: MILIEU–СЕРЕДИНА, 1ND1CE_PIV0T – ИНДЕКС
                 ГЛАВНОГО ЭЛЕМЕНТА;
        REAL IABS, JABS, MILABS;
        REAL T;
        COMMENT: T – ПРОМЕЖУТОЧНАЯ ПЕРЕМЕННАЯ ДЛЯ ОБМЕНОВ;
        MILIEU := (I + J) DIV 2;
        IABS = ABS A (I); JABS = ABS A (J);
        MILABS = ABS A (MILIEU);
```

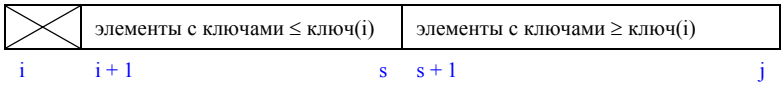
```
        COMMENT: ПОМЕСТИТЬ В ПОЗИЦИЮ I СРЕДНИЙ ИЗ A(I), A(J), A
                 (MILIEU);
        INDICE_PIVOT: =
            IF IABS <= MILABS THEN
                IF MILABS <=JABS THEN MILIEU
                ELSE IF IABS <= JABS THEN J ELSE I
            ELSE IF MILABS >= JABS THEN MILIEU
            ELSE IF IABS >= JABS THEN J ELSE I;
        T: = A(1); A(I) := A(INDICE_PIVOT);
        A(INDICE_PIVOT) := T;
        COMMENT: ДЕЛЕНИЕ;
        U : = I; V: = J; IABS : = ABS A (I);
        WHILE U < V DO
            BEGIN
            WHILE (U < V) AND (ABS A(U) <= IABS) DO
                U:=U + 1;
            WHILE (V > U) AND (ABS A (V) >= IABS) DO
                V: = V – 1;
            T := A(U); A(U) := A(V); A(V) := T
            END;
        COMMENT:  НИЖЕ УКАЗАН РЕЗУЛЬТАТ ПОДПРОГРАММЫ ПРИ
                  J > I;
        U
        END PAPTITION;
    PROCEDURE TRIREC (INTEGER VALUE I, J);
        COMMENT: СОБСТВЕННО РЕКУРСИВНАЯ ЧАСТЬ;
        BEGIN
        INTEGER INDICE_PIVOT;
        WHILE J > I DO
            BEGIN
            INDICE_PIVOT : = PARTITION (I, J);
            IF INDICE_PIVOT – I <= J – INDICE_PIVOT THEN
                BEGIN
                TRIREC (I, INDICE – PIVOT–1)
                I := INDICE_PIVOT + 1
                END
            ELSE
                BEGIN
                TRIREC (INDICE – PIVOT + 1, J);
                J := INDICE_PIVOT – 1
                END
            END
        END TRIREC;
    COMMENT: ТЕЛО ПРОЦЕДУРЫ БЫСТРАЯ СОРТИРОВКА;
    TRIREC (1, N)
    END TRI_RAPIDE;
```

Именно этот последний критерий труднее всего обеспечить. Следующий метод, предложенный Седжуиком, кажется лучше обычно публикуемых методов:

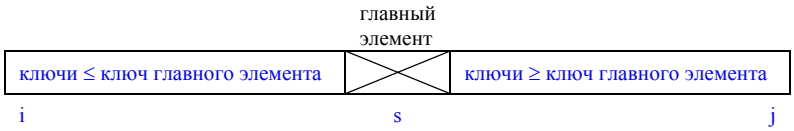
а) Поместить главный элемент в позицию i, поменяв его, если это необходимо, с элементом i;



б) Разделить подмассив a[i + 1 : j] с помощью значения главного элемента ключ(i) оставляя a[i] на своем месте; получается разделение промежуточной позицией, скажем, s:



в) Переставить элементы i и s. Подпрограмма выдает значение s:



Так достигается хорошее «разделение» a[i : j]. Трудным этапом является, очевидно, этап б); действительно, он достаточно тонок для оправдания того, что мы реализуем его, применяя методы Хоара (III.4)[1]

В начальной ситуации ничего не известно об элементах a[i + 1],... ..., a[j] и ключах. Требуемая конечная ситуация, символизируемая показанной выше схемой б), может быть представлена условием

$$(H)\quad \begin{cases} \text{ключ (k)} \le \text{ключ (i) для всех k, таких, что } i+1 \le k \le s \\ \text{ключ (k)} \ge \text{ключ (i) для всех k, таких, что } s+1 \le k \le j \end{cases}$$

Задача будет решена, когда обнаружено значение s, обеспечивающая истинность (H).

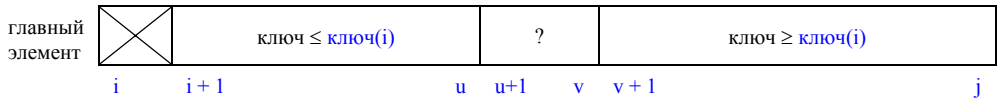
Часто применяемый и дающий хорошие результаты метод, обеспечивающий истинность такого условия, как (H), которое формируется из двух условий, «спаренных» с помощью одного общего элемента (здесь s), состоит в отыскании цикла, *инвариантом* которого является «расчлененная» версия (H), обозначаемая (H') и получаемая «раздваиванием» s на две переменных u и v:

$$(H')\quad \begin{cases} \text{ключ (k)} \le \text{ключ (i) для всех k, таких, что } i+1 \le k \le u \\ \text{ключ (k)} \ge \text{ключ (i) для всех k, таких, что } v+1 \le k \le j \\ u \le v \end{cases}$$

Действительно, если невозможно обеспечить (H) исходно (в противном случае задача была бы решена), то существенно проще обеспечить менее сильное условие (H'). Далее, поскольку (H') становится (H) при u = v, программа принимает вид

```
обеспечение начальной истинности (H');
пока u ≠ v повторять
    | "уменьшение v – u с сохранением (H') в качестве инварианта"
```

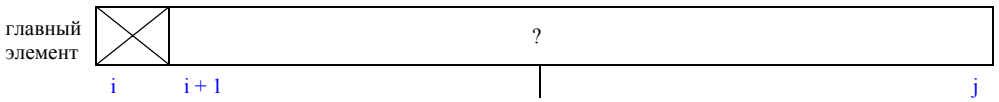
Можно заметить, что инвариант (H') является просто ослабленным вариантом схемы **б)** (см. выше):



где, кроме классов элементов, о которых известно, что их ключи либо не больше, либо не меньше ключа(i), рассматривается категория элементов, которые пока нельзя отнести к одному из двух предыдущих классов.

[1] Читатель, не интересующийся строжим доказательством, может сразу перейти к программе Деление на стр. 128.

Исходная истинность (H') обеспечивается сразу–достаточно взять в качестве области «?» массива a[i + 1 : j] произвольное целое, т.е. выполнить инициализацию:



Что касается цикла, имеющего (H') своим инвариантом, то его можно записать в виде

пока u ≠ v **повторять**
  {инвариант: (A) ключ (k) ≤ ключ (i) для i + 1 ≤ k ≤ u
  (B) ключ (k) ≥ ключ (i) для v + 1 ≤ k ≤ j
  (C) u ≤ v}
  пока u < v **и** ключ (u) ≤ ключ (i) **повторять**
    u ← u + 1;
  пока v > u **и** ключ (v) ≥ ключ (i) **повторять**
    v ← v – 1
  поменять элементы u и v
{здесь u = v,
(A) ключ (k) ≤ ключ (i) для i + 1 ≤ k ≤ u,
(B) ключ (k) ≥ ключ (i) для v + 1 ≤ k ≤ j}

Здесь (H') выражается как соединение трех свойств (A), (B) и (C). Доказательство их инвариантности просто: два внутренних цикла, которые можно называть «цикл по u » и «цикл по v», допускают u ≤ v, т.е. (C) в качестве инварианта. В точке (1) после выполнения цикла по u (A) становится:

$$(A_1)\quad \begin{cases} \text{ключ (k)} < \text{ключ (i)} \text{ для } i + 1 < k < u \\ \text{и (u = v } \textbf{или} \text{ ключ (u)} > \text{ключ (i))} \end{cases}$$

В точке (2) (B) остается истинными и, кроме того, имеют

(B_2)v = u **или** ключ (v) < ключ (i)

После выполнения операции поменять элементы u и v условия (A_1), (B) и (B_2) вновь дают(A_1) и (B). Кроме того, оба цикла завершаются: v – u есть управляющая величина циклов по u и по v ; это же имеет место и для внешнего цикла, так как цикл по u выполняется всегда по крайней мере один раз; действительно, в силу C (u ≤ v) и условия внешнего цикла (u ≠ v) в точке (0) имеют u < v; (A) порождает ключ (u) ≤ ключ (i).В (0), тогда цикл по u всегда выполняется по крайней мере один раз.

Условия (A) и (B), объединенные с u = v, дают таким образом, (H) на выходе из цикла и получается корректный вариант программы Деление:

программа Деление: ЦЕЛ (**модифицируемое данное массив** a[i : j]: ЭЛЕМЕНТ
  {неокончательная версия}
  переменные индекс–главного–элемента, u, v: ЦЕЛ,
    ключ–главного–элемента: КЛЮЧ;
  индекс–главного–элемента ← «индекс среднего из трех элементов с индексами i, j и $\left[\frac{i+j}{2}\right]$»
  поменять элементы и индекс–главного–элемента;
  {разделить}
  u ← i; v ← j; ключ–главного–элемента ← ключ (i);
  пока u ≠ v **повторять**
    {инвариант: (A) ключ (k) ≤ ключ–главного–элемента
    для i + 1 ≤ k ≤ u
    (B) ключ (к) ≥ ключ–главного–элемен–та
    для v + 1 ≤ k ≤ j

```
                    (C) u ≤ v}
    пока u < v и ключ (u) ≤ ключ–главного–элемента повторять
        u ← u + 1;
    пока v > u и ключ ≥ ключ–главного–элемента повторять
        v ← v – 1;
    поменять элементы и и v;
  {Здесь ключ (k) ≤ ключ–главного–элемента для i + 1 ≤ k ≤ u и
  ключ(k) ≥ ключ–главного–элемента для u + 1 ≤ k ≤ j}
  поменять элементы i и u;
  {массив разделен индексом u = v}
  Деление ← u
```

Видно, что эта программа имеет сложность O(j – i): действительно, цикл пока u ≠ v просматривает каждый элемент массива a[i + 1 : j] самое большее дважды, а остальное требует фиксированного времени. Однако ее практическую сложность, кажется, можно уменьшить: действительно, два самых внутренних цикла содержат двойную проверку – индексов и ключей; проверка индексов полезна, только когда соединяются u и v, т.е. в момент последнего выполнения внешнего цикла.

Существует, в самом деле, средство исключения ненужных проверок; оно далеко не тривиально, но заслуживает внимания, так как дает, несомненно, оптимальную версию Быстрой Сортировки

Новая версия Деления «тоньше» предыдущей:

- она уже не совсем симметрична относительно u и v;
- она позволяет переменным u и v пересекаться, но «останавливает» их сразу же после этого, потому что u < v + 1 является инвариантным свойством. В случае «пересечения» метод требует заключительного «восстановительного» обмена;
- она останавливает цикл по u и цикл по v на ключах, равных ключу главного элемента (а не только при больших или меньших ключах соответственно); априори это может показаться слишком жестким условием, но это фактически способствует обеспечению случайного характера массива;
- она требует в качестве дополнительной гипотезы наличия в массиве по крайней мере двух элементов. Поэтому надо вызывать эту версию Деления только в вариантах Быстрой Сортировки, имеющих порог ≥ 1.

Эта версия записывается в виде

```
программа Деление: ЦЕЛ(модифицируемое данное
                       массив a[i : j] .ЭЛЕМЕНТ)
  {окончательная версия}
  {гипотеза: 1 ≤ i < j ≤ n – подмассив имеет по меньшей мере два элемента}
  переменные:  индекс–главного–элемента, u, v: ЦЕЛ,
               ключ–главного–элемента: КЛЮЧ;
  индекс–главного–элемента ← «индекс среднего из трех
                              элементов i, j и ⌈(i + j)/2⌉»
  поменять элементы i и индекс–главного–элемента;
  {в силу этого обмена и принятой гипотезы можно утверждать, что
  существует k > i, такое, что k ≤ j и ключ (k) ≥ ключ (i)}
  u ← i; v ← j + 1; ключ–главного–элемента ← ключ (i);
```

пока u < v **повторять**
  {инвариант цикла:
    (A') ключ (k) ≤ ключ–главного–элемента для i + 1 ≤ k ≤ min (u, v)
    (B') ключ (k) ≥ ключ–главного–элемента для max (u, v) ≤ k ≤ j
    (C') u ≤ v + 1
    (D') существует k, такое, что u < k ≤ j
      и ключ (k) ≥ ключ–главного–элемента}
(1) → **повторять** u ← u + 1 **до** ключ (u) ≥ ключ–главного–элемента
(2) → **повторять** v ← v – 1 **до** ключ (v) ≤ ключ–главного–элемента
  поменять элементы u и v;
если u > v **то** {u = v.+ 1}
  поменять элементы u и v;
  u ← u – 1
поменять элементы u и v;
{здесь массив разделен индексом u}
Деление ← u

Строгое доказательство инвариантности (A'), (B'), (C) и (D) и правильности алгоритма труднее, чем в предыдущем случае, так как u ≤ v не является больше инвариантным свойством, чем определяется присутствие «min (u, v)» и «max (u, v)», а не u и v в (A') и (B'). Вот схема доказательства: (C) остается верным в (1) в силу (B') и в (2) благодаря (A'); если u и v пересекаются, то «дальше они не идут». С другой стороны,

в (1) (A') преобразуется в

(A') ключ (k) ≤ ключ–главного–элемента для i + 1 ≤ k ≤ u;

в (2) (B') преобразуется в

(B'_2) ключ (k) ≥ ключ–главного–элемента для v ≤ k ≤ j

(A') и (B'_2), соединенные с (C) (u ≤ v + 1), вновь дают (A') и (B'). Истинность (D) в конце каждого выполнения внешнего цикла доказывается от противного: если бы (D) было ложно, то из (B') следовало бы, что все элементы a[i + 1 : j] имели бы ключи, меньшие, чем ключ i, что противоречило бы выбору главного элемента (ср. комментарий после поменять элементы с индексами i и индекс–главного–элемента).

Заметьте, что совершенно нет никакой необходимости в свойстве, по которому выбранный главный элемент *не является абсолютным минимумом* (т.е. что существует по крайней мере один элемент с большим или равным ключом); это свойство обеспечивается здесь методом среднего элемента. Этот пункт главный, потому что без него алгоритм мог бы зациклиться, как мы это видели.

Если внутренние циклы по u и по v завершаются, то завершается и алгоритм: действительно, каждое из присваиваний u ← u + 1 и v ← v + 1 выполняется по крайней мере один раз. Так как условие, управляющее циклом **пока**, есть u < v и тем более u ≤ v + 1, то (C) – инвариант цикла и v – u + 1 есть управляющая величина для этого цикла. Цикл по u завершается; действительно, как следует из (D), существует k > u, такое, что ключ (k) ≥ ключ–главного–элемента. Цикл по v завершается: имеет место ключ(i) = ключ–главного–элемента. Завершающий оператор

если u > v **то** {u = v + 1}
  поменять элементы и и v;
  u ← u – 1

преобразует (A') и (B') с учетом (C) в

(A) ключ (k) ≤ ключ–главного–элемента для i + 1 ≤ k ≤ u

(B) ключ (k) ≥ ключ–главного–элемента для u + 1 ≤ k ≤ j

что и представляет собой разыскиваемое заключительное условие (H); этим завершается доказательство.

Можно было бы отказаться от этого «корректирующего» финального оператора, записав цикл в виде

```
пока u < v повторять
    | повторять u ← u + 1 до ключ (u) ≥ ключ–главного–элемента;
    | повторять v ← v – 1 до ключ (v) ≤ ключ–главного–элемента
    | если u < v то
    |     | поменять элементы u и v
```

Завершающая коррекция позволяет избежать выполнения проверки u < v при каждом выполнении цикла: она корректирует попадание, когда выполняется один лишний обмен. (Речь идет о классической задаче управляющих структур: средство для представления цикла, выполняемого «n + 1/2» раз (ср. упражнение III.4), состоит в том, чтобы представить его как цикл, выполняемый «n + 1» раз, с последующим оператором, аннулирующим последнюю «половину».) Такую предосторожность можно будет рассматривать бесполезной и приводящей к путанице. Мы хотели показать на важном примере, как далеко можно довести оптимизацию алгоритма; такой подход оправдывается только для «ядра» некоторого важного алгоритма, т.е. для наиболее часто выполняемой его части, время выполнения которой является критическим (что, впрочем, не всегда имеет место для сортировки). Мы вернемся к этим проблемам в следующей главе (VIII.4.7, «эффективность, оптимизация»).

Чтобы закончить с Делением, начальное определение среднего индекса–главного–элемента можно записать (с двумя или тремя проверками) в виде

```
переменные середина: ЦЕЛ,
                ключ–i, ключ–j, ключ–середины: КЛЮЧИ;
середина ← ⌊(i + j)/2⌋, ключ–i ← ключ(i); ключ–j ← ключ(j);
ключ–середины ← ключ(середина);
индекс–главного–элемента ←
    (если ключ–i ≤ ключ–середины то
        (если ключ–середины ≤ ключ–j то середина
        иначе (если ключ–i ≤ ключ–j то j
                иначе
                )
        )
        иначе {ключ–i > ключ–середины}
                (если ключ–середины ≥ ключ–j то середина
                иначе (если ключ–i ≥ ключ–j то j
                        иначе
                        )
                )
        )
```

VII.3.6.5. Исключение рекурсии и эффективная реализация

Читателю, добравшемуся до этого места, мы очень советуем подвергнуть испытанию свое понимание Быстрой Сортировки и заодно методов реализации рекурсии, рассмотренных в гл. VI попытавшись записать Быструю Сортировку как подпрограмму ФОРТРАНа.

Такая программа, *TR1RAP,* показана ниже. Задуманная как общая подпрограмма, она получает в качестве аргумента не массив, а две подпрограммы (*SUBROUTINE* и *FUNCTION* соответственно) обработки массива: *ECHANG* и *COMPCL* Их назначение:

CALL ECHANG (I, J)

должна менять местами два заданных пользователем элемента сортируемого массива:

COMPCL (I, J)

имеет результатом целое, которое равно –1, если ключ(*I*) < ключ(*J*), 0, если ключ(*I*) = ключ(*J*) и 1, если ключ(*I*) > ключ(*J*) Как это принято в ФОРТРАНе для

аргументов–«подпрограмм», *ECHANG* и *COMPCL* объявлены *EXTERNAL* (IV.4.6).

Эта подпрограмма *TRIRAP* содержит несколько усовершенствований по сравнению с «механическим» переводом рекурсивной версии. Главное из них–идея, принадлежащая Седжуику; вместо того чтобы записывать в стек пары [*I*, *J*] соответствующие малым подмассивам (*J – I < SEUIL*), а затем выбирать их из стека и сортировать включением, их просто «бросают», но в конце *TRIRAP* выполняют сортировку включением всего массива целиком – сортировку, которая должна быть быстрой, поскольку массив разделен на возрастающие «куски» размером не больше порога: максимальное число сравнений при этом $\sum_{i=1}^{m} \frac{t_i(t_i-1)}{2}$, где $t_i, \ldots, t_m$ – последовательность размеров кусков, т.е. менее $\frac{m(\text{порог})^2}{2}$ сравнений, m – число кусков. Таким образом исключается часть управления стеком и повторяющаяся инициализация сортировки включением. В связи с Быстрой Сортировкой полезно посмотреть упражнения VII.5, VII.6 (устойчивость алгоритма), VII.7 и VII.8.

ФОРТРАН

```
      SUBROUTINE TRIRAP (N, ECHANG, COMPCL) INTEGER N
      EXTERNAL ECHANG, COMPCL INTEGER COMPCL C
C
C     ********************************************************************
C     * * *                                                          * * *
C     * * *     ВОСХОДЯЩАЯ СОРТИРОВКА ПО МЕТОДУ ХОАРА               * * *
C     * * *     ("QUICKSORT" МАССИВА, К КОТОРОМУ                     * * *
C     * * *     ОБРАЩАЮТСЯ С ПОМОЩЬЮ ПОДПРОГРАММ                     * * *
C     * * *     ECHANG (I, J) (ПЕРЕСТАНОВКА ДВУХ ЭЛЕМЕНТОВ)          * * *
C     * * *     И COMPCL (I, J) (СРАВНЕНИЕ ДВУХ КЛЮЧЕЙ)              * * *
C     * * *     (РЕЗУЛЬТАТ –1, 0 ИЛИ 1 В ЗАВИСИМОСТИ ОТ              * * *
C     * * *     ТОГО, КЛЮЧ (I) МЕНЬШЕ, РАВЕН ИЛИ БОЛЬШЕ              * * *
C     * * *     КЛЮЧА (J))                                           * * *
C     ********************************************************************
C
      INTEGER SEUIL
      INTEGER SEPAR, I, J, ISUIV, JSUIV
      INTEGER NIVPIL, PILE (40)
      INTEGER PARTIT
C     SEUIL – ПОРОГ, SEPAR – РАЗДЕЛИТЕЛЬ, ISUIV – СЛЕДУЮЩЕЕ,
C     JSUIV – СЛЕДУЮЩЕЕ, PILE – СТЕК, NIVPIL – УРОВЕНЬСТЕКА
      DATA SEUIL/15/
C     --- ИНИЦИАЛИЗАЦИЯ ---
      NIVPIL= 0
      IF (N – 1 .LE. SEUIL) GOTO 1000
          I = 1
          J = N
100       CONTINUE
C             --- ЗДЕСЬ J – I > ПОРОГ (ИНВАРИАНТ ЦИКЛА) ---
              SEPAR = PARTIT (ECHANG, COMPCL, I, J)
              IF (SEPAR – 1 .GT. J – SEPAR) GOTO 130
                  ISUIV = SEPAR + 1
                  JSUIV = J
                  J = SEPAR – 1
```

```
                GOTO 300
130           ISUIV = I
              JSUIV = SEPAR–1
              I = SEPAR + 1
C
C             --- ВОЗМОЖНОЕ ЗАПОЛНЕНИЕ СТЕК А ---
300           IF (JSUIV – ISUIV .LE. SEUIL) GOTO 400
                  NIVPIL = NIVPIL + 2
                  PILE(NIVPIL) = ISUIV
                  PILE (NIVPIL – 1) = JSUIV
C
C             --- ИЗМЕНЕНИЕ ПОДМАССИВА С ВЫБОРКОИ ИЗ СТЕКА ЕСЛИ
C                НЕОБХОДИМО
400           IF (J–I GT SEUIL) GOTO 100
            IF (NIVPIL. EQ. 0) GOTO 1000
              I = PILE(NIVPIL)
              J = PILE(NIVPIL – 1)
              NIVPIL = NIVPIL – 2
              GOTO 100 C
1000    CONTINUE
C       --- ЗАКЛЮЧИТЕЛЬНАЯ ПРОСТАЯ СОРТИРОВКА ВКЛЮЧЕНИЕМ ---
        CALL TRIINS (N, ECHANG, COMPCL)
        RETURN
        END
```

ФОРТРАН

```
        INTEGER FUNCTION PARTIT (ENCHANG, COMPCL, I, J)
C       PARTIT – ДЕЛЕНИЕ
        INTEGER I, J
        EXTERNAL ECHANG, COMPCL
        INTEGER COMPCL
C
C       ***************************************************************
C       * * *     ДЕЛЕНИЕ МЕЖДУ ИНДЕКСАМИ I И J,                  * * *
C       * * *      ПРЕДНАЗНАЧЕННОЕ ДЛЯ ВОСХОДЯЩЕЙ СОРТИРОВКИ      * * *
C       * * *     МЕТОДОМ ХОАРА МАССИВА, К КОТОРОМУ               * * *
C       * * *     ОБРАЩАЮТСЯ ПОДПРОГРАММЫ                         * * *
C       * * *     ECHANG (I, J) (ПЕРЕСТАНОВКА ДВУХ ЭЛЕМЕНТОВ)     * * *
C       * * *     И COMPCL (I, J) (СРАВНЕНИЕ ДВУХ КЛЮЧЕЙ)         * * *
C       * * *     (РЕЗУЛЬТАТ –1, 0 ИЛИ 1 В ЗАВИСИМОСТИ ОТ ТОГО,   * * *
C       * * *     КЛЮЧ (I) МЕНЬШЕ, РАВЕН                          * * *
C       * * *     ИЛИ БОЛЬШЕ КЛЮЧА (J))                           * * *
C       ***************************************************************
C
        INTEGER MILIEU, U, V
C       MILIEU – СЕРЕДИНА
        PARTIT = I
        IF (J .LE. I) GOTO 5000
            MILIEU = (I + J)/2
            U = 1
            V = J + 1
C
C           --- УСТАНОВКА В ПОЗИЦИЮ СРЕДНЕГО ЭЛЕМЕНТА ---
```

```
            IF (COMPCL (J, MILIEU) .LT. 0) CALL ECHANG (J, MILIEU)
            IF (COMPCL (J, I) .LT. 0) CALL ECHANG (J, I)
            IF (COMPCL (I, MILIEU) .LT. 0) CALL ECHANG (I, MILIEU)
C
C           /ПОКА U < V ПОВТОРЯТЬ/
1000        CONTINUE
C               --- ЦИКЛ ПО U ---
1100            CONTINUE
                    U = U + 1
                    IF (COMPCL(U, I) .LT. 0) GOTO 1100
C               --- ЦИКЛ ПО V ---
1200            CONTINUE
                    V = V – 1
                IF (COMPCL(V, I). GT. 0) GOTO 1200
C
                CALL ECHANG (U, V)
                GOTO 1000
2000        IF(U GT V) CALL ECHANG (U, V)
            U = MIN0(U, V)
            CALL ECHANG (I, U)
            PARTIT = U
5000    RETURN
        END
```

ФОРТРАН

```
        SUBROUTINE TRIINS (N, ECHANG, COMPCL)
C       TRIINS – СОРТИРОВКА ВКЛЮЧЕНИЕМ
        INTEGER N
        EXTERNAL ECHANG, COMPCL
        INTEGER COMPCL
C
C       ***************************************************************
C       * * *                                                     * * *
C       * * *     ВОСХОДЯЩАЯ СОРТИРОВКА ПРОСТЫМ                  * * *
C       * * *     ВКЛЮЧЕНИЕМ МАССИВА, К КОТОРОМУ                 * * *
C       * * *     ОБРАЩАЮТСЯ С ПОМОЩЬЮ ПОДПРОГРАММ               * * *
C       * * *     ECHANG (I, J) (ПЕРЕСТАНОВКА ДВУХ ЭЛЕМЕНТОВ)    * * *
C       * * *     И COMPCL (I, J) (СРАВНЕНИЕ КЛЮЧЕЙ)             * * *
C       * * *     (РЕЗУЛЬТАТ –1, 0 ИЛИ 1В ЗАВИСИМОСТИ ОТ ТОГО    * * *
C       * * *     КЛЮЧ (I) МЕНЬШЕ, РАВЕН ИЛИ БОЛЬШЕ              * * *
C       * * *     КЛЮЧА (J))                                     * * *
C       * * *                                                     * * *
C       ***************************************************************
C
        INTEGER K, L
C
C       / ДЛЯ K ОТ 2 ДО N ПОВТОРЯТЬ /
        K = 2
100     IF (K .GT. N) GOTO 1000
            L = K – 1
C           /ПОКА L> = 1 И COMPCL (L, L + 1) > 0 ПОВТОРЯТЬ/
150         IF (L.LT.1) GOTO 170
            IF (COMPCL(L, L + 1) .LE. 0) GOTO 170
```

```
                CALL ECHANG (L, L+ 1)
                L = L – 1
                GOTO 150
170         K = K + 1
            GOTO 100
C
1000    RETURN
        END
```

VII.3.7. Сортировка Извлечением: Древесная Сортировка

Сортировка извлечением, которую можно описать в нерекурсивной форме, состоит из n – 1 этапов; k–й этап разыскивает элемент с максимальным ключом среди тех, которые еще не отсортированы окончательно, и привязывает его к позиции n - k + 1

```
программа Сортировка Извлечением
    (модифицируемое данное массив a[1 : n])
    переменные макс: КЛЮЧ,
                индекс–макс: ЦЕЛ,
    для i от n до 2 шаг –1 повторять
        макс ← ключ (i); индекс–макс ← i;
        для j от i – 1 до 1 шаг –1 повторять
            если ключ (j) > макс то
                макс ← ключ (j);
                индекс–макс ← j;
        поменять элементы i и индекс–макс
    {здесь подмассив a[i : n] отсортирован {инвариант цикла}
```

Можно было бы просматривать массив в обратном направлении. Тогда k–й этап размещает наименьший из еще не отсортированных элементов в позицию k.

Эта программа имеет сложность $0(n^2)$ во всех случаях, при этом число выполнении внутреннего цикла равно $\frac{n(n-1)}{2}$. Можно было бы сразу предложить некоторые ее улучшения, но легко видеть основной недостаток метода: выполняемые на каждом этапе сравнения дают намного более богатую информацию, чем та, что эффективно используется, чтобы установить i–й элемент на его место. Этот часто встречающийся алгоритм ненамного лучше Пузырьковой Сортировки.

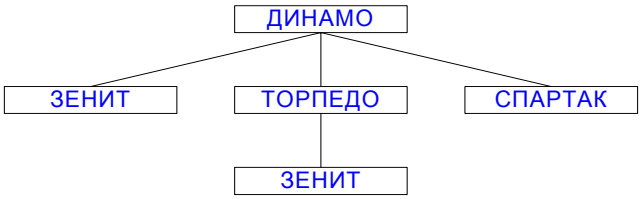
Чтобы добиться существенного улучшения, следует использовать более развитую структуру данных, позволяющую, насколько возможно, сохранять информацию, получаемую последовательными проверками. Идея фактически заимствуется у организаторов спортивных чемпионатов. Если

«Динамо» победило «Зенит»,
«Динамо» победило «Торпедо»,
«Динамо» победило «Спартак»
и «Торпедо» победило «Зенит»[1],

то ясно, что «Динамо» – победитель (незавершенного) турнира (эта команда победила три другие команды, среди которых одна победила четвертую); чтобы определить участника, занявшего второе место, нет необходимости начинать все сначала: им не может быть «Зенит», побежденный командой «Торпедо». Ситуация

[1] В оригинале фигурируют названия западноевропейских футбольных клубов.– *Прим. перев.*

может быть изображена деревом, где y является сыном x тогда и только тогда, когда команда x – победитель команды y:



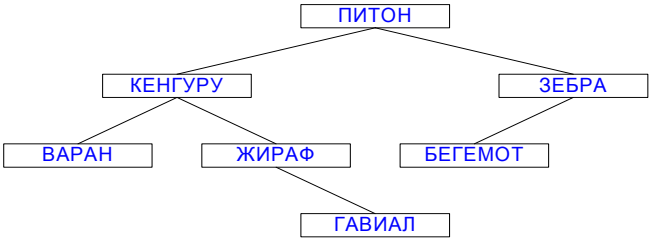
Рассмотрение такого дерева сыгранных матчей позволяет ограничиться только вершинами глубиной 2 для определения участника, занявшего второе место и т.д., корректируя дерево на каждом этапе в зависимости от результатов. Так можно уйти от бесполезного просмотра результатов многочисленных матчей среди $\frac{n(n-1)}{2}$ возможных.

Было построено много алгоритмов сортировки, основанных на этом принципе и использующих деревья. Последовательные усовершенствования, предложенные, в частности, Уильямсом и Флойдом, привели к алгоритму, который сейчас здесь обсуждается под именем Древесной Сортировки (Heapsort).

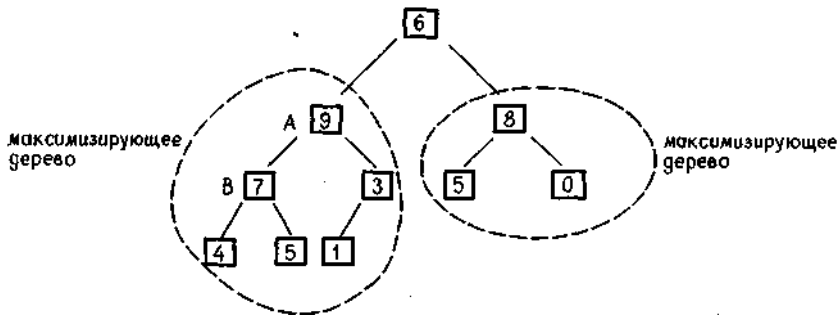
Древесная Сортировка использует конкретное двоичное дерево, с которого легко «собирать» готовый максимум и которое из этих соображений называется **максимизирующим деревом** (дерево, которое дает максимум или, если угодно, максимумы, как яблоня дает яблоки).

Максимизирующее дерево характеризуется интересным свойством: данные, связанные с каждой вершиной, которая представляет собой объект типа ЭЛЕМЕНТ, содержит ключ, больший или равный ключам двух сыновей этой вершины, если они существуют.

Например, следующее двоичное дерево является максимизирующим деревом для алфавитного порядка, если вершины его изображают только ключи:



Если вернуться к задаче сортировки, сразу же видно, что максимальный элемент задается корнем, следующий элемент–один из двух его сыновей, но не будем забегать вперед. Интерес к максимизирующему дереву определяется следующим свойством: допустим, что два поддерева двоичного дерева являются максимизирующими; при этом взятое целиком двоичное дерево не является необходимо максимизирующим, если с корнем связывается произвольный ключ. Например,



Чтобы реорганизовать дерево с тем, чтобы сделать его максимизирующим, достаточно пройти некоторый путь, исходящий от корня (максимальной длины h,

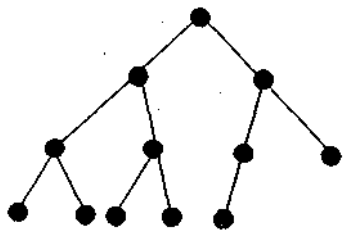
глубины дерева), а не все дерево целиком. Здесь, например, элемент ключа 6 переместится в вершину, обозначенную B, а 9 и 7 «поднимутся» соответственно на один уровень.

Тогда применяемый алгоритм можно записать, предполагая обычное определение двоичных деревьев (VII.2.4):

```
программа Реорганизация
    (модифицируемое данное a: ДВОДЕР_ЭЛЕМЕНТ)
    {формирование максимизирующего дерева a при условии, что поддеревья
    слева (a) и справа (a) – исходно максимизирующие}
    переменные x, y: ДВОДЕР_ЭЛЕМЕНТ,
                значкор: ЭЛЕМЕНТ;
    значкор ← корень (a); x ←a;
    пока x имеет по крайней мере одного сына,
    ключ которого ≥ ключ (значкор) повторять
        | y ← сын x, имеющий наибольший ключ;
        | корень (x)← корень (y);
        | x ←y;
    корень (x) ← значкор
    {здесь a – максимизирующее дерево}
```

Реорганизация выполняется почти как цикл Сортировка Включением, заставляя подниматься элементы y с более «тяжелыми» ключами кверху по дереву. Но существует и важное отличие: цикл **пока** выполняется здесь самое большее $h - 1$ раз, где h – глубина дерева. В силу того что нам известно о полных двоичных деревьях, $h \le \log_2 n + 1$, где n – число вершин дерева. Следовательно, Реорганизация имеет сложность $O(\log n)$ на полных максимизирующих деревьях.

Но какую пользу может принести двоичное дерево, если мы ограничиваемся задачей сортировки массивов? Ответ прост: потому что массив – это двоичное дерево. В V.7.6 мы видели, что «полное» двоичное дерево, т.е. такое, в котором до некоторого уровня существуют все вершины, кроме, возможно, самых «правых» вершин последнего уровня, допускает физическое сплошное представление в виде массива, где левый и правый сыновья элемента с индексом i имеют соответственно индексы $2i$ и $2i + 1$. И обратно, всякий массив может рассматриваться как представление полного двоичного дерева.



При таком представлении нужно применить подпрограмму Реорганизация к двоичному поддереву двоичного дерева, соответствующего массиву $a[1 : n]$. Это поддерево выделяется двумя индексами i и j ($1 < i < j < n$), которые могут быть переданы подпрограмме в качестве параметров; сыновья элемента, представляемого индексом x, будут иметь тогда индексы $2x$ и $2x + 1$, если эти значения не превосходят j.

Подпрограмма Реорганизация, применяемая к части массива, записывается таким образом:

```
программа Реорганизация (модифицируемое данное
                          массив a[l : n]: ЭЛЕМЕНТ;
                          аргументы i, j: ЦЕЛЫЕ)
{преобразование, обеспечивающее, что в двоичном дереве,
соответствующем массиву a, двоичное поддерево, содержащее элементы
a[i], a[i + 1], ..., a[j], образует максимизирующее дерево. Исходная гипотеза:
1 ≤ i ≤ j ≤ n, а поддеревья, соответствующие (a[2i], ..., a[j] и (a[2i + 1], ..., a[j],
если они определены, являются максимизирующими деревьями}
переменные  x, левый–сын, правый–сын, следующий: ЦЕЛЫЕ,
            левый–ключ, правый–ключ: КЛЮЧ;
следующий ← i;
повторять
    x ← следующий; левый–сын <– 2 × x; правый–сын ← левый–сын + 1;
    левый–ключ ← (если левый–сьш ≤ j то ключ (левый–сын)
                  иначе –∞);
    правый–ключ ← (если правый–сын ≤ j то ключ (правый–сын)
                   иначе –∞);
    если левый–ключ > ключ (x) то
        следующий ← левый–сьн;
    если правый–ключ > ключ (следующий) то
        следующий ← правый–сын;
    поменять элементы x и следующий;
до x = следующий
```

Если обмены более трудоемки, чем полуобмены, то можно немного улучшить алгоритм, воспользовавшись тем, что во всех последовательных обменах участвует один и тот же элемент (исходно a[i]), и заменить в цикле обмен элементов x и следующий на

```
a [x] ← a [следующий]
```

при условии, что в начале следует (после следующий ← i)

```
значкор ← a [i]
```

а в заключение (на выходе из цикла **повторять...до**)

```
a [следующий] ← значкор
```

где значкор – объявленная локальная переменная типа ЭЛЕМЕНТ. Доказательство правильности программы Реорганизация (в версии «двоичное дерево» и в версии «массив») оставлено читателю.

В связи с комментарием, включенным в начало последней программы, можно заметить, что двоичное дерево, соответствующее *подмассиву* a[i : j], это не то же самое, что поддерево двоичного дерева, которое соответствует a, содержащему только элементы с индексами, заключенными между i и j. В первом случае действительно корень имеет своим индексом i, а сыновья элемента с индексом k, если они существуют, имеют индексы 2k – i + 1 и 2k – i + 2; Во втором – это элементы 2k и 2k + 1 (корень всего дерева имеет индекс 1). Именно с поддеревьями второго типа (поддеревьями двоичного дерева, соответствующего a[1 : n]) работает программа Реорганизация.

Программа Реорганизация представляет интерес тем, что ее сложность равна $0(h_{ij})$, где h. – глубина поддерева, соответствующего элементам a[i], a[i + l], ..., a[j] в двоичном дереве, связанном с a. Поэтому ясно, что

$$h_{ij} = \lfloor \log_2(j - i + 1) \rfloor + 1$$

Таким образом, сложность Реорганизации равна O(log(j – i)), более точно, она выполняет не более $2\lfloor \log_2(j - i + 1) \rfloor$ сравнений и $\lfloor \log_2(j - i + 1) \rfloor + 2$ полуобменов.

Программа собственно сортировки, Древесная Сортировка, разделяется на две части, каждая из которых использует Реорганизацию. Первая, называемая Посадка, преобразует исходный массив в максимизирующее дерево; вторая часть названа Сортировка Максимизирующего Дерева, она выполняет собственно сортировку, учитывая структуру максимизирующего дерева:

```
программа Древесная Сортировка
| (модифицируемое данное массив a[1 : n]: ЭЛЕМЕНТ)
| Посадка (a);
| Сортировка Максимизирующего Дерева (a)
```

Чтобы написать Посадку, заметим, что Реорганизация применяется к таким индексам i, что сыновья i, если они–существуют, являются уже максимизирующими деревьями, а i, заключенные между $\left\lfloor \frac{n}{2} \right\rfloor + 1$ и n, и представляют именно такой случай, так как они неимеют сыновей. Поэтому естественно действовать рекуррентно, двигаясь «назад». Алгоритм записывается в виде

```
программа Посадка
|   (модифицируемое данное массив a[1 : n]: ЭЛЕМЕНТ)
|   для 1 от ⌊n/2⌋ до 1 шаг –1 повторять
|   |   {инвариант цикла: a[i + 1 : n] – максимизирующее дерево}
|   |   Реорганизация (i, n)
```

Посадив максимизирующее дерево, нам остается собрать с него плоды, а заодно и плоды наших усилий. Действительно, a[1] имеет теперь значение максимума массива. Могут возразить, конечно, что массив надо упорядочивать в *возрастающем* порядке, а не в убывающем, тогда как подпрограмма начала делать противоположное. Стоит ли об этом говорить! Достаточно выполнить

```
поменять элементы 1 и n
```

чтобы a[n] стало теперь максимальным элементом; остается только отсортировать a[1 : n – 1]. Но этот подмассив сам по себе представляет собой максимизирующее дерево с единственным возможным исключением, касающимся элемента a[1], бывшего a[n]; поэтому к подмассиву применяют подпрограмму Реорганизация, чтобы получить максимум подмассива в a[1], а затем поменять a[1] и a[n – 1] и т.д. Таким образом, сортировка максимизирующего дерева записывается в виде

```
программа Сортировка Максимизирующего Дерева
| (модифицируемое данное массив a[1 : n]: ЭЛЕМЕНТ)
| {сортировка массива a, который исходно считается максимизирующим
| деревом}
| переменная i: ЦЕЛ;
| i ← n;
| пока i ≥ 2 повторять
|   | {инвариант цикла: a[i + 1 : n]–упорядоченный подмассив, содержащий
|   | n – i наибольших элементов a, а подмассив a[1 : i] – максимизирующее
|   | дерево, содержащее остальные i элементов}
|   | поменять элементы 1 и i;
|   | i ← i – l;
|   | Реорганизация (1, i)
```

Древесная Сортировка – это простая цепочка Посадки и Сортировки Максимизирующего Дерева. Какова сложность алгоритма? Сложность Посадки, выводимая из сложности Реорганизации, равна

$$\sum_{i=\lfloor n/2 \rfloor}^{n} h_{in}$$

где h_{in}–глубина двоичного поддерева, соответствующего a[i : n]. Так как среди рассматриваемых подмассивов, соответствующих вершинам дерева, существуют

1 подмассив глубины $h = \lfloor \log_2 n \rfloor + 1$ массив a[1 : n]

2 подмассива глубины h – 1: a[2 : n] и a[3 : n]

4 подмассива глубины h – 2

...

2^{h-2} подмассивов глубины 2 (подмассивы a[i : n] для $\lfloor n/4 \rfloor < i \le \lfloor n/2 \rfloor$

то сложность Посадки можно записать в виде

$$O(1 \times h - 2 \times (h - 1) + 4 \times (h - 2) + \ldots + 2^{h-2} \times 2) = O(2^{h-1}) = O(n)$$

Что касается Сортировки Максимизирующего Дерева, ее сложность равна

$$T(n) = \sum_{i=2}^{n} O(h_{ij})$$

где h_{ij} есть глубина поддерева, соответствующего a[1 : i].

Поскольку $h_{ij} = \lfloor \log_2 i \rfloor + 1$, получают

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

Таким образом, сложность Древесной Сортировки равна $O(n + n\log n) = O(n\log n)$. Важным обстоятельством является отсутствие какой–либо гипотезы о распределении ключей: *средняя и максимальная сложности Древесной Сортировки равны* $O(n\log n)$. Следовательно, этот метод–самый **надежный** из всех рассмотренных. В практических результатах появляется, однако, константа, превосходящая соответствующую величину для Быстрой Сортировки в ее «благоприятных» случаях.

На следующей странице представлена фортрановская версия этого алгоритма, написанная с учетом ранее сделанных соглашений (в частности, использованы обмены, а не полуобмены; это можно легко изменить, если известен тип массивов, к которым применяется подпрограмма).

ФОРТРАН

```
      SUBROUTINE TRIARB (N, ECHANG, COMPCL)
C     TRIARB – ДРЕВЕСНАЯ СОРТИРОВКА
      INTEGER N
      EXTERNAL ECHANG, COMPCL
      INTEGER COMPCL C C
C     ***********************************************************************
C     * * *                                                           * * *
C     * * *     ВОСХОДЯЩАЯ СОРТИРОВКА ПО МЕТОДУ                      * * *
C     * * *     ФЛОЙДА–УИЛЬЯМСА МАССИВА, К КОТОРОМУ                   * * *
C     * * *     ОБРАЩАЮТСЯ С ПОМОЩЬЮ ПОДПРОГРАММ                      * * *
C     * * *     ECHANG (I, J) (ПЕРЕСТАНОВКА ДВУХ ЭЛЕМЕНТОВ)           * * *
C     * * *     И COMPCL (I, J) (СРАВНЕНИЕ ДВУХ КЛЮЧЕЙ)               * * *
C     * * *     (РЕЗУЛЬТАТ –1, 0 ИЛИ 1 В ЗАВИСИМОСТИ ОТ ТОГО,         * * *
C     * * *     КЛЮЧ (I) МЕНЬШЕ, РАВЕН ИЛИ БОЛЬШЕ КЛЮЧА (J))          * * *
C     * * *                                                           * * *
C     ***********************************************************************
C
      INTEGER I
C
C     --- ФОРМИРОВАНИЕ МАКСИМИЗИРУЮЩЕГО ДЕРЕВА
C     ИЗ МАССИВА ---
C     ("ПОСАДКА")
C     /ДЛЯ I ОТ N/2 ДО 1 ШАГ –1 ПОВТОРЯТЬ/
      I = N/2
100   IF(L .LE. 0) GOTO 200
C           --- "РЕОРГАНИЗАЦИЯ" МЕЖДУ ИНДЕКСАМИ I И N ---
            CALL REORG (I, N, ECHANG, COMPCL)
            I = I – 1
            GOTO 100
C
C           --- СОРТИРОВКА МАКСИМИЗИРУЮЩЕГО ДЕРЕВА ---
C     /ДЛЯ I ОТ N ДО 2 ШАГ –1 ПОВТОРЯТЬ/
200   I = N
300   IF (I. LE. 1) GOTO 1000
C            --- ЗДЕСЬ (ИНВАРИАНТ ЦИКЛА): ЭЛЕМЕНТ С ИНДЕКСОМ 1
C            ИМЕЕТ КЛЮЧ, РАВНЫЙ ИЛИ ПРЕВОСХОДЯЩИЙ ОСТАЛЬНЫЕ I – 1
C            КЛЮЧЕЙ МАССИВА ---
             CALL ECHANG (1, I)
             I = I – 1
C            --- "РЕОРГАНИЗАЦИЯ" МЕЖДУ ИНДЕКСАМИ 1 И I ---
             CALL REORG (1, I, ECHANG, COMPCL)
             GOTO 300
      C
      C
1000  RETURN
      END
```

ФОРТРАН

```
      SUBROUTINE REORG (K, L, ECHANG, COMPCL)
      INTEGER K, L
      EXTERNAL ECHANG, COMPCL
      INTEGER COMPCL
C     ********************************************************************
C     * * *                                                          * * *
C     * * *     "РЕОРГАНИЗАЦИЯ" МЕЖДУ ИНДЕКСАМИ К И L ДЛЯ            * * *
C     * * *     ВОСХОДЯЩЕЙ СОРТИРОВКИ ПО МЕТОДУ                     * * *
C     * * *      ФЛОЙДА–УИЛЬЯМСА МАССИВА, К КОТОРОМУ                * * *
C     * * *     ОБРАЩАЮТСЯ С ПОМОЩЬЮ ПОДПРОГРАММ                    * * *
C     * * *     ECHANG (I, J) (ПЕРЕСТАНОВКА ДВУХ ЭЛЕМЕНТОВ)         * * *
C     * * *     И COMPCL (I, J) (СРАВНЕНИЕ ДВУХ КЛЮЧЕЙ)             * * *
C     * * *     (РЕЗУЛЬТАТ –1, 0 ИЛИ 1 В ЗАВИСИМОСТИ ОТ ТОГО,       * * *
C     * * *     КЛЮЧ (I) МЕНЬШЕ, РАВЕН ИЛИ БОЛЬШЕ КЛЮЧА (J))       * * *
C     * * *                                                          * * *
C     ********************************************************************
C
      INTEGER J, GAUCHE, DROITE, INDMAX
C     GAUCHE–СЛЕВА, DROITE–СПРАВА
C
C
      J = K
100   GAUCHE = 2* J
            IF (GAUCHE .GT. L) GOTO 1000
            INMAX = GAUCHE
            DROITE = GAUCHE + 1
            IF (DROITE.GT.L) GOTO 150
                IF (COMPCL (GAUCHE, DROITE) .LT. 0) INDMAX = DROITE
150         IF (COMPCL (IN DM AX, J).LE.O) GOTO 1000
                CALL ECHANG (J, INDMAX)
                J = INDMAX
                GOTO 100 C
1000  RETURN
      END
```

VII.3.8. Понятие сортировки распределением

Если не очень углубляться в сортировку распределением, то кажется интересным представить алгоритм, использующий этот метод, который отвечает самым различным из рассмотренных до сих пор принципам и, в частности, не выполняет никаких сравнений ключей: значение ключа или компоненты ключа используется, чтобы обратиться *прямым доступом* к области, в которой размещен элемент.

Предположим, что каждый ключ с имеет K «компонент» и каждая компонента может принимать M различных значений; для того чтобы алгоритм был эффективным, надо, чтобы M оставалось относительно небольшим (несколько десятков). Например, ключи могли бы быть текстами из K алфавитных литер: i–я компонента–это i–я буква текста, а M = 26[1]. Этот метод можно в такой же мере применять и для числовых ключей, представляемых последовательностями цифр в десятичной или некоторой другой системе.

Не выполняя никаких сравнений, алгоритм требует области вспомогательной

[1] Здесь снова число 26 обусловлено количеством букв во французском алфавите.– *Прим. перев.*

памяти для M файлов, где M – число возможных значений для каждой компоненты c_i; это типичный пример компромисса «пространство – время». Будем рассматривать c_i целым, заключенным между 1 и M.

```
программа Сортировка Распределением (модифицируемое данное
                                      массив a[1 :N]: ЭЛЕМЕНТ)
    переменные c:КЛЮЧ, e:ЭЛЕМЕНТ;
    массив корзина [1 : M]: ФАЙЛ_элемент;
    {M «корзин» служат для временного размещения элементов}
    для i от K до 1 шаг –1 повторять
        для j от 1 до M повторять
            корзина [j] ← ПУСТО;
        для m от 1 до N повторять
            c – i–я компонента ключа (m);
            включение (a[m], корзина [c])
            {включение в c–й файл};
        {передать вновь элементы файлов в массив}
        i ← 0;
        для j от 1 до M повторять
            пока корзина [j] ≠ ПУСТО повторять
                e ← выборка (корзина [j]);
                {извлечение элемента e}
                (*)     i ← i + 1;
                a[i] ← e
            {здесь необходимо i = N}
```

Видно, что этот алгоритм оперирует с самим массивом a как с файлом. Временная сложность алгоритма равна, очевидно, O(K(M + N)) (заметьте, что цепочка действий, помеченная (*), выполняется точно N раз при каждом выполнении внешнего цикла по i); Пространственная сложность равна O(M + N) при цепном представлении (потому что все файлы вместе никогда не содержат более N элементов).

Когда ключи имеют переменную длину, можно использовать вспомогательный массив, для получения алгоритма с временной оценкой O(L + N), где L – сумма длин ключей, так, что простое применение предыдущего алгоритма давало бы O(K(M + N)), где K – максимум длин ключей. Это расширение оставлено в качестве упражнения.

Заметьте также, что, применяя предыдущий алгоритм к случаю K = 1, получают интересный алгоритм для ситуации, в которой число M возможных значений ключей относительно невелико; как пространственная, так и временная сложности алгоритма в этом случае равны O(M + N).

VII.3.9. Практическое сравнение различных методов

Рассмотренные методы используют очень разные принципы, и выбор может показаться затруднительным. Он зависит, очевидно, от многих факторов; надеемся, однако, что читатель увидел, как можно просто и быстро запрограммировать каждый из этих методов, и обращение к такому алгоритму, как Пузырьковая Сортировка, по той причине, что «его можно быстро написать», не оправдает себя ни в коем случае.

Оставляя за пределами нашего «банка алгоритмов» сортировку распределением, отвечающую задачам особого типа, можно сделать следующие рекомендации:

- для малых n (например, до 100 и, может быть, больше), если не пытаться «наскрести» несколько микросекунд, сортировка простым включением дает совершенно достаточные результаты, особенно если данные обладают уже некоторым порядком;
- для n от нескольких сот до нескольких тысяч метод Шелла дает отличный

результат. Есть у него, однако, некоторое неудобство: в системах с виртуальной памятью его не следует применять, если массив располагается на большом числе страниц;

- для n > 100 (примерно) Быстрая Сортировка является, вероятно, наилучшим алгоритмом в общем случае; но, как было показано, он может «вырождаться» до $O(n^2)$ с ненулевой вероятностью, хотя и весьма малой, если хорошо написано Деление;
- при n > 100 Древесная Сортировка требует примерно вдвое больше времени в среднем, чем Быстрая Сортировка, но зато гарантировано ее поведение с O(nlogn).

В показанной ниже таблице приведены сравнительные **экспериментальные временные величины** различных методов, примененных к реальному массиву, в котором каждый элемент был своим собственным ключом. Массив был образован из псевдослучайных элементов; это обстоятельство несколько благоприятствует Быстрой Сортировке по сравнению с реальными ситуациями, но уж совсем трудно определить понятие «типового не случайного массива»! Заметим, что выводы могут оказаться разными в зависимости от сравнительной стоимости сравнений ключей, обменов и других базовых операций, но здесь также трудно сказать, что такое «типичная ситуация».

Тесты были выполнены на ФОРТРАНе на ИБМ 370 модели 168–2 с операционной системой MVS (виртуальная память). Пустые клетки соответствуют непро–водившимся испытаниям: нет необходимости расходовать машинное время, чтобы доказать, что слишком медленная программа не должна использоваться!

n	10	100	1000	10000	25000	50 000
Пузырьковая Сортировка	0,16	20	2400			
Сортировка Извлечением	0,12	7,3	650			
Сортировка Включение*	0,12	6,7	610			
Сортировка Шелла	0,07	2	37	600	1800	4200
Древесная Сортировка	0,2	3,5	50	660	1830	3960
Быстрая Сортировка (порог =16)	0,07	2	28	365	1000	2140

Времена в таблице заданы в **миллисекундах.** Версии тестированных подпрограмм работали непосредственно с массивом вещественных так, что каждый элемент массива был его собственным ключом:

```
SUBROUTINE TRI (N,A)
INTEGER N
REAL A(N)
```

Это позволило обойтись без подпрограмм сравнения ключей и перестановки элементов, что оказалось существенным из–за все той же иллюзорности определения «типичной» ситуации.

VII.4. Обработка текстов: алгоритм «сопоставления с образцом»

VII.4.1. Задача и тривиальный алгоритм

Представляемый здесь алгоритм–это модифицированная версия алгоритма, описанного Ахо и Карасиком в статье 1975 г. Этот алгоритм, который опирается на содержащиеся в многочисленных предшествующих публикациях идеи и предлагает их усовершенствования, интересен, помимо своей большой эффективности, возможностью применения к важной нечисловой предметной области–обработке текстов. Он очень ясно иллюстрирует идею компромисса «интерпретация–компиляция», которая, как говорилось, является важным средством построения алгоритмов.

Кроме практического интереса описываемого метода, ориентированного на обработку длинных текстов, какие часто встречаются в приложениях, наличие его в этой секции оправдывается еще и критерием, который мог бы показаться ненаучным, – критерием эстетическим. Описываемый алгоритм действительно кажется нам на редкость изящным.

Задача **сопоставления с образцом** (pattern matching) формулируется так: рассматривается текст t; надо обнаружить, встречаются ли в тексте t некоторые другие заданные тексты, называемые **ключевыми словами.**

Более строго, предположим, что текст t образован из n литер, обозначаемых

t[l], t[2], ..., t[n]

и существует m ключевых слов, обозначаемых $M_1, M_2, ..., M_m$.

Каждое из ключевых слов имеет длину, которая обозначается длина(M_i) ($1 \le i \le m$). Ключевое слово образовано из литер:

$M_i[1], M_i[2], ..., M_i[$длина $(M_i)]$

Для каждого ключевого слова M_i требуется определить, является ли M_i подтекстом t, т.е. существует ли k такое, что

$$\begin{cases} 1 \le k \le n - \text{длина}\ (M_i) + 1 \\ t[k] = M_i[1] \\ t[k+1] = M_i[2] \\ t[k+2] = M_i[3] \\ \ldots \\ t[k + \text{длина}(M_i - 1)] = M_i[\text{длина}(M_i)] \end{cases}$$

В качестве примера рассмотрим текст

t = "ANNIE N'HONNIT NI NINA NI IRENE"[1]

и ключевые слова

$$\begin{cases} M_1 = \text{"NI"} \\ M_2 = \text{"REIN"} \\ M_3 = \text{"RENE"} \\ M_4 = \text{"IRENE"} \end{cases}$$

В этом тексте M_1 встречается пять раз, M_3 и M_4 – по одному разу, а M_2 не встречается ни разу:

[1] Эта фраза, имеющая, вообще говоря, смысл во французском языке, умышленно не переведена: это позволяет с одной стороны, подчеркнуть важное обстоятельство формальности обработки текстов, а с другой–сохранить приводимые авторами в этом разделе примеры. – *Прим. перев.*

ANNIE N'HONNIT NI NINA NI IRENE

Наиболее типичным приложением такого алгоритма является, очевидно, документальный поиск: задан фонд документов, состоящий из последовательности библиографических ссылок; каждая ссылка сопровождается "дескрипторами", указывающими темы соответствующих ссылок; надо найти некоторые ключевые слова, встречающиеся среди дескрипторов. Мог бы иметь место, например, запрос

("ПРОГРАММИРОВАНИЕ" **и** "ВЫЧИСЛИТЕЛЬНАЯ МАШИНА") **или**
("ИНФОРМАТИКА" **и** ~"ЭЛЕКТРОННАЯ СХЕМА")

Такой запрос можно трактовать следующим образом: существуют ли статьи, обладающие дескрипторами "ПРОГРАММИРОВАНИЕ" и "ВЫЧИСЛИТЕЛЬНАЯ МАШИНА" или дескриптором "ИНФОРМАТИКА" без дескриптора "ЭЛЕКТРОННАЯ СХЕМА". При помощи такого запроса можно ограничиться статьями, относящимися к программному обеспечению ЭВМ, но не к техническому.

Ответ на такой вопрос требует просмотра текста t (в нашем примере – фонда документов) для того, чтобы знать, встречается ли в нем каждое из выделенных ключевых слов.

Практически учитываются два обстоятельства:

- вообще говоря, недостаточно обнаружить вхождение ключевого слова в текст: нужно еще выдать позицию (индекс) каждого из таких вхождений;
- часто бывает важно рассматривать только такие вхождения слов, которые выделены разделителями пробелами или какими–нибудь знаками пунктуации; это позволит не обращать внимание на обнаружение в слове

  "ОБИТЕЛЬ"

  вхождения ключевого слова "БИТ", которое к задаче не имеет ни малейшего отношения. Для этого достаточно включить разделители в состав ключевых слов; можно, например, интересоваться ключевым словом "БИТ".

Существует тривиальный алгоритм, решающий задачу с учетом этих двух правил:

```
программа распознавание (аргументы t, M1, M2, ..., Mm: СТРОКИ)
    переменная равно: ЦЕЛ;
    для i от 1 до m {число ключевых слов} повторять
        {определение, встречается ли в тексте t ключевое слово Mi}
        для j от 1 до n – длина (Mi) + 1 повторять
            {определение, встречается ли Mi в тексте t в позиции j}
            равно ← 0;
            пока равно < длина (Mi) и t[j + равно] = Mi[равно +1] повторять
                равно ← равно + 1;
            если равно = длина(Mj) то
                "i–е ключевое слово встречается в в позиции j"
```

Этот алгоритм, названный здесь «тривиальным алгоритмом», для каждого ключевого слова повторяет просмотр текста t Это очень плохо: его минимальная сложность (для случая, в котором начальные литеры ни одного из ключевых слов не встречаются в тексте t) равна O(m ×. n) (если m много меньше n).

Среднюю сложность алгоритма невозможно определить строго, потому что не ясны понятия «типовых» или «средних» текстов и наборов ключевых слов. Читатель может исследовать сложность алгоритма в случае (экстремальном!), когда

t = "UUUU...U"
n литер

M_1 = "U"

M_2 = "UU"

...

M_m= "UU...U"

m литер

Во всяком случае алгоритм очень неэффективен, а значит, неприменим для значительных фондов документов (n – от сотен тысяч до нескольких миллионов литер, m – от нескольких единиц до нескольких десятков при группировании поисков).

Неэффективность алгоритма связана с тем, что самый внутренний цикл всегда возвращается «к нулю», не сохраняя никакой информации о предшествующих проверках; надо сделать возможным сохранение такой информации.

На нашем примере это очевидно:

t = "ANNIE N'HONNIT N1 NINA N1 IRENE"

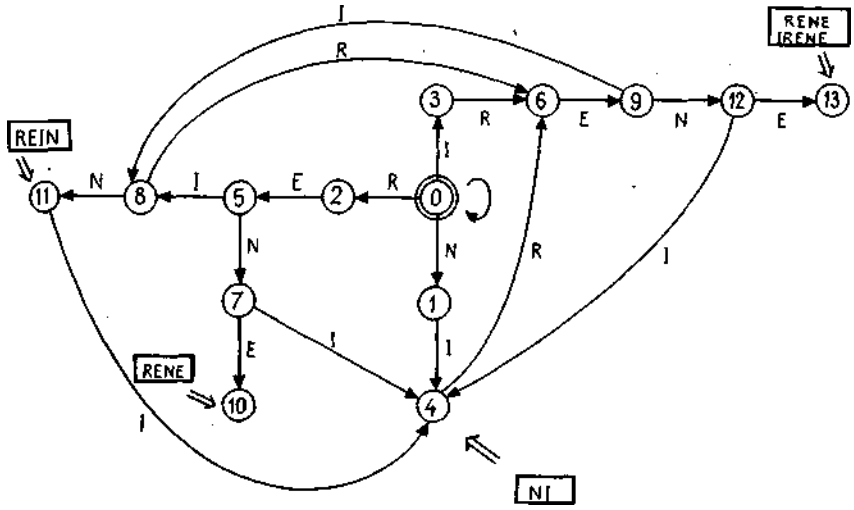
M_t = "NI", M_2 = "REIN", M_3 = "RENE", M_4 = "IRENE"

Разыскивая M_t в t, сравнивают "N" –первую литеру M_t–с первой литерой t и обнаруживают различие; затем сравнение со второй литерой текста t дает совпадение. Тогда сравнивают "I" в M_t со следующей литерой t, "N"; совпадения нет. Вновь возвращаются к первой литере M_t, "N", чтобы сравнивать ее с третьей литерой t. Ясно, что это сравнение бесполезно, потому что эта литера только что рассматривалась и было обнаружено, что она была "N". Однако эта информация не была сохранена.

Таким образом, недостаток этого метода состоит в том, что он повторно *интерпретирует* набор ключевых слов, не сохраняя в памяти их взаимоотношения; например, тот факт, что всякое "N", даже не приводящее к успеху при поиске ключевого слова, может быть началом "NI"; точно так же то, что "RE", сопровождаемое "N", означает неудачу в поиске "REIN", но может привести к успеху при поиске "RENE", наконец, наличие ключевого слова "IRENE" означает сразу же присутствие слова "RENE".

Одним словом, для улучшения тривиального алгоритма надо заменить интерпретацию ключевых слов *компиляцией.*

Форма, выбираемая в качестве результата такой компиляции, представляет собой достаточно естественно диаграмму переходов типа той, какая была рассмотрена в III.3.6. В нашем примере диаграмма перехода, полученная для представления ключевых слов "NI", "REIN", "RENE" и "IRENE", такова:



Эта диаграмма перехода содержит *состояния,* отмеченные кружочками и пронумерованные (здесь от 0 до 13), и *переходы* из состояния в состояние, указанные стрелками и помеченные буквами. Некоторым состояниям соответствуют множества

ключевых слов (связываемых с соответствующим состоянием с помощью двойной стрелки); такое множество называется *результатом* рассматриваемого состояния.

Один из переходов соединяет состояние 0 с самим собой: предполагается, что этот переход имеет в качестве метки любую из букв, не являющихся метками других переходов, начинающихся в состоянии 0 (т.е. здесь все, кроме I, R и N).

Интерпретация этой диаграммы переходов для задачи сопоставления с образцом очевидна: текст обрабатывается, следуя диаграмме и отправляясь от состояния 0. На каждом этапе, если мы находимся в состоянии i, исследуется первая, еще не рассмотренная литера текста t, например л. Если существует переход, начинающийся в состоянии i, помеченный меткой л и ведущий в состояние j, то переходят в состояние j; в противном случае используется переход, помеченный меткой л и начинающийся в состоянии 0 (такой переход существует по построению). Каждый раз при переходе через состояние, в котором множество «результат» не пусто, фиксируется, что обнаружены все ключевые слова, принадлежащие этому множеству.

VII.4.2. Алгоритм сопоставления образцов

Программа сопоставления, как и все программы, «управляемые таблицами» (III.3.6), имеет простую структуру. Считается, что существует числит возможных литер; что число состояний диаграммы перехода, отличных от состояния 0, есть числосостояний; что переходы представляются массивом

```
массив переход [0 : числосостояний, 1 : числит]: ЦЕЛ
```

т.е. если с – номер состояния, а л – литера, то переход [с, л] есть номер состояния, в которое приходят по диаграмме, отправляясь из состояния с, когда прочитанной литерой является л. Наконец, множество «результат», возможно, пустое, соответствующее состоянию с, обозначается

```
результат [с]
```

результат является массивом, представляющим множество ключевых слов:

```
массив результат [0 : числосостояний]: МНОЖЕСТВО_строка
```

В предположении, что числосостояний и массивы переход и результат известны, программа распознавания записывается в виде

```
программа распознавание (аргументы: СТРОКИ, числосостояний: ЦЕЛ,
                массив переход [0 : числосостояний, 1 : числит]: ЦЕЛ,
                массив результат [0: числосостояний]: МНОЖЕСТВО_строка)
    переменная состояние: ЦЕЛ;
    состояние ← 0;
    для i от 1 до длина (t) повторять
        состояние ← переход [состояние, t[i]];
        если результат [состояние] ≠ ПУСТО то
            для из результат [состояние] повторять
                зафиксировать, что m встретилось в тексте t между
                индексами i– длина (m) + 1 и i
```

По сравнению с тривиальным алгоритмом этот алгоритм является значительным шагом вперед: его временная сложность равна O(n), где n = длина (t) и не зависит от количества и длин ключевых слов; какой бы ни была диаграмма переходов, каждая литера текста t просматривается один и только один раз. Разумеется, к этой оценке временной сложности следует прибавить время построения диаграммы, но мы увидим, что оно пренебрежимо мало, если длина ключевых слов существенно меньше длины текста.

Мы «поменяли» пространство и время, потому что взамен на улучшение временной оценки приходится хранить в памяти представление диаграммы переходов – массивы переход и

результат. Будет показано, однако, что можно использовать представление, не намного более дорогое, чем непосредственно сами ключевые слова, которые приходится хранить и в случае тривиального алгоритма.

VII.4.3. Алгоритм построения

Для построения диаграммы переходов алгоритм Ахо и Корасика использует три этапа и дополнительный массив, названный неудача. Запишем алгоритм в виде

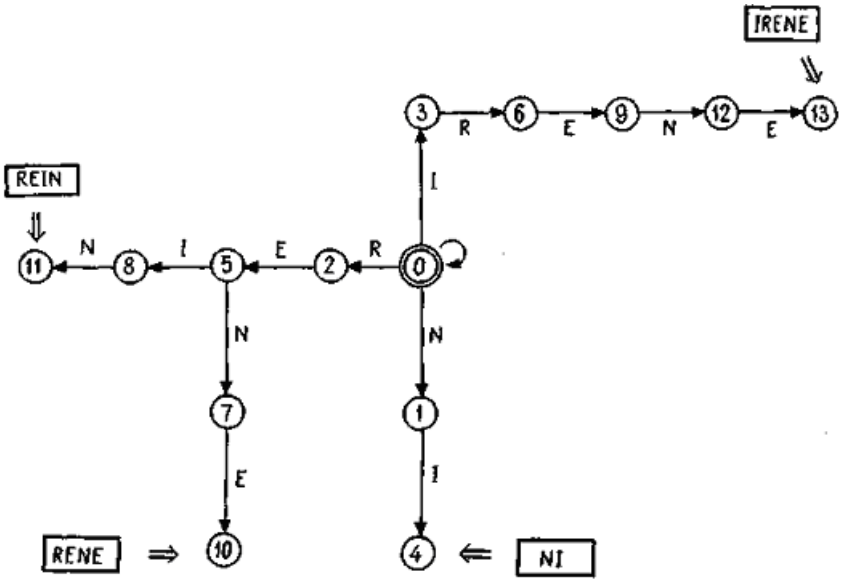
переменная числосостояний: ЦЕЛ;
массивы переход [0 : N, 1 : n лит]: ЦЕЛ
неудача [0 : N]: ЦЕЛ,
результат [0 : N]: МНОЖЕСТВО$_{строка}$;
построение–дерева;
построение–неудачи;
дополнение

Число N, выбранное в качестве размера массивов (и необходимое в силу того, что значение числосостояний заранее неизвестно), равно

$$N = \sum_{i=1}^{m} \text{длина}(M_i)$$

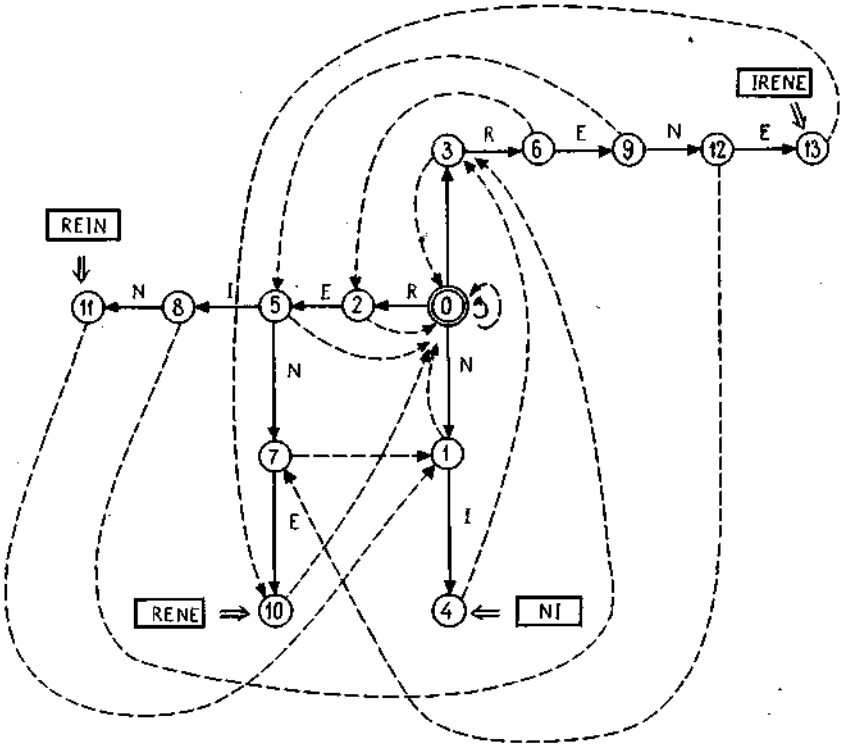
поскольку окончательное число состояний не должно превышать сумму длин ключевых слов.

Прежде всего вкратце посмотрим, что делают три этапа построения диаграммы, построение–дерева создает «скелет» диаграммы в виде дерева, которое, как и в случае тривиального алгоритма, совершенно не учитывает многообразия отношений между ключевыми словами. Результатом для нашего примера служит схема



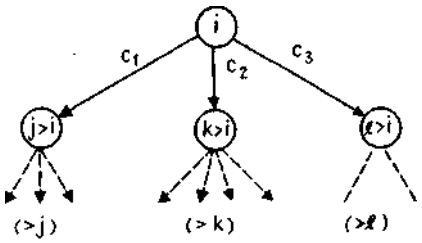
Подпрограмме построение–неудачи поручено на основании предыдущего дерева обнаружить подтексты, общие для ключевых слов, с целью заполнения массива неудача; неудача[с] – это «лучшее» состояние, из которого можно отправиться, находясь в состоянии с при прочтении литеры л такой, что никакой помеченный этой литерой переход не исходит из с. Заметьте, что неудача[с] существует всегда, потому что, по крайней мере, всегда можно отправляться от состояния 0. На приведенной ниже схеме значения неудача[с] изображены штриховыми стрелками. Например, в состоянии 9, «распознавшем» "IRE", если исследуемая литера не есть N, нельзя идти к состоянию 12, но, может быть, можно прийти в это состояние из 5, соответствующего "RE" (которое есть начало "REIN"), и поэтому проведена штриховая стрелка из 9 в 5: неудача [9] = 5.

Заметим, что неудача [0] = 0 и что для всякого с последовательность неудача[с], неудача[неудача[с]] и т.д. заканчивается, достигнув состояния 0. .



Наконец, последний этап – дополнение – состоит в пополнении дерева диаграммы всеми переходами, вытекающими из значений массива неудача Так, поскольку в нашем примере неудача[9] = 5, а переход[5, I] = 8, можно добавить переход с меткой I из 9 в 8: переход [9, I] = 8. Этап дополнение прибавляет также элементы массива результат[неудача[с]] для каждого состояния с к множеству результат[с]; например, множество результат[13] содержит не только "IRENE", но и "RENE", поскольку неудача [13] = 10, а результат[10] содержит "RENE".

Теперь можно написать алгоритм. Этап построение–дерева прост; заметим, однако, что для дальнейшего полезно пронумеровать состояния в порядке возрастания уровней; т.е. всякое состояние с будет иметь номер, меньший, чем номера его «сыновних» состояний – состояний, соответствующих литерам л, таким, для которых определено значение переход [с, л]:



В целом нумерация состояний произвольна; учитывается только это последнее ограничение. Подпрограмму построение–дерева можно теперь записать так:

```
переменная новое–состояние : ЦЕЛ;
массив последнее–состояние [1 : m] : ЦЕЛ;
числосостояний ← 0;
для с от 0 до N повторять
    | для всех литер л повторять
    |     | переход [с, л] ← 0;
для i от 1 до m {число ключевых слов } повторить
    | последнее–состояние[i] ← 0;
для j от 1 до максимум(длина (Mi)) повторять
    | для i от 1 до m {число ключевых слов) повторять
    |     | если j ≤ длина(Mi) то
    |     |     | новое–состояние ← переход [последнее–состояние [i], Mi[j]];
    |     |     | если новое–состояние = 0 то
    |     |     |     | {создание нового состояния}
    |     |     |     | числосостояний ← числосостояний + 1;
    |     |     |     | новое–состояние ← числосостояний;
    |     |     |     | переход [последнее–состояние [i], Mi[j]] ←новое состояние
    |     |     | {иначе уже существующий переход Mi у которого по крайней мере
    |     |     | первые j литер совпадают с первыми литерами другого ключевого
    |     |     | слова Mk с k < i};
    |     |     | если j = длина [Mi] то
    |     |     |     | включение M; в множество результат[новое–состояние]
    |     |     |     | последнее–состояние [i] ← новое–состояние
```

Замечание: Этот алгоритм можно описать проще, рассматривая последовательно все ключевые слова, а не последовательные позиции от 1 до максимум (длина (M_i)). Но тогда, чтобы считаться с ограничением, наложенным на номера состояний, нужно выполнить затем «топологическую сортировку» (см. упражнение V.1).

Вторая часть, построение–неудачи, должна для каждого состояния с присваивать элементу неудача[с] значение *состояния, которому– соответствует самый длинный суффикс текста, соответствующего состоянию с,* договорившись, что:

- каждому **состоянию** соответствует **текст,** образованный из последовательности литер, помечающих переходы на пути, который ведет из 0 в это состояние в дереве; так, в нашем случае состояние 13 соответствует "IRENE", состояние 8 – "REI", состояние 1 – "N" и т.д.;
- **суффикс** (собственно) текста x, образованного литерами x[1], x[2], ..., x[m], это текст, образованный из литер x[j], x[j + 1], ..., x[m] при $1 < j \le m$. Так, "ЗВУК" – это суффикс слова "УЛЬТРАЗВУК", "ЗИНА" – это суффикс слова "КОРЗИНА" и т.д.

Следующая версия построения–неудачи решает задачу; она основывается на обеспечиваемом построением–дерева свойстве, по которому состояния нумеруются в возрастающем порядке уровней в дереве.

```
переменная новое–состояние: ЦЕЛ;
неудача [0] ← 0;
для всех литер л повторять
    если переход [0, л] ≠ 0 то
            | неудача [переход [0, л]] ← 0
    {все состояния уровня 1 имеют значение неудачи, равное 0}
для с от 1 до числосостояний
{т.е. для всех состояний, кроме 0, в порядке возрастания уровней}
```

```
повторять
    {инвариант цикла: для всякого состояния i, такого, что 0 ≤ i ≤ c, неудача[i]
    имеет окончательное значение:
            неудача[i] есть состояние, соответствующее самому длинному
            суффиксу текста, связанного с i;
              0 < неудача [i] < i}
    для всех литер повторять
            новое–состояние ← переход [c, л]
            если новое–состояние ≠ 0 то
                {определение значения неудача[новое–состояние]}
                x ← c;
                повторять
                    x ← неудача [x];
                    y ← переход [x, л]
                до y ≠ 0 или x = 0;
                неудача [новое–состояние] ← y
```

Цикл **повторять ... до** имеет следующую интерпретацию. Пусть требуется определить значение «неудачи», соответствующее состоянию новое–состояние. Пусть C_1 C_2, ..., C_{j-1}, C_j – литеры текста, соответствующего новому–состоянию; текст, соответствующий c, – это $C_1C_2...Cj–1$ и имеет место $c = C_i$. неудача [новое состояние] – это состояние, соответствующее самому длинному суффиксу $C_1C_2...C_i$. Такой суффикс, следовательно, имеет вид αC_i, где α – суффикс текста, соответствующего c. По гипотезе рекуррентности (инвариант цикла) неудача [c] известна; она меньше c, и соответствующий текст есть самый длинный суффикс α, например β, которому соответствует некоторое состояние. Точно так же неудача [неудача [c]] соответствует самому длинному суффиксу β, с которым связано некоторое состояние, и т.д.

Последовательность неудача[c], неудача[неудача[c]], неудача[неудача [неудача[c]]] и т.д., которая становится известной при достижении 0, содержит, таким образом, все состояния, соответствующие суффиксам c в порядке убывания длин этих суффиксов. Первый элемент x этой последовательности, такой, что y = переход [x, C_j] определен, или такой, что x = 0, представляет собой состояние, соответствующее наиболее длинному возможному суффиксу $C_1C_2... C_i$.

Наконец, алгоритм дополнение использует массив неудача, чтобы дополнить диаграмму перехода:

```
для с от 1 до числосостояний
{т.е. для всех состояний, отличных от 0, в порядке возрастания уровней}
повторять
    для всех литер с повторять
        если переход [c, л] = 0 то
            переход [c, л] ← переход неудача [c, л];
    результат [c] ← результат [c] ∪ результат [неудача [c]]
```

VII.4.4. Временная сложность

Какова временная сложность алгоритма построения?

а) построение–дерева имеет, очевидно, сложность 0(m x M), где M –максимум длин ключевых слов;

б) дополнение имеет сложность O (числосостояний x числит), где числит–число литер;

в) сложность алгоритма построение–неудачи вычисляется несколько тоньше в связи с наличием самого внутреннего цикла повторять... до; априори можно было бы подумать, что она имеет вид O (числосостояний x числит x

числосостояний). На самом деле это не так: доказывается (примерно так же, как в упражнении VII.9), что сложность алгоритма построение–неудачи тоже представляет собой O(числосостояний × числит).

Поскольку известно, что

$$\text{числосостояний} \le \sum_{i=1}^{m} \text{длина}(M_i) \le m \times \text{максимум}(\text{длина}(M_j)), i = 1,2,...,m$$

то сложность алгоритма построения равна

O(m × M × числит)

или, рассматривая числит как константу,

O(m × M)

где M – максимум длин ключевых слов.

VII.4.5. Пространственная сложность. Структуры данных

Коэффициент числит, который входит во временную сложность и в размер массива переход, ((N + 1) × числит), может создать некоторые затруднения. Действительно, ясно, что на практике массив переход будет «разреженным»: значения переход[с, л] для заданного состояния с равны переход[0, л] для всех литер л, за небольшим исключением. Поэтому естественно применять особые способы представления для этого массива, с тем чтобы обеспечить быстрый доступ к нетривиальным элементам массива переход. Это сразу позволит уменьшить временную сложность алгоритмов построение–неудачи и дополнение, где внутренние циклы для могут быть представлены в виде

```
для всех литер л таких, что переход [с, л] ≠ 0 повторять
    | ...
```

и снизить пространственную сложность алгоритма (размер массива переход) до k × числосостояний, где k – некоторая малая константа.

Методы представления разреженных массивов были рассмотрены в V.8 и применяются здесь. Напомним, что в основном существует два типа таких методов:

а) Приведение компромисса «пространство–время» к компромиссу вида «прямой доступ–цепной доступ», размещая в массиве размера числосостояний × s, где s невелико (например, s = 3 или 4), s наиболее часто используемых переходов для каждого состояния; остальные нетривиальные переходы организуются в виде цепи

```
тип  СПИСОК СОСТОЯНИЙ =(ПУСТО | НЕПУСТОЙСПИСОК);
тип  НЕПУСТОЙСПИСОК =
       (начало: ЦЕЛ; продолжение: СПИСОК СОСТОЯНИЙ);
тип  ЭЛЕМЕНТ = (прямой: массив [1 : s]: ЦЕЛ,
                                  цепной: СПИСОК СОСТОЯНИЙ);
массив переход [0 : числосостояний]: ЭЛЕМЕНТ;
```

б) Представление массива переход в виде таблицы ассоциативной адресации; входами в такую таблицу служат номер состояния и литера; в таблице тоже хранятся только нетривиальные переходы.

В алгоритме, впервые представленном Ахо и Корасиком, в массиве переход размещались только переходы, соответствовавшие исходному дереву; массив неудача должен в этом случае передаваться алгоритму сопоставления. Этот метод является предметом упражнения VII.9, которое полезно попытаться решить, чтобы лучше понять идею метода.

БИБЛИОГРАФИЯ

Две главные ссылки, касающиеся принципов написания и анализа алгоритмов, – это «энциклопедия» Кнута [Кнут 68, Кнут 69, Кнут 73; еще четыре тома должны выйти в свет] и менее полный, но принципиально отличающийся обзор Ахо, Хопкрофта и Ульмана [Ахо 74]. Существует введение в методы анализа алгоритмов на французском языке [Кнут 76]. На французском см. также [Мал 77].

Библиография по алгоритмам сортировки весьма значительна. Есть объемная библиографическая статья [Лорен 71].

Алгоритму Быстрой Сортировки, введенному впервые Хоаром [Хоар 62], была посвящена важная работа [Седжуик 75]; можно обратиться также к статьям [Седжуик 77], [Седжуик 77а], которые анализируют конкретные положения.

Алгоритм сопоставления с образцом, описанный в этой главе, в компактной форме представлен в [Ахо 74]; он был разработан в [Ахо 75].

УПРАЖНЕНИЯ

VII. 1. O в математике

Исследуйте математические свойства операции «O» («порядка...»), определенной в разд. VII.1.1.

VII.2. Минимум и максимум сразу

Определение минимума *или* максимума в массиве из n элементов требует, очевидно, по меньшей мере n – 1 сравнений ключей. Доказать (применяя принципы «разделяй и властвуй» и «равновесие»), что существует алгоритм, позволяющий определить одновременно минимум *и* максимум массива примерно за 3n/2 сравнений (уточните эту величину), а не за 2n – 2 сравнений, необходимых, когда один определяется вслед за другим. (*Рекомендация*: Возьмите для начала $n = 2^k$.)

VII.3. Отправляясь от середины

При ассоциативной адресации, когда используются внутреннее разрешение коллизий и линейные упорядоченные списки (VII.2.5.5), может показаться интересным двигаться в том или ином направлении (+ приращение или = приращение) в зависимости от того, больше или меньше разыскиваемый ключ, чем ключ первого рассмотренного элемента (t[индекс]) который, таким образом, будет занимать «середину» линейного списка находящихся в коллизии элементов. Этот метод должен позволить вдвое сократить длину обрабатываемого линейного списка как при включении, так и при поиске. Насколько хороша эта идея?

VII.4. K–й наименьший

Пусть имеется массив a из n элементов с ключами. Условимся называть «k–м наименьшим элементом» из a (для $1 \le k \le n$) элемент, ключ которого не меньше k – 1 других ключей массива и не больше n – k других ключей массива; такой элемент не обязательно единственный, если не все ключи различны. Докажите, что не обязательно сортировать массив для определения k–го наименьшего элемента и что, в частности, подпрограмма Деление, используемая для Быстрой Сортировки (VII.3.6), позволяет построить эффективный алгоритм для решения этой задачи. Исследуйте сложность алгоритма.

VII.5. Пять последних наносекунд

Программист, стремящийся сделать Быструю Сортировку еще более эффективной, мог бы заметить, что приведенная в VII.3.6.5 фортрановская версия в некоторых случаях выполняет бесполезные засылки в стек, так как тут же следуют выборки из стека (случай, когда *JSUIV – ISUIV > SEUIL,* но *J – I ≤ SEUL*). Можно ли исправить этот «недостаток»? Какое практическое улучшение можно ожидать от такой коррекции?

VII.6. Устойчивость Быстрой Сортировки

Стабильна ли Быстрая Сортировка? Исследуйте соотношение равных ключей в двух версиях подпрограммы Деление.

VII.7. Деление заданным значением

Доказать, что подпрограмма Деление в Быстрой Сортировке может быть эффективно использована для разделения массива t[a : b] заданным значением x, т.е. найти f такое, что

$$\begin{cases} a \le f \le b \\ \text{для всякого } i, \text{ такого, что } a \le i \le f, \text{ ключ } (i) \ge x \\ \text{для всякого } i, \text{ такого, что } f < i \le b, \text{ ключ } (i) \le x \end{cases}$$

Какова сложность алгоритма?

VII.8. Французский флаг

[Dijkstra 76] Есть набор из N кеглей, который можно описать с помощью массива

массив t[1 : N]: КЕГЛЯ

Устройство, управляемое вычислительной машиной, умеет выполнять операцию
переставить (i, j) {1 ≤ i ≤ j ≤ N}

Это значит, что устройство берет кеглю t[i] и кеглю t[j] и переставляет их местами.

Кроме того, оптический датчик этого устройства может выполнять проверку
цвет (i) {1 ≤ i ≤ N}

определяющую цвет кегли t[i]. Возможные цвета – голубой, белый и красный.

Требуется написать программу, реорганизующую массив так, чтобы кегли располагались, как на французском флаге[1]. Строго построить программу и доказать ее правильность, опираясь на процедуру Деление в Быстрой Сортировке (VII.3.6.4).

(*Замечание:* Нет гарантии, что в массиве будут представлены все три цвета.)

VII.9. Сопоставление без дополнения

Алгоритм построения диаграммы для задачи сопоставления образцов (VII.4.3) в том виде, в каком он был первоначально представлен в [Ахо 75], не включал этап, который мы назвали дополнение. Доказать, что без него можно эффективно обойтись, модифицировав алгоритм сопоставления, который должен будет иметь доступ не только к массиву переход, но также и к массиву неудача. Доказать, что при этом можно сохранить O(n) в качестве сложности алгоритма распознавания (опираясь на разд. V.4.4.в).

[1] Напомним читателю, что на полотнище французского флага цвета следуют в таком порядке: голубой, белый, красный. – *Прим. перев.*

VII.10. Указатель

Вы только что написали книгу по алгоритмам и программированию и желаете автоматически составить предметно–именной алфавитный указатель.

Вы имеете список записанных (например, на перфокартах) слов с указанием номера страницы. Такой список может содержать, например, элементы

499 Быстрая Сортировка Деление Дейкстра Французский флаг

500 Ахо Альфред Распознавание образцов Сложность Указатель
Алгоритмический Программирование

Можете ли вы написать программу, которая по этому списку напечатает требуемый указатель?

ГЛАВА VIII.
НА ПУТЯХ К МЕТОДОЛОГИИ

...это было для меня открытием. Я понял, что значит пользоваться орудием, называемым алгеброй. Но, черт возьми, никто мне об этом ничего не говорил. Дюпюи постоянно изрекал на этот счет напыщенные фразы, но ни разу не произнес этих простых слов: это то самое разделение труда, которое производит чудеса, как всякое разделение труда, позволяя уму направить все свои силы на одну сторону предметов, на одно из их свойств. Все пошло бы для нас иначе, если бы Дюпюи сказал нам .«Этот сыр мягкий или жесткий; он белый или голубой; старый или молодой; мой или твой; легкий или тяжелый, из всех этих свойств будем рассматривать только вес. Каков бы ни был этот вес, назовем его А. Теперь, забыв о сыре, применим к А все то, что мы знаем о количествах». Такую простую вещь никто не мог нам сказать в этой далекой провинции...

Стендаль
Жизнь Анри Брюлара[1]

НА ПУТЯХ К МЕТОДОЛОГИИ

Здесь вы найдете все, что вы всегда хотели знать о программировании и не осмеливались об этом спросить.

VIII.1. Сомнения

Методология программирования является модным сюжетом. Авторы всевозможных книг, статей и докладов наперебой предлагают всяческие рецепты тем, кто порой с горечью сталкивается с трудностями ремесла. В последнее время продажа панацей, порой с доставкой на дом, стала доходным делом.

К такому обилию советов и рецептов как–то даже неловко добавлять свои собственные размышления, и велико искушение отделаться простой отсылкой к предыдущим главам. Вернитесь к программам, которые вам были предложены, – могли бы мы сказать читателю, пожертвовав нашими творениями. Перечитайте их, тщательно разберите, раскритикуйте. Изучите их читаемость, их правильность, их сопротивляемость к ошибкам в данных, приводимые доказательства, их эффективность. Обсудите управляющие структуры, структуры данных, характеристики языков программирования, которые в них используются. Исследуйте, легко ли их мо-

[1] Перевод с франц. Б. Г. Реизова под ред. А. А. Смирнова (Стендаль, Собр. соч. в 15 томах. Т. 13. М., изд–во «Правда», 1959).

дифицировать, чтобы они соответствовали несколько измененным спецификациям. Проанализируйте вводящие их рассуждения и проверьте, убедительны ли они, могут ли быть обобщены или, напротив, являются специфическими; и, самое главное, попробуйте написать лучшие программы. С помощью подобных размышлений вы больше узнаете о программировании, чем с помощью сотни страниц теории.

И если мы все же не отказались от написания этой последней главы, то лишь потому, что методология программирования, несмотря ни на что, представляет собой настоящую тему, достойную изучения ради нее самой. Помимо проблем, упомянутых в предыдущих главах, которые можно до известной степени считать «техническими», программист сталкивается с трудностями высшего порядка, связанными одновременно с ограничениями, внутренне присущими человеческому уму, и с общей проблемой передачи идей и разделения труда. Сложность и расплывчатость этих вопросов объясняют одновременно необходимость их исследования и трудность получения небанальных формулировок.

Разумеется, можно было бы сказать, что наслоение технических проблем и проблем человеческих не является специфической чертой информатики. С подобной ситуацией можно было бы столкнуться при изложении любой другой дисциплины; невозможно подробно говорить об урбанизме, гражданском строительстве или о телеграфной и телефонной связи, не затронув вопросов, касающихся психологии, социологии и даже политики. Однако трудность различения этих двух видов проблем в программировании усложняется оригинальной чертой, присущей этому виду деятельности: чисто нематериальным, концептуальным характером конечного продукта – программ. Это отсутствие осязаемого результата (цвет карт, диаметр диска или вес бумажного листа дают очень мало информации о программе, чьим субстратом они являются) объясняет, почему так трудно разработать объективные критерии, помогающие вынести полное суждение о том, что собой представляет работа программиста. С виду такие простые характеристики, как правильность программы или ее эффективность, на практике не вмещаются так легко в рамки строгого определения; а когда переходят к таким понятиям, как ясность программы, ее читаемость, «прозрачность», простота использования, «переносимость», когда надо оценить качества сопровождающей ее документации (если таковая существует), приходится полагаться на критерии, в большинстве своем субъективные.

Можно надеяться, что все эти оговорки послужат оправданием частного характера приводимых ниже рассуждений, задуманных скорее как перечисление важных проблем и отдельных набросков решений, чем как строгая теория методологии программирования. На самом деле мы в первую очередь хотели, чтобы эти рассуждения смогли оправдать в глазах читателя три следующих постулата, которыми мы руководствовались при составлении этой книги:

а) программирование представляет собой трудное упражнение;

б) существуют методы для систематизации некоторых его аспектов, и каждый программист должен их знать;

в) в настоящее время не существует чудодейственного средства, гарантирующего, что применением неизменных рецептов можно получить надежный желаемый результат.

VIII.2. Причины и цели

Осознание того факта, что методология программирования (и даже просто программирование) дает тему, достойную изучения, – явление новое. Часто еще подготовка программы, предназначенной для выполнения вычислительной машиной, рассматривается как самый простой и наименее престижный этап в процессе решения

задачи. Об этой позиции свидетельствует традиционное различение между «спецификацией», «концепциями», «исследованием функций», «анализом схем» и «кодированием» с убывающим порядком значимости. Другим ее свидетельством является преподавание информатики; часть, иосвященная программированию, сводится зачастую к языкам программирования и несколько сомнительным «трюкам» для улучшения эффективности программ, а основное внимание уделяется изучению логических контуров или полупроводников–вещей, очень интересных и полезных электронщикам, но бесполезных для среднего программиста.

Изменение позиции по отношению к задачам программирования, без сомнения, может быть отнесено к середине или к концу 60–х годов. Действительно, это тот момент, когда после периода экспансии и безбрежного оптимизма, сопутствующего небывалому развитию «аппаратуры» – увеличение в несколько тысяч раз скорости центральных процессоров, в несколько сотен раз объема памяти и т.д. – и сравнимому с ним возрастанию амбиций по поводу размера задач, которые можно решать, пользователи вычислительной техники начали чувствовать на своей шкуре, и порой с горечью, что «программное обеспечение» в своем развитии не поспевало за аппаратурой.

Этот возврат к более скромному взгляду на вещи принял в зависимости от области различные аспекты. Некоторые проблемы, до решения которых, казалось, рукой подать и которые возбудили большие надежды, оказались при исследовании более упрямыми, чем можно было вообразить; так было, например, в случае автоматического перевода с естественных языков, в случае мечты об «автоматизированном управлении» предприятиями, во многих областях искусственного интеллекта, таких, как роботехника. Во всех этих дисциплинах были произведены порой решительные пересмотры, и общая тенденция состоит в настоящее время в том, чтобы заниматься ограниченными задачами, которые можно надеяться решить в разумное время и разумными средствами; решение же более общих и более честолюбивых задач, если такие еще ставятся, отнесено на будущее.

В промышленности то, что называлось «кризисом программного обеспечения» 60–х годов, выразилось двумя явлениями: огромным ростом затрат на программирование и появлением в журналах по информатике отчетов о некотором числе сенсационных неудач, происшедших порой при осуществлении очень крупных проектов.

Что касается огромности затрат на информатику, некоторые цифры, даже если они и не совсем точны, не могут не впечатлять. Одна из этих цифр, цитируемая в работе [Энслоу 75], касается затрат на информатику в мире, оцениваемых в 1965 г. в 250 млрд. франков. В соответствии с другими источниками затраты на информатику одного лишь французского общественного сектора достигли в 1977 г. 8 млрд. франков. Однако в противоположность порой еще распространенному мнению эти затраты в основном и все в большей степени являются затратами не на машины, не на *техническое оборудование,* а затратами человеческими, на *программное обеспечение.* Кривая на Рис. VIII.1, представляющая, конечно, грубо эволюцию расходов на техническое оборудование и расходов на программное обеспечение по отношению к глобальным затратам на информатику, показывает, что речь идет о явлении непрерывного характера, и не видно причин, почему бы оно прекратилось.

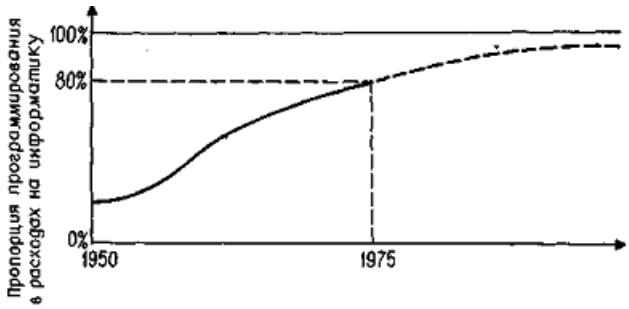


Рис. VIII.1 Сравнительная стоимость технического и программного обеспечения.

Хуже того, анализ этих затрат строго с точки зрения непосредственной рентабельности дает результаты, которые экономист может найти ужасающими. Хотя и мало подлинно научных исследований было доведено до конца (одно из них [Боем 73]), известны статистические результаты, которые, по–видимому, достаточно хорошо отражают действительность. В соответствии с одним из них, полученным отнесением размера нескольких больших проектов, измеренных в числе операторов результирующей программы (критерий, очевидно, спорный), к числу программистов и ко времени программирования, можно констатировать, что все происходит так, как будто каждый программист «производит» пять операторов в день–есть от чего побледнеть руководителю, радеющему за производительность труда. Другое исследование [USAF 72], чье быстрое распространение само по себе указывает на тот резонанс, который оно вызвало среди руководителей больших проектов программирования, было посвящено десятилетней эволюции программы, разрабатываемой администрацией, зависящей от американского министерства обороны: в соответствии с этим исследованием каждый оператор программы стоил в среднем 75 долларов при своем начальном осуществлении и 4000 долларов при последующем применении в ходе «жизни» программы (ее «сопровождения»). Даже если добавить, что речь шла о той отрасли, где требования «надежности» особенно велики (реальное время), соотношение этих двух сумм может вызвать некоторые вопросы относительно использованных методов программирования.

По легко понятным причинам еще труднее, чем подробные данные относительно затрат, найти в литературе верные и точные сведения о крупных **неудачах** программирования. Однако всем известно, что в этой области случались весьма крупные катастрофы. Так, говорят о пресловутом контракте, заключенном местными властями, ведающими мостами и дорогами, с неким предпринимателем в одной европейской стране для строительства множества одинаковых мостов. При разрушении десятого моста местные власти захотели возбудить судебное дело против этого предпринимателя, а тот в свою очередь сумел снять с себя ответственность, доказав, что ошибка кроется не в конструкции, а в представленных местными властями спецификациях, которые сами проявились в результате работы программы. Называют некий космический проект, предусматривающий запуск метеорологических спутников, при реализации которого после нескольких недель измерений спутники распались вследствие команды, которая была выработана программой через несколько секунд после запуска спутника.

> В той же области космических исследований [Хоар 73], как всегда в этой области в весьма осторожных выражениях, утверждают, что неудача запуска первого американского спутника к Венере, возможно, произошла (как говорят) из–за ошибки в программе. Исследование этой ошибки чрезвычайно интересно. Известно, что в ФОРТРАНе пробелы не являются значащими, за исключением того случая, когда они находятся в константах СТРОКА; случайная замена запятой на точку в таком, например, операторе
>
> *DO 50 I = 12,523*
>
> не делает эту строку синтаксически неверной в ФОРТРАНе: она будет

воспринята как оператор присваивания значения действительной константы *12.523* действительной переменной *DO50I;* поскольку идентификатор этой переменной начинается с *D,* известно, что не обязательно объявлять ее в явном виде как принадлежащую типу *REAL.*

Тот факт, что подобная ошибка могла явиться причиной провала всего проекта, вызывает два вопроса. Мыслимо ли для проекта подобной стоимости, требующего огромной степени надежности в малейших деталях, использовать язык, в котором транслятор может пропустить подобные ошибки? Мыслимо ли, чтобы не проводились систематические проверки, например, при помощи таблицы «перекрестных ссылок», простое обращение к которой позволяет немедленно выявить переменные, используемые всего один раз в программном модуле. Это *всегда* свидетельствует о ненормальном положении. Все это – проблема методов программирования, к которой мы еще вернемся.

Если не все программисты сталкивались непосредственно с неудачами подобного масштаба, то многие из них имели дело с одним из тех «монстров» – с гигантскими программами, которые вечно модифицируются, «улучшаются», «штопаются», дополняются и наконец становятся похожими на лицо, изъеденное оспой (образ навеян всеми этими «пороками» – малейшая модификация спецификаций приводит к цепной реакции, к кускам программы, не связанным друг с другом, и к новым ошибкам, приходящим на смену исправляемым). Часто при работе с такими программами, которые вечно «едва работают» независимо от численности группы программистов, лишь один или два человека знают код во всех деталях и знают, что если присвоить *I* значение 1, а не значение 0 в строке 8546, то *K* в строке 17451 увеличится на 2.

На обложке монографии, посвященной управлению большими проектами математического обеспечения [Брукс 75], помещена гравюра, на которой изображены динозавры и другие представители мезозойской эры, безнадежно застрявшие в трясине, в которой они по прошествии веков превратятся в ископаемые. Намерение ясно: показать, как за порогом гигантизма плохая адаптация к изменениям внешнего мира влечет за собой более или менее скорое исчезновение вида. Остается порой только пожелать, чтобы этот образ более соответствовал действительности: как много существующих программ, подобно дожившим до наших дней реликтовым животным, пережили не одно поколение программистов, таща за собой свои странности прошлых эр!

В свете весьма невысокого уровня производительности труда по сравнению с нормативами, применяемыми в других видах деятельности, в свете весьма нечетких проблем, которые трудно обойти, полагают, что руководители проектов примутся искать методы, позволяющие систематизировать процесс программирования.

Хотя волна недоверия задела пользователей информатики в *промышленности* (этот термин здесь взят в широком смысле для обозначения всех профессиональных приложений, использующих ЭВМ в качестве орудия, в противоположность научным исследованиям в области информатики), теоретики–исследователи пытались глубоко изучать проблемы программирования, которое рассматривалось как самостоятельная в *научном* отношении дисциплина. Это движение имеет, несомненно, древнее происхождение и можно наметить вехи с самого начала истории информатики (среди самых важных из этих вех следует упомянуть выполнение и издание работ Кнута– терпеливо собранные вместе элементы подробнейшего и почти исчерпывающего трактата). Однако наиболее ярким свидетельством этого было, вне всякого сомнения, издание в 1968 г. открытого письма Дейкстры «О вредности оператора *GOTO*» [Дейкстра 68].

Намеренно вызывающая, эта статья утверждала сразу, что «квалификация программистов обратно пропорциональна числу операторов *GOTO,* включаемых ими в программу», что этот оператор должен быть «уничтожен во всех языках программирования, кроме, быть может, машинных языков самого низкого уровня», и что динамика программ оценивается более адекватным образом

статической структурой, ясно показывающей ход ее выполнения благодаря циклам, альтернативам, рекурсивным вызовам и т.д.; эта точка зрения обсуждалась в гл. III.

Резонанс, который имела эта статья, и история ее последствий весьма поучительны. Со строго технической точки зрения можно пожалеть, что она была вынуждена в течение десяти лет питать (и еще не перестала это делать) оживленный спор вокруг вопроса, в конечном счете второстепенного, об операторах перехода. Еще и теперь некоторые серьезно спорят об их полезности, другие отважно предлагают языки, в которых *GOTO* отсутствует, либо автоматические методы «устранения» операторов перехода из программ, при этом подобные новшества никак не помогают программисту. Сторонники чистоты «слога» могли бы также сожалеть, что статья Дейкстры того же времени [Дейкстра 68а], по–видимому, намного более важная, не имела такого же распространения; в ней было показано, как принципы доказательства правильности программ могут быть плодотворно применены к построению программ (доказательство– и программа при этом развиваются параллельно, «рука об руку»), а не к исследованию a posteriori существующих программ. Можно также задать вопрос, имела ли важная книга [Дал 72], опубликованная в 1972 г. Далом, Дейкстрой и Хоа–ром, иное глубокое последствие, чем превращение ее названия – *структурное программирование* – в модный жаргон.

Как бы ни были оправданы эти оговорки, было бы, однако, ошибкой оспаривать post factum благотворный эффект статьи Дейкстры. Для значительного числа ее читателей (либо читателей бессчетных вариантов ее «популяризации», вышедших затем «из вторых рук») перенесенный шок был спасительным – и в меньшей мере в части существа проблемы (оператора перехода), чем в части формы, – потому что в первый раз обсуждалось качество программы ; не только говорилось, медленная ли она или быстрая, экономная или «жадная» до места в памяти, но изучалась ее ясность, ее выразительная сила. О программисте судили не по количеству строк, выданных в конце каждого месяца, а, наоборот, по качеству на первый взгляд отрицательного толка, по отсутствию некой характеристики, по воздержанности, сопротивляемости запретному плоду, по *дисциплине.* Одним словом, программа и программист стали **предметом изучения** – и это была целая революция.

Научное направление, символизируемое Дейкстрой – и к нему следует причислить наряду с многими другими имена Хоара, Вирта, Парнаса, Лисков и в некоторой степени Уорньера и Миллза, – продолжает развивать **теорию программирования,** рассматриваемого как интеллектуальная дисциплина, как методология понимания сложных проблем (см., например, [Вирт 71], [Лисков 75], [Дейкстра 76] и т.д.). Основная задача, которую оно пытается разрешить, это задача *овладения сложностью:* поскольку человеческий ум имеет явно ограниченные способности для понимания и умеет обращаться лишь с задачами небольших размеров, возникает вопрос, как можно при помощи систематической декомпозиции свести задачи очень большой сложности к комбинации .простых задач, чтобы при этом синтез решений можно было легко осуществить?

Достаточно очевидно, что когда начинают заниматься вопросами такого рода, то не всегда встречают тот же отклик, как в случае, когда предлагают непосредственно применимые решения. Достаточно компетентный и пользующийся высоким научным и личным авторитетом руководитель проекта может без лишнего шума «уничтожить операторы *GOTO*» в руководимых им программах. А сделать привычной другую недавно высказанную Дейкстрой мысль:

Программирование–одна из самых трудных отраслей, прикладной математики

– задача другого порядка трудности.

В самом деле, проводимые в «промышленных» кругах поиски в области программирования, стимулируемые в самом начале глубокими «университетскими»

исследованиями, впоследствии часто шли другим путем. В методах Миллза, Бейкера или Джексона основной упор, безусловно, делается на лежащую в основе здоровую методологию для индивидуального программиста, но, кроме того, в большой степени на техническую организацию и оснащенность программистских бригад, участвующих в крупных проектах. Расхождение интересов этих двух различных направлений в той дисциплине, которую принято называть «структурным программированием», привело к тому, что представители их не всегда говорят на одном и том же языке. Последователи одного направления упомянули бы Декарта там, где приспешники другого обратились бы к Тейлору; первые стали бы мечтать о «правильности» там, где другие стали бы стремиться к рентабельности и соблюдению сроков; одни стали бы предлагать язык программирования, позволяющий определять структуру данных при помощи их «функциональной спецификации» (гл. V), в то время как другие стали бы изучать «препроцессор», позволяющий использовать сложные операторы в КОБОЛе или ФОРТРАНе. Одни стали бы исследовать, как разбить сложную задачу таким образом, чтобы разобраться в сущности, и стали бы опираться на пример гауссовской задачи о «восьми ферзях» (упражнение VI.10) – там, где другие стали бы пытаться определить разделение труда и иерархию ответственности в группе программистов.

Разумеется, это расхождение – всего лишь один аспект более общей проблемы: той пропасти, которая разделяет в информатике промышленную практику и научные исследования. Можно сожалеть о таком положении вещей, но нельзя его отрицать, даже если вы считаете, подобно авторам этой книги, что расхождение по многим пунктам имеет несущественный характер, и сближение двух типов проблем будет только плодотворным.

VIII.3. Уровни программирования

Программирование представляет собой **сложный интеллектуальный процесс.** Таким образом, чтобы попытаться его понять, необходимо принять методы разложения на различные уровни. На деле возможно много разных разбиений в соответствии с тем, какой аспект проблем программирования желают осветить. Мы посвятим этот раздел различным подходам, но прежде всего необходимо напомнить основной принцип, который очень важно не потерять из виду: принцип *глубокого единства* процесса программирования.

VIII.3.1. Единство программирования

Первым естественным следствием современных взглядов на программирование является пересмотр обычного разделения работ: спецификация, постановка задачи, функциональное исследование, анализ схемы задачи, кодирование. Жесткое разделение этих уровней приведет лишь к тому, что к работам по постановке задачи и программированию добавится работа по связи, «интерфейсу», тем более трудная, что человеческий фактор в ней занимает важное место.

Полагать, что можно решить задачу, доверив вначале анализ небольшой группе высококвалифицированных специалистов, а затем кодирование – армии программистов, работающих над спецификациями, полученными на предыдущем этапе, это значит получить неоднородные и малонадежные программы, о которых ни у кого нет полного представления. Применение к программированию рабочей техники, скопированной с военной структуры или с организации автомобильных сборочных конвейеров, привело бы к «раздроблению» работ, что из–за требований высокой степени надежности имело бы катастрофические последствия для качества конечного продукта.

Примем следующее соглашение: мы ни в коем случае не отрицаем

необходимости иерархического разбиения задач, которое является единственным путем к уменьшению сложности решаемых проблем. В гл. V мы описали методологию спецификации структур данных, основанную на использовании трех уровней («функционального», «логического» и «физического»); ниже встанет вопрос об общем подходе к программированию, основанном на аналогичном подходе, так называемом языке Z. Однако важно не терять из виду глубокое единство процесса программирования. На всех этапах решения задачи от первоначальной формулировки до кодирования в процесс включаются одни и те же мыслительные механизмы. Этих механизмов немного, и все они направлены на разрешение одной фундаментальной проблемы: выделить трудности, чтобы уменьшить уровень сложности задач, которые надо решить. Все основные приемы программирования, которые мы встречали в предыдущих главах, являются лишь различными, но имеющими общую цель способами разбиения задач, программ и данных: рекурсия (в применении к программам и данным), управляющие структуры, структуры данных, подпрограммы. Понимание фундаментального единства процесса программирования (повторяем, что последний термин берется в самом широком смысле слова) является главным условием получения надежных и однородных конечных продуктов. Проблемы, встречающиеся на различных этапах, имеют отношение к одной и той же методологии; отличием является лишь степень подробности, с которой они рассматриваются, точнее, требуемый уровень абстракции.

VIII.3.2. Уровень абстракции. Виртуальные машины

VIII.3.2.1. Введение и определения

Ничто лучше не определило бы понятия уровня абстракции, чем цитата, взятая в качестве эпиграфа к этой главе; перенося ее область применения из математики в информатику (но это ведь всего лишь изменение освещения?), можно будет извлечь из нее чрезвычайно красноречивое объяснение основной задачи программиста: пренебрегать на каждом этапе всеми не относящимися к делу деталями, *абстрагироваться* от всего, что может быть отложено «на потом», чтобы постоянно держать под контролем исследуемые задачи.

Всякое систематическое решение задачи, какой бы простой она ни была, требует действия, основанного на принципе разбиения на уровни абстракции, на «логические слои». Интересная интерпретация состоит в рассмотрении этого метода как создания на каждом этапе программы для виртуальной машины [Пратт 76] или для абстрактной машины [Шербонно 75] требуемого уровня абстракции. Самый высокий уровень–это уровень гипотетической машины, способной непосредственно решать исходную задачу. Самый низкий уровень–это уровень реальной машины, но на деле программист останавливается на уровне абстракции, представленном языком программирования, которым он располагает. Все происходит так, как если бы он имел в своем распоряжении «машину АЛГОЛ», «машину ФОРТРАН», «машину ЛИСП» или «машину АПЛ» и т.д., исходный набор команд которых определяется правилами АЛГОЛа, ФОРТРАНа, ЛИСПа или АПЛ и т.д. (Эта смесь понятий из области технического и программного обеспечения все более оправдывается развитием техники: «машина ФОРТРАН» может быть на практике представлена транслятором на ФОРТРАНе на классической ЭВМ, микропрограммой на ФОРТРАНе на микро–программируемой вычислительной машине или же действительно вычислительной машиной, аппаратно приспособленной для выполнения операторов ФОРТРАНа. Точно так же систематическое использование стандартных схем, сочетание которых все более напоминает работу программиста, позволяет получить модули желаемых характеристик, помогает стереть грань между традиционными понятиями программного обеспечения и аппаратуры.)

Промежуточные элементы, находящиеся между двумя крайними уровнями абстракции, должны хорошо стыковаться, чтобы дать пригодный для работы конечный продукт. Таким образом, каждый элемент программш и данных, записанный на уровне *n + 1* для виртуальной машины уровня и, является в основном практической реализацией (на жаргоне – «внедрением») части машины уровня *n + 1* на машине уровня *n*. Заметим, что понятие виртуальной машины точно соответствует понятию подпрограммы; однако есть важная разница в нюансах, так как виртуальная машина соответствует логическое слою не только в смысле программ, но и в смысле соответствующих данных. В дальнейшем мы уточним эту идею при изучении расширенного понятия подпрограммы – понятия *модуля.*

VIII.3.2.2. Пример: программы сортировки

Простым примером разбиения на уровни абстракции является алгоритм «Быстрой сортировки» (VII.3.6). В этой программе четыре внешних уровня абстракции проявляются очень явно. Самый высокий уровень –это уровень виртуальной машины, которая должна уметь использовать любую программу «сортировки». Структура данных, с которыми она оперирует, представляет собой массив (элементов типа ЭЛЕМЕНТ, характеризуемых ключом КЛЮЧ); она обладает одним–единственным оператором, который можно обозначить как

Сортировка (a)

и действие которого состоит в сортировке массива a, что можно выразить с помощью комментариев или утверждений, определяющих свойства машины, подобно тому, как аксиомы Хоара (III.4) определяют свойства управляющих структур:

```
{a есть массив типа ЭЛЕМЕНТ с границами i и j}
        Сортировка (a)
{a содержит те же элементы, что и в предыдущем случае;
        при i ≤ k ≤ℓ ≤ j,
ключ (k) ≤ ключ (ℓ)}
```

Напомним, что обозначение {P} A {Q} означает, что, если свойство P истинно для объектов одной программы и выполняется действие A, свойство Q будет истинно после этого выполнения.

Мы ввели обозначение Сортировка, а не Быстрая Сортировка для виртуальной машины самого высокого уровня, так как пользователь должен ее рассматривать как сортировку вообще, *абстрагируясь* от частных методов.

Можно было бы взять любой из методов, изученных в гл. VII. Мы выбираем здесь Быструю Сортировку. Таким образом, осуществление сортировки будет связано с обращением к рекурсивной машине, которую можно назвать «сортировщиком»; структура данных, над которыми оперирует сортировщик, представляет собой под массив. Вспомнив, что Быстрая Сортировка (a[i : j]) (в самой примитивной форме) записывается как

```
если j > i то
    | s ← Деление (a[i : j]);
    | Быстрая сортировка (a[i : s – 1);
    | Быстрая сортировка (a[s + 1 : j])
```

видим, что сортировщик использует в качестве основных операций сравнение целых, присваивание целого и операцию, называемую Деление, которая манипулирует также и с подмассивами.

Эта операция представляется третьей виртуальной машиной–«разделителем», характеризуемой следующим образом:

{a[i:j] является подмассивом типа ЭЛЕМЕНТ}

s ← Деление(a[i : j])
{a[i:j] содержит те же элементы, что в предыдущем
случае; $i \le s \le j$;
при $i \le k \le s$ ключ (к) ≤ ключ (s);
при $s \le \ell \le j$ ключ (ℓ) ≥ ключ (s)}

Разделитель использует в свою очередь четвертую виртуальную машину, основные операции которой таковы:

поменять элементы с индексами i и j

и

сравнить ключи элементов с индексами i и j

Эти операции сами могут быть закодированы на каком–нибудь языке программирования, который будет представлять пятый уровень абстракции.

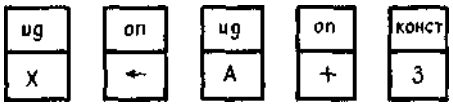
Важным моментом во всем этом разложении является то, что можно совершенно **отличными** друг от друга способами исследовать задачи, поставленные различными виртуальными машинами: управление рекурсией и представление стека для сортировщика; усовершенствование внутренних циклов для разделителя. Безусловно, эти$_{\text{ч}}$задачи не являются полностью независимыми, но отношения между машинами различных уровней должны быть *полностью определены* внешними свойствами, или *спецификациями,* которые здесь представлены в виде комментариев в фигурных скобках. Эпитет **«внешний»** является принципиальным: должна существовать возможность определить абстрактную машину описанием результата, когда ее применяют к известным данным независимо от того, каким образом получен этот результат; другими словами, уточняется, что делается, но не как это делается. Понятие спецификации было развито в гл. V (функциональная спецификация структур данных); оно будет исследовано ниже (VIII.3.5).

VIII.3.2.3. Второй пример: транслятор

Проблема трансляции разбивается на несколько уровней абстракции, соответствующих абстрактным машинам, детальное описание которых можно найти в любой специализированной монографии (например, [Грис 71] или [Ахо 77]), но для наших целей полезно ее вкратце напомнить. В противоположность предыдущему описанию будем исходить из самого низшего уровня (восходящий метод, а не нисходящий; ср. ниже, VIII.3.3):

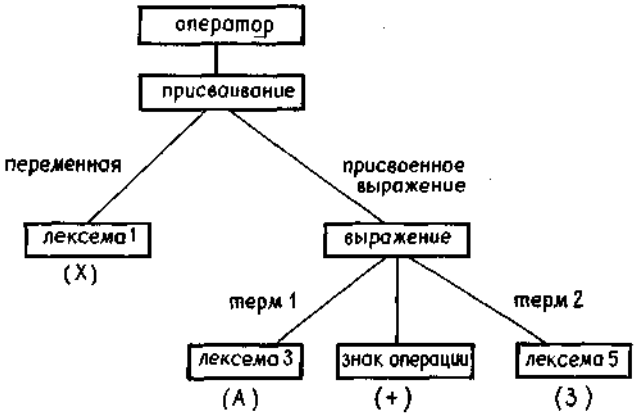
- **читающее устройство** должно «переварить» последовательные литеры, чтобы выдать их на высшие уровни. Важным здесь является то, что читающее устройство «уладит» все детали представления (переход от одной карты к другой, карты–продолжения в ФОРТРАНе, устранение пробелов, если они не являются значащими, и т.д.); таким образом, последующие машины видят программу как поток литер, при этом абстрагируются от проблем носителя: нет больше карт или строк за пределами читающего устройства;

- задачей **лексического анализатора** является группировка последовательных литер, чтобы выдать основные элементы на высший уровень в виде символа. Эти элементы, называемые *лексемами,* – идентификаторы, операторы, константы, зарезервированные слова и т.д.; каждый из них является парой [тип, значение]; например,



На последующих уровнях программа будет представлена после окончания работы как последовательность лексем: после окончания работы лексического анализатора «литер» больше не существует;

- **синтаксический анализатор** группирует лексемы в структуры, представляющие «логический анализ» фразы языка (выражение, оператор и т.д.). Следующую структуру можно назвать *синтагмой:*



- **программа, управляющая таблицей символов,** хранит сведения о всех переменных (и других именах, появляющихся в программе), их типах, декларациях, использованиях их областей действия, если имеется блочная структура, и т.д.

Следующие уровни могут быть различными в разных трансляторах. Можно предположить, например, что существует

- **программа, переводящая в постфиксную польскую запись,** которая приводит предыдущие «синтагмы» к линейному бесскобочному виду (соответствующему обходу ЛПК двоичного дерева); например,

  X ← A + B * C

  будет переведено как

  XABC* + ←

- **генератор объектного кода**, который создает программу в машинном языке, исходя из предыдущего вида.

Если отвлечься от технических деталей, то важной здесь является абстракция, осуществляемая каждой машиной: «переводчик в постфиксную польскую запись» выдает польскую запись на высшие уровни, синтаксический анализатор «маскирует» лексемы, выдавая лишь синтагмы, и т.д. Подобную декомпозицию можно наблюдать среди объектов, которыми манипулирует операционная система (ср. [Дейкстра 68в]).

VIII.3.2.4. Третий пример: бухгалтерия

Чтобы закончить эту иллюстрацию понятием абстракции, возьмем пример из тематики АСУ. Пусть надо регистрировать записи счетов банка, страховой компании или промышленного предприятия. Эти счета имеют очень различное происхождение: закупка материала, продажа товаров, выплата налогов или процентов, образование задела или запасов и т.д. Напротив, конечный продукт процесса бухгалтерского учета

соответствует общей очень точной модели: запись в некотором количестве условных состояний дебета и кредита, которые друг друга «балансируют» (двойная бухгалтерия). Это можно описать, например, таким списком, состоящим из триплетов [счет, направление, значение]:

- каждый «счет» – это номер, записанный в сводном списке счетов предприятия;
- каждое «направление» – это либо «дебет», либо «кредит»;
- каждое «значение» – это сумма во франках и сантимах;
- сумма значений списка, соответствующих направлению «кредит», равна сумме значений, соответствующих направлению «дебет».

Тогда удобно разложить обработку бухгалтерских записей на несколько «элементарных» программ, таких, как

- **«управление»,** которое «собирает» данные, поставляемые соответствующими службами предприятия, и «каталогизирует» их в соответствии с перечнем заранее установленных бухгалтерских операций (закупка, продажа, налог, зарплата, взносы и т.д.);

- **«расчет»,** который преобразует представленные «управлением» данные в списки вида

[[41, кредит, 13048,25], [562, дебет, 13000], [66, дебет, 48,25]]

не учитывая их происхождения; на этом уровне можно поместить проверку формальной правильности записей (существование в бухгалтерском плане трех счетов и общий баланс записей);

- **«регистрация»,** которую можно подразделить на несколько частей: одна присваивает номер записи, другая записывает его в гроссбух, третья–в журнал, четвертая заносит на баланс, в то время как пятая производит, например, если надо, проверку некоторых банковских операций.

VIII.3.3. Нисходящая и восходящая концепции

VIII.3.3.1. Определения

Приняв принцип декомпозиции задачи на уровни абстракции, программист сталкивается с важным вопросом: в каком порядке подступать к этим уровням? Естественным образом представляются два метода: можно исходить из наиболее абстрактной виртуальной машины, машины наиболее высокого уровня, чтобы понемногу «спуститься» посредством последовательных декомпозиций к реальной машине (представленной языком программирования), либо, напротив, заложить сначала самые нижние кирпичи здания, составляя постепенно программу, опирающуюся на каждом этапе на эффективно реализованные виртуальные машины, а затем «восходить» к общей задаче, которую надо решить. Первый подход повсеместно называется *нисходящим,* второй–*восходящим.* И тот и другой достойны изучения.

Принцип **нисходящего программирования** был изложен Виртом [Вирт 71] под названием разработки программ **«последовательными уточнениями»** (successive refinements). Он состоит в пошаговых действиях, исходя из самого высокого уровня абстракции, который является уровнем формулировки задачи, например

сортировать массив a

На каждом шаге «уточняют» различные элементы полученной на предыдущем шаге программы, выражая их в терминах элементарных операций более низкого уровня абстракции. Постепенно таким образом приближаются к конечному уровню, уровню языка программирования, используемого на этой машине.

> Важный момент, подчеркнутый Виртом, состоит в том, что этот процесс последовательного уточнения программ должен происходить одновременно с таким же процессом уточнения данных; другими словами, каждый шаг ведет к описанию на логическом или физическом уровне структур, использованных на предыдущем шаге посредством их функциональной спецификации. Это описание может обращаться к новым функционально характеризуемым структурам данных, которые будут уточнены на следующем уровне. При уточнении данных, а также и программ важно на каждом шаге выражать в явном виде лишь строго необходимые детали для понимания этого уровня, чтобы сохранить полное понимание каждого элемента.

Нисходящее программирование можно понимать как построение, начиная с корня, «дерева программы», листья которого соответствуют конструкциям языка программирования и в котором каждая вершина представляет собой программу и данные, полученные комбинацией с помощью управляющих структур и методов структурирования данных, программ и данных, соответствующих своим сыновьям ([Джексон 75] предлагает образное представление этих способов комбинирования: * для цикла, о для альтернативы или «разделения вариантов», смежные вершины для цепочки операторов или подстановки данных). На Рис. VIII.2 приведена схема представления в виде дерева программы Древесная Сортировка (VII.3.7).

Обратный метод – **восходящий** – используется, возможно, более естественным образом, когда еще не особенно изучена стратегия решения задачи. Он состоит в «подстраховке», как говорят спортсмены, путем постепенного построения все более сложных программ, начиная с самого низкого уровня и поднимаясь к решаемой задаче; на каждом этапе следует объединять предварительно построенные элементы, чтобы перейти к более высокому уровню абстракции. Таким образом, здесь, прежде чем приступить к задаче, следует построить средства, позволяющие ее решить, в то время как в нисходящем методе первоначальная задача разбивается на некоторое число подзадач, при этом определяется необходимый аппарат, который затем пытаются реализовать. Восходящий метод убедительным образом изложен (при этом не называется по имени) в описании реализации операционной системы THE [Дейкстра 68].

Рис. VIII.2 Древовидная структура Древесной сортировки

Прежде чем обсуждать сравнительные достоинства обоих методов, следует обратить внимание на тот факт, что в нашем изложении их описания представляют собой крайние варианты; на практике их различие выглядит более умеренным. Нисходящий метод в чистом виде невозможен: декомпозиция задачи не может быть начата, если нет представления об окончательной форме, которую, например, должна принять программа, на языке программирования, даже если это представление остается нечетким и намеренно неявным. Напротив, в восходящем методе выбор элементов, которые надо построить, и их объединение проводятся в соответствии с некоторыми априорными предположениями об общей структуре задачи. Таким образом, как

восходящий, так и нисходящий методы представляют собой в большей мере идеализированные общие схемы, нежели некие абсолютные и несовместимые категории.

VIII.3.3.2. Сложности, возникающие при использовании нисходящего метода

Вне всякого сомнения, нисходящий метод является в принципе наиболее рациональным выбором; в действительности наиболее логичным методом решения задачи является ее многократное разбиение на более простые подзадачи до тех пор, пока не будет достигнут тот уровень, на котором решение получается непосредственно (полезно сравнить со стратегиями автоматического решения задач, упомянутыми в VI.5, в частности с «деревьями и/или»).

Однако нисходящий метод ставит некоторые проблемы:

a) **начальный выбор,** выполняемый на самом высоком уровне, становится решающим; если он делается преждевременно, то бывает трудно к нему возвращаться, не переписав заново всю программу. Это могло бы произойти, например, в случае неудачного выбора структуры данных: структура, наличие которой подразумевайся на некотором уровне, может оказаться на более низких уровнях невозможной или слишком дорогостоящей для физической реализации. Это также могло бы произойти в случае производимого слишком рано арбитража между решениями типа «компиляции» (ввести все данные, преобразовать их к внутреннему виду и затем обрабатывать в этом внутреннем виде с самого начала) и «интерпретации» (обработка данных по мере их считывания); понятия «компиляции» и «интерпретации» в применении к обработке произвольных данных изучались в гл. VII (VII.1.2.3 и VII.4).

Это кажущееся неудобство нисходящего метода может иметь в конечном счете благоприятный эффект* если программирование производится чрезвычайно строго и внимательно; в самом деле, этот метод обязывает с самого начала ставить действительно фундаментальные проблемы, освобождая их от менее важных вопросов, решаемых на более поздних шагах. Реализацией этой идеи является «статическое программирование» (VIII.3.5 ниже).

b) Может оказаться очень стеснительным, если связи между элементами сложной программы ограничиваются только отношениями, которые могут быть представлены **деревом.** Основная задача состоит в том, чтобы поддеревья некоторого дерева не пересекались; однако некоторый элемент программы или одинаковые структуры данных могут оказаться необходимыми на различных этапах решения. Эта проблема возникает, например, при нисходящем описании Древесной Сортировки (Рис. VIII.2**),** где поддерево, соответствующее программе Реорганизация, вводится в двух местах, при уточнении Посадки и Сортировки максимизирующего дерева. Кроме того, можно отметить, что в гл. VII Древесная Сортировка не была представлена чисто нисходящим образом по той причине, что основой алгоритма является виртуальная машина «Реорганизация» сравнительно низкого уровня.

Наиболее серьезной проблемой является, однако, не то, что некоторые одинаковые элементы встречаются в нескольких местах дерева программы; в этом случае можно считать, что древовидная структура правильна с концептуальной точки зрения, но при физической реализации нет никаких оснований не допускать, чтобы несколько модулей делили между собой один элемент программы или данных. Более серьезен частый случай, когда приходится в различных точках дерева определять похожие, но не одинаковые элементы, например программы сортировки или структуры, позволяющие управлять массивами быстрого доступа, с несколько

различающимися внешними спецификациями. В этом случае нисходящее программирование, которое приводит к построению инструментария по мере того, как он становится необходимым для конкретных потребностей, по–видимому, приходит в противоречие с традиционным и здоровым принципом программирования, в соответствии с которым надо при написании программ стремиться к общности, чтобы их можно было легко приспособить к малейшим изменениям спецификаций.

Ответ на этот вопрос, очевидно, состоит в том, что к нисходящему подходу надо прибавить восходящую компоненту и на практике на каждом этапе следить за развитием элементов, поддающихся обобщению. Когда такой элемент обнаружен, следует снова обратиться к нему, описав его так, чтобы превратить в удобное и легко применимое орудие. Обнаружение таких поддающихся обобщению элементов, является, однако, трудной задачей, поскольку, если и следует избегать повторения похожих работ, не менее важно сохранить независимость различных составляющих программы, ограничивая, насколько возможно, число соотношений между ними, т.е. объем «интерфейса».

c) ретьим тонким моментом в использовании нисходящего метода является вопрос тестов: можно ли подвергать систематическим испытаниям каждый элемент программы? На первый взгляд это кажется сложным, поскольку каждый элемент разработан в предположении полной абстракции от других элементов, которые он использует, и так, что соединение этих элементов откладывается на более поздний этап. В решении, предложенном Миллзом [ACM 73], используется тот факт, что если эти элементы нижнего уровня и не описаны, то все же функционально специфицированы; их можно, таким образом, заменить, чтобы провести тестирование элемента, в котором они используются, при помощи **заглушек** (*"stubs"* в терминологии Миллза), выполняющих простые процедуры, соответствующие, по крайней мере частично, функциональной спецификации, но без реальной связи с решаемой задачей (чтобы избежать необходимости полностью развертывать поддеревья вершины, которую надо тестировать). Обычно элемент P спользующий среди прочего целую, достаточно сложно вычисляемую функцию f(x, y, ..., z), которая выдает значение, заключенное между –1000 и +1000, мог бы быть тестирован путем замены f на «заглушку», единственной задачей которой является выдача псевдослучайного целого значения, заключенного между этими двумя границами. В Быстрой Сортировке заглушка, представляющая собой Деление, построила бы массив, состоящий из данных, расположенных вокруг некоторого элемента.

Однако следует отметить, что построение заглушки может оказаться нетривиальной задачей, в частности, на самых нижних уровнях дерева, где заглушку может быть так же сложно построить, как настоящий элемент программы, который она заменяет; вспомните, например, о заглушке, представляющей собой Посадку или Реорганизацию в Древесной Сортировке. Кроме того, особенно трудно гарантировать, чтобы были протестированы все логические возможности, но это общая проблема тестов (VIII.4.2.5).

VIII.3.3.3. Проблемы, возникающие при использовании восходящего метода

Основное неудобство восходящего метода, которое является, напротив, преимуществом нисходящего метода, связано с проблемой **интеграции.** Если несколько элементов одного и того же уровня написаны по отдельности и их пытаются объединить, чтобы составить непосредственно следующий за ним более высокий уровень, возникает деликатная проблема *интерфейсов,* т.е. способов связи: списки параметров, общие данные, общие программы. Однако опыт доказывает, что в восходящем программировании почти систематически приходится делать слишком

сильные и не обязательно сознательные предположения об условиях, в которых описанные элементы будут использоваться. Тогда в момент интеграции нескольких элементов возникают порой серьезные конфликты между противоречивыми предположениями, ни одно из которых, возможно, не соответствует действительности. Проблема становится чрезвычайно острой, если проект выполняет группа сотрудников, но **она** возникает уже и в том случае, когда работает всего один человек.

Таким образом, **интеграция** является бичом восходящего программирования; эта проблема объясняет, почему при использовании этого метода первые этапы и их тестирование могут быть довольно быстрыми (не возникает вопроса заглушек), тогда как последующие фазы занимают часто непредсказуемо много времени, которое в основном посвящается решению болезненных проблем связи. Кроме того, эти проблемы могут проявиться с опозданием, т.е. в момент интеграции элементов более высокого уровня, чем те, с которыми в действительности имеют дело и предварительное тестирование которых, казалось, указывало на их правильное функционирование. Таким образом, кажется, что неправильное использование восходящей интеграции частично является причиной ненормальных, упомянутых в начале главы соотношений между временем, предназначенным для «творческого» программирования, и временем, которое требуется для поиска и исправления ошибок.

Тем не менее восходящий метод обладает определенным числом достоинств; в частности, это относится к реализации средств, которые необходимы каждому программисту, как любому другому хорошему ремесленнику, для любого дела. Но упомянутые трудности показывают, что восходящий метод не приспособлен для построения наиболее высоких уровней абстракции большой программы.

VIII.3.3.4. Заключение

Какой метод следует принять? Было бы абсурдом давать на этот вопрос категорический и окончательный ответ. Однако в свете ранее сказанного можно рекомендовать с самого начала использовать нисходящий метод, применяя его с особой осторожностью на самых высоких уровнях абстракции, построение которых практически необратимо обусловливает общее осуществление проекта; однако в ходе разработки программы нужно быть готовым «изменить скорость», перейти к восходящему методу, чтобы можно было справиться со следующими ситуациями:

- виртуальная машина, существование которой являлось неявным на предыдущем уровне, становится невозможной или слишком трудно осуществимой; тогда следует совершить возврат назад и изменить предыдущий уровень;
- замечают, что несколько ветвей дерева требует осуществления виртуальной машины с похожими спецификациями. Тогда принимаются за реализацию элемента, достаточно общего, чтобы он мог отвечать требуемым спецификациям (и, возможно, другим из того же класса). Этот элемент будет рассматриваться как *инструмент.* Отметим, что его осуществление может быть само по себе нетривиальной задачей и, таким образом, вызывать новые проблемы разбиения на уровни;
- среди требующихся, но еще не реализованных виртуальных машин различают элемент, чьи спецификации близки к спецификациям существующего инструмента. Чтобы суметь извлечь пользу из этого инструмента, возможно, придется его модифицировать, чтобы сделать более общим, или приспособить спецификации, необходимые на более высоких уровнях (либо сделать и то и другое). Заметьте, что в конце концов всегда приходится сталкиваться с этой ситуацией, потому что нисходящий метод должен давать продукт, совместимый с существующими средствами, как в плане программного обеспечения (язык программирования, операционная

система), так и в плане аппаратуры (средства, предоставляемые машиной), даже если эти средства не отвечают идеальным спецификациям (как это чаще всего бывает).

Из предыдущего обсуждения вынесем, кроме того, два особенно важных момента:

a) создании большой программы самым трудным является не написание отдельных элементов–«кирпичей», из которых состоит здание, а определение их отношений и развитие стратегии разработки. Следовательно, всегда будет плодотворным для дальнейшего увеличить время, посвященное разработке ансамбля, размышлениям, предваряющим кодирование;

b) если элементы программы ставят сами по себе мало проблем, то, напротив, решающим является тот факт, что роль каждого из них всегда полностью определена. Другими словами, как в нисходящем, так и в восходящем методах важно концентрировать свое внимание на интерфейсах между элементами программы, и в частности важно всегда явно определять сами элементы, прежде чем их детализировать.

Чтобы специфицировать интерфейс между элементами программы и окружением, можно использовать обозначение, основанное на том, которое было предложено в гл. IV для подпрограмм, и имеющее такой вид:

программа xxx : T
(**аргументых**, y, z : T_1,
u, v, w : T_2, ...;
результаты h, i, j : T_3,
k, 1, m : T_4, ...,;
модифицируемые данные a, b, c : T_s, ...)
{...внешнее описание действия xxx на свои параметры...}

«Внешнее описание» может иметь вид {P}xxx{Q} аксиом Хоара (III.4), позволяющий рассматривать вызов подпрограмм в некоторой программе как оператор; это хорошо соответствует концепции, согласно которой подпрограмма является абстракцией, представляющей сложный оператор. Можно также принять «функциональное» обозначение, похожее на обозначение гл. V.

VIII.3.4. Понятие модуля

Итак, чтобы построить программу, необходима стратегия декомпозиции; примеры этого только что были показаны. Однако остается еще уточнить такой вопрос: каким должен быть основной элемент в построении программы, единица декомпозиции, получаемая на каждом шаге при восходящем, нисходящем или смешанном методе?

Наиболее простой и наиболее классический ответ состоит в том, чтобы рассматривать просто элемент программы, определенный своими входами, выходами и, возможно, своими «модифицируемыми данными», т.е., как мы только что видели, подпрограмму–по крайней мере концептуально; при этом практическая реализация, разумеется, может быть совсем иной.

Это решение, однако, не определяет сути. дела. Действительно, большая организационная трудность программного проекта чаще происходит не из–за отношений между элементами *программы,* а из–за управления и передачи данных. И даже если строго определять способы связи данных между программными модулями, как было показано выше, то остается сложная проблема распределения ответственности: какой программный модуль должен обеспечивать основное управление набором данных, используемых многими элементами?

Осознание того факта, что эта проблема в конечном счете самая трудная,

приводит к стратегии декомпозиции, основанной на понятии «модуль данных» в большей степени, чем на понятии «программный модуль».

Такой основной элемент мы будем называть модулем; близкими терминами являются «класс» в СИМУЛе 67, «форма» в ALPHARD [Вулф 75], «кластер» в CLU [Лисков 77]; см. также LIS [Ишбиа 75] и т.д. Отметим, однако, что мы отклоняемся от смысла, часто приписываемого слову «модуль», а именно единица программы со строго определенными интерфейсами. Сторонники «модульного программирования», ограниченного этим узким смыслом, настаивают на необходимости логического разбиения на мелкие единицы с хорошо определенными отношениями. Надеемся, что для читателей этот вопрос очевиден, хотя он и не позволяет узнать, *как* производить такое разбиение.

С нашей точки зрения модульное программирование будет в основном методом декомпозиции, основной элемент которой, или модуль, есть *структура данных* (гл. V), доступная извне при помощи некоторого набора подпрограмм, и только при помощи их.

Вернемся к примеру транслятора. Самый простой подход состоит в том, чтобы рассматривать различные элементы программы на различных уровнях абстракции: «транслятор», «синтаксический анализатор», «лексический анализатор», программу управления таблицей символов, «переводчик в польскую постфиксную запись», «генератор объектного кода», программу читающую символы, и т.д.

> Однако такое разбиение ставит серьезные проблемы в смысле доступа к данным. Например, синтаксический анализатор повторно обращается к лексическому анализатору, чтобы узнать последовательные элементы оператора. Он должен уметь расшифровывать эти элементы, т.е. знать их тип, их возможное положение в таблице символов и т.д. Таким образом, лексический и синтаксический анализаторы обобществляют довольно сложную структуру данных, такую, которая позволяет описывать лексические элементы, или «лексемы», и их различные свойства. Трудность общения между этими двумя элементами программы вытекает не из точного определения их задач (это определение сравнительно тривиально), а из ограничения их прав доступа к общей структуре данных; речь идет об уточнении того, к какой составляющей лексемы может каждый из них *обращаться,* какую составляющую он может *модифицировать,* может ли он создать новую лексему и т.д.

При модульном программировании эти проблемы «атакуются в лоб»; за единицу разбиения берут в точности основные структуры данных, а не программы, которые ими манипулируют:

- **литера,** базовая единица процесса чтения (а не программа чтения);
- **лексема,** базовая единица процесса лексического анализа (а не лексический анализатор);
- **синтагма** (а не синтаксический анализатор);
- **символ** и **таблица символов;**
- **выражение в польской постфиксной записи;**
- **оператор на машинном языке;**
- **исходная программа;**
- **объектная программа.**

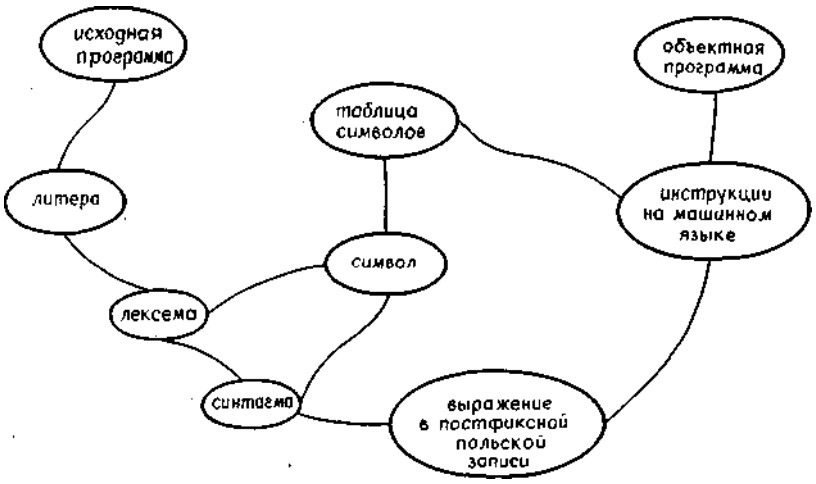


Рис. VIII.3 Модули компилятора.

Каждый из этих элементов находится в центре модуля (Рис. VIII.3), включающего, кроме того, все *программы доступа*. Эти программы доступа являются единственными средствами, позволяющими манипулировать рассматриваемыми элементами. Например, модуль «лексема» соответствует в терминологии гл. V определению нового типа ЛЕКСЕМА и функциям:

строкалекс: ЛЕКСЕМА → СТРОКА {поскольку ℓ – лексема, строкалекс (ℓ) является строкой, составленной из цепочки литер, составляющих ℓ}

типлекс: ЛЕКСЕМА → ("целая константа", "идентификатор", "оператор", "строковая константа...")
{поскольку ℓ – лексема, типлекс (ℓ) может равняться "целой константе", "идентификатору" и т.д. в зависимости от того, что собой представляет ℓ}

аналекс: → ЛЕКСЕМА
{начиная с "входа", т.е. с литер, составляющих программу, аналекс анализирует литеры, составляющие лексему, и выдает эту лексему как результат}

Естественно, что «лексический анализатор» вновь появляется здесь в форме аналекс; но это всего лишь программа, составляющая модуль лексема. Гипотезой модульного программирования является то, что программа строкалекс играет такую же важную роль в построении транслятора, что и аналекс, даже если написание строкалекс сравнительно тривиально, поскольку обычно речь будет идти о засылке значения элемента массива, поля «записи» (АЛГОЛ W, АЛГОЛ 68, ПАСКАЛЬ, СИМУЛА 67 и т.д.) или составной части «структуры» (КОБОЛ, ПЛ/1).

Все модули организованы как объединение программ доступа, которые могут общаться с другими модулями при помощи программ доступа к этим модулям. Например, модуль синтагма, без сомнения, будет включать в себя программу

аналопер: → СИНТАГМА

в задачу которой входит вырабатывать, исходя из данных, синтагму, представляющую оператор. Эта программа могла бы состоять из элементов такого рода:

```
переменные ℓ1 ℓ2; ЛЕКСЕМЫ;
ℓ1 ← аналекс; {читать лексему благодаря модулю ЛЕКСЕМА}
если типлекс(ℓ1) = "идентификатор" то
        ℓ2 ← аналекс
        если типлекс (ℓ2) = "оператор" и строкалекс(ℓ2) ="←" то
                {анализируемый оператор является присваиванием: продолжить
                декодирование}
                ...
```

Цель этого примера–наброска – показать, что модули должны общаться между собой только путем использования соответствующих программ, за исключением прямого доступа к самим данным (это ограничение является непременным условием применимости метода).

Этот метод имеет множество важных преимуществ:

a) каждый модуль нечувствителен к внутренним изменениям других модулей: общаясь с ними только посредством программ доступа, он не имеет никаких сведений о возможных последовательных и противоречивых выборах представлений данных, лишь бы при этом происходили соответствующие изменения в программах доступа;

b) столь же важным моментом в «модульном» проектировании большой программы является то, что неизбежным ограниченным изменениям спецификаций должны соответствовать ограниченные модификации модулей. Это условие здесь выполнено, поскольку небольшое изменение должно выразиться исключением, добавлением или модификацией единственной программы доступа либо малого числа таких программ.

По контрасту заметим, насколько эта проблема становится острой при декомпозиции, основанной на программах, а не на данных. Например, в случае транслятора такой элемент, как «синтаксический анализатор», мог бы при этом подходе общаться с «лексическим анализатором» способом, похожим на описанный выше, посредством обращения к подпрограмме

аналексик (tx, ty)

где tx и ty представляют собой параметры типа **результат**, позволяющие получать текст и тип анализируемой лексемы. Но предположим теперь, что происходит изменение спецификации, а именно, от лексического анализатора требуют выдать, кроме текста лексемы и его типа, целую величину, которая должна соответствовать: для целой константы–своему числовому значению; для знака операции–коду, позволяющему его обозначить (например, для " + ", число 1, для " –" число 2 и т.д.), для идентификатора – индексу в таблице символов и т.д. Поскольку у подпрограммы аналексик теперь имеется лишний параметр, все обращения к этой подпрограмме во всех использующих ее модулях должны быть модифицированы! Работа может оказаться значительной, и имеются многочисленные возможности для ошибок или упущений. При нашем «модульном» подходе, напротив, не надо ничего изменять из того, что уже существует: достаточно добавить к спецификации модуля ЛЕКСЕМА программу

значлекс: ЛЕКСЕМА –ЦЕЛОЕ

Программы, для которых действительно нужно знать значение, соответствующее лексеме ℓ могут иметь к нему доступ посредством значлекс(ℓ). Подобные изменения или усиления спецификаций очень распространены при построении большого программного проекта. Поэтому особенно важно, чтобы этим методом они предусматривались и принимались в расчет без особого ущерба.

Заметим, что одним из первых, кто ясно выразил основные идеи этого метода, был Парнас, который предложил [Парнас 72] для лучшего обеспечения защиты против недопустимых обращений давать модулям и функциям, имена без всякого мнемонического значения; таким образом, говорит он, пользователь модуля не будет иметь соблазна делать предположений a priori относительно внутренней организации данных модуля и будет полностью полагаться на абстрактную функциональную спецификацию. В нашем примере аналекс мог бы называться xx7tz02, текстлекс – uuvrs45 и т.д. Это нам не кажется серьезным: эта выдумка Парнаса предполагает, что программисты являются совершенными существами, обладающими непогрешимой логикой и безупречной способностью немедленного понимания любой абстрактной формулы. На деле достаточно использовать операционную систему, в которой фортрановский транслятор отзывается на нежное имя *IFEAAB,* чтобы, *наоборот,* оценить на всю жизнь достоинства простых и осмысленных имен.

c) третьим преимуществом модульного метода служит то, что он позволяет четко определить отношения между элементами программы, поскольку интерфейсы явно определены программами доступа. Таким образом,

возникает ясная картина ситуации, позволяющая знать, «кто кого вызывает». В частности, построение текстового окружения состоит в замене некоторых программ доступа «заглушками».

Однако предположенный нами метод имеет определенные неудобства:

a) повторное использование подпрограмм, часть которых ограничивается лишь считыванием или модификацией одного поля в структуре данных, влечет за собой потерю эффективности, происходящую из–за громоздкости вызова подпрограмм во многих современных системах. Конечно, можно было бы считать, что использование программ доступа происходит на уровне составления программ и что на уровне их практической реализации можно непосредственно манипулировать с желаемыми полями; но это было бы равноценно отказу от всех преимуществ метода при отладке текстов, диагностике выполнения и т.д.;

b) метод не решает проблемы выбора базовых модулей, для которых остается трудным делом сформулировать принципы, одновременно общие, правильные и непосредственно применимые. Модульный метод предлагает, однако, несколько указаний, которые могут служить руководством: строить модули вокруг структур данных, которые используются чаще всего и играют наиболее решающую роль (остается лишь их определить); пытаться минимизировать взаимодействия между модулями и сохранять функциональное единство каждого из них.

Заметим, что модульный подход ни в коей мере не стимулирует и не запрещает использование восходящего или нисходящего метода; и тот и другой могут в действительности определять стратегию разработки модулей. Тем не менее модульный подход позволяет видеть более широкие горизонты, где схема разработки была бы менее строго иерархизирована и более «самоуправляема», чем традиционные нисходящие и восходящие схемы. Однако разработка методов, позволяющих реализовать такую схему, сохраняя при этом строгость построения и избегая идейной беспорядочности, остается открытой проблемой.

VIII.3.5. Статическое программирование. Язык Z

VIII.3.5.1. Определения: язык Z_0

Ранее всюду мы настаивали на идее *функциональной спецификации.* В гл. V было показано, что в применении к данным это понятие позволяло определить структуру данных полностью внешним образом при помощи списка функций и их абстрактных свойств без обращения к способам представления их в памяти. Подобный способ определения был применен к программам, начиная с гл. III**,** и, в частности, к подпрограммам, начиная с гл. IV, когда мы в виде комментариев вставляли *утверждения,* которые позволяли сформулировать некоторые свойства, относящиеся к состоянию программы. Если попытаться сделать исчерпывающими эти утверждения, то программный модуль полностью характеризуется своим начальным и своим конечным утверждениями. При попытке развернуть эти утверждения и предельно их детализировать мы быстро заметим, что это трудная работа, *такая же трудная, как само программирование.*

В свете этого замечания определяющая идея, принадлежащая, в частности, Абриалю, состоит в том, что надо проделать дополнительный шаг и предположить, что разработка утверждений *входит в процесс программирования;* другими словами, программа может быть полностью описана последовательностью статических соотношений, просто создающих образ, находящийся на уровне абстракции, который отличен от уровня обычного описания и более фундаментален.

Язык Z [Абриаль 77], [Абриаль 77a] для создания и описания программ основан, таким образом, на той идее, что программа может и должна быть описана на нескольких уровнях. Первый из этих уровней, названный Z_0, допускает чисто статическое описание решаемой задачи; он полностью неарифметический, т.е. программы Z_0 не содержат операторов–команд, управляющих динамическим поведением реальной или виртуальной машины, – а имеют только список функций и их свойств. Таким образом, речь идет о функциональной спецификации не только структур данных, но и более широко – программ и задач[1].[2]

VIII.3.5.2. Уровень Z_0

Язык Z_0 полностью основан на языке общения и описаний, который на протяжении многих лет имел возможность проявить свои достоинства: это математический язык, и в частности язык теории множеств с небольшим количеством расширений, относящихся к задачам, встречающимся в программировании. Таким образом, основными элементами, с которыми манипулируют в языке Z_0, являются «объекты», множества и упорядоченные множества, или «кортежи». Основными операциями являются обычные теоретико–множественные операции (проверка принадлежности объекта множеству, проверка включения одного множества в другое, объединение двух множеств, пересечение, разность, прямое произведение и т.д.) и операции над кортежами, такие, как конкатенация, обозначенная знаком *; конкатенация кортежей $t_1 = [a, b, c]$ и $t_2 = [d, e, f, g]$ есть $t_1 * t_2 = [a, b, c, d, e, f, g]$; наконец, программа в языке Z_0 содержит определения функций и их свойств. Например, если A и B – два множества, то существование функции f из A в B обозначается следующим образом:

$$A \overset{f}{-} B$$

Преимущество этого обозначения (по сравнению с тем, которое мы использовали: $f: A \to B$) состоит в том, что оно позволяет наглядно уточнить некоторое число возможных свойств функций, важных для программиста. Например, могла бы существовать обратная функция $f^{-1} = g$; это обозначалось бы так:

$$A \underset{g}{\overset{f}{-}} B$$

f могла бы быть не всюду определенной функцией, т.е. не давать результата для всех переменных; здесь речь идет о важном понятии, которое мы обозначим, указав нижнюю грань 0 числа результатов f, как

$$A \underset{g}{\overset{f(0)}{-}} B$$

Вообще f могла бы быть, возможно, многозначной, т.е. ставить в соответствие любому элементу из A подмножество f(A) множества B (а не 0 или один элемент). Имена многозначных функций, так же как и имена множеств, начинаются с прописных букв; если F ставит в соответствие каждому элементу из A от 0 до 10 элементов из B, то это будет обозначаться так:

$$A \underset{G(1, -)}{\overset{F(0, 10)}{-}} B$$

(функция F здесь сюръективна). Надо уточнить здесь две границы, нижнюю и

[1] Приводимое здесь описание языка Z является кратким и частичным; мы несколько вольно обращаемся с обозначениями Абриаля, чтобы обеспечить их совместимость с обозначениями данной книги.

[2] Представленная здесь версия языка в дальнейшем сильно эволюционировала.
См. Abrial J. R., Schuman S.A., Meyer B.; Specification Language; dans Proceedings of Summer School on the Construction of Programs (Belfast, 1979); Cambridge University Press, Cambridge (Grande–Bretagne, 1980).

верхнюю. Кроме того, мы предположили, что обратная функция G является всюду определенной, но верхняя грань числа ее результатов неизвестна, что в обозначениях представлено с помощью тире. В случае когда частичная функция f или F не определена, условимся, что ее значение является специальным пустым элементом. Отношения между функциями являются логическими свойствами, выраженными при помощи обычных операций ∀ (для всякого), ∃ (существует), и, или, не и т.д.

VIII.3.5.3. Уровень Z_1 и его преобразования

Язык следующего уровня Z_1 позволяет «алгоритмизацию» программы Z_0 в том смысле, что в нем вводятся операторы. Действительно, Z_1 очень близок к использованной в этой книге алгоритмической нотации с той особенностью, что в нем широко используются теоретико–множественные выражения (**если** x ∈ A **то**...) и теоретико–множественные циклы типа (ср. с III.3.1.3)

```
для x ∈ A повторять
    | ...
```

или

```
для x ∈A пока c(x) повторять
    | ...
```

Совершенно особенный смысл этой характеристики языка Z_1 состоит в том, что он позволяет систематическим образом получать «алгоритмизованный» вид статических программ, выраженных на языке Z_0. [Абриаль 77a] дает таким образом целый ряд правил преобразования отношений Z_0 в операторы Z_1. Например, чтобы оценить следующее условие (логическое выражение) Z_0:

```
∀ x ∈ A.c(x)
```

в языке Z_1 записывают

```
e ← истина;
для x ∈A пока e повторять
   | e ← c(x)
```

Полученная таким образом программа на Z_1 не является окончательной: надо еще последовательными преобразованиями приблизиться к уровню, близкому к уровню обычных языков. Следующий языковый уровень, не получивший особого названия (можно было бы назвать его Z_2), полностью лишен каких бы то ни было теоретико–множественных ассоциаций: множества, представленные файлами, массивами, списками и т.д., просматриваются явным образом благодаря командам открыть (открытие «файла» и доступ к первому элементу) и читать (доступ к следующим элементам); множества исчерпаны тогда и только тогда, когда считанный элемент является специальным элементом пусто. Например,

```
для x ∈ A повторять
    | p(x)
```

записывается теперь как

```
x ← открыть(A);
пока x ≠ пусто повторять
    | p(x);
    | x ← читать (A)
```

Следующие преобразования позволят еще более уточнить реализацию программы: возможное удаление рекурсивности, выбор представления множеств более точными структурами данных (ср. с V.4), чтобы позволить эффективные доступы в соответствии с потребностями задачи, и т.д. Разумеется, целью является достижение в

конце концов какой–нибудь версии ФОРТРАНа, АЛГОЛа, ПЛ/1, КОБОЛа и т.д.

VIII.3.5.4. Пример

Чтобы суметь оценить метод Абриаля, удобно показать его на примере, хотя приводимый пример слишком прост. Речь идет об обработке файла документов, осуществляющей его «инвертирование»[1].

Пусть файл B состоит из «записей». В файл D надо включить «записи», выведенные из некоторых записей файла B в соответствии с «входными форматами», содержащимися в файле C. Файл A содержит список «ключей», позволяющих выбрать из B элементы, подлежащие обработке (Рис. VIII.4).

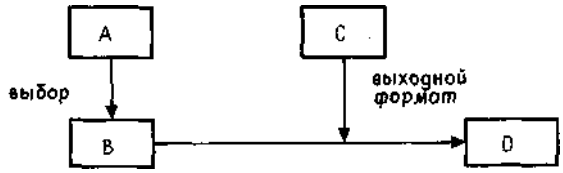


Рис. VIII.4

Элемент B есть «статья», состоящая из «ключа» и некоторого числа пар [имя поля, значение]; «имя поля» является кодом вида C10, C20 и т.д.; «значения» – это слова вида "насос", "турбина", "мотор" и т.д. Два различных элемента не могут иметь один и тот же ключ.

Если файл A пуст, будут обработаны все статьи B; в противном случае A содержит ключи, и будут обработаны лишь те статьи из B, чьи ключи содержатся в A.

При обработке статьи b, принадлежащей файлу B, в файл D будут занесены записи для каждой из пар [имя поля, значение], принадлежащих ; эти записи имеют вид [ключ, значение, номер подфайла], где «ключом» является ключ статьи b, «значением» – значение, найденное в обрабатываемой паре, а «номер подфайла» выводится из «имени поля» с помощью файла C, содержащего список пар [имя поля, номер подфайла].

Мы преднамеренно изложили задачу целиком, прежде чем перейти на язык Z; чтобы обработать задачу, полезно выделить уровни. Попробуем сначала изложить ее на языке Z_0, на глобальном уровне. (Пронумеруем формулы, определяющие множества: E_1, E_2 и т.д.; функции: F_1 и т.д.; отношения: R_1, и т.д.)

Множества, участвующие в задаче, это прежде всего файлы A, B, C, D. Исходный файл B содержит «статьи»; постулируем, таким образом, существование множества:

Статьи

и имеем

$B \subseteq$ Статьи

Мы видели, что всякая статья состоит из ключа и некоторого числа пар [имя поля, значение]. Это записывается как декартово . произведение

Статьи = Ключи × **Кортеж** (Именаполей × Значения) (E5)

где Ключи – множество ключей, Именаполей – множество имен полей, Значения – множество значений. Если X – множество, то обозначение **Кортеж**(X) означает на языке Z_0 множество

Кортеж(X) = X∪X × X∪X × X × X∪ ...

т.е. множество конечных последовательностей элементов из X. Так, [3], [8, 5, 9], [45, 12, 50, 10] и т.д. принадлежат множеству **Кортеж**(Целые).

[1] Этот пример был исследован совместно с Демунком, Майаром и Муленом.

Наконец, элемент из D является триплетом [ключ, номер подфайла, значение]; таким образом, имеем

$$D \subseteq \text{Ключи} \times \text{Номподфайла} \times \text{Значения} \qquad \text{(E9)}$$

где Номподфайла – множество возможных номеров подфайлов. Напоминаем еще раз, что речь идет о следующих множествах:

(E1)	Ключи
(E2)	Именаполей
(E3)	Значения
(E4)	Номподфайла
(E5)	Статьи = Ключи × Кортеж (Именаполей × Значения)
(E6)	B ⊆ Статьи
(E7)	A ⊆ Ключи
(E8)	C ⊆ Именаполей × Номподфайла
(E9)	D ⊆ Ключи × Значения × Номподфайла

На глобальном уровне реализуемую обработку можно выразить как функцию без параметра, исходное множество которой является специальным одноэлементным множеством, обозначаемым 1:

$$1 \xrightarrow{\text{Обраб}(0.-)} D \qquad \text{(F1)}$$

Обраб – это многозначная функция, о чем свидетельствует использование заглавной буквы: создаются 0, 1 или несколько записей из D.

Статьи файла B обрабатываются, «переводятся» по одной, что подсказывает форму записи:

$$\text{Обраб} = \bigcup_{b \in B} \text{Перевод}(b) \qquad \text{(R'l)}$$

с функцией

$$B \xrightarrow{\text{Перевод}(0,-)} D \qquad \text{(F2)}$$

где Перевод – многозначная функция, так как из каждой статьи, принадлежащей B, можно создать несколько статей файла D. Соотношение (R'l), приведенное выше, однако, неполно, так как было показано, что не все элементы B обрабатываются, если A непусто: его надо заменить соотношением

$$\text{Обраб} = \bigcup_{b \in B.\gamma(b)} \text{Перевод}(b) \qquad \text{(R1)}$$

Смысл этого обозначения состоит в том, что действие «объединения» производится лишь над теми b, которые удовлетворяют условию y (o), такому, что

$$\text{Статья} \xrightarrow{\gamma(1)} (\textbf{истина}, \textbf{ложь}) \qquad \text{(F3)}$$

и

$$\forall b \in \text{Статьи}.\gamma(b) = A = \varnothing \text{ или } b(1) \in A) \qquad \text{(R2)}$$

где Статьи – декартово произведение

Ключи × **Кортеж**(Именаполей × Значения);

b(1) означает первую компоненту статьи b, ее ключ.

Теперь надо выразить функцию Перевод. Она оперирует с каждой из пар [имя поля, значение], образуя вторую составляющую b(2) статьи b. Эти пары обрабатываются по отдельности, и этот факт мы выразим, введя функцию перевода пары:

$$\text{Именаполей} \times \text{Значения} \xrightarrow{\text{перевэлем}(1)} \text{Значения} \times \text{Номподфайла} \qquad \text{(F'3)}$$

и выражая Перевод с помощью

$$\forall b \in B.\ \text{Перевод} = \bigcup_{x \in b(2)} b(1) * \text{перевэлем}(x) \qquad \text{(R'3)}$$

Напомним, что звездочкой обозначается конкатенация кортежей.

Чтобы выразить функцию перевэлем, ставящую каждой паре [имя поля, значение] в соответствие пару [значение, номер подфайла], вводится функция номпод, позволяющая поставить номер подфайла в соответствие имени поля:

$$\text{Именаполей} \xrightarrow{\text{номпод}(1)} \text{Номподфайла} \qquad \text{(F4)}$$

Напомним, что эта функция представлена файлом C. В силу (E8) $C \in \text{Именаполеи} \times \text{Номподфайла}$, и более точно получается:

$$\forall c \in C.\text{номпод}(c(1)) = c(2) \qquad \text{(R4)}$$

Отметим, что существование файла C эквивалентно существованию функции номпод; эта эквивалентность выражается соотношением (R4). Подобная избыточность – одно из свойств программирования на уровне Z_0: выбор между представлением при помощи функции (= подпрограммы) и представлением с помощью множества (= структуры данных) делается только в момент алгоритмизации. На уровне Z_0 отказываются от выбора между пространством и временем, поскольку этот вопрос преждевременный, и сохраняют две избыточные формулировки.

Функция номпод позволяет нам явно выразить функцию перевэлем:

$$\forall b \in B.\ \forall x \in b(2).\ \text{перевэлем}(x) = \left[x(2),\ \text{номпод}(x(1))\right] \qquad \text{(R'4)}$$

в соответствии с заданным описанием обработки пары.

Остается указать, что если файл A пуст, то он содержит ключи, однозначно обеспечивающие доступ к элементам файла B, поскольку два различных элемента B имеют различные ключи. Это выражается в существовании инъективной функции:

$$B \xrightarrow[\text{статья с ключом}(0)]{\text{ключ}(1)} \text{Ключи} \qquad \text{(F5)}$$

Важной функцией здесь является обратная функция статья с ключом; записывая ее как однозначную функцию, мы выразили, что к элементу файла B можно получить доступ единственным способом – при помощи ключа. Эта функция является не всюду определенной, на что указывает 0 в ее определении: не все возможные ключи обязательно представлены в B. Заметьте, что, как и раньше, утверждение о существовании всюду определенной функции ключ избыточно при наличии предыдущих теоретико–множественных соотношений (E5) и (E6); соответствие выражено отношением

$$\forall b \in B.\ \text{ключ}(b) = b(1) \qquad \text{(R5)}$$

Наконец, располагая промежуточной функцией перевэлем, с помощью (R'4) можно переписать (R'3), что позволяет получить в итоге следующую окончательную программу Z_0:

МНОЖЕСТВА

$$\text{Ключи} \tag{E1}$$

$$\text{Именаполей} \tag{E2}$$

$$\text{Значения} \tag{E3}$$

$$\text{Номподфайла} \tag{E4}$$

$$\text{Статьи} = \text{Ключи} \times \text{Кортеж}\,(\text{Именаполей} \times \text{Значения}) \tag{E5}$$

$$B \subseteq \text{Статьи} \tag{E6}$$

$$A \subseteq \text{Ключи} \tag{E7}$$

$$C \subseteq \text{Именаполей} \times \text{Номподфайла} \tag{E8}$$

$$D \subseteq \text{Ключи} \times \text{Номподфайла} \times \text{Значения} \tag{E9}$$

ФУНКЦИИ

$$1\,\frac{\text{Обраб}(0.-)}{}\,D \tag{F1}$$

$$B\,\frac{\text{Перевод}(0,-)}{}\,D \tag{F2}$$

$$\text{Статья}\,\frac{\gamma(1)}{}\,(\textbf{истина}, \textbf{ложь}) \tag{F3}$$

$$\text{Именаполей}\,\frac{\text{номпод}(1)}{}\,\text{Номподфайла} \tag{F4}$$

$$B\,\frac{\text{ключ}(1)}{\text{статья с ключом}(0)}\,\text{Ключи} \tag{F5}$$

ОТНОШЕНИЯ

$$\text{Обраб} = \bigcup_{b \in B.\gamma(b)} \text{Перевод}(b) \tag{R1}$$

$$\forall b \in \text{Статьи}.\gamma(b) = A = \varnothing \text{ или } b(1) \in A) \tag{R2}$$

$$\forall b \in B.\ \text{Перевод} = \bigcup_{x \in b(2)} b(1) * \text{перевэлем}(x) \tag{R3}$$

$$\forall c \in C.\text{номпод}(c(1)) = c(2) \tag{R4}$$

$$\forall b \in B.\ \text{ключ}(b) = b(1) \tag{R5}$$

Отметим, что если порядок записей в файле D важен и обязательно такой же, как и в файле B, то необходимо объединения заменить конкатенациями (*).

Чтобы перейти к программе на языке Z_1 мы переведем вначале (R1) при помощи следующей версии:

```
        Обраб ← ∅;
        для b ∈ B повторять
(V1)        | если γ (b) то
            |     | Обраб ← Обраб ∪ Перевод (b)
```

т.е. записывая в явном виде условие γ(b):

```
        Обраб ← ∅;
        для b ∈ B повторять
(V2)        | если A = ∅ или b(1) ∈ A то
            |     | Обраб ← Обраб ∪ Перевод (b)
```

Поскольку условие «A = ∅» не зависит от b, его можно вынести за цикл, чтобы отдельно обрабатывать случаи A = ∅ и A ≠ ∅. Речь идет об одном из стандартных «преобразований», приведенных в книге [Абриаль 77a]. Оно дает следующую версию:

```
        Обраб ← ∅;
        если A = ∅ то
            | для b ∈ B повторять
            |     | Обраб ← Обраб ∪ Перевод (b)
(V'3)   иначе {A ≠ ∅}
            | для b ∈ B повторять
            |     | если b(1) ∈ A то
            |     |     | Обраб ← Обраб ∪ Перевод (b)
```

Можно, однако, продолжать разработку программы несколько иным образом. Существование не всюду определенной однозначной функции статья с ключом (свойство (F5)) действительно показывает, что в случае, когда A ≠ ∅, можно получить доступ к элементу b, удовлетворяющему свойству b(1) ∈ A, т.е. ключ(b) ∈ A (отношение (R5)). Таким образом, в случае A ≠ ∅ можно построить алгоритм, основывающийся на последовательном просмотре файла A, а не файла B. Здесь выбран именно этот путь; заметьте, что такое решение оправдано лишь тогда, когда есть надежда на выигрыш в эффективности, другими словами, если в файле A гораздо меньше ключей, чем статей в B. Получаем

```
        Обраб ← ∅
        если A = ∅ то {как раньше: B обрабатывается последовательно}
            | для b ∈ B повторять
            |     | Обраб ← Обраб ∪ Перевод (b)
        иначе {A ≠ ∅: «прямой» доступ к элементам B
(V3)        |           посредством их ключей}
            | для a ∈A повторять
            |     | b ← статья с ключом (a);
            |     | если b ≠ пусто (т.е. если функция «статья с ключом»
            |     |                 определена над a} то
            |     |     | Обраб ← Обраб ∪ Перевод (b)
```

Используем теперь (R3) для того, чтобы развернуть Перевод (b); обозначение «X : э x», где X – множество, а x – объект, означает в языке Z оператор включения, добавляющий элемент x к множеству X[1]

[1] Отметим формальное соответствие между обычным оператором присваивания a : = b, делающим истинные условие «a = b», и теоретико–множественным присваиванием X : э x, которое дает значение истина условию «X э x» или «X содержит x»

```
Обраб ← ∅
если A = ∅ то
    для b ∈ B повторять
        для x ∈ b(2) повторять
            Обраб: э [b(1), x(2), компод (x(1))]
иначе
(V4)
    для a ∈A повторять
        b ← статья с ключом (a);
        если b ≠ пусто то
            для x ∈ b(2) повторять
                Обраб: э [b(1), x(2), компод (x(1))]
```

Следующий этап состоит в устранении теоретико–множественных циклов для, выражая их путем явных последовательных просмотров. В соответствии с приведенными выше определениями операций **открыть**, **читать** и специального значения пусто получим версию (V5):

```
Обраб ← ∅;
a ← открыть (A);
если a = пусто то
    b ← открыть (B);
    пока b ≠ пусто повторять
        x ← открыть (b (2));
        пока x ≠ пусто повторять
            Обраб: э [b(1), x(2), компод (x(1))];
            x ← читать (b (2))
(V5)        b ← читать (B)
иначе
    повторять
        b ← статья с ключом (a);
        если b ≠ пусто то
            x ← открыть (b(2));
            пока x ≠ пусто повторять
                Обраб: э [b(1), x(2), компод (x(1))];
                x ← читать (b(2))
        a ← читать (A);
    до a = пусто
```

Заметим, что, кроме введения обозначения b(1) вместо ключ(b) мы приняли мало решений, связанных с реализацией программы. В частности, операторы **открыть** и **читать** не обязательно определяют физический ввод–вывод, но могут служить доступом к массиву (i ← 1 для **открыть**; i ← i + 1 для **читать**) или к цепной структуре ℓ(x ← первое(ℓ); x ← следующее (x)).

Остается еще сделать много преобразований этой программы: надо выбрать способ доступа к файлу A и файлу B, заменить теоретико–множественное присваивание в Обраб на оператор записи и т.д. Однако метод должен быть ясен.

В качестве дополнения приведем программу на КОБОЛе (один раз не в счет), которая дает работающую версию приведенной выше программы на Z и полученную ее последовательным преобразованием. Приведены лишь *РАЗДЕЛ ПРОЦЕДУР*[1] и часть *РАЗДЕЛА ИДЕНТИФИКАЦИИ*.

[1] Учитывая наличие ГОСТа языка КОБОЛ (КОБОЛ 77), допускающего русские ключевые слова и идентификаторы, а также несколько «русских» реализаций этого языка, ключевые слова и часть идентификаторов в приводимом ниже примере программы на КОБОЛе даются на русском языке. – *Прим. перев.*

Чтобы понять эту программу, полезно знать, что принятые в этой реализации физические представления таковы:

- A – внешний «последовательныйфайл», который может находиться на перфокартах, на диске или на магнитной ленте. Доступ к этому файлу осуществляется оператором чтения *ЧИТАТЬ* считывающим записи по порядку до конца файла, на который указывает специальное условие *В КОНЦЕ.* Считанные записи помещаются в цепочку литер *ЗАП–А*: связь между файлом и его «областью ввода» осуществляется в КОБОЛе при трансляции с помощью *СЕКЦИИ ФАЙЛОВ* в *РАЗДЕЛЕ ДАННЫХ;*
- D ¯ последовательный файл, аналогичный A; он строится с помощью последовательных операторов *ПИСАТЬ,* которые записывают каждую имеющуюся в *ЗАП–А* запись; что касается связи файла с его «областью вывода», она осуществляется аналогично связи между файлом и его «областью ввода»;
- C – представлен массивом *TABC,* упорядоченным по номерам подфайлов: значение *TABC(I)* – это имя поля, соответствующее подфайлу с номером *I.* Загрузка массива в память не представлена;
- наконец, B – это «база данных», управляемая системной программой IMS, хранящейся на диске. Ее иерархическая структура такова: каждая запись из B, как она была определена выше, представляется «корнем», содержащим ключ b(1). к которому присоединены «зависимые сегменты» в некотором количестве; каждый из них представляет пару [имя поля, значение] записи b(2); их «тип сегмента»–это имя поля, а их содержимое–соответствующее значение.

Обращения к базе данных выполняются с помощью программы *CBLTDLI,* которой сообщают:

- параметр, который характеризует тип операции ввода–вывода: *GU* означает «считать первый корень из базы данных»; под *GN* понимается «считать следующий сегмент или корень»; *GNP* означает здесь «считать последовательно сегменты, которые иерархически зависят от последнего считанного корня»;
- параметр *PCB*, который сообщает операционной системе некоторые необходимые для оптимального управления базой данных сведения, среди которых имеются два подмножества, полезных для программы пользователя: *КОД–ВЫДАЧИ* ('*GE*' означает «ненайденный корень или сегмент», *GB* означает «конец файла достигнут») и *ТИП–СЕГМЕНТА,* означающий в данном случае имя поля, которому поставлено в соответствие считанное значение;
- параметр, уточняющий, какую область надо использовать для засылки считанного корня или сегмента;
- факультативный параметр, позволяющий отбирать некоторые записи; он составляется из цепочки литер, записанных на специальном управляющем языке, названном DL/1 и декодируемом при выполнении программы с помощью интерпретатора, содержащегося в системе управления базами данных IMS: здесь *ВЫБОР–КОРНЯ* заставляет считывать одни лишь корни, а *ВЫБОР–КОРНЯ–КЛЮЧОМ* требует прямого доступа к корню с данным ключом (прочитанным в файле A). Это было выражено в Z утверждением существования не всюду определенной однозначной функции «статья с ключом», позволяющей доступ к элементу файла B при помощи его ключа, содержащегося в файле A.

С методологической точки зрения можно заметить, что именно в этом случае конкретная структура файлов была известна априори; таким образом, программа на Z_0 представляет собой попытку абстракции от их характерных свойств. Нам кажется, что форму на языке Z_0 более легко использовать, чем подробное описание базы в используемой системе управления базами данных (IMS), и она менее зависит от нефундаментальных технических решений, которые могут понадобиться позже. Но программирование на Z позволяет вернуться к конкретной системе, что мы и делаем здесь, сохраняя описание самого высокого уровня абстракции.

КОБОЛ

```
РАЗДЕЛ ПРОЦЕДУР.
...
    ОТКРЫТЬ ВХОДНОЙ A
    ОТКРЫТЬ ВЫХОДНОЙ D
    ПОМЕСТИТЬ ' BI' В ИМЯ–ВЫБОРА
    ЧИТАТЬ A В КОНЦЕ ПЕРЕЙТИ К A–ПУСТО.
МЕТКА1.
    ПОМЕСТИТЬ ЗАП–A В КЛЮЧ–ВЫБОРА
```

```
        ВЫЗВАТЬ 'CBLTDLI' ИСПОЛЬЗУЯ
            GU PCB ОБЛ–В1 ВЫБОР–КОРНЯ КЛЮЧОМ
        ЕСЛИ КОД–ВЫДАЧИ НЕ = 'GE'
            ВЫЗВАТЬ 'ПЕРЕВ' ИСПОЛЬЗУЯ ОБЛ–В1 PCB.
        ЧИТАТЬ А В КОНЦЕ ЗАКРЫТЬ А ВЕРНУТЬСЯ.
        ПЕРЕЙТИ К МЕТКА1.
    А–ПУСТО.
        ВЫЗВАТЬ 'CBLTDU' ИСПОЛЬЗУЯ GU PCB ОБЛ–В1
                                              ВЫБОР–КОРНЯ
        ЕСЛИ КОД–ВЫДАЧИ = 'GE' ЗАКРЫТЬ А, D
                                              ВЕРНУТЬСЯ.
    МЕТКА2.
        ВЫЗВАТЬ ПЕРЕВ ИСПОЛЬЗУЯ ОБЛ–В1 PCB
        ВЫЗВАТЬ 'CBLTDLT' ИСПОЛЬЗУЯ GN PCB ОБЛ–В1
                                              ВЫБОР–КОРНЯ
        ЕСЛИ КОД–ВЫДАЧИ = 'GB' ЗАКРЫТЬ А, D
        ВЕРНУТЬСЯ.
        ПЕРЕЙТИ К МЕТКА2.
```

КОБОЛ

```
    РАЗДЕЛ ИДЕНТИФИКАЦИИ.
    ПРОГРАММА.
       ПЕРЕВ.
       ...
    РАЗДЕЛ ПРОЦЕДУР ИСПОЛЬЗУЯ 0БЛ–В1 PCB.
    МЕТКА1.
            ВЫЗВАТЬ 'CBLTDLT' ИСПОЛЬЗУЯ GNP PCB ОБЛ–Х2
            ЕСЛИ КОД–ВЫДАЧИ = 'GE' ИЛИ 'GB' ВЕРНУТЬСЯ.
            ВЫПОЛНИТЬ ПОИСК–НОМПОЛ ПО КОНЕЦ–ПОИСКА
            ЕСЛИ I = О ПЕРЕЙТИ К МЕТКА1.
            ПОМЕСТИТЬ ОБЛ–В1 ЗАП–D (1)
            ПОМЕСТИТЬ ОБЛ–Х2 В ЗАП–D (2)
            ПОМЕСТИТЬ НОМПОД (I) В ЗАП–D (3)
            ПИСАТЬ ЗАП–D
            ПЕРЕЙТИ К МЕТКА1.
    ПОИСК–НОМПОД.
            ПОМЕСТИТЬ 0 В I.
    ЦИКЛ.
            СЛОЖИТЬ 1 С I
            ЕСЛИ 1 > I MAX ПОМЕСТИТЬ 0 В I
                    ПЕРЕЙТИ К КОНЕЦ–ПОИСКА.
            ЕСЛИ ТИП–СЕГМЕНТА НЕ = ТАВС (I)
            ПЕРЕЙТИ К ЦИКЛ.
    КОНЕЦ–ПОИСКА.
    ВЫЙТИ.
```

VIII.3.5.5. Обсуждение

Основное преимущество использования языка Z заключается в том, что он позволяет ясно формулировать некоторые из принципиальных вопросов, являющихся ключевыми моментами реализации программы. А в программировании, как и в любой научной дисциплине, обнаружение принципиальных проблем зачастую представляет собой наиболее трудный этап в отыскании решений.

При осуществлении крупного проекта программирования большая часть трудностей зачастую происходит из–за неточности решаемой задачи: мы сталкиваемся с тяжелым, неоднородным, неполным, противоречивым, беспорядочным «техническим заданием».

Каждая из элементарных задач зачастую бывает проста; по крайней мере, почти всегда действительные принципиальные трудности сводятся к небольшому числу подзадач. Настоящие трудности связаны с множеством требований, с определением их взаимных отношений, с тем, что «кишмя кишат» спецификации, вызывая отчаяние программистов и руководителей проектов.

При столкновении с таким положением, типичным для информатики АСУ, но встречающимся также во всех других предметных областях, часто возникает чувство, что стоит только полностью поставить задачу, как последовательность разработки программы «потечет сама собой». Именно полная и строгая постановка задачи является целью языка Z_0. Но в отличие от технического задания, составленного на французском языке[1] (и не специалистами в области информатики), программа на Z_0 непосредственно пригодна для работы: из нее можно получить программу в обычном смысле слова, т.е. алгоритмическую программу при помощи вышеуказанных преобразований и в принципе механизируемым способом, хотя и при настоящем состоянии техники интуиция и опыт программиста, безусловно, продолжают играть важную роль.

Опыт программирования на Z показывает, что иногда возникает искушение при решении какой–нибудь задачи быстро перейти от уровня Z_0 к «алгоритмической» программе на Z_1 даже несмотря на то, что задача полностью специфицирована на Z_0; традиции обычных языков программирования дают основание считать, что подзадачи, которые мы не потрудились выразить подробно, тривиальны и что их можно будет решить по ходу дела при написании программы на Z_1. Однако весьма часто в языке Z_1 сталкиваются с трудностями или противоречиями, кажущимися непреодолимыми; если попытаться найти причину этого, можно заметить, что в «алгоритмизации» заключена тенденция систематически *переопределять* задачу, обязывая принимать произвольные решения, которые могут оказаться очень ограничительными и связать программиста до такой степени, что на каком–нибудь более позднем этапе он может зайти в тупик.

Типичным примером переопределенности подобного рода является **порядок перебора** элементов множества: очень часто бывает необходимо произвести действие a(x) над всеми элементами x множества A, так что порядок обработки элементов этого множества безразличен. При записи программы в классическом смысле необходимо заранее указать порядок перебора элементов множества, даже если он задан неявно; например, если элементы множества сортируются по некоторому ключу и если к ним обращаются последовательно, порядок просмотра предполагается соответствующим ключу. На более позднем этапе может обнаружиться, что порядок перебора становится существенным, и при этом требуется другой порядок; в этом случае можно оказаться связанным слишком рано выбранным нами решением. В языке Z_0, напротив, никакой порядок перебора элементов не задается при представлении преобразования f, соответствующего a, и его свойств:

$$\begin{cases} A \xrightarrow{f(\ldots)} B \\ \forall x \in A.\ f(x) = \ldots \end{cases}$$

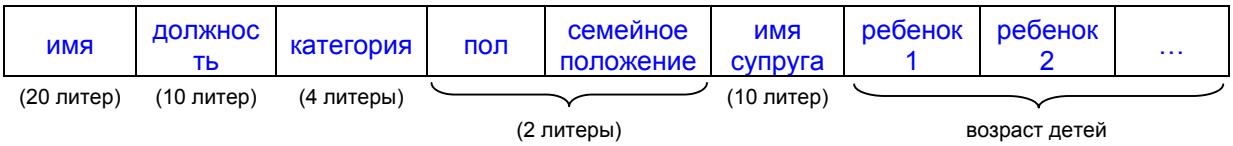
Заметим, что здесь первый уровень Z_1 позволяет еще не специфицировать порядок перебора элементов, если он не необходим:

для x ∈A **повторять**
  | a(x)

[1] или на русском.– *Прим. перев.*

Однако, начиная с этапа Z_1 каждый раз, когда пишут два оператора подряд, предписывается порядок выполнения, который может быть несущественным.

Подобная же проблема переопределенности возникает в связи со структурами данных и компромисса пространство–время. Сколько программистских проектов тормозится слишком поспешным выбором представления! Возьмем хотя бы весьма простой пример файла персонала при расчете заработной платы. Предположим, что раз и навсегда договорились считать его составленным из записей такого вида:



Хорош ли этот первоначальный выбор или плох, не имеет значения. Но поскольку данный формат фиксирован и известен, все программы будут иметь доступ к должности как к полю, содержащему литеры с 21–й по 30–ю в каждой записи, к семейному положению – как к 36–й литере и т.д. (предполагается, что авторы этих программ не читали гл. V).

Предположим теперь следующее: при правильных программах в момент загрузки файла замечают, что он требует неприемлемого объема памяти (оперативной или внешней). Тогда предпринимается попытка сократить необходимый объем памяти, и, конечно, этого удается достигнуть не очень дорогой ценой. Число возможных должностей, разумеется, ограничено самое большее несколькими тысячами, и 14 битов, представляющих указатель к «массиву должностей», без сомнения, достаточно. Фамилии повторяются, и, следовательно, может быть оправдано применение типичного метода ассоциативной адресации и даже специальных методов, использующих естественную избыточность обычных языков (*«сортировка»;* см. [Кнут 73], 6.3) и позволяющих экономить место в значительных пропорциях. Простое принятие переменного формата может уже дать хорошие результаты, если большинство фамилий состоит менее чем из 10 букв. Пол можно представить одним битом. Фамилия супруги (или супруга) обычно та же, что и фамилия служащего: при переменном формате используют один бит, указывающий, что такой факт имеет место, и только в противном случае используется область, описывающая фамилию супруги (супруга), или указатель, если эта фамилия уже фигурирует в файле.

Подобные модификации являются результатом использования технических навыков профессионального программиста; они могут сократить объем весьма значительно, даже позволить в некоторых случаях хранить в ОЗУ весь файл, который, казалось, требовал для своего хранения целый диск. Заметьте, что обратный прием тоже допустим: если «драгоценным товаром» является время вычисления, а места в памяти достаточно, целесообразно повторить во многих экземплярах избыточные данные, чтобы сократить до предела выполняемые вычисления.

Важным моментом является то, что здесь решения о представлении данных зависят от окружающих условий, тестов и т.п. Принимать такие решения на первоначальной стадии было бы несвоевременно; это может привести к многочисленным переписываниям и цепным изменениям данных при каждом их новом упоминании, что обычно и происходит.

Напротив, пусть дана «программа» на Z_0:

$$\text{Служащие} \xrightarrow[\text{Служащий с именем}(1,-)]{\text{имя}(1)} \text{Имена}$$

{Служащие – это множество служащих, Имена – множество имен; они связаны между собой функцией имя: всякий служащий имеет одно и только одно имя; одно имя может принадлежать одному или нескольким служащим}

$$\text{Служащие} \xrightarrow{\text{должность}(1)} \text{Должности}$$

$$\text{Служащие} \xrightarrow{\text{категория}(1)} \text{Категории}$$

$$\text{Служащие} \xrightarrow{\text{пол}(1)} (\text{мужской, женский})$$

$$\text{Служащие} \xrightarrow{\text{семейное-положение}(1)} (\text{вдовец, разведен, женат, холост})$$

$$\text{Служащие} \xrightarrow{\text{Дети}(0,-)} \text{Возраст-детей}$$

Среди соотношений может найтись, например, такое (предположим для этой цели, что учреждение нетерпимо к нарушению морали своим персоналом):

$$\forall x \in \text{Служащие.семейное–положение}\ (x) = \text{холост} \Rightarrow \text{Дети}(x) = \varnothing$$

Если каждой должности соответствует одна и только одна категория, заметим, что существует функция

$$\text{Должности} \xrightarrow[\text{Соответствующие должности}(1,\ -)]{\text{соответствующая степень}(1)} \text{Категории}$$

В терминах физического представления существование этой функции означает, что не строго необходимо, чтобы каждая запись содержала поле «категория», так как категория может быть «вычислена», исходя из должности (при помощи таблицы). Но программа на Z_0 ничего не предписывает: она дает элементы, позволяющие выбирать наилучшее физическое представление. Среди этих сведений самыми важными для последующего выбора в языке. Z_1 являются те, которые определяют, каковы для функции f максимальное и минимальное число элементов, соответствующих каждому элементу (например, (1,1) для функции должность, приведенной выше), и те, которые указывают характеристики обратной функции. Это объясняет, почему для них было выработано специальное обозначение (они могут быть выражены с помощью отношений, как в функциональных спецификациях гл. V).

Вывод из этих замечаний тот, что надо устоять перед упоминаемым выше искушением слишком быстро переходить на язык Z_1 и *всегда полностью специфицировать задачу в языке* Z_0, прежде чем перейти к следующим уровням.

Бегло представленный здесь метод программирования, использующий первоначально «статический» подход к программам и повторные преобразования, по–видимому, дает интересные решения для многих важных задач программирования. Фактически язык Z отражает важную эволюцию. Учитывая это, можно, однако, сделать несколько оговорок:

- использование формализма, основанного на сравнительно прогрессивных математических обозначениях теории множеств, ставит некоторое число психологических и педагогических проблем–впрочем, может быть, в меньшей степени из–за внутренней сложности этих обозначений, в конечном счете более простых, чем ФОРТРАН, КОБОЛ или (разумеется!) ПЛ/1, чем из–за особого статута («тотем и табу»), который имеют

математики в нашем обществе;

- важное различие статического и динамического аспектов не разрешает проблемы декомпозиции сложной задачи. Фактически программа на языке Z_0, если она достаточно велика, ставит те же проблемы уменьшения сложности, что и обычная программа. Применимы обычные методы: можно говорить о нисходящем и восходящем программировании на языке Z_0, и напрашивается стратегия решения задач, обращающаяся к этим понятиям, как только проекты превосходят некоторый объем. Схематично процесс программирования на языке Z_0, состоящий в определении множеств, функций и отношений (подход, связанный с нисходящим методом) приводит к выделению функций, относящихся к выполняемой обработке, прежде чем детально описывать множества, к которым они применяются; подход же, связанный с восходящим методом, напротив, приводит к полному определению множеств до того, как мы заинтересовались применяемыми к ним функциями;
- можно спросить, оправданы ли лишние затраты труда, которые требуются для построения программы на Z_0. Эта работа действительно достаточно тяжела. Единственный ответ на это возражение–это констатация того факта, что статическая программа в любом случае неявно строится при обычном программировании, и важно иметь ее явную версию, чтобы уметь различать фундаментальный выбор и выбор представления; при этом дополнительными преимуществами являются возможности оптимизации и наличие строгой и поддающейся обработке документации (программа на Z_0), действительно соответствующей реализованным программам.

VIII.3.6. Программа и ее преобразования

Полезно подчеркнуть мысль, естественно вытекающую из всех изложенных в предыдущих разделах замечаний: **нельзя составить программу за один этап.** Программирование–слишком сложный процесс, чтобы можно было рассчитывать получить немедленно по мановению волшебной палочки программу, удовлетворительную с точки зрения всех разумных требований. Программа должна рассматриваться как некоторый постоянно совершенствуемый объект: из начальной версии, от которой требуется лишь, чтобы она была корректной даже, может быть, в ущерб легкости ее отладки, эффективности и т.д., путем последовательных преобразований получается наконец такой вариант, который обладает различными нужными нам качествами. Выше рассматривались подобные преобразования – переход с некоторого уровня абстракции к уровню, более близкому к машине, переход от статической программы к динамической и т.д. Другими важными преобразованиями являются те, которые позволяют получить из концептуально точного, но, возможно, неэффективного варианта программы последовательность программ, все более приспособленных к конкретной среде ее выполнения (язык, машина). В частности, это случай преобразований, уменьшающих степень рекурсивности программы либо выражающих ее нерекурсивным способом для реализации на ФОРТРАНе, например преобразований, заменяющих структуру таблицы более специфичной реализацией (ассоциативной адресацией, двоичным деревом АВЛ, В–деревом...), которая учитывает свойства доступа, моделирующего динамическое распределение памяти, и т.д.

Этот метод итеративного улучшения программ, по–видимому, единственный, позволяющий получить полностью удовлетворительный конечный продукт. При его использовании возникает целый ряд вопросов: действительно, надо гарантировать корректность последовательных версий, другими словами, надо, чтобы правильность программы относительно некоторой спецификации оставалась «инвариантом» при последовательных преобразованиях. Законным является, например, такой вопрос: во

что превращаются в ходе последовательных преобразований промежуточные спецификации, «утверждения», связанные с отдельными пунктами программы?

В работе [Кнут 76] можно найти такую полную разработку программы (решающей задачу об «устойчивых паросочетаниях»). Можно предвидеть, что в будущем диалоговые системы помогут программисту осуществить подобные преобразования; такая система, позволяющая, в частности, устранить бесполезные рекурсии, описана в статье [Дарлингтон 76]. Теоретические основы этих методов можно найти в [Берсталл 77] и [Арсак 77].

VIII.4. Качества программы и возможности программиста

VIII.4.1. Должна ли программа быть хорошей?

Должна ли программа быть хорошей? Вопрос может показаться нелепым. Однако очень важно его задать: всякое усилие по улучшению методов программирования должно прежде всего быть направлено на осознание того, что программу можно обсуждать, критиковать, улучшать и сравнивать с другими программами, что она может быть хорошей или плохой и что важно сделать ее «хорошей» относительно некоторых критериев. Далее рассматриваются главные из этих критериев, а также те принципы и методы, которые даются программисту для каждого из этих критериев.

VIII.4.2. Правильность – Доказательства – Тесты

VIII.4.2.1. Правильность программы

Основным качеством программы, которое в конечном счете позволяет ее принять или заставляет отбросить, является ее **правильность** : всякая программа создается для того, чтобы отвечать некоторому требованию (некоторой *спецификации*), и должна ему удовлетворять.

> Весьма естественное требование – но какая сложная проблема! Во–первых, потому что невозможны никакие отклонения: в то время как в обычных технических дисциплинах дается некоторый допуск, а не требуется совершенное соответствие, например, допуск, равный 1%, 0,1% или 0,001% и т.д., в программировании неприемлем никакой допуск: программа, которая «почти работает», обычно вовсе не работает у пользователя, незнакомого с тонкостями программирования; он получает с помощью такой программы ложные или случайные результаты. И если кажется, что программа работает почти во всех случаях, то это еще хуже из–за сравнительного доверия, которое она вызывает, в то время как другие случаи могут оказаться решающими–как в том анекдоте, когда совершеннейшие ракеты при стрельбах повернули обратно, против стреляющих, во время первого же испытания в Гималаях: программа «работала» во всех случаях, кроме наклонов, превышающих 45°...

Проблема правильности программы трудно разрешима также и потому, что нужно уметь определить требования, которым программа должна удовлетворять. Это все та же проблема спецификаций, а мы видели, исследуя этот вопрос, что редактирование полных спецификаций является столь же сложным делом, как и само программирование, и даже, как мы заключили, изучая язык Z, оно является процессом, подобным программированию.

> [Вейнберг 71] отмечает по этому поводу, насколько могут быть нежелательными ускоренные курсы программирования для руководящих

кадров, желающих за одну неделю постигнуть «в чем там дело». Полностью специфицировать задачу нетрудно, если она находится на уровне вычисления первых восьми десятичных знаков числа π или решения уравнения второго порядка (по крайней мере в первом приближении, когда рассматривается «нормальный» случай и не учитываются численные проблемы, связанные с ошибками, переполнениями и т.д.; в противном случае задача очень быстро осложняется...). Необходимость строгой спецификации и строгой дисциплины программирования не встает на этом уровне, и средний руководящий работник, даже если он входит в состав дирекции предприятия, в конце концов получает программу на БЭЙСИКе, которая делает почти то, что от нее требуют. Наш руководитель укрепится в мысли, что программирование не слишком трудная задача. Но если перейти к спецификациям и программированию задач из такой области, как, скажем, бухгалтерский учет большого предприятия...

Если намечают целью создание программы, правильность которой должна быть абсолютной, и если решена проблема спецификации решаемых задач, остается третья серьезная проблема: проблема связи между программой и спецификациями, которым она претендует удовлетворять, т.е. *доказательство* правильности программы. Возможность такого доказательства является свойством «хорошей» программы, и мы посвящаем этому следующий раздел.

VIII.4.2.2. Доказательства: их роль и границы

То, что можно доказать в математическом смысле слова некоторые свойства программы, и в частности правильность выходных утверждений, исходя из правильности входных утверждений, не является открытием для читателя этой книги. В разд. III.4 мы ввели аксиоматические свойства управляющих структур и основных операторов, принадлежащие Хоару. Эти свойства позволяют показать, что некоторые *утверждения* удовлетворяются в определенных точках программы. Эти свойства часто использовались в предыдущих главах при изложении алгоритмов, и в частности понятия *инварианта цикла.*

Уточним, что аксиоматика Хоара не является единственной системой, позволяющей обрабатывать программы как формальные объекты со строго определенными математическими свойствами. Среди соперничающих систем можно указать семантику Скота–Стречи [Донэгю 76], [Теннент 76], [Стой 74]; метод «символического выполнения» [Хантлер 76]; метод абстрактных интерпретаторов (Венский язык) [Вегнер 72].

Какой бы формализм ни использовался, на практике, однако, при попытке доказать правильность даже простых программ быстро сталкиваются с непредвиденными трудностями и доказательства становятся очень скоро весьма тяжеловесными. При современном состоянии техники полное доказательство правильности программы большого размера неосуществимо. Эта цель кажется еще более недостижимой, если учесть, что построение доказательства является видом человеческой деятельности примерно того же концептуального уровня, как и программирование, и так же, как и программирование, допускает возможность ошибок. Следовательно, нет априорной причины предполагать, что доказательство программы имеет больше шансов быть правильным, чем само программирование.

Мы кончим сгущать краски, добавив, что не надо дать себя обольстить оптимизмом, скрытым в формуле: «при современном состоянии техники», которую мы применили выше, так как некоторое число серьезных теоретических невозможностей не дает нам права надеяться на изобретение некоего универсального метода доказательства–даже если, как можно предвидеть, современные методы улучшатся (об этих теоретических невозможностях см. [Минский 67]; некоторые основные идеи даны в упр. III.11 и в разд. VI.1.3 этой книги).

Означает ли это тогда, что доказательства не представляют никакого интереса для программиста? Ни в коем случае! Из оговорок, высказанных ранее, следует, что (вообще говоря) напрасно пытаться доказать апостериори корректность уже существующей программы. Напротив, исключительно плодотворный метод, впервые предложенный в книге [Дейкстра 68a], состоит в **написании программы, исходя из доказательства,** или, в более общем виде, в одновременном построении программы и доказательства. Мы пытались несколько раз применять этот метод, в частности при изложении Деления (для Быстрой Сортировки) в VII.3.6 и дихотомического поиска в VII.2.3. Формализм Хоара особенно хорошо для этого подходит; в частности, мы видели тесную связь между понятием *цикла* – динамического элемента, являющегося частью программы, – и понятием *инварианта цикла* – статического элемента, принадлежащего доказательству. В самом деле, цикл является реализацией инварианта и условия останова (можно вспомнить перевод свойства Z_0, использующего кванторы $\forall$ и $\exists$ в цикле Z_1); цикл будет тем лучше понят и будет иметь тем больше шансов быть правильным, если он будет строиться и изучаться одновременно с соответствующими ему инвариантом и условием останова. Мы обращаем внимание на проблему циклов по той причине, что речь идет об управляющей структуре, которую труднее всего понять: представляющее неизвестное априори и неограниченное число действий повторение типа пока требует от того, кто пишет или читает программу, хорошего умения абстрагироваться. Такие вспомогательные средства, как методы доказательства и, в частности, понятие инварианта цикла, позволяют строить и понимать циклы более уверенно. При рассмотрении дихотомического поиска мы видели, насколько просто было допустить «легкие» ошибки при построении цикла, если рассуждать интуитивно, при этом результатом был бесконечный цикл или (что еще опаснее) цикл, который работает, за исключением «некоторых» случаев. Задача о граничных условиях особенно сложна.

Ясно, что невозможно, даже если это желательно, применять такой прием полно и строго ко всем элементам большой программы: современные методы доказательства тяжелы и их систематическое применение быстро наскучило бы. Напротив, можно рекомендовать некий процесс, находящийся посредине между интуицией и формальной строгостью, а именно: подкреплять построение программы частичными доказательствами, более или менее строгими в зависимости от места в программе; вводить в программу набор утверждений, указывающих, какие свойства удовлетворяются на разных этапах; снабжать эти утверждения убедительными аргументами, дающими наброски возможных доказательств; относиться с особым вниманием к циклам, возможно чаще указывая соответствующие им инварианты; строго строить самые «тонкие» элементы программы и те, правильность которых особенно важна, полностью доказывая при этом их правильность; писать другие элементы так, как будто вы убеждены (и убедите читателя) в том, что они *могли бы* быть доказаны, если это бы потребовалось. (Так, изучая иностранный язык, мы порой «проскакиваем» отдельные куски при чтении романа на этом языке, так как систематическое изучение всех оборотов помешало бы нам добраться до конца романа. Однако время от времени мы выбираем страницу, описывающую, например, важный момент действия, и пытаемся ее подробно проанализировать и, может быть, перевести, чтобы убедиться, что в принципе мы все можем понять.)

Идея, согласно которой программа должна быть написана так, чтобы уметь оправдать ее с помощью доказательства, нам кажется, очень способствует улучшению ее качества. В этой связи очень важно, чтобы каждый программист хотя бы один раз в жизни самостоятельно полностью доказал правильность какой–нибудь нетривиальной программы. Среди опубликованных доказательств с большой пользой можно прочитать доказательство правильности программ ПОИСКА [Хоар 71a] и БЫСТРОЙ СОРТИРОВКИ [Хоар 7Id].

Связь с предыдущим разделом устанавливается, если заметить, что

использование языка Z приводит к такому методу программирования, при котором первый этап, Z_0, легко уподобить полной редакции доказательства, а следующие, Z_1 – получению программ, все более близких к выполняемой версии, из статического «доказательства».

VIII.4.2.3. Сравнение с традиционными методами

Можно было бы спросить: в чем различие между предложенной методикой и обычным программированием? Некоторым образом всякая программа предполагает более или менее сознательное доказательство своей правильности; точнее, программа составляется с убеждением, оправданным или нет, что каждый из ее операторов должен привести к определенному состоянию программы. Так, следующие фортрановские операторы:

```
ISIG = 0
IF(Y – X .GT. EPSILO)ISIG = 1
IF(Y – X .LT. –EPSILO)ISIG = –1
```

являются «алгоритмическими» реализациями утверждения, что «целая переменная *ISIG* должна иметь значение 0, если *X* и *Y* отличаются друг от друга самое большее на *EPSILO*, и в противном случае +1, если *X* < *Y*. и –1, если *X* > *Y*», это утверждение может находиться в голове программиста в виде Рис. VIII.5:

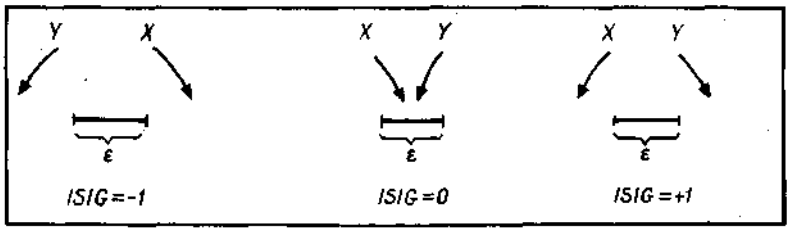


Рис. VIII.5

Это замечание поясняет принцип упомянутого метода программирования: в основном речь идет о том, чтобы явно выразить все предположения, которые обязательно имеются в сознании програм–миста при написании программы и которые вначале могут не быть ясно сформулированными. Например, по внешнему виду элемента программы на ФОРТРАНе

```
IF(I. EQ. LIMITE) H = 0.0
IF(I .NE. LIMITE) H = 1./SQRT(FLOAT(I– LIMITE))
```

ясно, что программист считал очевидным отношение *I* ≥ *LIMITE* в этом месте программы (если только квадратный корень не является неопределенным). Важно перевести на сознательный уровень предположения подобного рода–и, разумеется, в момент написания программы, а не после. В данном случае первым следствием могла бы быть замена *.NE.* на *.GT.*, но этого недостаточно, так как *H* не было бы в этом случае определено в результате несомненной ошибки при *I* < *LIMITE*. Вторая реакция, и, несомненно, лучшая, состоит в том, чтобы написать утверждение *"ЗДЕСЬ I >= LIMITE"* в качестве комментария в соответствующем месте программы и формально убедиться в его правильности. Рекомендуется на этом не останавливаться и включить динамический текст, проверяющий при выполнении (по меньшей мере на стадии отладки), что *I* ≥ *LIMITE*. Программисту особенно удобно проделывать проверки такого рода, когда существует специальное языковое средство для этого–как, например, в АЛГОЛе W оператор *ASSERT*.

VIII.4.2.4. Оператор ASSERT в АЛГОЛе W

Как бы ни был программист убежден в правильности программы, выраженной убедительным образом при помощи промежуточных утверждений, он не считает себя

непогрешимым и хотел бы, чтобы его утверждения иногда на деле сталкивались с действительностью. Что может быть проще: достаточно соотнести им тесты в окончательной программе.

Это можно осуществить на любом языке:

если утверждение неправильно **то**
    печатать соответствующее сообщение об ошибке
    останов программы

Однако на практике мы в большей степени будем склонны выполнять верификацию, если в языке имеется более простой синтаксический оборот. Например, в АЛГОЛе W оператор

ASSERT условие

выполняет проверку того, что значение *условия* является истинным (*TRUE*), и если нет, то останавливает выполнение программы, напечатав сообщение о том, что произошло. Можно найти (искусственный) пример использования оператора *ASSERT* на выходе из программы *Быстрой Сортировки* (VII.3.6) на Рис. VIII.6, другие элементы которого станут понятны после чтения разд. VIII.4.2.5.

```
0107      |      PROCEDURE TRIREC(INTEGER VALUE I, J)
0108   215.--|      BEGIN
0109         |      INTEGER S;
0110         |      IF J - I <= SEUL_DE_TRI_SIMPLE THEN
0110      108.--|      TRI_SIMPLE(I, J)
0110         |  |      ELSE
0110      107.--|      BEGIN S:= PARTITION(I, J);
0112            |      IF S - I < J - S THEN
0112          26.--|      BEGIN TRIREC(I, S - 1);
0114          26.--|      TRIREC(S + 1, J)
0114            |  |      END
0114            |      ELSE
0114          81.--|      BEGIN TRIREC(S + 1, J);
0116          81.--|      TRIREC(I, S + 1)
0116               |      END
0116            |      END
0016         |      END

0117      |      NIVEAUPILE := 0; SEUL_DE_TRI_SIMPLE := 14; TEMPS := TIME(1); REMPLIR_LE_TABLEAU;
0121      |      IMPRIMERTABLEAU;
0122      |      BEGIN
0123      |      TRIREC(1, N);
0124   1.--|      IMPRIMERTABLEAU; IMPRIMER_LE_TEMPS; WRITEON(" SEUL :", SEUL_DE_TRI_SIMPLE, ")");
0127      |      SEUL_DE_TRI_SIMPLE := SEUL_DE_TRI_SIMPLE + 1; REMPLIR_LE_TABLEAU;
---- ERROR ------------------------------------------------------------------------------------
0129      |      ASSERT 0 ¬= 0;
---- ERROR ------------------------------------------------------------------------------------
0130      |      END;
0131   0.--|      END
0131      |      END
VALUES OF LOCAL VARIABLES

 TAB(1) = 8414        TAB(2) = 9092        TAB(3) = 1411                          TAB(4) = 7569
 TAB(5) = 9590        TAB(6) = 2795        TAB(7) = 6569        ...               TAB(1000) = 8268
 NIVEAUPILE = 0
 PILE(1) = ?          PILE(2) = ?          PILE(3) = ?                            PILE(4) = ?
 PILE(5) = ?          PILE(6) = ?          PILE(7) = ?                            PILE(30) = ?
BLOCK> WAS ACTIVATED FROM (MAIN). NEAR COORDINATE0013
```

Рис. VIII.6. Использование оператора *ASSERT* в АЛГОЛе W и фрагмент трассировки.

Когда программируют на языке АЛГОЛ W и начинают использовать такие операторы *ASSERT* (вначале для очистки совести), вызывает удивление полезность этого оператора и число ошибок, которые он позволяет немедленно «фильтровать», – обычно простые и «глупые» ошибки, но только такие и остаются или должны оставаться, если пытаться программировать внимательно.

Надо надеяться, что в будущем все языки будут содержать подобный оператор.

(*Упражнение:* обсудите, не ожидая разд. VIII.4.7, довод: «это уменьшает эффективность».)

VIII.4.2.5. Надо ли тестировать программы?

Надо ли тестировать программы? Многим программистам подобный вопрос кажется абсурдным: программу не пускают в эксплуатацию, пока ее не подвергнут некоторому числу «холостых» выполнений, используя данные, для которых известен результат, чтобы проверить, что получается именно этот результат, – и, если число испытаний достаточно, а случаи выбраны представительными, считается убедительным, что и во всех других случаях все будет хорошо.

Сам принцип тестов, однако, живо оспаривался [Дейкстра 72], [Дейкстра 72a]. Действительному метода тестов есть вызывающий беспокойство недостаток: поскольку число возможных данных для алгоритма вообще бесконечно, никакая серия тестов не сможет никогда гарантировать правильность алгоритма; даже если множество данных конечно, его размеры настолько велики во всех реальных случаях, что было бы напрасным надеяться выполнить исчерпывающий тест. Классическим примером служит программа, выполняющая умножение двух целых чисел (если предположить, что эта операция действительно требует программу): если целые числа представлены 32 битами, имеется $2^{32} \times 2^{32} = 2^{64} \approx 2 \times 10^{19}$ возможных случаев, и тогда, если выполнять одно умножение в секунду (!), понадобится около 10^{14} лет для верификации тривиальной программы. Другими словами, в соответствии с формулой Дейкстры *«тесты могут служить для демонстрации наличия ошибок, но не их отсутствия».* Дейкстра из этого делает вывод о необходимости доказательств правильности программ и о «бесполезности» тестов.

> Эти замечания Дейкстры важны: они должны полностью дискредитировать определенную небрежность в обычном программировании, основанную на идее, что если программа содержит несколько ошибок, то их можно обнаружить и исправить при первом испытании. Эта позиция прямо ведет к таким катастрофическим ситуациям, как те, что были описаны выше, когда время «жизни» программы состоит из одного короткого периода первоначальной отладки» и непрерывно поддерживаемого «технического обслуживания», которое может длиться годами. Ясно, что продукция программирования может быть приемлема лишь в том случае, когда программирование понимается как систематическая деятельность, а программист одновременно с программой строит если не доказательство, то по крайней мере убедительные доводы в пользу ее правильности. Когда же программа уже написана, слишком поздно исправлять в ней идейные ошибки.

Достаточно ли в соответствии с этими нападками полностью исключить тесты из практики информатики? Это было бы не серьезно. Никто не выпустил бы, например, программу управления движением поездов, не протестировав ее интенсивно с помощью моделирования. Более чем кто–либо другой, программист на опыте знает, что человеческий ум не является непогрешимым и подвержен самовнушению. С другой стороны нет оснований априори предполагать, что даже самое строгое доказательство имеет меньшую вероятность содержать ошибки, чем программа. Действительно, изучение языка Z (VIII.3.5) нам подсказывает, что доказательство можно рассматривать как программу определенного уровня, подверженную тем же ошибкам, что и другие уровни. Таким образом, важно не пренебрегать никаким методом, который помогает гарантировать правильность программы. И если тесты рассматривать с этой точки зрения, то им сразу найдется настоящее место: они могут рассматриваться как вспомогательная техника, помогающая находить ошибки, которые могли бы пройти незамеченными на предыдущих стадиях отладки.

Другой важный вывод, который вытекает из этого размышления о тестах,

состоит в том, что испытания, выполненные для оценки правильности программы, не должны быть произвольными, если надо получить ценные сведения. В частности, не стоит ожидать многого от тестов программы, если они не были предусмотрены при написании программы: если тесты присутствуют, они являются предусмотренным заранее и запланированным этапом разработки, а не просто дополнительной верификацией программы, которая была задумана без учета этого ограничения. Заметим, что построение представительных «отладочных наборов» для данной программы–задача нетривиальная: надо выбрать множество случаев, позволяющее протестировать если не каждую ветвь программы (так как число возможных случаев бесконечно, когда программа содержит неопределенные циклы), то хотя бы представительное множество. По этим вопросам можно обратиться, например, к статьям [IEEE 76]. С практической точки зрения заметим, что в большой группе программистов необходимо поручать тестирование элемента программы не тем программистам, которые его писали, а другим. Ведь порой программисты при написании программы имеют в виду более или менее сознательно не обязательно самый общий случай обработки и тестируют лишь этот частный случай.

Стоит также заметить, что тесты могут служить важным средством оценки эффективности программы в тех случаях, когда математический анализ ее сложности оказывается слишком трудным. В этом случае отладочные наборы выбираются в соответствии с другими критериями, чем при проверке правильности программы. Берется множество «средних» отладочных наборов в соответствии с некоторым вероятностным смыслом (оценка средней практической сложности) либо отладочный набор, требующий как можно более медленного выполнения или максимального использования памяти (сложность самого нежелательного случая). Отметим, однако, что в этом случае следует учитывать и оптимизировать общие затраты, т.е. затраты выполнения с реальными данными *и* тестами!

VIII.4.2.6. Средства диагностики при выполнении

Как мы уже сказали, всякому программисту легко себя убедить, что «человеку свойственно ошибаться» и что ошибки происходят часто. Смирившись с таким положением вещей – что может причинить серьезный ущерб его самолюбию, – программист имеет право ожидать от используемой им системы, что **она поможет** ему **найти** допущенные **ошибки,** понять их происхождение и устранить их. Под «системой» здесь следует понимать множество средств, к которым программист имеет прямой или непрямой доступ, – используемую им «виртуальную машину», состоящую из технических средств, операционной системы ЭВМ, транслятора и «системы исполнения», которая, если она существует, осуществляет контроль за ходом программы пользователей.

Следует пожалеть, что среди широко распространенных систем существует очень мало таких, которые в этом смысле имеют приличные возможности. Таким образом, настоящий раздел посвящается в основном сетованиям и сожалениям.

А причин для сетований очень много. Через тридцать лет после появления вычислительных машин большинство пользователей могут в случае анормальной ситуации, обнаруженной при выполнении (переполнение как результат арифметической операции; переполнение выделенной в памяти области, являющееся, например, следствием недопустимой индексации в массиве; превышением времени, необходимого для вычислений, и т.п.), рассчитывать лишь на более или менее двусмысленную диагностику, выданную операционной системой – разумеется, на английском языке, – возможно снабженную этим странным объектом, называемым на жаргоне *дампом:* несколько килограммов бумаги, покрытой цифрами в восьмеричной системе или, что еще хуже, в шестнадцатеричной. Впрочем, в большинстве систем это диагностическое средство не всегда отлажено, как бы примитивно оно ни было; например, немногие из распространенных систем позволяют при выполнении систематически тестировать соответствие индексов массивов, так что серьезная

ошибка может долго оставаться незамеченной, а при этом программа дает, разумеется, неправильные результаты. Все это отражает типичную, упомянутую в начале этой главы ситуацию в информатике–отставание программного обеспечения от технического. Попытка исправить с помощью «дампа» и выданного операционной системой шестнадцатеричного адреса программу, написанную на языке высокого уровня, равносильна, если перенестись в другую область человеческой деятельности, постройке сверхзвукового самолета в кузнице.

Прекратим наши жалобы и перейдем к пожеланиям. Ни одна система программирования (транслятор + система исполнения + операционная система) не должна, по нашему мнению, получать распространение, если она не может предложить пользователю перечисляемые ниже возможности[1]:

a) в случае ошибки, обнаруженной при выполнении, система должна выдавать **ясную диагностику** (точнее, на языке пользователя), указывающую тип происшедшей ошибки;

b) все диагностические сообщения должны быть сформулированы в терминах исходной программы (операторы, переменные, массивы...) без привязки к понятиям технического обеспечения (адрес в памяти, физический номер периферийного устройства, содержимое слова...);

c) программист должен уметь с помощью специальных средств системы получать в случае ошибки список активных в момент ошибки программных объектов и их значения, выраженные, разумеется, во внешнем формате (целые, тексты и т.д.);

d) другое необходимое средство, разумеется являющееся дополнительным в системе, но действующее не только когда ошибка уже произошла, это **след выполнения,** или прокрутка, печатающийся в конце выполнения программы и указывающий, сколько раз каждая группа операторов выполнялась. На Рис. VIII.6 (стр. 196) показан пример такого «следа», выданного системой АЛГОЛ W. Здесь группы операторов, для которых число выполнений вынесено на поля, соответствуют блокам программы, «сверстанной» системой выполнения в соответствии с правилами смещений, близкими тем, которые приняты для представления программы в этой книге. В примере на Рис. VIII.6, соответствующем случаю ошибки, указывается место этой ошибки.

Такая прокрутка является очень важной информацией для улучшения программ. Фрагмент программы, выполненный 0 раз, немедленно привлекает внимание; он часто соответствует (но, конечно, не всегда) незамеченной ошибке программирования. Кроме того, прокрутка является важным средством улучшения программы, указывающим на наиболее значительные части программы (внутренние циклы). Если вы заранее не проводили математического исследования алгоритма, то с удивлением обнаружите, изучая прокрутку, что наиболее часто выполняемые фрагменты программы не обязательно те, которых вы ожидали. Дополнительным средством является след выполнения, дающий не только число выполнений, но также время выполнения каждого фрагмента программы; см. [Инглз 71] для систем такого типа, применяемых на ФОРТРАНе. Перечисленные средства были описаны в предположении, что ЭВМ работает в

[1] Мы обсуждаем здесь лишь ошибки, обнаруженные при выполнении. Само собой разумеется, что всякий хороший транслятор должен давать ясную и явную диагностику ошибок, обнаруживаемых на стадии трансляции, а также полезные для программиста сведения: таблицу символов, таблицу перекрестных ссылок, бесполезные операторы или переменные и т.д. На деле большинство трансляторов знают о программе больше, чем они говорят программисту [Хорнинг 75], [Касьянов 78]; это, в частности, случай трансляторов, называемых «оптимизирующими» [Кок 70], которые проводят чрезвычайно глубокий анализ программы, что позволяет им обнаружить нелогичности, такие, как наличие переменной, значение которой может быть использовано прежде, чем будет установлено его начальное значение, объявленные, но не использованные переменные, недоступные части программы и т.д.

«пакетном» режиме, когда пользователь не может вмешиваться в ход своей программы. Ясно, что их использование может быть легко приспособлено к режиму диалога, когда пользователь может при обнаружении ошибки исправить ее при помощи «текстового редактора»–программы управления исходными программами, ориентированной преимущественно на конкретный язык и систему (транслятор, система исполнения).

> Заметим, что все описанные средства имеются (за исключением измерения процессорного времени) в системе АЛГОЛ W для ИБМ 360/370 и совместимых с ними машин [Саттеруэйт 72]. Показательно, что этот транслятор предназначался в основном для учебных целей, так как «профессионалы» считаются слишком серьезными людьми, чтобы тратить время на такие смешные предосторожности. Немного перефразировав метафору Хоара [Хоар 73], можно сказать, что все происходит так, как если бы на учебных самолетах была бортовая радиоаппаратура, а на рейсовых нет!

Против использования этих средств можно выдвинуть возражение, что они требуют дополнительного процессорного времени. [Саттеруэйт 72] указывает, что в системе АЛГОЛ W перерасход времени и места был сведен к совершенно терпимому уровню. Однако в случае сильно «оптимизирующего» транслятора потери могут быть более серьезными: некоторые оптимизации несовместимы с операциями, необходимыми для сохранения эффективного контроля за выполнением программы. Можно, однако, предложить два серьезных ответа на вопрос об эффективности:

a) Управляющая система реализуется только в виде вспомогательной версии; она, следовательно, может предназначаться для отладки программ, «эксплуатация» которых обеспечивается оптимизирующими трансляторами. Заметьте, что в этом есть отмечавшийся уже парадокс: как только вещи усложняются, мы уничтожаем всякий контроль, и ... была не была!

b) Стоит еще раз напомнить, что проблема эффективности имеет смысл лишь в глобальном масштабе, в применении ко всему процессу программирования: к программному обеспечению (т.е. человеческий фактор) и техническому обеспечению. Поиски гипотетического выигрыша 10% процессорного времени абсурдны, если они приводят к процессу, когда программист–который может быть инженером или специалистом высокого уровня–будет проводить часы над шестнадцатеричным дампом в поисках тривиальной ошибки. Стоимость секунды центрального процессора имеет смысл лишь в сравнении с часовой зарплатой программиста.

VIII.4.3. Читаемость, выражение, стиль, комментарии, документация

VIII.4.3.1. Читаемость программ

Это очевидная истина – больше, конечно, для тех, кто пишет о программировании, чем для тех, кто пишет программы, – что программа значительно реже *пишется,* чем *читается.* Тот, кто пробовал погрузиться в программу, написанную кем–нибудь другим – даже им самим несколько месяцев тому назад, – если только этой проблеме с самого начала не было уделено особое внимание, знает на собственном опыте, насколько часто повторное чтение программы превращается в такое испытание, что порой, если известны внешние спецификации, предпочтительнее переписать эту программу с нуля, чем пытаться ее понять с целью сделать небольшие изменения.

Мы не будем долго останавливаться на этом качестве не потому, что оно не является капитальным, а потому, что трудно сформулировать объективные критерии читаемости. Единственный из этих критериев, который нам представляется абсолютно общим, – это избыточность: легко читаемая программа позволяет читателю ее понять и, что так же важно, убедить себя в том, что он ее понял, придав ей многочисленные сопутствующие знаки (операторы, комментарии, утверждения, спецификации и т.п.).

Эта идея избыточности лежит в основе понятия языка программирования высокого уровня: ничто не бывает менее избыточным, чем программа на машинном языке, где каждый знак (каждый бит) играет одинаковую роль, неся информацию, и только ее.

> Не так–то легко, как это кажется, добиться повсеместного признания принципа избыточности: естественной тенденцией всех программистов является стремление сделать программу возможно более компактной и «непрозрачной» из–за своего рода профессиональной гордости, которая выливается порой в своеобразные соревнования («задача о восьми ферзях из 7 операторов и 35 литер»). Некоторые аспекты языка АПЛ являются иллюстрацией этой позиции. Поставить крест на этом не так–то легко: эта позиция соответствует реальной психологической и социальной проблеме–потребности программиста утвердить свою техническую квалификацию перед другими и перед самим собой. Добиться признания своей квалификации порой трудно, поскольку программирование на ЭВМ кажется со стороны такой простой задачей–особенно, если после недели освоения БЭЙСИКа вычисляют четырнадцатый десятичный знак числа *e*! Именно для признания квалификации программистов необходимо установить некоторые профессиональные нормы, касающиеся свойств программ, в частности их читаемости.

Чтобы установить такие нормы, простого утверждения о необходимости избыточности, к сожалению, недостаточно. Действительно, не всякая форма избыточности одинаково желательна–некоторые формы могут оказаться пагубными. Пусть дана подпрограмма на ФОРТРАНе

```
      SUBROUTINE ERNEST (A, B, M, N)
```

с объявлениями

```
      INTEGER M, N
      REAL A (M, N), B (M, N)
```

Чтобы переписать в *A* значения элементов *B* в теле подпрограммы,
запишем (учитывая случай, когда массивы пусты):

```
      IF (M.LE.0.OR.N.LE.0)GOTO 1000
            DO 200 I = 1, M
                  DO 100 J = 1, N
                        A(I, J) = B(I, J)
100                     CONTINUE
200               CONTINUE
1000  продолжение программы
```

На ПЛ/1 или на АПЛ мы бы просто написали (с точностью до синтаксических различий)

A ← *B*

Можно спросить, какая из двух форм более ясная; не затеняют ли циклы и проверки, которые содержатся в первой из них, тот факт, что речь идет всего лишь о присваивании массива? Не превращаются ли циклы в слишком общую структуру для представления глобальной обработки массива тем фактом, что границы *M* и *N* могут быть заменены произвольным значением (не говоря о возможности ошибки)?

Легко видеть, что избыточность не является простым критерием, и надо уметь ее использовать с достаточным основанием. В частности, ключом к читаемости программы являются не нагромождения объяснений (комментарии, документация), которые рассматриваются далее, а сама программа.

VIII.4.3.2. Стиль и выражение

Надо найти, таким образом, как можно более простые и естественные **формы выражения,** позволяющие читателю установить непосредственное соответствие между *текстом* программы и тем, что происходит при ее *выполнении.* Всякий программист должен сосредоточить свое внимание на этом основном свойстве программы.

Одним из ключевых моментов стиля и выражения программы является, очевидно, знаковая система, в которой она представлена, – язык программирования. Ясно, что выбор языка является решающей проблемой, а читаемость–далеко не единственный критерий; действительно, приходится выбирать всю «систему программирования» целиком, и решение зависит от качеств транслятора, простоты его использования, возможностей его диагностики, его связей с операционной системой управления исполнением (VIII.4.2.6), «переносимости» и т.п. Не обсуждая сейчас важный вопрос сравнительной оценки языков программирования, напомним, однако, что решающим критерием, связанным с избыточностью, является возможность *статического контроля.* Правила языка должны позволять обнаружить ошибки как можно быстрее и, в частности, при *трансляции,* а не при *выполнении.* Этот критерий дает преимущество языкам, где понятие типа играет важную роль. В частности, мы видели, насколько соглашение в ФОРТРАНе или ПЛ/1, по которому идентификатор не обязательно должен быть объявлен, может оказаться опасным. В противоположность тому, что происходит в АЛГОЛе, ПАСКАЛе или СИМУЛЕ, неправильная орфография идентификатора на ФОРТРАНе или ПЛ/1 может остаться незамеченной при компиляции. Однако ошибки тем труднее понять, чем позднее в процессе написания и отладки программы они обнаружены.

Очевидно, что оценка стиля программы является очень субъективной. Однако можно привести важные и трудно оспариваемые принципы, которые, надеемся, для читателя будут лишь напоминанием:

a) **управляющие структуры**. Самый трудный элемент при чтении программы это, без сомнения, статическое понимание динамической последовательности операторов. Поэтому исключим анархистские операторы перехода типа *GOTO,* которые мешают последовательному чтению программы снизу вверх и слева направо; никто не может одновременно идти двумя путями и тем более четырьмя, восемью, шестнадцатью, ... Ограничимся небольшим числом основных структур, таких, как структуры гл. III; сделаем это, даже если используемый язык программирования более примитивен – в этом случае потребуется «перевод», который, разумеется, выявит (снабдив их комментариями) структуры низкого уровня, такие, как *GOTO*, но совсем в ином контексте (ср. программы на ФОРТРАНе, разд. VII.3);

b) **использование обозначения высокого уровня**. Последнее замечание позволяет подчеркнуть, что обычные языки программирования вообще не являются достаточными средствами выражения алгоритмов. Это очевидно в случае ФОРТРАНа и КОБОЛа, например, но верно также и для ПЛ/1, АЛГОЛа и т.д. Все эти языки слишком ориентированы на связь с машиной и слишком подвержены влиянию оправданных или неоправданных исторических решений, чтобы не создавать препятствий свободному и естественному выражению алгоритмов – препятствий, к которым в большинстве языков добавляется еще то, что они основаны на английском языке. Абсурдно и опасно программировать непосредственно на ФОРТРАНе, например; предпочтительно было бы использовать нотацию высокого уровня, близкую к французскому (русскому. – *Перев.*) языку, типа той, что использована в этой книге. Такая нотация весьма трудна при отладке; ее следует принимать для «человеческого» общения, она должна

быть ясной и естественной, но в то же время и достаточно строгой, чтобы ее без особого труда можно было перевести на существующие языки программирования;

c) **выбор имен.** Программа составлена из заренее определенных символов (операторов, зарезервированных имен и т.д.) и имен, выбранных программистом (имен переменных, подпрограмм, меток и т.д.).

Они–то и дают программисту один из основных способов понять, что происходит. Поэтому их надо выбирать особенно внимательно. Запомните принцип–надо рассматривать имя как *инвариант* программы, имея в виду доказательства правильности: называть переменную *МАКСИМУМ–МАССИВА* действительно является способом утверждения инвариантности свойства «значение переменной *МАКСИМУМ–МАССИВА* является самым большим в рассматриваемом массиве» (при условии что не возникнет двусмысленностей в идентификации массива).

Можно оспаривать это правило, заметив, что оно опасно: имя переменной не оказывает никакого влияния на ход программы. Программа, в которой все вхождения идентификатора *X* заменены на *Y* (в предположении, что это не вызовет конфликта с именем какого–нибудь другого объекта программы), остается строго неизменной по своей семантике и будет идентичным образом обработана компилятором. Программа может быть неправильной, хотя, судя по идентификаторам, можно считать, что она корректна, что может ввести программиста в заблуждение. Например, вследствие ошибки, приведшей к замене знака операции «>» на «<», переменная *МАКСИМУМ–МАССИВА* могла бы в действительности обозначать минимум, и было бы трудно обнаружить ошибку из–за особенно выразительных имен. К этому примыкает поставленная Парнасом проблема имен модулей (VIII.3.4), и с такой же проблемой мы сталкиваемся в связи с комментариями. Однако нам не кажется, что эта опасность серьезно противостоит изложенным выше принципам. Чтобы в этом убедиться, достаточно рассмотреть результат одного из следующих опытов:

- взять любую нетривиальную программу (например, в гл. VII) и заменить все идентификаторы выбранными случайным образом именами: u73t, abzef5, zzxy549u, ...;

- поменять в программе Реорганизация (из программы Древесной Сортировки, VII.3.7) идентификаторы «правый» и «левый» и соответственно «сын» и «отец».

Конечно, не все языки предоставляют всегда очень широкий выбор для идентификаторов, и 6 литер ФОРТРАНа являются серьезным препятствием (не говоря уже о соглашениях в БЭЙСИКе). Это ограничение не служит серьезным оправданием для распространения в программах таких имен, как *ЕРУНДА, ШТУЧКА* и т.д., или использование *A, AA, AAA, AAAA* – подлинно мазохистская практика, поскольку очень возможна опечатка или ошибка при написании. С этой точки зрения заметим, что, если используются похожие имена, психологически более надежно делать их различающимися в начале, а не в конце, т.е. писать *XПРОЕКЦИЯ* и *YПРОЕКЦИЯ* а не *ПРОЕКЦИЯХ* и *ПРОЕКЦИЯY*; надо также вспомнить, что в таких языках, как ФОРТРАН (и в меньшей мере в ПЛ/1**),** которые не требуют явного объявления переменных, создаваемые чаще всего трансляторами списки переменных или массивы перекрестных ссылок являются важным средством обнаружения ошибок при редактировании программ.

d) **расположение программ.** Все языки высокого уровня разрешают программисту пользоваться незначащими пробелами. Это свойство особенно полезно, чтобы показать динамическую структуру программы. В нашей книге систематически использовались *смещения* для выделения управляющих структур и их размещения:

```
A;
WHILE B DO
        BEGIN
        C;
        D;
        IF E THEN
            BEGIN
            F;
            FOR G : = H UNTIL I DO
                    J;
            K;
            IF L THEN
                    M
            END
            ELSE
                    N;
            O;
            WHILE P DO
                    BEGIN
                    Q;
            IF R THEN
                S;
            T;
            U
            END;
        V
        END;
```

Рекомендуется всегда располагать программы с помощью этого метода или эквивалентного соглашения; это особенно важно в языках, где структура не следует непосредственно из синтаксиса:

```
C       /ПОКА А ПОВТОРЯТЬ/
100     IF (.NOT.A) GOTO 200
            опер 1
            опер 2
C           /ЕСЛИ В ТО/
            IF (.NOT.B) GOTO 150
                опер 3
                опер 4
                GO TO 170
C           /ИНАЧЕ/
150             опер 5
                опер 6
                опер 7
170         CONTINUE
            опер 8
            опер 9
            GOTO 100
```

```
C
200   CONTINUE
      опер 10
      опер 11
```

И однако, сколько программ на ФОРТРАНе предстает перед теми, кто их читает, идеально выровненными «по 7–й колонке», полностью маскируя использованные управляющие структуры!

е) **объем программных модулей.** Все языки программирования позволяют разбивать программы на элементарные единицы, называемые «подпрограммами», «функциями», «процедурами» и т.д. В зависимости от ситуации программист имеет более или менее богатые возможности. В АЛГОЛе W или в ПЛ/1, например, он может объявлять подпрограммы внутри блока или другой подпрограммы, совершая иерархию декомпозиций; в ФОРТРАНе и КОБОЛе это не так, и все программные модули имеют одинаковый уровень. Каков бы ни был используемый язык, программный модуль должен оставаться тем не менее легко понимаемым объектом. Можно сформулировать следующее правило: *всякий программный модуль, содержащий более десятка имен* (параметров, локальных переменных, массивов и т.д.) *либо более двадцати строк выполняемой программы* (учитывая только операторы в строгом смысле, за исключением объявлений, комментариев и т.д.), *по–видимому, слишком сложен и должен быть исследован с целью лучшей декомпозиции.* Причина этого правила проста: нормальный человек не может без значительного умственного усилия понять программный модуль, превосходящий (приблизительно) указанные нормы.

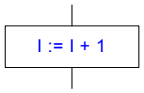
Все эти правила и многие другие, которые, надо полагать, читатель может сформулировать и сам, можно найти в книге [Керниган 74], которая может служить «учебником стиля» в программировании. Заметим, впрочем, что использование слова «стиль» в этом разделе и в упомянутой книге свидетельствует о пока еще рудиментарном состоянии современного программирования: приведенные правила настолько просты и соответствуют здравому смыслу, что они в большей степени отвечают на вопрос «как выразить алгоритм?», чем на более сложный–«как выразить его наилучшим образом?», а ведь только в этом случае было бы оправдано слово «стиль».

VIII.4.3.3. Комментарии

«Комментируйте ваши программы», – говорят теперь программистам. Рекомендация похвальна и будет одобрена любым, кому приходилось бы погружаться в программу на ФОРТРАНе из 8000 операторов *без единого комментария.* К несчастью, здесь тоже мало одних благих намерений, и не все комментарии одинаково полезны. Многие комментарии представляют собой всего лишь «шум» и не несут никакой информации, как в следующем примере (синтаксис ПЛ/1):

```
/* ПРИБАВИТЬ 1 К ПЕРЕМЕННОЙ I */
I = I + 1;
```

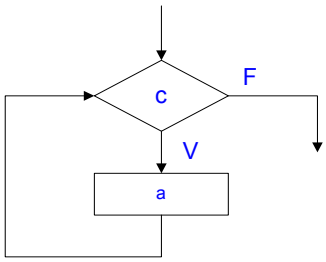
Такие нередкие, впрочем, комментарии бесполезны и могут служить помехой, так как они затемняют программу в ущерб важным и тонким элементам. Некоторые специальные журналы регулярно публикуют программы, представленные с помощью блок–схем (мы забегаем вперед и обращаемся к вопросу о документации), а затем в виде эквивалентных программ на ПЛ/1; переход от блок–схем к программам состоит как раз в основном в замене элементов типа



операторами

I = I + 1;

Можно усомниться в полезности первой формы (в которой к тому же использовано обозначение, отличающееся от принятого). Точно так же можно спросить себя, необходимо ли рисовать



чтобы растолковать

```
DO WHILE c;
    a;
    END;
```

(где c – данное условие, а a – данное действие). Первая форма здесь имеет менее высокий уровень, чем вторая, и скорее затемняет, чем поясняет.

Бывают бесполезные комментарии; могут существовать и явно вредные, в частности, если они неверны, когда противоречат программе, как в следующем примере, приведенном в книге [Керниган 74]:

```
/* ПРОВЕРИТЬ ЯВЛЯЕТСЯ ЛИ ЧИСЛО НЕЧЕТНЫМ*/
IF MOD (X, 2) = 0 THEN
        DO;
        SOMME = SOMME + X;
        COMPTEPAIRS = COMPTEPAIRS + 1;
        END;
```

Здесь программа обрабатывает четные числа в противоположность тому, что говорит комментарий. Читатель попадает в неудобное положение: либо он верит комментарию, не проверяя его соответствия программе, и будет введен в заблуждение относительно поведения программы; либо он обнаруживает несогласованность, и в этом случае комментарий бесполезен, так как пришлось подробно Прочитать программу; неясно, какая из этих возможностей лучше.

Бывают, следовательно, плохие комментарии. А существуют ли хорошие? Конечно, но и в этом случае не следует давать общих правил. Особенно необходимыми представляются три типа комментариев:

a) комментарии, указывающие на сложный фрагмент программы, на использование нетривиальной техники программирования (комментарий в этом случае может содержать библиографическую ссылку, если эта техника слишком сложна, чтобы ее излагать в теле программы). Обязательно нужно указывать на нарушения нормы языка программирования, на элементы программы, в которых используется особенность машины или операционной системы, а также «уловки» программирования, которые программист обязательно хочет сохранить. В некоторых математических работах подобным же образом используется знак *S* (опасный поворот), чтобы отметить на полях деликатные пассажи;

b) комментарии также очень полезны для выражения утверждений,

доказываемых в определенных местах программы. Мы видели, однако, что может оказаться полезным превратить эти утверждения в управляющие операторы (*ASSERT*). Для так называемых «входных» и «выходных» утверждений подпрограмм комментарии особенно желательны; их используют также, чтобы описать роль различных формальных параметров и для указания, какие параметры являются аргументами, какие результатами, какие **модифицированными данными,** если таких указаний не требует синтаксис языка;

c) при современной концепции программирования, основанной на поэтапном составлении программы (будь то уровни абстракции в нисходящем программировании, или статические и динамические аспекты в Z, или просто последовательные версии в результате преобразований программы), комментарии являются идеальным средством выражения, позволяющим в окончательной программе отразить эти этапы. Например, при нисходящем подходе элемент программы, полученный на уровне *n,* является уточнением виртуального оператора уровня $n - 1$; поэтому естественно записывать его как параграф, заголовок которого, оформленный комментарием, как раз является этим виртуальным оператором. В качестве примера не полностью развернутая версия *Быстрой Сортировки* может иметь вид

```
{рекурсивная сортировка массива a[i : j]}
если j > i то
        | разделить массив индексом s;
        | Быстрая Сортировка (a[i : s – 1]);
        | Быстрая Сортировка (a[s + 1 : j])
```

На следующем уровне, предположив, что мы уточнили виртуальный оператор разделить, получаем

```
{рекурсивная сортировка массива a[i : j]}
если j > i то
        | {разделить массив индексом s}
        | u ← i; v ← j;
        | пока u ≠ v повторять
        |       | …
        |       | {ср. VII.3.6}
        | s.← u;
        | Быстрая Сортировка (a[i : s – 1]);
        | Быстрая Сортировка (a[s + 1 : j])
```

Точно так же, если используется язык Z, то «статические» версии, естественно, предлагают свои определения множеств, функций и отношений в качестве комментариев для последующих версий. Если опираться на элементы доказательства, то всякий зачаток довода в пользу правильности, как бы ни был он нестрог, всякий инвариант цикла даст полезные и эффективные комментарии.

Общей чертой всех таких комментариев является мысль, что они не должны служить украшением программы, добавленным в нее искусственно, а быть составными частями программ, задуманными и построенными одновременно с этими программами. И только таким образом они становятся элементами, привносящими значащую информацию, только так можно избежать бесполезных и противоречивых комментариев.

Эта же идея должна применяться к инструменту, обобщающему понятие комментария, – к документации.

VIII.4.3.4. Документация

Документация программы состоит из всей информации, которая позволяет понять и использовать эту программу. Обычно различают внешнюю документацию, предназначенную для лиц, желающих пользоваться программой, не вникая в ее подробную организацию, и внутреннюю документацию, предназначенную для тех, в чью обязанность входит ее внутреннее понимание, например, в целях ее отладки на конкретной машине и в конкретной операционной системе, транслировать ее для другой машины или модифицировать.

Предлагались многочисленные методы документации; создавались, в частности, различные «средства», позволяющие решить все проблемы документации благодаря многоцветным формулярам, частично заполненным заранее, линейке из пластмассы с трафаретами, чтобы вычерчивать схемы, и т.д. В наши намерения не входит критика этих изделий, появление которых свидетельствует об осознании необходимости в документации. Кроме того, немного опыта, а также привычка к языкам программирования и к операционным системам или просто к программам, доступным окольным путем через призрачную или бредовую документацию, дают возможность оценить любые усилия в этом направлении. Однако сомнительно, чтобы проблемы документации могли быть разрешены изобретением графического носителя, каким бы элегантным он ни был.

Мы не предлагаем теории документации. И все же следующие правила представляются достойными размышления:

a) всякая программа, которая предназначена для использования несколько раз или несколькими лицами, должна иметь документацию (*или* в этой фразе не является исключающим);

b) документация должна составляться одновременно с программой; в противном случае она *никогда* не будет написана. Это правило каждый день заново подтверждается на опыте: невозможно найти руководителя проекта, который предоставил бы шесть месяцев отсрочки одному из своих программистов для составления документации после того, как разработка программы была успешно закончена. Впрочем, можно быть еще более категоричным и заметить, что было бы полезным писать документацию до написания программы;

c) документация – это трудная задача, требующая особых качеств и времени. В любом проекте программирования, выходящем за рамки обычного, группа программистов должна включать специалистов в этой области, хотя и не должна состоять исключительно из них. Мы вернемся к этому вопросу в разд. VIII.5.1;

d) внешняя документация особенно важна, поскольку она дает возможность пользоваться программой; она должна быть действительно внешней, т.е. не содержать явных или неявных намеков на функционирование программы. Это *спецификация,* и ясно, что упор надо делать на формализмы спецификаций алгоритмов, таких, как Z_0, хотя, без сомнения, желательны формы выражения, более близкие к обычному языку;

e) нам кажется, что внутренняя документация выигрывает, если она, насколько возможно, представлена внутренним для программы образом–либо непосредственно элементами программы, либо комментариями. Проблема в том, что, столкнувшись с многочисленной информацией о программе–собственно программой с комментариями, средствами отладки, блок–схемами, объяснительными замечаниями, инструментами отладки этих замечаний и т.д., – мы не знаем, как быть, какому описанию доверять. Если вся эта информация собрана вокруг того элемента программы, к которому она

относится, то это меньше дезориентирует. Наверное, идеала достичь невозможно, даже если бы программа являлась своей собственной документацией, но современные языки программирования для этого не приспособлены так же, как и некоторые специальные форматы для комментариев.

f) важным моментом является то, что документация всегда должна иметь более *высокий уровень абстракции,* чем объекты (программа, данные), к которым она применяется. Это полностью исключает подробные блок–схемы, сопровождающие написанные на развитых языках программы, но это не касается функциональных схем, предназначенных для графического изображения отношений между уровнями абстракции, между модулями и т.д.

VIII.4.4. Надежность; защищающее программирование

VIII.4.4.1. Надежность

Надежность является важным и естественным требованием пользователя программы. Ее определение очевидно в некоторых областях применения, таких, как управление процессами, операционные системы и т.д., где физические результаты действия программы непосредственно измеримы извне. Однако понятие надежности существует для всех типов программ: самым общим образом ее можно определить как *способность программы давать разумные результаты во всех возможных окружениях, и в частности в анормальных условиях.*

Самый простой пример «анормальных» условий–ситуация, в которой программе дают необычные данные (ср. склоны Гималаев); другой пример – случай ошибочных данных, которые должны быть выявлены и отброшены, чтобы не привести к катастрофе.

Как и многие рекомендации этой главы, эти замечания могут показаться очевидными. Однако весьма часто, открывая свою ежедневную газету, мы находим там сообщения такого рода: «Вследствие ошибки ЭВМ двести пенсионеров не получили свою пенсию – женщина–инвалид попыталась покончить с собой» или же: «номер телефона был занесен в ячейку «сумма вклада» – клиент был ошибочно обвинен в выдаче необеспеченного чека». Обычно такие ошибки поспешно приписывают ЭВМ: «Вычислительная машина ошиблась». Публика остается этим удовлетворенной; не иссяк еще мифический ореол вычислительных машин. Однако такая доверчивость не будет вечной, и со стороны специалистов по информатике было бы весьма уместно задать такие вопросы: как могла сложиться такая ситуация? были ли учтены все возможные случаи? достаточно ли хорошо была изучена программа, прежде чем ее стали распространять? проводила ли программа систематический контроль правильности своих данных? их правдоподобности? и т.д. Разрешение этих проблем приводит естественным образом к старой идее, а именно к защищающему программированию.

VIII.4.4.2. Защищающее программирование

Принципы, развитые в предыдущих разделах, основаны на одной общей идее, которую можно резюмировать так: ошибка–это реальное явление и лучше попытаться с ней примириться. Перед лицом такой печальной ситуации естественно сделать программирование «защищающим»–идея состоит в том, чтобы предвидеть в программе не идеальное, а такое реальное окружение, в котором все может произойти. В какой–то мере следует возвести в ранг принципа программирования так называемое правило

Мэрфи, по которому «если худшее может случиться, то оно произойдет». Ошибки, против которых пытаются вооружиться, могут быть разного рода: ошибки программ, ошибки данных, к которым надо добавить проблему предельных случаев и возможные ошибки аппаратуры. Природа их самая разнообразная.

Ошибки данных

Против ошибок данных программист априори никогда не имеет никакой гарантии: если программа велика, она поздно или рано будет работать с данными, подготовленными не ее авторами, а посторонними лицами, или (в случае программного модуля, принадлежащего системе, которая содержит много других модулей) с другими программными модулями, когда предосторожности обращения, возможно, не всегда выполняются. Следует предусмотреть эту ситуацию, применив некоторое число правил, продиктованных здравым смыслом:

a) составляя программу, внимательно задуматься, как она будет использоваться: предусмотреть естественные и гибкие форматы ввода данных, ясные и понятные результаты и диагностические сообщения об ошибках; печатать данные по мере того, как они читаются. Сколько программ считывают свои данные в жесткой (колонки перфокарты с 17–й по 22–ю) и не очень логичной форме (на каждой карте целое из 6 литер, действительное – из 7 литер, два целых из 4 и 8 литер соответственно...); сколько из них, чтобы считать n элементов массива t, где n может изменяться от выполнения к выполнению, используют конструкцию типа

```
читать n;
для i от 1 до n повторять
    | читать t[i]
```

заставляющую пользователя пересчитывать свои данные, открывая тем самым двери для любых возможных ошибок, в то время как совсем просто заставить ЭВМ, читая t[i], считать до установленного предела...

b) вспомнить, что программа является механизмом перевода данных в результаты, и, таким образом, задумывать формат ввода как синтаксис некоторого малого (?) языка программирования, применив к нему критерии элегантности, простоты выражения и т.д. Именно по этой причине, а не из–за технического интереса к существующим языкам и внутреннему функционированию компиляторов так необходимо изучение методов трансляции и языков программирования. Деятельность программиста – это в значительной мере изобретение малых языков программирования.

Стоит, однако, указать, что не все программы и подпрограммы могут регулярно проверять правильность всех данных, которые им передаются; даже не упоминая проблемы эффективности, ясно, что такие программы быстро достигли бы такой длины и сложности, которые неизбежно породили бы новые ошибки, несомненно более многочисленные, чем те, которые мы пытаемся выявить. Таким образом, лекарство оказалось бы хуже, чем болезнь. К тому же можно показать, что проблема в общем случае неразрешима в теоретическом плане.

На практике решение о степени контроля над данными и аргументами, передаваемыми подпрограммам, будет результатом компромисса между различными факторами: ясностью (в любом случае выгодно обрабатывать по отдельности ошибки и нормальные случаи, доверяя их, например, специальным подпрограммам), эффективностью, степенью «взаимного недоверия» между различными составными частями системы (и программистами, которые их написали!), степенью серьезности требований надежности окончательной программы и т.д.

Ошибки программы

Стратегия защиты от ошибок программы была уже изложена в VII.4.2.4–5–6 в связи с рассмотрением средств контроля при выполнении, и в частности оператора *ASSERT*. Если эти средства,.не существуют в системе, программист всегда более или менее успешно сможет их добавить. Ограничимся здесь тем, что напомним основную идею: речь идет о том, чтобы сделать как можно более явными те мысленные предположения, которые предшествуют написанию программы, и превратить их затем в динамические проверки. Такую ситуацию представляет, в частности, случай свойств, которым удовлетворяют параметры подпрограммы, когда эти свойства можно систематически контролировать. Например, в программе

программа uuuu : REEL (**аргументы** x, y : REELS)
| {x > 0, y > 0, $y^2 - x^2 \geq 40$}
| ... вычисление uuuu
| {uuuu таково, что ...}

начальное утверждение может дать повод к проверке

```
ASSERT(X > 0) AND (Y>0)
              AND(Y**2– X**2 >= 40)
```

с другой формулировкой в языке, отличном от АЛГОЛа W. Некоторые фортрановские трансляторы позволяют считать, что карты, начинающиеся некоторой специальной литерой, содержат оператор или комментарий в зависимости от варианта трансляции; точно так же в языке АЛГОЛ W оператор *ASSERT* может обрабатываться как *COMMENT,* если принят соответствующий вариант трансляции. В обоих случаях можно, следовательно, выполнять регулярную верификацию .только в момент отладки.

Граничные случаи

«Прочность» программы измеряется наряду с другими способами и по ее способности корректно обрабатывать граничные случаи, которые «нормально» не должны происходить, но в конце концов все же происходят в контексте сложной системы, состоящей из многих взаимодействующих между собой программных модулей.

Самым типичным случаем является подпрограмма, оперирующая с массивом, например, чтобы его отсортировать. Подпрограмма должна уметь корректно обрабатывать случай, когда массив пуст (проверьте с этой точки зрения подпрограммы сортировки гл. VII). Заметим, что в этом случае цикл *DO* на ФОРТРАНе является опасной конструкцией, поскольку, как мы уже видели, цикл

```
DO метка I = 1, N
        ...
CONTINUE
метка
```

всегда будет выполнен по меньшей мере один раз, даже если *N* отрицательно или равно нулю. Это объясняет, почему мы так часто – ср., например, Древесную Сортировку (VII.3.7) – переводили для с помощью эквивалента цикла пока (ветвление и логическое *IF*) с явным увеличением индекса (эта практика предпочтительнее, чем включение проверки перед циклом *DO*). Таким образом, цикл будет эквивалентен пустому оператору, если *N* равно нулю или даже отрицательно.

Близкая проблема связана с распространенной ошибкой, которая приводит к тому, что «ошибаются на 1» на одной из границ цикла; цикл будет систематически выполняться на один раз больше или меньше, чем надо. Можно, однако, заметить, что применение строгих методов, в частности определение каждого цикла с помощью инварианта и соответствующего условия окончания цикла, которые дают конечное утверждение, позволяет избежать этого чрезвычайно неприятного типа ошибок.

Ошибки аппаратуры

К классическим типам ошибок, о которых говорилось выше, следует добавить сравнительно новую категорию, по меньшей мере по сравнению с ситуацией последних двадцати лет – категорию ошибок аппаратуры. С появлением микропроцессоров, малых по размерам и по цене вычислительных машин, у которых возможности обнаружения ошибок в настоящее время весьма ограничены, были построены системы децентрализованных структур, образованных из малых автономных устройств, способных восстановить себя в случае ошибки («помехоустойчивые системы»); при этом не обязательно выдается внешнее предупреждение (в частности, программисту) о том, что происходит. Результатом этой ситуации стала утрата полного и постоянного доверия к абсолютной безгрешности аппаратуры. Программистам, по–видимому, придется привыкнуть к подобной ситуации, против которой трудно себя предохранить.

VIII.4.5. Гибкость. Адаптируемость. Универсальность

Очень мало программ являются застывшими объектами. Для программиста признаком успеха программы служит то, что пользователи снова просят и добиваются расширения работающей программы, с тем чтобы она могла отвечать все более общим спецификациям. Даже если это не так, хорошо известно, что почти всегда, если эксплуатация программы должна продолжаться некоторое время, спецификации подвергнутся изменениям, которые надо отразить в программе.

Модифицировать программу в таких условиях может оказаться трудным делом. Поэтому в плане *«защищающего программирования»,* когда стремятся предупредить возможные неприятности, а не исправлять их, законным становится такой вопрос: существуют ли методы, позволяющие при первоначальном написании программы предусмотреть простоту будущих модификаций?

Обычно предлагаемый ответ носит имя *«модульного программирования».* Это выражение означает функциональное разбиение программы на автономные модули, такое, что «малому» изменению спецификаций соответствует малое изменение программы, локализованное в одном модуле этого разбиения. Выше было дано более ограниченное определение «модульного программирования», единицы, или «модули», которого должны, по нашему мнению, соответствовать фундаментальным структурам данных системы.

Следует, однако, сказать, что эти идеи не дают полного решения проблемы. Первой причиной является то, что трудно сформулировать точные директивы для проведения конкретного разбиения. Даже если вышеприведенный совет–основывать разбиение на распознавании фундаментальных структур данных–кажется нам более проясняющим ситуацию, чем часто расточаемые симпатичные, но туманные рекомендации (использовать «функциональные модули», «логические функции системы» и т.д.), он оставляет еще широкое поле для интуиции в выборе элементов разбиения.

Другая трудность состоит в отсутствии определения того, что такое «нормальное» расширение или модификация. [Дейкстра 76] указывает, что обобщение программы, вычисляющей НОД чисел 28 и 77, может быть либо программой, которая находит НОД двух произвольных целых, либо программой, способной выполнять над 28 и 77 какую–нибудь двоичную операцию – сложение, умножение, НОК, т.д. Конечно, довод здесь доведен до парадокса, но он указывает на реальную проблему: трудно заранее угадать, какими будут наиболее вероятные расширения данной спецификации.

Каков же вывод? Имеются два вида возможных обобщений: те, о которых с начала написания программы можно предположить, что они могут потребоваться, и те, потребность в которых возникает неожиданно. Чтобы предусмотреть первые, уместны все приемы «хорошего» программирования: программы заботливо разбиваются на части, чтобы облегчить предусматриваемые расширения, выделяются главные функции системы и основные структуры данных, чтобы позднейшие модификации оставались

локальными; по возможности пишут немного более общие элементы программ, чем это строго необходимо («восходящий просмотр»), или по крайней мере «оставляется место» для предусматриваемых модификаций.

Для расширений второго типа можно дать очевидные советы (например, такой, как: операторы программ никогда не должны содержать константы в явном виде, а должны использовать только символические константы, по крайней мере если язык этого не запрещает, как в командах *DIMENSION* и *FORMAT* ФОРТРАНа), но не более: не существует рецепта, применение которого гарантировало бы превращение расширений в легкую задачу. Можно лишь вновь подчеркнуть важность таких качеств, как ясность, стиль и документация, которые во всяком случае позволяют легче разобраться в существующей программе, чтобы ее модифицировать.

VIII.4.6. Переносимость

Среди тех качеств, которые делают программу «хорошей», наиболее ценным является ее *переносимость* или *транспортабельность,* т.е. легкость ее адаптации к изменению среды, понимая под «средой» различные компоненты системы программирования, относящиеся к программному и техническому обеспечению: вычислительная машина, периферийные устройства, системы доступа к файлам, операционная система, трансляторы и т.д.

Заметим, что существование и важность этого качества полностью зависят от обстоятельств и определяются современным состоянием информатики, когда различия между машинами, периферийными устройствами, программным обеспечением и т.д... значительны и граничат с анархией. По той же причине гораздо легче проповедовать переносимость, чем давать полезные советы для ее осуществления.

Возможные советы легко перечислить. Если хотят написать переносимую программу–что, очевидно, не всегда имеет место, поскольку некоторые программы предназначаются лишь для единственной среды, – то надо выбрать язык программирования, достигший высокого уровня стандартизации (среди широко распространенных языков можно назвать ФОРТРАН, КОБОЛ, ПАСКАЛЬ), и следует строго соблюдать стандарты языка, запрещая обращения к каким бы то ни было «расширениям» или «улучшениям», реализованным в конкретных трансляторах.

Надо сказать, что все эти предосторожности в большой мере иллюзорные. Многие трансляторы более или менее тайно позволяют себе свободное обращение со стандартами языка; хотя это относится, как правило, к неясным и считающимся незначительными вопросам, такая свобода осложняет любую попытку перемещения. Достаточно раз и навсегда констатировать (и прочувствовать) эффект, который оказывает на большие программы изменение версии того же самого транслятора, на той же самой машине и под управлением той же операционной системы, чтобы убедиться на всю жизнь в остроте проблемы. К тому же вопросы языка программирования составляют в действительности лишь видимую часть айсберга: для всякой программы значительных размеров настоящие трудности начинаются (ср. [Пек 78]) вместе с изменениями операционных систем, языков управления заданиями, терминалов, условий доступа к файлам, систем ввода–вывода, управления памятью, форматов магнитной ленты, способов доступа к вычислительному центру, к информационным сетям...

Таким образом, переносимость на 99% является открытой проблемой. С практической точки зрения программист должен знать, что он обязан что–нибудь изменить в своих программах, чтобы приспособить их к любому изменению среды. Следующее замечание подкрепляет нашу несколько моралистскую точку зрения (которая, хочется надеяться, не очень утомляет читателя); когда с целью произвести небольшое изменение принимаются читать и перечитывать программу, ничто так не ценится, как ее читаемость, стиль, выражение, комментарии, документация и т.д.

VIII.4.7. Эффективность: оптимизация

Среди качеств программы, разумеется, надо упомянуть «эффективность», т.е. оптимальное экономическое использование ресурсов, особенно *времени* (центрального процессора) и *пространства* (прежде всего в оперативной, а затем и во внешней памяти).

Мы приступаем к обсуждению этого качества после изучения остальных по той причине, что ему издавна уделялось преувеличенное и плохо ориентированное внимание, в частности при преподавании программирования. Важно напомнить одну истину, которая должна быть очевидной: эффективность имеет смысл лишь для правильной программы и нет смысла «оптимизировать» программу, не убедившись в ее адекватности требуемой задаче. Следующая программа на ФОРТРАНе – самая эффективная в мире по использованию времени и пространства:

```
STOP
END
```

Загвоздка лишь в том, что она отвечает лишь небольшому числу интересных спецификаций.

Вторая из «основных истин оптимизации» состоит в том, что не всегда следует оптимизировать; действительно, оптимизация – это исключение, а не правило. Этому есть две причины. Первая состоит в том, что в стоимости процесса программирования подавляющую часть составляет стоимость человеческого труда: выигрыш в 25% времени выполнения программы, предназначенной для использования раз в месяц в течение двух лет и требующей каждый раз десяти секунд машинного времени, приведет к прямым потерям, если «улучшением» будут заниматься два инженера в течение месяца. Вторая причина заключается в том, что выигрыш пренебрежим, если оптимизируют наугад: большинство программ, за исключением некоторых специальных областей, где обработка очень «последовательна», проводят очень неодинаковые отрезки времени в разных своих фрагментах.

В действительности можно считать правдоподобным часто приводимое интуитивное правило, по которому около 20% объектного кода ответственны за 80% времени выполнения; правило рекурсивно применяется и к этим 20% (т.е. 4% множества команд ответственны за 64% времени выполнения и т.д.). Это показывает, что было бы абсурдно оптимизировать за пределами этих 80%. Многие программы «научного» типа свидетельствуют о еще более предельном поведении: «3%–90%». Значительным моментом для программиста является часто то, что определение этих основных 20% или 3% не очевидно, если исходить из статического текста программы: лишь тесты позволяют их выделить при выполнении. Любая система программирования должна бы давать соответствующие средства для измерения; система АЛГОЛ W, например, дает, как мы видели, в качестве дополнительного варианта распечатку программы, указывающую число выполнений каждого блока, каждой ветви альтернативы и т.д. Опыт доказывает, что чтение этих результатов часто вызывает удивление: значащими циклами часто бывают не те, которых мы ожидали. И именно они одни оказываются достойными оптимизации.

Таким образом, прежде чем пытаться улучшить эффективность программы, необходимо проверить, будет ли такое улучшение полезным, и точно определить, где именно она должна быть улучшена. Оптимизация, которой не предшествовало такое систематическое исследование, не имеет смысла.

Установив этот весьма важный момент, тем не менее надо сказать, что оптимизация иногда необходима. Ведь если программа используется 100 раз в день и требует одну минуту машинного времени и 500 килослов, то, вероятно, было бы полезно изучить вопрос о возможных улучшениях. Таким образом, следующая пробле-

ма такова: что надо улучшать?

Традиционное преподавание программирования тоже явилось источником многих ошибочных представлений. Например, сколько начинающих программистов усвоило, что никогда не следует писать

```
B**2 –4.*A*C                                    (1)
```

а надо писать

```
B*B–4.*A*C                                      (2)
```

под тем предлогом, что умножение «происходит быстрее», чем возведение в степень! Сколько раз предписывалось программировать на ФОРТРАНе не так:

```
Z = ((A – 2 * B) * C + D)/((A –2*B)*E + F)      (3)
```

а так:

```
X=A–B–B
Z = (X*C + D)/(X*E + F) )                       (4)
```

Оправдывать эти правила соображениями эффективности абсурдно, и, вообще говоря, не правильно. Вычисление (1) может оказаться столь же быстрым, как и вычисление (2), если транслятор умеет распознавать случай, когда показатель равен 2, и представляет это возведение в степень в виде умножения. Вычисление по формуле (3) может оказаться более эффективным, чем по формуле (4), которая требует отведения места для дополнительной переменной *X,* размещения в памяти и загрузки, в то время как любой хороший «оптимизирующий» траслятор обнаружит в (3) общие подвыражения и сможет умножить *B* на 2 простым сдвигом на один бит влево одной части регистра. Выбор между (1) и (2), между (3) и (4) может обсуждаться, но не по причине эффективности, а по причине удобства чтения. Можно, например, считать (4) более ясным, так как формулы там менее длинные, наоборот, если (3) соответствует обычному виду, в котором, например, физикам известно это уравнение, то (4) будет нежелательным с точки зрения «производительности» программы, поскольку это потребует большего усилия для понимания.

> Трансляторы хорошо приспособлены ко всем локальным, «микроскопическим» улучшениям программ: обнаружению общих подвыражений, величин, которые не надо заново вычислять, тех величин, которые не изменяются за время от одного выполнения тела цикла до другого, и т.д. При современном состоянии техники трансляции абсурдно пытаться повлиять на способ, каким программисты пишут свои программы, посредством доводов эффективности этого уровня. Единственным критерием должно служить соображение удобства чтения.

Как же, каким образом улучшить программу? Ответ был намечен в гл. VII: лишь существенное улучшение алгоритма может внести чувствительный прогресс в эффективность. Это может быть настоящий «концептуальный скачок», который изменит *теоретическую сложность* алгоритма с $O(n^2)$ на $O(n\log n)$. Смехотворно, например, пытаться с помощью «нечестных приемов» улучшить программу сортировки, имеющую сложность $O(n^2)$, чтобы получить выигрыш в несколько процентов: лишь переход к алгоритму сложности $O(n\log n)$ может действительно что–то дать. Важные методы, представленные в VII.1 (*Разделяй и властвуй, Компромисс пространство–время, Уравновешивание, Компиляция/Интерпретация*), являются здесь ключевыми для возможных улучшений, и именно поэтому каждый программист должен их знать. Применение идеи компромисса пространство–время может также дать удивительную выгоду, если заметить, например, что некоторые величины среди 10 000 возможных постоянно перевычисляются, хотя они редко изменяются, и их можно, следовательно, представить в виде массива из 10 000 элементов и массива из 10 000 логических величин, указывающих, правильна ли соответствующая величина из перво-

го массива, или ее надо перевычислять.

> Приходится только поражаться, до какой степени наивный подход к проблемам эффективности, к несчастью распространяемый многими курсами для начинающих, может на деле привести к снижению эффективности программ. Книга [Керниган 74] кишит примерами программ (взятыми из учебников для начинающих), авторы которых посчитали полезным различать многочисленные частные случаи, когда алгоритм, по их мнению, был бы более быстрым, забыв при этом, что все тесты, переходы и т.п., необходимые для распознавания этих частных случаев, приводят к потере времени, гораздо большей, чем выигрыш, который они надеялись получить! Наиболее поразительный пример, приведенный Керниганом, это пример программы сортировки, основанной на Пузырьковой сортировке, в котором все возможные улучшения были заботливо учтены, рассмотрены все частные случаи и т.д.: программа длиной в 30 строк на ФОРТРАНе занимала в действительности на 30% больше времени, чем грубое кодирование алгоритма Пузырьковой сортировки (VII.3.6.1) для сортировки 2000 элементов, причем этот процент увеличивался с увеличением числа сортируемых элементов. Сомнительный характер такого улучшения объясняется временем, необходимым для обнаружения «улучшаемых» случаев, и увеличением сложности программы. Заметим, в частности, что в современных операционных системах «пространство» и «время» не являются независимыми: «большая» (длительная. – *Перев.*) программа обычно «длиннее». Заметим также (что должно быть очевидным), что в реальности «улучшения» надо убедиться с помощью доказательства, либо теста.

Существуют некоторые приемы улучшения эффективности внутренних циклов без подлинных «концептуальных скачков». Они находятся на полпути между «локальными» и «глобальными» оптимизационными задачами, немногие из современных трансляторов используют их для облегчения работы программиста, хотя теоретически это возможно. Можно привести, например, иногда возможное упразднение проверки $i \le n$ в

```
i ← l;
пока i ≤ n и p(a [i]) повторять
    | q(a[i]);
    | i ← i + 1
```

где a – массив с границами [1 : n]; при каждом выполнении цикла экономится одна проверка с помощью искусственной подстановки **«сторожевого» значения** x в a[n + 1], такого, что p(x) имеет значение **ложь** (ср. упр. III.5). Это, однако, может создать ряд проблем, например, если цикл находится в подпрограмме, аргументом которой является массив a. С риском наскучить читателю напомним, что такой род манипуляций оправдан, лишь если есть уверенность, что мы занимаемся значительной проблемой, т.е. проблемой самых внутренних циклов.

Другой классической оптимизацией является **«развертывание циклов».** Имеется элемент программы:

```
для i от 1 до 100 повторять
    | a(i)
```

При классическом представлении на машине используется проверка и ветвления

```
                i ← 1;
цикл:           если i > 100 то на продолжение;
                a(i);
                на цикл;
продолжение: ...
```

Если действие a(i) занимает сравнительно небольшое время, сравнимое со временем проверки, то можно получить выигрыш в несколько процентов, записав

```
для i от 1 до 99 шаг 2 повторять
    | a(i);
    | a(i +1)
```

т.е. в терминах представления

```
                i ← 1;
цикл:           если i > 99 то на продолжение;
                α(i);
                i ← i + l;
                α(i);
                на цикл;
продолжение: ...
```

Здесь экономится половина проверок i > 100. «Коэффициент развертывания» здесь равен 2; в общем случае он обязательно должен быть делителем общего числа выполнений цикла. Развертывание можно делать, разумеется, только для тех циклов, у которых начальный и конечный индексы–константы. Читатель должен был заметить, что ничто не мешает дальнейшему развертыванию, однако с очевидностью встает вопрос компромисса пространство–время. Обсуждение всех этих приемов можно найти в упомянутых в библиографии работах Кнута.

Мы закончим, напомнив, что тщательная оптимизация требует ориентации на характерные особенности машины, на которой должна работать программа, т.е. в конечном счете использования языка ассемблера (эквивалентного машинному языку). Нет никакой причины возражать против написания некоторых элементов программы на ассемблере, если только выполнены следующие условия:

- рассматриваемый участок программы является действительно критическим в смысле эффективности и весьма не велик по своему объему;
- существует точная функциональная спецификация и эквивалентное описание в языке высокого уровня, непосредственно доступное в случае перемены «среды».

Чтобы завершить эти рассуждения об эффективности, можно было бы сказать: «не оптимизируйте!»; а затем смягчить эту рекомендацию следующим замечанием:

- надо сначала получить правильную версию, без этого последующая оптимизация лишена смысла;
- надо затем определить, необходима ли оптимизация, изучив соотношение между ожидаемым выигрышем при использовании ресурсов оборудования и предусматриваемые затраты программного обеспечения (время размышления, тесты, вновь появляющиеся ошибки, отладка и т.д.);
- если ответ на предыдущий вопрос положителен, надо очень тщательно определить, какие элементы заслуживают оптимизации, и применить к ним методы, оперирующие *сложностью* используемых *алгоритмов,* а не деталями, с которыми хорошо справляется транслятор.

VIII.5. Программист и другие

До сих пор мы изучали программирование как вид индивидуальной деятельности программиста, сидящего перед листом бумаги или перед терминалом. Но это, разумеется, только часть вопроса. Если не считать программиста, потихоньку, исподволь подготавливающего чудо, предназначенное для него одного и ни для кого другого (безусловно, реальная ситуация, которая иногда случается в карьере того или

иного пользователя ЭВМ, но которая тем не менее остается экономически незначительной), то программирование чаще всего–это работа, выполняемая несколькими лицами для других лиц. Такое состояние вещей ставит особые проблемы.

Этому вопросу были посвящены многочисленные уже книги; среди наиболее заметных можно привести книгу [Вейнберг 71], описывающую психологические проблемы, связанные с программированием, и книгу [Брукс 75], посвященную, в частности, организации больших проектов программного обеспечения. Для пока еще фрагментарного состояния исследований в этой области типично, что эти две работы, хотя и содержат интересные замечания и сведения, имеют в основном второстепенное значение: первая–как документ об американском образе мышления и о психологии, как она понимается в США, вторая – благодаря изложению этапов разра–ботки некоторых широко распространенных больших программ, несчастные пользователи которых сами оценивают апостериори всю соль этих историй.

VIII.5.1. Бригада программистов

Первая из возникающих в проекте проблем, относящаяся одновременно к психологии, социологии и технике информатики, это проблема организации бригады. Два первых аспекта занимают нас здесь лишь в той мере, в какой они испытывают влияние третьего, и тем самым отличаются в программировании от того, чем они могли бы стать в любой другой дисциплине, требующей коллективных усилий.

VIII.5.1.1. Задачи

Какие работы надо распределить? Некоторые из них очевидны: нужны **программисты** в широком смысле, в котором мы употребляем этот термин («аналитики», «функциональные аналитики», «проектировщики» и т.д.). Вероятно, будет еще **руководитель проекта.** Заметим по этому поводу, что «технический» руководитель может отличаться от «административного», поскольку имеются две одинаковые опасности доверить техническое руководство проектом «менеджеру», некомпетентному в программировании, или загрузить крупного специалиста в информатике административными вопросами. [Брукс 75] обсуждает этот вопрос весьма остроумно и исследует вопрос, когда оба типа ответственности делятся: кто при этом должен иметь преимущество, «техник» или «администратор»?

Менее широко признаны, но все же важны в большом проекте следующие обязанности:

- **технический помощник** – член бригады, который обладает особым опытом в одной или нескольких конкретных областях: язык программирования, используемое оборудование, система управления базами данных, документирующие системы... Распределяя определенным образом свое время, такой специалист поступает в распоряжение своих коллег, чтобы по-мочь в разрешении встречающихся им технических трудностей.

Ясно, что эти обязанности могут делить несколько членов бригады, в частности, если их знания дополняют друг друга или если круг задач не требует полной загрузки программиста; таких технических помощников можно выбирать по очереди среди «программистов». Важным моментом здесь является то, что эта роль должна быть признана официально. Опыт показывает, что она всегда требуется: если такая должность не запланирована, программисты берут за обыкновение обращаться к наиболее компетентным членам бригады, тем самым мешая их собственной работе. Лучше признать необходимость такого вида деятельности, чтобы можно было для нее создать лучшие условия.

- **документалисту** поручается составление внутренней и внешней документации проекта. Необходимость документалиста (или группы

документалистов) оправдана тремя следующими аксиомами: внутренняя и внешняя документация необходима; ее составление является трудной работой, уровень сложности которой сравним с самим программированием; если она не будет успешно завершена одновременно с построением программы, она никогда не будет завершена. Можно оспаривать необходимость делать из этой работы отдельный вид деятельности и поручать программистам составлять документацию выполняемых ими программ. Но программирование является слишком всепоглощающей деятельностью, что делает весьма случайной эффективность такого решения; кроме того, даже если техническое ядро документов доверено тому, кто действительно пишет программы, остается весьма значительная работа по координации усилий и согласованию документов. Заметим, что функция документалиста одна из самых трудных: она требует одновременно солидной технической компетенции, «литературных» способностей и склонности к синтезу.

- **секретарь бригады программистов,** функция, введенная Милзом и Бейкером (бригада Главного программиста, см. ниже), полностью обеспечивает в некоторых проектах связь между программистами и программной системой, а также управляет общей документацией (протоколы собраний и т.п.). Вообще говоря, это может быть лицо, не имеющее образования в области информатики, но эту должность может занимать секретарь, желающий повысить свою квалификацию. В методологии, разработанной Милзом и Бейкером, секретари бригады программистов полностью освобождают программистов от всех забот, связанных с отладкой и тестированием программ (пробивка перфокарт, обращение к терминалу в интерактивном режиме, управление файлами, восстановление результатов), позволяя тем самым программистам посвятить полностью свое время сложным аспектам программирования. Отметим, что введение этой должности, хотя и очень полезной, ставит серьезные проблемы в человеческом плане: секретарь бригады программистов выполняет одновременно работы, поручаемые обычно программистам, и выполняет организационные функции; существует риск, что в результате ему будут поручаться все самые нудные работы, а необычный характер его деятельности и его квалификации – наполовину программистской, наполовину секретарской – затрудняет присвоение ему удовлетворительного статута в бригаде и на предприятии.

VIII.5.1.2. Бригада главного программиста

Хорошо изученная организация «бригады главного программиста» (*Chief Programmer Teams*) была разработана Милзом и Бейке–ром из ИБМ и широко распространяется с тех пор этой фирмой (см. [Бейкер 72], [ИБМ 71] и статьи из [ACM 73]). Эта техника в основном была описана в связи с ее применением к большому проекту программированного обеспечения банка данных газеты «Нью–Йорк таймс». Она заслуживает краткого описания.

Основная идея «бригады главного программиста» – сделать организацию бригады схожей по структуре со структурой программы, которая получается в результате применения строго нисходящей разработки (VIII.3.3). Таким образом, получается бригада с сильной иерархической структурой; точнее, каждому узлу дерева программы сопоставляется сотрудник (обычно вместе с помощником), руководящий всей подгруппой, соответствующей поддереву (Рис. VIII.7). Корню дерева соответствует руководитель проекта, названный «главным программистом», вместе с «заместителем» (Backup Programmer).

Рис. VIII.7 Бригада Главного программиста.

Информация циркулирует тоже по древовидной схеме: каждый узел имеет доступ ко всем программам, написанным в поддереве, но может сообщаться с внешней по отношению к этому поддереву частью схемы только с помощью «отцовского» узла. В частности, Главный программист имеет доступ ко всему дереву и сам пишет наиболее ответственные элементы программы. Он должен «читать, понимать и утверждать все программы, написанные остальными членами бригады» (!) и может это делать «благодаря принципам структурного программирования» (Милз, в [ACM 73]).

Техника Милза и Бейкера имеет также другие аспекты: участие «секретаря бригады программистов» (эта должность была определена выше); использование «библиотеки программных носителей», позволяющей стандартизировать всевозможные способы реализации, хранения, документации и развития программ. Другим интересным элементом является проведение еженедельных конференций, на которых каждый программист объясняет своим коллегам свои последние программы; представить программу группе специалистов, которые ее до этого не знали, – это неоценимый способ извлечения большого числа ошибок.

Если оставить в стороне эти соображения, то метод «бригад главного программиста», надо сказать, не представляет никакого научного интереса. Под видом остроумной шутки можно было утверждать, что структура программы часто отражает структуру предприятия, откуда она происходит. В методе Милза и Бейкера – все та же идея, воспринятая всерьез и возведенная в ранг методологии программирования, но наизнанку: принять за структуру бригады структуру дерева программы! Конечно, теоретическое одеяние служит здесь только предлогом: словно случайно вновь обращаются к жесткой иерархической структуре американских предприятий. На ум также приходят огромные интерьеры[1], позволяющие каждому руководителю не терять из виду своих подчиненных.

Метод бригады главного программиста, без сомнения, даст материал для интересного социологического исследования на тему о том, как псевдонаучные доводы могут служить оправданием установлению жесткой власти на предприятии[2]. Ограничимся здесь замечанием технического характера о том, что фигура Главного программиста, некоего «Бога–Отца», царствующего над своей бригадой, некоего Наполеона, знающего все закоулки программы из 80000 перфокарт, мало правдоподобна. Ясно, что такой руководитель может быть только тираном или гением. Некоторые авторы (ср., например, [Демнер 78]), испробовав этот метод, удивляются, обнаружив «понижение индивидуальных творческих возможностей» программистов!

[1] Так называемые «bureaux paysagés» – бюро, расположенные в большом зале, где все сотрудники находятся «на глазах» у руководителя; этот зал может быть разделен небольшими стеклянными перегородками, вазонами с цветами и т.п., создающими «пейзаж». – *Прим. перев.*

[2] В момент, когда эта книга готовилась к печати, вышла работа американского социолога [Крафт 77]. В ней фигурируют многие элементы упомянутых исследований. Выводы Крафта могут показаться преувеличенными, однако его доводы солидно обоснованы. Он считает, что путь, которым промышленность восприняла и развила идеи «структурного программирования», разработанные в научных центрах, свели их к специальному варианту процесса раздробления труда и снижению квалификации трудящихся («тэйлоризация»), процесса, который еще с прошлого века обусловливал во всех отраслях переход от ремесленной стадии к индустриальной.

Программирование – это слишком сложное интеллектуальное занятие, чтобы можно было надеяться навязать ему узы иерархической системы, которая душит всякую инициативу. Проблема распределения заданий в бригаде программистов очень сложна; в основном она остается открытой проблемой, но можно надеяться когда–нибудь получить наброски решений, менее примитивных, чем метод Милза и Бейкера.

VIII.5.2. Программист. Заказчик. Пользователь

Вместо заключения расширим несколько область приложения наших рассуждений, вновь вернувшись вкратце к деятельности программиста или бригады программирования в привычном для них окружении, т.е. во взаимодействии с «заказчиками» (лицами, финансирующими разработку) и «пользователями» (теми, кто на деле будет эксплуатировать программу и ее результаты). Заметим, что «заказчик», «пользователь» и «программист» в самом простом случае могут быть единственным лицом; в других случаях категории заказчиков и пользователей могут быть неразличимы; но весьма часто имеет место тот общий случай, когда все три категории различны, что добавляет ко всем другим трудностям такую проблему: хорошо ли соответствуют определенные официальным «заказчиком» спецификации будущим нуждам реальных «пользователей». Проблема эта тем более деликатна, что пользователи часто располагаются на более низкой ступени иерархической лестницы. Но, в конце концов, это не входит в компетенцию программиста, и здесь можно ограничиться лишь несколькими замечаниями о прямых отношениях программистов с двумя другими категориями.

Эти отношения часто ставят программиста в неудобное положение, и, без сомнения, виноваты в этом как те, так и другие. Когда пользователи вступают в контакт с программистами и их программами, они вправе формулировать три требования: чтобы программы было легко использовать при помощи простого формализма ввода данных (т.е. входного языка), чтобы документация была адекватная и ясная, чтобы в случае трудностей можно было бы легко получить у специалиста надлежащие разъяснения. И если эти условия не всегда выполнены, то следует признать, что вина за это лежит частично на специалистах в области информатики: из–за их жаргона, из–за их склонности забывать, что программа должна использоваться людьми, которые ничего в информатике не понимают, и из–за того, что определение формата ввода данных или способа опроса является проблемой восприятия языка, должно рассматриваться как таковое и заслуживает углубленного изучения.

Однако программисты могут привести смягчающие вину обстоятельства. Одно из них–это уровень средств, которые они имеют в своем распоряжении: очень часто операционная система или язык программирования, предложенные руководством по совсем ненаучным причинам (привычка, коммерческие контракты, стремление подражать соседям...), мешают достичь некоторых желаемых целей либо позволяют их достичь лишь ценой значительных усилий. Так, фортрановские программы могут допускать данные в свободном формате, лишь если будет написан «лексический анализатор». В других случаях предложенная спецификация оказывается очень ограничительной или непоследовательной, что приводит ко второй категории отношений между программистами и внешним миром: отношению с «заказчиками».

Представим себе на минуту, какой должна быть нормальная ситуация, хотя сегодня она может показаться многим идиллической. Когда потенциальный «заказчик» собирается решить задачу с помощью вычислительной машины, он обращается к «программисту», чтобы проконсультироваться относительно возможности и уместности того или иного метода решения. Если «программист» советует выбрать решение с помощью методов информатики (что, несомненно, не является системой, поскольку ЭВМ не приспособлены для решения любых задач), он просит и получает полную *спецификацию* задачи, которую надо решить, в форме «технического задания»,

непосредственно пригодного к использованию. Он может тогда дать окончательную *смету:* необходимое время, персонал, программные и технические средства, общую стоимость, предполагаемую эффективность окончательной программы, так, как она представляется извне (определенную, например, временем реакции на события или на запросы), и т.д. Если программист получает согласие заказчика на смету, он выполняет работу вместе со своей бригадой и дает по истечении условленного времени требуемый продукт. Затем путем сравнения со спецификацией определяют, выполнен ли контракт.

Подобное описание вызовет улыбку у тех, кто привык к обычной атмосфере программирования «индустриального» типа, поскольку для наступления описанной выше ситуации имеется множество препятствий. Одно из них связано с проблемой спецификаций: большинство «технических заданий», большого размера и написанных на естественном языке, являются непонятными и нуждаются в полном переписывании, чтобы стать пригодными для эксплуатации в какой–нибудь формальной системе типа Z (VIII.3.5). Проблема спецификаций – это трудная задача; она должна быть выполнена программистом совместно с заказчиком. Кроме того, не существует простого и повсеместно принятого способа проверки соответствия программы и спецификации; обсуждая тесты, мы видели, что корректное поведение во множестве фиксированных заранее пробных случаев дает лишь призрачную гарантию.

Хуже того, очень часто не удается даже получить «техническое задание», которое было бы полным, хотя и неразборчиво написанным. Специалист в области информатики вынужден сам восстанавливать большую часть головоломки, встречаясь с различными «заказчиками», угадывая, какие вопросы следует задать, интерпретируя недоговорки, учитывая при этом постоянные изменения и разрешая многочисленные противоречия.

Положение еще усложняется из–за двусмысленности статуса программистов. С некоторой точки зрения их считают специалистами и доверяют их способностям. На деле же вера в неограниченные возможности информатики, в миф о всемогуществе вычислительной машины и об ее адекватности всем задачам настолько распространена, что любой прикладной математик будет считаться некомпетентным, если он признается в неспособности решить какую–нибудь задачу или просто выскажет сдержанность относительно применения информатики в каком–то конкретном случае. Но чаще всего программисту отказывают в должном признании его квалификации, т.е. в умении и знании методов работы и профессиональном мастерстве.

Интересно по этому поводу сравнить положение математика–прикладника со специалистами в других современных видах техники. Когда «заказывают» конструкцию моста инженеру–мостостроителю или создание электронного устройства инженеру–электронщику, то предоставляют детально разработанную спецификацию, однако быстро получают отпор, если пытаются предписывать частные методы практической реализации: вид бетона либо структуру схемы.

> Это означает, что в подобном случае специалист считается технически компетентным профессионалом, и ему предоставляют свободу действия, лишь бы можно было судить о результатах. Ничего подобного нет в информатике. Один из наиболее удивительных и раздражающих моментов профессии программиста–это значительная **техническая переопределенность,** в условиях которой специалист зачастую должен работать: выбор деталей (представление структуры данных, отладка алгоритмов, использование языков или существующих программ) навязывается ему преждевременно. Мотивы этого чаще всего носят чисто иррациональный характер: директор–заказчик после обсуждения с торговым агентом, пораженный магическими выражениями, которые пышным цветом расцветают в нашей профессии, считает, что неважны технические обоснования, но надо использовать *структурное программирование, систему баз данных* (еще лучше, *распределенных баз*

данных) или же *микропроцессоры;* благодаря какому–либо из этих «сезамов» все проблемы исчезают, и худо нашему программисту, если он через три месяца не представит удовлетворительную систему.

Эта техническая переопределенность тем более предосудительна, что она обычно является следствием описывавшейся выше *недостаточной функциональной спецификации:* «заказчики» стремятся давать всяческие советы по отладке, вместо того чтобы предоставить точное и полное описание своей задачи. Другими словами, программисту не удается получить спецификацию того, *что* надо делать, но он получает множество указаний, *как* это делать! В таком проекте программист, в чью задачу входит организация файла, т.е. структуры данных, пытается законными путями получить полную функциональную спецификацию, прежде чем пускаться в детали представления: читал ли он или нет гл. V, но он знает, что выбор представления бесполезен и нежелателен, пока неизвестно, какие именно операции будут производиться над структурой данных, каковы свойства этих операций, какова их соответствующая частота; он знает, что в игру вступила тонкая проблема компромисса пространство–время и т.д. Но пока он бегает в поисках ненадежного и меняющегося технического задания, он получает директиву использовать «указатели»: «заказчику» не удается уточнить, почему он будет использовать файл, но он знает, как надо работать с данными, из которых файл состоит! Может, кому–нибудь удалось склонить его за десять минут в пользу заманчивого принципа, выраженного в другом магическом правиле: *независимость данных от программ!*

Причины этой ситуации очень многочисленны, и вина лежит на многих. Очень долго сами программисты производили не очень серьезное впечатление каких–то ревностных хранителей коллекции секретных «трюков», в конечном итоге весьма низкого уровня. Надо также сказать, что искушение произвести *дамп* очень присуще человеку, хотя и не приносит пользы, как это выясняется впоследствии: какой кандидат на выполнение вычислительного проекта осмелится сказать, что проект требует для своего выполнения 18 месяцев, если он знает, что менее совестливый конкурент даст понять «заказчику», что он справится за шесть месяцев, забыв упомянуть, что отладка, тестирование и т.д. могут с большой вероятностью растянуться на три года?;

Со своей стороны «заказчики» и административные руководители проектов программирования, конечно же, несут свою долю ответственности: некоторые из них имеют тенденцию слишком часто путать информатику и организацию, программирование и работу на конвейере, закупку оборудования и рост возможностей системы. Заметим, что программистский жаргон обманчив: в нем используются слова, заимствованные из обычного языка в новом, техническом значении («база», «банк» (или «структура») «данных», «техническое обеспечение», «память», «операционная» «система», «язык», «программа», «вывод» и т.д.). Неспециалистам может показаться, что они понимают соответствующие понятия, многие из которых не так–то просты. Наконец, еще один уже упоминавшийся момент лежит в основе непонимания между программистами и «заказчиками»: подлинные трудности программирования не проявляются явно и, следовательно, техническая квалификация программистов становится оценимой только за пределами некоторого **барьера сложности,** определяемой размерностью решаемой задачи. Речь идет об одном из самых трудно понимаемых явлений в информатике: действующие механизмы фундаментально просты, но их многократная комбинация создает сложность. Только тогда **методы программирования** приобретают весь свой интерес.

Мы видим, что невозможно говорить об этих методах совсем абстрактно, не учитывая практических приемов работы программиста. Без сомнения, мы только наметили эти вопросы, и, быть может, этого было достаточно, чтобы убедить читателя, что эволюция программирования зависит не только от программистов. Однако их роль будет значительной в превращении программирования в научную дисциплину,

достойную уважения, в том свершившемся уже в ряде областей процессе, в котором предстоит еще так много сделать. Пусть эта книга будет небольшим вкладом в это дело.

ЭСКИЗ БИБЛИОГРАФИИ

За последнее время появилось множество литературы, посвященной методологии программирования, и представляется трудным произвести ее сортировку. Мы позволим себе предложить два совета: не слишком полагаться на сборники чудесных рецептов и глубоко изучать «непреходящие ценности»: работы Дала, Дейкстры, Хоара, Кнута, Наура, Вирта... . Другие важные ссылки встречаются в ходе изложения этой главы по мере изучения соответствующих тем.

Среди важных работ, которые следует напомнить, трилогия Дала, Дейкстры и Хоара [Дал 72] останется, без сомнения, самой полезной; в этой работе развиваются принципы «структурного программирования», примененного к программам, данным и «квазипараллельным» процессам (сопрограммы, ср. IV.7). Аскетическая дейкстровская позиция относительно программирования доведена до своего крайнего выражения в работе [Дейкстра 76]. Интересное, полное нюансов обсуждение некоторых аспектов «структурного программирования» можно найти в работе [Кнут 74], помещенной в специальном номере Computing Surveys of ACM, в котором содержатся другие полезные статьи, посвященные проблемам программирования.

Недавние размышления о методологии программирования завершились созданием нескольких экспериментальных языков: CLU [Лисков 77], Алфард [Вулф 75], [Шоу 77], ЕВКЛИД [Лэмпсон 77]. О проблеме создания языков см. также [Хоар 73].

Чтобы характеризовать множество работ, стремящихся превратить программирование в «инженерную» дисциплину, появилось выражение «конструирование программного обеспечения» (génie logiciel). Синтез состояния техники в этой области дается в статье [Боэм 77], содержащей огромную библиографию, см. также [Боэм 77a]. Интересные моменты можно найти также в работах: [Вейнберг 71], посвященной психологическим аспектам, [Брукс 75] и [Майерс 76]. Этому же вопросу посвящен целый журнал (IEEE Transactions on Software Engineering).

Напомним, наконец, недавнее появление первой книги, посвя–щенцой «социологии программирования» – [Кнут 77]. И если ее автор несколько увлекается полными энтузиазма пророчествами некоторых последователей «структурного программирования» (вовсе не очевиден факт, что вскоре можно будет производить программу как автомобиль (т.е. собирать из готовых деталей. – *Перев.*)), то в том, что касается деятельности программиста, он вносит новый и обнадеживающий взгляд.[1]

[1] В последнее время появился ряд хороших книг по технологии программирования в переводе на русский язык – [Брукс 75], [Йодан 79], [Хьюз 80]. Технологическим проблемам посвящена также книга [Вельбицкий 80]. – *Прим. Перев.*

ОТВЕТЫ К УПРАЖНЕНИЯМ И ЗАДАЧАМ

Глава I

1.1. Алвафитный порядок

Эта задача иллюстрирует одну из принципиальных трудностей программирования: трудность перехода от интуитивного понимания явлений повседневной жизни к необходимой формализации процессов в информатике. Не надо, например, забывать случай, когда некоторое слово является «приставкой» другого слова (слово КОМ предшествует слову КОМПОЗИЦИЯ в алфавитном порядке); ничего не надо предполагать «неявно» (так, ничто не позволяет предполагать, что каждое слово дополняется справа бесконечным числом пробелов, хотя пробел считается предшествующим всем буквам в алфавитном порядке) и т.д.

Мы предлагаем два ответа на поставленный вопрос. Первый рассматривает слова x и y длиной соответственно тип как последовательность букв $x_1x_2...x_m$ и $y_1y_2...y_n$. x считается *предшествующим* y в алфавитном порядке тогда и только тогда, когда существует индекс i, такой, что

а) $1 \le i \le m + 1$

и б) для всякого j, такого, что $1 \le j \le i$, если такие j существуют, $x_j = y_j$

и в) **либо** $1 = m + 1$ и $i \le n$

либо $i \le m$, $i \le n$ и буква x_i предшествует букве y_i в алфавитном порядке.

Второе, эквивалентное определение, возможно, покажется несколько удивительным для тех, кто привык к принятому нами способу рассуждений. Слово x **предшествует** слову y тогда и только тогда, когда верно одно из трех следующих условий (взаимно исключающих):

а) x – пустое слово (не имеющее ни одной буквы), а y – непустое;

или б) x – непустое, y – непустое и первая буква x предшествует первой букве y в алфавитном порядке;

или в) x – непустое, y – непустое и их первые буквы одинаковы, а слово, получаемое из x зачеркиванием его первой буквы, *предшествует* слову, получаемому из y зачеркиванием его первой буквы.

Читателю предлагается убедиться, что определение не содержит логических ошибок. Такое определение, внешне «циклическое», называется *рекурсивным;* рекурсия подробно изучается в гл. VI.

Глава II

II.1. Простительная ошибка

В ФОРТРАНе пробелы не являются значащими символами за пределами констант типа СТРОКА, даже если они встречаются посредине идентификатора или числовой константы. Здесь последние две строки эквивалентны такой записи:

*363143E – 11 *RW1**7363143E – 11 *RW1 **7

Это представляет собой конец выражения, совершенно правомочного с точки зрения транслятора. По отношению к желаемому выражению это сводится к умножению последнего члена на следующий дополнительный множитель:

*RW1**7.363143E – 11*

значение которого к силу крайней малости показателя очень близко к 1 даже для весьма больших *RW1*. Отклонения от нормального выполнения могли бы возникнуть только для отрицательных *RW1*. С другой стороны, несомненно, что с точки зрения времени выполнения программы эффект бесполезного возведения числа в дробную степень может оказаться существенным.

По поводу необнаруживаемых ошибок в ФОРТРАНе из–за его удивительной снисходительности к промежуточным пробелам см. также в разд. III.2.

II.2. Машинный нуль

а) При сложении оба слагаемых должны быть приведены к единому порядку. При наличии S цифр в системе с основанием B для представления мантиссы дробного числа (см. обозначения в разд. II.1.1.5) такое приведение может повлечь за собою потерю всех значащих цифр одного из операндов. Так, если B = 2, а S = 4, то сложение чисел

$$1 = 0{,}5 \times 2^{1}, \text{ представляемого} \begin{cases} e = 1 \\ m = 0{,}100001_2 \end{cases}$$

и $$0{,}03125 = 0{,}5 \times 2^{-4}, \text{ представляемого} \begin{cases} e = -4 \\ m = 0{,}1000_2 \end{cases}$$

математически давало бы результат

$$1{,}035125 = 0{,}515625 \times 2^{1}, \text{ что представимо как} \begin{cases} e = 1 \\ m = 0{,}100001_2 \end{cases}$$

однако фактически дает 1, так как последняя цифра утеряна.

б) Если результат сложения округляется, то $\varepsilon = \frac{B^{-S+1}}{2}$; если результат обрубается, то $\varepsilon = B^{-S+1}$

в)

```
переменная эпсилон: ВЕЩ;
эпсилон ← 1.0;
пока 1.0 + эпсилон > 1.0 повторять
        | эпсилон ← эпсилон / 2
```

Глава III

III.1. Сюрприз мусорной корзины

а) Программа вычисляет *NOT1* и *IOUI* в зависимости от *J* и *ITOUR* по следующей таблице:

ITOUR \ J	<1	≥1
≤ 1	NOT1 = NO2 IOUI = 1	NOT1 = NO1 IOUI = 1
> 1	NOT1 = NO1 IOUI = 0	NOT1 = NO2 IOUI = 0

(Замечание: *IOUI* должно быть, вероятно, логической переменной, а не целой.)

б)

ФОРТРАН

```
NOT1 = NO2
IOUI = 1
IF (ITOUR .GT. 1) IOUI = 0
IF((J.GE.1.AND.ITOUR.LE.1).OR.(J.LT.1.AND.ITOUR.GT.1) NOT1=N01
```

III.2. Слияние двух массивов

а) Операторы *GOTO FBOUCLE* в строках 8 и 14 означают, что если не исчерпан ни один из массивов *X* и *Y*, то продолжается заполнение массива *Z* следующим элементом, вызывая непосредственно завершение итерации в цикле по *K*. Более ясным было бы управление циклом с помощью *WHILE*, позволяющего «прямо» выйти из цикла, когда исчерпан один из массивов.

Смещения здесь непоследовательны: строка 12 начинается левее, чем строка 11! *END* строки 10 не выровнен с соответствующим *DO* строки 5 и т.д.

ELSE строк 9, 11 и 15 не нужны, потому что они следуют за *GOTO*: если выполняется ветвь *THEN* альтернативы, то программа в своем выполнении, во всяком случае, не вернется к операторам, следующим за ветвью *ELSE*.

Строка 17 означает, что из цикла нет нормального выхода, и это является еще одним основанием рассматривать его не как цикл для, а как цикл **пока**: не позднее 100-й итерации один из двух массивов необходимо окажется исчерпанным, что порождает ветвление – переход к *YDANSZ* или *XDANSZ* (это означает: в *Z* осталось поместить только то, что не было переписано из *X* или *Y* соответственно).

б) Среди прочих возможностей

ПЛ/1

```
...
I, J, K = 1;
DO WHILE (I <=50 & J <= 50);
    /* ЗДЕСЬ K = I + J – 1, А ЗНАЧЕНИЯ ОТ X(1) ДО X(I – 1) И ОТ Y(1) ДО Y(J
    – 1) РАСПОЛОЖЕНЫ В ВОЗРАСТАЮЩЕМ ПОРЯДКЕ В ЭЛЕМЕНТАХ ОТ
    Z(1) ДО Z(K – 1):
    ИНВАРИАНТ ЦИКЛА */
    IF X(1) < Y(J) THEN DO;
```

```
                    Z(K) = X(I);
                    I = I + 1;
                    END;
               ELSE DO;
                    Z(K)=Y(J);
                    J = J + 1;
                    END;
          K = K + 1;
          END
     /* ЗДЕСЬ ЛИБО I, ЛИБО J РАВНО 51, НО НЕ ОБА СРАЗУ */
          IF I <=50 THEN DO I1 = I TO 50;
                    K = K + 1;
                    Z(K) = X(I1);
                    END;
               ELSE DO J1=J TO 50;
                    K = K + 1;
                    Z(K)=Y(J1);
                    END;
```

Чтобы доказать правильность этой программы, надо воспользоваться имеющимся инвариантом цикла и теми фактами, которые легко из него выводятся для циклов по *I1* и *J1*.

Можно получить более простой алгоритм, если в элементах *X(51)* и *Y(51)* поместить равные между собой «предупреждающие» значения, которые превосходят все другие значения обоих массивов. Тогда больше не будет необходимости заботиться об обнаружении момента исчерпания массивов: достаточен единственный цикл для, содержащий одну альтернативу (близкую к первой половине исходной программы, но без анархических ветвлений).

III.3. Введение в программирование

Эта программа собрала столько практических опасностей, что не стоит пытаться побить этот рекорд!

Массив *G* надо рассматривать как символическую константу, получающую значение с помощью директивы *DATA* или с помощью подпрограммы инициализации, но не обязательно включаемую в поток данных для чтения первой же использующей ее подпрограммой! Числовые константы, характеризующие техническое обеспечение (как длина строки, здесь – 94 литеры, или число литер в одном машинном слове, здесь – 6), от использования которых не всегда можно уйти, также должны обрабатываться как символические константы; их всегда следует пояснять комментариями. Заметим, что здесь ФОРТРАН не дает по–настоящему удовлетворительного решения, потому что в предписаниях *DIMENSION* и *FORMAT* надо использовать константу в числовой форме. Напомним, кроме того, что единственный способ обеспечить некоторую переносимость программ–это размещение одной литеры в области, выделяемой для одного целого (ср. II.3.3.5). Всякое более компактное расположение представляет собой мероприятие по оптимизации использования памяти; необходимость такой оптимизации мало целесообразна в случае таблицы из 94 литер.

Строки 11 и 12 проверяют попадание числа в интервал [*ZMIN, ZMAX*] способом, предрасположенным к ошибкам и путанице. В предлагаемом решении используется, разумеется, логическое выражение, явно указывающее выполняемые сравнения. Вот полное решение:

ФОРТРАН

```
      SUBROUTINE GRAPH (Z, ZMIN, ZMAX, N)
      INTEGER N
      REAL Z(N), ZMIN, ZMAX
C     --- НАПЕЧАТАТЬ ГРАФИК ФУНКЦИИ, ЗАДАННОЙ
C     N ЗНАЧЕНИЯМИ Z, ОГРАНИЧИВШИСЬ ЗНАЧЕНИЯМИ,
C     ЗАКЛЮЧЕННЫМИ МЕЖДУ ZMIN И ZMAX ---
C
C     СТРОКА ГРАФИКА СОДЕРЖИТ 94 ЛИТЕРЫ---
      INTEGER LIGNE(94), BLANC, ETOILE
      DATA BLANC /1H/; ETOILE /1H*/
      INTEGER NUMCAR, COLONN
      DATA NUMCAR /94/
      LOGICAL DEDANS
C     ИДЕНТИФИКАТОРЫ: BLANC–ПРОБЕЛ,
C     ETOILE–ЗВЕЗДОЧКА, LIGNE–СТРОКА, С NUMCAR – НОМЕР ЛИТЕРЫ,
C     COLONN – СТОЛБЕЦ, DEDANS–ВНУТРИ
C
C     --- ИНИЦИАЛИЗАЦИЯ ---
      DO 100 I = 1, NUMCAR
          LIGNE(I) = BLANC
100       CONTINUE
C
C     --- СЛЕД КРИВОЙ СТРОКА ЗА СТРОКОЙ ---
      DO 200 1 = 1, N
C     --- I–Я СТРОКА ---
C     --- ПРОВЕРКА ПОПАДАНИЯ ЗНАЧЕНИЯ В ТРЕБУЕМЫЕ ПРЕДЕЛЫ ---
          IF(DEDANS) COLONN =
     1                INT((Z(I) – ZMIN*FLOAT(NUMCAR)/
     1                (ZMAX – ZMIN)) + 1
C                     --- ПО ПОСТРОЕНИЮ, ЕСЛИ DEDANS ИСТИННО,
C                         ТО COLONN НАХОДИТСЯ МЕЖДУ 1 И NUMCAR ---
          IF (DEDANS) LIGNE (COLONN) = ETOILE
C           --- ПЕЧАТЬ СТРОКИ С НОМЕРОМ I ---
          PRINT 10000, LIGNE, I
10000     FORMAT (1X, 94A1, 2X, 16)
C           --- ПРОПУСК В СЛУЧАЕ СТРОКИ ПРОБЕЛОВ ---
          IF (DEDANS) LIGNE (COLONN) = BLANC
200       CONTINUE
      RETURN
      END
```

III.4

а) Эта схема соответствует классическому для задач АСУ случаю обработки записей последовательного файла с проверкой на конец файла или с включением специальных видов обработки при прерывании последовательности значений, принимаемых некоторым заданным полем записей.

б) Можно писать на выбор

```
A;                          повторять
пока C повторять              | A
    | B;                      | если C то B
    | A                     до ~C
```

Форма с ветвлениями такова:

```
| начало: A;
|         если C то
|             | B;
|             | на начало
```

в) Многие авторы предлагают структуру цикла с условным оператором выхода из цикла вида

```
цикл
  | действие 1;
  | если условие выход;
  | действие 2
```

Вообще говоря, синтаксис этих конструкций позволяет выходить из нескольких различных точек цикла. Свойства

– **цикл** A; **если** C **выход**; B {C}
если {P} A {Q} и {Q ~C} B; A{Q}
то {P} **цикл** A; **если** C **выход**; B{Q}

В Q можно узнать эквивалент «инварианта» **цикла пока.** Структуры, более богатые, чем предложенные в этой главе, можно найти также в [Кнут 74].

III.5. Последовательный поиск

Ясность программы–понятие, несущее отпечаток субъективности. Вместе с тем можно заметить, что:

- программы с *GOTO* требуют гораздо больших усилий для понимания процесса выполнения операторов во времени; в то же время этот процесс непосредственно демонстрируется управляющими структурами, используемыми в III.4;
- программы с ветвлениями можно легко сделать более эффективными как во времени (отсутствие лишних проверок), так и в пространстве (отсутствие повторяющихся операторов). Заметим, впрочем, что многие трансляторы «оптимизируют» даже программы с «чистыми» циклами без ветвлений. Кроме того, архитектура современных машин (виртуальная память) рассматривает повторяющиеся ветвления между удаленными точками программы как весьма неблагоприятное обстоятельство (впрочем, такие ветвления могут быть результатом представления цикла);
- управляющие структуры удобны для доказательств, которые при более или менее свободном использовании ветвлений становятся затруднительными, практически невозможными.

Прибавление дополнительного элемента в конец массива невозможно (вообще

говоря) в подпрограмме. Поэтому метод применим не всегда.

III.6. Кено и три маленьких шустрых горошины

Программа может управляться диаграммой перехода, так что каждому состоянию и каждому ответу «Да» или «Нет» соответствует печатаемый текст и номер следующего состояния:

```
программа Кено
    массив текст–вопрос [0 : чисост, 1 : 2]: СТРОКА,
           след–состояние [0 : чисост, 1 : 2] :ЦЕЛ;
    переменные состояния: ЦЕЛ, t: СТРОКА,
                          тип–ответ : ЦЕЛ;
    {--- инициализация массивов: текст–вопрос [0,1] должен содержать
    текст "Хотите ли Вы узнать историю о трех маленьких шустрых
    горошинах?", а след-состояние [0,1] должно равняться 1 ---}
    состояние ← 0; t – "Да";
    повторять
        тип–ответ ← если t = "Да" то 1 иначе 2;
        писать текст–вопрос [состояние, тип–ответ];
        состояние ← след–состояние [состояние, тип–ответ]:читать t
    до состояние = 0 или t = "Достаточно"
```

III.7. Только программное управление

В новой программе цикл содержит

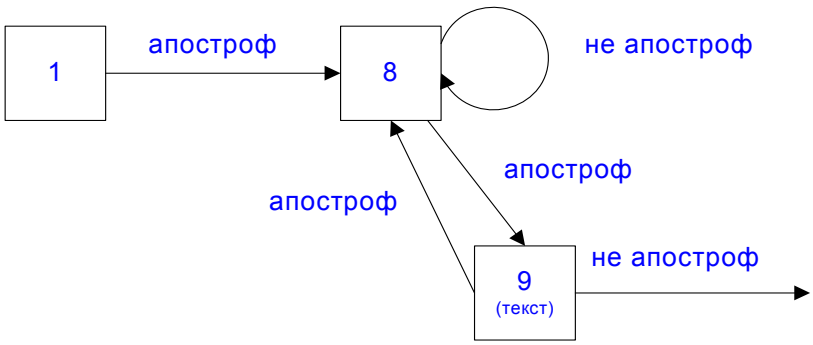
```
если класс (след–литера) = класс–цифра то
    повторять
        имя ← имя || след–литера;
        след–литера ← читать–лит
    до класс (след–литера) ≠ класс–цифра;
    если класс (след–литера) = класс–точка то
        повторять
            имя ← имя || след–литера;
            след–литера ← читать–лит
        до класс (след–литера) ≠ класс–цифра;
        пред–состояние ← 3
    иначе пред–состояние ← 2
иначе если класс (след–литера) = класс–буква то
    повторять
        имя ← имя || след–литера;
        след–литера ← читать–лит
    до класс (след–литера) ≠ класс–буква и класс (след–литера) ≠ класс–цифра;
    пред–состояние ← 7
{и т.д.}
```

Если внимательно вглядеться в эту программу и ее действия, можно констатировать значительное увеличение длины, а значит, увеличение вероятности ошибок и трудностей корректировок в случае возможных изменений спецификаций.

Управление с помощью таблиц требует распознавания некоторых свойств «грамматики» языка, тогда как только программное управление менее требовательно; см. следующее упражнение.

III.8. А для текстов?

Для констант типа СТРОКА, ограниченных апострофами, решение состоит в пополнении диаграммы следующей частью:



Но «холлеритовские константы» не подчиняются этому методу, потому что они не ограничены какими–либо особыми литерами, а определены с помощью числа *n*, записываемого перед буквой *H*. В этом случае диаграмма перехода должна бы включать по меньшей мере столько дополнительных состояний, сколько возможных значений может принимать *n*, т.е. теоретически бесконечное число. С другой стороны, для программы, работающей без таблицы, не составит никаких трудностей вычисление *n* по прочитанным цифрам и выполнение затем цикла **для**.

III.9. Индийское возведение в степень

Надо доказать, что начальная гипотеза

$\{A > 0, B \geq 0\}$

приводит к завершению цикла и что на выходе из него можно утверждать

$\{z = A^B\}$

Докажем прежде всего, что если цикл заканчивается, то программа правильна, отложив на дальнейшее доказательство завер–шимости цикла («частичное доказательство правильности»). Инвариант (P) подсказывает:

$\{x > 0, y \geq 0$ **и** $z \times x^y = A^B\}$ (P)

тривиально верно после трех присваиваний инициализации, так как тогда $x = A$, $y = B$ и $z = 1$. Если нам удастся доказать, что P – инвариант для тела цикла, то на выходе из этого цикла получается в силу характеристических свойств цикла **пока** (III.4.4):

$\{P$ **и** $y = 0\}$

т.е., в частности,

$z \times x^y = z = A^B$

что и требуется доказать.

По форме тела цикла, являющегося условным оператором и аксиоматическим свойствам этого типа операторов (III.4.3), нужно доказать отдельно

а) $\{P$ **и** $y \neq 0$ **и** y нечетно$\}$
$z \leftarrow z \times x;$
$y \leftarrow y - 1$
$\{P\}$

б) $\{P$ **и** $y \neq 0$ **и** y четно$\}$
$x \leftarrow x \times x;$
$y \leftarrow \frac{y}{2}$

$\{P\}$

Чтобы доказать **а)**, достаточно в силу характеристического свойства присваивания (III.4.1) и цепочки (III.4.2) доказать, что в результате первого присваивания $z \leftarrow z \times x$ свойство $P[y - 1 \to y]$ верно, т.е.

$$\{x > 0,\ y > 0,\ y \text{ нечетно и } z \times x^y = A^B\} \tag{1}$$

$$z \leftarrow z \times x$$

$$\{x > 0,\ y - 1 \geq 0 \textbf{ и } z \times x^{y-1} = A^B\} \tag{2}$$

другими словами, используя снова свойство присваивания (замена z на $z \times x$ в утверждении (2)), нужно доказать, что из начального утверждения (1) следует

$$\{x > 0,\ y - 1 \geq 0,\ \textbf{и } (z \times x) \times x^{y-1} = A^B)$$

что непосредственно проверяется.

Чтобы доказать **в)**, применим тот же метод ко второй ветви альтернативы. После замены y на $y/2$ в P это означает доказательство

$$\{x > 0,\ y > 0,\ y \text{ четно } \textbf{и } z \times x^y = A^B\} \tag{3}$$

$$z \leftarrow x \times x$$

$$\{x > 0, \frac{y}{2} \geq 0,\ \textbf{и } z \times (x \times x)^{\left(\frac{y}{2}\right)} = A^B\} \tag{4}$$

т.е. после замены x на $x \times x$ в (4) это приводит к доказательству того, что из (3) следует

$$\{x \times x > 0, \frac{y}{2} \geq 0,\ \textbf{и } z \times (x \times x)^{\left(\frac{y}{2}\right)} = A^B\}$$

что верно, поскольку y четно, а x положительно.

$$(x \times x)^{\left(\frac{y}{2}\right)} = x^y\}$$

Заметим, что в любом случае рассуждения ведутся «от противного», от заключения к гипотезе.

Чтобы доказать, что цикл завершается, достаточно отметить, что y – это «управляющая величина» (III.4.4), так как $\{y \geq 0\}$ есть инвариант цикла и каждое выполнение условного оператора заставляет y увеличиваться по крайней мере на 1. Это завершает доказа–теельство правильности программы.

Значение этой программы для больших значений в состоит в том, что она вместо $B - 1$ умножений, требуемых тривиальным алгоритмом (умножение A самого на себя $B - 1$ раз), выполняет число умножений, равное

$$\lfloor \mathrm{Log}_2 B \rfloor + h(B)$$

где первый член (целая часть логарифма B по основанию 2) соответствует умножениям $x \times x$, а второй является числом единиц в двоичном представлении B, соответствующим умножениям $z \times x$

Об этом алгоритме (известном в Индии еще в III в. до нашей эры) и методах быстрого возведения в степень см. [Кнут 69.4.6.3].

III.10. Аксиометрическое расширение

Для цикла **повторять ... до** получают

повторять A **до** C $\{C\}$

если $\{P\}$ A$\{Q\}$ *и* $\{Q$ *и* $\sim C\}$ A$\{Q\}$

то $\{P\}$ **повторять** A **до** C$\{Q\}$

Свойства цикла для выводятся либо из свойств цепочки с рекуррентным использованием свойств P_m, P_{m+i}, P_{m+2i}, ..., P_n либо из свойств бесконечных циклов, частный случай которых представляет собой цикл **для**; свойства **выбрать** и *CASE OF* легко выводятся обобщением из свойств альтернативы.

III.11. Неразрешимость

Определим программу Магия следующим образом (с использованием обозначений, введенных в гл. IV**):**

программа Магия (**аргумент** р: ПРОГРАММА)
| пока Р(р, р) **повторять** пустое действие

Если программа р, имеющая в качестве аргумента свой собственный текст, не завершается, то Р(р, р) есть **ложь** и Магия завершается; и наоборот, если р завершается, то Р(р, р) **– истина** и Магия «зацикливается».

Выполним теперь программу Магия(Магия): из только что рассмотренного следует, что Магия завершается тогда и только тогда, когда она не завершается! Это противоречие и доказывает, что исходная гипотеза о существовании Р неверна.

Глава IV

IV.1. Различные способы передачи

	a[1]	a[2]	замечания
значение	3	3	
результат	не определено	3	x не имеет начального значения в подпрограмме
значение–результат	1	3	третий оператор программы р работает с *начальным* значением a[1]
адрес	2	3	первый и третий операторы взаимно уничтожаются
имя	3	2	x становится синонимом a[2] после выполнения i ← i + 1

IV.2. Повторение фактического параметра

Когда передача параметров осуществляется по адресу или по имени, всякая модификация одного из «синонимичных» формальных параметров вызывает одновременную модификацию других; их прежние значения становятся недоступными в подпрограмме, и возможные последствия трудно предусмотреть.

Если два или более идентичных параметров передаются как **результаты** или **значения–результаты** и если значения формальных параметров не совпадают с их значениями в конце выполнения подпрограммы, то получаемый результат зависит от порядка завершающих переписываний.

Единственным безопасным случаем является тот, при котором . все вхождения, кроме, быть может, одного, передаются **значением.** Условия, определяющие такую ситуацию, требуют также одновременного контроля над подпрограммой и всеми ее вызовами; в так называемых языках с «раздельной трансляцией», таких, как ФОРТРАН или ПЛ/1, эта работа возлагается на «редактор связей», который, вообще говоря, для этого не предназначен.

IV.3. Чтение–упаковка–воспроизведение

Используются три сопрограммы: чтение, упаковка и выход. Начальное обращение адресуется к чтению, отсюда же осуществляется возврат.

Именем конец обозначается некоторая литера, не появляющаяся на входе:

```
программа чтение
    массив {свободный} карта [1 : 80] ЛИТ;
    пока ~ конец–файла повторять
        чтение карта;
        для i от 1 до 80 повторять
            возобновить упаковка(карта[i]);
        возобновить упаковка (" ") {пробел};
    возобновить упаковка (конец)
сопрограмма упаковка (аргумент лит: ЛИТ)
    переменная {свободная} предлит: ЛИТ;
    предлит ← лит;
    повторять бесконечно
        если предлит = "  " то
            возобновить выход (предлит);
            возобновить чтение до лит ≠ "  "
        иначе если предлит = "*" то
            возобновить чтение;
            если лит = "*" то
                возобновить выход ("↑")
                возобновить чтение
            иначе
                возобновить выход ("*")
        иначе
            возобновить выход (предлит);
            возобновить чтение;
        предлит ← лит
сопрограмма выход (аргумент лит: ЛИТ)
    массив {свободный} строка [1 : 132] ЛИТ;
    переменная {свободная} i, j : ЦЕЛ; i ← 0;
    повторять бесконечно
        если i = 132 то
            печать строка;
            i ← 0
        иначе если лит = конец то
            для j от i + 1 до 132 повторять
                строка[i] ← "  "
            печать строка;
        i ← i + 1; строка [i] ← лит;
        возобновить упаковка (лит)
```

Глава V

V.1. Топологическая сортировка

Чтобы доказать правильность предложенного алгоритма, заметим прежде всего, что свойство «C задает строгий частичный порядок на A» есть инвариант цикла: извлечение из A элемента a, не встречающегося ни в одной из пар [x, a] в C, и из C – всех пар вида [a, x] не нарушает частичного порядка среди оставшихся элементов.

Поскольку C задает частичный порядок на A, то в A всегда существует элемент a, такой, что никакой x не предшествует a. Действительно, если бы это было не так, то для всякого элемента из A нашелся бы элемент, который ему **предшествует** и, следовательно, обязательно от него отличен; тогда, отправляясь от некоторого произвольного элемента a, можно было бы построить бесконечную цепь $a_1 = a, a_2, a_3, ...$, такую, что a_{i+1} **предшествует** a_i для всякого i. Поскольку A – конечное множество, существуют i и k, такие, что a_{i+1} **предшествует** a_i, a_{i+2} **предшествует** a_{i+1}, ..., a_{i+k} **предшествует** a_{i+k-1} и $a_i = a_{i+k}$, что невозможно по определению строгого частичного порядка.

Каждое выполнение цикла извлекает один элемент из a; алгоритм завершается в силу конечности A. К тому же никакой из оставшихся элементов не может по построению предшествовать a. Это завершает доказательство.

С точки зрения используемых структур данных существует много возможностей. Простое решение состоит в представлении каждого элемента из A в виде целого, заключенного между 1 и n, а отношения порядка – в виде массива множеств, позволяющего для каждого элемента a реализовать доступ ко всем x, таким, что a предшествует x

массив преемники [1 : n] $МНОЖЕСТВО_{цел}$

Кроме того, с каждым элементом a связывают значение, обозначаемое число-предшественников[a], указывающее число элементов x, которые **предшествуют** a на каждом этапе:

массив число–предшественников [1 : n] ЦЕЛ

Важно правильно инициализировать массивы преемники и число-предшественников. С учетом этого оператор «найти в A элемента a, такой, что никакой другой элемент A не **предшествует** a» состоит в том, чтобы найти индекс a, такой, что число–предшественников [a] = 0 (здесь, выбирая из нескольких элементов, удовлетворяющих этому условию, можно воспользоваться каким–нибудь критерием оптимизации). Оператор «извлечь a из A и все пары вида [a, x] из C» состоит просто в выполнении

```
число–предшественников [a] ← 1
{после этого a никогда не рассматривается}
для x из преемника [a] повторять
        | число–предшественников[x] ← число–предшественников[x] – 1
```

Возможны многочисленные усовершенствования. Можно освободиться от последовательного просмотра массива число–предшественников при каждом выполнении цикла, использовав файл, называемый безпред и содержащий все элементы, которые больше не имеют предшественников: если безпред соответствующим образом инициализирован, то достаточно занести в безпред любой x, такой, что число–предшественников [x] становится нулевым в вышеприведенном цикле **для**; чтобы найти на каждом этапе алгоритма элемент без предшественников, достаточно извлечь элемент из файла безпред, не просматривая массив число-предшественников.

Для представления множеств преемники[a] ($1 \le a \le n$) заметим, что единственная применяемая к ним операция – последовательный просмотр, отдельный для каждого из них.

Более подробно с топологической сортировкой можно познакомиться в [Кнут 68] (2.2.3) и [Вайемэн 75].

V.2. Обходы дерева влюбленных

ЛПК (постфиксный обход):

ВАШИ ПРЕКРАСНЫЕ ГЛАЗА МАРКИЗА МНЕ СУЛЯТ СМЕРТЬ ОТ ЛЮБВИ

КЛП (префиксный обход):

ОТ ЛЮБВИ МАРКИЗА ВАШИ ПРЕКРАСНЫЕ ГЛАЗА СМЕРТЬ МНЕ СУЛЯТ

ЛКП:

ВАШИ ПРЕКРАСНЫЕ ГЛАЗА МАРКИЗА ОТ ЛЮБВИ МНЕ СМЕРТЬ СУЛЯТ

V.3. Законность *РАВЕНСТВА*

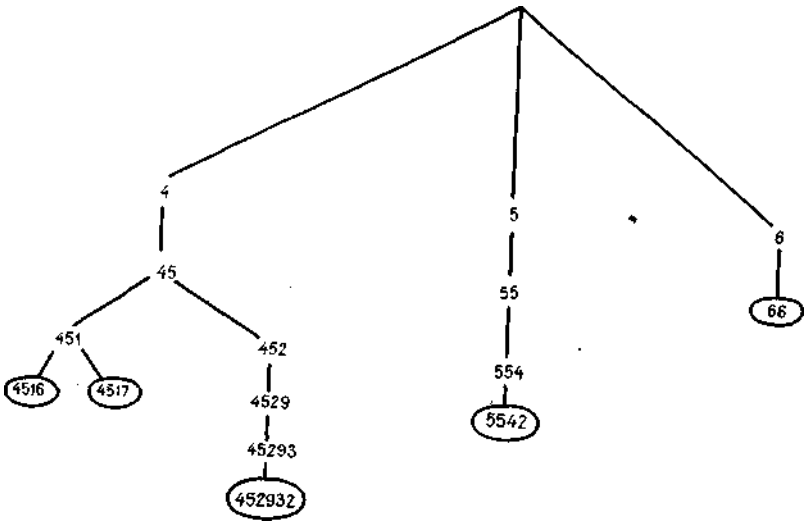
Правильность этой процедуры обеспечивается тем фактом, что она никогда не пытается вычислить несуществующие объекты, благодаря правилам вычисления логических выражений, содержащих операции *AND* и *OR* в АЛГОЛе W (III.5.3.1.a). Так, если *B1 = B2 = NULL,* то выражение *(B1 = B2)* не определено. Однако оно не вычисляется: первый член *OR* принимается равным *TRUE,* таким же будет, следовательно, и результат процедуры; если *B1 = NULL,* а *B2 ≠ NULL,* то три последних члена не вычисляются и не выполняется никакой рекурсивный вызов, результат тривиально равен *FALSE.*

Глава VI

VI.1. Бухгалтерская задача

Предлагаются два рекурсивных решения этой задачи; второе вытекает из первого, но его эффективность, несомненно, выше. Нерекурсивные решения получаются из них обычными методами (стек). Их можно также получить и непосредственно путем нерекурсивного .рассуждения. Заметим, что в своих рассуждениях мы не стремимся представлять дело так, будто один из этих способов в чем–то превосходит другой, даже если нерекурсивные решения имеют неудобство, состоящее в смешении управляющих структур и структур данных, присущих задаче, с элементами, которые служат только реализации рекурсии: управление вызовами, управление стеками. Мы просто пытаемся объяснить применение рекурсии как одного из фундаментальных механизмов, которым должен владеть каждый программист, включая такие области, как АСУ, где сейчас использования рекурсии редки.

Множество счетов нашей задачи удобно описать как множество листьев дерева номеров. Дети вершины, соответствующей номеру $x_1x_2...x_iy$, соответствуют номерам $x_1x_2...x_i$, где y – некоторая цифра. Номера счетов могут быть связаны только с листьями, так как указано, что никакой номер счета не является корнем другого номера счета. Дерево, соответствующее счетам из нашего примера, имеет вид



Корень соответствует пустому номеру. Эта интерпретация с помощью дерева уточняет поставленную задачу: речь идет о том, чтобы связать с каждым листом остаток, соответствующий номеру счета, такому, как его задает файл F, и с каждой внутренней вершиной–сумму значений, соответствующих сыновьям этой вершины. Печатаются значения, соответствующие листьям, и значения, связанные с внутренними вершинами глубины, не превосходящей m («прерывание последовательности»); условия a и b добавляют просто, что значение, соответствующее внутренней вершине, которая имеет единственного сына, как 4 или 452, не должно печататься. Заметим, наконец, что файл F упорядочен «лексикографически», т.е. соответствующее дерево обходится алгоритмом КЛП, или «префиксным» обходом (V.7.5).

Структура дерева подсказывает рекурсивный алгоритм. Первое простое решение состоит в том, чтобы отправляться не от множества номеров счетов, встречающихся в файле F, что могло бы априори показаться естественным, а от дерева, представляющего структуру множества номеров счетов предприятия. Номера имеют от 1 до n = 9 цифр; для каждого номера $x = x_1x_2... x_i$ определим множество преемники(x), образованное из номеров $x_1x_2...x_iy$, которые являются либо существующими номерами счетов, либо корнями существующих номеров счетов (как в структуре, изображенной на представленном выше дереве, множество преемники (45) содержит номера 451 и 452). Множество преемники (x) пусто, если x – номер из 9 цифр или номер счета (поскольку по предположению никакой номер счета не является корнем другого номера счета).

Получают следующий алгоритм:

```
h ← состояние (ПУСТО)
```

где ПУСТО – пустой номер (корень дерева), а подпрограмма состояние задается описанием

```
программа состояние: ЦЕЛ (аргумент с: НОМЕР)
    {печать части бухгалтерского отчета, соответствующей с и счетам корня с
    (поддерево); задаваемый результат – итог по с}
    переменные состояние–преемника, число–ветвей : ЦЕЛ;
    если с номер счета, встречающийся в F вместе со своим остатком то
            печатать с, остаток (с);
            состояние ← остаток (с)
    иначе
            {просуммировать по с и напечатать итог, если он есть}
            число–ветвей ← 0; состояние ← 0;
            для x из преемники (с) повторять
                {просуммировать по поддереву корня x}
                состояние–преемника ← состояние (x);
```

```
                    если состояние–преемника ≠ 0 то
                        число–ветвей ← число–ветвей + 1;
                        состояние ← состояние + состояние–преемника
                если с имеет не более m цифр и число–ветвей > 1 то
                {прерывание рассматриваемой последовательности}
                печать с, состояние
```

Как и во всяком алгоритме «последовательных испытаний» или «возврата», или, более широко, обхода дерева, несколько таинственным для тех, кто не привык к рекурсии, кажется то, что продвижение по дереву, «подъемы» и «спуски», не описаны явно: все это поручено механизму рекурсии.

Этот алгоритм верен, но практически не вполне удовлетворителен: вместо последовательной обработки файла F, т.е. просмотра множества номеров счетов, фактически участвующих в отчетной ведомости, он просматривает множество всех возможных номеров (которое, вообще говоря, существенно больше) и каждый раз проверяет «принадлежит ли этот номер F?».

Для получения более реалистического алгоритма надо заменить эту проверку последовательным чтением элементов F, используя упорядоченность F и префиксный порядок обхода дерева. Условимся, что оператор

```
чтение с, остаток (с)
```

читает «запись» из F, т.е. номер счета с и соответствующий остаток. Если x и y – два номера, можно проверять, является ли x корнем y; например, 76 – это корень 765 и 76842, но не является корнем для 754 и 7. Пустой номер, не содержащий никаких цифр, считается корнем всех других номеров. Условимся, наконец, что F заканчивается записью, которая содержит специальный номер, обозначающий «конец файла».

Программа печати ведомости вызывается с помощью операторов:

```
чтение x, остаток (x)
h ← состояние (x, 0)
```

где программа состояние теперь описывается:

```
программа состояние : ЦЕЛ
        (модифицируемое данное с : НОМЕР; аргумент число–цифр : ЦЕЛ)
        {инвариант вызова это номер счета из i или более цифр, принадлежащий
        вместе со своим остатком файлу F}
        {эта подпрограмма выдает в качестве результата итог, соответствующий
        первым i цифрам с, и печатает его, если он есть}
        {при возврате из подпрограммы новое значение с' величины с будет
        обозначать первый встречающийся в F счет с номером с и не
        начинающийся с тех же i первых цифр}
        переменные число–ветвей : ЦЕЛ,
                   кор : НОМЕР;
        если i = число–цифр с то
            печатать с, остаток(с); состояние ← остаток(с);
            {перейти к следующей записи} чтение с, остаток (с)
        иначе {с имеет более чем i цифр: подвести итог по первым цифрам
              счета}
            кор ← номер, образованный из i первых цифр с;
            число–ветвей ← 0; состояние ← 0;
            пока с ≠ конец файла и кор не является корнем с повторять
            {подвести итог по счетам корня кор}
            число–ветвей ← число–ветвей + 1;
            состояние ← состояние + состояние (с, i + 1);
```

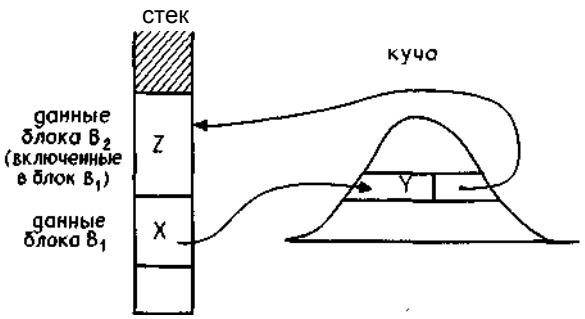
```
                    если i ≤ m и число–ветвей > 1 то
                    {прерывание рассматриваемой последовательности}
                    печать кор, состояние
```

VI.2. Поиск корня

Будем основываться на нерекурсивных решениях задачи о Ханойской башне. Достаточно занести в стек значение, указывающее, является ли выбранный полуинтервал левым или правым: тогда в случае неудачи легко восстановить предыдущие интервалы удвоением влево или вправо; для этого надо предполагать, что ошибки округления не составляют здесь серьезную задачу.

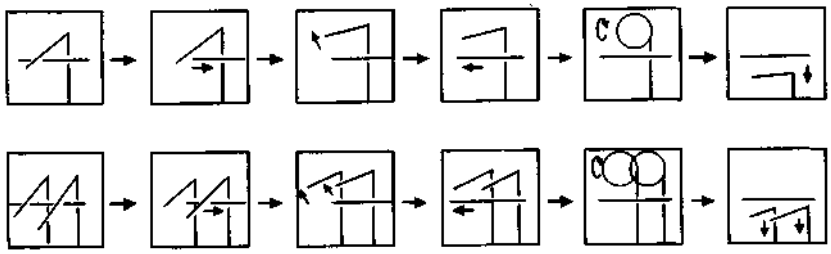
IV.3. Стек и куча

В следующей конфигурации в конце выполнения блока B2 указатель Y продолжает отмечать данное Z, которое было отменено; это классический случай «висячей ссылки» в ПЛ/1



VI.4. Головоломка

Эта головоломка может быть решена рекурсивным алгоритмом, сходным с алгоритмом Ханойской башни, хотя, очевидно, более сложным. Изображенные здесь схемы подсказывают «тривиальный случай» и «сведение к общему случаю» для задачи с одним и двумя кольцами (вид спереди):



Так же как общий случай задачи о Ханойской башне решается операторами переместить(x, y) или переместить(x, z) в зависимости от четности числа n колец, общий случай этой игры решается, начиная с «пересечения» одного или двух колец в зависимости от четности общего числа колец. Здравый смысл здесь говорит, что первый этап декомпозиции задачи с n кольцами состоит в решении рекурсивной задачи, ставящейся для n – 2 самых левых колец.

VI.5. Язык ПЛ/ –1

В ПЛ/ – 1 могут быть определены следующие функции:

```
---     сумма (x, y) = сум1 (x, y, 0)
        где сум1 (x, y, t) =  если равно (x, t) то y
                              иначе сум1 (x, преемник (y), преемник (t))
---     разность (x, y) = раз1 (x, y, 0)
        где раз1 (x, y, t) =  если равно (x, y) то
```

```
                                иначе раз1 (x, преемник (y), преемник (t))
```

Заметим, что эта функция не определена для x < y (вычисление «зацикливается»).

```
---     произведение (x, y) = прol (x, y, 0)
        где прol (x, y, t) =  если равно (x, 0) то
                              иначе про1 (разность (x, 1), y, сумма (y, t))
---     и (a, b) =            если a то b иначе ложь
---     или (a, Ь) =          если a то истина иначе b
---     не (a) =              если a то ложь иначе истина
---     больше (x, y) =       если равно (x, 0) то ложь
                              иначе если равно (y, 0) то истина
                              иначе больше (разность (x, 1), разность (y, 1))
---     Фибоначчи (n) =       если или (равно (n, 0), равно (n, 1)) то 1
                              иначе сумма (Фибоначчи (разность (n, 1))),
                                    (Фибоначчи (разность (n, 2)))
```

и т.д. и т.п. Поскольку любая обработка информации может рассматриваться как обработка целых натуральных, ПЛ/–1 имеет (теоретически) ту же внутреннюю мощность, что и любая машина или любой язык программирования.

VI.6. Функция Аккермана

Для нескольких малых значений m получают

Аккерман (0, n) = n + 1

Аккерман (1, n) = n + 2

Аккерман (2, n) = 2n + 3

Аккерман (3, n) = $2^{n+3} - 3$

Аккерман (4, n) = $2^{2^{\cdot^{\cdot^{2}}}} - 3$ с n + 2 вложенными степенями.

Реализация вычислений вначале достаточно проста, так как только первый аргумент, m – 1, рекурсивного вызова в третьей строке определения должен заноситься в стек для исключения рекурсии. Сначала, действительно, можно написать

```
программа Аккерман: ЦЕЛ (аргументы m, n : ЦЕЛЫЕ)
    пока m . ≠ 0 повторять
        если n = 0 то
            m ← m – 1;
            n ← l
        иначе
            {порядок двух следующих операторов существен}
            n ← Аккерман (m, n – 1);
            m ← m – 1;
    Аккерман ← n + 1
```

В этой версии теперь остается только один явный рекурсивный вызов, исключение которого дает следующая программа:

```
программа Аккерман: ЦЕЛ (аргументы m, n : ЦЕЛЫЕ)
    инициализация–стека;
    Аккерман ← 0;
    повторить
        пока m ≠ 0 повторять
            если n = 0 то
                m ← m – 1;
                n ← 1
            иначе
                засылка (m – 1);
                n ← n – 1;
        если стекпуст то
            {это присваивание будет последним выполняемым}
            Аккерман ← n + 1
        иначе
            m ← выборка; n ← n + 1
    до Аккерман ≠ 0
```

Если внимательно проанализировать выполнение этой программы, можно констатировать все же засылку и выборку большого количества нулей; действительно, стек, необходимый для вычисления функции Аккерман (m, n), должен иметь размер, равный Аккерман (m, n) – 1! Поэтому необходимо попытаться улучшить управлением стеком, по крайней мере избавившись от засылки нулей: для этого достаточно использовать целое чиснулей, представляющее число нулей, которое надо было бы заслать в стек в первой нерекурсивной версии (см. выше). Тогда программа становится такой:

```
программа Аккерман: ЦЕЛ (аргументы m, n : ЦЕЛЫЕ)
    {вторая нерекурсивная версия}
    переменная чиснулей: ЦЕЛ;
    инициализация–стека;
    Аккерман ← 0;
    повторять
        чиснулей ← 0;
        пока m ≠ 0 повторять
            если n = 0 то
                m ← m – 1;
                n ← 1
            иначе
                {это заменяет n последовательных выполнений «иначе»,
                  соответствующих предыдущей версии}
                n ← 0;
                если m = 1 то чиснулей ← n
                иначе для i от 1 до n повторять
                        засылка (m – 1)
        если стекпуст то
            {это заменяет чиснулей последовательных выполнений
              последнего «иначе», сопровождаемого «Аккерман ← n + 1»}
            Аккерман ← n + чиснулей + 1
        иначе
            m ← выборка; n ← n + чиснулей + 1
    до Аккерман ≠ 0
```

Можно еще улучшить алгоритм: доказывается, что стек может содержать в порядке от дна к верхушке только одну строго монотонную последовательность чисел, заключенных между m – 1 и 0. Тогда вместо засылки значений m – 1, m – 2, ... можно засылать в стек число вхождений m – l, m – 2 и т.д. В том случае, когда дополнительно решена проблема кодирования целых чисел, превосходящих обычно представимые числа в используемой машине, жела–. тельно все–таки пропускать этот последний этап, который, действительно, имеет смысл только для m и n, не очень малых.

Оптимальный нерекурсивный алгоритм вычисления функции Аккерман, получаемый нисходящим методом (VI.4), можно найти в [Берри 76].

VI.7. Факториал и К°

Для f_1 и f_2 доказательства получаются непосредственно из рекуррентности, для f_3 достаточно начать с доказательства, что z = y! остается инвариантом в ходе выполнения рекурсивных вызовов; это свойство, которое тоже доказывается рекуррентно, объединяется с условием завершения x = y, чтобы дать z = x!.

Эти три функции, однако, не идентичны: для n < 0 f_2 равно 1, тогда как f_1 и f_3 неопределены (их вычисление неопределенно «зацикливается»).

Заметим, что не претендующая на излишние ухищрения реализация f_1 и f_2 обычно использует стек: нужно сохранить n в связи с умножением, выполняемым *после* рекурсивного вызова (более совершенные методы позволяют, однако, обойтись без стека благодаря некоторым ассоциативным свойствам и т.д.; ср. VI.3.5.3). Для f_3, напротив, замена рекурсии на цикл **пока** без стека осуществляется непосредственно, как для всякой функции (VI.3.5) вида

$F(a, b, ...) =$ **если** $C_0(a, b,...)$ **то** E
иначе если $C_1(a, b,...)$ **то** $F(a, b,...)$
иначе если $C_2(a, b,...)$ **то** $F(a_2, b_2,...)$
иначе если $C_n(a, b,...)$ **то** $F(a_n, b_n,...)$

где E, a_l, b_1, ..., a_2, b_2, ..., a_n, b_n ... – это выражения, зависящие от a, b, ... и вычисляемые без рекурсивных вызовов.

VI.8. Функция 91

Для всякого целого n, положительного, отрицательного или нуля

маккарти (n) = max (n – 10, 91)

Для n > 100 это очевидно, поскольку маккарти (n) = n – 10 ≥ 91. Для 90 ≤ n ≤ 100 имеют

маккарти(n) = маккарти(маккарти(n + 11)) = маккарти(n + 1),

так как n + 11 > 100; из этого вытекает, что

маккарти(90) = маккарти(91) = ... = маккарти (100) = маккарти(101) = 91.

Это позволяет пол_учить отправную точку для рекурсивных рассуждений на отрезке [90 – 11m, 100 – 11m] с m, меняющимся от 0 до +∞; рассуждения, которые в силу того, что эти отрезки образуют разбиение интервала] –∞, 100], доказывают, что

n ≤ 100 ⇒ маккарти (n) = 91

VI.9. Рекурсия и передача параметров

Если в определении функции кадью отвлечься от второго параметра, который никогда не используется в вычислении значения функции, то получают

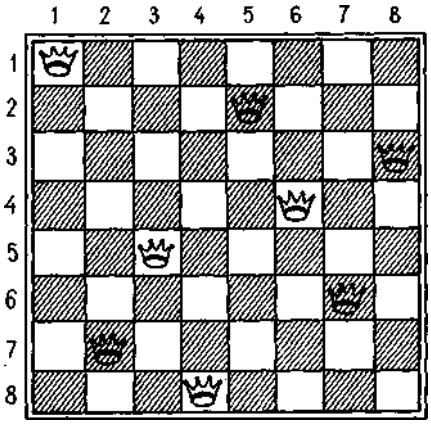
∀y. кадью (x, y) = f(x) с f(x) = **если** x = 0 **то** 1 **иначе** x.f(x – 1)

и обнаруживается просто–напросто функция факториал!

Однако тот факт, что второй параметр не является необходимым при вычислении функции, учитывается, когда он передается именем; когда он передается значением (это единственный из других возможных способов, поскольку речь идет о выражении), программа попытается вычислить второй параметр рекурсивного вызова до его выполнения и не сможет этого сделать, так как это повлекло бы бесконечную последовательность вызовов, первый параметр которых константа, а второй возрастает.

VI. 10. Восемь ферзей

Каждая горизонталь необходимо содержит одного ферзя. Поэтому решение может быть представлено набором восьми номеров вертикалей, занятых соответственно ферзями, располагающимися на горизонталях с первой по восьмую. Так, изображенное ниже решение записывается: (1, 5, 8, 6, 3, 7, 2, 4); это первое из решений задачи, если их классифицировать в лексикографическом порядке, определяемом восьмью номерами вертикалей.



Кажется естественным алгоритм последовательных испытаний, похожий на тот, что был получен в VI.5.4 для «текстов длины 100»: на каждом этапе предполагается, что i ферзей уже правильно размещены на горизонталях 1, 2, ..., i, и надо разместить одного ферзя на горизонтали i + 1 так, чтобы его не могли взять предыдущие ферзи; если это оказывается невозможным, надо вернуться к причине выбора вертикали, соответствующей ферзю горизонтали i.

В рекурсивной форме алгоритм записывается просто. «Параметризация» задачи использует число i горизонталей шахматной доски; точно так же можно было взять число вертикалей. Программа использует внешний массив:

массив вертикаль [1 : 8]: ЦЕЛ

и записывается в виде рекурсивной подпрограммы е результатом ЛОГ, например Гаусс; если решение существует, то вызов

s ← Гаусс (1)

выдает результат **истина,** и это решение задается значениями вертикаль[1], вертикаль[2], ..., вертикаль[8], указывающими номера вертикалей, на которых расположены ферзи горизонталей 1, 2, ..., 8.

```
программа Гаусс: ЛОГ (аргумент i : ЦЕЛ) {рекурсивная версия}
| {инвариант вызова: предполагается, что 1 ≤ i ≤ 9 и что позиции (j, вертикаль
| [j]] для 1 < j < i взаимно не бьются}
| {определить, существует ли решение, при котором ферзи i – 1 первых
| горизонталей размещены в вертикалях соответственно вертикаль [1], ...,
| вертикаль [i – 1]}
| если i = 9 то
|     | Гаусс ← истина {решение всей задачи целиком}
```

```
    иначе
        {определить адекватный выбор для вертикали}
        вертикаль[i] ← 0; Гаусс <– ложь;
        повторять {испытать другую вертикаль}
            вертикаль [i] ← вертикаль [i] + 1;
            если ферзь в позиции [i, вертикаль [i]]
                не бьется другими ферзями в позициях [j, вертикаль [j]]
                для 1 ≤ j ≤ i то
            Гаусс ← Гаусс (i + 1)
        до Гаусс {успех} или вертикаль [i] = 8 {неудача}
```

Условие «ферзь в позиции [i, вертикаль [i]] не бьется другими ферзями в позициях [j, вертикаль [j]] для 1 < j < i» играет ту же роль, что правильное–расширение в задаче о текстах длины 100; его подробная запись не представляет никакой трудности (нужно проверить вертикаль и диагонали, содержащие позицию [i, вертикаль [i]]), и мы предоставляем это читателю.

Легко получается нерекурсивная версия:

```
программа Гаусс: массив вертикаль [1:8]: ЦЕЛ {нерекурсивная версия}
    {нет необходимого аргумента}
    переменные i, j : ЦЕЛ
    {номер последней горизонтали, для которой была выбрана позиция}
    i ← l;
    повторять
        для j от 1 до 8 повторять вертикаль [j] ← 0;
        повторять
            {испытать другую вертикаль для горизонтали i}
            вертикаль [i] ← вертикаль [i] + 1
        до вертикаль [i] > 8 или ферзь в позиции [i, вертикаль [i]] не бьется
                                другими ферзями в позициях [j, вертикаль [j]]
                                для 1 ≤ j < i;
        если вертикаль [i] = 8 то
            {временная неудача, отступить}
            вертикаль [i] ← 0;
            i ← i – 1
        иначе
            {временный успех, продолжать}
            i ← i + 1
    до i = 0 или i = 9
    {если i = 0, то не найдено никакого решения и программа выдает нуль в
    массив «вертикаль»; если i = 9, то решение найдено и определяется
    массивом «вертикаль», никакой элемент которого не является нулевым}
```

Чтобы ответить на вопрос б), достаточно в этой нерекурсивной версии заменить цикл **повторять** ... **до** i = 0 **или** i = 9 двойным циклом, позволяющим получить не одно, а все решения:

```
повторять
    повторять
        ...
    до i = 0 или i = 9
    если i = 9 то
        писать вертикаль [1 : 8];
        i ← i – 1 {моделируется неудача}
до i = 0
```

Соответствующая модификация столь же непосредственно получается в рекурсивной версии.

Для ответа на вопрос б) нет необходимости модифицировать программу Гаусс: подпрограмма, определяющая, правильно ли «расширение», включает проверку, позволяющую отбросить повторяющиеся конфигурации, получающиеся поворотом или симметрией из тех, что были рассмотрены ранее в лексикографическом порядке решений (это тот порядок, в котором решения рассматриваются программой Гаусс).

VI.11. Коды Грея

Задача кодов Грея может быть решена алгоритмом последовательных испытаний, похожим на алгоритм работы с «текстами длины 100» (VI.5.4) или «восемью ферзями» (предыдущее упражнение), но более сложным из–за нетривиального определения «эквивалентности» двух кодов. Отправляясь от кода, у которого определено только первое слово (например, 0000), выполняют последовательные испытания, чтобы на каждом этапе расширить код новым словом, получающимся заменой одного бита.

Подпрограмма, которой поручена проверка правильности расширения, должна здесь обеспечивать не только отличие получаемого слова от предыдущих, но также и то, что шестнадцатое отличается от первого значением единственного бита.

Можно уменьшить число проверяющих испытаний и упростить отбрасывание эквивалентных кодов, обоснованно выбрав первые слова кода. Можно выбрать 0000 и 0001 в качестве двух первых слов без всякой потери общности, выполняя в случае необходимости перестановку слов кода или перестановку битов каждого слова. Третье слово тогда имеет вид $x_1x_2x_3l$, где только одно x_1 равно 1; тогда 0011 можно получить ценой одной возможной дополнительной перестановки битов. Действительно, можно доказать, что пять первых слов могут быть зафиксированы без потери общности:

0000 0001 0011 0111 0110

Существуют 11 кодов–решений.

VI.12. Беседы

Выразим общее число возможных бесед с помощью

беседы (0, 7. 5, 8, 4, 6, 7, 6, 7)

где беседы(i, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8) есть общее число возможных бесед, если известно, что восемь собеседников имеют соответственно n_1 n_2, ..., n_8 возможностей вступить в разговор и что последним говорившим был собеседник с номером i ($1 < i \le 8$, за исключением начального вызова). Получают рекурсивную формулу, удобно интерпретируемую как обход дерева, если рассматривать вновь получаемую ситуацию как результат последовательного включения первым в беседу каждого из собеседников, который может это сделать:

беседы (i, n_1, n_2, ..., n_8) =

если $\sum_{1\le j\le 8} n_j = 0$ **то** 1 {пустая беседа или конец беседы: одна возможность}

иначе если $\sum_{\substack{1\le j\le 8\\ j\ne i}} n_j = 0$ **то** 0 {единственно последний собеседник имеет что-либо сказать: невозможна никакая беседа}

иначе $\sum_{\substack{1\le j\le 8\\ j=i\\ n_j>0}} \textit{беседы}(j, n_1, n_2, \ldots n_{j-1}, n_j - 1, n_{j+1}, \ldots, n_8)$

{исследовать дерево возможностей, пытаясь по очереди включить первым в беседу каждого из тех, кто может это сделать}

Вторая ветвь альтернативы фактически бесполезна: это частный случай третьей ветви.

Описанный выше алгоритм в реальности неприменим для данных значений: он перечисляет, он *моделирует* все перечисляемые беседы; однако число их огромно. Для улучшения алгоритма можно применить компромисс «пространство–время»; (VII.1.2.4). Поскольку многочисленные «эквивалентные» ситуации анализируются в ходе выполнения алгоритма, можно на пределе доступной памяти запоминать максимальное количество встретившихся уже конфигураций и связанных с ними значений беседы, чтобы не повторять каждый раз просмотр соответствующего поддерева. Этот метод, представляющий собой частный случай «восходящих вычислений» рекурсивных программ (VI.4) называется также **динамическим программированием** [Axo 74].

Другой возможностью является, очевидно, нахождение аналитического решения. Авторы его не знают и были бы признательны, если бы им его указали.

Замечание: Та же задача допускает и иную формулировку: семейства Седёзи (n_1 человек), Ибеэм (n_2 человек) и т.д. приглашены на ужин семейством Седесэ[1] (n_8 человек). Каким числом способов мог бы глава семьи г-н Седесэ разместить своих гостей так, чтобы два члена одного и того же семейства никогда не располагались рядом друг с другом?

Глава VII

VII. 1. O в математике

Речь должна идти, конечно, не об «операторе» O, а об отношении эквивалентности функций «f порядка g», обозначаемом O(f) = O(g) или в силу языковых злоупотреблений f = O(g), если g есть некоторая классическая функция (возведение в степень, логарифмы, показательная функция и т.д.). Это отношение удовлетворяет обычным свойствам эквивалентности (рефлексивность, симметрия, транзитивность), так что

$$O(f) = O(g) \Rightarrow O(f + g) = O(f) = O(g)$$

$O(k.f) = O(f)$, если k есть константа или, в более общем виде, функция O(1)

$$\left.\begin{array}{l} O(f) = O(f') \\ O(g) = O(g') \end{array}\right\} \Rightarrow O(f.g) = O(f'.g')$$

Заметим, что это отношение эквивалентности не сохраняется при «переходе к пределу» в *простой* сходимости функций, т.е. если функции $f_1, f_2, \ldots, f_i \ldots$ все порядка g и последовательность f; стремится в каждой точке к функции f, когда i стремится к бесконечности, то f не обязательно имеет порядок g; так, функции $f_i(n) = n/i$ все порядка O(n), тогда как их предел есть тождественный нуль. То же самое можно сказать о сходимости *«по норме»*. Напротив, отношение $O(f_i) = O(g)$ сохраняется при переходе к пределу, если сходимость *равномерная.*

VII.2. Минимум и максимум одновременно

Алгоритм можно описать:

[1] Нетрудно заметить, что в звучании этих фамилий пародируются названия известных западных фирм–производителей *ЭВМ. – Прим. перев.*

```
программа Минимакс (аргумент массив T [1 : n] : ЭЛЕМЕНТ;
                    результаты m, M: ЭЛЕМЕНТЫ)
  переменные m1 m2, Ml, M2 : ЭЛЕМЕНТЫ;
  если n = 1 то
      m ← T[1];
      M ← T[1]
  иначе если n = 2 то
      если T[1] < T[2] то
          m ← T[1];
          M ← T[2]
      иначе
          m ← T[2];
          M ← T[1]
  иначе
      Минимакс(T[1 : 2], m1, M1);
      Минимакс(N[3 : n], m2, M2);
      m ← если m1 > m2 то m1 иначе m2;
      M ← если M1 < M2 то M1 иначе M2
```

Если через C(n) обозначить число необходимых сравнений при выполнении этой программы, получим

$$\begin{cases} C(1) = 0 \\ C(2) = 1 \\ n > 2 \Rightarrow C(n) = C(2) + C(n-2) + 2 = C(n-2) + 3 \end{cases}$$

Эта рекуррентность решается, например, путем отдельной обработки случаев «n четное» и «n нечетное»; так, находят формулы C(2k) = 3k – 2 и C(2k + 1) = 3k, которые объединяются в виде

$$C(n) = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Можно удивляться тому, что здесь не применен принцип равновесия; если практически воспользоваться здесь этим принципом, который состоит в замене двух рекурсивных вызовов *Минимакс* на

Минимакс (T[1 : $\left\lfloor \frac{n}{2} \right\rfloor$], m_l, M_1);

Минимакс (T[$\left\lfloor \frac{n}{2} \right\rfloor$ + 1 : n], m_2, M_2)

то отношение рекурсивности принимает вид

$$n > 2 \Leftarrow C'(n) = C'(\left\lfloor \frac{n}{2} \right\rfloor) + C'(\left\lceil \frac{n}{2} \right\rceil) + 2$$

Решение его, конечно, менее просто, чем в предыдущем случае, и, полагая $n = 2^k + \ell$ с $0 \le \ell \le 2^k$ получаем

$$C'(n) = 3.2^{k-1} + \ell + \mathrm{Max}(2^{k-1}, \ell) - 2$$

Из этого можно вывести $C(n) \le C'(n) \le \frac{5n}{3} - 2$; значение $\frac{3n}{2} - 2$ достигается для $n = 2^k$ и только в этом случае, тогда как значение $\frac{5n}{3} - 2$ достигается для $n = 2^k + 2^{k-1}$ и только в этом случае. Тот факт, что равновесие здесь не рекомендовано, не находится в противоречии с разд. VII.1.2.2, касающимся алгоритмов, сложность которых

подчиняется рекуррентному отношению вида

$$T(n) = T(n') + T(n - n') + O(n)$$

Если в этом случае и желательно принять $n' = \left\lfloor \frac{n}{2} \right\rfloor$ то это совсем не так, когда рекуррентное отношение имеет общий вид

$$T(n) = T(n) + T(n - n') + O(1)$$

VII.3. Двигаясь от середины

Упомянутая возможность действительно заманчива ... но осторожно ! Правильный доступ к элементам массива возможен, только если в распоряжении имеется функция расстановки, устойчивая с точки зрения отношения порядка на ключах, т.е. такая, что для двух произвольных ключей c и c'

$$c < c' \Rightarrow f(c) \le f(c')$$

Может быть, легче найти такую функцию, если работать не с исходными ключами, а с предварительно преобразованными ключами t(c), где t есть специальное преобразование, построенное для того, чтобы можно было найти такую функцию f. См. [Эмбл 73].

VII.4. k–й наименьший

Следующий алгоритм решает задачу:

```
программа К–й–наименьший: ЭЛЕМЕНТ;
        (аргументы массив t[a : b] :ЭЛЕМЕНТ;
        аргумент k: ЦЕЛОЕ)
  {найти k–й наименьший элемент в t; предполагается, что 1 ≤ k ≤ b – a + 1}
  переменная s : ЦЕЛ;
  s ← Деление (t [a : b])
  {теперь известна позиция (s – a + 1)–го наименьшего элемента, а другие
    «хорошо расположены» по отношению к нему}
  К–й–наименьший ←   если k = s – a + 1 то t[s]
                     иначе если k < s – a + 1 то
                              {поиск в левом подмассиве}
                              К–й–наименьший (t[a : s – 1], k)
                     иначе {k > s – a + 1}
                              {поиск в правом подмассиве}
                              К–й–наименьший (t [s + 1 : b], k – s + a – 1)
```

Его реализация в нерекурсивном виде элементарна:

```
...
переменные i, j, s : ЦЕЛЫЕ;
i ← a; j ← b;
повторять
    | s ← Деление (t[i : j]);
    | если k < s – i + 1 то
    |       | j – s – 1
    | иначе {проверять в обратном направлении бесполезно}
    |       | i ← s + 1;
    |       | k ← k – s + i – 1
до k = 0
К–й–наименьший ←t [s]
```

Существует важная улучшающая модификация этой программы, которая

получается применением принципа уравновешивания и позволяет довести максимальную сложность (здесь $O(n^2)$) до $O(n)$. См. [Ахо 74].

VII.5. Пять последних наносекунд

Интересующая нас часть программы эквивалентна

```
если j след – i след > порог то
    | засылка (i след); засылка (j след);
если j – i ≤ порог то
    | если ~стекпуст то
    |     | j ← выборка;
    |     | i ← выборка
```

Это позволяет сократить бесполезные засылки и выборки в программе

```
если j след – i след > порог то
    | если j – i ≤ порог то
    |     | i ← след;
    |     | j ← след;
    | иначе
    |     | засылка (i след); засылка (j след)
иначе {здесь, j – i < порог, так как j – i ≤ j след – i след}
    | если ~стекпуст то
    |     | j ← выборка;
    |     | i ← выборка
```

Таким образом, можно было бы заменить в фортрановской программе пять присваиваний двумя, что дает достаточно незначительный выигрыш, поскольку он проявляется не более одного раза после каждого вызова Деления.

VII.6. Устойчивость Быстрой Сортировки

Быстрая Сортировка неустойчива, и не существует простых средств сделать ее устойчивой. Причиной тому являются последовательные и чередующиеся обмены Деления. Если, например, выполняется Быстрая Сортировка в версии ФОРТРАНа из разд. VII.3.6.5 на массиве (a_1, a_2, a_3, a_4, a_5), где все элементы равны, с порогом = 2, то получают (a_4, a_5, a_1, a_3, a_2). О поведении одинаковых ключей в Быстрой Сортировке см. в работе [Седжуик 77а].

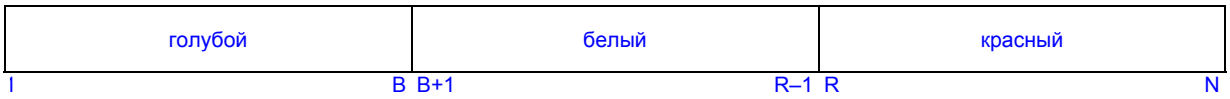
VII.7. Деление заданным значением

```
i ← a; j ← b;
повторять
    | f ← деление (t[i : j]);
    | если ключ (f) ≤ x то i ← f
    | иначе j ← f – 1
до ключ (f) ≤ x и ключ(f + 1) ≥ x
```

VII .8. Французский флаг

Конструкция программы непосредственно переписывается с Деления в VII.3.6.4. Завершающей желаемой ситуацией является существование целых чисел B и R, проверяющих следующее свойство (C):

$$(C)\begin{cases} 0 \le B \le R \le N+1 \\ \text{для всякого } i, \text{ такого, что } 1 \le i \le B, \text{ цвет}(i) = \text{синий} \\ \text{для всякого } i, \text{ такого, что } B < i < R, \text{ цвет}(i) = \text{белый} \\ \text{для всякого } i, \text{ такого, что } R \le i \le N, \text{ цвет}(i) = \text{красный} \end{cases}$$

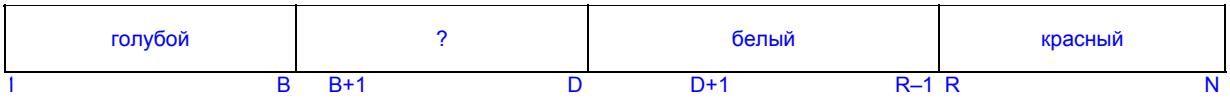


Обратите внимание на исключительную важность выбора границ в этих неравенствах и использование $<$ или $\le$; другие комбинации сделали бы решение невозможным в некоторых случаях (когда представлены не все цвета) или неопределенным в других случаях.

Как и в VII.3.6.4, исходя из (C), легко получается инвариант цикла (C'), представляющий собой ослабленный вариант (C), путем «раздвоения» двух из условий (C) благодаря новому целому D; (C') вновь дает (C) при $B = D$:

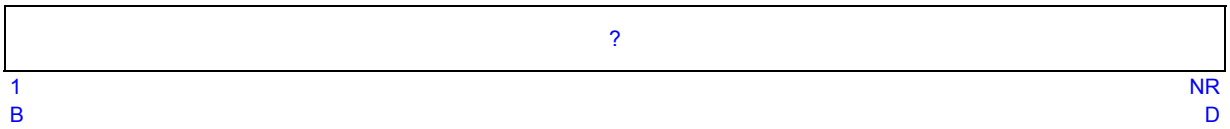
$$(C)\begin{cases} 0 \le B \le D < R \le N+1 \\ \text{для всякого } i, \text{ такого, что } 1 \le i \le B, \text{ цвет}(i) = \text{синий} \\ \text{для всякого } i, \text{ такого, что } D < i < R, \text{ цвет}(i) = \text{белый} \\ \text{для всякого } i, \text{ такого, что } R \le i \le N, \text{ цвет}(i) = \text{красный} \end{cases}$$

Это заставляет рассматривать, кроме областей «синяя», «белая» и «красная», еще одну область «?», элементы которой имеют пока неизвестный цвет:



При инициализации эта область «?» покрывает целиком массив

$B \leftarrow 0$; $D \leftarrow N$; $R \leftarrow N + 1$



После этой инициализации программа состоит из одного цикла:

пока $B < D$ **повторять**
  | «уменьшить $D - B$, сохраняя правильность (C')»
{здесь (C') – истинно и $B = D$, следовательно, (C) есть истина}

Действие «уменьшить...» состоит в рассмотрении элемента $t[D]$ и определении его судьбы в зависимости от цвета:

выбрать
  | цвет (D) = белый: $D \leftarrow D - 1$
  | цвет (D) = голубой:
  | | $B \leftarrow B + 1$;
  | | переставить (B, D)
  | цвет (D) = красный:
  | | $R \leftarrow R - 1$; {здесь, в силу (C), $D < R$}
  | | переставить (D, R)
  | | $D \leftarrow D - 1$

VII.9. Распознавание без пополнения

Предположим, что массив переход представляет диаграмму неполного дерева. Тогда алгоритм распознавания может использовать массив неудача: достаточно заменить в подпрограмме распознавание (VII.4.2)

состояние ← переход [состояние, t[i]]

на

```
x ← переход [состояние, t[i]];
пока x = 0 и состояние ≠ 0 повторять
        | состояние ← неудача [состояние];
        | x ← переход [состояние], t [i]];
состояние ←x
```

Несмотря на цикл **пока ... повторять** ..., сложность этого алгоритма остается равной O(n), где n – длина обрабатываемого текста; действительно, действие состояние ← переход[состояние, t[i]], выполняемое n раз заставляет увеличиваться не менее чем на 1 глубину в дереве той вершины, которая связана с состоянием; действие состояние ← неудача [состояние], которое уменьшает ту же самую глубину по крайней мере на 1, не может, таким образом, выполняться более чем n раз на множестве просмотров текста t.

VII.10. Указатель

Надо использовать таблицу T, где располагаются индексированные слова; эта таблица управляется, например, ассоциативной адресацией (VII.2.5) и позволяет поставить в соответствие всякому встречающемуся в ней терму x некоторый линейный список, обозначаемый вхождения(x) и содержащий все номера страниц, на которых появляется x. Этот список управляется как *файл* (включение в хвост).

Алгоритм записывается в виде

```
построить таблицу T;
сортировать таблицу T {в алфавитном порядке};
для x из T повторять
      | печать x, вхождения (x)
```

Для составления таблицы T считывают данные, которые, напомним, представляют собой последовательность номеров страниц и индексируемых термов на каждой из этих страниц. Предположив, что эти данные заканчиваются псевдо–«номерами страницы» 0, получим

```
{заполнение таблицы T, изначально пустой}
читать x;
пока x ≠ 0 повторять
      | читать y;
      | пока y не является номером страницы повторять
      |       | включение y в T {если там его нет}
      |       | включение x в файл вхождения (y);
      |       | читать y;
      | x ← y
```

Этот алгоритм можно улучшить многочисленными способами. Так, чтобы составить подпрограммы или чтобы учесть термы, встречающиеся на нескольких последовательных страницах, нужно, например, печатать 11–16, а не 11, 12, 13, 14, 15, 16. Заметим, что предлагаемый метод позволяет подготовить указатель до. того, как станет известной окончательная нумерация страниц; поскольку, как отмечалось, термы индексируются в порядке их появления с включением указаний, относящихся к их

распределению по страницам, то достаточно заменить в таком данном старые сведения о страницах на новые, когда определяется окончательная нумерация страниц.

ПРИЛОЖЕНИЯ

Приложение А

Алгоритмическая нотация

В этом приложении сведены различные конструкции нотации высокого уровня, которая использовалась в нашей книге для представления алгоритмов. Каждая из конструкций сопровождается примерами и напоминанием места, где эта конструкция встретилась впервые.

Базовые конструкции (гл. II)

Общие сведения (II.2)

- Формат: свободный (стр. 42 т.1):
- Литеры: строчные и прописные.
- Ключевые слова: полужирным шрифтом (стр. 43 т.1): **пока программа** и т.д.
- Комментарии: в фигурных скобках (стр. 42 т.1): {комментируем, как комментируют комики коми на Каме}

Типы и значения (II.2)

- Типы, обозначаемые прописными буквами (стр. 43 т.1): ЦЕЛОЕ СТРОКА ЛИТЕРА ЛОГИЧЕСКОЕ ВЕЩ ЦЕЛОЕ НАТУРАЛЬНОЕ (базовые типы) и типы, определяемые по желанию программиста (ср. ниже).
- Целые константы: обычные обозначения:

  25 – 7654
- Вещественные константы: обычное математическое обозначение, но с точкой вместо десятичной запятой (стр. 43 т.1):

  76.54 – 8.5904

Константы типа ЛИТЕРА или СТРОКА: в кавычках (стр. 43 т.1):

"А"

"А РОЗА УПАЛА НА ЛАПУ АЗОРА"

Переменные, массивы, символические константы, объявления (II.2)

- Идентификаторы: произвольное число литер; внутренние пробелы разрешены, но часто заменяются дефисами:

  а и zztluv переход по метке переход–по–метке
- Обязательные объявления для переменных и массивов (стр. 43 т. 1).

- Объявления переменных или массивов: ключевые слова **переменная** или **массив**. Для массива уточняются границы каждого измерения (стр. 44):
  переменная uuu: ЛИТЕРА;
  массив t [0 : 50]:ВЕЩ;
  массив эрнестина [–5 : 12, 0 : 10, 1 : 20] : ЦЕЛ
- Последовательные объявления: разделение с помощью точки с запятой (стр. 43 т.1).
- Сгруппированные объявления: использование множественного числа (и родовых форм) (стр. 43 т.1):
  переменные a, b, hhh : ЦЕЛЫЕ;
  массивы a[1 : 10], в[1 : 20] : ВЕЩЕСТВЕННЫЕ;
  переменные немного, очень, страстно: ЛОГИЧЕСКИЕ,
  совсем нет: ВЕЩЕСТВЕННАЯ
- Символические константы (стр. 43 т. 1):

константы пи = 3.1415926,
хороший стих = "БЕЛЕЕТ ПАРУС ОДИНОКИЙ"

- Элементы массива (стр. 52 т.1):
t[e_1] эрнестина [e_2, e_3, e_4] a [e_5] b[e_6]

где e_1, e_2, e_3, e_4, e_5, e_6 – это объекты типа ЦЕЛ (константы, переменные, выражения и т.п.), значения которых соответствуют границам, установленным для каждого измерения массива.

Выражения (II.2)

- Арифметические выражения: обычные обозначения (стр. 43 т. 1):

$$65a + \frac{25u - 40c}{3}$$

$$3.72 \times (a + b)^7$$

- Логические выражения: логические операции **и**, **или**, ~ (**не**); операторы отношений <, >, ≤, ≥, =, ≠
- Выражения типа СТРОКА: операция конкатенации || (стр. 43 т.1);
  выделение подстроки: Т (начало | длина)
- Условные выражения: **если** с **то** e_1 **иначе** e_2, где с – логическое выражение (стр. 43 т.1).

Базовые операторы (II.2)

- Присваивание (стр. 44 т.1):
  переменная ← выражение
- Последовательное чтение (стр. 44 т.1):
  читать (ф) nepl, пер2, ..., перn
- где ф обозначает внешнее устройство или файл; если не возникает двусмысленности, пишут просто
  читать nepl, пер2, ..., перn
- Последовательная запись (стр. 44 т.1):
  писать(ф) выр1, выр2, ..., выpn
  или, если нет разночтения, то допускается запись:
  писать выр1, выр2, ..., выpn

Управляющие структуры (гл. III)

Ниже используются обозначения:

c, c_1, c_2, ..., c_n обозначают логические выражения;

A, A_1, A_2,..., A_n – операторы;

m, n и p – объекты типа ЦЕЛ (константы, переменные, выражения и т.п.);

E – множество.

Циклы (III.3.1)

- Бесконечное повторение (стр. 80 т.1):

  повторять бесконечно

- Цикл «пока» (стр. 81 т.1):

  пока c **повторять**
  | A

- Цикл «до» (стр. 82 т.1):

  повторять
  | A
  до c

- Циклы с параметром (стр. 82 т. 1):

  для x **из** E **повторять**
  | A
  для x **из** E **пока** c **повторять**
  | A
  для x **от** m **до** n **шаг** p **повторять**
  | A
  для x **от** m **до** n **повторять**
  | A

В последнем случае шаг принимается равным 1.

- Цикл с многозначным недетерминированным выбором «цикл Дейкстры», (стр. 85 т.1):

  выбрать и повторять
  | c_1 : A_1,
  | c_2 : A_2,
  | c_3 : A_3,
  | ...
  | c_n : A_n,

Ветвления (III.3.2)

- Альтернатива (стр. 83 т. 1):

  если c **то**
  | A_1
  иначе
  | A_2

  или просто:

  если c **то**
  | A

- Многозначный переключатель (не детерминированный) (стр. 84 т.1):
  выбрать
  | $c_1 : A_1$
  | $c_2 : A_2$
  | ...
  | $c_n : A_n$
  иначе
  | A

Цепочка (III.3.3)

- Использование точки с запятой (стр. 86 т.1):
  $A_1; A_2; A_3$
- Использование вертикальной черты для построения составного оператора (стр. 86 т.1):
  | $A_1; A_2;$
  | $A_3; A_4$

Композиция этих различных структур (III.3.5)

- С помощью смещений и вертикальных черт (стр. 88 т.1):
  A;
  пока c **повторять**
  | **если** c_1 **то**
  | | $A_1;$
  | | A_2
  | **иначе**
  | | $A_3;$
  | **повторять**
  | | A_4
  | **до** c_2

Подпрограммы (гл. IV)

Определение подпрограммы (IV.2.3)

- Подпрограмма «оператор» (стр. 126 т.1):
  программа ппп (список объявлений параметров)
  | тело подпрограммы ппп

  Часть (список объявлений параметров) может отсутствовать.
- Подпрограмма «выражение» (стр. 126 т.1):
  программа ффф: ЦЕЛ (список объявлений параметров)
  | тело подпрограммы ффф,
  | где имя ффф воспринимается как имя переменной (здесь типа ЦЕЛ),
  | которая должна получить значение до конца выполнения
  | подпрограммы

  Часть (список объявлений параметров) может отсутствовать.
- Список объявлений параметров (стр. 126 т.1): последовательность объявлений, сходных с рассматривавшимися выше, но с заменой ключевого слова переменная на одно из трех ключевых слов: аргумент, результат или модифицируемое данное. Для параметра, являющегося массивом, эти ключевые слова добавляются к слову массив:
  программа x (**аргументы** a, b : ЦЕЛЫЕ,
  c, d: ВЕЩЕСТВЕННЫЕ;

модифицируемое данное массив t [– 5 : + 5] : СТРОКА)

- Тело подпрограммы (стр. 126 т.1): последовательность операторов, которой, возможно, предшествуют объявления локальных переменных.

Вызов подпрограммы (IV.2.3)

- p, если описание p не содержит списка параметров;
- p(пар$_1$, ..., пар$_n$) в противном случае, пар$_1$, ..., пар$_n$ – это фактические параметры, соответствующие формальным параметрам в описании p, включая соответствие типов (стр. 126 т.1). Фактический параметр, соответствующий формальному параметру аргумент, может быть переменной, выражением, константой или элементом массива и т.д.; фактический параметр, соответствующий результату или модифицируемому данному, должен быть объектом, способным изменять значение (переменная, элемент массива, формальный параметр вызываемой программы.
- Рекурсивный вызов: отмечается *курсивом* (стр. 7 т.2).
- Возврат из подпрограммы в вызвавшую программу в конце ее выполнения (без явного оператора «возврата»).

Тип «подпрограмм» (IV.2.4)

Объект «подпрограмма» имеет тип, определяемый типами своих аргументов и результатов (заменяемых на ПУСТО, когда параметры отсутствуют):

(ЦЕЛ × (ЦЕЛ × ВЕЩ) → ВЕЩ × ЛИТЕРА)

Сопрограммы (IV.7.2)

- определение: аналогичное определению подпрограммы с заменой ключевого слова **подпрограмма** на ключевое слово **сопрограмма** (стр. 173 т. 1).
- вызов: либо типа «подпрограммы» (ср. выше) с неявным оператором «возврата» (для начального вызова множества сопрограмм);
- либо типа «сопрограммы» с помощью ключевого слова **возобновить** (стр. 173 т. 1):
  возобновить сп (пар$_1$ ..., пар$_n$)

Сложные типы (гл. V)

Определение новых типов (V.1.3)

тип TTTT = ...

- соединение (стр. 185 т. 1):
  тип T = (α : U; β : V; γ : W)
- разделение вариантов (стр. 190 т. 1):
  тип TT = (A | B)
- перечисление (стр. 186 т. 1):
  тип TTT = (a, b, ..., ℓ)
  где a, b, ..., ℓ константы одного типа.
- возможная рекурсивность (стр. 189 т.1).

Доступ к компоненте объекта сложного типа (V.1.3)

- соединение (стр. 185 т. 1):

α (x) β(x) γ(x)

- разделение вариантов: к компоненте обращаются только с помощью конструкции выбрать, если она определена (стр. 189 т.1)

Константы сложного типа (V.1.3)

- соединение (стр. 191 т.1):
  T(u, v, w) (u типа U, v типа V, w типа W)
- разделение вариантов (стр. 191 т.1):
  TT(a)
  TT(b)
  (a типа A, b типа B).

Приложение Б

Англо–Франко–Русский словарь основных терминов программирования

Actual (parameter)	(paramètre) réel	фактический (параметр)
Add	ajouter	сложить
Address	adresse	адрес
Alphanumeric	alphanumérique	алфавитно–цифровой
Alternative	choix	альтернатива
Array	tableau	массив
Assembly, assembly language	assemblage, langage d'ssemblage	ассемблирование, ассемблер, язык ассемблирования
Assign, assignment	affecter, affectation	присвоить присваивание
Balanced	équilibré (arbre)	равновесное (дерево)
Binary	binaire	двоичный
Binary search	recherche dichotomique	дихотомический поиск («делением пополам»)
Boolean	logique, booléen	логический, булев
Bottom–up	ascendant	восходящий, возрастающий
Bound	limite, borne	граница, предел
Byte	caractère, octet	байт, литера
Call	appel, appeler (un sousprogramme)	вызов, вызвать (подпрограмму)
Ceiling	plafond (fonction $\lceil\ \rceil$)	$\lceil x \rceil$ – наименьшее целое, большее или равное x
Character	caractère	литера
Check, checking	verifier, verification	проверять проверка, верификация
Column	colonne (d'unecarte ou d'un tableau)	столбец, колонка (карты или массива)
Comment	commenter, commentaire	комментировать, комментарий
Compile, compiler	compiler, compilateur	транслировать, транслятор, компилятор
Complete	complet; plein (pour un arbre binaire)	полное (двоичное дерево)
Compute, computation, computer	calculer, calcul, ordinateur	вычислять, вычисление, ЭВМ
Computer science	informatique (en tant que science)	информатика
Connected	connexe (graphe)	связный (граф)
Control	commande	управление
Control structure	structure de contrôle	управляющая структура
Core (memory)	mèmoire centrale	оперативная память
Dangling reference	référence folle	висячая ссылка
Data	donnée, information	данное, информация
Data processing D. P.	informatique (en tantque technique)	обработка данных
Data set	fichier (physique)	набор данных, файл
Data structure	structure de données	структура данных
Delete, deletion	détruire, destruction	стереть, уничтожить стирание, уничтожение

Deque (double–ended queue)	file a double èntrée	файл с двойным доступом
Digit	chiffre décimal	десятичная цифра
Disk	disque magnétique	магнитный диск
Display	montrer, presenter; terminal à écran	выдать; терминал с экраном
Drum	tambour magnétique	магнитный барабан
Dummy (variable, parameter)	(variable) muette, (parametre) formel	свободная (переменная) формальный (параметр)
Edge	sommet (d'un graphe)	вершина (графа)
Empty	vide	пустой
Entrance, entry	enrée (d'un programme)	вход (программы)
Entry point	point d'entrée (d'un programme, d'une structure de données)	точка входа (в программу, в структуру данных)
Exchange	échanger, échange	обменивать, обмен
Exit	sortie	выход
FIFO stack (first–in, first–out)	file, file d'attente	файл, очередь
File	fichier	файл
Finite	fini	конечный
Fixed point	virgule fixe	фиксированная запятая (в представлении вещественных чисел)
Flag	sentinelle, marque, indicateur, «top»	метка, указатель
Floating point	virgule flottante	плавающая запятая
Floor	plancher (fonction $\lfloor\_\rfloor$)	$\lfloor x \rfloor$ – наибольшее целое, меньшее или равное x
Flow chart	organigramme, schema de programme	блок–схема
Formal (parameter)	paramètre formel	формальный параметр
Front	tête	голова (файла)
Function subprogram	sous–programme «expression»	подпрограмма–«выражение», функция
Garbage collector collection	ramasse–miettes, ramassage de miettes	сборщик мусора
Graph	graphe	граф
Hardware	matériel, machine	техническое обеспечение ЭВМ
Hash–coding, hashing	adressage associatif	ассоциативная адресация, перемешивание
Hash function	fonction d'adressage	адресная функция, функция размещения
Heap	tas (en gestion des structures de données), maximier (arbre binaire de tri)	куча (в управлении структурами данных); максимизирующее дерево (двоичное дерево сортировки)
Height	hauteur	высота; глубина дерева
Identifier	identificateur	идентификатор
Implementation	réaliser (–ation), mettre (mise) en oeuvre	реализовать, реализация
Infinite, infinity	infini, l'infini	бесконечный, бесконечность
Input	entrée	вход, ввод
Integer	entier	целое
Interpret, interpreter	interprèter, interpréte	транслировать (интерпретировать), интерпретатор
Job	travail	работа, задание
Keypunch	perforatrice (machine)	перфоратор
Keyword	mot–clé	ключевое слово
Label label(l)ed	étiquette, étiqueté (arbre)	метка, помеченное (дерево)
Language	langage	язык
Leaf	feuille	лист
LIFO stack (last–in, first–	pile	стек, магазин

out)		
Line	ligne	строка
Linear	linéaire	линейный
Link	lien, chainage, pointeur	связь, звено, указатель
Linked structure	structure chainée	цепная структура
List	liste	список
Load	charger	загрузить
Logical	logique	логический
Loop	boucle	цикл
Macro generateur	générateur de macroinstractions	генератор макрокоманд
Mantissa	mantisse	мантисса
Matrix (pi. matrices)	matrice	матрица
Merge, merging	fusionner, fusion	слить, слияние
Name (call by)	(appel par) nom	имя (вызов по имени)
Nest, nesting	imbriquer, imbrication	включать, вложение
Node	noeud (d'un arbre, d'un graphe)	гнездо, вершина (дерева, графа)
Null	vide, nil (pointeur)	пустой, нулевой (указатель)
Number	nombre, numéro	число, номер
Number system	système de numération	система счисления
Operate	exploiter, faire fonctionner	эксплуатировать, запускать
Operating system	système d'exploitation	операционная система
Optimize, optimizer	optimiser, optimiseur	оптимизировать, оптимизатор
Output	sortie, sortir (des résultats)	выход, вывод выводить (результаты)
Overflow	dépassement de capacité	переполнение
Path	chemin (dans un arbre, un graphe, etc.)	путь (в дереве, графе и т.д.)
Peripheral unit, device	(unité) périphérique	периферийное, внешнее (устройство)
Pointer	pointeur	указатель, пойнтер
Pop, pop up	dépiler	выбрать, взять (со стека)
Power	puissance (d'un nombre)	степень (числа)
Prime (number)	(nombre) premier	простое (число)
Print, printer	imprimer, imprimante	печатать, печатающее устройство
Process, processing	traiter, traitement	обрабатывать, обработка
Programming	programmation	программирование
Proof	démonstration	доказательство
Property	propriété	свойство
Punch, punched	perforer, perforé (e)	перфорировать, отперфори–рованный (–ая)
Push, push down	empiler	заслать (в стек)
Pushdown list	pile	стек
Queue	file, file d'attente	файл, очередь
Radix (pi. radices)	base (d'un systeme de numeration)	основание (системы счисления)
Read, readable, readability	lire, lisible, lisibilité	читать, читаемый, воспринимаемый читаемость, восприятие
Real	réel; fractionnaire	вещественный; дробный
Rear	arrière (d'une file d'attente)	хвост (очереди)
Recognize, recognition	reconnaître, reconnaissance	распознавать, распознавание
Record	enregistrement, enregistrer	запись (структуры данных), записывать

Reference	référence, pointeur	ссылка, указатель
Result	résultat	результат
Root	racine	корень
Row	rangée, ligne (d'un tableau)	выровненная (упорядоченная смежным размещением) строка
Run	executer, faire passer (un programme); «tourner», s'executer	выполнить (программу)
Scale	échelle	масштаб; масштабный множитель
Search, searching	chercher, recherche	искать, поиск
Sequence, sequencing	suite, enchaînement	последовательность, цепочка
Set	ensemble	множество, набор
Set–theoretic	ensembliste, appartenant à la théorie des ensembles	теоретико–множественный
Shift, shifting	décaler, décalage	смещать, сдвигать смещение, сдвиг
Software	logiciel	программное (математическое) обеспечение
Sort, sorting	trier, tri	сортировать, сортировка
Space	espace	пространство, место
Sparse	creux (tableau), creuse. (matrice)	ненасыщенный, разреженный (массив) ненасыщенная разреженная (матрица)
Split	partager, diviser	делить, разделять, обобществлять: разделение (времени), но обобществление (данных)
Stack	pile, empiler	стек засылать в стек
State	état	состояние
Store	ranger (en mémoire le contenu d'un registre, un résultat, etc.)	разместить, поместить (в памяти содержимое регистра, результат и т.д.)
Storage	rangement; memoire	размещение; память
String	texte, chaîne (de caractères)	строка, текст, цепочка литер
Sub–	sous–	под–
subarray	sous–tableau	подмножество
subset	sous–ensemble	подмассив
substring	sous–texte	подстрока
subtree	sous–arbre	поддерево
subroutine	sous–programme «instruction»	подпрограмма «оператор»
Subscript	indice (dans un tableau)	индекс (в массиве)
Subtract	soustraire	вычесть
Swap	échanger	обменивать
Tape (magnetic, punched)	bande (magnétique) ruban (perforé)	лента (магнитная), перфолента
Time	temps	время
Top–down	descendant	нисходящий, убывающий
Traversal	parcours (d'une structure de données, en partic d'un arbre)	просмотр, обход (структуры данных, в частности дерева)
Tree	arbre	дерево
Update updating	mettre à jour mise à jour	корректировать, корректировка
Value (call by)	(appel par) valeur	значение (вызов по значению)
Vertex (pi. vertices)	arc (d'un graphe)	дуга (графа)
Virtual	virtuel	виртуальный
Word	mot	слово
Write	écrire	писать

Приложение В

Встроенные (стандартные) функции в ФОРТРАНе

Функция	Аргументы и результат	Определение
ABS	*REAL→ REAL*	Абсолютное значение
IABS	*INTEGER → INTEGER*	
DABS	*D.P. → D.P.*	
CABS	*COMPLEX → REAL*	Модуль $(x + iy \to \sqrt{x^2 + y^2})$
FLOAT	*INTEGER → REAL*	Преобразование без округления
SNGL	*D.P. → REAL*	
DBLE	*REAL → D.P.*	
IFIX	*REAL → INTEGER*	Преобразование с округлением $\lfloor x \rfloor$, если $x \geq 0$, $\lceil x \rceil$, если $x \leq 0$
INT	*REAL → INTEGER*	
IDINT	*D.P. → INTEGER*	
AINT	*REAL → REAL*	Округление, как и выше, без преобразования
CMPLX	*REAL × REAL → COMPLEX*	CMPLX(x, y) = x + iy
REAL	*COMPLEX → REAL*	вещественная часть
AIMAG	*COMPLEX → REAL*	мнимая часть
AMOD	*REAL × REAL → REAL*	Остаток от деления нацело первого аргумента на второй
DMOD	*D.P. × D.P. → D.P.*	
MOD	*INTEGER × INTEGER → INTEGER*	
AMAX0	*INTEGER × ... ×INTEGER → REAL*	Максимум из аргументов с возможным преобразованием
AMAX1	*REAL ×. ... × REAL → REAL*	
DMAX1	*D.P. × ... × D.P. → D.P*	
MAX0	*INTEGER × ... × INTEGER → INTEGER*	
MAX1	*REAL × ... × REAL → INTEGER*	
AMIN0	*INTEGER × ... × INTEGER → REAL*	Минимум из аргументов с возможным преобразованием
AMIN1	*REAL × ... × REAL → REAL*	
DMIN1	*D.P. × ... × D.P. →. D.P.*	
MIN0	*NTEGER × INTEGER → INTEGER*	
MIN1	*REAL × ... × REAL → INTEGER*	

Функция	Аргументы и результат	Определение
SIGN *ISIGN* *DSIGN*	*REAL* × *REAL* → *REAL* *INTEGER* × *INTEGER* → *INTEGER* *D.P.* × *D.P.* → *D.P.*	Число, имеющее знак второго аргумента и абсолютное значение первого
DIM *IDIM*	*REAL* × *REAL* → *REAL* *INTEGER* × *INTEGER* → *INTEGER*	$(1) DIM(x, y) = x - y$, если $x > y$ 0 в противном случае
CONJG	*COMPLEX* → *COMPLEX*	Комплексное сопряженное *(x + iy → x – iy)*
EXP *DEXP* *CEXP*	*REAL* → *REAL* *D.P.* → *D.P.* *COMPLEX* → *COMPLEX*	Экспонента
ALOG *DLOG* *CLOG*	*REAL* → *REAL* *D.P.* → *D.P.* *COMPLEX* → *COMPLEX*	Логарифм натуральный
ALOG10 *DLOG10*	*REAL* → *REAL* *D.P.* → *D.P.*	Логарифм десятичный
SQRT *CSQRT* *DSQRT*	*REAL* → *REAL* *COMPLEX* → *COMPLEX* *D.P* → *D.P*	Квадратный корень
SIN *DSIN* *CSIN*	*REAL* → *REAL* *D.P.* → *D.P* *COMPLEX* → *COMPLEX*	Синус
COS *DCOS* *CCOS*	*REAL* → *REAL* *D.P.* → *D.P* *COMPLEX* → *COMPLEX*	Косинус
ATAN *DATAN*	*REAL* → *REAL* *D.P.* → *D.P*	Арктангенс
ATAN2 *DATAN2*	*REAL* × *REAL* → *REAL* *D.P.* × *D.P.* → *D.P*	Арктангенс частного от деления первого аргумента на второй
TANH	*REAL* → *REAL*	Тангенс гиперболический

Примечание: Можно заметить достаточно нелогичное отсутствие функций арксинуса, арккосинуса, синуса и косинуса гиперболических, хотя арктангенс и тангенс гиперболический предусмотрены. Некоторые трансляторы исправляют эту неполноту стандарта, разрешая функции *ASIN*, *ACOS*, *SINH* и *COSH*.

Приложение Г

Соглашения АЛГОЛа W о типе результата арифметических операций

Сложение, вычитание (A ± B):

A \ B	INTEGER	REAL	LONG REAL	COMPLEX	LONG COMPLEX
INTEGER	INTEGER	REAL	LONG REAL	COMPLEX	LONG COMPLEX
REAL	REAL	REAL	REAL	COMPLEX	COMPLEX
LONG REAL	LONG REAL	REAL	LONG REAL	COMPLEX	LONG COMPLEX
COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX
LONG COMPLEX	LONG COMPLEX	COMPLEX	LONG COMPLEX	COMPLEX	LONG COMPLEX

Деление:

Как и выше, кроме деления двух целых операндов; в этом случае результат имеет тип *LONG REAL*

*Умножение (A * B):*

A \ B	INTEGER	REAL или LONG REAL	COMPLEX или LONG COMPLEX
INTEGER	INTEGER	LONG REAL	COMPLEX
REAL или LONG REAL	REAL	LONG REAL	COMPLEX
COMPLEX или LONG COMPLEX	LONG COMPLEX	LONG COMPLEX	LONG COMPLEX

Возведение в степень (A^B):

A \ B	INTEGER
INTEGER	LONG REAL
REAL	LONG REAL
LONG REAL	LONG REAL
COMPLEX	LONG COMPLEX
LONG COMPLEX	LONG COMPLEX

Встроенные (стандартные) процедуры и переменные АЛГОЛа W

1. Процедуры–операторы

Процедура	Аргументы тип и число	Определение
READ	Произвольное число аргументов произвольных типов с атрибутом *RESULT*	Читать значения, соответствующие аргументам, из входного файла в «свободном» формате. Чтению предшествует переход к следующей записи
READON	Произвольное число аргументов произвольного типа с атрибутом *RESULT*	Так же, без смены записи
READCARD	*STRING (80) RESULT* $x_1, x_2, \ldots, x_n$	Читать записи целиком по 80 литер
WRITE	Произвольное число аргументов произвольного типа с атрибутом *VALUE*	Писать в выходной файл значения аргументов в формате, соответствующем их типу. Предварительно сменяется запись (например, строка на АЦПУ)
WRITEON	Произвольное число аргу ментов произвольного типа с атрибутом *VALUE*	Так же, без смены записи

2. Процедуры–выражения

Процедура	Аргументы	Тип результата	Определение
TRUNCATE	*REAL VALUE X*	*INTEGER*	«Усеченное» значение X, а именно: $[X]$, если $X \geq 0$, $\lceil X \rceil$ если $X < 0$
ENTIER	*REAL VALUE X*	*INTEGER*	$\lfloor X \rfloor$ независимо от знака X
ROUND	*REAL VALUE X*	*INTEGER*	Округленное значение X: $\lfloor X + 0{,}5 \rfloor$, если $x \geq 0$, $\lceil X - 0{,}5 \rceil$, если $x < 0$
ROUNDTOREAL	*LONG VALUE X*	*REAL*	Ближайшее к аргументу «вещественное» число
REALPART *LONGREALPART*	*COMPLEX VALUE Z* *LONG COMPLEX VALUE Z*	*REAL* *LONG REAL*	(вещественная часть аргумента

Процедура	Аргументы	Тип результата	Определение
IMAGPART *LONGIMAGPART*	*COMPLEX VALUE Z* *LONG COMPLEX VALUE Z*	*REAL* *LONG REAL*	Мнимая часть аргумента
IMAG *LONGIMAG*	*REAL VALUE X* *LONG REAL VALUE X*	*COMPLEX* *LONG COMPLEX*	Комплексное число *xi*
ODD	*INTEGER VALUE N*	*LOGICAL*	Нечетность *N*: *ODD(N) = (N REM 2 = 1)*
SQRT *LONGSQRT*	*REAL VALUE X* *LONG REAL VALUE X*	*REAL* *LONG REAL*	Квадратный корень
EXP *LONGEXP*	*REAL VALUE X* *LONGREALVALUEX*	*REAL LONG REAL*	Экспонента
LN *LONGLN*	*REAL VALUE X* *LONG REALVALUE X*	*REAL* *LONG REAL*	Натуральный логарифм
LOG *LONGLOG*	*REAL VALUE X* *LONG REAL VALUE X*	*REAL* *LONG REAL*	Десятичный логарифм
SIN *LONGSIN*	*REAL VALUE X* *LONG REAL VALUE X*	*REAL* *LONG REAL*	Синус
COS *LONGCOS*	*REAL VALUE X* *LONG REAL VALUE X*	*REAL LONG REAL*	Косинус
ARCTAN *LONGARCTAN*	*REAL VALUE X* *LONG REAL VALUE X*	*REAL* *LONG REAL*	Арктангенс (заключенный между $-\pi/2$ и $+\pi/2$

3. Встроенные переменные

Имя	Значение на ИБМ 360/370 (в качестве примера)	Определение
MAXINTEGER	*2147483647*	Наибольшее представимое в машине целое положительное число
EPSILON	*9.536743'–07*	Наибольшее положительное число типа *REAL*, которое, будучи прибавленным к 1, оставляет результат равным 1
LONGEPSILON	*2.22044604925031'–16L*	Наибольшее положительное число типа *LONG REAL*, которое, будучи прибавленным к 1, оставляет результат равным 1
MAXREAL	*7.23700557733226'+75L*	Наибольшее представимое в машине положительное число типа *LONG REAL*
PI	*3.14159265358979L*	Наилучшее в машине приближение π числом типа *LONG REAL*

Приложение Д

Встроенные функции в ПЛ/1 (частичный список)

Функция	Аргументы и результат	Определение
BIT	*Выражение* (× *Целый*) → *BIT*	Преобразование в цепочку битов; второй аргумент (необязательный) предписывает длину результата
CHAR	*Выражение* (× *Целый*) → *CHARACTER*	Преобразование в цепочку литер; то же замечание
INDEX	*CHARACTER* × *CHARACTER* → *FIXED BIN* или *BIT* × *BIT* → *FIXED BINARY*	*INDEX (C1, C2)* = позиции первой литеры *C2* в *C1*, если *C2* является подстрокой *C1*, и 0 в противном случае
LENGTH	*CARACTER VARYING* → *FIXED BINARY* или *BIT VARYING* → *FIXED BINARY*	Текущая длина литерной или битовой цепочки переменной длины
REPEAT	*CHARACTER* × *Целый* → *CHARACTER* или *BIT* × Целый → *BIT*	*REPEAT (C, I)* = контактенапии *I* + 1 экземпляров цепочки *C*
SUBSTR	*BIT* × Целый (× Целый) → *BIT* или *CHARACTER* × Целый (× Целый) → *CHARACTER*	*SUBSTR (C, I, J)* = подстроке *C* длины *J*, начинающейся с *I*–й литеры или *I*–го бита; *SUBSTR (C, I)* = подстроке *C* от *I*–й литеры до конца *C*
VERIFY	*BIT* × *BIT* → *FIXED BINARY* или *CHARACTER* × *CHARACTER* → *FIXED BINARY*	*VERIFY (X, Y)* = 0, если каждая литера или каждый бит *X* содержится в *Y*; в противном случае это позиция первой литеры или первого бита *X*, которые не содержатся в *Y*
ABS	*Числовой* → *Числовой не комплексный*	Абсолютное значение или модуль
BINARY	*DECIMAL* → *BINARY*	Перевод в двоичную систему
CEIL	*Вещественный* → *Целый*	Функция $\lceil x \rceil$: *CEIL(x)* = $x + iy$
COMPLEX	*Вещественный* × *Вещественный* → *Комплексный*	*COMPLEX(x, y)* = $x + iy$

Функция	Аргументы и результат	Определение
CONJG	*Комплексный* → *Комплексный*	Комплексное сопряженное
DECIMAL	*BINARY* → *DECIMAL*	Перевод в десятичную систему
FIXED	*FLOAT* → *FIXED*	Преобразование способа представления
FLOAT	*FIXED* → *FLOAT*	Преобразование способа представления
FLOOR	*Вещественный* → *Целый*	Функция $\lfloor x \rfloor$: *FLOOR(x)* = $\lfloor x \rfloor$
IMAG	*Комплексный* → *Вещественный*	Мнимая часть
MAX	*Числовой* × ... × *Числовой* → *Числовой*	Максимум аргументов
MIN	*Числовой* × ... × *Числовой* → *Числовой*	Минимум аргументов
MOD	*Числовой* × *Числовой* → *Числовой*	Остаток от деления нацело первого аргумента на второй
REEL	*Комплексный* → *Вещественный*	Вещественная часть
ROUND	*Числовой* × *Целый* → *Числовой*	*ROUND(X, N)* = значению *X*, округленному до *N*–й цифры (двоичной или десятичной) после запятой
SIGN	*Числовой* → *Числовой*	*SIGN(x)* = –1, если $x < 0$, 0, если $x = 0$, 1, если $x > 0$
TRUNC	*Числовой* × *Целый* → *Числовой*	*TRUNC(X, N)* = значению *X*, обрезанному на *N*–ой цифре (десятичной или двоичной) после запятой
ATAN	*Числовой* (× *Числовой*) → *Числовой*	*ATAN(x)* = Арктангенс *x* *ATAN(x, y)* = Арктангенс *y/x*

Функция	Аргументы и результат	Определение
ATANH	*Числовой* → *Числовой*	Арктангенс гиперболический
COS		Косинус
COSH		Косинус гиперболический
EXP		Экспонента
LOG		Натуральный логарифм
LOG10		Десятичный логарифм
LOG2		Логарифм по основанию 2
SIN		Синус
SINH		Синус гиперболический
SQRT		Квадратный корень
TAN		Тангенс
TANH		Тангенс гиперболический
ALL	*Массив BIT* → *BIT*	Логическое (побитовое) умножение всех элементов
ANY	*Массив BIT* → *BIT*	Логическое (побитовое) сложение всех элементов
DIM	*Массив* × *Целое* → *FIXED BINARY*	*DIM(T, I)* – «расширение» *I*–го измерения исходного массива *T* *HBOUND(T, I)* - *LBOUND(T, I)* + 1
HBOUND	*Массив* × *Целое* → *FIXED BINARY*	*HBOUND(T, I)* – верхняя граница *I*–го измерения массива *T*
LBOUND	*Массив* × *Целый* → *FIXED BINARY*	*LBOUN(T, I)*=нижняя граница *I*–го измерения массива *T*
POLY	*Числовой массив* × *Числовой* → *Числовой* или *Числовой массив* × *Числовой массив* → *Числовой*	Если *x* имеет размерность (*m : n*), а *x* – *число, то P0LY(T, x)* = $T_m + T_{m+1}x + \ldots + T_n x^{m-n}$ Если *T* имеет размерность *(m : n)* а *x* – размерность *(p : q)*, то *P0LY(T, x)* = $T_m + T_{m+1}x_p + T_{m+2}x_p x_{p+1} + \ldots + T_n x_p \ldots x_q$
PROD	*Числовой* → *Числовой массив*	Произведение всех элементов массива
SUM	*Числовой* → *Числовой массив*	Сумма всех элементов массива
ALLOCATION	*Переменная CONTROLLED* → *B1T(1)*	*'1'B* или *'0'B* в зависимости от того, обладает ли аргумент «активным» поколением

Примечание: В силу множественности существующих в ПЛ/1 числовых типов и многочисленных преобразований, выполняющихся над аргументами и результатами функций, выше было использовано некоторое число «абстрактных типов»: *Целый, Вещественный, Числовой, Комплексный, Числовой не комплексный.* Они представляют собой объединения (в теоретико–множественном смысле) различных определенных в ПЛ/1 типов; например, тип *Целый* группирует все типы *FIXED, BINARY*(n, 0) и *FIXED DECIMAL(n, 0)* Кроме того, неполные спецификации, такие, как *BINARY* или *FIXED,* порождают объединение всех типов, получающихся изменением других аргументов. Наконец, *BIT* означает *B1T(n)* с произвольным *n,* а *CHARACTER* означает *CARACTER(n)* с произвольным *n.*

Функции *BINARY, DECIMAL, FLOAT* и *FIXED* могут использоваться с одним или двумя дополнительными аргументами, позволяющими задать точность результата.

БИБЛИОГРАФИЯ

В этой библиографии используются следующие сокращения для наиболее часто упоминаемых зарубежных, но знакомых советским программистам журналов:

CommACM	:	Communication of the ACM (Association for Computing Machinery).
JACM	:	Journal of the ACM.
CompSurv	:	Computing Surveys (ACM).
SIGPLAN Notices	:	заметки, публикуемые группой SIGPLAN из ACM,
SoftPracExp	:	Software, Practice and Experience, Великобритания.
RAIRO–голубой	:	Французский журнал по автоматике, информатике и исследованию операций – Revue française, d'Automatique, d'Informatique et de Recherche Opérationnelle; выпускается парижским издательством Dunod для АФСЕТ–Французской ассоциации экономической и технической кибернетики (AFCET–Association Françause pour la Cybernétique Economique et Technique); голубая серия – информатика.
RAIRO–красный	:	то же; красная серия–теоретическая информатика.
Actlnf	:	Acta Informatica; журнал, выпускаемый Западноберлинским издательством Springer–Verlag.
CompJ	:	The Computer Journal, Великобритания.
IBMSyst	:	IBM System Journal, издается фирмой ИБМ.
IBMResDev	:	IBM Journal of Research and Development.
SiamC	:	SIAM Journal on Computing; издается обществом Society for Industrial and Applied Mathematics.

Ссылки, добавленные при переводе на русский язык, отмечены звездочкой.

[Абриаль 74] Abrial J.R. Data Semantics; Université Scientifique et Médicate de Grenoble, Laboratoire, d'Informatique, 1974; см. также в материалах Рабочей конференции ИФИП ТС2, Cargèse (Corse), 1974.

[Абриаль 77] Abrial J.R. Manuel du Langage Z (Z/13); неопубликованный отчет, март 1977.

[Абриаль 77а] Abrial J.R. Mécanismes de Transformations du Langage Z (Z/14); неопубликованный отчет, июнь 1977.

[АНСИ 66] USA "National Standart FORTRAN; ANSI X 3.9–1966, American National Standards Institute, 1966.

[Арсак 65] Arsac J., Lentin A., Nivat M., Nolin M. ALGOL, Théorie et Pratique; Gau–thier–Villars, Paris, 1965.

[Арсак 70] Arsac J. La Science Informatque; Dunod, Paris, 1970.

[Арсак 77] Arsac J. Nouvelles leçons sur la programmation; Dunod, Paris, 1977.

[ACM 73] The Impact of Structures Programming on Software Engineering and Production; материалы семинара, организованного ACM в Париже в 1973 г.; воспроизведение статей Милза, Дейкстры, Бэйкера.

[ACM 76] Proceeding of (an ACM) Conference on Data: Abstraction, Definition and Structure; SIGPLAN Notices, 8, 2 специальный номер, 1976.

[ACM 76a] Draft Proposed FORTRAN Standart проект нового стандарта ФОРТРАНа; SIGPLAN Notices, март 1976.

[ACM 766] Специальный выпуск: Data Base Management System, специальный номер по системам управления базами данных: CompSurv, 8, 1, март 1976.

[ACM 77] Proceedings of an ACM conference on Language Design for Reliable Software; SIGPLAN Notices, 12, 3, март 1977; № 2,2, Software Engineering Notes, март 1977.

[АФСЕТ 75] AFCET 75. Группа АЛГОЛа в АФСЕТ: Manuel du Langage algorithmi–que ALGOL 68; Hermann, Paris, 1975.

[Ахо 74] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. Пер. с англ. – М.: Мир, 1979.

[Ахо 75] Aho A. V., Corasick J. Efficient String Matching: An Aid to Bibliographic Search; CommACM, 18, 6, p. 333–343, люнь 1975.

[Ахо 77] Aho V., Ullman D. Principles of Compiler Design; Addison–Wesley, Reading (Mass.), апрель 1977.

[Бакл 76] Buckle J. K. Some Aspects of 2900 Series Architecture; RAIRO–голубой, 10, 9, 21–29, сентябрь 1976.

[Баррон 69] Barron D. W. Recursive Techniques in Programming; McDonald Londres, et American Elsevier, New York, 1969.

[Баррон 77] Баррон Д. Введение в языки программирования. Пер. с англ.–М.: Мир, 1980.

[Бауэр 73] Bauer F. L. (редактор). Advanced Course on Software Engineering; Springer–Verlag, Berlin, 1973.

[Бауэр 75] Bauer F. L. (редактор). Advanced Course on Compiler Construction; Lecture Notes m Computer Science; Springer–Verlag, Berlin, 1975.

[Бауэр 78]* Бауэр Ф.Л., Гооз Г. Информатика. Пер. с нем., М., «Мир», 1978.

[Бердж 75] Burge W. Recursive Programming Techniques; Addison–Wesley, Reading (Mass.), 1976.

[Белл 71] Bell G., Newell Al. Computer Structures: Readings and Examples; McGraw–Hill, New York, 1971.

[Беллман 62] Беллман Р., Дрейфус С. Прикладные задачи динамического программирования. Пер. с англ.–М.: Наука, 1965.

[Бемер 69] Bemer R. W., A. Politico–Social History of ALGOL; Annual Review on Automatic Programming, Halpern et Shaw (редакторы), Pergamon Press, Oxford, 1969.

[Берри 76] Berry G. Bottom–up Computation of Recursive Programs; RAIRO–красный, 10, 3, p. 47–82, март 1976.

[Берстал 76] см. [Дарлингтон 76].

[Берстал 77] Burstall R. M., Darlington J. A transformation System for Developing Recursive Programs; JACM, 24, 1, p. 44–61, январь 1977.

[Бертэ 71] Berthet C. Le Langage PL/1; Dunod, Paris, 1971.

[Биртуистл 73] Birtwistle G.M., Dahl O.J., Myrhaug B., Nygaard K. Simula BEGIN; Studentliteratur, Lund Suède, et Auerbach Publichers, Philadelphie (Penn.), 1973.

[Болье 64] Bolliet L., Gastinel N., Laurent et al. ALGOL, Manuel pratique; Hermann, Paris, 1964; содержит очень полную библиографию по языку АЛГОЛ 60 до 1963 г.

[Боэм 66] Boehm C, Jacopini, G. Flow Diagrams, Turing Machines, and Languages With Only Two Formation Rules; CommACM, 9, p. 366–371, 1966.

[Боэм 73] Boehm W. Software and Its Impact: A Quantitative Assessment; Datamation, 19, 5, p. 48–59, май 1973.

[Боэм 77] Boehm W., Software Engineering; IEEE Transactions on Computers, 1977.

[Боэм 77a] Boehm W. Seven Basic Principles of Software Engineering; Rapport TRW, 1977.

[Браун 74] Браун П. Макропроцессоры и мобильность программного обеспечения. Пер. с англ.–М.: Мир, 1977.

[Браун 77] Brown, P.J. (редактор). Software Portability, An Advanced Course; Cambridge University Press, Cambridge, Великобритания, 1977.

[Бринч–Хансен 73] Brinch Hansen P. Operating System Principles; Prentice–Hall, Englewood CUffs (N.J.), 1973.

[Брудно 71]* Брудно А. Л. АЛГОЛ 60.–М.: Наука, 1971.

[Брукс 75] Брукс Ф.П. Как проектируются и создаются программные комплексы. Пер. с англ.–М.: Наука, 1979.

[Бэйкер 72] Baker F.T. Chief Programmer Team Management of Production Programming; IBM Syst, 11, 1, p. 56–73, см. также [ACM 73].

[Бэкус 57] Backus J.W. et al The FORTRAN Automatic Coding System; Proc. Western Joint Computer Conference, Los Angeles, p. 188–198, февраль 1957.

[Васильев 72]* Васильев В. А. Язык АЛГОЛ–68. Основные понятия.–М.: Наука, 1972.

[Вегнер 69] Wegner P. Programming Languages, Information Structures and Machine Organisation; McGraw–Hill, New York, 1969.

[Вегнер 72] Wegner P. The Vienna Definition Language; CompSurv, 4, 1, p. 5–63, март 1972.

[Вейнберг 71] Weinberg G.M. The Psychology of Computer Programming; Van Nostrand Reinhold, New York, 1971.

[Вейон 76] Veillon G. Transformation de Programmes recursifs; RAIRO–голубой, 10, 9, 7–20, сентябрь 1976.

[Вельбицкий 80]* Вельбицкий И. В., Ходаковский В.Н., Шолмов Л. И. Технический комплекс производства программ на машинах ЕС ЭВМ и БЭСМ–6.–М.: Статистика, 1980.

[Вирт 66] Wirth N., Hoare C.A.R. A Contribution to the Development of ALGOL; CommACM, 9, 6, p. 413–432, июнь 1966.

[Вирт 71] Wirth N. Program Development by Stepwise Refinement; CommACM, 14, 4, p. 221–227, апрель 1971.

[Вирт 71a]* Вирт Н. Язык программирования Паскаль. Пер. с англ.–В сб. «Алгоритмы и организация решения экономических задач», вып. 9.–М.: Статистика, 1977.

[Вирт 72] Вирт Н. Систематическое программирование. Введение. Пер. с англ.–М.: Мир, 1977.

[Вирт 76] Wirth N. Algorithms Data Structures Programs; Prentice–Hall, Englewood Cliffs (N.J.), 1976.

[Вулф 71] Wulf W.A., Russel D.B., Habermann A.N. BLISS: A Language for System Programming; CommACM, 14, 12, p. 780–790, декабрь 1971.

[Вулф 75] Wulf W.A., London R. L., Shaw M. Abstraction and Verification in ALPHARD; см. в [Шуман 75].

[Вюйемен 73] Vuillemin J. Proof Techniques for Recursive Programs; Computer Science Department, Stanford University, Stanford (Californie), октябрь 1973.

[Вюйемен 75] Vuillemin J. Structures de Donnees; Cours de l'Ecole d'Ete d'informatique EDF–IRIA–CEA, Le–Breau–en–Yvelines, сентябрь 1975.

[Гилман 79]* Гилман Л., Роуз А. Курс АПЛ: диалоговый подход. Пер. с англ.–М.: Мир, 1979.

[Гир 69] Gear W.C. Introduction to Computer Organization and Data Structures; McGraw–Hill, New York, 1969.

[Головкин 78]* Головкин Б. А. Надежное программное обеспечение.–Зарубежная радиоэлектроника, 1978, № 12.

[Грессей 78] Greussay P. Introduction à la Programmation en LISP; Dunod, Paris, 1978.

[Грис 71] Грис Д. Конструирование компиляторов для цифровых вычислительных машин. Пер. с анл.–М.: Мир, 1975.

[Грисуолд 71] Грисуолд Р., Поудж Дж., Полонски И. Язык программирования Снобол–4. Пер. с англ.–М.: Мир, 1980.

[Грисуолд 75] Griswold R.E. String and List Processing in SNOBOL 4: Techniques and Applications; Prentice–Hall, Englewood Cliffs (N.J.), 1975.

[Грифитс 75] Griffiths M. Requirements for and Problems with Intermediate Languages for Programming Language Implementation; dans: Language Hierarchies and Interfaces (Ecole d'Ete, MarkOberdorff, 1975), Lecture Notes in Computer Science, Springer–Verlag, Berlin, 1977.

[Гуттаг 77] Guttag J. Abstract Data Types and the Development of Data Structures: CommACM, 20, 6 p. 396–404, июнь 1977.

[Дал 69]* Дал У., Мюрхауг Б., Нюгорд К. Симула–67– универсальный язык программирования. Пер. с англ.–М.: Мир, 1969.

[Дал 72] Дал У., Дейкстра Э., Хоор К. Структурное программирование. Пер. с англ.–М.: Мир, 1975.

[Дал 72a] Dahl O. J., Hoare C.A.R. Hierarchical Program Structures; см. в [Дал 72].

[Дарлингтон 76] Datlington J., Burstall R. M. A System Which Automatically Improves Programs; Actlnf, 6, 1, p. 41–60, январь 1976.

[Дейкстра 61] DijkstraE.W. A Primer on ALGOL 60 Programming; Academic Press, Londres, 1961.

[Дейкстра 68] Dijkstra E.W. GOTO Statement Considered Harmful; CommACM, 11, 3, p. 147–148, март 1968.

[Дейкстра 68a] DijkstraE.W. A Constructive Approach to the Problem Correctness; BIT, 8, p. 174–186, 1968.

[Дейкстра 686] Dijkstra E.W. The Structure of THE Multiprogramming System; CommACM, 11, 5, p. 341–346, май 1968.

[Дейкстра 71] Dijkstra E.W. A Short Introduction to the Art of Programming; rapport EWD 316 (Dictaathr. 2.268), Technische Hogescholl Eindhoven, 1971.

[Дейкстра 72] Dijkstra E.W. Notes on Structured Programming; см. в [Дал 72].

[Дейкстра 72a] Dijkstra E.W. The Humble Programmer (ACM Turing Awars Lecture); CommACM, 15, 10, p. 859–866, октябрь 1972.

[Дейкстра 76] Дейкстра Э. Дисциплина программирования. Пер. с англ.–М.: 1978.

[Демнер 78] Демнер И., Крал Я. На пути к надежному программному обеспечению: задачи реального времени; в [Хиббард 78], т. 1, стр. 10–19.

[Джексон 75] Jackons M.A. Principles of Program Design; Academic Press, Londres, 1975.

[Донагю 76] Donahue J.E. Complementary Definitions of Programming Languages; Springer–Verlag, Berlin, 1976.

[Дэвис 58] Davis M. Computability and Unsolvability; McGraw–Hill, New York, 1958.

[Ершов 79]* Ершов А. П. (редактор). Пересмотренное сообщение об АЛГОЛЕ 68. Пер. с анлг.–М.: Мир, 1979.

[Зан 75] Zahn Ch.T., Jr. A User Manuel for the MORTRAN2 Macro–Translator; CGTM n° 167, SLAG Computation Research Group, Stanford (CaL), август 1975.

[ИБМ 70] IBM System/360 Operating System–PL/1 (F) Language Reference Manual; GC–28–8201–3, IBM, 1970.

[ИБМ 71] Chief Programmer Teams Principles and Procedures; Rapport FSC 71–5108, Federal Systems Division, International Business Machines Corporation, Gaithersburg (Maryl.), 1971.

[IEEE 76] Special Section on Testing (Специальное дополнение по методам тестирования программ), Stucki L.G. (редактор) IEEE Trans, on SE, SE-2, 3. p. 194–231, сентябрь 1976.

[IEEE 77] Special Issue on Stack Machines (специальный номер о вычислительных машинах со стековой архитектурой); Computer, 10, 5, май 1977.

[Инглз 71] Ingalls D. A FORTRAN Execution Time Estimator; Rapport STAN–71–204. Computer Science Department, Stanford University, Stanford (Cal.), февраль 1971.

[Ишбиа 75] The System Implementation Language LIS; Reference Manual; Document 4549 E/EN, Compagnie Internationale pour l'lnformatique, июль 1975.

[Иенсен 75] Jensen K., Wirth N. PASCAL User Manual and Report; Springer–Verlag, Berlin, 1975.

[Йодан 79] Йодан Э. Структурное проектирование и конструирование программ. Пер. с англ.–М.: Мир, 1979.

[Кадью 72] Cadiou J. M. Recursive Definitions of Partial Functions and their Computation by Machine; thèse de Ph. D., Artificial Intelligence Project, rapport STAN–CS–266–72 et AIM–163, Computer Science Department, Stanford University, Stanford (Californie), 1972.

[Карпов 76]* Карпов В.Я. Алгоритмический язык ФОРТРАН–М.: Наука, 1976.

[Касьянов 78]* Касьянов В. Н., Поттосин И. В. Применение методов оптимизации в проверке правильности программ; в [Хиббард 78], т. 1, стр. 225–237.

[Катцан 70] Katzan H. APL Programming and Computer Techniques; Van Nostrand Reinhold, New York, 1970.

[Кеман 76] Quement B. Construction de Modèles et de Types dans un système extensible; doctorat de troisième cycle, Univerité Pierre et Marie Curie (Paris VI), ноябрь 1976.

[Керниган 74] Kernighan W., Plauger P.J. The Elements of Programming Style; McGraw–Hill, New York, 1974.

[Керниган 76] Kernighan B.W., Plauger P.J. Software Tools; Addison–Wesley, Reading (Mass.), 1976.

[Кетков 78]* Кетков Ю.Л. Программирование на Бэйсике.–М.: Статистика, 1978.

[Кибурц 75] Kieburtz R.B. Structured Programming and Problem Solving with ALGOL; Prentice–Hall, Englewood Cliffs (N.J.), 1975.

[Кнут 68] Кнут Д. Искусство программирования, т.1, Основные алгоритмы. Пер. с англ.–М.: Мир, 1976.

[Кнут 69] Кнут Д. Искусство программирования, т.2, Получисленные алгоритмы. Пер. с англ.–М.: Мир, 1977.

[Кнут 72] Knuth D.E. Ancient Babylonian Algorithms, CommACM, 15, 7 p. 671–678, июль 1972.

[Кнут 73] Кнут Д. Искусство программирования, т.3, Сортировка и поиск. Пер. с англ.–М.: Мир, 1978.

[Кнут 74] Knuth D.E. Structures Programming with GOTO Statements; CompSurv, 6, p. 261–301, 1974.

[Кнут 76] Knuth D.E. Manages stables et leurs relations avec d’autres Problèmes combinatoires, Introduction à l’Analyse mathematique des Algorithmes; Presses de l'Université de Montréal, 1970.

[КОБОЛ 77]* Государственный стандарт языка КОБОЛ 22558–77.–М.: ПС стандартов СССР, 1977.

[Кодд 70] Codd E.F. A Relational Model for Large Shared Data Banks; CommACM 13, 6, p. 377–387, июнь 1970.

[Кок 70] Cocke J., Schwartz J. Programming Languages and Their Compilers: Universite de New York, 1970.

[Конвей 73] Conway R.E., Gries D. An Introduction to Programming: A Structured Approach Using PL/1 and PL/C; Winthrop, Cambridge (Mass.), 1973.

[Королев 78]* Королев Л.Н. Структуры ЭВМ и их математическое обеспечение.–М.: Наука, 1978. '

[Крафт 77] Kraft Ph. Programmers and Managers: The Routinization of Computer Programming in the United States; Springer–Verlag, Berlin, 1977.

[Крокус 75] Crocus (авторский коллектив) Systèmes d'Exploitation des Ordinat–eurs–Principes de Conception; Dunod, Paris, 1975.

[Купер 66] Cooper D. C The Equivalence of Certain Computations; CompJ, 9, 1, p. 45–52, май 1966.

[Курочкин 68]* Курочкин В. М. (редактор). Универсальный язык программирования ПЛ/1. Пер. с англ.–М.: Мир, 1968.

[Куртэн 74] Courtin J., Voiron J. Introduction à l'Algorithmique et aux Structures de données; Traduction des Schémas de Programme en FORTRAN – Applications aux Structure de données; cours polycopiés de l'IUT B de Grenoble, Department d'informatique, 1974–1975.

[Лавров 72]* Лавров С. С. Универсальный язык программирования.–М.: Наука 1972.

[Лавров 78]* Лавров С. С. Методы задания семантики языков программирования.–Программирование, № 6, с. 3–10, 1978.

[Лавров 78а]* Лавров С. С, Силагадзе Г. С. Автоматическая обработка данных. Язык ЛИСП и его реализация.–М.: Наука, 1978.

[Лам 71] Lum V.Y., Yuen S.T., Dodd M. Key–to–address Transform Techniques: A fundamental Performance Study on Large Existing Formatted Files; CommACM, 14, 4, p. 228–239, апрель 1971.

[Ламуф 73] Larmouth J. Serious FORTRAN; SoftPracExp, 3, p. 87–107 (1–ère partie), et p. 197–225 (2éme partie), 1973.

[Ледгард 75] Ledgard H.F., Marcotty M.A. A Genealogy of Control Structures: CommACM, 18, 11, p. 629–638, ноябрь 1975.

[Линдси 73]* Линдси Ч.С., ван дер Мюйлен. Неформальное введение в АЛГОЛ 68. Пер. с англ.–М.: Мир, 1973.

[Лисков 74] Liskov B.H., Zilles S.N. Programming with Abstract Data Types; MIT, Project MAC, Computation Structures Group, Memo no. 99, Cambridge (Mass.) (Сокращенная версия в SIGPLAN Notices, 9, p. 50–59, апрель 1974.)

[Лисков 75] Liskov B.H., Zilles S.N. Specification Technique for Data Abstractions; IEEE Trans, on S.E, SE–1, 1, март 1975.

[Лисков 75а] Liskov B. H. An Introduction to CLU; в [Шуман 75].

[Лисков 77] Liskov B. H., Snyder A., Atkinson R., Schaffen C. Abstraction Mechanisms in CLU; CommACM, 20, 8, p. 564–576, август 1977.

[Лорен 71] Lorin H. A Guided Bibliography to Sorting; IBMSyst, 3, p. 244–254, 1971.

[Лэмпсон 77] Lampson B. W., Horning J. J., London R., Popek G. L. Report on the Programming Language Euclid; SIGPLAN Notices, 12, 2, p. 1–79 (специальный номер), февраль 1977.

[Люка 83] Lucas E. Recreations Mathematiques; Paris, 1883.

[Маджинис 79]* Маджинис Дж. Программирование на стандартном КОБОЛе. Пер. с англ.–М.: Мир, 1979.

[Майерс 76] Myers G.J. Software Reliability – Principles and Practices; Wiley, New York, 1976.

[Мак–Карти 60] McCarthy J. Recursive Function of Symbolic Expressions and Their Computation by Machine, Part. 1, CommACM, 3, 4, p. 184–185, апрель 1960. Воспроизведено в исправленном виде в [Розен 67].

[Мак–Карти 62] McCarthy J. et al: LISP 1.5 Programmer's Manual; The M.I.I. Press, Cambridge (Mass.), 1962.

[Мал 77] Mahl R., Boussard J.C.: Algorithmique et Structures de Données; Universite de Nice, Laboratoire d'Informatique, 1977.

[Манна 74] Manna Z. Mathemacial Theory of Computation; McGraw–Hill, New York–Paris, 1974.

[Маркотти 76] Marcotty M., Ledgard H.F., Bochmann G. V. A Sumpler of Formal Definitions; CompSurv, 8, 3, p. 191–276, июнь 1976.

[Маурер 76]* Маурер У. Введение в программирование на языке ЛИСП. Пер. с англ.–М.: Мир, 1976.

[Мейер,76] Meyer B. Description des Structures de Données: Bulletin de la Direction des Études et Recherches d'EDF, 2, p. 81–90, декабрь 1976.

[Мейнадье 71] Meinadier J.–P. Structure et Fonctionnement des ordinateurs, Larousse, Paris, 1971.

[Милз 72] Mills H. D. How to Write Correct Programs and Know It; воспроизведено в [ACM 73].

[Милз 75] Mills H. D. The New Maths of Computer Programming; CommACM, 18, 5, p. 43–48, январь 1975.

[Минский 67] Minsky M.E. Computation: Finite and Infinite Machines; Prentice–Hall, Englewood Cliffs (N.J.), 1967.

[Мишельанжели 71] Michelangeli P., Subsol J. FORTRAN IV (IBM Version H); Manuel D.I. Saclay no. 018, CEA (Saclay), октябрь 1971.

[Наур 60] Naur P. (редактор) Rapport sur le Langage algorithmique Algol 60 (Traduction); Chiffers 3, p. 1–44, 1960 (это сообщение существует в более поздней пересмотренной версии [Наур 63]).

[Наур 63] Naur P. (редактор). Revised Report on the Algoruthmic Language ALGOL 60; CommACM, 6, 1, p. 1–17, 1963; также в CompJ, и в Annual Review in Automatic Programming, 4, p. 217–599. Русский перевод: Алгоритмический язык АЛГОЛ 60. Пересмотренное сообщение. Пер. с англ. Ершов А. П., Лавров С.С, Шура–Бура М.Р. (редакторы).–М.: Мир, 1963.

[Наур 74] Naur P. Concise Survey of Computer Methods; Studentlitteratur, Lund (Suède), et Petrocelli Books, New Yorks, 1974.

[Николз 75] Nicholls J.E. The Structure and Design of Programming Languages; Addison–Wesley, Reading (Mass.), 1975.

[Нильсон 71] Нильсон Н. Искусственный интеллект. Методы поиска решений. Пер. с англ.–М.: Мир, 1973.

[Органик 73] Organick E.I. Computer System Organization; Academic Press, New York, 1973.

[Парнас 72] Parnas D. L. A Technique for Software Module Specification with Examples, CommACM, 15, 5, p. 330–336, май 1972.

[Парнас 72a] Parnas D. L. On The Criteria to be Used in Decomposing Systems into Modules; CommACM, 15, 12, p. 1053–1058, декабрь 1972.

[Пейган 79]* Пейган Ф. Практическое руководство по АЛГОЛу 68. Пер. с англ.–М.: Мир, 1979.

[Пек 78] Пек Дж., Шрак Г. Ф. Мобильность качественного программного обеспечения. Опыт с языком BCPL; см. в [Хиббард 7S], т. 1, стр. 90–104.

[Первин 72]* Первин Ю.А. Основы ФОРТРАНа.–М.: Наука, 1972.

[Пратт 75] Пратт Т. Языки программирования. Разработка и реализация. Пер. с англ.–М.: Мир, 1979.

[Пэр 77] Pair C. Gaudel M. Les Structures de Donnees et leur Representation en Memoire, IRIA, Rocquencourt, 1977.

[Пэр 77a] Pair C. La Construction des Programmes; rapport № 77–R–019, Centre de Recherche en Informatique de Nancy, 1977.

[Рейнолдс 70] Reynolds J. C GEDANKEN: A Simple Typeless Language Based on the Principle of Completeness and the Reference Consept; CommACM, 13, 5, p. 308–319, май 1970

[Рибенс 69] Ribbens D. Programmation non numerique – LISP 1.5; Dunod, Paris, 1969.

[Рич 73] Risch T. REMREC, A Program for Automatic Recursion Removal in LISP; Datalogisklaboratoriet DLU 73/24, Université d'Uppsala, декабрь 1973.

[Розен 67] Rosen S. (редактор). Programming Systems and Languages; McGraw–Hill, New Work, 1967 (в сборник включены многие важные ранее вышедшие статьи по языкам и системам программирования).

[Салтыков 76]* Салтыков А. И., Макаренко Г. И. Программирование на языке ФОРТРАН.–М.: Наука, 1976.

[Саммет 69] Sammet J.E.: Programming Languages: History and Fundamentals; Prentice–Hall, Englewood Cliffs (N.J.), 1969.

[Седжуик 75] Sedgewick R.E. Quicksort: These de Ph.D. Rapport STAN–CS–75–492. Computer Science Department, Stanford University, Stanford (Cal.), 1975.

[Седжуик 77] Sedgewick R.E. The Analysis of Quicksort Programme; Actlnf, 7, 4, p. –327–355, Г977.

[Седжуик 77a] Sedgewick R.E. Quicksort with Equal Keys: Siamc, 6, 2, p. 240–267, февраль 1977.

[Сеттеруэйт 72] Satterthwaite E. Debugging Tools for Higher–Level Languages; SoftPracExp, 2, 3, p. 197–218, июль–сентябрь 1972.

[Сиклоси 76] Siklossy L., Let's Talk LISP; Prentice–Hall, Englewood Cliffs (N.J.), 1976.

[Сите 72] Sites R.L. ALGOL W Reference Manual; Rapport STAN–CS–71–230, Computer Science Department, Stanford University (Cal.), февраль 1972.

[Скотт 70] Scott D. Outline of a Mathematical Theory of Computation; Proceedings of the Fourth Annual Princeton Conference on Information Sciences and System, p. 169–176, 1970.

[Скотт 77]* Скотт Р., Сондак Н. ПЛ/1 для программистов. Пер. с англ.–М.: Статистика, 1977.

[Слейгл 71] Слейгл Дж. Р. Искусственный интеллект. Подход на основе эвристического программирования. Пер. с англ.–М.: Мир, 1973.

[Стербенц 74] Sterbentz P. H. Floating–Point Computation; Prentice–Hall, Englewood Cliffs (N.J.), 1974.

[Стой 74] Stoy J. The Scott–Strachey Approach to the Mathematical Semantics of Programming Languages; Massachussets Institute of Technology, Project Mac, декабрь 1974.

[Стоун 72] Stone H.S. Introduction to Program and Data Structures; McGraw–Hill, New York, 1972.

[Стронг 70] Strong H.R. Translating Recursion Equations into Flow Charts; Priceed–ings of the Second Annual ACM Conference on Theory of Computing, ACM, New York, p. 184–197, 1970.

[Сэмюэль 67] Samuel A. Some Studies in Machine Learning Using the Game of Checkers: II, Recent Progress; IBMDev, 11, 6, p. 601–617, ноябрь 1967.

[Табурье 75] Tabourier Y., Rochfeld A., Frank C La Programmation Structurée en Informatique; Les Editions d'Organisation, Paris, 1975.

[Тейтельман 73] Teitelman W., Bobrow D.G., Hartley A.: INTERLISP Reference Manual; Xerox Reseach Center, Palo Alto (Calif.), 1973.

[Тинент 76] Tennent R.D. The Denotational Semantics of Programming Languages; CommACM, 19, 8, p. 437–453, август 1976.

[Тьюринг 36] Turing A.M. On Computable Numbers, with an Application to the Entscheidungsproblem; Proceedings of the London Mathematical Society (2), 1936–1937 (воспроизведено в Annual of Automatic Programming (R. Goodman, редактор), 1960).

[Уилкс 68] Wilkes M.V. The Outer and Inner Syntax of a Programming Language; CompJ, март 1968.

[Унгер 75] Unger C. (ред.) Command Languages; Proceedings of the IFIP Working Conference on Command Languages, Lund, Suède, 1974; North–Holland, Amsterdam, et American Elsevier, New York, 1975.

[Фейгенбаум 71] Feigenbaum E., Buchanan B., Lederberg J. Generality and Problem Solving: A Case Study Using the DENDRAL Program; dans Machine Intelligence 6, B. Meltzer et D. Michie (редакторы), American Elsevier, New York, 1971.

[Флойд 67] Floyd R. W. Nondeterministic Algorithms; JACM, 14, 4, p. 636–644, октябрь 1967.

[Флойд 70] Floyd R.W.: Introduction to Programming and the ALGOL W Language: Stanford University, Stanford (Cal.), 1970.

[Форсайт 72] Форсайт Дж. Вычислительные методы для математических вычислений. Пер. с англ.–М.: Мир, 1980.

[Хантлер 76] Hantler S.L., King J. C. An Introduction to Proving the Correctness of Programs; CompSurv, 8, 3, p. 331–353, октябрь 1976.

[Хаски 76] Huskey H. D., Haskey V.R. Chronology of Computing Diveces; IEEE Trans, on Computer, C–25, 12 p. 1190–1199, декабрь 1976.

[Хиббард 78] Schuman S.A. (редактор); IFIP TC2 Working Conference on Constructing Quality Software; Novossibirsk, 1977; North–Holland, Amsterdam, 1978; Русский перевод: Труд рабочей конференции ИФИП 1977; Создание качественного программного обеспечения; Пер. с англ.–Новосибирск: ВЦ СО АН СССР, 1978.

[Хигман 67] Хигман Б. Сравнительное изучение языков программирования. Пер. с англ.–М.: Мир, 1974.

[Хоар 62] Hoare C.A.R. Quicksort; CompJ, 5, p. 10–15, 1962.

[Хоар 69] Hoare C.A.R. An Axiomatic Basis for Computer Programming; CommACM, 12, 10, p. 576–582, октябрь 1969.

[Хоар 71а] Hoare C.A.R. Proof of a Programm: FIND; CommACM, 14, 1 p. 39–45, январь 1971.

[Хоар 71б] Hoare C A. R. Procedures and Parameters: an Axiomatic Approach; в: Simposium on Semantics of Algorithmic Languages, Engeler (ed), Springer–Verlag, Berlin, 1971.

[Хоар 71в] Hoare C.A.R., Foley M. Proof of a Recursive Program: QUICKSORT; CompJ, 14, 4, p. 391–395, 1971.

[Хоар 72] Hoare C A. R. Notes on Data Structuring; в [Дал 72].

[Хоар 72а] Hoare C.A.R., Allison D.C. S. Incomputability; CompSurv, 4, 3, p. 169–178, сентябрь 1972.

[Хоар 72б] Hoare C. A. R., Clint M. Program Proving: Jumps and Functions; Actlnf, 1, p. 214–222, 1972.

[Хоар 72в] Hoare C A. R. Proof of a Structured Programm: The Sieve of Eratosthenes; CompJ, 14, 4, p. 321–325, 1972.

[Хоар 73] Hoare C.A.R. Hints on Programming Language Design; Rapport STAN-CS-73-403, Computer Science Department, et AIM–224 Artifical Intelligence Laboratory, Stanford University, Stanford (Cal.), 1973.

[Хорнинг 75] Horning J.J. What the Compiler Should Teel the User; в [Бауэр 75].

[Хьюз 80]* Хьюз Дж., Митчом Дж. Структурный подход к программированию. Пер. с англ.–М.: Мир, 1980.

[Шербоно 75a] Cherbonneau B., Galinier M., Lagasse J. P., Massie H., Mathis A., Paul J.I. Programmation structurée; rapport numéro 112, Université Paul Sabatier (UER Informatique), Toulouse, 1975.

[Шербоно 756] Cherbonneau B., Galinier M, Lagasse J. P., Massie H., Mathis A., Paul J.L.: Développement d'un projet en Programmation Structurée; rapport numéro 113, Universite Paul Sabatier (UER Informatique), Toulouse, 1975.

[Шоу 77] Shaw M., Wulf W.A., London R.L. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators; CommACM, 20, 8, p. 553–564, август 1977.

[Шуман 75] Schuman S.A. (ed.). New Directions in Algorithmic Languages 1975; рабочая группа ИФИП по АЛГОЛУ, издано ИРИА, 1975.

[Шьон 73] Chion J. S., Cleeman E.F. Le Langage ALGOL W. Initiation aux Algori–thmes; Presses Universitaires de Grenoble, 1973.

[Эккерт 76] Eckert J. P. Thoughts on The History of Computing; Computer (IEEE), 9, 12, p. 58–65, декабрь 1976.

[Элсон 73] Elson M. Concepts of Programming Languages; Science Research Associates, Chicago, 1973.

[Эмбл 73] Amble O.J., Knuth D.E. Ordered Hash Tables; Rapport STAN-CS-76-405, Computer Science Department, Stanford University, Stanford (Cal.) январь 1973; см. также в CompJ, 17, 2, p. 135–142, май 1974.

[Энслоу 75] Enslow Ph. H. Summary of the IFIP Working Conference on Operating System Command Languages; в [Унтер 75] p. 389–395.

[Эрли 71] Earley J. Toward an Understanding of Data Structures; CommACM, 14, 10, p. 617–627, октябрь 1971.

[Эрнст 69] Ernst G., Newell A. GPS: A Case Study in General Problem Solving; ACM Monograph Series, Academic Press, New York, 1969.

[Эшкрофт 75] Ashcroft E. E., Manna Z. Translating Program Schemas to White Schemas; SiamC, 4. 2, p. 125–146, июнь 1975.

[USAF 72] The High Cost of Software; Congres, Monterey, 1977.

[Ющенко 73]* Ющенко Е.Л. и др. КОБОЛ.–Киев: Вища школа, 1973.

[Ющенко 76]* Ющенко Е.Л. и др. ФОРТРАН.–Киев: Вища школа, 1976.

Мейер, Бодуэн

МЕТОДЫ ПРОГРАММИРОВАНИЯ,
т. 2

Научный редактор: А. А. Бряндинская. Младший научный ред.: Н.С. Полякова. Художник С. А. Бычков. Художественный редактор В. И. Шаповалов. Технический редактор Л. П. Бирюкова. Корректор М. А. Смирнов. ИБ № 2163
Сдано в набор 13.08.81. Подписано к печати 17.04.82. Формат 60 x $90^{1}/_{16}$. Бумага офсетная № 2. Гарнитура таймс. Печать офсетная. Объем 11,50 бум. л. Усл. печ. л. 23,00. Усл. кр.–отт. 46,36. Уч.–изд. л. 20,79. Изд. № 1/1648. Тираж 30000 экз. Зак. 581. Цена 1 р. 60 к.

ИЗДАТЕЛЬСТВО «МИР»

Москва, 1–й Рижский пер., 2.
Можайский полиграфкомбинат Союзполиграфпрома при Государственном комитете СССР
по делам издательств, полиграфии и книжной торговли.
г. Можайск, ул. Мира, 93.