О. А. Авдеюк, Л. Г. Акулов, В. Ю. Наумов

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ: ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПАСКАЛЬ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

О. А. Авдеюк, Л. Г. Акулов, В. Ю. Наумов

Информатика и программирование: основы программирования на языке Паскаль

Учебно-методическое пособие

2-е издание, исправленное

Волгоград
2018

УДК 004 (075)

Р е ц е н з е н т ы :

кафедра «Физика, методика преподавания физики, математики, ИКТ» ВГСПУ,
зав. кафедрой д-р пед. наук, профессор *Т. К. Смыковская*;

доцент кафедры «Биотехнические системы и технологии» ВолгГМУ
канд. физ.-мат. наук *М. В. Петров*

Печатается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Авдеюк, О. А.

Информатика и программирование: основы программирования на языке Паскаль : учеб.-метод. пособие / О. А. Авдеюк, Л. Г. Акулов, В. Ю. Наумов ; ВолгГТУ. – 2- изд., испр. – Волгоград, 2018. – 268 с.

ISBN 978–5–9948–2869–4

Излагаются основные положения науки информатики. Особое внимание уделяется таким вопросам, как информационные технологии и системы, информационные меры, основы программирования на языке Паскаль.

Пособие отвечает требованиям к курсам «Информатика», «Информатика и программирование», «Основы программирования», «Информационные технологии», предъявляемым в ФГОС3+ к техническим направлениям подготовки бакалавров в ВолгГТУ.

Ил. 115. Табл. 10. Библиогр.: 12 назв.

ISBN 978–5–9948–2869–4

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На сегодняшний день информатика стала одной из самых популярных научных дисциплин. Вопросам этой науки посвящено много книг, журналов, различных публикаций. Однако не всегда люди, интересующиеся проблемами информатики, могут достаточно четко определить круг вопросов, которые охватывает информатика как наука. Часто бытует мнение, что эта дисциплина включает в себя лишь задачи программирования, обработки данных или учение о вычислительных машинах. Да, информатика этим занимается, но частично. На самом деле эта наука включает в себя множество математических, инженерных и даже философских аспектов, через которые она становится фундаментальной наукой, занимающейся схематичным, "формализованным" представлением информации, вопросами ее обработки, а также различными средствами, с помощью которых можно производить необходимую обработку информации. Это включает в себя вопросы анализа и моделирования взаимосвязей и структур в самых различных областях применения. При этом возникает необходимость в разработке способов решения задач информационной обработки на вычислительных машинах, а также в разработке, организации и эксплуатации самих вычислительных машин и систем. Формирование моделей информатики нацелено на представление определенных структур, взаимодействий и процессов в какой-либо области применения с помощью формальных средств - таких как, структуры данных, языки программирования или логические формулы.

Задача информатики состоит в том, чтобы исследовать свойства формальных моделей и развивать их дальше, и не в последнюю очередь - устанавливать связи между формальными моделями и реальным миром в данной предметной области в смысле постановки задачи.

Следует заметить, что в различных источниках информации можно найти разнообразные определения информатики. Это обусловлено двумя моментами.

Во-первых, дело в том, что информатика - наука многогранная. Как говорилось выше, в ней рассматриваются различные аспекты - от математических до философских. Поэтому в определении информатики автор может сделать упор на тот аспект, которому уделяется больше внимания или который он считает более важным.

Во-вторых, следует отметить, что информатика - наука развивающаяся. Следовательно, как любое развивающееся явление, информатика постоянно претерпевает изменения. А это естественным образом влечет за собой изменения понятий, терминов, определений.

В учебно-методическом пособии излагаются основные положения науки информатики. Особое внимание уделяется таким вопросам, как информационные технологии и системы, информационные меры, теория кодирования, основам программирования на языке Паскаль.

1. БАЗОВЫЕ СВЕДЕНИЯ ОБ ИНФОРМАТИКЕ, ПРОГРАММАХ И ЭВМ

1.1 Введение в предмет информатика. Краткая характеристика ее основных направлений

Информатика – это наука, изучающая все аспекты получения, хранения, преобразования, передачи и использования информации. Информатика как наука стала развиваться в середине XX столетия с появлением ЭВМ. Термин «информатика» был заимствован из французского языка в 70-х гг.

Информатика состоит из следующих научных направлений:

теоретическая информатика;

кибернетика;

программирование;

искусственный интеллект;

информационные системы;

вычислительная техника;

информатика в обществе;

информатика в природе.

Дадим краткую характеристику каждого из этих направлений.

Теоретическая информатика (ТИ) – математическая дисциплина, которая используют методы математики для построения и изучения модели обработки, передачи и использования информации, создается теоретический фундамент для всех разделов информатики. На рис.1.1 представлены дисциплины, которые изучаются в ТИ.

В рамках *математической логики* разрабатываются методы, позволяющие использовать достижения логики для анализа процессов обработки информации с помощью ЭВМ (теория алгоритмов и параллельных вычислений), а также методы, с помощью которых можно на основе модели логического типа изучать процессы, протекающие в компьютере во время вычислений (теория автоматов, теория сетей Петри).

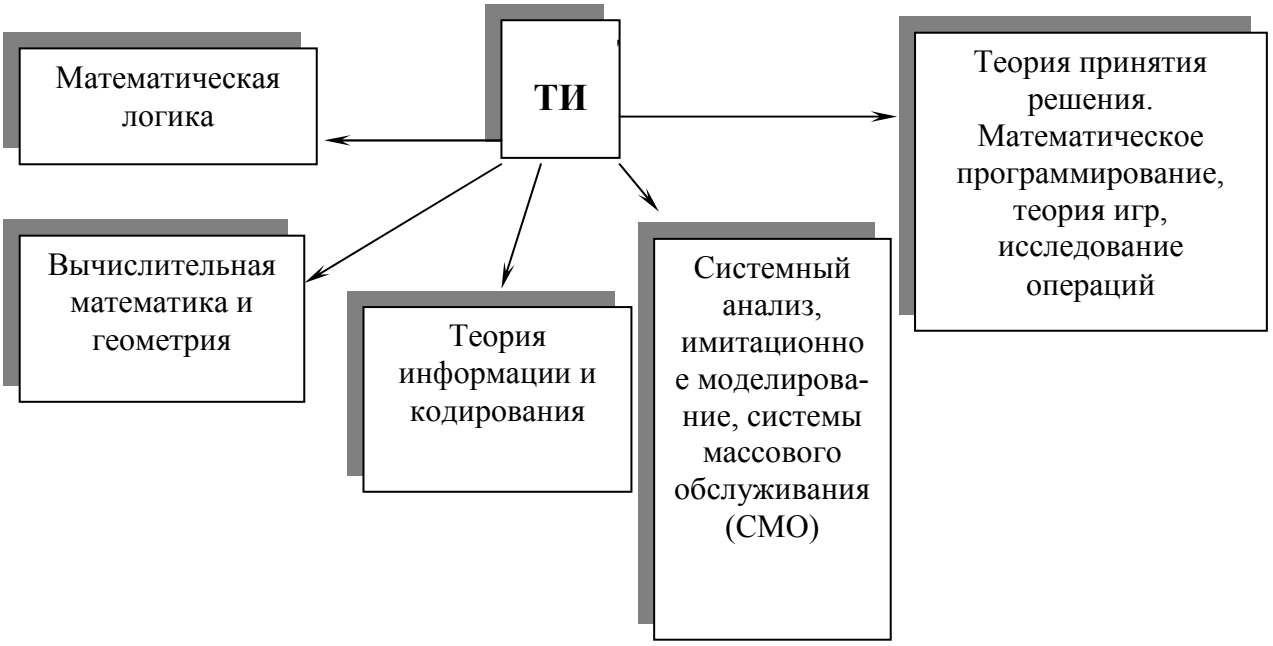


Рис 1.1 Дисциплины теоретической информатики

Вычислительная математика и геометрия направлены на создание методов, ориентированных на реализацию их на компьютере.

Теория информации и *теория кодирования* изучают информацию в виде абстрактного объекта, лишенного конкретного содержания, выявляют общие свойства информации, законы, управляющие ее возникновением, развитием и уничтожением, а также изучаются вопросы передачи информации по каналам связи, кодирование и декодирование информации при ее посылке. Теория информации использует понятия и методы теории вероятностей. Причем в ней не придерживаются специальной

«информационной» терминологии. Понятие сообщения можно заменить на понятие случайной величины, понятие последовательности сообщений – на случайный процесс и так далее.

Начало теории информации было положено в 1928 г, когда американский ученый Хартли опубликовал статью « Передача информации» в журнале «Белл систем техникл». В этой статье он предложил логарифмическую меру количества информации для равновероятных сообщений. Позже (примерно в 1948 г.) Клод Шеннон предложил другую меру, которая принята в настоящее время. Роль Шеннона состояла в том, что он, систематизируя результаты своих предшественников, определил основные понятия теории информации: источник информации, дискретные и непрерывные сообщения, которые составляют фундамент статистической теории информации. В Шенноновской теории информации появилось хорошо известное в термодинамике понятие – **энтропия**. В ней, казалось, нет места (в ТИ) для энергии и других аналогичных термодинамических потенциалов. В этом отношении теория выглядела однобокой по – сравнению с термодинамикой. После осознания того, что в прикладной теории информации, понимаемой как теория передачи сигналов, аналогом энергии является функция штрафов и т.д., положение изменилось. Термин энтропия прочно вошел в словарь теории информации. Причем понятие количества информации тесно связано с понятием энтропии.

Наряду с перечисленными выше мерами Хартли и Шеннона, существуют и другие подходы к определению количества информации (например, информационная концепция Холмогорова).

Системный анализ - совокупность методов и средств исследования сложных, многоуровневых и многокомпонентных систем, объектов, процессов, опирающихся на комплексный подход, учет взаимосвязей и

взаимодействий между элементами системы. Системный анализ изучает структуру реальных объектов и дает способы их формализованного описания, что дает возможность самые разные системы можно изучать с единых позиций.

Имитационное моделирование создает и использует специальные приемы воспроизведения процессов, протекающих в реальных средах, в тех моделях этих объектов, которые реализуются в ЭВМ.

В рамках направления систем массового обслуживания (*СМО)* изучаются модели передачи и переработки информации, которые в дальнейшем используются при проектировании новых типов ЭВМ, операционных систем и т. д.

Системный анализ, имитационное моделирование, СМО занимают пограничное положение между теоретической информатикой и кибернетикой.

Теория принятия решений изучает общие схемы, используемые людьми при выборе нужного решения из множества альтернативных. Если выбор происходит в условиях конфликта, то приходится обращаться к средствам *теории игр*; если из множества альтернативных решений необходимо выбрать оптимальное – к возможностям *математического программирования*.

Исследование операций – математическая дисциплина, изучающая методы поиска наилучших решений для случаев, когда и сами решения, и факторы, которые надо учесть при их принятии, могут быть выражены в количественной форме или в виде предпочтений.

2*)Кибернетика* (от греч. kybernetike – искусство управления, от kybernáo – правлю рулём, управляю) – наука об управлении, связи и переработке информации в различных системах: технических, биологических, социальных и др.

Рассмотрение различных объектов живой и неживой природы как преобразователей информации или как систем, состоящих из элементарных преобразователей информации, составляет сущность так называемого кибернетического подхода к изучению этих объектов. Современная кибернетика состоит из большого количества разделов, представляющих собой самостоятельные научные направления. Теоретическое ядро кибернетики составляют такие разделы, как теория информации, теория кодирования, теория алгоритмов и автоматов, общая теория систем, теория оптимальных процессов, методы исследования операций, теория распознавания образов, теория формальных языков. В рамках кибернетики развиваются такие научные направления, как бионика, нейрокибернетика, техническая кибернетика и т.д.

3) В рамках направления ***«Программирование»*** изучаются и разрабатываются языки программирования, трансляторы, операционные системы, языки протоколов связи, пакеты прикладных программ, банки данных.

4) ***Искусственный интеллект*** **–** область науки и техники, связанная с компьютерным моделированием и изучением интеллектуального поведения, а также с созданием устройств, которые обладают таким поведением.

5) В рамках направления ***информационных систем*** решаются следующие задачи:

1) анализ и прогнозирование потоков информации в обществе с целью минимизации, стандартизации и приспособление эффективной обработки на ЭВМ;

2) исследование способов хранения и представление информации создание специальных языков для формального описания информации; разработка приемов сжатия и кодирования; аннотирование и реферирование документов;

3) построение различных процедур и технических средств для их реализации с помощью которых можно автоматизировать процесс извлечения информации из документов, не предназначенных для ЭВМ;

4) создание информационно-поисковых систем;

5) создание сетей хранения, обработки и передачи информации.

6) В направлении ***вычислительной техники*** разрабатываются новые структуры ЭВМ, новые принципы их работы, модифицируется и качественно улучшается элементная база, создаются комплексы и сети обработки данных.

7-8) Основные задачи прикладных направлений ***информатики в обществе*** – использование новых информационных технологий в профессиональной деятельности людей, а ***информатики в природе*** – изучение информационных процессов, протекающих в биологических системах, и использование накопленных знаний при организации и управлении природными системами и создание технических систем.

1.2 Общие сведения об информации

Прежде всего, определимся с терминологией, используемой в информатике (в связи с чем дадим основные определения).

Из определения информатики можно увидеть, что центральное место в информатике занимает понятие **информации**. Термин "информация" происходит от латинского слова informatio, что означает разъяснение, осведомленность, изложение. Человек использовал информацию уже в самом начале своего существования. Доисторический период характерен тем, что люди добывали пищу в основном за счет охоты. Поэтому они криками предупреждали друг друга об опасности, свистом, стуком передавали какие-то сведения, то есть какую-то информацию. В дальнейшем появилась потребность расширить арсенал звуков, поэтому

стала появляться осмысленная речь, а с ней и возможность лучше передавать и получать информацию. Затем человек занимался земледелием, приручал и использовал в своем хозяйстве диких животных, начал производить то, что ему необходимо, а не брать готовое у природы. Так происходил на ранних этапах прогресс человечества. При этом значение информации постоянно возрастало. И потребности человека в получении новой информации тоже постоянно возрастали. Можно сделать вывод, что этот процесс начался с появлением человека, продолжается до сих пор, и будет продолжаться вместе с дальнейшим развитием человеческого общества.

Однако, при изучении информатики как научной дисциплины, важно иметь точное определение информации. Без этого будет невозможным глубокое понимание предмета изучения.

Информация является абстрактной категорией и связана с процессом познания человеком окружающего мира. Поэтому понятие "информация" используется в различных смыслах в зависимости от конкретной области приложения. Так, например, в философии говорится, что "информация есть отражение реального мира". Другие науки, наоборот, значительно сужают понятие информации. Это делается для того, чтобы придать сугубо практический, конкретный смысл этому понятию и использовать только в рамках своей дисциплины. Под информацией *в технике* понимают сообщения, передаваемые в форме знаков или сигналов. Под информацией *в теории информации* понимают не любые сведения, а лишь те которые, снимают полностью или уменьшают существующую неопределенность. Под информацией *в кибернетике*, по определению Н. Винера понимают ту часть знаний, которая используется для ориентирования, активного действия, управления, т.е. в целях сохранения, совершенствования, развития системы. В *информатике* **информацию** понимают как

абстрактное значение выражений, графических изображений, указаний и высказываний.

Различные трактовки термина «*информация»* можно обобщить в два основных подхода к выяснению его сущности: атрибутивный; функционально – кибернетический. *Первая* (атрибутивная) точка зрения трактует информацию как свойство движения материи, состоящее в структурности, упорядоченности, разнообразии, организации и др. ее состояний. Отличительная черта этой концепции: признание всеобщности той стороны реальных объектов и процессов, которая отражается в понятии информации. *Второй* подход рассматривает информацию как свойство определенного класса материальных систем, которое возникает и обогащается в процессе становления, развития этих систем, как свойство их функционального взаимодействия между собой и внешним миром. Такими системами являются – живые организмы и их сообщества; человек и человеческое общество. Они образуют класс систем, которые получили название самоуправляющихся и самоорганизующихся.

Многообразие подходов к трактовке термина информация сохраняет актуальность задачи их систематизации и обобщения.

Информацию можно рассматривать также в следующих **аспектах**:

1) семантическом (содержание или значение информации);
2) аксиологическом (ценность информации для самоуправляемых систем);
3) семиологическом (обозначение информации в определенной знаковой системе);
4) коммуникативном (информационные связи);
5) теоретико-отражательном (восприятие информации объектом);
6) гносеологическом (информация как средство познания);

7) физическом (материальное воплощение информации) и так далее.

Анализируя информацию, мы сталкиваемся с необходимостью оценки ее **качества** и **количества**. Существующие различные подходы к оценке количества информации (структурный, статистический и семантический методы).

За минимальную единицу измерения количества информации принято принимать бит. ***Бит*** – минимальная единица измерения информации (в двоичной системе 0 или 1).

Более точные определения бита и информации будут даны ниже. На практике, помимо бита для демонстрации информационной емкости, пользуются ***байтами*** и кратными им величинами:

1 байт = 8 бит[1]

1 Кбайт = 2^{10} байт = 1024 байт

1 Мбайт = 2^{10} Кбайт = 1024 Кбайт

1 Гбайт = 2^{10} Мбайт = 1024 Мбайт

1 Тбайт = 2^{10} Гбайт = 1024 Гбайт

Информационный объем сообщения – количество бит в этом сообщении.

Бит в секунду – единица измерения скорости передачи информации[2].

1 кбит/с = 1000 бит/с

1 Мбит/с = 1000 Кбит/с

1 Гбит/с = 1000 Мбит/с

1 Тбит/с = 1000 Гбит/с

[1] Помимо 8-битного представления байта, существуют иные подходы. Так, например, в знаменитом, ставшем классикой, трехтомнике по программированию «Искусство программирования» Дональда Кнута, в одном байте - 6 бит. Потому и определение этому понятию дают более общее: байт – это минимально адресуемая область памяти.

[2] В отношении скорости передачи информации существуют разногласия по поводу того, каким образом представлять кратные величины: степенью двойки, или десятичным множителем. Однако наиболее распространенным является подход образования кратных величин согласно правилам, принятым в системе СИ (умножение на 10). При работе с байтами же, приходится чаще иметь дело со степенями двойки.

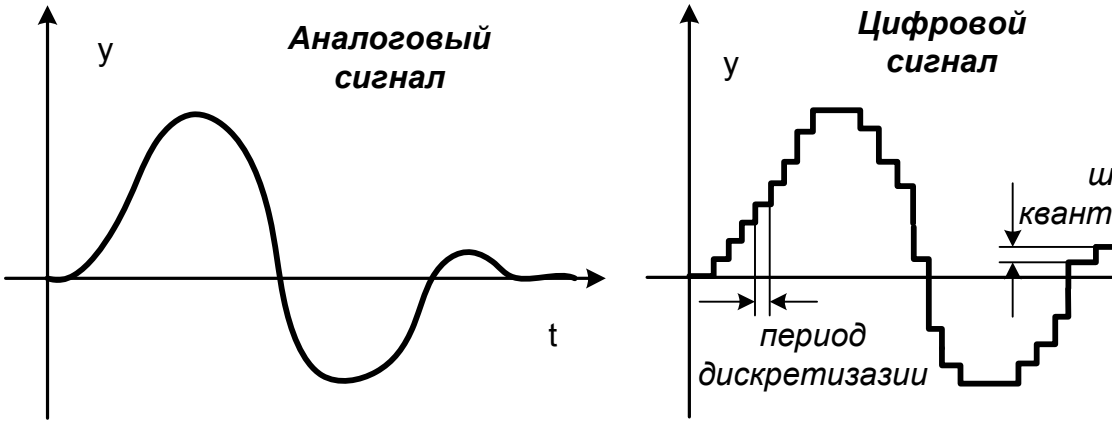


Рис. 1.2 Аналого-цифровое преобразование сигнала

Вообще, процессы, происходящие в физическом мире, принято считать непрерывными. Для того чтобы можно было говорить об их информационной обработке, необходимо физическому процессу сопоставить число. Процесс сопоставления чисел непрерывным физическим процессам называется *оцифровкой*. Очевидно, что оцифровка не полностью отражает реальный процесс. При оцифровке часть информации теряется. Однако, как говорилось ранее, способ задания информативности зависит от конкретной системы. Если нам не нужно высокое разрешение сигнала, то достаточно ограничиться малым.

Оцифровку можно наглядно представить, например, при переводе аналоговой записи (магнитофонной кассеты) в цифровую форму (рис. 1.2). Видно, что через равные промежутки времени мы снимаем показания уровня сигнала, причем шкала, по которой этот уровень снимается, имеет конечную разрешающую способность, что выражается в «ступенчатости» графика. Величина ступеньки по оси y называется разрешением по уровню или *шагом квантования*, а длина ступеньки вдоль оси t - *периодом дискретизации*.

1.3 Двоичная система счисления

Исторически сложилось, что люди используют десятичную систему счисления, т. е. систему счисления, в которой цифровой алфавит состоит из десяти цифр: 0–9. Это связано с тем, что у человека 10 пальцев на руках, которые он с давних времен использовал для счета.

Прежде всего, рассмотрим отличия двух понятий – «цифра» и «число».

Цифры – символы, с помощью которых можно записать число.

Число – смысл (количественное значение), вкладываемый в запись, состоящую из одной или нескольких цифр.

Рассмотрим, по какой схеме формируются натуральные числа по порядку в десятичной системе счисления. Выбирается по очереди весь цифровой алфавит этой системы. Первые десять чисел от (0 до 9) совпадают по написанию с десятью цифрами алфавита десятичной системы, но мы можем записать эти числа не 0, 1, 2, …, 9, а, например, 00, 01, 02, …, 09. Тем самым мы поставили на первое место цифру ноль (незначащий ноль), что мы можем сделать перед любым числом, не меняя его количественного значения. Справа же мы перебрали весь цифровой алфавит. Чтобы записать числа 10, 11 и т. д., на первом месте пишут 1, а справа перебирают весь цифровой алфавит, тем самым, получая числа от 10 до 19. Для получения числа 20 ставят 2 на первое место, и т. д., пока не переберут на первом месте весь цифровой алфавит. Затем слева добавляют третий разряд и перебирают на нем весь цифровой алфавит по тем же правилам, которые применялись для формирования двузначных чисел.

Аналогичным образом формируются натуральные числа в любой другой системе исчисления, например, двоичной (табл. 1.1). ***Двоичной системой*** называют систему исчисления, в которой не 10, а лишь две цифры: 0 и 1.

Таблица 1.1

Соответствие десятичных и двоичных чисел

Десятичная запись	Двоичная запись	Десятичная запись	Двоичная Запись
0	0	8	1000
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111
		16	10000

Вся информация в памяти компьютера хранится в цифровом представлении, в том числе и звук, и графика, и текст, не говоря уж, собственно, о числах. Для представления нечисловой информации в цифровом виде используют различные виды кодировок.

Для кодирования графики изображение разбивают, например, на строки и столбцы, на пересечении которых располагаются точки, называемые *пикселями*. Количество пикселей выбирают из требуемого качества изображения, а также из предполагаемого размера экрана монитора или листа бумаги, на котором будет в дальнейшем представлено это изображение. Очевидно, что чем больше точек, тем качественнее изображение. В связи с эти существует, если можно так сказать, единица измерения качества изображения – *dpi* (*dot per inch*) – точек на дюйм; чем выше эта величина, тем качественнее изображение, но больше места занимает в памяти компьютера. Присутствие черной точки (или ее отсутствие) позволяет строить лишь черно-белое изображение, такое, как, например, с использованием аналогового копира. Чтобы получить цветное изображение, необходимо измерять цвет каждой точки. Средний человеческий глаз отличает до 10 млн. оттенков цветов. В связи с этим

цвет каждой точки достаточно закодировать с помощью 10 млн. чисел, чтобы человеческий глаз не отличил «искусственности» изображения. Обычно цвет кодируют с помощью $2^{16} = 65536$ (*High Color*) или $2^{24} = 16\,777\,216$ (*True Color*) чисел.

Для кодирования текста используют так называемые таблицы кодировок, в которых каждому символу (в том числе и буквам) поставлен в соответствие какой-либо код (число). Примером подобной таблицы может служить 8-битная таблица *ASCII* (говорят «*АСКИ*») (в которой описано 256 кодов, соответствующих, в том числе, латинским и русским буквам) или 16-битная таблица *Unicode* (содержащая символы алфавитов практически всех языков мира).

Итак, мы рассмотрели, как может быть перекодирована различная нечисловая информация для представления в памяти ЭВМ. Разберемся теперь, как перекодировать число из десятичной в двоичную систему исчисления.

Рассмотрим подробнее табл. 1.1. Обратим внимание на числа 1, 2, 4, 8, 16. Их можно записать как 2^0, 2^1, 2^2, 2^3, 2^4. В двоичной записи им соответствую числа 1, 10, 100, 1000, 10000. Посчитаем количество нулей каждого двоичного числа и обратим внимание на показатель степени соответствующего десятичного. Получается, что количество нулей в двоичных числах совпадает с показателем степени двойки в десятичном представлении числа. Можно сделать следующий вывод: всякое круглое двоичное число можно представить в десятичном виде как 2 в степени, равной количеству нулей, стоящих после 1 в двоичной записи числа.

Рассмотрим один из возможных способов перевода десятичного числа в двоичный вид.

ПРИМЕР

Возьмем число 1247 для перевода в двоичный вид. Договоримся, в десятичном виде число записывать – как есть, а в двоичном виде указывать нижний индекс 2.

Составим таблицу натуральных степеней двойки (табл. 1.2).

Таблица 1.2

Целые степени двойки

n	0	1	2	3	4	5	6	7	8	9	10	11
$\mathbf{2}^n$	1	2	4	8	16	32	64	128	256	512	1024	2048

Разложим число 1247 на степени двойки. Для этого сначала найдем в табл. 1.2 число, ближайшее к 1247 снизу; таким числом является 1024.

1247=1024+223.

Теперь найдем в табл. 1.2 число, ближайшее к 223 снизу, – число 128 и т. д., т. е.:

1247=1024+128+95=1024+128+64+31=1024+128+64+16+15=

$=1024+128+64+16+8+4+2+1=2^{10}+2^7+2^6+2^4+2^3+2^2+2^1+2^0=$

$=10000000000_2+10000000_2+1000000_2+10000_2+1000_2+100_2+10_2+1_2$

Для того чтобы сложить эти числа, запишем их в столбик:

$$
\begin{array}{r}
+10000000000 \\
+10000000 \\
+1000000 \\
+10000 \\
+1000 \\
+100 \\
+10 \\
\underline{\qquad\qquad 1} \\
10011011111
\end{array}
$$

Итак, получили $1247=10011011111_2$.

Возвращаясь к определению бита, переформулируем его так: ***бит*** – один разряд двоичного числа.

1.4 Основные положения теории кодирования

Теория кодирования – это раздел теории информации, связанный с задачами кодирования и декодирования сообщений, посылаемых из источника к приемнику информации. Эти задачи теория кодирования решает с учетом задачи наилучшего согласования посылаемой информации с каналами связи.

Рассмотрим основные понятия и определения теории кодирования.

Код – это система соответствий между элементами исходного сообщения и сочетаниями символов, при помощи которых эти сообщения могут быть зафиксированы и при необходимости переданы на расстояние или использованы для дальнейшей обработки; это совокупность условных символов (сигналов), обозначающих определенное сообщение; это множество слов в некотором алфавите, поставленное во взаимнооднозначное соответствие другому множеству.

Цель кодирования – представить информацию в более компактной форме для дальнейшей передачи и обработки.

Условные сигналы, составляющие код, называют *кодовыми комбинациями* (словами). Число элементов или знаков, образующих кодовую комбинацию, называют *значимостью* кода. Например, 0111 – 4-значный код; 01 – 2-значный код.

Алфавит, в котором представлено исходное сообщение, называется *первичным* алфавитом, а алфавит, в котором представлено закодированное

сообщение, - *вторичным* алфавитом. Таблицу соответствий между совокупностью используемых сообщений и кодовыми комбинациями, называют *первичным* кодом.

Приведем один из вариантов ***классификации*** кодов по различным признакам.

1) По основанию вторичного алфавита:

- двоичные коды:
- троичные коды и т.д.

2) По длине кодовых комбинаций:

- равномерные (все комбинации имеют одинаковую кодовую длину);
- неравномерные (кодовые комбинации имеют различное количество знаков).

3) По способности обнаружения ошибок:

- коды без обнаружения ошибок;
- коды с обнаружением ошибок;
- коды с исправлением ошибок.

В случае, если код является неравномерным, необходимо решать проблему правильного (однозначного) декодирования кодового сообщения, поскольку коды делятся на два класса:

- Обратимые или префиксные (без запятой). В этом случае кодовые комбинации разных сообщений различны и любая более короткая кодовая комбинация не является началом другой более длинной кодовой комбинации. Например, 01; 001; 1111.
- Необратимые. Более короткая кодовая комбинация может является началом другой более длинной кодовой комбинации. Поэтому этот код требует специальных разделительных знаков, которые ставятся между кодовыми комбинациями для правильного декодирования. Например, в коде Морзе пауза является разделительным знаком. Для оценки и

сравнения различных кодов применяется такое понятие, как *экономичность* кода. Эта величина измеряется при помощи максимального числа элементарных знаков, требующегося для кодирования одной буквы исходного сообщения. Чем меньше это максимальное число, тем более экономен код. На практике экономность кода определяют подсчетом *среднего числа* элементарных знаков, приходящихся на одну кодируемую букву.

В теории кодирования фундаментальное значение имеют две теоремы, доказанные К. Шенноном. Первая теорема говорит о том, что при канале, не вносящем своих помех, можно закодировать сообщение таким образом, чтобы среднее число элементов кода, было бы минимальным (этот минимум определяется энтропией источника информации). Другими словами, среднее число двоичных элементарных сигналов, приходящихся в закодированном сообщении на одну букву исходного сообщения, не может быть меньше энтропии H.

Вторая теорема Шеннона относится к каналам с помехами. Согласно этой теореме, для таких каналов всегда существует способ кодирования, при котором сообщения будут передаваться с какой угодно достоверностью, если только скорость передачи не превышает пропускной способности канала связи.

Рассмотрим два наиболее распространенных метода кодирования на примере двоичных кодов.

В **методе кодирования *Шеннона – Фано*** (1949 г.) изначально считается, что буквы статистически не связаны между собой, причем получаемый код является неравномерным и обратимым. Код строится следующим образом: символы алфавита сообщений выписываются в таблицу в порядке убывания вероятностей. Затем (в случае двоичного кода) они разделяются на две группы так, чтобы суммы вероятностей в

каждой из групп были по возможности одинаковыми. Всем буквам верхней половины в качестве первого символа присваивается 0, а второй половины – 1. Каждая из полученных групп, в свою очередь разбивается на две подгруппы с одинаковыми суммарными вероятностями и т. д. Процесс повторяется до тех пор, пока в любой подгруппе не останется по одной букве.

ПРИМЕР

Закодировать алфавит A={$A_1 \ldots A_5$} двоичным кодом, если вероятности букв следующие:

$p(A_1) = 1/4, \quad p(A_2) = 1/4, \quad p(A_3) = 1/4, \quad p(A_4) = 1/8, \quad p(A_5) = 1/8.$

Решение:

A_i	p(Ai)			
A_1	1/4	0	0	
A_2	1/4		1	
A_3	1/4		0	
A_4	1/8	1		0
A_5	1/8		1	1

В результате получаем:

$A_1 - 00, \quad A_2 - 01, \quad A_3 - 10, \quad A_4 - 110, \quad A_5 - 111.$

Для оценки эффективности (экономичности) неравномерного кода применяется не длина отдельных кодовых слов, а средняя их длина, которая вычисляется по следующей формуле:

$$\bar{l} = \sum_{i=1}^{n} l_i \cdot p(A_i),$$

где n – общее число сообщений, l_i – длина кодового обозначения для сообщения, $p(A_i)$ – вероятность.

В нашем примере средняя длина кодовых сообщений равна 2.25.

По методу Шеннона-Фано получается, что чем более вероятно сообщение, тем быстрее оно образует самостоятельную группу и тем более коротким кодом оно будет представлено. В результате, хотя некоторые кодовые обозначения могут иметь весьма значительную длину, среднее значение длины такого обозначения оказывается лишь немногим большим минимального значения H, допускаемого соображениями сохранения количества информации при кодировании. Метод Шеннона-Фано выгоден при блочном кодировании.

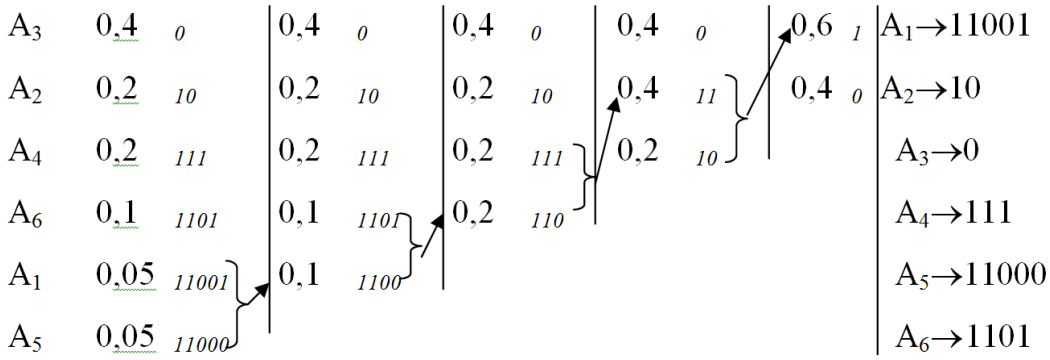
Близок к коду Шеннона-Фано, но еще выгодней, чем последний, так называемый код *Хаффмана*. Получаемый по этому методу код является неравномерным и обратимым.

Суть **а*лгоритма* кодирования по методу Хаффмана** заключается в следующем. Буквы алфавита A выписываются в основной столбец по убыванию их вероятностей. Две последние буквы объединяются в одну вспомогательную букву. В результате получается новый алфавит A_1 из алфавита A путём однократного *сжатия* последнего. Затем вероятности букв, не участвующих в объединении, и полученная суммарная вероятность, снова располагаются в порядке убывания вероятности и процесс сжатия алфавита A_i повторяется до тех пор, пока не получим единственную вероятность буквы, равную единице. При этом условимся приписывать буквам алфавита A_i значения 1 или 0.

ПРИМЕР

Закодировать по методу Хаффмана буквы алфавита, имеющие следующие вероятности: A={0,05; 0,2; 0,4; 0,2; 0,05; 0,1}.

Решение:



Подсчитаем среднюю длину кодовых сообщений:

$$\overline{l_{cp}} = 5 \cdot 0{,}05 + 2 \cdot 0{,}2 + 1 \cdot 0{,}4 + 3 \cdot 0{,}2 + 5 \cdot 0{,}05 + 4 \cdot 0{,}1 = 2{,}3 \ .$$

Кодирование некоторого алфавита по методам Хаффмана и Шеннона-Фано не является однозначно определённой процедурой. Так, например, на любом этапе построения кода можно заменить цифру 1 на 0 и наоборот. При этом получим два разных кода, отличающихся, правда, весьма несущественно друг от друга и имеющих те же длины кодовых комбинаций.

В теории кодировании принят следующий ***тезис:*** ни для какого другого метода кодирования букв некоторого алфавита среднее число элементарных сигналов, приходящихся на одну букву, не может быть меньше того, какое получается при кодировании по методу Хаффмана.

Методы кодирования Шеннона-Фано и Хаффмана можно использовать для построения m-ичных кодов.

1.5 Информационные меры

Данные выше определения, связанные с понятием информации, не совсем удачны, поскольку выражают некое субъективное отношение к информатике, в общем, и к информации, в частности. Субъективизм, конечно, вполне оправдан, однако, его нужно почувствовать. Информатика является наукой точной и потому требует более строгого подхода к

определению своего ключевого понятия. Для этого следует сделать информацию измеряемой. Тем более, что важный вопрос теории передачи и преобразования информации – это установление меры количества и качества информации. Информационные меры, как правило, рассматриваются в трех аспектах: структурном, статистическом и семантическом.

Структурная мера информации. В этом аспекте рассматривается строение информационных массивов, причем их измерение производят либо простым подсчётом информационных элементов, либо с помощью комбинаторных методов. Структурный подход применяется для оценки возможностей информационных систем вне зависимости от условий их применения.

Суть данного подхода заключается в следующем. Рассматривают информацию только в виде дискретного *сообщения*, элементарной единицей которого является *символ.* Сообщение, оформленное в виде слов или отдельных символов, всегда передается в материально-энергетической форме.

При структурном подходе различают *геометрическую*, *комбинаторную* и *аддитивную меры* информации.

Геометрическая мера предполагает измерение параметров геометрической модели информационного сообщения (длина количество, площадь, объём) в дискретных единицах.

Максимально возможное количество информации в заданных структурах определяет информационную ёмкость модели, которая оценивается как сумма дискретных значений по всем координатам.

В *комбинаторной мере* количество информации определяется как число комбинаций элементов (символов). Оцениваемое количество

информации совпадает с числом возможных перестановок, сочетаний и перемещений элементов.

*Аддитивная мера (мера Хартли).*Этот подход был впервые осуществлен Хартли[3] в 1928 г. Пусть у нас имеется некий канал, по которому мы получаем сообщение в виде двоичных кодов (есть сигнал – «1», нет сигнала – «0»). Иллюстрация к описанному процессу представлена на рис. 1.2. Предположим, что наше сообщение состоит из четырех принятых цифр. Тогда полное количество различных его вариантов есть 2^4 = 16. Допустим, что наше сообщение хотят перехватить. Зададимся вопросом: «Какова неопределенность полученного нами сообщения при известном числе переданных знаков?». Если мы получаем 4 знака, то неопределенность равна 16. А если мы к этим 4-м знакам прибавим, скажем, еще 4, то неопределенность составит 16×16=256. Вот как получается, если просто прибавить к исходному сообщению еще несколько знаков; общая неопределенность сообщения умножается на неопределенность, внесенную этими знаками. Хотя логичным кажется не умножение, а складывание неопределенности.

В математике для ухода от операций умножения к эквивалентным операциям сложения часто применяют логарифмирование. Действительно, логарифм произведения равен сумме логарифмов:

$$\ln\left(p_1 \cdot p_2 \cdot \ldots \cdot p_k\right) = \ln\left(p_1\right) + \ln\left(p_2\right) + \ldots + \ln\left(p_k\right).$$

Согласно Хартли, количество информации, находящейся в одном сообщении, кодированном «0» или «1», равно

$H = \log_2\left(1/p\right)$, где p – вероятность возникновения некоторого события. Если событие состоит из последовательности равновероятных, то общая вероятность их последовательности будет $p = p_1 \cdot p_2 \cdot \ldots \cdot p_k$. Для

[3] Дуглас Хартли (1897-1958) – английский физик, занимался вопросами вычислительной математики, квантовой механики. Один из пионеров внедрения в Великобритании цифровых ВМ.

равновероятного выпадения «0» или «1» это будет $p = \frac{1}{2} \cdot \frac{1}{2} \cdot \ldots \cdot \frac{1}{2} = \frac{1}{2^k}$, и $H = \log_2\left(2^k\right) = k$.

Если кодирование происходит большим числом знаков, скажем тремя: «0», «1» и «3», а мера неопределенности сообщения есть $W = \frac{1}{p}$, то информационная емкость сообщения будет определяться как

$H = \log_3\left(W\right)$.

Однако на практике наибольшее распространение получила система, основанная на двоичном счете, и потому для бинарного сообщения, состоящего из k, знаков информационная емкость составляет

$$H = \log_2\left(2^k\right) = k. \qquad (1.1)$$

Эта формула справедлива только для таких сообщений, у которых возникновение «0» или «1» на любом этапе приема сообщения равновероятно. Это происходит далеко не всегда.

Так, например, для сообщения, состоящего из четырех двоичных знаков, информационная емкость составит $H = \log_2\left(2^4\right) = 4$.

Если, к примеру, мы пишем текст на русском языке и пользуемся для его передачи кодировкой *ASCII*, то каждый передаваемый символ будет содержать 8 бит информации. Общая мера неопределенности составит 256 вариантов на символ. Очевидно, что наиболее часто встречающимся символом будет «*пробел*», который имеет номер $32_{10}=20_{16}=100000_{2.}$

Особенностью любого языка (и русского, в частности) является неравномерность вероятности появления различных букв. Так, например, буква «о» в русском языке является наиболее частой, а буква «ф» наиболее редкой.[4] Т. е. если мы будем передавать, скажем, роман Льва Николаевича Толстого «Война и мир», то вероятность $p_{"1"}$ встретить в двоичном коде сообщения «1» будет отличной от вероятности $p_{"0"}$ встретить «0». Равные вероятности, для которых применима формула (1.1), - это такие вероятности, при которых $p_{"1"} = p_{"0"} = 1/2$, так как полная вероятность единица (т. е. $p_{"1"} + p_{"0"} = 1$).

Статистическая мера информации. В статистической теории информации вводится общая мера информации, в соответствие с которой рассматривается не само событие, а информация о нем. Этот вопрос был глубоко проработан К. Шенноном в работе «Математическая теория связи». Если появляется сообщение о часто встречающемся событии, вероятность появления которого близка к единице, то такое сообщение для получателя малоинформативно. Столь же малоинфоративны сообщения о событиях, вероятность появления которых близка к нулю.

События можно рассматривать как возможные исходы некоторого опыта, причем все исходы этого опыта составляют ансамбль, или полную группу событий. К. Шеннон ввёл понятие неопределённости ситуации, возникающей в процессе опыта, назвав её *энтропией*.

Энтропия (удельная информативность) ансамбля – это количественная мера его неопределённости и, следовательно, информативности, выражается как средняя функция множества вероятностей каждого из возможных исходов опыта.

[4] Ради интереса можете просто взять и посчитать встречаемость букв, скажем, на этой странице.

Именно обобщение для случая $p_{"1"} \neq p_{"0"}$ было введено в 1948 г. К. Шенноном[5]:

$$\begin{cases} J = p_{"1"} \log_2 \left(p_{"1"}^{-1} \right) + p_{"0"} \log_2 \left(p_{"0"}^{-1} \right) = -\left(p_{"1"} \log_2 \left(p_{"1"} \right) + p_{"0"} \log_2 \left(p_{"0"} \right) \right) \\ p_{"1"} + p_{"0"} = 1 \end{cases} \quad (1.2)$$

При $p_{"1"} = p_{"0"}$ $J = 1$. Если же $p_{"1"} \neq p_{"0"}$, то эта величина будет меньше. В частности, при $p_{"1"} = 0$ и $p_{"0"} = 1$, получим $J = 0$.

Если перейти от двоичной формы представления информации к форме представления произвольным числом знаков n, то получим для $i = 1, 2, ..., n$ обобщение выражения (1.2):

$$\begin{cases} J = -\sum_{i=1}^{n} p_{"i"} \log_n \left(p_i \right) \\ \sum_{i=1}^{n} p_{"i"} = 1 \end{cases} \quad (1.3)$$

,

где p_i – вероятность встречи i-го знака.

Если всего было передано m знаков, то нужно просуммировать информацию, переносимую каждым знаком.

Если вероятности $p_{"i"}$ одинаковы для всех цифр, то суммарная информация будет в m раз больше чем значения, полученные по формуле (1.3). Запишем случай для двоичного сообщения:

$$J(m) = -\sum_{k=1}^{m} \left(p_{"1"k} \log \left(p_{"1"k} \right) + p_{"0"k} \log \left(p_{"0"k} \right) \right)$$

$$J(m) = \begin{cases} -m \left(p_{"1"} \log \left(p_{"1"} \right) + p_{"0"} \log \left(p_{"0"} \right) \right) & при\ p_{"1"} = p_{"1"k}, p_{"2"} = p_{"2"k} \\ m & при\ p_{"1"} = p_{"0"} \end{cases}$$

Величину, введенную Шенноном (1.2) и (1.3), часто называют ***энтропия информации***.

---

[5] Клод Шеннон (1916–2001) - американский инженер и математик, один из создателей мат. теории информации.

$I_{ср} = -\sum_{i=1}^{k} p_i \log_2 p_i = H$ -общая формула энтропии.

Таким образом, количественно ***энтропия*** – это удельное количество информации, приходящейся на один элемент сообщения. Она обладает следующими *свойствами*:

1. Энтропия есть величина вещественная, ограниченная и неотрицательная.

2. Энтропия минимальна и равна нулю в том крайнем случае, когда одно из p_i равно 1, а все остальные - нулю. Действительно, если $p_1=1$; $p_2=p_3= .. p_к=0$, *то* $H_{min}= -1*\log_2 1=0$. Это тот случай, когда об опыте или величине все известно заранее и результат не дает новую информацию.

3. Энтропия имеет наибольшее значение, когда все вероятности равны между собой*:* $p_1=p_2=...=p_k=1/k$. При этом $H_{max}=-k*(1/k)*\log_2(1/k) =\log_2 k$, где k – количество исходов опыта.

4. Энтропия бинарных сообщений изменяется от 0 до 1.

Действительно, в бинарном сообщении $к=2$. Тогда $H=-p_1*\log_2 p_1 – p_2*\log_2 p_2$. Обозначим $p_1 = p$, и тогда $p_2 = 1-p$. В результате получаем, что

$$H=-p*\log p – (1-p)*\log(1-p).$$

Проанализируем это выражение. Если p=0, то H=0; если p=1, то H=0. При $p_1=p_2$=0.5: $H= H_{max}$=1. Что и требовалось доказать.

5. Энтропия объекта *AB*, состояния которого образуются совместной реализацией состояний объектов *A* и *B*, равна: $H(AB)=H(A)+H(B)$.

Таким образом, справедливы следующие **выводы**:

1. Если все события равномерны и статистически независимы, то оценки количества информации по Хартли и Шеннону совпадают. Это свидетельствует о полном использовании информационной емкости системы.

2. В случае неравных вероятностей количество информации по Шеннону меньше количества информации, чем по Хартли.

В общем случае следует считать, что нарастание информации есть уменьшение энтропии вследствие опыта. Если неопределенность снимается полностью, то информация равна энтропии.

В случае неполного разрешения имеет место частичная информация, являющаяся разностью между начальной и конечной энтропией: $I=H_1-H_2$, где H_1 – начальная энтропия, H_2 – конечная энтропия.

Максимально возможное количество информации получается тогда, когда полностью снимается неопределенность, причем эта неопределенность была наибольшей – вероятности всех событий были одинаковы. Это соответствует максимально возможному количеству информации, оцениваемому мерой Хартли: $I=\log_2 N=-\log_2 p$, где $p=1/N$, N – число событий, а p – вероятность их реализации в условиях равной вероятности событий.

В общем случае $H<=H_{max}$. Сообщения называются *оптимальными* в смысле наибольшего количества передаваемой информации, если $\Delta H = H_{max} - H \to 0$.

*Замечание***.** Следует различать понятия «количество информации» и «объём информации». Объем данных в сообщении измеряется количеством символов (разрядов) в этом сообщении. В различных сообщениях один разряд имеет различный вес и соответственно меняется единица измерения данных:

в двоичной системе счисления единица измерения – бит;

в десятичной системе счисления единица измерения – дит.

Объём данных зависит от длины сообщения n и количества сообщений k: $Q = kn$. Причем, при повторной передачи одного и того же

сообщения объем данных в сообщении остается неизменным (постоянная величина).

Количество информации уменьшается с числом повторений по экспоненте: $I = -rlog_2П_i$, где r – коэффициент, зависящий от характера повторения, а $П$ – количество повторений, то есть при чтении одного и того же текста его информативность падает, а число знаков остаётся постоянным.

Рассмотрим пример. Необходимо определить объём и количество информации при передаче русского текста из 350 букв (k=350) при помощи пятизначного двоичного кода.

Известно, что энтропия русского алфавита без учета взаимосвязи между буквами равна H = 4,358 (бит/букв). Тогда количество информации можно посчитать по формуле $I=k*H$. I=350*4.358≈1523.3 (бит). Определим теперь объем данных Q. Из условия n=5, тогда $Q=k*n$=350*5=1750 (дв. зн.).

Отсюда видно, что в общем случае $I<=Q$. Укажем ряд условий случаи, при которых количество информации равно объему в соизмеримых единицах:

символы первичного алфавита (в котором представлено исходное сообщение) встречаются в сообщениях с равной вероятностью;

количество символов первичного алфавита является целой степенью двух, в случае, если вторичный алфавит (в котором кодируется сообщение) – двоичный, и целой степени m_2 (m_2 – основание системы счисления вторичного алфавита), если $m_2>2$.

Семантическая мера информации. Семантический аспект предполагает учет смыслового содержания информации, что не затрагивалось в структурной и статистической мерах информации. На этом уровне анализируются те сведения, которые отражает информация,

рассматриваются смысловые связи. Это важно для формирования понятий и представлений, выявления смысла, содержания информации и ее обобщения, а также ценности для получателя сообщения.

1.6. Информационные технологии и информационные системы

Термин «***технология***» в переводе с греческого (techne) означает «искусство», «мастерство», «умение», а это не что иное, как процессы. Под *процессом* следует понимать определенную совокупность действий, направленных на достижение поставленной цели.

Информация является одним из важнейших ресурсов общества наряду с такими материальными видами ресурсов, как нефть, газ, а значит, и процесс ее переработки по аналогии с процессами переработки материальных ресурсов можно воспринимать как технологию (рис.1.3).

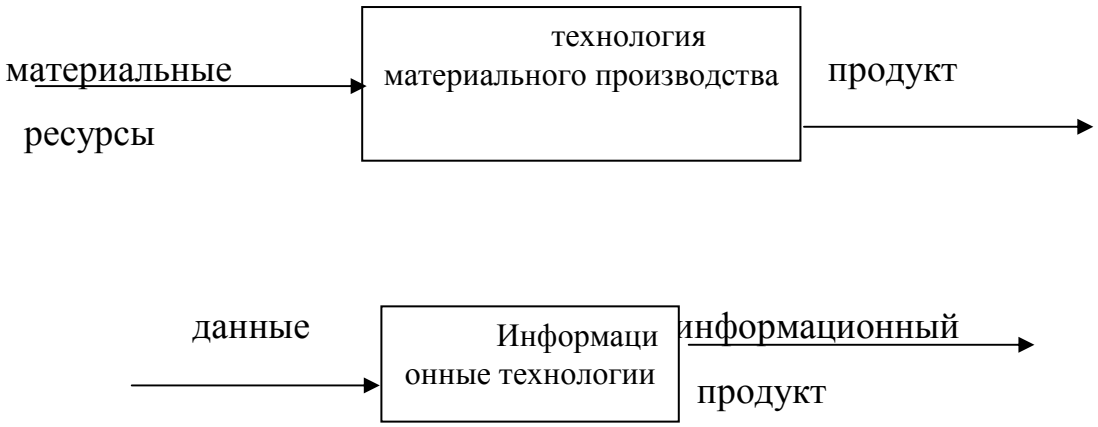


Рис. 1.3. Информационная технология как аналог технологии переработки материальных ресурсов

Информационная технология **(ИТ) –** процесс, использующий совокупность средств и методов сбора, обработки и передачи данных

(первичной информации) для получения информации нового качества о состоянии объекта, процесса или явления (информационного продукта).

Цель информационной технологии **–** производство информации для ее анализа человеком и принятия на его основе решения по выполнению какого-либо действия.

Основу *современной информационной технологии* (компьютерной ИТ) составляют три технических достижения: появление новой среды накопления информации; телекоммуникации; возможность автоматизированной обработки информации с помощь компьютера.

Основными принципами новой (компьютерной) ИТ являются: интерактивный (диалоговый) режим работы с компьютером; интегрированность с другими программными продуктами; гибкость процесса изменения как данных, так и постановок задач.

В истории развития цивилизации произошло несколько ***информационных революций*** – преобразований общественных отношений из-за кардинальных изменений в сфере обработки информации. Первая революция связана с изобретением письменности; вторая (середина XVI века) вызвана изобретением книгопечатания; третья (конец XIX века) обусловлена изобретением электричества, благодаря которому появились телеграф, радио и т.д.; и четвертая (70-е годы XX века) связана с изобретением микропроцессорной технологии и появлением персонального компьютера.

Можно выделить следующие этапы в развитии ***информационных технологий:***

1) до середины XIX века – «ручная» информационная технология; инструментарий: перо, чернила, книга; цель: представление информации в требуемой форме;

2) с конца XIX века – начало XX века – «механическая» ИТ; инструментарий: телефон, пишущая машинка; цель: представление информации более удобными средствами;

3) 40-е – 60-е года XX века – «электрическая» ИТ; инструментарий: большие ЭВМ, электрические пишущие машинки, ксероксы; акцент в информационной технологии перемещается с формы представления информации на формирование ее содержания;

4) с начала 70-х гг. – «электронная» технология; инструментарий: ЭВМ, автоматизированные системы управления и информационно-поисковые системы; цель: внедрение ИТ в управленческую среду общественной жизни;

5) середина 80-х годов – «новая» («компьютерная») информационная технология; инструментарий: ПЭВМ; цель: использование ПЭВМ в различных сферах деятельности общества, создание глобальных и локальных сетей.

Возможны следующие *классификации информационных технологий:*

1) по типу обработки информации (например, СУБД, табличные процессоры, текстовые процессоры и так далее);

2) по типу пользовательского интерфейса: командный; WIMP; SILK; общественный интерфейс;

3) по степени взаимодействия ИТ между собой: дискретная и сетевая.

В настоящее время, к наиболее *распространенным информационным технологиям* можно отнести следующие: графические, текстовые, табличные процессоры; гипертекстовую технологию; технологию мультимедиа; технологию автоматизации офиса (текстовый редактор, электронные таблицы, СУБД и т.д.); автоматизированные рабочие места

(персональный компьютер, оснащенный развитой системой периферии, устройства для подключения и т. д.

Появление компьютеров наряду с разнообразной информационной теорией создало мощную индустриальную основу для возникновения общества, в котором информация как предмет труда играет все более заметную роль. Значение информатики как науки об информации в современном мире велико, как никогда. Качество и эффективность такого рода работ все в большей степени начинают определять качество и эффективность экономики. По подсчетам академика А.А. Харкевича, суммарный информационный поток возрастает в среднем пропорционально квадрату промышленного потенциала (увеличение вдвое производительных сил требует 4-кратного увеличения информации).

Примерно с середины 70-х годов западные ученые (М. Порет, А. Тоффлер) в полный голос заговорили об « информационном обществе».

Информационное общество – это общество, в котором большинство работающих занято производством, хранением, переработкой и реализацией информации, особенно высшей ее формы – знаний.

Информатизация общества – организационный социально-экономический и научно - технический процесс создания оптимальных условий для удовлетворения информационных потребностей как отдельных людей, так и общественных объединений на основе формирования и использования информационных ресурсов.

Характерные черты информационного общества:

1) решение проблемы информационного кризиса;

2) обеспечение приоритетов информации по сравнении с другими ресурсами;

3) главная форма развития - информационная экономика;

4) в основу общества заложены автоматизированные накопление, хранение, обработка, удаление и использование знаний с помощью новейшей информационной техники и технологии;

5) информационная технология имеет глобальный характер;

6) с помощью средств информатики реализуется доступ к мировым ресурсам.

Кроме положительных моментов возникают и опасные тенденции:

1) большое влияние на общество средств массовой информации;

2) проблема отбора качественной и достоверной информации;

3) трудности адаптации к среде информационного общества.

В период перехода к информационному обществу человек должен иметь определенный уровень культуры общения с информацией.

Информационная культура – умение целенаправленно работать с информацией и использовать ее для получения, обработки и передачи с помощью современных информационных технологий и технических средств.

Информационная культура проявляется в следующих аспектах: в конкретных навыках по использованию технических устройств; в способности использования в своей деятельности новые информационные технологии, базовой составляющей которых являются многочисленные программные продукты; в овладении основами аналитической переработки информации; в умении работать с различной информацией; в знании особенностей информационных потоков в своей области деятельности.

Для информационного общества важны ***информационные ресурсы*** – знания, подготовленные людьми для социального использования в обществе и зафиксированные на математическом носителе.

Информационные ресурсы являются базой для создания *информационных продуктов*.

Информационный продукт – совокупность данных, сформированная производителем для распространения в вещественной или невещественной форме с помощью *информационных услуг*.

Информационная услуга – получение и предоставление в распоряжение пользователя информационных продуктов.

В информационном обществе существует *рынок информационных продуктов и услуг*. ***Рынок информационных продуктов и услуг –*** это система экономических, правовых и организационных отношений по торговле продуктами интеллектуального труда на коммерческой основе.

В связи с применением новой ИТ, основанной на использовании средств связи, компьютеров, широко используется понятие «*информационная система*».

Информационная система – это взаимосвязанная совокупность средств, методов и персонала, используемых для хранения, обработки и выдачи информации в интересах достижения постоянной цели.

Структуру информационной системы составляет совокупность отдельных ее частей, называемых подсистемами. К ним относятся: подсистема технического обеспечения; подсистема математического обеспечения; подсистема программного обеспечения; подсистема информационного обеспечения; подсистема организационного обеспечения; подсистема правового обеспечения.

Информационные системы тесно связаны с информационными технологиями, но это разные понятия. Поскольку ИТ является основной составляющей часть любой информационной системы, то ИТ является более емким понятием, отражающим современное представление о процессах преобразования информации в информационном обществе.

1.7 Программное обеспечение

Следует отметить, что в современных вычислительных системах выделяют два основных компонента: ***программная часть*** (*software*) и ***аппаратная часть*** (*hardware*). На особенностях реализации аппаратной части остановимся чуть позже. Здесь же дадим классификацию *программной части*.

Программное обеспечение (ПО) – это определенный *набор информации,* с которой приходится работать аппаратной части (ЭВМ).

Вся *информация* в ЭВМ делится на две группы:

1. ***Программы*** – последовательность инструкций или команд (*алгоритмов*), предназначенных для выполнения определенных действий.

2. ***Данные*** – объект воздействия программ. Информация называется данными, если существует программа, в которой эта информация поступает на ее вход, либо генерируется на ее выходе (т. е. данными для текстового редактора является текстовый документ). Для мультимедийного плеера данными будет музыка или видео-ролик. А, например, для программы записи информации на диск, данными могут быть другие программы. Так что приведенная классификация достаточно условна и зависит от того, как вся эта информация используется.

ПО принято делить на три группы (рис. 1.3):

1. ***Системное*** – ПО для управления ресурсами и технического обслуживания ЭВМ.

К системному ПО относят, прежде всего, ***операционные системы (ОС)*** – специальный комплекс программ для обеспечения взаимодействия пользователя и аппаратных ресурсов, а также для управления взаимодействием программ с аппаратными ресурсами и между собой.

Операционная система, можно сказать, – это главная программа в компьютере. Для разных аппаратных платформ существуют разные ОС.

Так, например, существуют свои ОС для *сотовых телефонов*, *карманных ПК*, *маршрутизаторов*, *серверов* и прочих сложных устройств, способных производить вычисления. Свои ОС разрабатываются даже для самолетов, автомобилей и другой техники. Для персональных ЭВМ наибольшее распространение получило семейство ОС *Windows*.

Стоит отметить, что в качестве альтернативы *Windows* существуют и другие достаточно удачные ОС, самые известные из которых – это семейство ОС *GNU Linux*. Достоинством *Linux* является бесплатная лицензия на использование и открытый исходный код системы. Кроме этих двух семейств, существуют и другие ОС, способные работать на большинстве персональных ЭВМ. Это, например: *MacOS*, *DOS*, *OS/2*, *QNX*, *FreeBSD*, *OpenBSD* и др.

Кроме ОС, к системному ПО можно отнести ***драйверы*** – специальные программы, которые обеспечивают взаимодействие конкретной ОС и конкретного аппаратного устройства. Например, драйвер видеокарты, драйвер сканера, драйвер платы мониторинга радиационного фона и т. д.

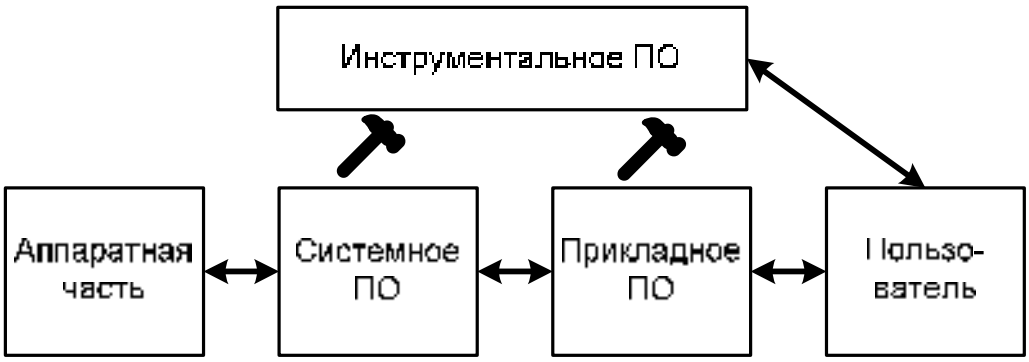


Рис. 1.4. Взаимодействие между различными видами программного обеспечения и человеком

Далее, к системному ПО относят ***программные кодеки*** (*ogg, mp3, mpeg4, mpeg2* и т. д.), различные ***оболочки ОС*** (*файловые менеджеры*,

диспетчеры процессов, *редакторы системного реестра* и т. д.), ***утилиты*** (*дефрагментаторы дискового пространства, средства индексации файлов, оптимизаторы памяти* и т. д.), ***программные средства защиты*** (*антивирусы, межсетевые экраны, средства аутентификации и разграничения доступа, антиспам фильтры* и т. д.)

2. ***Прикладное*** – основная масса ПО, ради которого вообще существует ЭВМ. С помощью этого вида ПО пользователи решают свои задачи в некоторой проблемной области, не прибегая при этом к программированию.

К прикладному ПО относят *офисные приложения*, *системы управления и мониторинга бизнес-процессов*, *системы проектирования и производства*, *научное ПО*, *игры*, *мультимедиа приложения* и т. д.

3. ***Инструментальное*** – ПО для создания нового ПО (как системного и прикладного, так и, собственно, инструментального).

К инструментальному ПО прежде всего относят интегрированные среды разработки (*IDE – Integrated Development Environment*). Это такие среды, как *Eclipse*, *Visual Studio*, *Delphi*, *Builder*, *Turbo C*, *Turbo Pascal*, *Composer Studio*, *Code Warrior* и др. Вышеперечисленные среды являются средствами написания, отладки и компиляции программного кода. Причем, многие из упомянутых IDE в своем составе имеют компиляторы для различных языков программирования (*Pascal, Assembler, Java, C, C++, Visual Basic*). В качестве практических заданий для настоящего пособия выступают различные алгоритмические задачи. Потому именно инструментальным ПО придется в наибольшей степени пользоваться читателю для полноценного усвоения материала. Основным языком программирования выступает *Pascal*, для которого существует достаточно много компиляторов и сред разработки. Тексты всех программ из

настоящего пособия набирались и компилировались в IDE *Turbo Pascal 7.0*.

Следует отметить, что приведенная классификация достаточно условна, поскольку конкретное ПО не всегда можно однозначно отнести в ту или иную группу. Так, например, современные текстовые редакторы имеют в своем составе средства программирования; мультимедийные системы могут, помимо программных плееров, также включать в свой состав драйверы устройств и программные кодеки. И таковых примеров можно привести достаточно много.

1.8 Архитектура персональной ЭВМ

Компьютер (ЭВМ) – универсальное, электронное, программно-управляемое устройство для хранения, обработки и передачи информации.

Архитектура ЭВМ – общее описание структуры и функции ЭВМ на уровне, достаточном для понимания принципов работы и системы команд ЭВМ.

Основные компоненты архитектуры ЭВМ – процессор, внутренняя и внешняя память, и различные периферийные устройства (рис. 1. 4).

Процессор – главное устройство ЭВМ, обеспечивающее обработку и передачу данных, управление внешними устройствами.

Разрядность процессора – число одновременно обрабатываемых битов информации. Различают *разрядность по командам* и *разрядность по данным*. Чем выше командная разрядность процессора, тем больше элементарных команд ему доступно и, соответственно, тем эффективнее он сможет обработать информацию. Чем выше разрядность по данным, тем больший объем памяти доступен процессору для адресации. Если

данные и команды хранятся в разных областях памяти, то такие процессоры называются процессорами с *Гарвардской* архитектурой. Если данные и команды могут храниться в одной области памяти, тогда архитектура будет называться *фон Неймановской*[6]. У Гарвардских процессоров разрядность команд и данных может быть различной. У фон Неймановской же архитектуры разрядности данных и команд одинаковые.

Быстродействие процессора – число выполняемых элементарных операций в единицу времени. Часто используют показатель *MIPS* (*Millions of Instructions Per Second*) – величину, которая показывает сколько миллионов инструкций в секунду выполняет процессор в некотором синтетическом тесте.

Тактовая частота – величина, показывающая сколько раз в единицу времени происходит смена состояния процессора. Имеет линейную связь с производительностью процессора определенной архитектуры. Измеряется в Герцах. Как правило, тактовые частоты современной электроники большие, а потому применяют кратные величины МГц, ГГц.

Память компьютера – электронные ячейки, в которых хранится информация. Делится на внутреннюю (ОЗУ, ПЗУ) и внешнюю (гибкие и жесткие магнитные диски, оптические диски, ленточные накопители, флэш-карты и т. д.).

ОЗУ – оперативное запоминающее устройство, в котором располагаются программы, выполняемые в данный момент, а так же данные, к которым необходим быстрый доступ. При выключении питания

[6] Джон фон Нейман (1903–1957) – немецкий математик и физик, эмигрировавший в США. Занимался вопросами квантовой механики, функциональным анализом, логикой и метеорологией. Большой вклад внес в создание первых ЭВМ и методов их применения. Его теория игр сыграла важную роль в экономике.

компьютера информация в ОЗУ теряется. В английском варианте ОЗУ называется *RAM* (*Random Access Memory* – память произвольного доступа).

ПЗУ – хранит программу начальной загрузки ЭВМ и программы самотестирования. Программа начальной загрузки носит название *BIOS* (*Basic Input/Output System* – базовая система ввода/вывода).

Каналы ввода/вывода – устройства, через которые производится

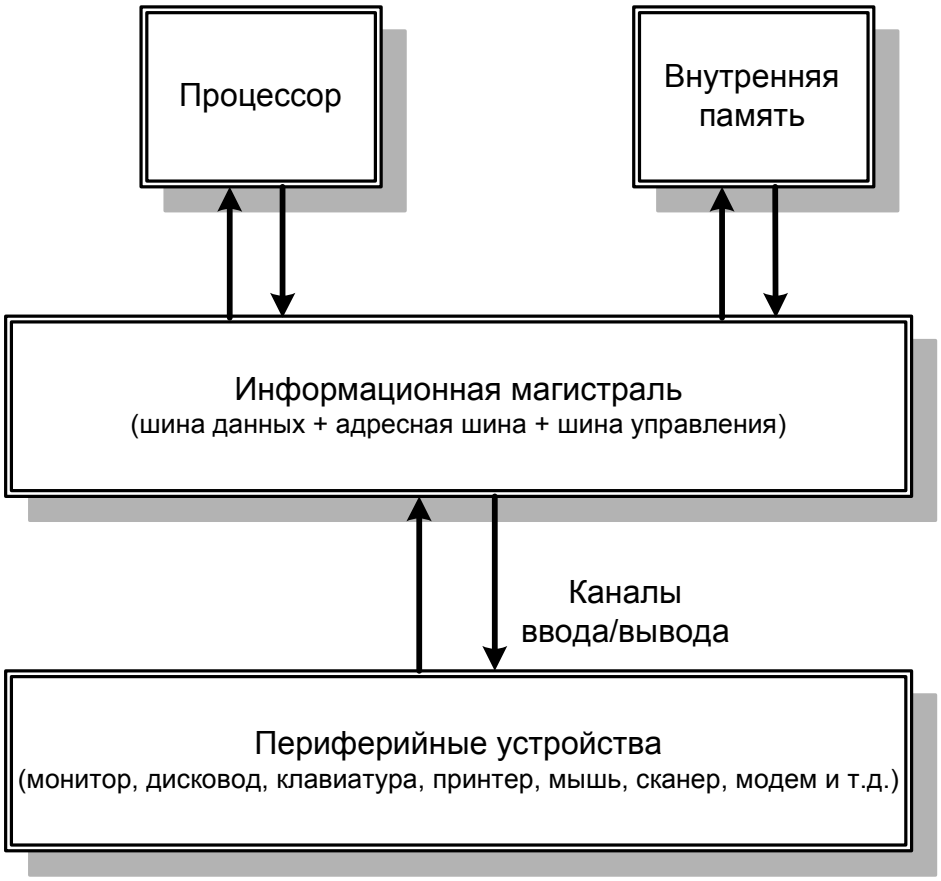


Рис. 1.5. Магистрально-модульный принцип построения компьютера

обмен данными с периферийными устройствами.

Жесткий диск («*Винчестер*») – внешнее запоминающее устройство, служащее для долгосрочного хранения информации. При выключении питания ЭВМ информация на «винчестере» не теряется и может быть в дальнейшем многократно использована. По-английски жесткий диск называют *HDD* (*Hard Diskette Driver*).

2. МЕТОДЫ РЕШЕНИЯ ЗАДАЧ. АЛГОРИТМИЗАЦИЯ. ЛОГИКА

2.1 Этапы решения задач на ЭВМ

При решении любой задачи с использованием ЭВМ принято выделять восемь основных этапов:

1. ***Постановка задачи***. На этом этапе определяются цели, которые необходимо достичь. Уточняются начальные (граничные) условия. Задача делится на несколько подзадач. Из них выделяются те, которые необходимо решать численно (на ЭВМ) и те, которые невозможно или нерационально решать с использованием вычислительной машины.

2. ***Построение математической модели***. Определяются аналитические зависимости различных факторов, влияющих друг на друга, т. е. записываются различные уравнения, как логические (равенства, неравенства, в том числе с использованием логических операций), так и алгебраические.

3. ***Анализ и оптимизация математической модели***. На этом этапе математическая модель приводится к такому виду, который позволяет увеличить скорость расчета при заданной точности.

4. ***Разработка алгоритма***. С использованием полученной математической модели строится вычислительный алгоритм. На данном этапе обычно используют один из *домашинных* способов его описания.

5. ***Программирование***. Полученный алгоритм записывается на одном из машинных языков программирования (*Pascal*, *Delphi*, *C*, *C++*, *C#*, *Fortran*, *BASIC*, *Assembler*, *Java*, *Lisp*, *Python* и т. д.).

6. ***Отладка программы***. Даже самый опытный программист в процессе написания более или менее сложной программы допускает ошибки. Если это синтаксические ошибки, то, например, как это

происходит в *IDE Turbo Pascal*, компилятор сам укажет на них. Но часто возникают и алгоритмические ошибки, например, неверная последовательность вычислительных шагов или использование не той функции, которая необходима. Такие ошибки не видны сразу. Для того чтобы выявить их, необходимо подать на вход тестирующий набор данных и независимо (без использования полученного алгоритма) получить набор выходных данных, например, на калькуляторе. Полученные таким образом выходные данные можно считать эталонными. Если они совпадают с выходными данными, полученными с использованием тестируемого алгоритма, то он работает верно, и программа с определенной степенью вероятности считается отлаженной. Совокупность тестирующих входных данных и эталонных выходных называют *тестовым* или *проверочным примером*. Для всеобъемлющей проверки программы необходимо подготовить такое количество тестовых примеров, которое позволит проверить все ветви тестируемого алгоритма (с учетом всех развилок и циклов).

7. ***Использование программы для получения выходных данных***. Решая поставленную задачу, мы преследуем определенную цель – получение некоторого набора (наборов) выходных данных. На данном этапе эта цель достигается, т. е. проводятся расчеты с использованием полученной программы.

8. ***Интерпретация результатов***. На этом этапе происходит процесс, обратный описанному в *пункте* 1, т. е. результаты решения различных подзадач собираются воедино, из них формируется всеобъемлющий ответ на поставленную задачу и проводится анализ всего решения в целом. Вероятно обнаружение некоторых новых закономерностей или глобальных ошибок, в результате чего возможно повторение каких-либо этапов с учетом полученных уточнений.

2.2 Алгоритмизация

Алгоритм – это строго заданная последовательность шагов, в результате исполнения которых набор *входных данных* преобразуется в набор результатов решения задач (*выходных данных*). Другими словами, алгоритм можно определить как метод или механизм, который предписывает, каким образом можно достичь поставленной цели.

Слово «*algorithm*» произошло от имени аль-Хорезми[7] – автора известного арабского учебника по математике (от его имени также произошли слова «*алгебра*» и «*логарифм*»).

Алгоритм следует отличать от некого *эвристического правила*. Эвристическое правило лишь предлагает, каким образом можно достигнуть цели, но не дает четкой последовательности действий. Например, задача найти дискриминант, чтобы выделить корни квадратного уравнения, является эвристическим правилом для компьютера. Если мы хотим, чтобы она была решена, нужно составить подробную последовательность действий, понятных тому устройству, которое будет ее решать. Таким образом, эвристическим правилом можно назвать некий недетализированный алгоритм, или алгоритм, составленный на языке непонятном машине, которая будет решать нашу задачу.

Классический алгоритм обладает рядом свойств:

1. ***Дискретность***. В один момент времени может выполняться лишь один вычислительный шаг. Одновременное выполнение двух и более шагов невозможно.

2. ***Элементарность каждого вычислительного шага***. Каждый шаг алгоритма должен быть наипростейшим, т. е. его нельзя разделить на более мелкие шаги.

3. ***Детерминизм***. Постоянство результатов при постоянстве входных значений. Обрабатывая несколько раз один и тот же набор входных данных, алгоритм каждый раз должен получать один и тот же набор выходных данных. Но стоит отметить, что этим свойством может не обладать алгоритм, использующий так называемый *генератор случайных чисел* (например, выбор числа из квазислучайной последовательности).

4. ***Массовость***. Алгоритм должен быть универсален на определенном классе задач. Например, алгоритм решения квадратного уравнения должен выдавать результат при решении любого квадратного уравнения, но он не применим при решении дифференциальных уравнений.

5. ***Конечность***. Работа любого вычислительного алгоритма должна быть завершена в обозримое время. К примеру, алгоритм не считается конечным, если результат может быть получен за промежуток времени, сравнимый с возрастом вселенной. Это требование весьма условно, поскольку не всегда удается оценить примерное время работы алгоритма. Более того, бывают принципиально бесконечные алгоритмы, как, например, алгоритм поиска всех простых чисел.

Существует два основных класса способов описания алгоритма: ***домашинные*** и ***машинные***. Выделяют пару наиболее часто используемых домашинных способов: описание с помощью *естественного языка* (например, русского) и с помощью *блок-схем*. *Машинный способ* – описание алгоритма на одном из *языков программирования* (*программа*). Промежуточным вариантом между машинным и естественным способом задания алгоритма могут считаться так называемые *R-схемы* или

[7] Мухаммед бен Муса аль-Хорезми (787 – ок. 850) – среднеазиатский ученый, чьи основополагающие труды по арифметике и алгебре оказали большое влияние на развитие математики в Западной Европе.

синтаксические диаграммы, предложенные Н. Виртом[8] для описания синтаксиса создаваемого им языка программирования *Pascal*.

Естественный язык используется для описания, например, алгоритмов приготовления кулинарных блюд (рецепт), действий рабочих в тех или иных ситуациях на производстве (инструкции), решения квадратного уравнения в математике и т. д. Преимущество данного способа только в том, что для описания алгоритма нет необходимости в каких-то специальных знаниях (специальные обозначения, условные слова и т. д.), но есть ряд существенных недостатков, таких как отсутствие наглядности и проблемы восприятия при чтении более или менее сложных алгоритмов, описанных этим способом.

ПРИМЕР

Рассмотрим пример описания алгоритма решения квадратного уравнения с помощью естественного языка. Уравнение имеет вид:

$ax^2 + bx + c = 0$.

Алгоритм решения квадратного уравнения, описанный с помощью естественного языка:

1) задать значения коэффициентов a, b, c;

2) если $a = 0$, то выполнять *шаг 8*;

3) вычислить дискриминант $D = b^2 - 4ac$;

4) если $D < 0$, то выполнять *шаг 17*;

5) вычислить действительные корни

$$x_1 = \frac{-b + \sqrt{D}}{2a}, \qquad x_2 = \frac{-b - \sqrt{D}}{2a};$$

6) распечатать значения x_1, x_2;

[8] Никлаус Вирт (р. 1934) – швейцарский ученый, ведущий специалист в области информатики. Один из родоначальников структурного программирования. Создатель языков программирования Pascal, Modula-2, Oberon.

7) выполнить *шаг 18*;

8) если $b = 0$, то выполнить шаг 12;

9) вычислить действительный корень $x = -\frac{c}{b}$;

10) распечатать значение x;

11) выполнить *шаг 18*;

12) если $c = 0$, то выполнить *шаг 15*;

13) распечатать комментарий «$a = 0, b = 0, c \neq 0$, $\Rightarrow$, *решений нет*»;

14) выполнить *шаг 18*;

15) распечатать комментарий «$a = 0, b = 0, c = 0$, $\Rightarrow$, «*существует бесконечное множество решений*»;

16) выполнить *шаг 18*;

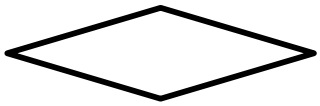
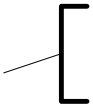
17) распечатать комментарий «*действительных корней не существует*»;

18) окончание алгоритма.

Описание алгоритмов с помощью *блок-схем* основано на графическом представлении последовательности действий. Для этого используют ряд специальных обозначений – *блоков*, которых существует достаточно много. Вообще, программирование относится к инженерной деятельности и потому совершенно естественным является тот факт, что правила составления программной документации регламентированы. Для этого существует целый ряд ГОСТов, которые называются *ЕСПД* (*Единая система программной документации*). Это, прежде всего, ГОСТ 19.002-80, ГОСТ 19.003-80, ГОСТ 19.701-90. Ради упрощения решения задач построения блок-схем, здесь приводится ограниченный набор обозначений (табл. 2.1).

Таблица 2.1

Условные графические обозначения блок-схем

Графическое обозначение	Название и предназначение
	*Блок начала и конца алгоритма (терминатор)* Является точкой рождения и уничтожения потока
	*Блок ввода/вывода* В этом блоке отражаются все диалоги с пользователем (когда у него нужно что-либо спросить или что-нибудь ему показать)
	*Блок обработки данных* Различные действия, скрытые от глаз пользователя
	*Блок логического выражения (предикатный узел)* Здесь записывается проверяемый предикат
	*Блок модификации* Используется для описания заголовка цикла с параметром
	*Блок вызова подпрограммы* Описывает сложный предопределенный процесс, описание которого содержится в том же документе, где он встречается
	*Точки соединения* Точки разрыва и сочленения поточных линий
	Направление выполнения вычислительных шагов (поточные линии) Показывают направление алгоритмического процесса
	*Межстраничный переход* Внутри записывается номер перехода и направление (номер страницы на которую/ с которой осуществлен переход)
	*Комментарии* Пояснения к отдельным блокам или группам блоков

2.3 Понятие переменной и операции присваивания

Выше в примере алгоритма решения квадратного уравнения проводились некоторые вычисления, результаты которых обозначались

букво-цифросочетаниями латинского алфавита. Это так называемые *переменные*. Вообще, понятие *переменной* – одно из ключевых в теории алгоритмизации, особенно в случае вычислительных процессов.

Переменная – именованная область памяти, способная хранить некоторые данные. При этом возможно обращение к ней по ее имени для чтения или модификации содержимого.

Переменным следует давать осмысленные имена, а не обозначать их безликими буквами или непонятным набором символов. Если, например, нужно найти максимум, то и переменной, хранящей его значение, нужно дать имя *Max*. Если это греческая буква, то и ее можно обозначить таким образом, чтобы имелось фонетическое сходство с оригиналом (*Alfa*, *Beta*, *Fi* и т. д.)

Понятие переменной дано; однако есть еще одно понятие программирования, тесно связанное с понятием переменной – понятие операции *присваивания*. ***Операция присваивания*** – операция прямой модификации содержимого переменной. Поскольку это операция, то у нее существуют операнды. Присваивание – операция с двумя операндами. Слева записывается имя переменной, которую подвергают модификации, а справа - выражение, результат которого будет перемещен в область памяти, хранящей содержимое переменной слева.

Операция присваивания всегда происходит *справа налево*. Т. е., вначале вычисляется выражение, стоящее справа, и лишь после этого происходит перемещение результата вычисления в переменную слева.

На блок-схемах операции присваивания обычно обозначают знаком «=» или «:=». В этом пособии принято обозначение «:=», т. е. как в языке *Pascal*. Именно такое обозначение может быть более предпочтительным для людей, ранее не изучавших программирования, поскольку позволяет, во-первых, четче видеть направление записи, а во-вторых, не путать

операцию присваивания с операцией отношения *равно* («=»), которая обычно записывается в составе логических выражений (наиболее часто в предикатных узлах).

В разных языках программирования присвоение записывается по-разному. Наиболее распространенными являются записи «=» и «:=». В некоторых ассемблерах операция присвоения вообще может иметь несколько различных модификаций, записанных в виде процессорных инструкций, например, ***mov***, ***add*** и т. д. Но, как правило, мнемоническое правило перемещения выражения справа в переменную слева всегда сохраняется.

ПРИМЕР

```
A := 2                    // A←2                (A=2)
C := A                    // C←A                (C=2)
Pokazatel := 6-3          //Pokazatel←6-3 (Pokazatel=3)
B := A^Pokazatel          // B←2^3              (B=8)
A:=A^2+B+2*C              // A←2^2+8+2*2        (A=16)
```

Приведенный выше пример показывает правила использования операции присваивания. Ниже приведем пример неправильного использования операции присваивания.

ПРИМЕР

```
2:=A //Ошибка!!! (нельзя изменять системную константу)
A+B:=6     // Ошибка!!! (слева записано выражение)
5-4:=3+B^2 // Ошибка!!! (слева константное выражение)
```
$\sqrt{A} := 1$ **// Ошибка!!! (слева записано выражение)**

2.4 Основы алгебры логики

Любая машина при решении алгоритмических задач выполняет некий вычислительный процесс, который называют машинной логикой. Только в отличие от логики человеческой, она чрезвычайно жесткая, поскольку подчиняется определенному набору правил. Эти правила возведены в ранг математических и носят соответствующее название:

математическая логика, или *машинная логика*. В основе алгебры логики находится так называемый *предикат*.

Предикат – это высказывание, относительно которого можно сказать истинно оно или ложно. Слово образовано от английского *Predicate* (*утверждение*). Примеры предикатов: «*Земля – третья планета от Солнца*», «*По календарю сейчас лето*» и т. д.

Часто логику предикатов называют *Булевой*[9] *алгеброй*, а выражения, принимающие всего два значения, - *Булевыми* (*Boolean*).

Для того чтобы научить ЭВМ «мыслить» логикой предикатов, нужно эти самые предикаты перевести на понятный машине язык. В случае языка программирования *Pascal* в качестве предиката могут быть *логические константы* и *логические выражения*.

Логическими выражениями будем называть выражения, состоящие из операций отношения и логических констант, связанных логическими операциями.

Операция отношения – операция сравнения результатов вычисления двух алгебраических выражений и/или числовых констант. Под операциями отношения понимают набор из шести операций сравнения: $<$, $>$, $=$, $\leq$, $\geq$, $\neq$. Результатом операции отношения всегда являются логические константы ***TRUE*** (*ИСТИНА*) или ***FALSE*** (*ЛОЖЬ*). Результат ***TRUE*** получается тогда, когда операция отношения записана верно, а ***FALSE*** - в противном случае.

Например, пусть $x = 7$, тогда операция отношения $x > 0$ даст результат ***TRUE***, т. е. истинно, что $7 > 0$. А при том же значении x операция $x < 5$ даст ответ ***FALSE***, что означает ложность утверждения $7 < 5$.

[9] Джордж Буль (1815–1864) английский математик и логик. Разработал алгебру логики и основы функционирования цифровых компьютеров.

Для простоты понимания, особенно студентам, ранее не изучавшим основы алгебры логики, нужно читать операции отношения с вопросительной интонацией и пытаться дать на заданный вопрос ответ *ДА* (***TRUE***) или *НЕТ* (***FALSE***). Т. е. приведенный выше пример необходимо прочесть так: «$x > 0$?», и в зависимости от значения x дать на него ответ ***TRUE*** или ***FALSE***.

Предикаты бывают *простыми* (содержащими единственное утверждение) и *сложными* (содержащими комбинацию утверждений). Комбинирование предикатов происходит по определенным правилам. Для предикативных конструкций, записанных на естественном языке, связками выступают союзы «*И*» и «*ИЛИ*». Вот примеры: «Земля – третья планета от Солнца ИЛИ Земля – четвертая планета от Солнца» – *ИСТИНА*; «Марс – первая планета от Солнца И Земля третья планета от Солнца» – *ЛОЖЬ*. Комбинация предикатов образует новый предикат. Кроме этого, по отношению к предикатам применяют модифицирующий предлог «НЕ», который отрицает предикат, переводя истинное утверждение в ложное, и наоборот. Например, «В одном километре 1000 метров» – *ИСТИНА*, а отрицание «В одном километре НЕ 1000 метров» – *ЛОЖЬ*.

Вообще, для составления сколь угодно сложных логических конструкций трех описанных выше операций достаточно. В языке же Pascal по умолчанию используют четыре логических операции: ***AND***, ***OR***, ***XOR*** и ***NOT***, что можно перевести как, соответственно, *И*, *ИЛИ*, *Исключающее ИЛИ* и *НЕ*. Логические операции делят на *бинарные* и *унарные*. *Бинарными* называют операции, для выполнения которых необходимо два операнда, *унарными* – один. Примерами бинарных математических операций являются операции умножения, деления, сложения и вычитания, т. е. умножить можно только одно число на другое; знак умножения теряет смысл, если стоит перед отдельным числом без

второго множителя. С другой стороны, из математики известно понятие *унарного минуса*, который превращает положительное число в отрицательное. Это типичный пример унарной операции, так как унарный минус записывается перед одним числом.

Таблица 2.2

Таблица истинности логических операций

A	**B**	**A *AND* B**	**A *OR* B**	**A *XOR* B**	***NOT* A**	***NOT* B**
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Операции ***AND*** – *логическое И*, называемое так же *логическим умножением* или *конъюнкцией* и обозначаемое «&», или «∧», ***OR*** – *логическое ИЛИ* (*логическое сложение* или *дизъюнкция*, обозначается «∨ »), ***XOR*** – *исключающее ИЛИ* (*сложение по модулю 2*, обозначается – «⊕») являются бинарными, т. е. для получения с их помощью результата необходимо иметь два операнда. Операция ***NOT*** – *логическое НЕ* (*логическое отрицание*, обозначается линией-надчеркиванием над константой или выражением, например, $\overline{P}$, или префиксным значком «¬») является унарной операцией. Для того чтобы пользоваться этими операциями, необходимо знать так называемые таблицы истинности. Таблицы истинности – таблицы, в которых описаны правила использования логических операций, т. е. результаты, получаемые при различных комбинациях операндов. Таблицы истинности являются своеобразными «таблицами умножения» для алгебры логики. При помощи этих элементарных правил составляются логические комбинации для выражений любой сложности. Для упрощения записи этих таблиц вводят обозначения: ***TRUE***≡1 (*логическая единица*), ***FALSE***≡0 (*логический нуль*).

Операнды обозначим ***A*** и ***B***. Обратим внимание на унарные операции ***NOT*** над операндами ***A*** и ***B*** (табл. 2.2).

Вообще, для двух операндов можно составить 16 различных таблиц. Однако все их обычно не записывают, поскольку существует возможность комбинации некоторого базового набора логических операций, которая приводит к эквивалентности таблиц истинности. Наиболее часто используются операции ***OR***, ***AND*** и ***NOT***. Операция ***XOR*** является дополнительной и выражается через базовый набор следующим образом:

$$A\ \textbf{XOR}\ B = \left(A\ \textbf{OR}\ B\right)\textbf{AND}(\textbf{NOT}(A\ \textbf{AND}\ B)).$$

Будучи записанными в сложном выражении, логические операции применяются в строгой последовательности, согласно установленному приоритету (подобно тому как операция умножения всегда выполняется раньше, чем операция сложения). Приоритет выполнения логических операций следующий (в порядке его убывания):

NOT

AND

OR, ***XOR***

Операции отношения

ПРИМЕР

Для $x = 5$, $y = 0$ получим результат следующего логического выражения:

$(x > 0)$ ***AND*** $(y < -2)$.

Результатом первой операции отношения $(x > 0)$ будет значение ***TRUE***, так как истина, что $5 > 0$. Результатом операции отношения ($y < -2$) будет значение ***FALSE***, так как ложь, что $0 < -2$. Осталось определить результат такого логического выражения:

TRUE AND FALSE, что эквивалентно 1 ***AND*** 0.

Из таблицы истинности следует, что это выражение равно 0 или ***FALSE***, т. е.

$(x > 0)$ ***AND*** $(y < -2)$ = ***FALSE***.

Для иллюстрации логических выражений часто применяют диаграммы, взятые из теории множеств (рис. 2.1). Заштрихованная площадь означает истинность того, что некоторая точка принадлежит этой области.

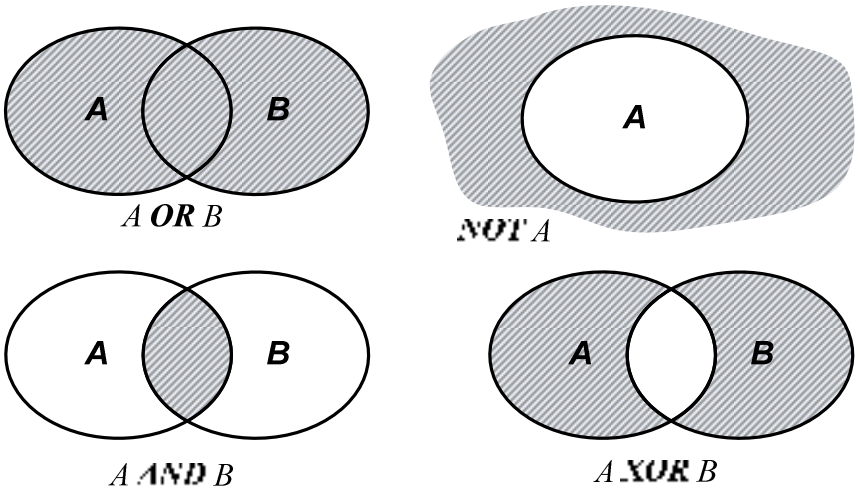


Рис. 2.1 Связь теории множеств и булевой алгебры

2.5 Правила использования логических выражений

При доказательстве в алгебре логики применяют набор правил и законов:

1) законы *идемпотентности*:

A = A ***AND*** A,

A = A ***OR*** A;

2) законы *коммутативности*:

A ***AND*** B = B ***AND*** A,

A ***OR*** B = B ***OR*** A;

3) законы *ассоциативности*:

A ***AND*** (B ***AND*** C) = (A ***AND*** B) ***AND*** C,

A ***OR*** (B ***OR*** C) = (A ***OR*** B) ***OR*** C;

4) законы *дистрибутивности*:

A ***AND*** (B ***OR*** C) = (A ***AND*** B) ***OR*** (A ***AND*** C),

A ***OR*** (B ***AND*** C) = (A ***OR*** B) ***AND*** (A ***OR*** C);

5) законы *нуля* и *единицы*:

A ***AND*** $\overline{\text{A}}$ = ***FALSE***, A ***AND TRUE*** = A,

A ***OR*** $\overline{\text{A}}$ = ***TRUE***, A ***OR FALSE*** = A;

6) правила *поглощения*:

A ***OR*** (A ***AND*** B) = A,

A ***AND*** (A ***OR*** B) = A;

7) правила *де Моргана*:

$\overline{(\text{A } \textbf{\textit{OR}} \text{ B})} = (\overline{\text{A}} \textbf{\textit{ AND }} \overline{\text{B}})$,

$\overline{(\text{A } \textbf{\textit{AND}} \text{ B})} = (\overline{\text{A}} \textbf{\textit{ OR }} \overline{\text{B}})$;

8) правила *склеивания*:

$(\text{A } \textbf{\textit{OR}} \overline{\text{B}}) \textbf{\textit{ AND }} (\text{A } \textbf{\textit{OR}} \text{ B}) = \text{A}$,

$(\text{A } \textbf{\textit{AND}} \overline{\text{B}}) \textbf{\textit{ OR }} (\text{A } \textbf{\textit{AND}} \text{ B}) = \text{A}$.

2.6 Базовые алгоритмические конструкции

Любой сколь угодно сложный алгоритм можно представить в виде комбинации трех базовых алгоритмических управляющих структур: *следование*, *развилка* и *цикл*. Данный тезис был высказан Дейкстрой[10] в

[10] Эдгер Вайб Дейкстра (1930−2002) – голландский математик, основоположник метода структурного программирования, занимался вопросами применения математической логики к компьютерным программам.

конце 70-х годов XX века и впоследствии только подтверждался. Этот подход носит название *структурного программирования*, одним из его основных достоинств является отказ от *оператора безусловного перехода* (***GOTO***), что многократно повышает наглядность и надежность программного кода. Каждой из трех управляющих конструкций соответствует свой тип вычислительного процесса, соответственно: *линейный*, *разветвляющийся* и *циклический*. Рассмотрим последовательно каждый из них.

2.6.1 Линейные вычислительные процессы

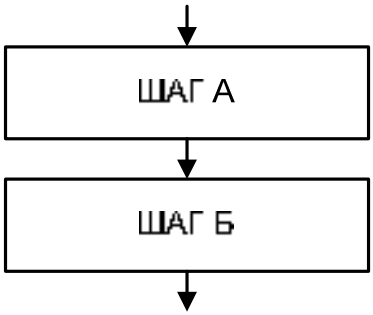


Рис. 2.2. Управляющая структура «следование»

Линейным вычислительным процессам соответствует алгоритмическая управляющая структура *следование*. В данном случае вычислительные шаги следуют один за другим без пропусков и возвратов, т. е. отсутствуют различного рода ветвления алгоритма – *развилки* и *циклы*. Блок-схема данной алгоритмической управляющей структуры имеет вид, показанный на рис. 2.2.

Для простоты обозначения шаги заключены в прямоугольники, но на месте прямоугольников могут быть блоки *обработки данных* (собственно прямоугольники), блоки *ввода/вывода* (параллелограммы), блоки вызова *подпрограмм* или иные управляющие конструкции (*цикл* или *развилка*).

2.6.2 Разветвляющиеся вычислительные процессы

Разветвляющимся вычислительным процессам соответствует алгоритмическая управляющая структура *развилка*. Во многих алгоритмах реализованы структуры с двумя альтернативными ветвями, одна из

которых выполняется в том случае, если проверка некоторого логического выражения (в частном случае условия, например, $x > 0$) L дает положительный результат (***TRUE*** – «*ИСТИНА*»), а другая – отрицательный (***FALSE*** – «*ЛОЖЬ*»). Эта структура и называется развилкой.

На блок-схемах развилки обозначают в виде ромба (*предикатного узла*) с одним входом и парой выходов (рис. 2.3). Причем выходы направляют влево и вправо, а не вниз (чтобы иметь возможность отличать развилки от циклов).

Возможна структура развилки с одной ветвью. Т. е., если проверка логического выражения L дает положительный результат, то выполняется одна ветвь развилки, иначе эта ветвь игнорируется (рис. 2.4).

Здесь показана развилка с одной ветвью – ветвью ***TRUE***; развилка же только с ветвью ***FALSE***, хотя и существует, но ею редко пользуются. Дело в том, что для ее реализации достаточно инвертировать логическое выражение L (применить операцию отрицания ***NOT***) и мы придем к обычной структуре с непустой положительной ветвью (рис. 2.5).

Часто вместо слов ***TRUE*** и ***FALSE*** на блок-схемах пишут «***T***» и «***F***», или «*Да*» и «*Нет*», или «+» и «–», или «*1*» и «*0*» и т. д.

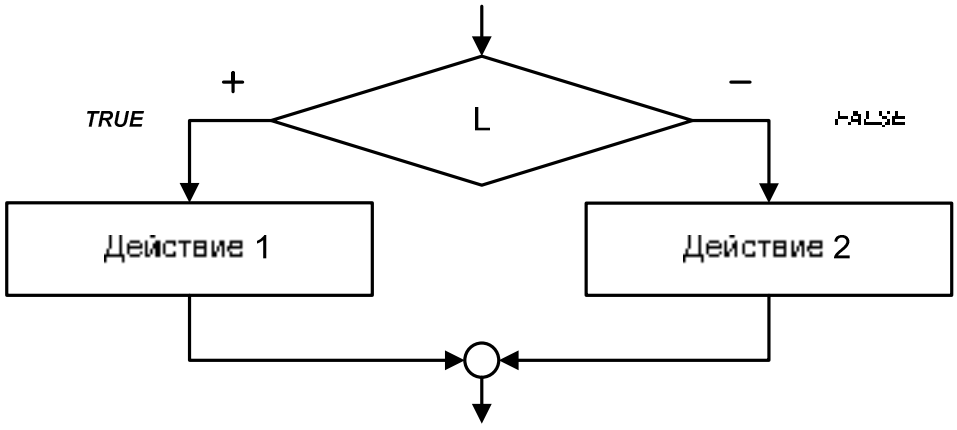


Рис. 2.3. Полная развилка

ПРИМЕР

Рассмотрим более сложный пример. Построим блок-схему алгоритма с использованием развилок (рис. 2.6). Пусть это будет блок-схема алгоритма решения квадратного уравнения $ax^2 + bx + c = 0$, описанного выше, в разделе «*2.2* Алгоритмизация» с помощью естественного языка. Номера шагов расставим согласно словесному описанию, приведенному ранее. Если сравнить два способа записи одного и того же алгоритма, можно заметить, что на блок-схеме отсутствуют шаги с номерами *7*, *11*, *14* и *16*. Это связано с тем, что действие «*выполнить шаг 18*» во всех этих

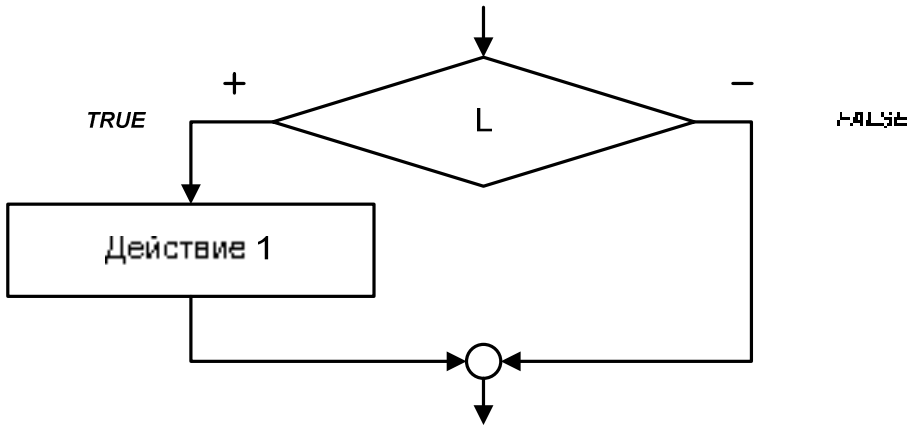


Рис. 2.4. Неполная развилка

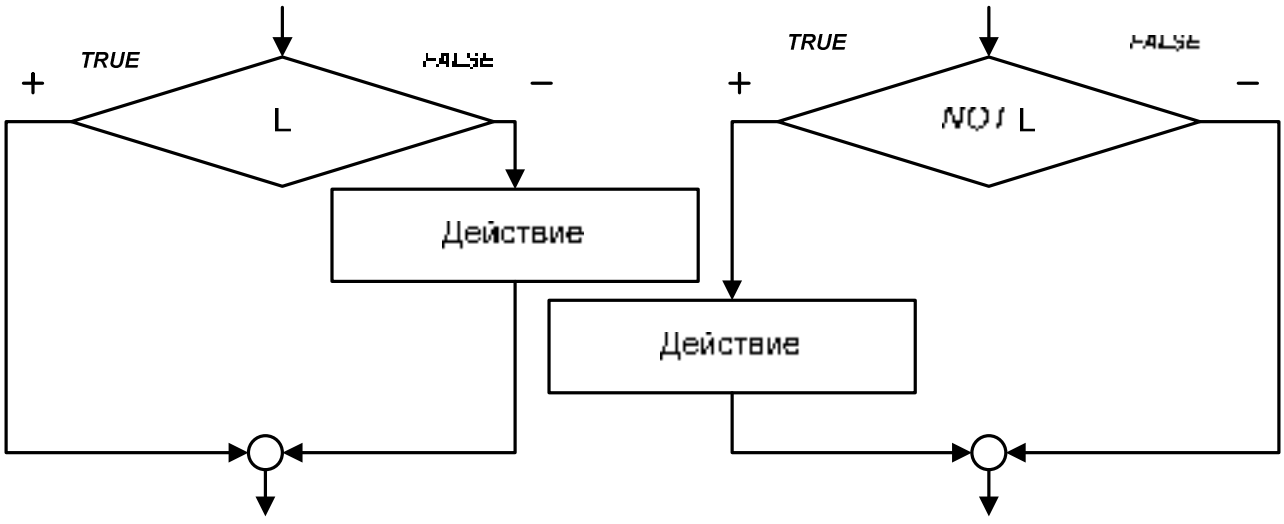


Рис. 2.5. Преобразование развилки с пустой положительной ветвью в развилку с непустой положительной ветвью

пунктах описано стрелками, направленными к *шагу 18*, соответственно, от *6*-го, *10*-го, *13*-го и *15*-го шагов.

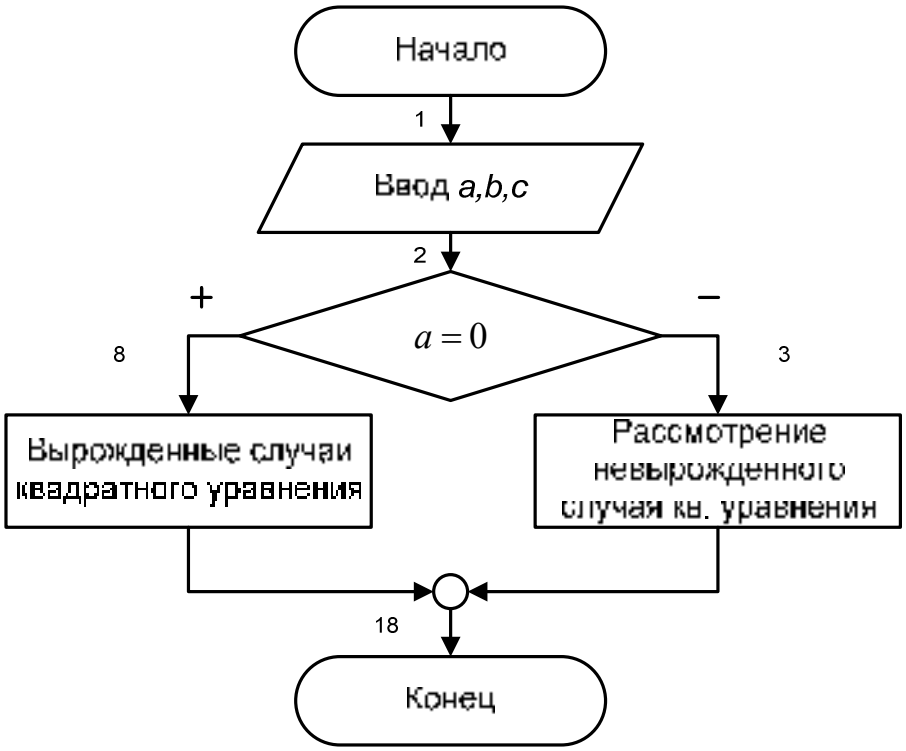


Рис. 2.6. Общая блок-схема решения квадратного уравнения

Еще одной особенностью изображения блок-схемы (рис. 2.6) является ее дробление на более мелкие части, поскольку она достаточно сложная и может не помещаться целиком на странице. Кроме того, это дает наглядное представление о характере решаемой задачи. А именно, сначала рассматриваем проблему в общем (говорим что нужно ввести коэффициенты уравнения a,b,c и далее, в зависимости от введенных данных, решаем один или другой вид уравнения (рис. 2.7 и рис. 2.8). После этого каждый из шагов детализируем до уровня, достаточного для понимания устройством, с помощью которого данная задача будет решаться (ЭВМ). Такой подход отлично соотносится с концепцией *структурного программирования* и носит название проектирования *сверху вниз*. Т. е. от общего мы постепенно переходим к частностям. Существует и иной подход, называемей проектирование снизу вверх (когда решаются

сначала частные случаи, а далее все они объединяются в общий алгоритм решения задачи).

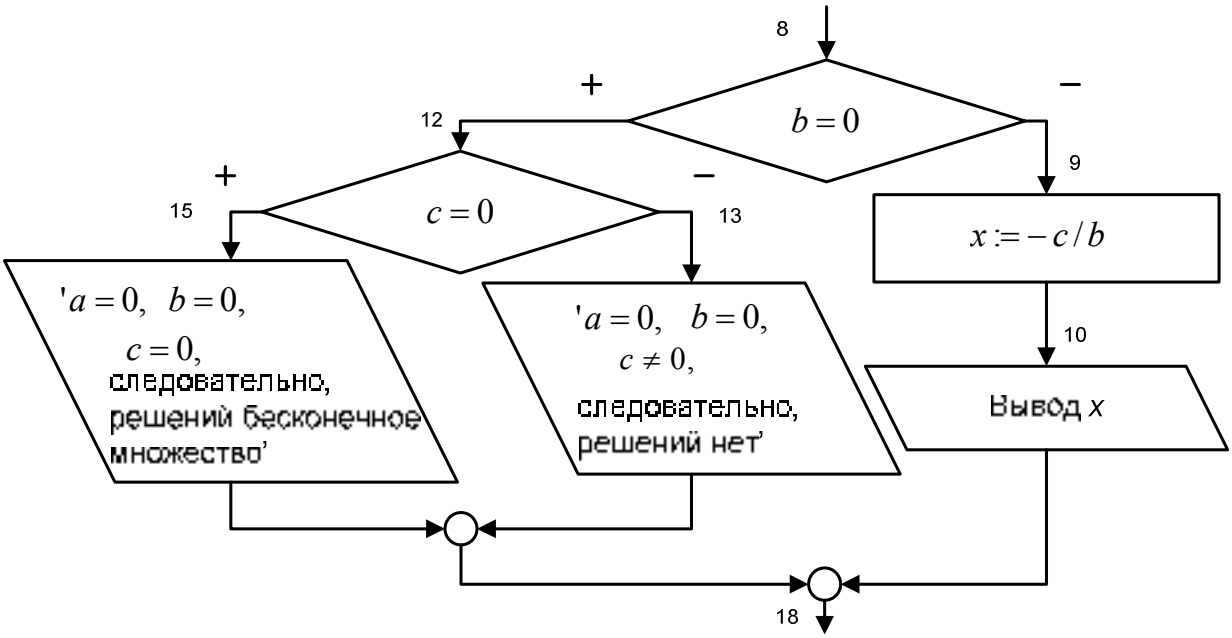


Рис. 2.7. Шаг 2-18: алгоритм решения вырожденного случая уравнения

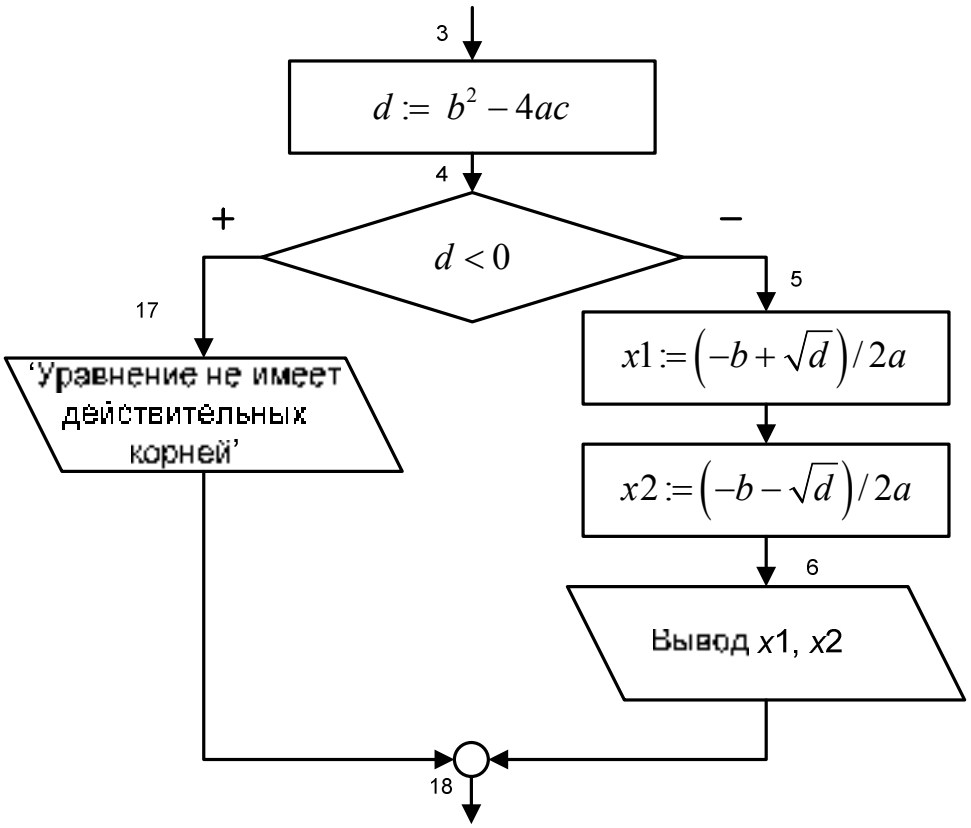


Рис. 2.8. Шаг 3-18: алгоритм решения невырожденного уравнения

2.6.3 Циклические вычислительные процессы

Циклические вычислительные процессы – это многократно повторяющиеся последовательности действий. Таким процессам соответствуют алгоритмические управляющие структуры – *циклы*. Собственно последовательность действий, которую необходимо многократно повторить, называется ***телом цикла***. Циклы могут быть вложенными. Цикл, находящийся в теле другого цикла, называется *внутренним*, а охватывающий его – *внешним*.

Вообще, в основе любого цикла лежит *итеративность*, т. е. многократное повторение одних и тех же действий. ***Итерация*** – однократное исполнение алгоритма тела циклического процесса. *Iteratio* – *повторение* (*лат.*).

Рассмотрим три основных вида циклов: «с предусловием», «с постусловием», «цикл-счетчик».

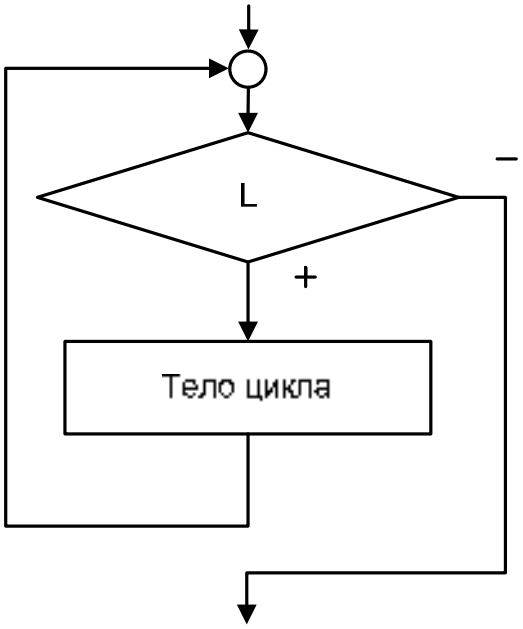


Рис. 2.9. Цикл с предусловием

Цикл с предусловием или цикл «***Пока***» (цикл выполняется, пока логическое выражение *L* дает результат ***TRUE***). Блок-схема изображена на рис. 2.9. В *Pascal* этот цикл называется «***while*** … ***do***». Как следует из названия цикла, его тело будет выполняться, пока верно логическое выражение (*L*=***TRUE***)[11]. Как только результат

[11] Очень часто в предикатных узлах для проверки логической переменной ее сравнивают с одной из двух возможных логических констант (***TRUE*** или ***FALSE***), что не является алгоритмической ошибкой. Однако не стоит забывать, что сама по себе логическая переменная уже равна ***TRUE*** или ***FALSE***, а потому нет необходимости в избыточной проверке. Описанную ситуацию можно отнести к стилистическим ошибкам, которые могут быть свойственны новичкам в программировании.

логического выражения L изменится на ***FALSE***, исполнение тела завершается и управление передается структуре, следующей далее по стрелке.

ПРИМЕР

Разработаем алгоритм (блок-схему) вычисления факториала некоторого натурального числа N.

$F(N) = N!$

Напомним, что

$N! = 1 \cdot 2 \cdot ... \cdot (N-1) \cdot N$.

Можно заметить, что для вычисления факториала справедлива *рекуррентная* формула $F(K) = F(K-1) \cdot K$.

Вообще, понятие *рекуррентности* и *рекурсии* в программировании играют большую роль. Следует напомнить, что *рекуррентной* называется последовательность, для которой текущий ее член определяется через предыдущий. Соответственно, функция называется *рекурсивной,* если она может вызывать сама себя. Всегда в рекуррентном соотношении должна существовать точка входа. Для факториала таковой является определение $0! = 1$.

Для вычисления факториала будем использовать рекуррентное выражение $F = F \cdot I$. В этой формуле вместо I последовательно подставляются натуральные числа от *1* до *N*. При первом ее использовании F возьмем равным 1 (точка входа), так как число *I,* умноженное на 1, останется равным *I*, т. е. после первого шага переменная *F* станет $F = I = 1$. Дальнейшее умножение на 2, 3 (и т. д. до *N)* даст факториал числа $F(N) = N!$.

Рассмотрим, как работает данный алгоритм (рис. 2.10). Пусть при вводе N будет введено значение N=4, тогда перед входом в цикл переменные будут иметь следующие значения: N=4; I=1; F=1.

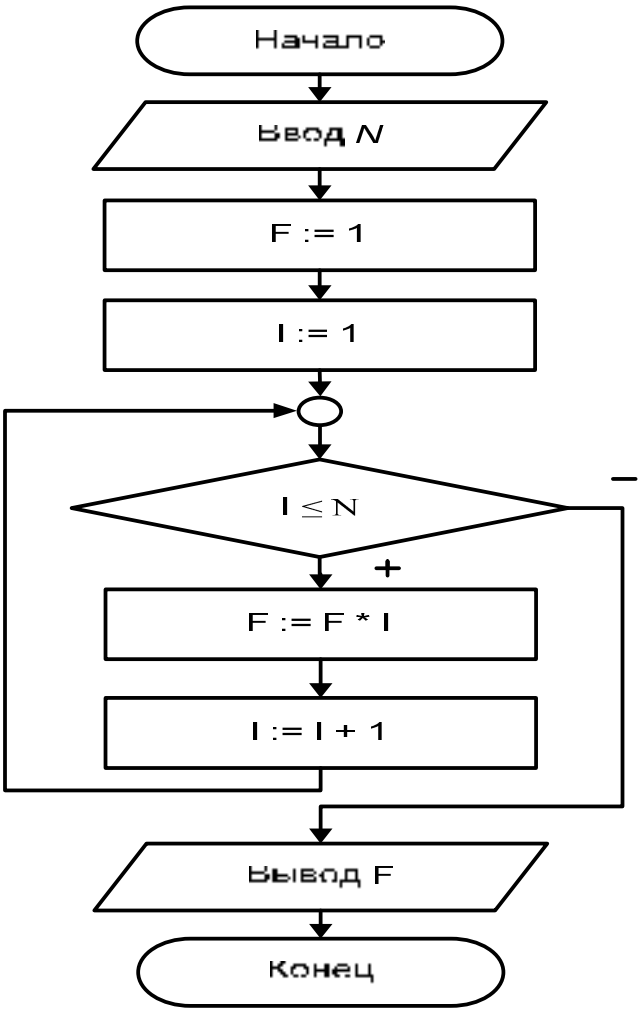


Рис. 2.10. Алгоритм расчета факториала (цикл с предусловием)

При входе в цикл анализируется логическое выражение «$I \le N$». Для текущих значений I=1 и N=4 оно даст результат ***TRUE***, т. е. будет выполняться тело цикла. В нем на первом шаге новое значение F получается путем умножения его старого значения на I (именно так нужно читать формулу $F := F * I$), т.е. $F = 1*1 = 1$. Вторая формула увеличивает I на 1, т. е. $I = 1+1 = 2$. В результате выполнения 1-го прохода тела цикла переменные примут следующие значения: N=4, I=2, F=1. Следуя далее по стрелке, возвращаемся к предикатному блоку с логическим выражением и анализируем его для новых значений переменных. Результат опять ***TRUE***. После выполнения 2-го прохода переменные примут такие значения: N=4, I=3, F=2. Логическое выражение снова даст результат ***TRUE***. В третьем проходе получим: N=4, I=4, F=6. Логическое выражение равно ***TRUE***. В четвертом проходе N=4, I=5, F=24.

И только после этого прохода логическое выражение становится равным ***FALSE***, т. е. выполнение цикла завершается и происходит переход на шаг вывода значения *F*. Результат *F*=24 легко проверить в уме: $F(4) = 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$, следовательно, алгоритм верен.

Следующий вид цикла, носит название «***цикл с постусловием***» или цикл «***До***» (цикл выполняется до получения в качестве результата логического выражения *L* значения ***TRUE***). В *Pascal* этот цикл записывается «***repeat*** … ***until***». На блок-схеме изображается так, как показано на рис. 2.11. В отличие от цикла с предусловием, его тело повторяется пока *L*=***FALSE***, как только *L* становится равным ***TRUE***, выполнение тела завершается.

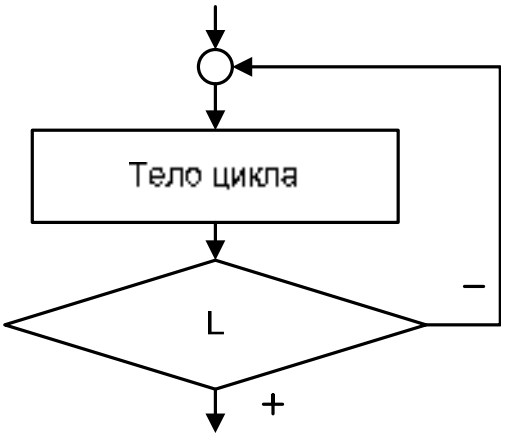


Рис. 2.11. Цикл с постусловием

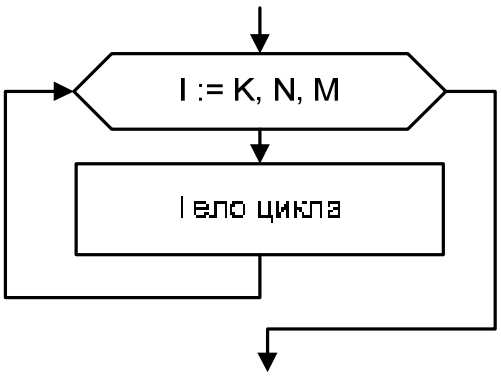


Рис. 2.12. Цикл с параметром

Важное замечание. Можно подобрать такие начальные значения операндов логического выражения, что тело цикла с предусловием ни разу не будет выполнено, но невозможно подобрать подобные значения для цикла с постусловием (его тело выполняется всегда хотя бы один раз). Это замечание может быть доказано, если проанализировать блок-схемы рассмотренных циклов.

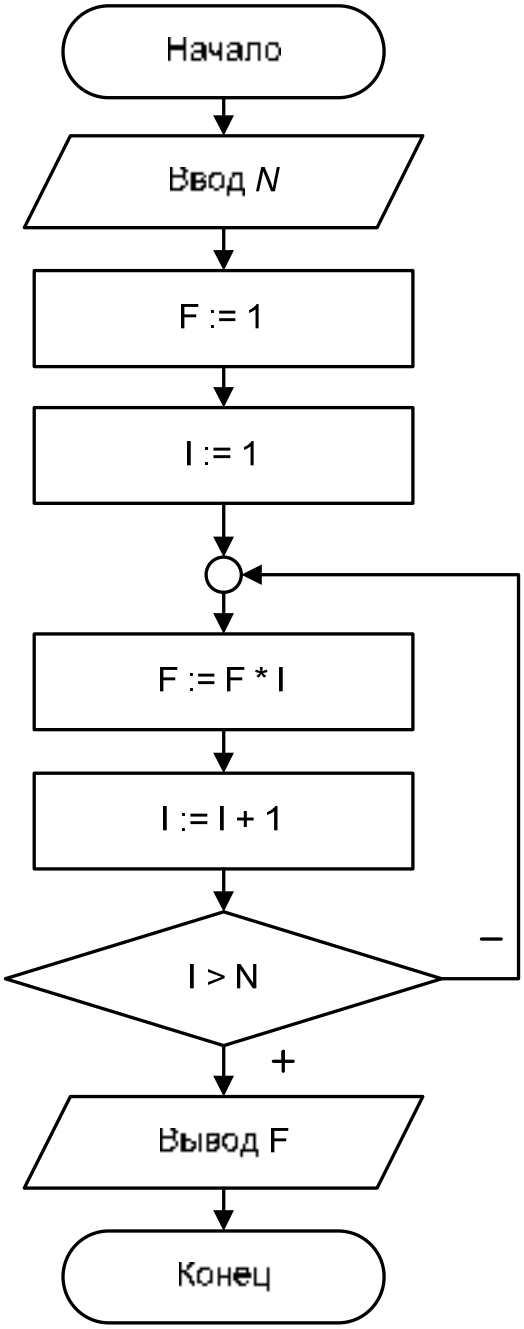


Рис. 2.13. Алгоритм расчета факториала
(цикл с постусловием)

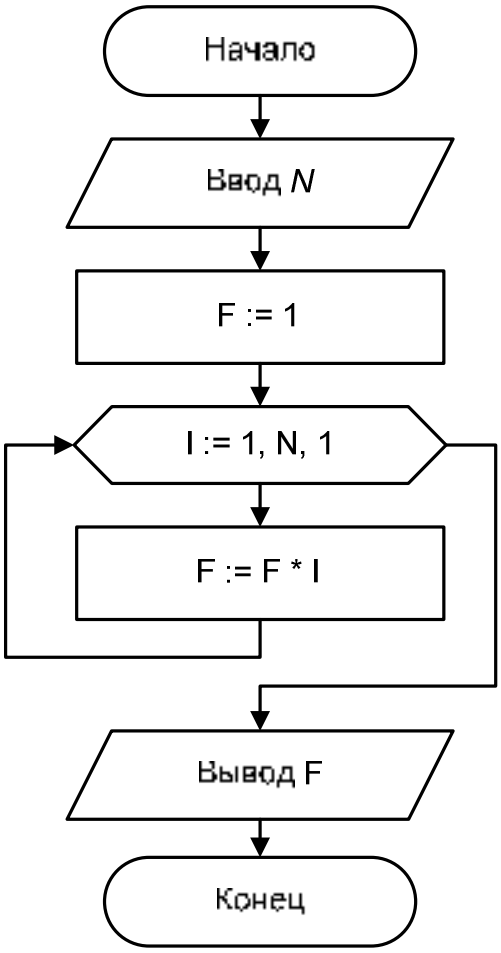


Рис. 2.14. Вычисление факториала
(цикл с параметром)

У цикла с предусловием существует стрелка обхода тела, а у цикла с постусловием такого обхода нет (через цикл можно пройти лишь только

сквозь его тело). На рис. 2.13 представлена блок-схема алгоритма вычисления факториала числа *N* с помощью цикла с постусловием.

Третий цикл – ***цикл с параметром***, или, как его еще называют «***цикл For***». Этот цикл работает следующим образом. Параметру *I* присваивается его начальное значение *K;* далее выполняется тело цикла, в котором этот параметр может быть использован; затем к текущему значению параметра *I* прибавляется шаг *M* и вновь выполняется тело цикла. Так продолжается до тех пор, пока условие $I \le N$ истинно (рис. 2.12).

Вообще, цикл *For* является модификацией цикла с предусловием. Просто настолько часто возникают задачи, в которых параметр цикла меняет свое значение с постоянным шагом на некотором диапазоне, что в языки программирования внедрили этот специальный цикл. Более того, именно его наиболее часто используют в большинстве программ. В языке *Pascal* цикл *For* работает только с целыми типами параметров, а шаг цикла *M* равен 1, либо –1. В первом случае его называют «***for*** … ***to*** … ***do***», а во втором «***for*** … ***downto*** … ***do***».

Задача вычисления факториала с помощью этого цикла выглядит гораздо проще (рис. 2.14).

2.6.4. Замечания по оформлению блок-схем

Как уже отмечалось ранее, блок-схемы представляют собой одну из форм записи последовательности действий. Поскольку решено использовать ограниченный набор базовых блоков, то, естественно, с их помощью можно изобразить не всякий алгоритм. Кроме того, базовых алгоритмических конструкций в рамках структурного программирования всего три. Каждая из этих конструкций имеет ровно один вход и ровно один выход. Потому, для сохранения наглядности, всегда изображают выход из конструкции строго под ее входом, на одной оси (рис. 2.15).

Следующее правило уже озвучивалось ранее и предназначено для различения конструкций развилок и конструкций циклов. Обе они изображаются при помощи блока предиката. Но в циклах принято вход в его тело (или выход из него) обозначать стрелкой, выходящей из крайней нижней точки ромба вниз, а в разветвляющихся алгоритмических

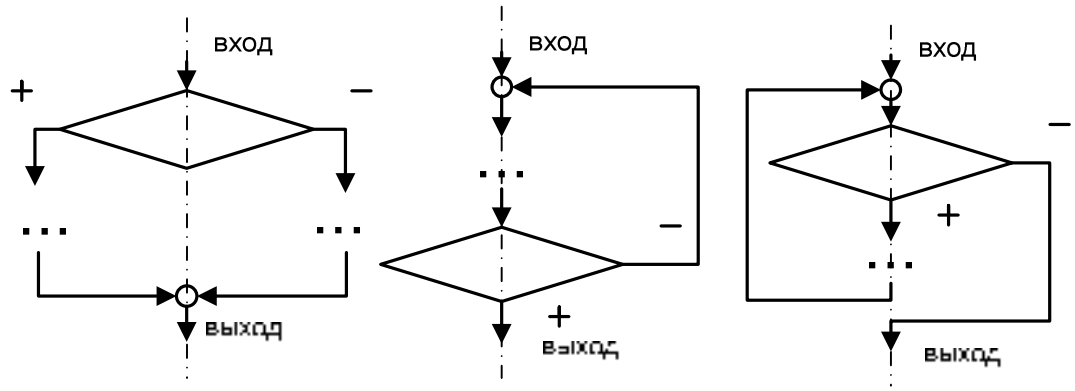


Рис. 2.15. Симметрия в обозначении базовых алгоритмических конструкций

конструкциях плечи рисуются горизонтально. Причем для развилок положительную ветвь можно рисовать как влево, так и вправо, а для циклов языка *Pascal* положительное направление всегда направляют вниз.

Особенность структурного программирования – наличие внутри каждой элементарной конструкции блока действия. Так вот, вместо этого блока может быть любая другая базовая конструкция; внутри этой базовой конструкции может опять стоять любая базовая конструкция и т. д. В итоге иерархия вложенных структур может быть весьма большой. Вследствие этого, блок-схема не помещается на одном листе и ее наглядность страдает. Для корректного дробления алгоритмов часть вложенных действий заменяют одним блоком (прямоугольником, внутри которого пишут описание производимого действия) с нумерацией потока входа и потока выхода. Далее, в удобном месте пишут название этого блока и шаги, которые он заменяет. Пример пошаговой детализации был приведен на рис. 2.16

Еще одним признаком корректности изображения блок-схемы является факт непересечения поточных линий. Если при составлении блок-схемы линии пересекаются, то нужно попытаться представить ее так, чтобы эти пересечения ушли. Если этого сделать нельзя, то блок-схема составлена неправильно (вне рамок структурного программирования).

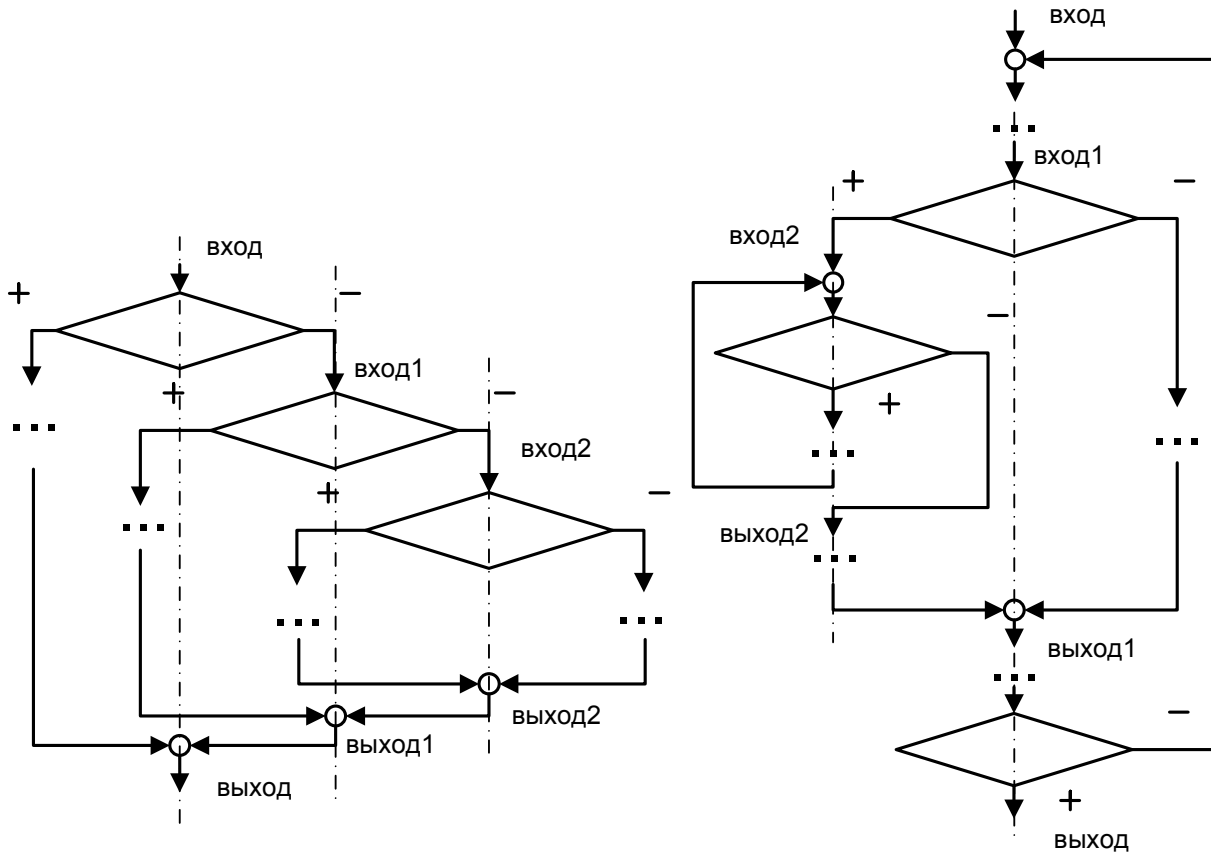


Рис. 2.16. Вложенные алгоритмические конструкции

Кроме того, на блок-схеме не должно быть блоков (кроме терминаторов), в которые есть вход и нет выхода, так называемых «висячих» блоков.

Вход в алгоритмическую структуру происходит сверху, а выход - снизу. То есть всегда строго сверху вниз, а не с боков и, тем более, не снизу наверх.

Использование базовых алгоритмических конструкций показано на рис. 2.16, на нем можно различить все их типы, а также входы и выходы.

3. ОСНОВНЫЕ СВЕДЕНИЯ О ЯЗЫКЕ PASCAL

3.1 Алфавит языка. Идентификаторы

Как и разговорный язык, язык программирования обладает своим алфавитом. Из символов алфавита выстраиваются слова, а из них, в свою очередь, - предложения. В *Pascal* из алфавита формируются *зарезервированные слова*, *идентификаторы* (имена процедур, функций, переменных, констант и т. д.), *выражения* (алгебраические, логические и т. д.), значения *констант* и *переменных*, описание *типов* и т. п.

Алфавит языка состоит из следующих наборов символов:

- по 26 прописных ***A***..***Z*** и 26 строчных ***a***..***z*** букв латиницы;
- арабские цифры ***0***..***9***;
- знаки арифметических операций: + – * /;
- знаки операций отношения: > < =;
- скобки: ***( ) [ ] { }***;
- разделители: ***. , : ;***;
- апостроф: ‘;
- специальные символы: ***@, #, $, ^, &, _***.

В языке *Pascal* прописные и строчные буквы латиницы не различаются. Кроме этого, здесь не отмечены иные символы таблицы *ASCII*, которые могут быть использованы в комментариях, в строковых или символьных константах. Это, прежде всего, символы кириллицы.

Язык *Pascal* обладает целым рядом служебных (зарезервированных) слов, которые нельзя использовать в качестве пользовательских (придуманных пользователем) имен типов, переменных и других идентификаторов. В этот список входят следующие слова:

absolute end inline procedure type

and	***external***	***interface***	***program***	***unit***
array	***file***	***interrupt***	***record***	***until***
begin	***for***	***label***	***repeat***	***uses***
case	***forward***	***mod***	***set***	***var***
const	***function***	***nil***	***shl***	***while***
div	***goto***	***not***	***shr***	***with***
do	***if***	***of***	***string***	***xor***
downto	***implementation***	***or***	***then***	
else	***in***	***packed***	***to***	

Идентификаторы – имена переменных констант, процедур, функций и т. д. Идентификаторы могут состоять только из букв латиницы (прописные и строчные – неотличимы), цифр и символа «_». Идентификатор не может начинаться с цифры; может иметь длину, равную максимальной длине строки – 127 символов, но лишь 63 из них будут значащими. Это означает, что если представить себе такую ситуацию, когда понадобится придумывать имена длиной более 63 символов, то два имени, состоящих, например, из совпадающих первых 63 и не совпадающих остальных букв, будут восприняты компилятором как одно и то же. Приведем примеры идентификаторов.

ПРИМЕР

Правильные:

```
A; B; Next; ffFRt; _EE;
IvanovIvan; d2w; x3; Ivanov_Ivan;
```

Неправильные:

```
1x {начинается с цифры};
Ivanov Ivan {пробел в имени};
Sob@ka {использование спецсимвола};
While {зарезервированное слово};
Q2-R {использование запрещенного символа}.
```

3.2 Структура программы на языке Pascal

Программа на языке *Pascal* может состоять из восьми основных разделов.

1. Раздел *заголовка программы*

Program *имя****;***

2. Раздел *подключаемых библиотек*

Uses *список библиотек****;***

3. Раздел *описания меток*

Label *список меток****;***

4. Раздел *описания констант*

Const *описание констант****;***

5. Раздел описания *пользовательских типов*

Type *описание типов****;***

6. Раздел *описания переменных*

Var *описание переменных****;***

7. Раздел *описания процедур и функций*

Описание процедур начинается со слова ***procedure***

Описание функций начинается со слова ***function***

8. Раздел *основной программы*

Начинается ***begin***

…

заканчивается ***end***.

Следует обратить внимание на то, что в конце программы необходимо ставить точку. Каждый из разделов является необязательным и используется по мере необходимости. Далее все их рассмотрим подробнее.

Первым по порядку идет ***заголовок программы***. Этот раздел, как и другие, не является обязательным, но его использование четко выделяет

начало программного кода. Имя, используемое в заголовке программы, составляется, как и любой идентификатор, из латинских букв, арабских цифр и символа «_». Это имя не может быть использовано при описании другого идентификатора, и, вообще, нельзя использовать это имя внутри программы для других целей. Имя программы стоит вводить еще и потому, что, например, оно служит для обращения к глобальным идентификаторам в случае совпадения их имен с именами локальных идентификаторов во внутренних модулях программы (процедурах и функциях).

В разделе ***подключаемых библиотек***, или другими словами *модулей,* перечисляются библиотеки, которые будут подключены к программе во время компиляции. *Модулем* или *библиотекой* можно назвать совокупность подпрограмм, объединенных произвольным образом в отдельный файл. Модули существуют как стандартные (поставляемые вместе с конкретной *IDE*), так и пользовательские, которые пользователь может написать самостоятельно или где-нибудь взять. Модули в *Pascal* носят название *TPU* (*Turbo Pascal Unit*).

К стандартным модулям относится, например, библиотека *CRT*. Для ее подключения во втором разделе следует написать фразу ***Uses CRT***. *CRT* содержит процедуры и функции работы со стандартными устройствами ввода/вывода (например, с экраном в текстовом режиме). Допустим, необходимо экран очистить. Для этого потребуется процедура ***clrscr***, содержащаяся в библиотеке *CRT*. Если использовать ее без ***Uses CRT***, то откомпилировать программу будет невозможно, так как для компилятора ***clrscr*** будет всего лишь набором букв.

В *IDE Turbo Pascal 7.0* существуют следующие стандартные модули: *CRT*, *Graph*, *Graph3*, *Overlay*, *Printer*, *Strings*, *System*, *Turbo3*, *WinAPI*, *WinCrt*, *WinDOS*, *WinPrn*, *WinProcs*, *WinTypes*. В рамках этого пособия

большинство модулей не будут рассмотрены подробно, но всегда можно найти информацию о них в разделе помощи по *Turbo Pascal*, нажав F1.

В разделе ***описания меток*** перечисляются метки, использующиеся в программе. Они перечисляются через запятую сразу за словом ***Label***. Например, так: «***Label m1,m2,m3,m4;***». Метки в программе используются для адресации строк, на которые возможен переход с помощью оператора безусловного перехода ***goto***. Пример использования меток:

```
program ex_label;
label m1,m2,m3;
var x:char;
begin
readLn(x);
  if x='1' then
    goto m1
  else
    goto m2;
  m1:writeLn('m1');
  goto m3;
  m2:writeLn('m2');
  m3:
end.
```

Технически возможность использовать метки в *Pascal* существует, но как было заявлено выше, любой алгоритм можно описать с помощью следования, развилки и цикла. Безусловный переход не относится ни к одной из этих алгоритмических управляющих структур. Из этого можно сделать простой вывод: в связи с тем, что безусловные переходы могут затруднить чтение программы, а так же они не относятся ни к следованиям, ни к развилкам, ни к циклам, их использование желательно избегать.

Следующий рассматриваемый раздел – ***описание констант***. Отметим, что константы в *Pascal* делят на два вида: *типизированные* и *нетипизированные*. В описании типизированных констант используются типы данных, поэтому отложим рассмотрение этого вопроса до изучения стандартных типов языка *Pascal*. Описать нетипизированные константы

очень просто. После имени константы ставится знак «=»[12], после чего - присваиваемое ей значение. При этом текстовые константы заключаются в апострофы. При описании константы можно не просто указать значение, но также записать математическое выражение. Операндами могут выступать как константы (числовые и других простых типов значения), так и имена выше описанных констант. В этих математических выражениях могут быть использованы стандартные математические операции (+, –, *, /, mod, div), а так же функции модуля *SYSTEM.TPU* (эту библиотеку нельзя подключить с помощью ***USES***; она, являясь библиотекой исполняющей системы, как бы постоянно подключена к любой программе): ***abs***, ***chr***, ***hi***, ***length***, ***lo***, ***odd***, ***ord***, ***pred***, ***prt***, ***sizeof***, ***succ***, ***swap***, ***trunc***.

ПРИМЕР

```
program ex_const;
const
  X=10;
  Y=20;
  Z=X+Y+30;
```

В разделе ***описания пользовательских типов***, если это требуется, можно, используя стандартные типы, описать новый тип данных[13].

ПРИМЕР

```
program ex_type;
type
  TInt = integer;
  TMassiv = array[1..20,1..10] of real;
  TFl = file of char;
var
  I, J, K : TInt;
  A, B : TMassiv;
  F: TFl;
```

[12] Здесь используется для операции присваивания знак «=», хотя ранее было сказано, что в Pascal присваивание обозначается «:=». Это особенность раздела описаний языка, поскольку разделы описания не содержат алгоритмических последовательностей в явном виде. Все присваивания происходят еще до начала работы программы, на этапе ее компиляции. А потому присваивание «=» можно читать как «тождественно равно», или «равно по определению».

[13] Вообще, названием пользовательского типа может быть любой идентификатор, однако часто этому названию добавляют литеру «T» для того, чтобы подчеркнуть, что это ничто иное, как тип данных.

В следующем разделе ***описываются переменные***. Это раздел ***var*** (*англ. Variable – переменная*). Переменной можно назвать поименованную область памяти, содержащую информацию заданного типа. *Pascal* является языком со строгим контролем типов данных. Во время работы программы содержимое переменной может меняться.

Для того чтобы программа могла хранить некоторые значения (начальные, промежуточные, конечные) в памяти, необходимо, чтобы для этих значений заранее было выделено место. Выделение места в памяти ЭВМ происходит автоматически на этапе компиляции, и пользователь не должен об этом заботиться. Он должен лишь указать неповторяющиеся имена описываемых переменных и их тип.

Указание типа связано с тем, что, как указывалось выше, вся информация в ЭВМ хранится в двоичном виде. Чтобы отделить, например, текст от чисел необходимо к каждой переменной привязывать способ перекодирования, т. е. определять ее тип в разделе описания переменных.

Собственно описание переменных происходит следующим образом. В разделе описания переменных, после слова ***var***, перечисляются через запятую однотипные переменные, затем ставится двоеточие и указывается их тип. После точки с запятой, обычно со следующей строки, перечисляются и описываются переменные другого типа и так далее, пока не будут описаны все требуемые в программе переменные.

В следующем разделе описываются ***подпрограммы***. В этом разделе не будем останавливаться на этой теме, так как она подробно раскрывается позже.

Последний раздел – ***основная программа***. Именно здесь помещается основной алгоритм работы программы. Она должна быть заключена в так называемые операторные скобки (слова ***begin*** и ***end***) и в конце обязательно стоит точка.

Можно выделить еще один условный вид блоков программ, неописанный выше, который называется *комментариями*. Комментарии играют очень большую роль при написании программы, поскольку помогают справиться с ее возрастающей сложностью. По прошествии некоторого времени, при повторном обращении к исходному коду для внесения в него исправлений или модификации, возникает ситуация забывания подробностей. Если комментарии расставлены грамотно, то вспомнить подробности работы алгоритма не составит труда. Кроме того, комментарии помогают разбираться в ваших алгоритмах тому, кто с ними будет работать в дальнейшем.

Сами по себе разделы комментариев пропускаются компилятором и на работу алгоритма никакого влияния не оказывают. В языке *Pascal* блок комментариев выделяется фигурными скобками «*{*» – начало комментария и «*}*» – конец комментария. Или, альтернативным способом: «*(**» – начало комментария и «**)*» – его окончание. Наиболее же часто используют так называемые однострочные комментарии. Начало такого комментария обозначается комбинацией «*//*», а конец – переходом на новую строчку. К сожалению, старые *IDE* языка *Pascal* этот тип комментариев не поддерживают (например, *IDE Turbo Pascal)*.

3.3 Типы данных в Pascal

Pascal является языком со строгим контролем типов, что означает четкое определение размера места занимаемого переменной в памяти, правила работы с этой переменной и порядок преобразования ее значения к другому типу (если оно вообще возможно).

Рассмотрим подробнее, какие типы данных существуют в языке Pascal. Начнем с *целочисленных*. Целое число проще всего привести к двоичному виду. Ранее уже рассматривалось, как это делается. С помощью

приведенного алгоритма можно перекодировать только натуральные числа, т. е. числа без учета знака. Но часто необходимо запоминать не только абсолютное значение числа, но и его знак. Для этого выделяют первый бит числа. Для целых чисел в *Pascal* выделяют от одного до четырех байт. Если число хранится без учета знака (типы ***byte*** и ***word***), то все разряды двоичного числа отводятся для его абсолютной величины. Если же необходимо запоминать знак, то в первом разряде записывают 1, если число отрицательное и 0, если положительное.

С помощью 8 бит (1 байт) можно закодировать 256 чисел. Это объясняется очень просто. В каждом разряде двоичного числа можно записать либо 0, либо 1. Следовательно, всего два варианта. Таких разрядов восемь, значит можно реализовать всего 2^8=256 вариантов. Например, переменная типа ***byte*** может хранить целые числа в диапазоне 0…255. Так как диапазон начинается с 0, то последнее число не 256, а 255.

Все целочисленные типы данных *Turbo Pascal* описаны в табл. 3.1.

Таблица 3.1

Целые типы Turbo Pascal

Целочисленный тип данных	Память (бит)	Диапазон возможных значений
Byte	8	$0..2^8-1$ (0…255)
Word[14]	16	$0..2^{16}-1$ (0…65535)
ShortInt	8	$-2^7..2^7-1$ (-128…127)
Integer	16	$-2^{15}..2^{15}-1$ (-32768…32767)
LongInt	32	$-2^{31}..2^{31}-1$(-2147483648…2147483647)

[14] Кстати, название типа «word» (что в переводе с английского значит «слово») обозначает размер машинного слова, т.е. той минимальной области памяти, которую может адресовать процессор. Соответственно, чем выше разрядность, тем больше памяти занимает переменная типа word. Потому на разных типах компьютеров величина word разная. Однако исторически сложилось, что word стал обозначать слово на 16-разрядном процессоре. И практически во всех современных видах языка Pascal применяется именно приведенный диапазон значений для данного типа, какие бы процессоры при этом не использовались.

Для хранения вещественных (дробных) чисел существует еще несколько типов данных, они приведены в следующей таблице.

Таблица 3.2

Вещественные типы Turbo Pascal

Вещественный тип данных	Память (бит)	Точность (десятичные разряды)	Диапазон возможных значений
Single	32	7−8	$\pm 1{,}5 \cdot 10^{-45} .. \pm 1{,}5 \cdot 10^{38}$
Real	48	11−12	$\pm 2{,}9 \cdot 10^{-39} .. \pm 1{,}7 \cdot 10^{38}$
Double	64	15−16	$\pm 5{,}0 \cdot 10^{-324} .. \pm 1{,}7 \cdot 10^{308}$
Extended	80	19−20	$\pm 1{,}9 \cdot 10^{-4951} .. \pm 1{,}1 \cdot 10^{4932}$
Comp	64	19−20	$-9{,}2 \cdot 10^{18} .. 9{,}2 \cdot 10^{18}$

Для использования этих типов данных, кроме типа ***Real***, необходимо подключение математического сопроцессора. На современных компьютерах он встроен в *CPU*, но в первых поколениях персональных ЭВМ он устанавливался дополнительно и мог физически отсутствовать, с тех пор компилятору необходимо отдельно указывать на подключение[15] математического сопроцессора.

Тип данных ***Real*** подключения математического сопроцессора не требует, а потому является наиболее часто используемым типом данных для хранения действительных чисел.

Стоит отметить, что тип ***Comp*** хранит не вещественные, а целые числа и по свой сути является расширением целочисленного типа ***Longint*** до 19 разрядов.

Любое вещественное число, а также число типа ***Comp*** в *Pascal* может быть представлено одним из следующих способов:

[15] Для его подключения в Turbo Pascal можно перед разделом заголовка программы написать директиву компилятора {$N+}, либо в меню Options/Compiler установить флажок в поле Numeric processing 8087/80287.

ПРИМЕР

123456.789 1.23456789E+05 123456789E–03.

Здесь показано, как можно разными способами записать одно и то же число 123456,789. Согласно отечественным стандартам, целую и дробную части числа отделяют с помощью плавающей запятой. В международных стандартах, которые заложены в *Pascal*, для этого используется плавающая точка, поэтому в примерах записана именно точка, а не запятая. Буква E используется для отделения *мантиссы* от *экспоненты*. *Экспонента* – это степень[16], в которую возводят число 10 при домножении на *мантиссу* числа.

Для большей наглядности поставим в соответствие каждому числу из примера обычную его математическую запись (табл. 3.3).

Таблица 3.3

Соответствие обычной и машинной записи чисел

Запись числа в TP	**Обычная математическая запись**
123456.789	123456,789
1.23456789E+0005	$1{,}23456789 \cdot 10^{5}$
123456789E–0003	$123456789 \cdot 10^{-3}$

В памяти ЭВМ можно хранить не только числа, но и другие виды информации (например, текст). В *Turbo Pascal* существует два вида простых текстовых типов данных. Первый из них – ***char***, это тип данных, позволяющий хранить в одной переменной один символ. Переменная данного типа занимает в памяти 1 байт. Из этого можно сделать вывод, что, как и переменные типа ***byte***, символьные переменные ***char*** могут хранить в себе один из 256 вариантов комбинаций двоичных цифр.

[16] Не путать с функцией экспоненты, которая возводит аргумент в степень основания натурального логарифма e=2,71828…

Выше уже говорилось о таблицах кодировок (например, *ASCII*). Т. е. каждый символ можно описать либо кодом (числом), либо собственно символом, набираемым с клавиатуры. Если переменной типа ***char*** присваивают значение, используя код, то перед его десятичной записью ставят символ ***#;*** если символ кодируют в шестнадцатеричной[17] системе исчисления, то ставят такое сочетание символов: ***#$***. Если же необходимо присвоить символ, не зная его кода, то, присваивая его переменной, заключают этот символ в апострофы. Рассмотрим на примере как переменной типа ***char*** можно присвоить значение латинской заглавной *A*.

ПРИМЕР

```
c:=#65;
c:=#$41;
c:='A';
```

Чтобы сохранить в памяти машины не один символ, а их последовательность, можно использовать тип ***string***. Переменная этого типа может хранить их до 255 одновременно, при этом в памяти она занимает 256 байт (первый байт содержит длину строки). ***String*** занимает особое место среди типов данных *Pascal*. И, хотя, мы рассматриваем его среди простых типов данных, к таковым он формально не относится. Правильнее тип ***string*** рассматривать как *массив* значений типа ***char***.

Последний простой тип данных, рассматриваемый здесь, – ***boolean***. Это логический тип, описание которого было достаточно подробно приведено, когда излагались основы алгебры логики. Переменные этого типа могут хранить лишь два значения: ***TRUE*** и ***FALSE*** (пишут без апострофов или иных знаков пунктуации). Пример задания значения переменной типа ***boolean***:

```
L:=TRUE;
```

[17] В языке Pascal для обозначения шестнадцатеричного формата чисел используется префикс $ (т. е., например, 12_{16}=$12=18_{10}$).

или так:

```
L:= (3>2) OR (3<2);
```

Кроме этого, в ***Pascal*** выделяют порядковые типы данных. К ним относят те типы, возможные значения которых можно пронумеровать. Все целочисленные типы – порядковые, так же как и ***char*** с ***boolean***. Можно сказать иначе – все простые типы данных, кроме действительных чисел, являются порядковыми.

К порядковым так же относят перечисляемый (тип-список) и ограниченный (тип-диапазон) типы. Ниже даны примеры описания.

Тип-список:

```
Nechet: (1,3,5,7,9);
DenNedeli: (Ponedelnik, Vtornik, Sreda, Chetverg, Piatnica, Subbota Voskresenie);
```

Тип-диапазон:

```
Mesiac: 1..12;
NomerDnia: 1..7;
```

Общая генеалогия типов представлена на рис. 3.1.

3.4 Математические операции и функции

Напомним, что оператор присваивания в *Pascal* записывается как «:=». В математике присваивают переменной значение с помощью знака «=», причем неважно, с какой стороны от знака записана переменная (например, «*X*=5» или «5=*X*», будет означать одно: переменной *X* присвоено значение 5). Здесь нужно отличать присвоение от сравнения. В *Pascal* с помощью знака «=» записывается операция отношения *равенство*. Поэтому чтобы отличить присвоение от сравнения, оператор присваивания записывают с двоеточием.

Вот пример:

```
X=5;
```

Здесь переменная *X* сравнивается со значением 5, но это действие бессмысленно, так как его результат никак не используется, не печатается и даже не запоминается. Правильнее было написать так:

```
Y:= X=5;
```

Эта строка читается следующим образом: переменной *Y* присвоить результат сравнения переменной *X* с числом 5. Если до этого переменной *X* было присвоено 5, то в *Y* в результате присвоения будет значение ***TRUE***, иначе – ***FALSE***.

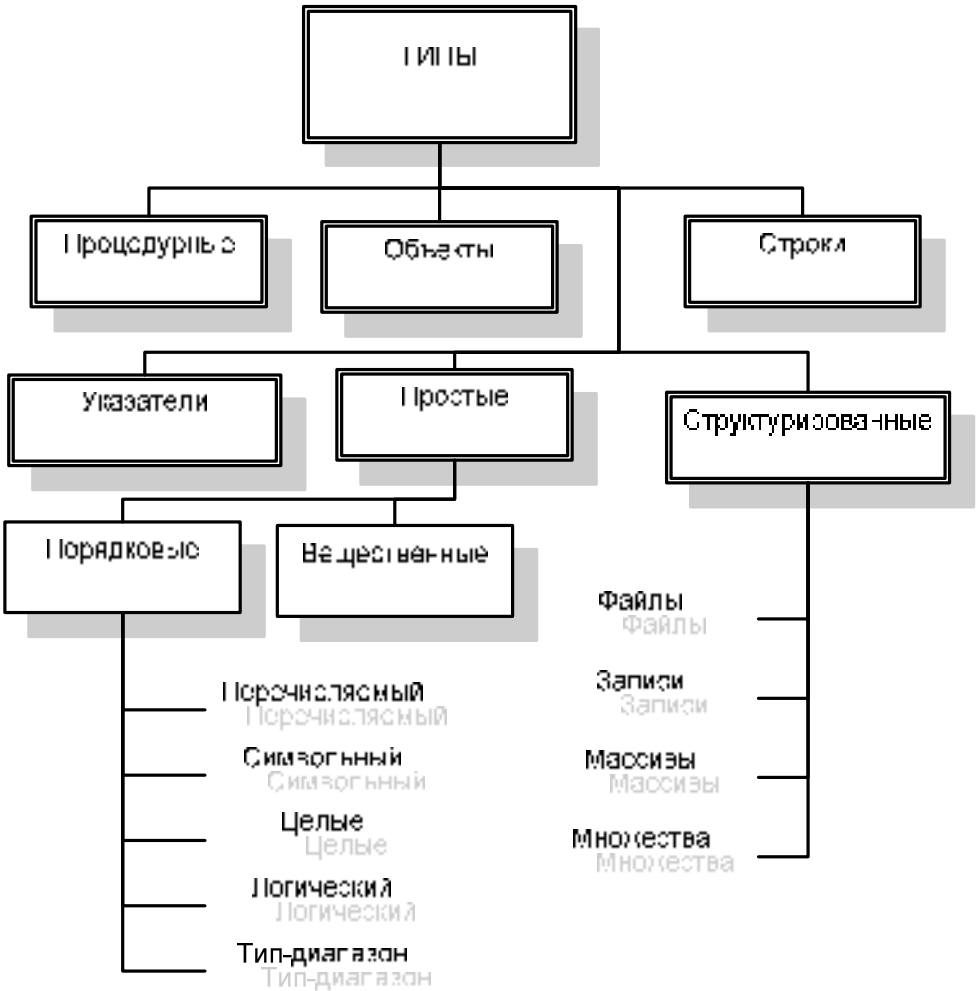


Рис. 3.1. Генеалогия типов языка Pascal

В языке *Pascal* определены шесть стандартных арифметических операций: *сложение*, *вычитание*, *умножение*, *деление*, *остаток от деления* и *деление без остатка* (+, –, *, /, ***mod*** и ***div***). Первые четыре операции не требуют комментария, а последние две разберем подробнее. Присвоим переменной *M* остаток от деления 19 на 7, а переменной *D* - результат вычисления операции деления без остатка 19 на 7:

```
M:=19 mod 7;
D:=19 div 7;
```

В результате в *M* будет число 5, а в *D* – число 2.

Кроме всего прочего, в *Pascal* определены операции, возвращающие логические значения; это, прежде всего, операции отношения:

больше: «>»; *меньше*: «<»; *больше или равно*: «>=»; *меньше или равно*: «<=»; *равно*: «=»; *неравно*: «<>».

Среди операций алгебры логики присутствуют все четыре рассмотренные ранее: ***OR***, ***XOR***, ***AND*** и ***NOT***.

Стандартный математические функции языка *Pascal* можно описать в виде таблицы (табл. 3.4).

Таблица 3.4

Стандартные математические функции Pascal

Математическая запись функции	Запись функции в TP
$\|x\|$	***abs(x)***
e^x	***exp(x)***
cos x	***cos(x)***
sin x	***sin(x)***
arctg x	***arctan(x)***
ln x	***ln(x)***
$\sqrt{x}$	***sqrt(x)***
x^2	***sqr(x)***
π	***Pi***

При использовании этих функций результат получается типа ***real***. Но есть и исключение. При вычислении $|x|$ результат получается того же типа,

что и аргумент x. Все тригонометрические вычисления производятся в радианах.

Кроме того, здесь представлены не все известные функции. Нет, например, тангенса. Для его вычисления потребуется воспользоваться известным тригонометрическим тождеством $\mathrm{tg}(x)=\frac{\sin(x)}{\cos(x)}$. Кроме того,

$$\arcsin(x)=\mathrm{arctg}\left(\frac{x}{\sqrt{1-x^2}}\right);\ \arccos(x)=\mathrm{arctg}\left(\frac{\sqrt{1-x^2}}{x}\right).$$

Ну а для возведения в степень можно использовать формулу, верную для положительных значений x: $x^y=e^{\ln(x^y)}=e^{y\ln(x)}=\exp(y\ln(x))$.

Некоторые другие системные процедуры и функции *Turbo Pascal* описаны в табл. 3.5.

Таблица 3.5

Системные процедуры и функции Pascal

Функция или процедура	**Тип аргумента**	**Тип результата**	**Описание**
random		***real***	Случайное число [0, 1), (может быть равно 0, но строго меньше 1)
random(x)	***word***	***word***	Случайное число [0, *x*), (может быть равно 0, но строго меньше *x*)
randomize			Процедура инициализации псевдослучайного ряда
inc(x,n)	***x: порядковый тип; n:integer***	***Порядковый тип***	Меняет значение *x*, присваивая ему значение, отстоящее на *n* от *x* в описании порядкового типа. Например, если *x*=2 (x:byte), *n*=7, то после выполнения функции x будет равен 9, а если задать *x*=255 (x:byte), то в результате получим *x*=6

Продолжение табл. 3.5

Функция или процедура	**Тип аргумента**	**Тип результата**	**Описание**
inc(x)	Порядковый тип	Порядковый тип	Меняет значение *x*, присваивая ему значение, отстоящее на 1 от *x* в описании порядкового типа. Например, если *x*=2 (***x:byte***), то после выполнения функции *x* будет равен 3, а если задать *x*=255 (***x:byte***), то в результате получим *x*=0
dec(x,n)	*x*: порядковый тип; *n*:integer	Порядковый тип	Меняет значение *x*, присваивая ему значение, отстоящее на *–n* от *x* в описании порядкового типа. Например, если *x*=9 (***x:byte***), *n*=7, то после выполнения функции *x* будет равен 2, а если задать *x*=0 (***x:byte***), то в результате получим *x*=249
odd(x)	***longint***	***boolean***	Дает ***TRUE***, если *x* – нечетное, ***FALSE***, если *x* – четное
dec(x)	порядковый тип	Порядковый тип	Меняет значение *x*, присваивая ему значение, отстоящее на –1 от *x* в описании порядкового типа. Например, если *x*=2 (***x:byte***), то после выполнения функции x будет равен 1, а если задать *x*=0 (***x:byte***), то в результате получим *x*=255
int(x)	***real***	***real***	Целая часть числа *x*. Если *x*=2.123, то ***int(x)***=2.000
frac(x)	***real***	***real***	Дробная часть числа *x*. Если *x*=2.123, то ***frac(x)***=0.123. Можно записать равенство: ***X=int(X)+frac(X)***

Окончание табл. 3.5

Функция или процедура	Тип аргумента	Тип результата	Описание
trunc(x)	***real***	***longint***	Целая часть числа *x* (дробная часть просто отбрасывается). Значение *x* должно лежать в диапазоне ***longint***. Например, если *x*=28.9, то ***trunc(x)***=28
round(x)	***real***	***longint***	Округление числа *x* по закону 4/5. Например, если *x*=28.56, то ***round(x)***=29, если *x*=28.4, то ***round(x)***=28

3.5. Простейший ввод/ вывод

Программа в процессе своей работы может взаимодействовать с пользователем. Т. е. она может запросить у него некоторую информацию, или, наоборот, предоставить ее ему. Общение с пользователем, впрочем, как и с любым другим устройством, происходит при помощи операций ввода/вывода. Если речь идет о выводе, то данные могут поступать на различные устройства (экран, принтер, файл, звуковая система). Наиболее распространенным устройством вывода при общении человека и ЭВМ является монитор. А устройством ввода – клавиатура.

Для вывода текстовой информации на экран в языке *Pascal* в консольном[18] режиме, служат две процедуры: ***write*** и ***writeLn***. Они имеют такой формат:

write(a, b, c, …);

writeLn(a, b, c, …);

[18] Консольным чаще всего называют взаимодействие с пользователем посредством ввода/вывода текстовой информации. Иногда консольный режим работы приложения называют текстовым интерфейсом пользователя.

Здесь *a*, *b*, *c* – параметры вывода (переменные, константы, выражения). Процедура ***writeLn*** может быть задана без параметров (в этом случае она просто переводит курсор на строчку ниже).

Разберем первую процедуру.

Чтобы вывести некоторый текст, можно воспользоваться ею так:

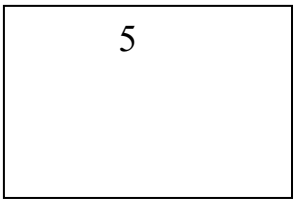
```
write('текст');
```

Содержимое экрана после выполнения этой процедуры:

Как видно из примера, выводимый текст заключается в апострофы. Если же требуется вывести значение некоторой переменной, то необходимо указать лишь ее имя.

```
var
  x:integer;
…
 x:=5;
write(x);
```

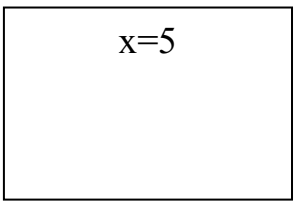
Содержимое экрана:



Эти два способа вывода можно комбинировать так:

```
var
  x:byte;
 …
x:=5;
write('x=',x);
```

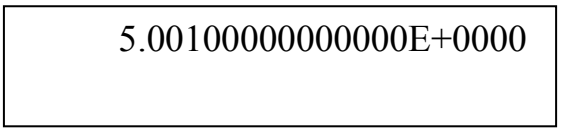
Содержимое экрана:



В этих примерах производится вывод значенй переменных типа ***byte***, но если выводить на экран переменную типа ***real*** (или любого типа описанного выше в одной таблице с ним), то она будет представлена на экране в экспоненциальном виде. Например:

```
var
  x:real;
 …
x:=5.001;
write(x);
```

Содержимое экрана:



Такая запись чисел удобна, если их значения имеют в экспоненте значения больше трех, но чаще всего в учебных задачах используют значения, для записи которых вообще нет необходимости в экспоненте. В этом случае лучше при выводе числа использовать форматирование или так называемую маску вывода: ***write(x:m:n)***.

Здесь ***m*** – количество знакомест на экране, выделяемое для печати всего числа; ***n*** – количество знаков после десятичной точки (до скольких знаков число округляется). Указанное форматирование применимо только для чисел, которые имеют вещественный формат (***real***). Для целочисленных переменных вывод их значения следующий: ***write(x:m)***. Т.е. число знаков после запятой не указывается, поскольку их там просто нет. Рассмотрим пример вывода:

```
var
  x:real;
```

```
…
x:=7.538;
write(x:6:2);
```

Содержимое экрана:

Символ «▒» обозначает пробел. В этом примере число округлено до двух знаков после десятичной точки, для него выделено на экране шесть знакомест, но так как оно заняло лишь четыре, то оставшиеся два знакоместа слева были заполнены пробелами.

Процедура ***writeLn*** выполняет те же функции, что и ***write***, но в отличие от нее после вывода текста переводит курсор в начало следующей строки. Вообще, в скобках у процедур вывода указывается через запятую все то, что нужно вывести. Выводу подвергаются константы и переменные простых типов непосредственно. Для вывода сложных типов, таких как, например, массивы и записи, необходимо указывать конкретное поле вывода или адрес элемента в массиве. Вывод строк осуществляется непосредственно цельно.

Для ввода значений с клавиатуры используют процедуру ***readLn***. Она приостанавливает работу программы, ожидая ввода значения. После нажатия клавиши *Enter* набранное значение помещается в первую переменную, указанную в качестве параметра процедуры. Затем после следующего ввода набранное значение помещается в следующую переменную и т. д. Формат у процедуры таков:

readLn(*a, b, c, …****);***

где *a*, *b*, *c* – параметры ввода (переменные). Процедура ***readLn*** может быть задана без параметров; в этом случае она просто приостанавливает выполнение программы до тех пор, пока не будет нажата клавиша *Enter*.

Здесь стоит сказать, что обычно ***readLn*** не используется отдельно от вывода комментария, поскольку просто приостанавливает работу алгоритма, ожидая нажатия клавиши. Пользователь программы, естественно, в большинстве случаев будет просто не в курсе того, что и в какой последовательности в данный момент нужно вводить. Для внесения ясности нужно обязательно делать *приглашение ко вводу*, как правило, при помощи процедур ***write*** / ***writeLn***. Потому всегда, где в блок-схемах алгоритмов идет блок ввода, подразумевается в программе наличие как минимум пары стандартных процедур. Это, помимо, ***readLn***, еще и предшествующий ***write*** / ***writeLn***, выводящий комментарии к тому, что нужно ввести в данном месте. Пример записи процедуры:

ПРИМЕР

Ввести с клавиатуры значение аргумента и вычислить значение функции y = sin(x).

```
Program ex_IO;
var
  x,y:real;
begin
  write('x=');
  readLn(x);
  y:=sin(x);
  writeLn('y=',y:6:2)
end.
```

Содержимое экрана:

x=–2.36 ↵
y=▒–0.70

Символом ↵ обозначено место, в котором нажата клавиша *Enter*.

3.6 Строковый тип данных

Как отмечалось выше, для работы с последовательностями символов используют строковый тип данных ***string***. *Pascal* отличается относительной простотой работы с этим типом данных. В *Pascal* существуют, прежде всего, строковые константы и строковые переменные. Любая строковая константа заключается в апострофы.

Строки в *Pascal* состоят не более чем из 255 символов типа ***char***, причем в нулевом байте строки содержится ее длина. Для ввода/вывода строк используют стандартные процедуры ***write***/***writeLn*** и ***readLn***.

Для объявления переменной строкового типа в разделе описания нужно указать ключевое слово ***string***. В этом случае под строку будет выделено максимально возможное количество памяти, т. е. 256 байт. Если заранее известно, что строка не будет принимать такие длинные значения, то нужно пользоваться строкой ограниченного размера. Для этого пишут в квадратных скобках количество значимых символов.

ПРИМЕР

```
var
  srt1: string;
  str2: string[10];
  str3: string[255];
```

Здесь переменные ***str1*** и ***str3*** имеют одинаковый размер (256 байт), а переменная ***str2*** занимает 11 байт и может хранить 10 полезных символов.

В *Pascal* строки можно обрабатывать двумя способами. Первый способ предполагает строку единым неделимым объектом, а второй, соответственно относится к строке как к сложной структуре, состоящей из отдельных символов.

Первый способ весьма удобен и является отличительной чертой именно языка *Pascal* по сравнению, например, с *C*. Так, для присвоения значения строковой переменной достаточно просто записать значение,

которое в нее будет помещено. Довольно легко организована операция строковой контактезации или сцепления. Это выражается, прежде всего, в том, что для ***string*** определен оператор «+».

Рассмотрим простейшую интерактивную программу.

ПРИМЕР

```
writeLn('Как Вас зовут?');
readLn(Name);
str1 := 'Привет, ';
str2 := 'от Деда Мороза';
str3 := str1 + Name + ', ' + str2 + ' и Снегурочки!';
writeLn(str3);
```

В результате на экране появится приглашение к вводу своего имени и далее программа поприветствует пользователя от имени сказочных персонажей.

При работе со строкой как с массивом символов возможно прямое обращение к ее составляющим.

ПРИМЕР

```
str1 := 'отдел Оптика';
str1[7] := 'А';
str1[10] := 'е';
```

В результате слово «Оптика» поменяется на «Аптека».

Некоторые полезные функции и процедуры работы со строками представлены в табл. 3.6.

Таблица 3.6

Процедуры и функции работы со строками

Имя	П/Ф	Описание
Length(s)	Ф	Возвращает длину строки
Delete(s, k, m)	П	Удаляет в строке *s* *m* символов, начиная с позиции *k*
Insert(subs,s,k)	П	Вставляет подстроку *subs* в строку *s* с позиции *k*
Str(x,s) ***Str(x:n,s)*** ***Str(x:n:m,s)***	П	Преобразует *x* к строковому представлению (во втором и третьем случаях согласно формату вывода, устанавливаемому *n* и *m,* где *n* – общее число знаков, *m* – из них после запятой) и записывает результат в строку *s*

Окончание табл. 3.6		
Val(s,v,err)	П	Преобразует строку *s* к числовому представлению и записывает результат в переменную *v*. Если преобразование возможно, то в переменной *err* возвращается 0, если невозможно, то в *err* возвращается ненулевое значение
Pos(subs,s)	Ф	Возвращает позицию первой подстроки subs в строке *s* (или 0 если подстрока не найдена)
UpCase(c)	Ф	Возвращает символ *c*, преобразованный к верхнему регистру (не везде корректно происходит работа с кириллицей)

3.7. Программирование развилок

Развилка является одной из наиболее часто употребляемых алгоритмических управляющих структур. В языке *Pascal* развилки записываются так:

if *P* ***then***

управляющий оператор положительной ветви

else

*управляющий оператор отрицательной ветви****;***

Это полная развилка (обратите внимание, что перед ***else*** «***;***» не ставится). Неполная же развилка имеет следующий вид:

if *P* ***then***

*управляющий оператор положительной ветви****;***

Здесь *P* – предикат развилки. Управляющий оператор положительной ветви выполняется если *P*=***TRUE***, иначе выполняется управляющий оператор отрицательной ветви.

ПРИМЕР

Наибольшее из трех неравных чисел (A, B, C) возвести в квадрат.

Этот пример демонстрирует возможность использования разветвляющихся процессов. Блок-схема программы представлена на рис. 3.2.

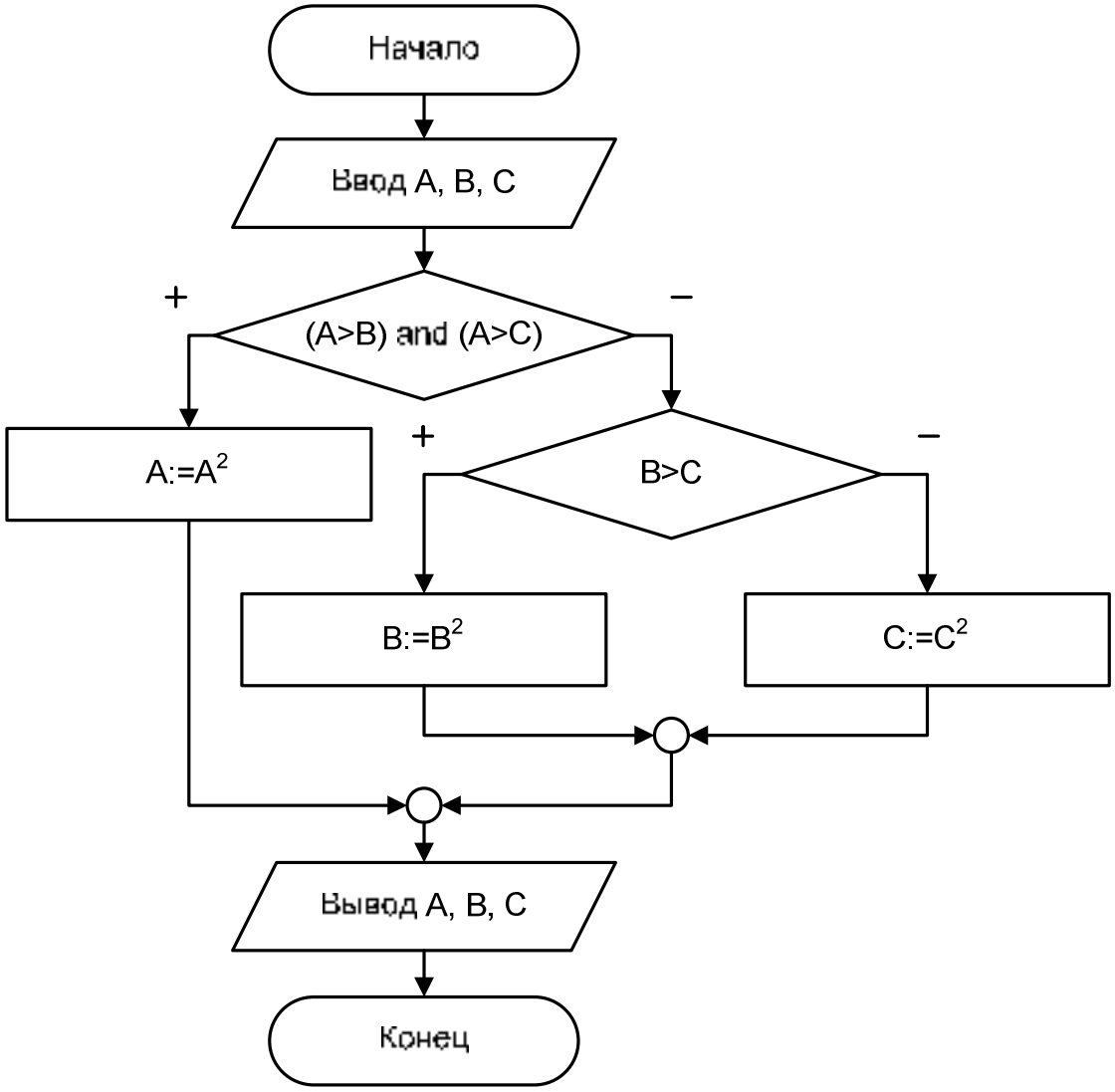


Рис. 3.2. Возведение в квадрат наибольшего из трех неравных чисел

```
Program Max3;
Var A,B,C : integer;
begin
  writeLn('Введите 3 числа (A,B,C)');
  readln(A,B,C);
  if (A>B) and (A>C) then
    A:=sqr(A)
  else
    if B>C then
      B:=sqr(B)
    else
```

```
        C:=sqr(C);
    writeLn('A=',A,' B=',B, ' C=', C);
end.
```

Дословно «*if… then… else…*» переводится как «*если… тогда… иначе…*».

Развилка есть процесс принятия решения относительно выбора дальнейшего движения в зависимости от некоторого условия. Наиболее часто используемыми являются развилки с вариантом решения «да» или «нет». Однако существует так называемая развилка множественного выбора (т. е. когда имеется возможность движения далее по алгоритму не в двух направлениях, а в гораздо большем их количестве).

Для этого используют развилку с множеством путей. Записывается она следующим образом:

case *n* ***of***

*мвз1****:*** *управляющий оператор для 1-го варианта****;***

*мвз2****:*** *управляющий оператор для 2-го варианта****;***

*мвз3****:*** *управляющий оператор для 3-го варианта****;***

…

else *управляющий оператор для ветви «иначе»****;***

end;

Опять подробности работы этой управляющей структуры удобнее рассматривать на примере. Блок-схема алгоритма приведена на рис. 3.3. Обратите внимание на закрывающую скобку ***end*** при отсутствии ***begin*** в структуре ***case***.

Здесь *n* – переменная-селектор, *мвз* – непересекающиеся множества возможных значений переменной-селектора, *мвз* можно описать как отдельным значением, так и списком значений (6,7) или диапазоном значений (1..3).

ПРИМЕР

По введенному номеру дня, определить к какой части недели он относится.

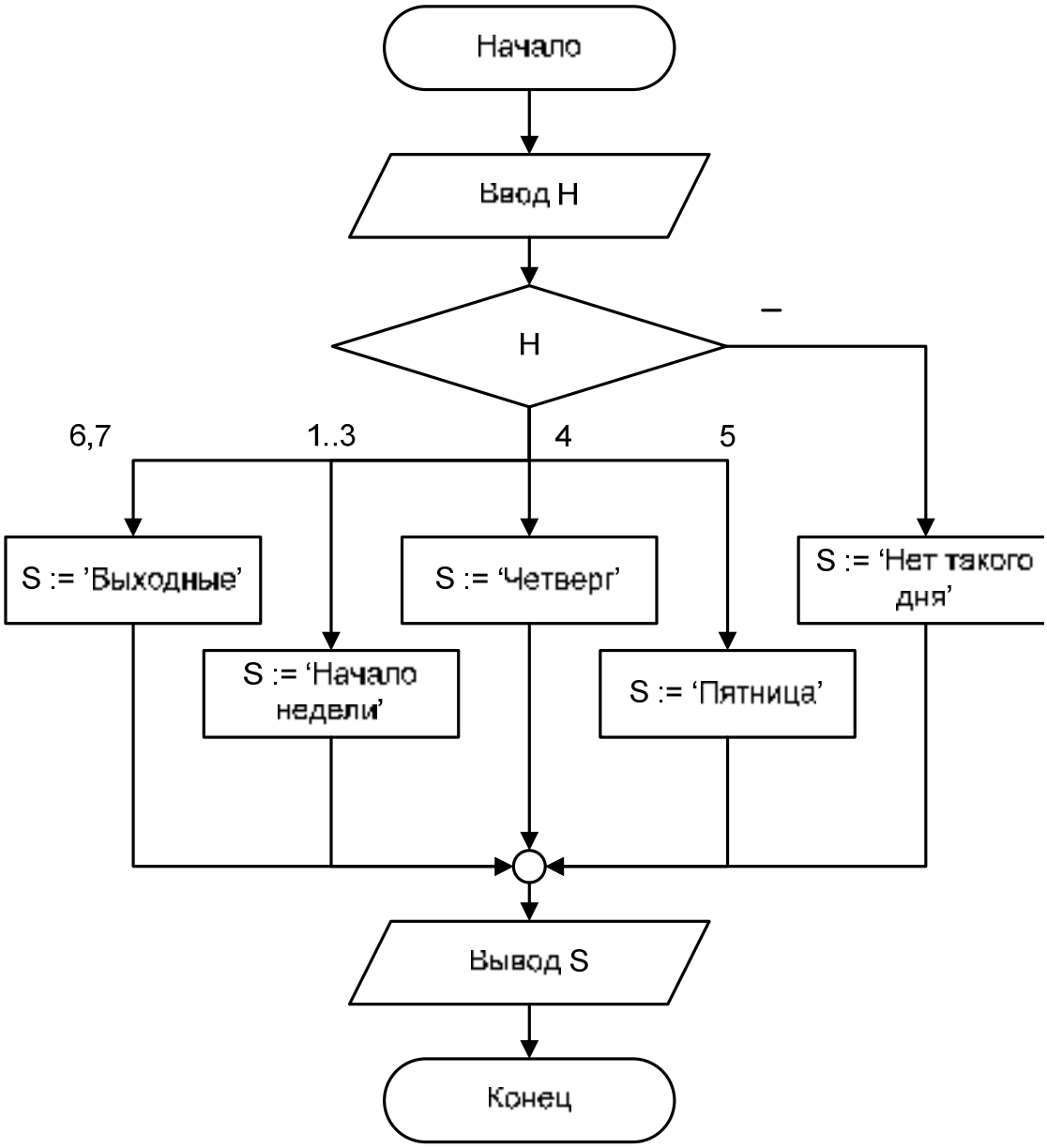


Рис. 3.3. Пример работы оператора Case

```
Program CaseWeek;
Var H :integer;
    S :string[20];
begin
  writeLn('Введите номер дня недели:');
  readln(H);
  case H of
```

```
    1..3: S:='начало недели';
       4: S:='четверг';
       5: S:='пятница';
     6,7: S:='выходные';
    else
          S:='нет такого дня';
  end;
   writeLn('Дню ', N, ' соответствует ', S);
end.
```

3.8. Программирование циклов

В Pascal представлены все три рассмотренные ранее циклические алгоритмические управляющие структуры. Цикл с *предусловием*, или цикл «*пока*» имеет следующую реализацию:

while** P **do

*тело цикла**;***

«*while … do*» можно перевести как «*пока истинно… выполняй*».

Цикл с постусловием записывается как

repeat

*тело цикла**;***

until** P**;

«*repeat… until*» можно перевести как «*повторяй… до тех пор пока не*».

Цикл с параметром и с шагом «+1» записывается так:

for k:=n to m do

*тело цикла**;***

Дословный перевод «*for k:=n to m do*» таков: «*для k присвоить n до m выполнять*». Если цикл идет в обратную сторону, т. е. шаг равен «–1», то он принимает вид:

for k:=n downTo m do

*тело цикла**;***

Рассмотрим работу всех трех циклов на примере одной задачи.

ПРИМЕР

Протабулировать функцию $y=x^2$ на промежутке [−2, 1] с шагом 0,5.

Табуляция – это табличное представление функции, т. е. каждому *x* из рассматриваемого диапазона ставится в соответствие вычисляемое *y*, которые в паре выводятся на экран.

Реализация программы, используя цикл «***while***» (блок-схема на рис. 3.4):

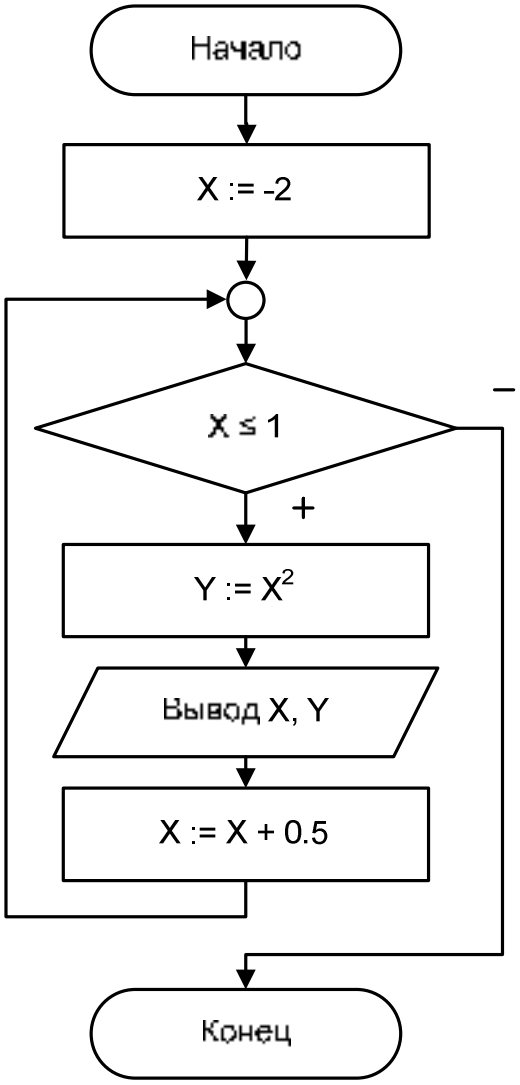


Рис. 3.4. Табулирование функции циклом while

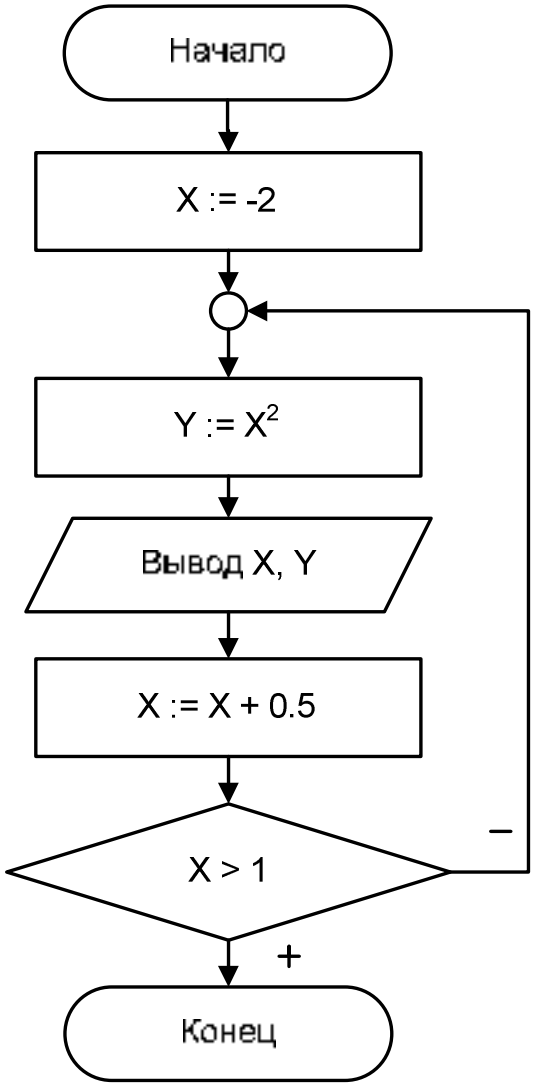


Рис. 3.5. Табулирование функции циклом repeat... until

```
Program TabFWhile;
Var X,Y : real;
begin
  X := -2;
  while X<=1 do
    begin
      Y := sqr(X);
      writeLn('f(',X:5:1,
')=',Y:8:3);
      X:=X+0.5;
    end;
end.
```

Реализация программы. используя цикл «***repeat*** … ***until***» (блок-схема на рис. 3.5):

```
Program TabFRepeat;
Var X,Y : real;
begin
  X := -2;
  repeat
    Y := sqr(X);
    writeLn('f(',X:5:1,
            ')=',Y:8:3);
    X:=X+0.5;
  until X>1;
end.
```

Реализация программы с помощью цикла «***for***» (блок-схема на рис. 3.6):

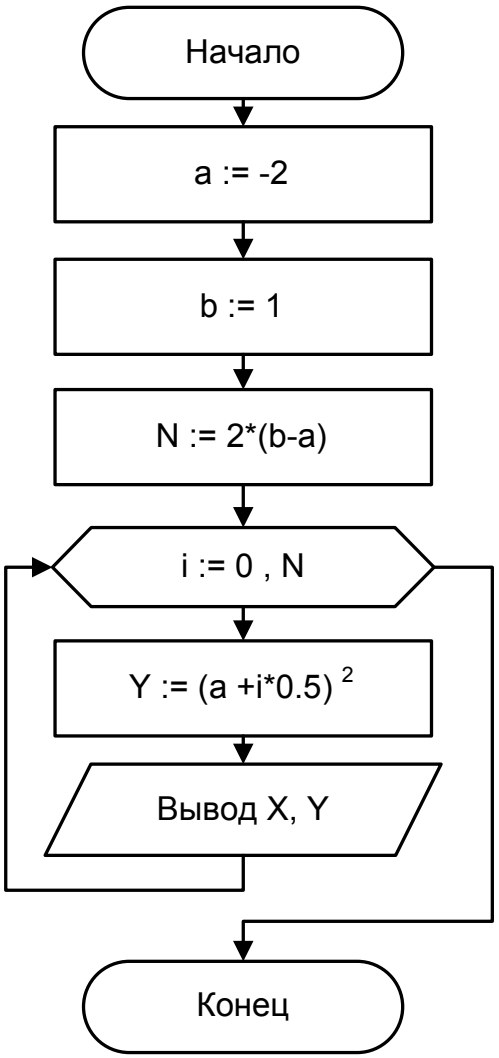


Рис. 3.6. Табулирование функции циклом for

```
Program TabFFor;
Var a,b,x,y : real;
    i,N: integer;
begin
  a:=-2;
  b:=1;
  N := trunc(2*(b-a));
  for i:=0 to N do
    begin
      Y := sqr(a+i*0.5);
      writeLn('f(', (a+i*0.5):5:1, ')=',Y:8:3);
    end;
end.
```

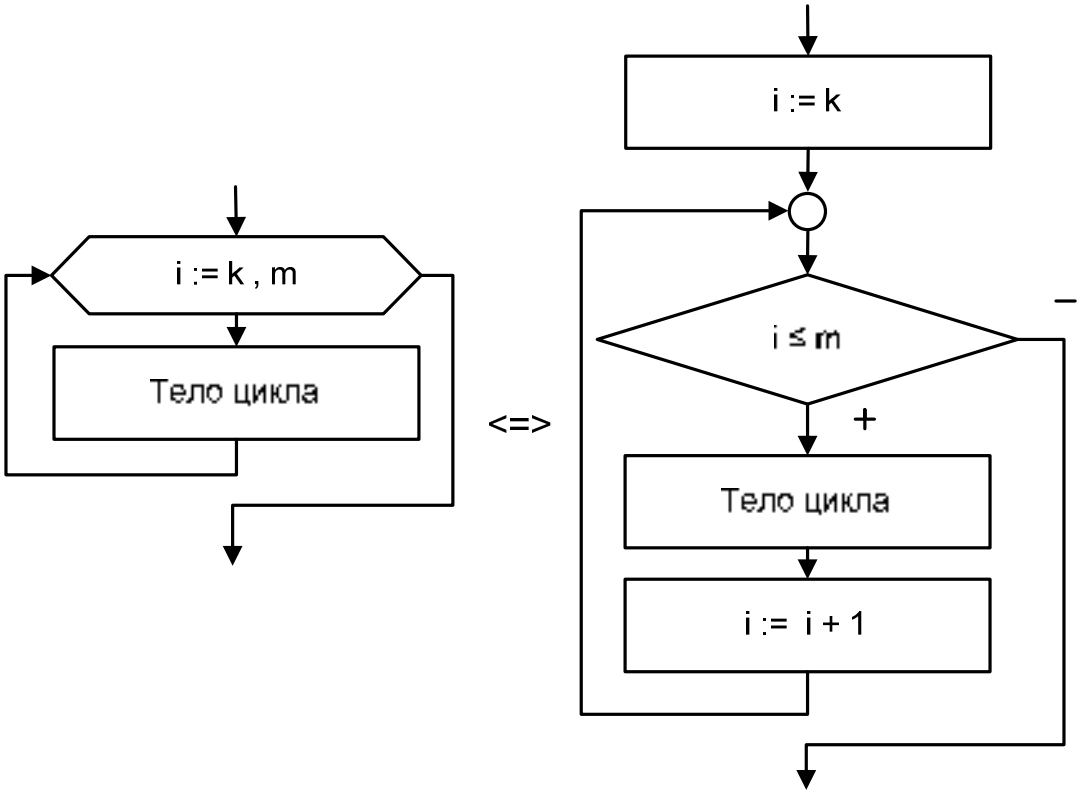


Рис. 3.7. Эквивалентное преобразование цикла for в цикл while

В результате работы всех трех программ на экране появится таблица значений функции:

f(-2.0)= 4.000

f(-1.5)= 2.250

f(-1.0)= 1.000

f(-0.5)= 0.250

f(0.0)= 0.000

f(0.5)= 0.250

f(1.0)= 1.000

Как отмечалось ранее, цикл ***for*** в *Pascal* является частным случаем цикла ***while***. Эквивалентность этих двух алгоритмов изображена на рис. 3.7.

3.9. Составной оператор

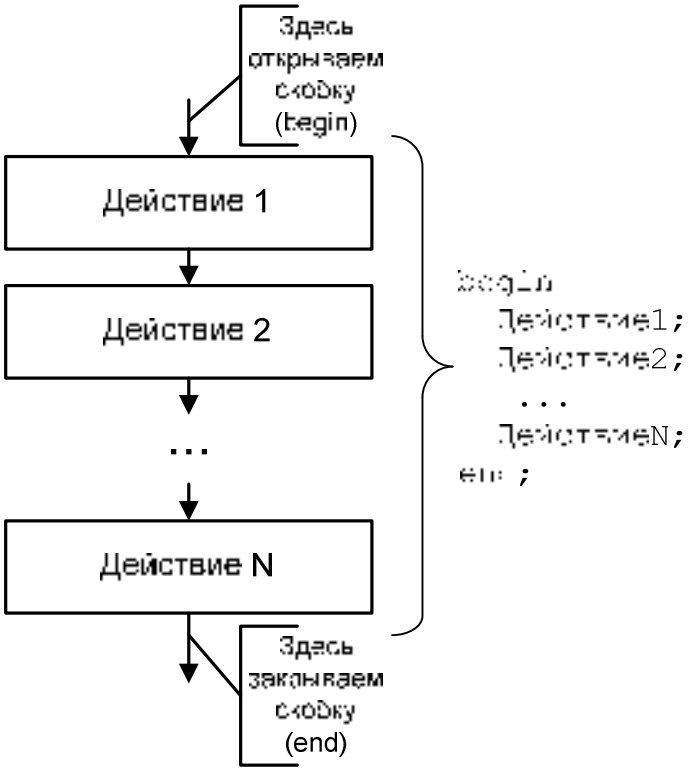


Рис. 3.8. Сборка составного оператора

Итак, выше были описаны основные алгоритмические конструкции языка *Pascal*. Если посмотреть на изображение блок-схем этих конструкций, то можно выделить в их составе блок действия (прямоугольник). Все достаточно просто и понятно, если действие выполняется одним оператором или процедурой; однако, если необходимо выполнить несколько подряд идущих строчек кода, то для этого их нужно объединить. Нужно сделать так, чтобы все эти строчки для компилятора представляли собой один сложный оператор. Для этого данную последовательность заключают в так называемые *операторные скобки*. А все то, что находится внутри этих скобок принято называть составным оператором.

В качестве операторных скобок языка *Pascal* выступают ***begin*** и ***end*** (***begin*** – открывающая скобка, ***end*** – закрывающая). Ранее при демонстрации решения задач на циклы мы уже пользовались операторными скобками для выделения границ тел циклов ***while*** и ***for***.

Иллюстрация правила расстановки скобок показана на рис. 3.8.

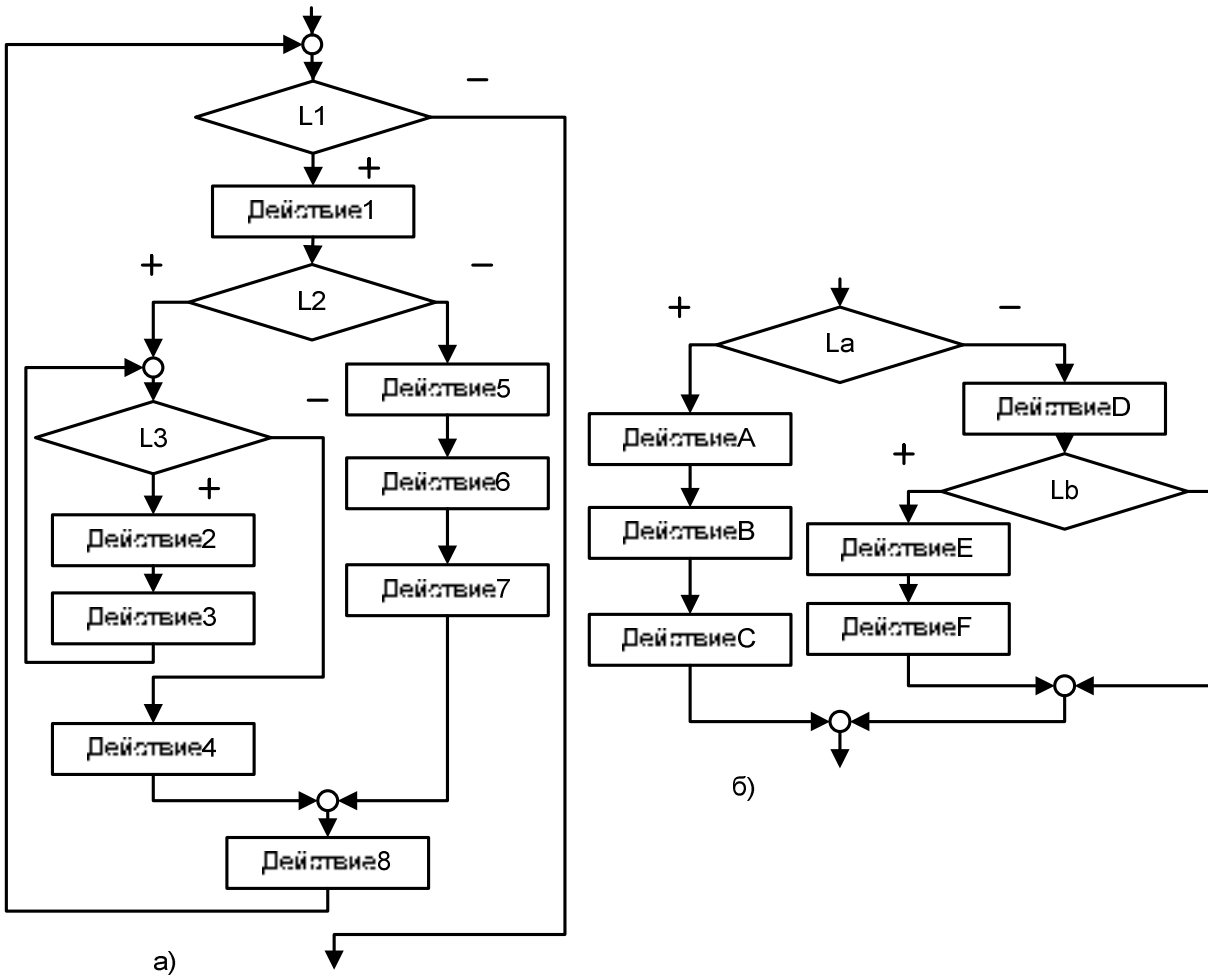


Рис. 3.9. Примеры алгоритмов, в которых используется составной оператор

Вообще, скобки ***begin end*** нужно ставить при составлении программы по блок-схеме в том месте, где на одной ветви встречается более одного независимого оператора. Некоторые примеры расстановки операторных скобок можно видеть на рис. 3.9, *а* и рис. 3.9, *б*. Для случая на рис. 3.9, *а* текст кода на *Pascal* имеет примерно такой вид:

```
while L1 do
  begin
    Действие1;
    if L2 then
      begin
        while L3 do
          begin
            Действие2;
            Действие3;
          end;
        Действие4;
      end
    else
      begin
        Действие5;
        Действие6;
        Действие7;
      end;
    Действие8;
end;
```

А для случая на рис. 3.9б *Pascal*-код таков:

```
if La then
  begin
    ДействиеA;
    ДействиеB;
    ДействиеC;
  end
else
  begin
    ДействиеD;
    if  Lb then
      begin
        ДействиеE;
        ДействиеF;
      end;
  end;
```

4. ОДНОМЕРНЫЕ МАССИВЫ

4.1. Понятие и объявление массива

В инженерной деятельности приходится в основном работать с большими объемами информации. Для корректной обработки приходится использовать различные методики ее анализа и представления. Для представления больших объемов данных можно применить объединение их по некоторому признаку, в результате чего получится *массив данных*.

В программировании ***массивами*** называют набор однотипных переменных объединенных общим названием. Элементы массива расположены в одном месте памяти и каждая переменная входящая в его состав имеет свой номер (*индекс*). Индекс в массиве может быть только порядкового типа, т.е., чаще всего, целым числом.

Очень условно массив можно представить как состав поезда, в который включено некоторое количество вагонов. Как известно, каждый вагон имеет свой номер, и этот номер представляет собой целое число. Т.е., например, не может быть вагона 2,5 или 5,78. В каждом вагоне содержится определенный груз и его можно переместить туда из другого места или наоборот извлечь.

Массив, у которого адрес (индекс) элемента представлен одним числом, называется одномерным. Если элементами одномерного массива являются простой числовой тип данных, то такие массивы называют векторами. Различные примеры одномерных массивов представлены на рис. 5.1.

Для того чтобы в программе сделать объявление переменной типа *одномерный массив*, нужно в заголовке записать следующее:

*ИмяПеременной****: array [****от..до****] of*** *ТипЭлементов****;***

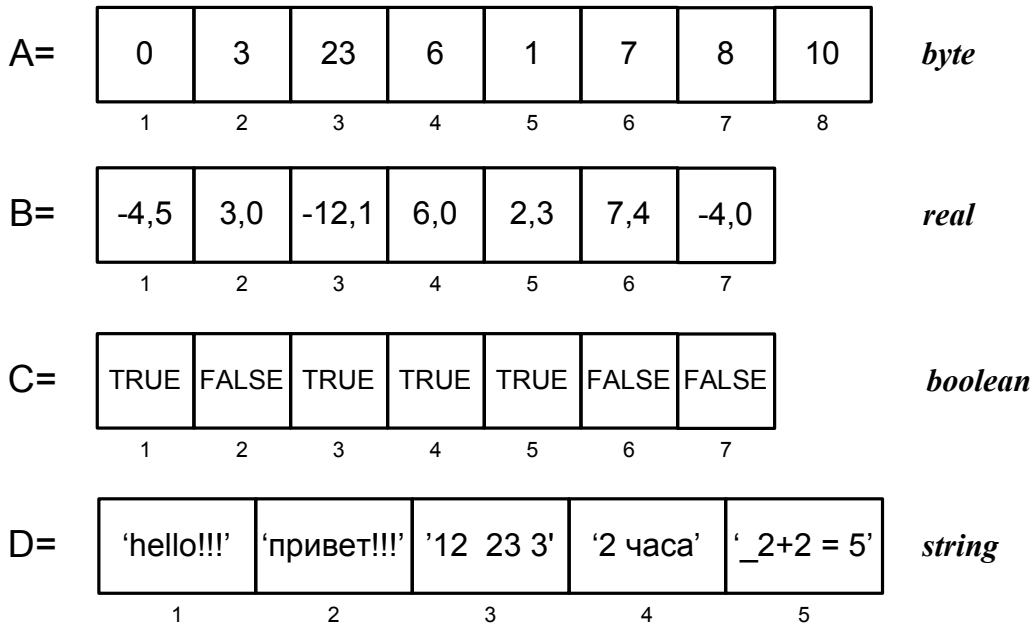


Рисунок 4.1 Примеры одномерных массивов разных типов. В ячейках записаны элементы, под каждой ячейкой записан индекс элемента в массиве.

Пример объявления массивов, изображенных на рис. 4.1 запишем ниже

```
Var
  A: array [1..8] of byte;
  B: array [1..7] of real;
  C: array [1..7] of boolean;
  D: array [1..5] of string;
```

Здесь представлены переменные типа «массив», которые имеют в своем составе ровно столько элементов, сколько заявлено на рисунке (рис. 4.1). Однако объявление переменной есть процесс выделения под нее памяти. Еще до начала работы программы необходимо знать, какого размера массив потребуется в процессе ее исполнения. К сожалению, не всегда заранее известно о точной его длине. Потому приходится использовать динамические массивы (которых пока касаться не будем), иные эвристические подходы или просто *избыточное выделение памяти*.

Избыточное выделение памяти есть действие, направленное на резервирование памяти достаточной для хранения наибольшего из

предполагаемых массивов в процессе выполнения программы. Так, например, если нужно объявить переменную, в которой будут храниться средние рейтинговые оценки академической группы студентов в 100-бальной шкале, то логичным будет длину массива ограничить 40-ка элементами, поскольку, практически не встречается групп, состоящих более чем из 40-ка человек. Хотя реально может использоваться, скажем, 25 из 40 ячеек. Остальные ячейки просто будут занимать память и не использоваться. В итоге программа становится более массовой, т.е. менее чувствительной к качеству входных данных. В программировании, как и в других отраслях деятельности человека, часто приходится идти на компромиссы.

Кроме прямого объявления можно использовать объявление через вспомогательный раздел ***type***. Приведем пример такого объявления для массивов изображенных на рис. 4.1.

```
Const
  LengthA = 40;
  LengthB = 50;
  LengthC = 120;
  LengthD = 40;
Type
  T1mByte  = array[1.. LengthA] of byte;
  T1mRe    = array[1.. LengthB] of real;
  T1mBool  = array[1.. LengthC] of boolean;
  T1mStr   = array[1.. LengthD] of string;
Var
  A: T1mByte;
  B: T1mRe;
  C: T1mBool;
  D: T1mStr;
```

Здесь объявление произведено не только с использованием раздела ***type***, но и с помощью раздела ***const***. Это позволяет оперативно вносить изменения в исходный текст программы, меняя размерность еще на этапе компиляции. Стоит отметить, что объявление в качестве границы диапазона массива переменной и последующее задание ее длины в тексте

программы является ошибочным, поскольку приводит к неопределенности размерности на этапе компиляции. Программе просто не сможет быть корректно выделено место в памяти и потому она даже не откомпилируется. В качестве границ массива могут выступать только константы. Как уже отмечалось выше, определение размерности в процессе работы программы называется динамическим массивом и имеет несколько иной способ объявления и инициализации. Более того, не все компиляторы *Pascal* поддерживают динамические массивы.

4.2 Поэлементная прямая обработка одномерных массивов

Как было сказано ранее, массив – переменная сложной структуры и потому не может быть подвергнута обработке целиком. Все действия с массивами следует проводить поэлементно, т. е. обращаясь непосредственно к каждой его ячейке. Типовой алгоритм последовательной обработки всех элементов очень прост и представлен на рис. 4.2.

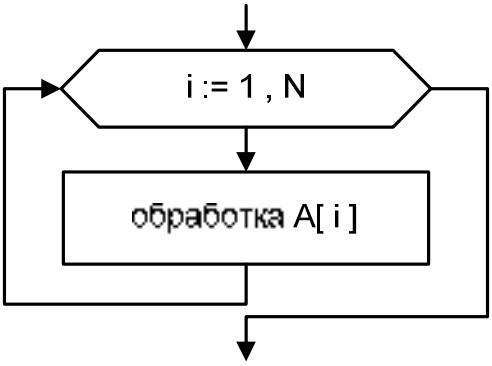


Рис. 4.2. Обработка каждого элемента массива

Естественно, что раз переменная типа *массив* была объявлена, то и в момент работы программы в области памяти, отведенной под него, существуют некоторые данные. Только до момента инициализации массива эти данные переставляют собой бессмысленную последовательность[19]. Для того чтобы появился смысл, массив нужно заполнить.

[19] В зависимости от версии компилятора *Pascal*, по умолчанию переменные могут заполняться нулями или нет. Вообще, для уверенности в корректности работы алгоритма всегда нужно инициализировать переменные вручную.

Процесс задания некоторой переменной первичного значения называется ***инициализацией***.

Рассмотрим инициализацию массива пользователем, т. е. такую реализацию программы, при которой все элементы массива вводятся вручную. Необходимая для использования размерность массива тоже вводится с клавиатуры во время работы программы. Блок-схема алгоритма ввода представлена на рис. 4.3, *а*.

```
writeLn('Введите количество элементов в массиве');
```

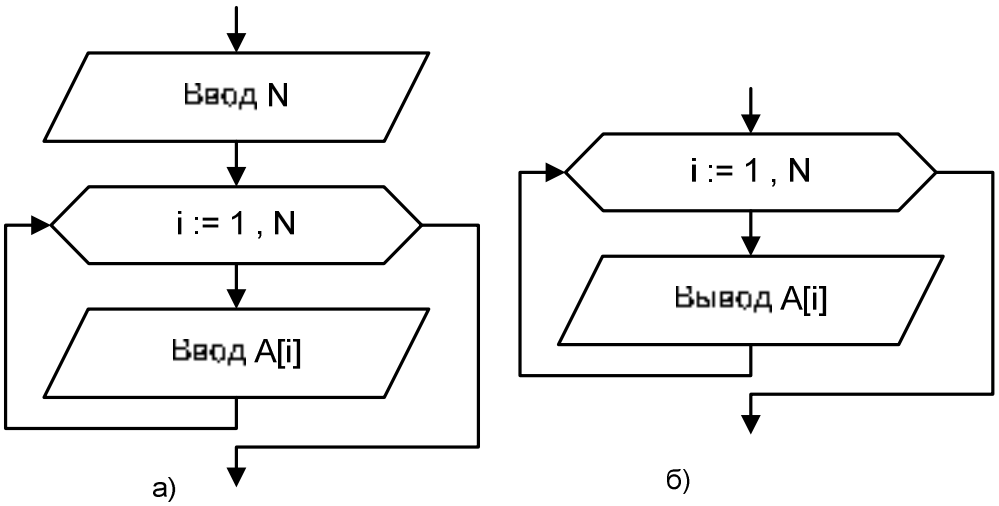


Рис. 4.3. Алгоритмы ввода и вывода одномерного массива

```
readLn(N);
for i:=1 to N do
  begin
    write('A[',i,']=');
    readLn(A[i]);
  end;
```

Для того чтобы вывести массив на экран можно воспользоваться следующим фрагментом программы (блок-схема алгоритма представлена на рис. 4.3, *б*):

```
writeLn('Массив A:');
for i:=1 to N do
write(A[i]:4)[20];
```

[20] Здесь в качестве формата вывода указано 4 знакоместа под каждый элемент. Это сделано с целью отделения элементов массива друг от друга. Если массив состоит из вещественных чисел, то следует форматировать их согласно правилам, например, так: Write(A[i]:7:2);

В качестве примера обработки всех элементов числового массива можно привести умножение/деление элементов на некоторое число или суммирование/вычитание элементов массива и некоторого числа.

ПРИМЕР

Умножить все элементы массива на 2.

```
…
for i:= 1 to N do
  A[i] := A[i]*2;
…
```

Типовыми можно назвать алгоритмы подсчета суммы и произведения элементов массива. Эти алгоритмы являются рекуррентными, т.е. каждая последующая итерация основывается на предыдущей. Отличие лишь в начальных значениях переменных (точках входа в итерацию). Накопление суммы начинают с нуля, а произведения с единицы.

Сумма:

```
…
S:=0;
for i:=1 to N do
  S:= S + A[i];
…
```

Произведение:

```
…
P:=1;
for i:=1 to N do
  P:=P*A[i];
…
```

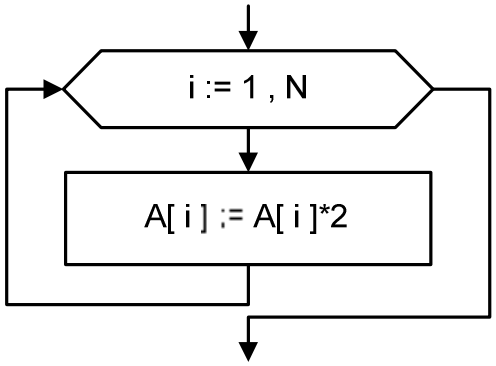


Рисунок 4.4 Умножение элементов массива на 2

Еще одним алгоритмом полной линейной обработки всех элементов массива является копирование его содержимого в новый.

Пусть, например, нужно скопировать элементы массива ***B*** в массив ***A***. Самый простой, как покажется, способ следующий: просто взять и присвоить один массив другому, т.е. ***A:=B***. Однако такой способ не всегда

приемлем, поскольку его суть сводится к *побитному* копированию одного объекта в другой. Это означает, что массивы ***A*** и ***B*** должны быть строго одного типа. Как вариант, у двух массивов могут просто не совпадать размеры. Более того, может не работать побитовое копирование даже в таком, на первый взгляд верном случае как описан далее. Пусть имеем следующую декларацию в разделе описания:

```
Type T1mass = array[1..50] of integer;
Var A: T1mass;
    B: array[1..50] of integer;
```

здесь операция ***A:=B*** является недопустимой с точки зрения синтаксиса языка *Pascal*, поскольку компилятор считает переменные ***A*** и ***B*** разнотипными несмотря на то, что они имеют одинаковую структуру с точностью до размера массива.

Правильной является запись:

```
Type T1mass = array[1..50] of integer;
Var A, B: T1mass;
```

Вот теперь четко видно, что обе переменные однотипны и потому могут быть побитно скопированы друг в друга. Это связано с тем, что *Pascal* является языком строгого контроля типов.

Следует отметить, что операция побитового копирования далеко не всегда является приемлемой, поскольку заставляет четко следить за типами переменных. Если речь идет о копировании более сложных объектов, то возможны трудноуловимые ошибки при работе программы.

Более предпочтительным является вариант поэлементного копирования. Что можно записать для двух одномерных массивов ***A*** и ***B*** длины ***N*** следующим образом:

```
For i:=1 to N do
  A[i] := B[i];
```

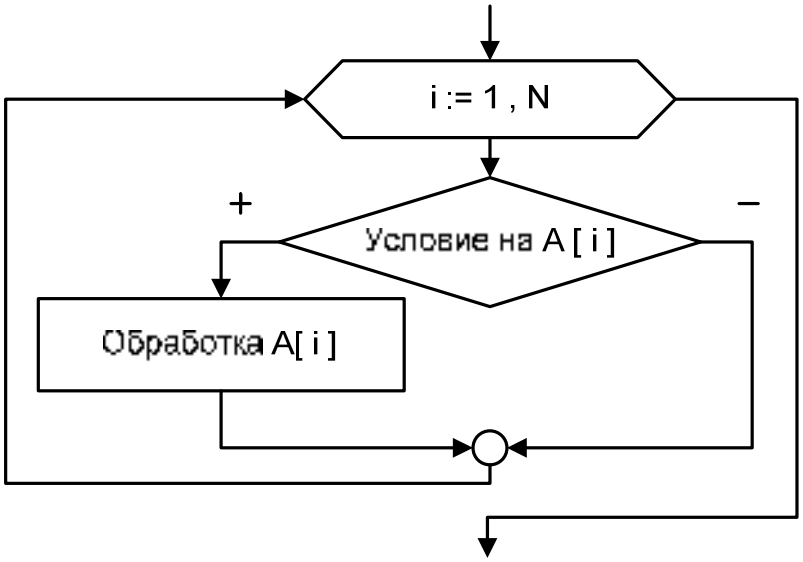


Рисунок 4.5. Обработка элементов массива, удовлетворяющих некоторому условию

4.3 . Элементы, удовлетворяющие некоторому условию

Зачастую в обработке массивов требуется обработать не все элементы, а лишь те, которые удовлетворяют некоторому условию. Для этого в тело цикла вставляют развилку с условием, накладываемым на элементы. Блок-схема таковой обработки представлена на рис. 4.5. Условие может быть каким угодно, например, положительность, равенство чему-либо, четность и т.д.

ПРИМЕР

Рассмотрим задачу *подсчета среднего арифметического четных элементов массива*. Решением будет являться следующий алгоритм, записанный в виде блок-схемы (рис. 4.6) и в виде программы:

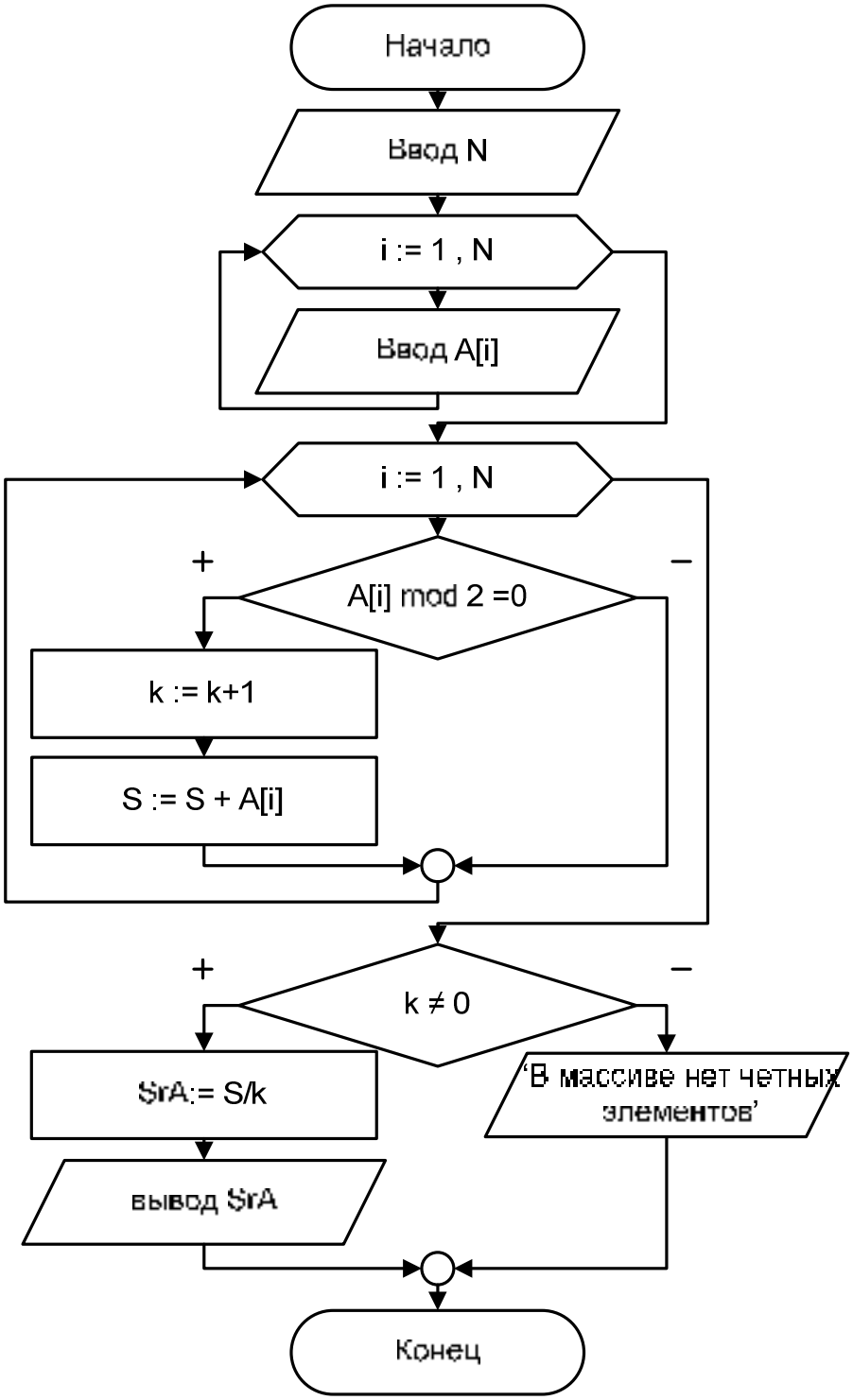


Рисунок 4.6. Подсчет среднего арифметического четных элементов массива

```
program massiv;
var A:array[1..100] of integer;
    i,k,N:byte;
    S:integer;
    SrA:real;
begin
writeLn('Введите количество элементов в массиве');
```

```
readLn(N);
for i:=1 to N do
  begin
    write('A[',i,']=');
    readLn(A[i]);
  end;
S:=0;
k:=0;
for i:=1 to N do
  if (A[i] mod 2) = 0 then
    begin
      S:=S+A[i];
      k:=k+1;
    end;
if k<>0 then
  begin
    SrA := S/k;
    writeLn('Среднее арифметическое: ', SrA:8:2);
  end
else
  writeLn('В массиве нет четных элементов');
end.
```

Здесь после подсчета суммы и количества делается проверка на существование четных элементов. Четные элементы существуют в том случае, если условие четности выполнилось хотя бы один раз. Это приведет к инкрементации переменной ***k***, служащей счетчиком четных элементов, на единицу. ***Инкрементация*** – процесс увеличения переменной. ***Декрементация***, соответственно, – процесс уменьшения значения переменной (от *англ. increase – возрастание*, *decrease – уменьшение*).

Использование счетчика ***k*** есть частный случай рекуррентного алгоритма подсчета суммы. Только в этом случае при каждой удачной итерации ***k*** увеличивается строго на единицу (счетчик «перещелкивается»).

В случае отсутствия в массиве четных элементов, переменная ***k*** останется равной нулю, что приведет к попытке деления на ноль. Для предотвращения этого в алгоритм после цикла вставлена развилка, которая

выводит значение среднего арифметического или сообщение о невозможности его подсчета.

ПРИМЕР

Аналогичным образом можно провести подсчет среднего геометрического модулей четных элементов. Только в этом случае будет использоваться рекуррентный алгоритм накопления произведения. При этом не забываем умножать на модуль значения найденного четного элемента. Найдем ***k*** – количество четных элементов массива, найдем ***P*** – произведение этих элементов. Далее, если ***k*** не равно нулю, вычислим ***SrG*** = $\sqrt[k]{P}$. Корень будем извлекать используя формулу возведения в степень с помощью функций доступных языку *Pascal*:

$$x^y = \exp(y \cdot \ln(x))$$

Не приводя блок-схемы, запишем фрагмент программы подсчитывающей среднее геометрическое модуля четных элементов одномерного массива:

```
…
P:=1;
k:=0;
for i:=1 to N do
  if (A[i] mod 2) = 0 then
    begin
      P:=P*abs(A[i]);
      k:=k+1;
    end;
if k<>0 then
  begin
    SrG := exp((1/k)*ln(P));
    writeLn('Среднее геометрическое: ', SrG:8:2);
  end
else
  writeLn('В массиве нет четных элементов');
…
```

Иногда по условию задачи требуется сформировать из заданного массива новый. Решим такую задачу:

ПРИМЕР

*Из положительных элементов массива **A*** сформировать массив ***B***, *а из отрицательных и кратных трем сформировать массив **C.***

Алгоритм достаточно прост (рис. 4.7):

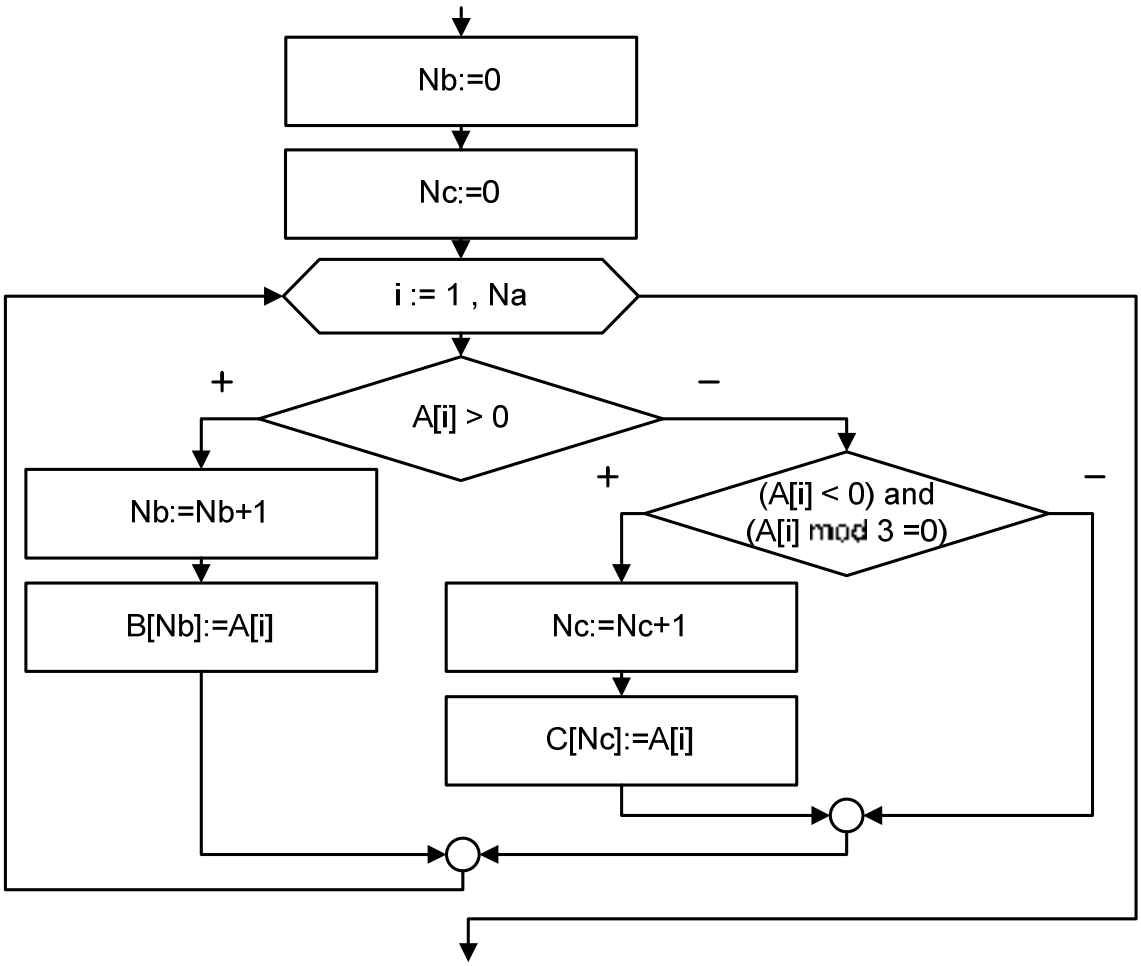


Рисунок 4.7 Формирование из одного массива пары новых

```
Nb:=0;
Nc:=0;
for i:=1 to Na do
  if A[i]>0 then
    begin
      Nb:=Nb+1;
      B[Nb]:=A[i];
    end
  else
    if (A[i]<0) and (A[i] mod 3 = 0 ) then
      begin
        Nc:=Nc+1;
        C[Nc] := A[i];
 end;
```

В блок-схеме, изображенной на рис. 4.7, внутри цикла записана полная развилка. На положительной ветви ведется формирование массива ***B***, а на отрицательной располагается еще одна развилка со сложным условием, в случае выполнения которого будет формироваться массив ***C***. В качестве переменных индексов массивов ***C*** и ***B*** используются переменные ***Nb*** и ***Nc***, которые после окончания цикла будут равны, соответственно, числу элементов массива ***B*** и ***C***.

Одними из самых важных алгоритмов поиска в массивах являются алгоритмы отыскания экстремальных элементов в них. Простейшими примерами экстремальных элементов являются максимальный и минимальный по значению.

Алгоритм поиска максимума и его места расположения (индекса)

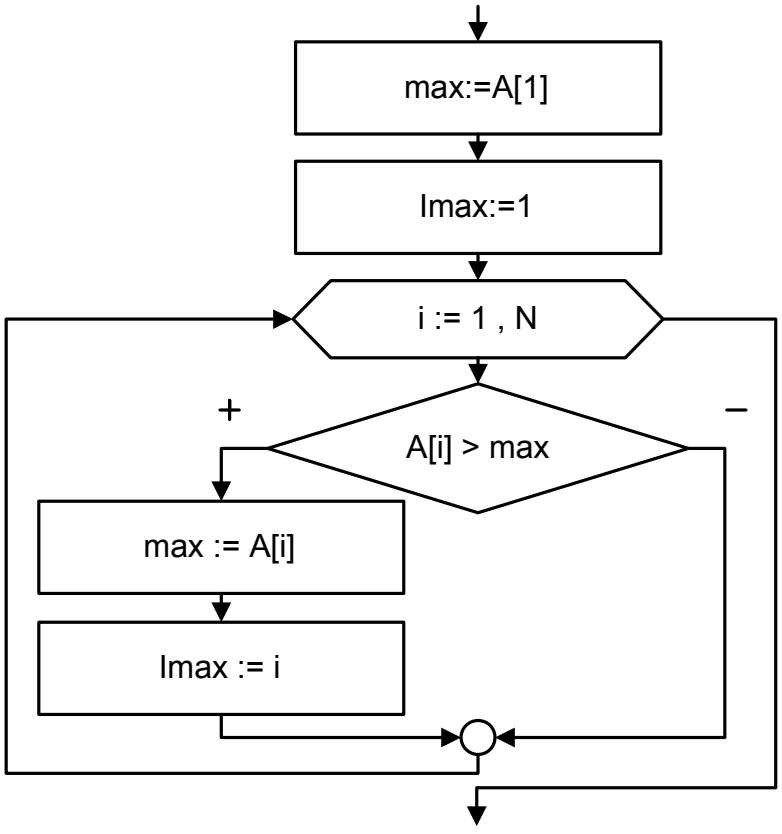


Рис. 4.8 Поиск максимума и его индекса

достаточно прост. Упрощенно его можно представить следующим образом: Пусть есть кучка камней и нужно найти наибольший среди них. Для этого берем в левую руку первый камень и считаем что он самый большой. Далее берем в правую руку следующий камень и сравниваем его с тем что находится в левой. Если камень в правой руке больше того, что в левой, то освобождаем левую руку и перекладываем в нее содержимое правой. Если ситуация обратная (в правой руке камень меньше чем в левой), то все оставляем без изменения. Правую руку освобождаем и вытаскиваем ею следующий камень для анализа. И так продолжает до тех пор, пока не переберем все камни в куче.

На языке алгоритма это будет выглядеть так (рис. 4.8):

```
…
max := A[1];
Imax := 1;
for i:=1 to N do
  if A[i]>max then
    begin
      max:= A[i];
      Imax:= i;
    end;
…
```

Алгоритм поиска минимума точно такой же, только знак «>» меняется на «<» и переменным даются имена, отражающие суть того, что ведется поиск минимального элемента (***min*** и ***Imin***).

Не всегда поиск элементов происходит по одному условию. Иногда этих условий несколько. Если их совокупность можно объединить операциями алгебры логики и просто заключить в один предикатный узел, то все достаточно просто. Однако, иногда не удается просто так включить сложное условие, поскольку может нарушаться свойство массовости алгоритма. Для этого поступают в каждом конкретном случае по-своему. Для примера рассмотрим достаточно типичную задачу такого характера:

ПРИМЕР

Найти наименьший элемент среди нечетных элементов массива.

Для этой задачи составим тестовый пример:

вход: $A =$

–8	–1	2	3	6	–7	4	–10	1	–5

выход: ***min*** = –7; ***Imin*** = 6

Если попытаться применить алгоритм, описанный ранее с добавлением в условие требования нечетности элемента, то результатом будет ***min***= –8, что явно неверно. Это связано с тем, что хотя и накладывается условие нечетности на элементы, оно не применится к точке входа (к первому элементу массива). Для корректной работы алгоритма требуется правильно задать первый элемент, принимаемый за минимум. Для этого его сначала надо найти. В качестве алгоритма решения задачи можно предложить такой (блок-схема на рис. 4.9):

```
Imin:=1;
while (not odd(A[Imin])) and (Imin<=N) do
  Imin:=Imin+1;
if Imin<=N then
begin
  min := A[Imin];
  for i:= Imin+1 to N do
    if (A[i]<min) and (odd(A[i]) then
    begin min:=A[i]; Imin:=i;
    end;
end
else
  writeLn('в массиве нет нечетных элементов');
```

Здесь при помощи цикла с предусловием сначала ищется первый нечетный элемент, после чего первый цикл прерывается и начинается второй с того места, где закончился предыдущий. А место это, как раз находится там, где встретился нечетный элемент массива. Второй цикл – обыкновенный алгоритм поиска минимума с дополнительным условием нечетности. Данный алгоритм предполагает наличие хоть одного нечетного элемента в составе массива.

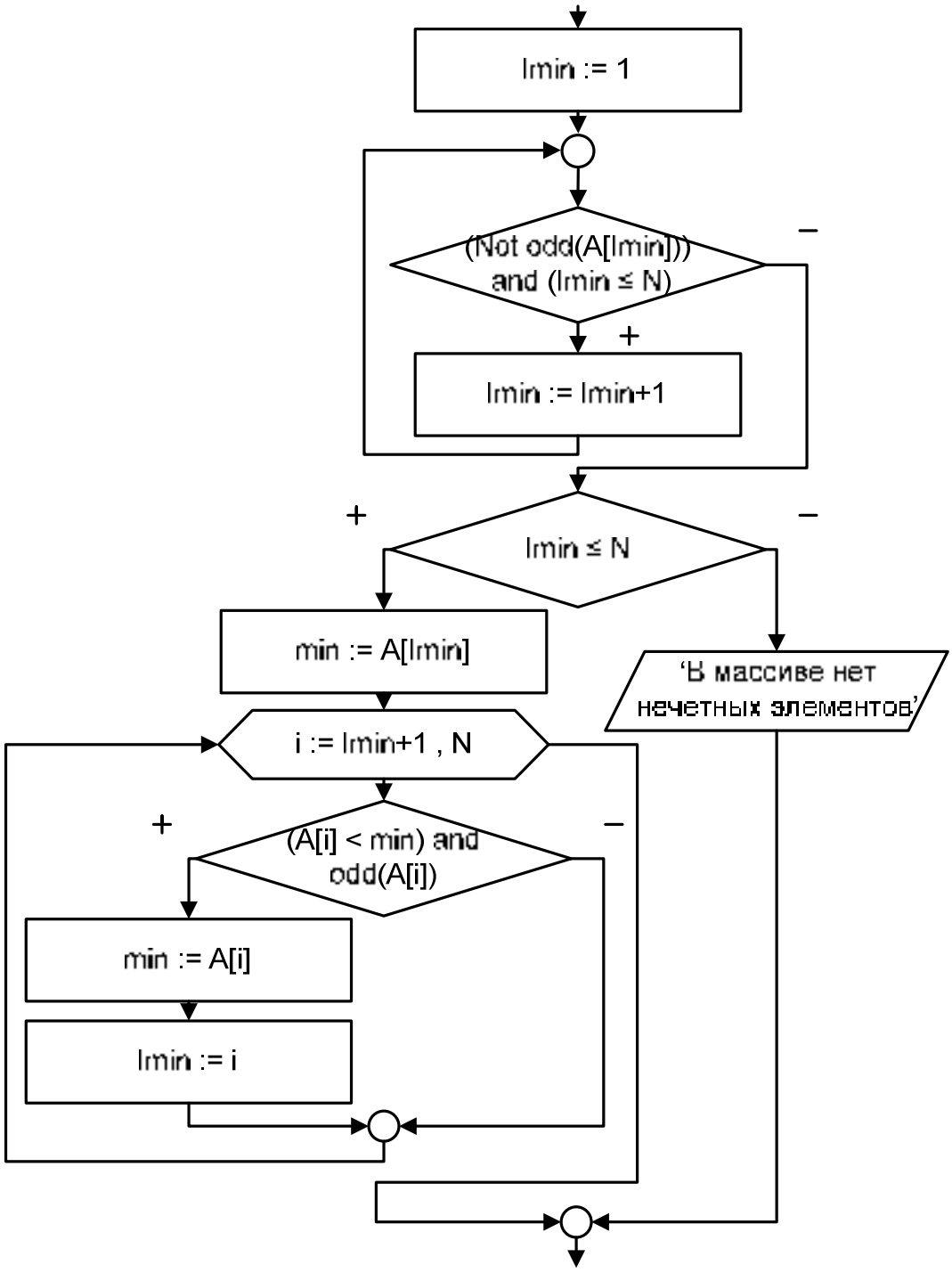


Рис. 4.9. Поиск минимального среди нечетных в два последовательных цикла

Для решения этой задачи можно предложить немного иной алгоритм, суть которого сводится к использованию всего одного цикла и переменной логического типа. Приведем решение задачи альтернативным способом целиком (рис. 4.10).

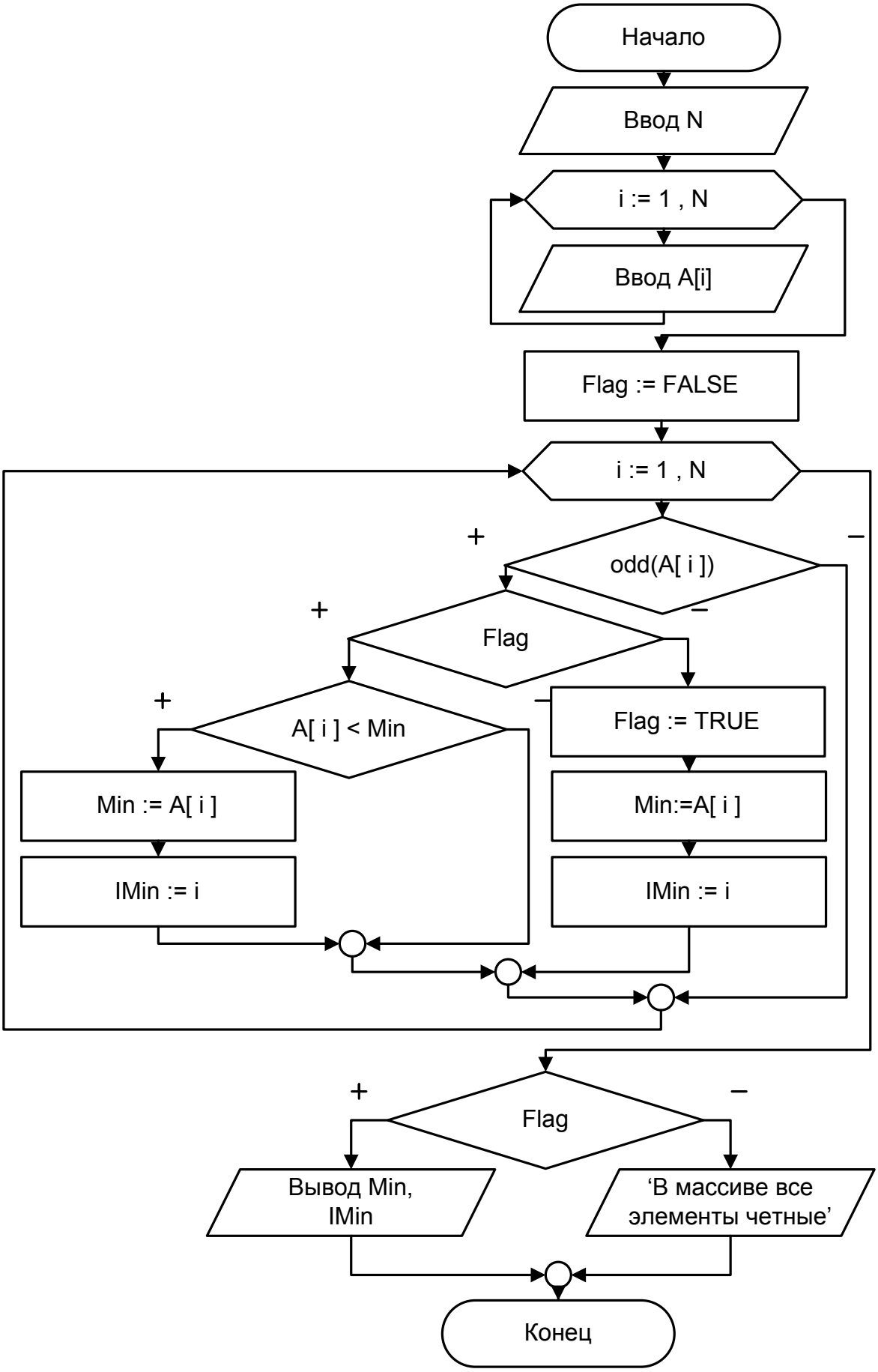


Рис. 4.10. Поиск минимального среди нечетных в один цикл

Программа будет такой:

```
program MinOdd;
var A:array[1..100] of integer;
    i,Imin,N:byte;
    Min:integer;
    Flag:boolean;
begin
writeLn('Введите количество элементов в массиве');
readLn(N);
for i:=1 to N do
  begin
    write('A[',i,']=');
    readLn(A[i]);
  end;
Flag:= false;
for i:=1 to N do
  if odd(A[i]) then
    if flag then
      begin
        if A[i]<Min then
          begin
            Min:=A[i];
            Imin:=i;
          end;
      end
    else
      begin
        Flag:=true;
        Min:=A[i];
        Imin:=i;
      end;
if Flag then
  writeLn('Min=A[',Imin,']=',Min)
else
  writeLn('В массиве все элементы четные');
end.
```

4.4. Обработка массивов по индексам

Достаточно часто критерием для обработки ячейки массива становится не ее содержимое, а месторасположение, т.е. индекс. Например, нужно взять и поменять местами последний элемент массива и средний элемент. Адрес последнего элемента – ***N***. А вот со средним не все так однозначно, ибо для массива нечетной длины средний элемент один и

его индекс ***N/2+0,5***, а для массива с четным количеством элементов серединой будут две ячейки: ***N/2*** и ***N/2+1***. Какую из них выбрать – зависит от условий задачи. Для простоты возьмем ***(N/2+0,5)***-ю ячейку.

Еще одной особенностью данной задачи является операция перестановки. Для обмена значениями разных ячеек требуется не потерять их. Для этого обычно используют третью переменную. Алгоритм подобен задаче обмена содержимым двух стаканов, скажем, с соком и с водой. Очевидно, что для исполнения задуманного нам потребуется третий, пустой стакан. Точно так же происходит обмен значениями произвольной пары переменных. Схема обмена представлена на рис. 4.11. В качестве еще одной иллюстрации описанной ситуации можно привести игру «пятнашки», в которой для возможности перемещения фишек предусмотрено пустое поле.

Итак, перестановка последней ячейки массива и его среднего

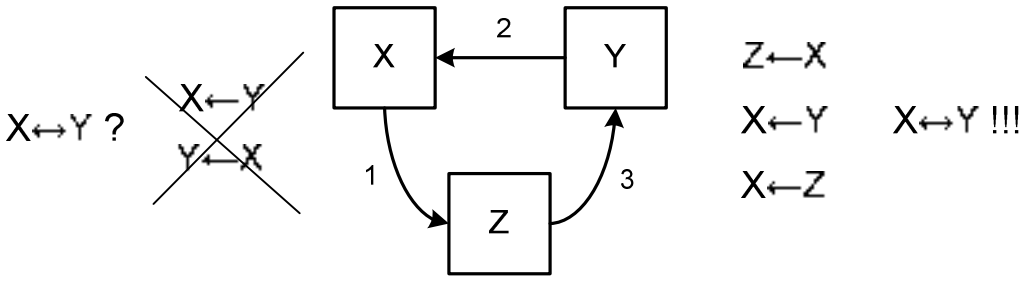


Рис. 4.11. Обмен значениями пары переменных в три действия

элемента может быть осуществлена следующим образом:

```
X := A[N];
A[N] := A[trunk((N+1)/2)];
A[trunc((N+1)/2)] := A[N];
```

Здесь применена функция ***trunc***, функция взятия целого числа от ***N/2*** по двум причинам. Во-первых, массив может быть нечетной длины, и во-вторых, индекс массива всегда целое число, а операция деления «/» возвращает результат типа ***real***.

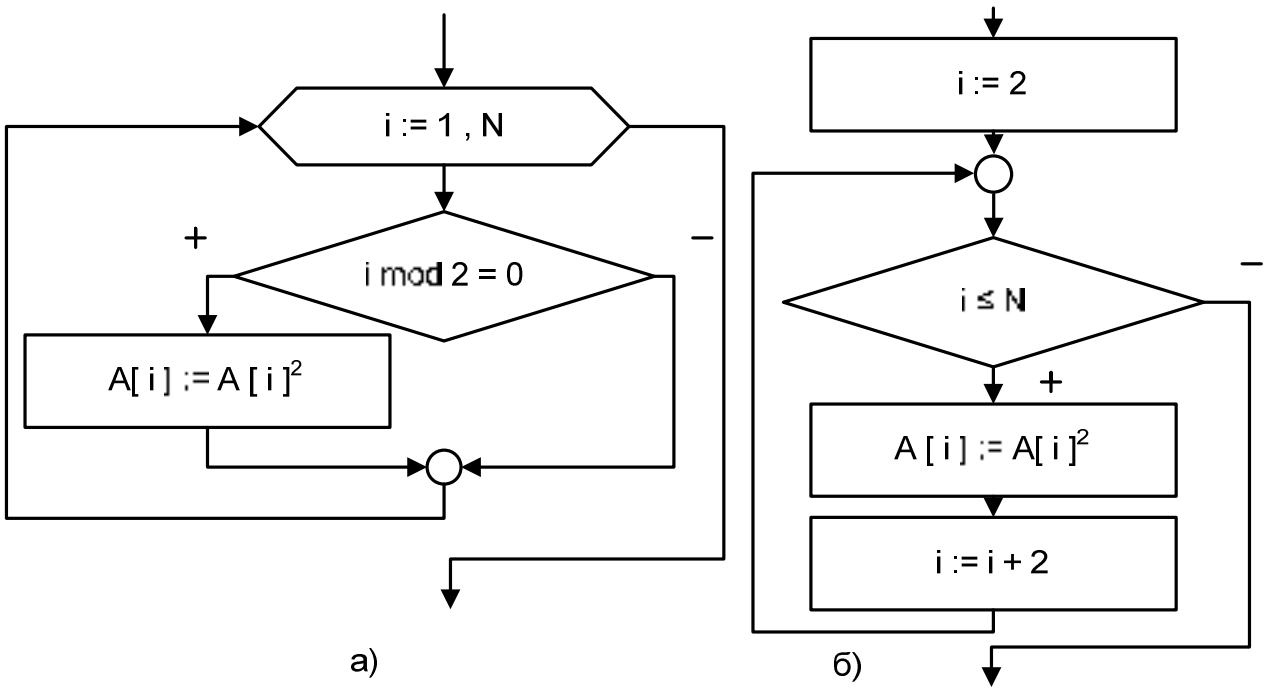


Рис. 4.12. Перебор всех индексов (*а*) и непосредственное вычисление (*б*)

В качестве еще одного примера обработки элементов массива стоящих на определенных местах можно привести алгоритм возведения в квадрат каждого второго элемента. Это можно сделать, как минимум, двумя способами. Первый способ предполагает перебор всех индексов массива и их анализ, а второй есть процесс прямого вычисления адреса интересующего элемента.

Переборный вариант таков (рис. 4.12 а):

```
for i:=1 to N do
  if i mod 2 = 0 then
    A[i] := sqr(A[i]);
```

Метод прямого вычисления адреса позволяет сократить число итераций, вдвое. Для этого логично воспользоваться циклом с предусловием (рис. 4.12 б)).

```
i:=2;
while i<=N do
  begin
    A[i]:= sqr(A[i]);
    i:= i+2;
  end;
```

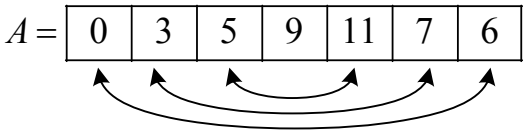
Теперь разберем некоторые более сложные алгоритмы перестановок в одномерном массиве.

ПРИМЕР

Переставить элементы массива в обратном порядке.

Тестовый пример для этой задачи выглядит следующим образом:

вход:



выход: $A =$ | 6 | 7 | 11 | 9 | 5 | 3 | 0 |

Можно просто взять исходный массив, и скопировать его во вспомогательный, а потом прочитать вспомогательный в исходный в обратном порядке. Но этот путь не является правильным, поскольку зря расходует память, ведь массивы занимают на порядки больше места, чем переменные простых типов.

Поступим иначе. Будем читать массив одновременно с двух сторон, двигаясь к его центру, в процессе движения крайние элементы будем обменивать местами (если не остановимся на центре, а продолжим движение от одного края до другого, то, фактически, перестановка каждого элемента произойдет дважды и массив на выходе опять примет вид массива поданного на вход алгоритма).

Реализация перестановки элементов такова (рис. 4.13 а)):

```
for i:=1 to trunc(N/2) do
  begin
    buf := A[i];
    A[i] := A[N-i+1];
    A[N-i+1] := buf;
  end;
```

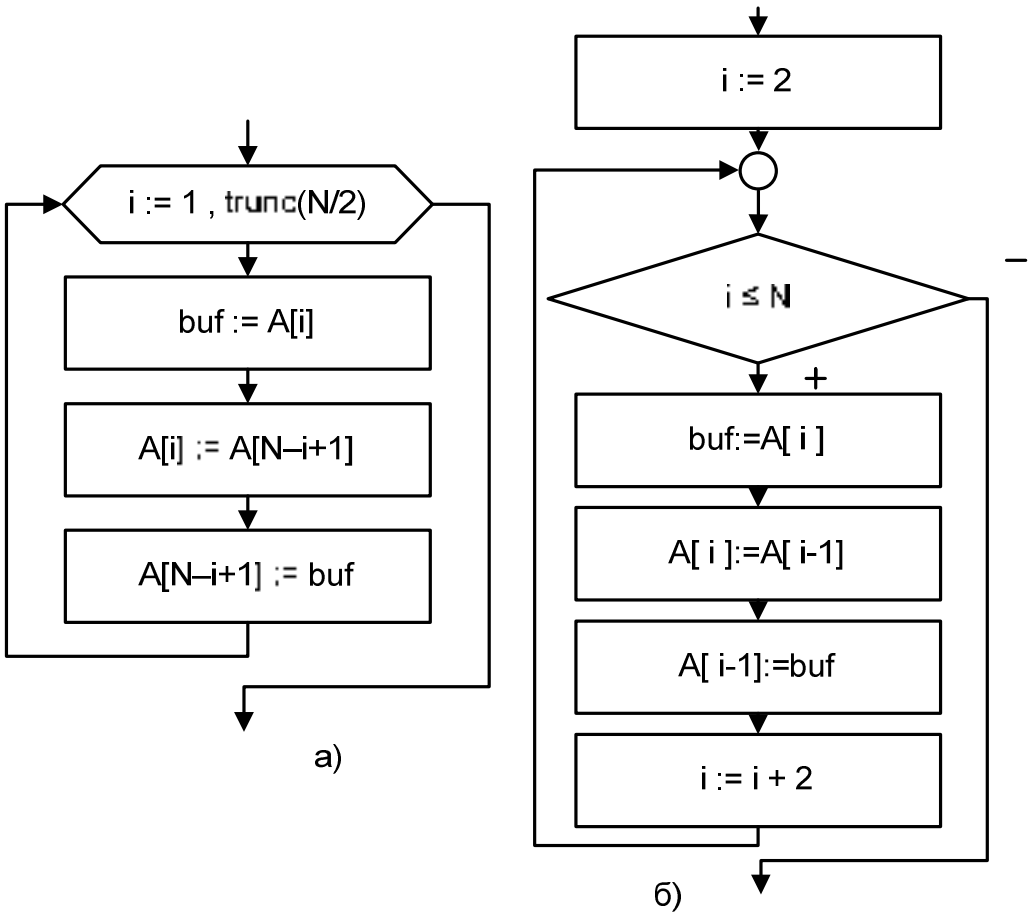


Рис. 4.13. Инверсия массива (*а*) и перестановка соседних элементов (*б*)

Все очень просто. Для движения в прямом направлении используем индекс ***i***, а для обратного прохода индекс вычисляется по формуле ***N–i+1***. Действительно, проведем *трассировку*[21] алгоритма для описанного выше тестового примера и посмотрим, как ведут себя индексы:

```
первая итерация: i=1, A[1]↔A[7] (7–1+1 = 7)
вторая итерация: i=2, A[2]↔A[6] (7–2+1 = 6)
третья итерация: i=3, A[3]↔A[5] (7–3+1 = 5)
```

Приведем еще один алгоритм парной перестановки элементов массива.

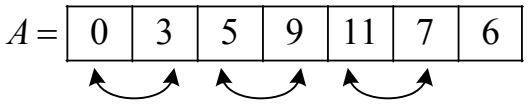
ПРИМЕР

Поменять местами соседние элементы массива.

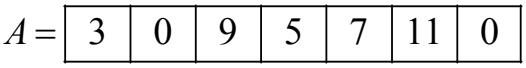
[21] Трассировка есть процесс записи значений переменных на каждом шаге работы программы.

Вот что требуется сделать:

вход:



выход:



Для решения задачи воспользуемся циклом с предусловием и алгоритмом обмена в три действия, при этом на каждой итерации меняя текущий индекс ячейки на 2. Вот такой алгоритм получается в результате (рис. 4.13, *б*):

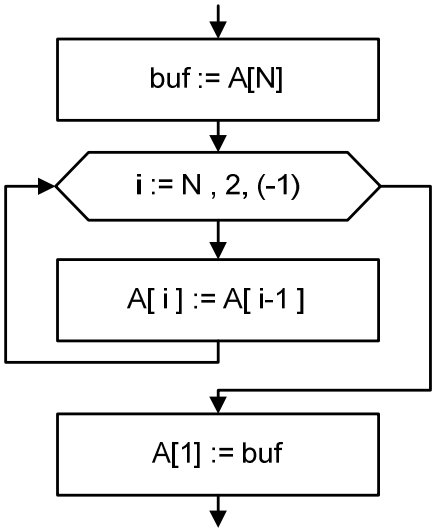


Рис. 4.14. Циклический сдвиг вправо

```
i:=2;
while i<=N do
  begin
    buf:=A[i];
    A[i]:=A[i-1];
    A[i-1]:=buf;
    i:=i+2;
  end;
```

ПРИМЕР

Произвести единичный *циклический сдвиг* элементов массива вправо. Под циклическим сдвигом понимается изменение положения каждого элемента на одну позицию (в данном случае вправо). Соответственно,

последний элемент окажется за пределами массива, а на месте первого образуется вакансия. При циклическом сдвиге будет происходить перемещение содержимого последней ячейки в первую. Это можно легко понять, если представить массив в виде ленты транспортира.

Тестовый пример таков:

Вход:

Выход:

$A =$

6	0	3	5	9	11	7

Алгоритм этого процесса следующий (рис. 4.14):

```
buf := A[N];
for i:=N downTo 2 do
    A[i] := A[i-1];
A[1] := buf;
```

Стоит обратить внимание на факт использования вспомогательной буферной переменной для хранения элемента, который оказался вытесненным. Эффективно организовать алгоритм получается если двигаться справа налево. Для этого цикл ***for*** пускается в обратную сторону. Если использовать прямой проход, то получится, что для корректной работы алгоритма потребуется не одна, а пара вспомогательных буферных переменных, да и число перестановок внутри цикла возрастет. Прямым проходом следует пользоваться при реализации *циклического сдвига влево*.

Еще одним типом достаточно распространенных задач являются такие, у которых при решении требуется найти тот или иной индекс.

ПРИМЕР

Найти третий положительный элемент массива. Индекс третьего положительного будет храниться в переменной ***Ik3pol*** (рис. 4.15).

```
k:=0;
Ik3pol:=0;
for i:=1 to N do
  if A[i]>0 then
    begin
      inc(k);
      if k=3 then
        Ik3pol:=i;
    end;
```

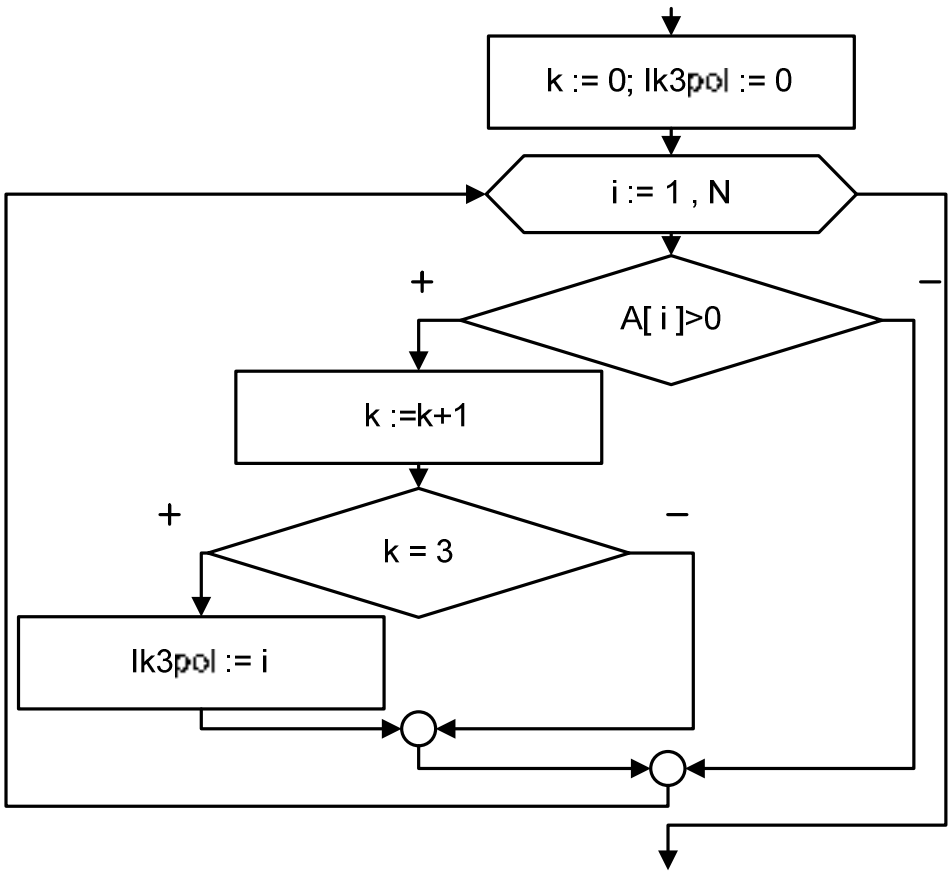


Рис. 4.15. Поиск третьего положительного

Если в массиве меньше чем три положительных элемента, то переменная хранящая индекс третьего положительного элемента ***Ik3pol*** останется равной нулю.

Или, вот такая задача:

ПРИМЕР

Найти первый отрицательный элемент массива.

Для ее решения можно воспользоваться алгоритмом предыдущей задачи, а можно немного упростить последовательность действий (на времени работы алгоритма на массивах малой длины это упрощение практически не отразится). Для этого воспользуемся тем свойством, что первый с начала отрицательный элемент является последним отрицательным с конца. Для использования этого свойства достаточно пустить цикл в обратном порядке, в результате получим последовательное

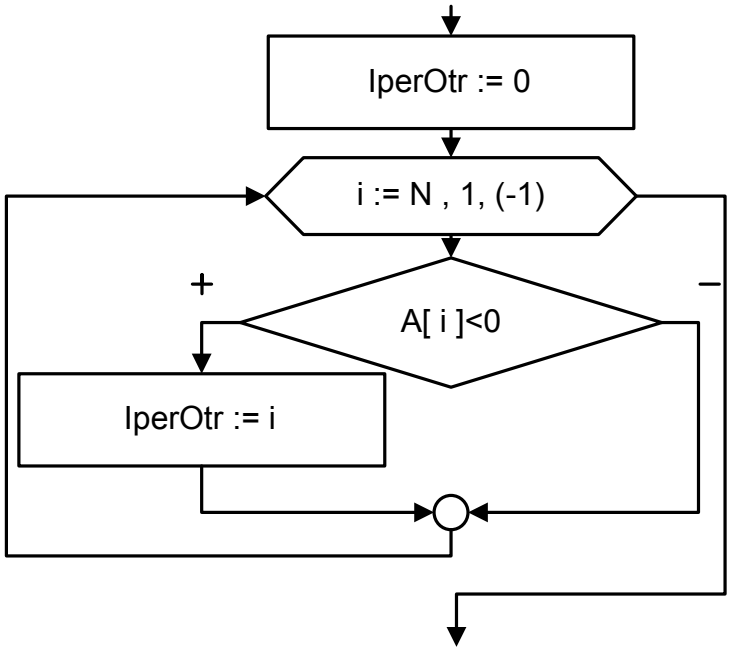


Рис. 4.16. Поиск первого отрицательного элемента

изменение переменной ***IperOtr*** хранящей интересующий нас индекс при каждой встрече отрицательного элемента. Последний раз такое изменение как раз произойдет на первом с начала элементе. Вот этот алгоритм (рис. 4.16):

```
IperOtr:=0;
for i:=N downTo 1 do
  if A[i]<0 then
    IperOtr:=i;
```

Если в массиве все элементы положительные, то переменная ***IperOtr*** останется равной нулю.

Можно рассмотреть целиком еще одну задачу, которая использует некоторые из вышеописанных алгоритмов.

ПРИМЕР

В одномерном массиве переставить в обратном порядке элементы заключенные между максимумом минимумом.

Решим задачу, воспользовавшись пошаговой детализацией алгоритма (рис. 4.17, *а*).

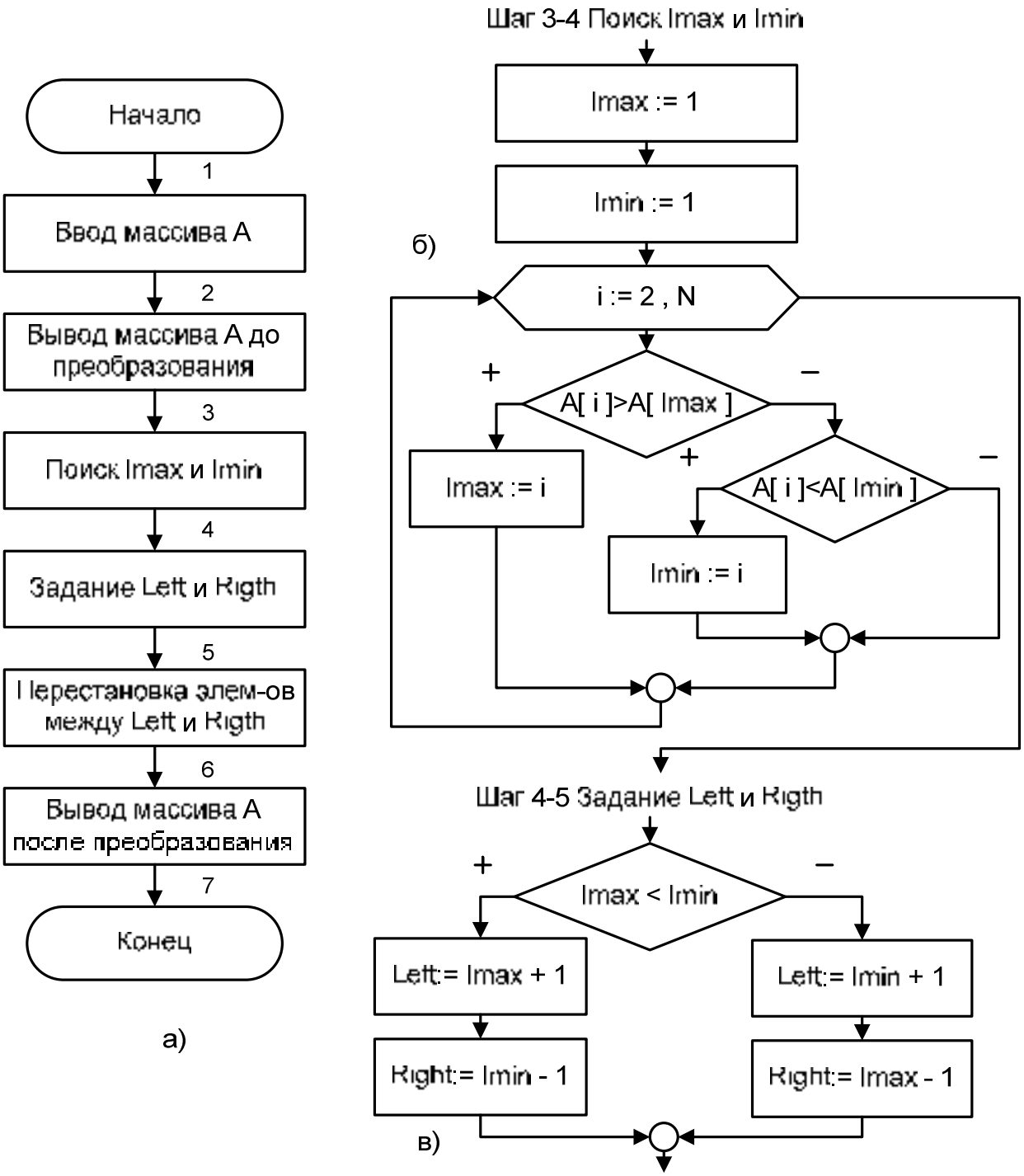


Рис. 4.17. Перестановка в обратном порядке элементов расположенных между максимумом и минимумом: *а* – общий алгоритм; *б* – поиск Imin и Imax; *с* – задание левой и правой границ

Тестовый пример к этой задаче может быть, например, таким:

вход:	2	5	10	2	4	5	7	9	1	3	0	2	7

Max=10, ***Imax***=3, ***Min***=0, ***Imin***=11, ***Left***=4, ***Right***=10.

выход:	2	5	10	3	1	9	7	5	4	2	0	2	7

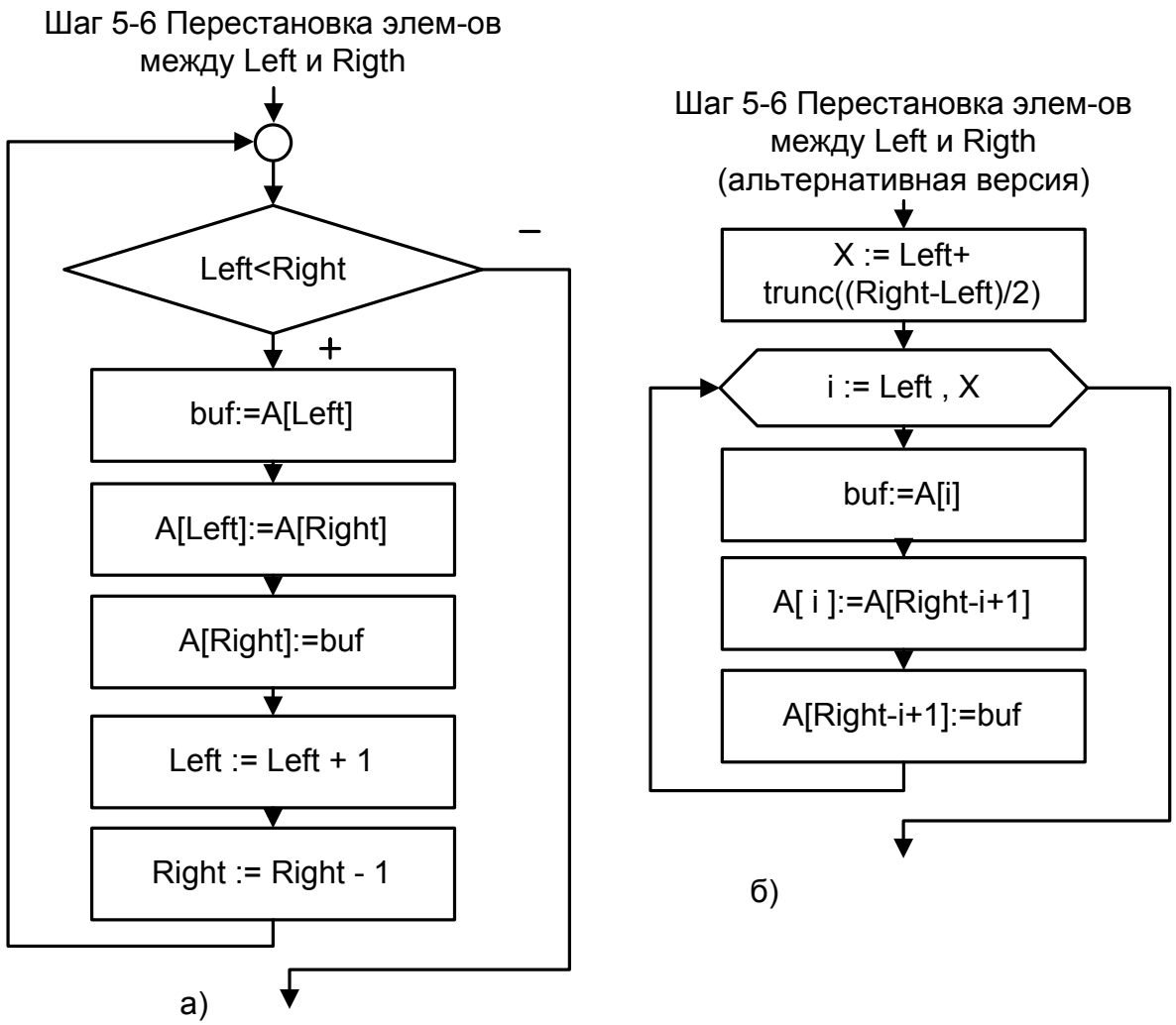


Рис. 4.18. Перестановка в обратном порядке элементов между Left и Right:
а – с изменением границ и *б* – непосредственно вычисляя граничные индексы

Чтобы решить эту задачу потребуется сразу после ввода массива найти положение максимума и минимума ***Imax*** и ***Imin*** (рис. 4.17, *б*). Сам же ввод (шаг 1-2) и вывод (как исходного (шаг 2-3), так и преобразованного (шаг 6-7)) здесь не расписывается, поскольку это стандартные алгоритмы и их блок-схемы изображены, например, на рис. 4.3.

Зная положение максимума и минимума теперь нужно определить, что из них встречается раньше для того, чтобы корректно задать границы изменения переменной цикла. Левая граница получила название ***Left***, а правая ***Right*** (рис. 4.17, *в*).

Далее следует сам алгоритм перестановки (шаг 5-6). Здесь можно воспользоваться перестановкой рассмотренной ранее, а можно рассмотреть несколько иную последовательность действий. Суть модифицированного алгоритма заключается в том, что не нужно следить за корректностью формул правой (***Rigth***) и левой (***Left***) границы диапазона. Нужно правильно задать начальные значения этим границам. Далее на каждой итерации правый индекс будет декрементироваться, а левый инкрементироваться. Продолжаться это будет до тех пор, пока значения ***Left*** и ***Right*** не пересекутся (рис. 4.18, *а*). Альтернативный алгоритм тоже приводим, в нем нужно внимательно следить за индексами (рис. 4.18, *б*).

Вот такая получилась программная реализация:

```
program MinMaxInv;
var A:array[1..100] of integer;
    i,N,Imin,Imax,Left,Right,k,j:byte;
    buf:integer;
begin
  writeLn('Введите количество элементов в массиве');
  readLn(N);
  for i:=1 to N do
    begin
      write('A[',i,']=');
      readLn(A[i]);
    end;
  writeLn('Вывод массива до преобразования:');
  for i:=1 to N do
    write(A[i]:4);
  writeLn;
  Imax:=1;
  Imin:=1;
  for i:=2 to N do
    begin
      if A[i]>A[Imax] then
        Imax:=i
      else
        if A[i]<A[Imin] then
          Imin:=i;
    end;
  if Imax<Imin then
    begin
      Left:=Imax+1;
      Right:=Imin-1;
```

```
    end
  else
    begin
      Left:=Imin+1;
      Right:=Imax-1;
    end;
  while Left<Right do
    begin
      buf:=A[Left];
      A[Left]:=A[Right];
      A[Right]:=buf;
      inc(Left);
      dec(Right);
    end;
  writeLn('Вывод массива после преобразования:');
  for i:=1 to N do
    write(A[i]:4);
end.
```

4.5. Алгоритмы с использованием вложенных циклов

Достаточно часто используются алгоритмы, для которых одного прохода по массиву недостаточно. Такие алгоритмы уже рассматривались ранее. Однако, есть более сложные последовательности действий, в которых для каждого прохода требуется свой проход. В этом случае возникает вложенный цикл. Алгоритмы, использующие вложенные циклы достаточно сложны, но в тоже время, отличаются важностью. Рассмотрим наиболее распространенные из таковых.

Наиболее часто встречающейся задачей требующей использования вложенных циклов является задача *упорядочивания* или *сортировки*. Так, если дан, например, массив состоящий из фамилий студентов, то логично их расположить в алфавитном порядке для удобства дальнейшего поиска. Такое упорядочивание будет называться алфавитным.

Рассмотрим, как произвести сортировку числового массива. Вообще все сортировки можно свести к числовым. В случае с алфавитной ее

разновидностью это легко сделать, если вспомнить, что в алфавите каждая буква имеет свой порядковый номер.

Итак, у нас есть массив произвольно заполненный числами. Требуется содержимое массива упорядочить по возрастанию, т.е. от меньшего к большему.

Одним из самых простых методов сортировки является сортировка методом линейного поиска. Для этого просматриваем массив, находим в нем максимальный элемент, запоминаем его позицию и отправляем найденный максимум в конец массива. Значение элемента с конца направляется на место максимума. Далее организуем еще один проход по массиву, но уже последний элемент не рассматриваем, т.к. он стал на свое место. Алгоритм поиска максимума повторяем, но теперь будет произведен обмен с предпоследней ячейкой. После второго прохода – уже два элемента на своих местах: последний и предпоследний. И так далее повторяем алгоритм, пока не достигнем начала массива.

Иллюстрация описанного алгоритма представлена ниже. Здесь переменная ***k*** обозначает номер прохода. Подчеркнуты числа, которые подвергаются обмену на текущем проходе. Элементы, которые не участвуют в текущем проходе, выделены вертикальными линиями.

```
0    4    5    9    1    7    6

0    4    5    6    1    7    9    k=1
0    4    5    6    1    7   |9|   k=2
0    4    5    1    6   |7|  |9|   k=3
0    4    1    5   |6|  |7|  |9|   k=4
0    1    4   |5|  |6|  |7|  |9|   k=5
0    1   |4|  |5|  |6|  |7|  |9|   k=6
```

Описанный алгоритм представлен на (рис. 4.19):

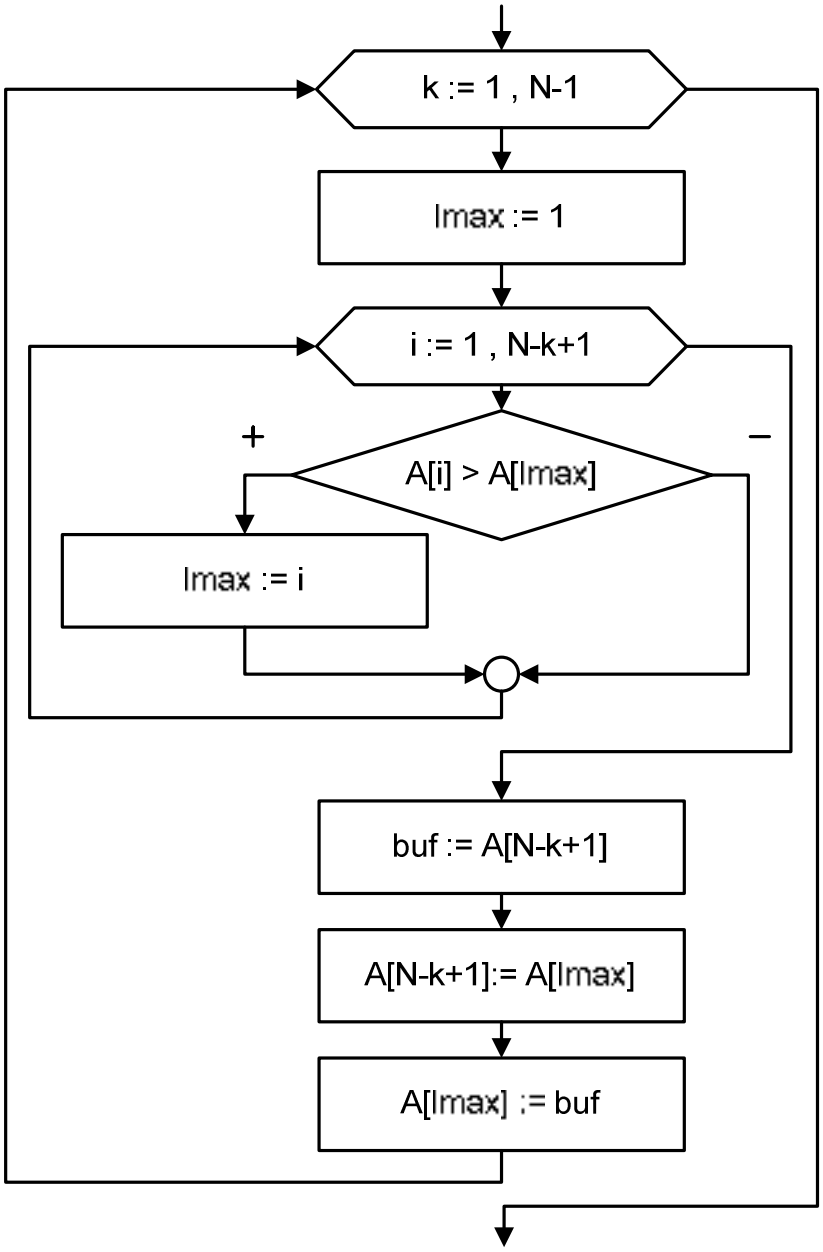


Рис. 4.19. Сортировка методом линейного поиска

```
for k:=1 to N-1 do
  begin
   Imax:=1;
   for i:=1 to N-k+1 do
      if A[i]>A[Imax] then
         Imax  := i;

    buf := A[N-k+1];
    A[N-k+1]:= A[Imax];
    A[Imax] := buf;
  end;
```

Еще одним достаточно простым методом сортировки является сортировка пузырьковым методом. Называется метод так потому, что на каждом проходе, при движении двигаясь вдоль массива, берем самый большой встретившийся элемент и далее двигаем его к концу массива пока не встретим элемент еще больше. Далее продолжаем движение уже с этим элементом. И так до тех пор, пока весь массив не будет пройден. Этот наибольший элемент можно представить пузырьком в стакане воды, который медленно поднимается на поверхность.

```
0   4   5   9   1   7   6   i=1 k=6
0   4   5   9   1   7   6   i=2 k=6
0   4   5   9   1   7   6   i=3 k=6
0   4   5   1   9   7   6   i=4 k=6
0   4   5   1   7   9   6   i=5 k=6
0   4   5   1   7   6   9   i=6 k=6

0   4   5   1   7   6  |9|  i=1 k=5
0   4   5   1   7   6  |9|  i=2 k=5
0   4   1   5   7   6  |9|  i=3 k=5
0   4   1   5   7   6  |9|  i=4 k=5
0   4   1   5   6   7  |9|  i=5 k=5

0   4   1   5   6  |7| |9|  i=1 k=4
0   1   4   5   6  |7| |9|  i=2 k=4
0   1   4   5   6  |7| |9|  i=3 k=4
0   1   4   5   6  |7| |9|  i=4 k=4

0   1   4   5  |6| |7| |9|  i=1 k=3
0   1   4   5  |6| |7| |9|  i=2 k=3
0   1   4   5  |6| |7| |9|  i=3 k=3

0   1   4  |5| |6| |7| |9|  i=1 k=2
0   1   4  |5| |6| |7| |9|  i=2 k=2

0   1  |4| |5| |6| |7| |9|  i=1 k=1
```

На каждом проходе анализируем соседние элементы. Если для них выясняется, что они не на своем месте, то меняем их местами. После проверки всех соседних паросочетаний переходим к следующему проходу.

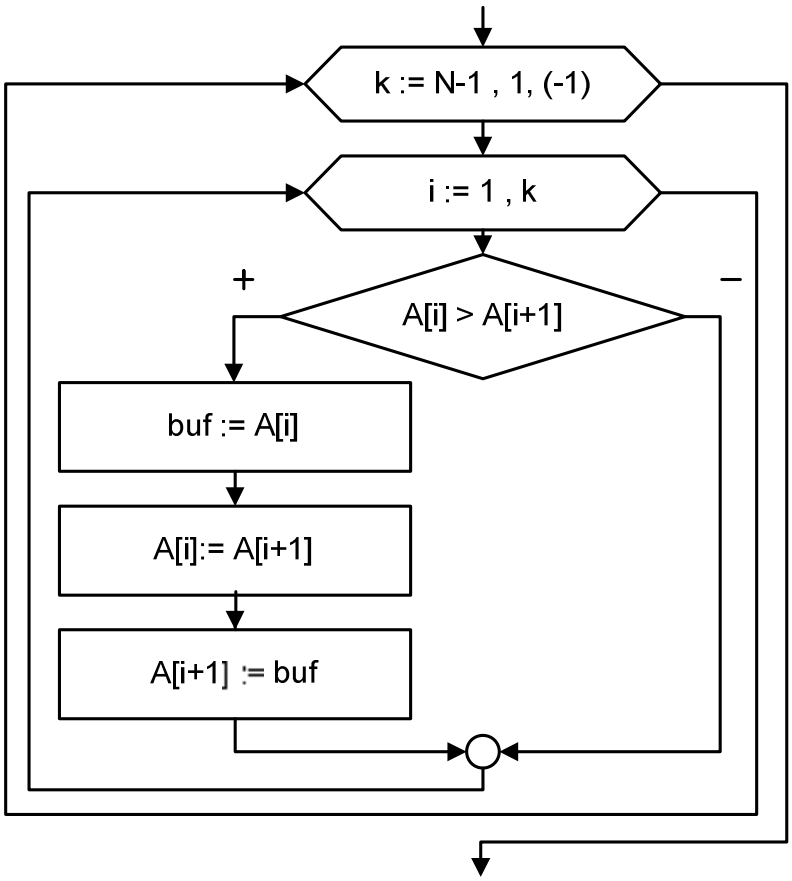


Рис. 4.20. Сортировка пузырьковым методом

Но последний элемент уже не участвует в сортировке, поскольку он уже на своем месте. И так продолжаем до тех пор, пока не достигнем начала массива. Иллюстрация описанного алгоритма представлена ниже. Подчеркнуты те ячейки массива, которые подвергаются анализу в текущий момент. ***k*** – количество проходов до окончания сортировки. ***i*** – номер анализируемой пары элементов.

Данный алгоритм представлен на рис. 4.20:

```
for k:=N-1 downto 1 do
  begin
    for i:=1 to k do
       if A[i]>A[i+1] then
          begin
            buf   := A[i];
            A[i]  := A[i+1];
            A[i+1]:= buf;
          end;
  end;
```

Как видно из примера, разобранного для пузырькового метода, массив оказывается отсортированным значительно раньше, чем закончатся все проходы. Уже на третьем проходе весь массив упорядочен. Оставшиеся три прохода идут впустую. Даже если массив будет изначально упорядочен, число проходов не изменится. Чтобы учесть высказанные замечания, можно модифицировать пузырьковый метод. Для этого внешний цикл сделаем циклом с постусловием, а внутрь него поместим логическую переменную ***sort***, которая будет принимать при каждой внешней итерации истинное значение. Однако, если массив не отсортирован на данной итерации, то она примет ложное значение. Как только условие упорядоченности соседних элементов выполнится для всех ячеек массива, сортировка прекратится.

Иллюстрация этого процесса представлена ниже.

```
 0    4    5    9    1    7    6    i=1 k=6
0    4    5    9    1    7    6    i=2 k=6
0    4    5    9    1    7    6    i=3 k=6
0    4    5    1    9    7    6    i=4 k=6
0    4    5    1    7    9    6    i=5 k=6
0    4    5    1    7    6    9    i=6 k=6
sort = false

0    4    5    1    7    6   |9|   i=1 k=5
0    4    5    1    7    6   |9|   i=2 k=5
0    4    1    5    7    6   |9|   i=3 k=5
0    4    1    5    7    6   |9|   i=4 k=5
0    4    1    5    6    7   |9|   i=5 k=5
sort = false

0    4    1    5    6   |7|  |9|   i=1 k=4
0    1    4    5    6   |7|  |9|   i=2 k=4
0    1    4    5    6   |7|  |9|   i=3 k=4
0    1    4    5    6   |7|  |9|   i=4 k=4
sort = false

0    1    4    5   |6|  |7|  |9|   i=1 k=3
0    1    4    5   |6|  |7|  |9|   i=2 k=3
```

```
0    1    4    5    |6|   |7|   |9|    i=3 k=3
sort = true
```

Описанный алгоритм записывается таким образом (рис. 4.21):

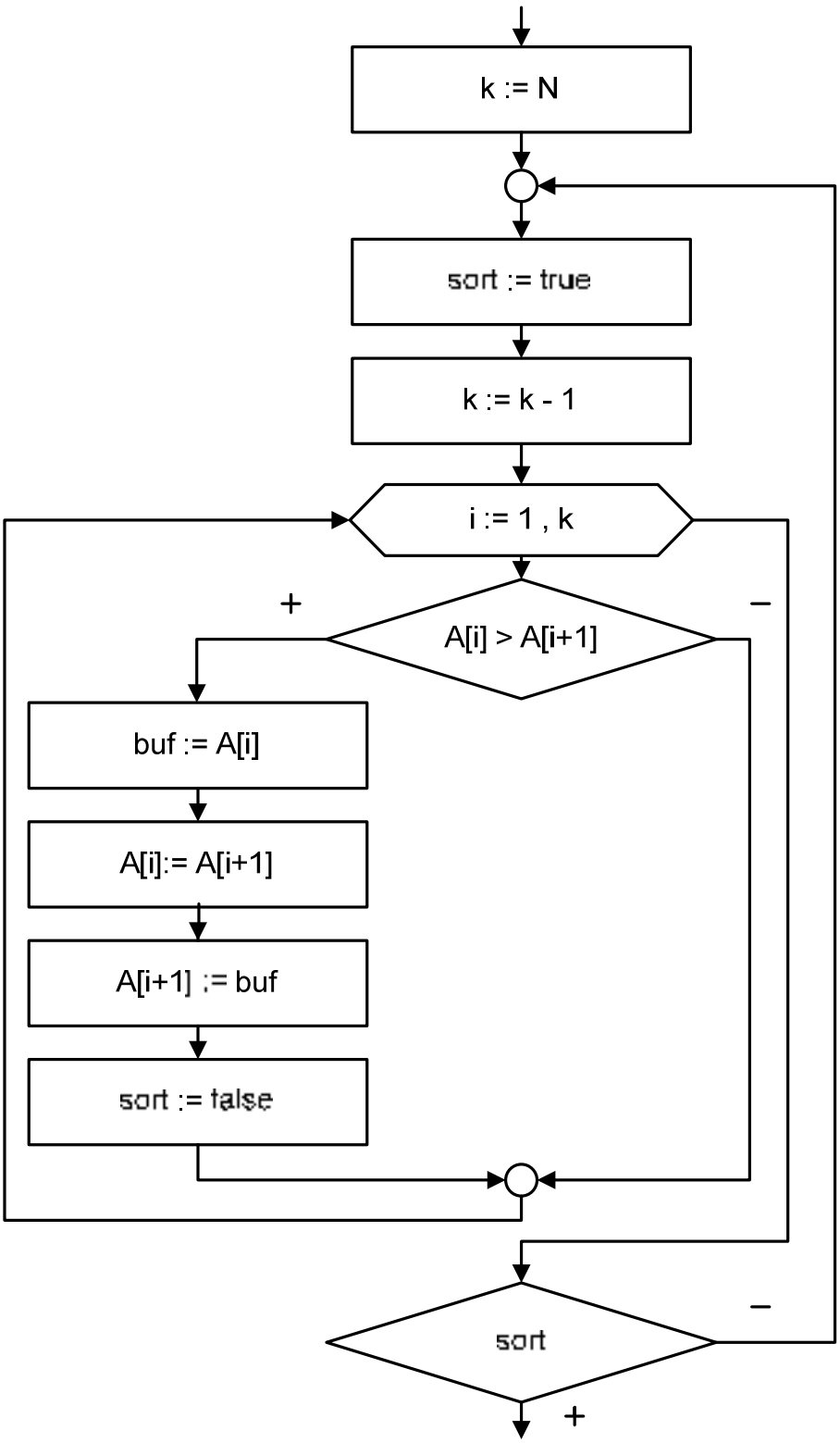


Рис. 4.21. Усовершенствованный метод пузырька

```
k:=N;
 repeat
   sort:=true;
   k:=k-1;
   for i:=1 to k do
      if A[i]>A[i+1] then
         begin
           buf   := A[i];
           A[i]  := A[i+1];
           A[i+1]:= buf;
           sort:=false;
        end;
  until sort;
```

Итак, для каждого из описанных методов требуется два цикла для осуществления сортировки. Число проверок условий в этих алгоритмах равно

$$W = \sum_{k=1}^{N-1} i = 1 + 2 + 3 + ... + (N-2) + (N-1)$$

Это сумма арифметической прогрессии, она равна

$$W = \frac{N(N-1)}{2} = \frac{N^2 - N}{2} \cong \Theta\left(N^2\right)$$

По данной формуле видно, что зависимость времени исполнения сортировки от размера массива квадратичная. Для этого мы введена функция $\Theta\left(N^2\right)$. Видно, что чем длиннее массив, тем больше времени требуется на его упорядочивание, причем время возрастает нелинейно и достаточно быстро. Про такие алгоритмы принято говорить. что время их исполнения порядка N^2. Вообще доказано, что для сортировки одномерного массива максимально быстрые алгоритмы не могут быть быстрее чем $N\log(N)$. Однако эти алгоритмы в данной главе рассматривать не будем, поскольку они достаточно сложны для неподготовленного читателя. Следует отметить, что сложность $N\log(N)$ гораздо более приемлемая, чем N^2, поскольку, например если длина массива возрастает в 100 раз, то время простой сортировки увеличится в

10 000 раз. А для быстрой сортировки увеличение времени произойдет примерно в $100 \cdot \log_2(100) \approx 664$ раза.

Еще одной интересной задачей для одномерных массивов является задача поиска одинаковых элементов. Одинаковые элементы можно найти если каждый элемент сравнить с каждым.

Т.е. первый элемент сравниваем со вторым, потом с третьим и так до конца массива. Далее сравниваем второй элемент с третьим, четвертым, пятым и т.д. Алгоритм продолжаем до тех пор, пока все пары не будут рассмотрены.

Очевидно, что эта задача решается в два цикла (рис. 4.22):

```
for k:=1 to N-1 do
  for i:=k+1 to N do
```

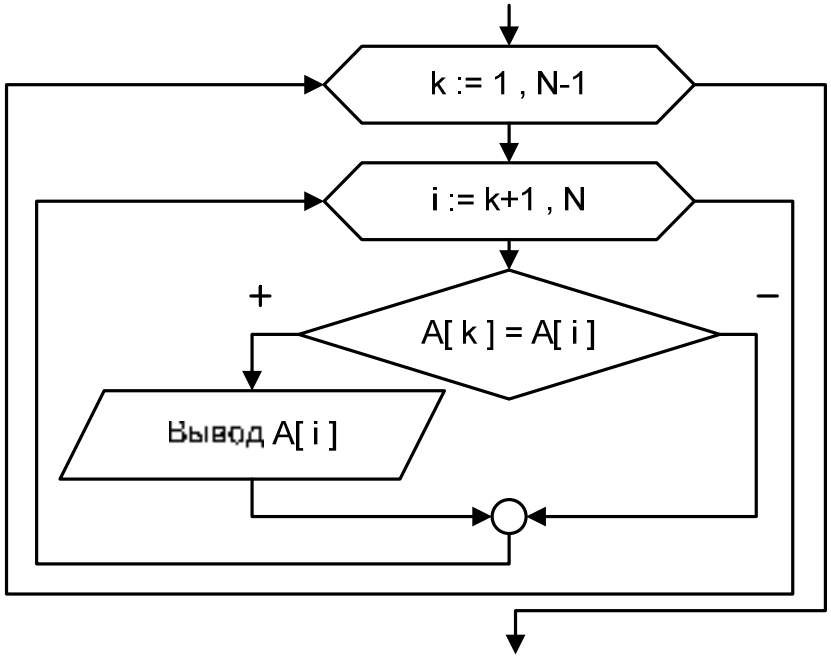


Рис. 4.22. Поиск одинаковых элементов
в одномерном массиве

```
if A[k]=A[i] then
  writeLn(A[i]);
```

Сложность алгоритма полиномиальная, квадратичная. Всего потребуется, как и для сортировки массива $\left(N^2 - N\right)/2$ проверок условия.

4.6. Линейная алгебра и векторы

Как утверждалось ранее, одномерные массивы с числовыми элементами называют векторами. Из курса математики известны аналитические операции над векторами, очень часто при программировании математических моделей требуется их автоматизировать. Рассмотрим, как известные действия линейной алгебры решаются при помощи массивов.

Сложение двух векторов

Если требуется сложить два вектора в *N*-мерном пространстве, то обычная запись выглядит следующим образом:

$$\vec{\mathbf{C}} = \vec{\mathbf{A}} + \vec{\mathbf{B}} = \{a_1, a_2, ..., a_N\} + \{b_1, b_2, ..., b_N\} =$$

$$= \{a_1 + b_1, a_2 + b_2, ..., a_N + b_N\} = \{c_1, c_2, ..., c_N\},$$

соответственно, если в *Pascal* вектор представлен одномерным массивом, то сложение двух массивов будет следующим:

```
for i:=1 to N do
  C[i] := A[i] + B[i]
```

Нельзя просто записать, ***C:= A+B***, поскольку операция «+», ровно как и «–» не применима к структурированным типам данных, коими являются, в частности, одномерные массивы.

Изменение длины вектора

Изменение длины вектора – это такая операция, при которой каждая из его координат домножается на скаляр (обыкновенное число):

$$\vec{\mathbf{B}}_{res} = k\vec{\mathbf{B}} = \{kb_1, kb_2, ..., kb_N\},$$

что на языке *Pascal* имеет следующий вид:

```
for i:=1 to N do
  B[i]:=k*B[i];
```

Следует отметить, что запись ***B:=k*B*** является неприемлемой, поскольку операция умножения «*» определена только для простых

числовых типов данных. Нельзя просто взять и умножить число на массив. Нужно обязательно описать всю процедуру подобно тому, как это было сделано выше.

Скалярное произведение двух векторов

Скалярное произведение двух векторов $\vec{\mathbf{A}}$ и $\vec{\mathbf{B}}$:

$$P = \vec{\mathbf{A}} \cdot \vec{\mathbf{B}} = \{a_1, a_2, ..., a_N\} \cdot \{b_1, b_2, ..., b_N\} = a_1 b_1 + a_2 b_2 + ... + a_N b_N$$

Для массивов имеет место следующая реализация алгоритма:

```
P:=0;
for i:=1 to N do
  P := P + A[i]*B[i];
```

Модуль вектора

По определению, модуль вектора – это величина представляющая собой квадратный корень из суммы квадратов координат, т.е.:

$$S = \left|\vec{\mathbf{A}}\right| = \sqrt{\sum_{i=1}^{N} a_i^2} = \sqrt{a_1^2 + a_2^2 + ... + a_N^2}\ . \qquad (4.1)$$

Для массивов имеем:

```
S:=0;
for i:=1 to N do
  S:=S + sqr(A[i]);
S:= sqrt(S);
```

Нормировка вектора

Нормировка – это такое преобразование вектора при котором все его компоненты по модулю становятся меньше единицы. Нормировка показывает относительную выраженность одной из координат вектора относительно других.

Есть несколько подходов к нормировке вектора. Рассмотрим наиболее часто употребляемые. А именно,

– нормировка на модуль

Нормировка на модуль основана на том основании, что в евклидовом пространстве модуль вектора больше чем любая из его координат (можно

привести теорему Пифагора, где гипотенуза всегда больше любого из катетов).

В соответствии с формулой (4.1), имеем:

$$\vec{\mathbf{A}}_{norm} = \frac{\vec{\mathbf{A}}}{|\vec{\mathbf{A}}|} = \frac{\vec{\mathbf{A}}}{S} = \left\{ \frac{a_1}{S}, \frac{a_2}{S}, ..., \frac{a_N}{S} \right\} = \{ a_{1norm}, a_{2norm}, ..., a_{Nnorm} \}$$

Это значит, что если известна величина нормы S, то алгоритм будет следующим:

```
for i:=1 to N do
  A[i]:=A[i]/S;
```

– нормировка на модуль максимального по модулю.

Здесь для нормировки нужно найти максимальный по модулю элемент и разделить все компоненты вектора на его модуль.

Поиск максимального по модулю элемента может быть представлен следующим образом:

```
maxA := abs(A[1]);
for i:=1 to N do
   If abs(A[i]) > maxA then
     maxA := abs(A[i]);
```

значит нормировка

```
for i:=1 to N do
   A[i]:=A[i]/maxA;
```

$$\vec{\mathbf{A}}_{norm} = \frac{\vec{\mathbf{A}}}{\max A} = \left\{ \frac{a_1}{\max A}, \frac{a_2}{\max A}, ..., \frac{a_N}{\max A} \right\} = \{ a_{1norm}, a_{2norm}, ..., a_{Nnorm} \}.$$

Векторное произведение

Векторное произведение определено для трехмерного пространства, что означает равенство длины вектора $N = 3$. Векторное произведение по определению равно

$$\vec{\mathbf{C}} = \vec{\mathbf{A}} \times \vec{\mathbf{B}} = \begin{vmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \vec{e}_1 (a_2 b_3 - a_3 b_2) - \vec{e}_2 (a_1 b_3 - a_3 b_1) + \vec{e}_3 (a_1 b_2 - a_2 b_1) =$$

$$= \{ (a_2 b_3 - a_3 b_2), -(a_1 b_3 - a_3 b_1), (a_1 b_2 - a_2 b_1) \}.$$

На *Pascal* это будет выглядеть следующим образом:

```
C[1]:=A[2]*B[3] - A[3]*B[2];
C[2]:=-A[1]*B[3] + A[3]*B[1];
C[3]:=A[1]*B[2] - A[2]*B[1];
```

5. ДВУМЕРНЫЕ МАССИВЫ

5.1 Понятие и объявление двумерного массива

Довольно часто при обработке больших объемов информации имеем дело с упорядочиванием данных по нескольким признакам. Если в структуре данных есть возможность выделения содержимого по этим признакам, то имеет смысл организовывать, так называемый, многомерный массив. Самым простым примером многомерного массива является *двумерный*.

Двумерный массив – это одномерный массив, каждым элементом которого является свой одномерный массив. Получается так называемый «*массив массивов*». Можно сказать и так: *двумерный массив* – это такой тип данных, элементы которого однотипны и каждый из них характеризуется уникальной парой чисел: *индексом строки* и *индексом столбца*.

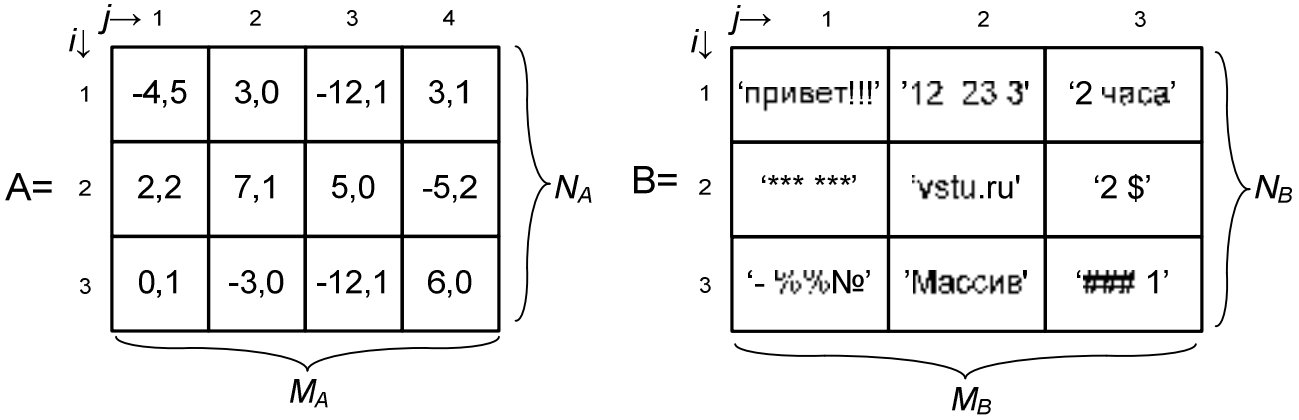


Рис. 5.1 Примеры двумерных массивов

Естественным отображением двумерного массива является таблица. Таблица есть двумерная структура, у которой вдоль горизонтального направления перечень одних свойств, а вдоль вертикального – других.

Пересечение столбца и строки дает нужный элемент, одновременно обладающий обоими свойствами. Для двумерного массива этими свойствами являются числа – индексы строк и столбцов.

Примеры двумерных массивов изображены на рис. 5.1.

На иллюстрации массив ***A*** – это массив, элементами которого являются дробные числа (тип ***real***). Объявление массива ***A*** следующее:

```
A: array[1..10, 1..10] of real;
```

Массив ***A*** объявлен с запасом по размерности. На рисунке размерность массива 3×4, а при объявлении выделяем память под 10×10=100 ячеек типа ***real***. Это значит, что размер будет 100×6 байт = 600 байт. Используемый размер 3×4×6 байт 72 байта. Видно как сильно зависит расход памяти от размерности массива. Потому следует следить за размерностью и, по возможности, не объявлять слишком больших массивов. Условимся далее для обозначения числа строк использовать переменную ***N***, а для столбцов ***M***. Т.е. для массива ***A*** N_A=3, M_A=4; для массива ***B*** N_B=3, M_B=3.

B – массив состоящий из наборов символов максимальной длины 15. Объявление массива ***B*** такое:

```
B: array [1..3, 1..3] of string[15];
```

Как отмечалось ранее двумерный массив – массив массивов, а потому правомерна такая запись для ***A*** и ***B***:

```
A: array[1..10] of array[1..10] of real;
B: array [1..3] of array[1..3] of string[15];
```

Для непосредственного обращения к элементу нужно указать его адрес. Адрес в двумерном массиве – пара чисел. Сначала идет номер элемента во внешнем одномерном массиве, а потом во внутреннем. Поскольку двумерные массивы представляем в виде таблиц, то условимся первым числом обозначать номер строки, а вторым – индекс столбца.

Вообще, строки и столбцы можно поменять местами, поскольку организовывать порядок их задания можно произвольно. Но далее всегда будем обозначать сначала строку, а затем столбец.

Для примеров на рис. 5.1 это выглядит следующим образом:

```
A[1,1]=-4,5  A[1,2]= 3,0  A[1,3]=-12,1  A[1,4]= 3,1
A[2,1]= 2,2  A[2,2]= 7,1  A[2,3]=  5,0  A[2,4]=-5,2
A[3,1]= 0,1  A[3,2]=-3,0  A[3,3]=-12,1  A[3,4]= 6,0
```

И, соответственно, массив ***B***:

```
B[1,1]= 'привет' B[1,2]='12 23 3'  B[1,3]='2 часа'
B[2,1]='*** ***' B[2,2]='vstu.ru'  B[2,3]=   '2 $'
B[3,1]=  '- %%№' B[3,2]= 'Массив'  B[3,3]= '### 1'
```

Порядок обращения к элементам двумерного массива сходен с порядком обращения к элементам одномерного, т.е. нельзя подвергать изменению целиком весь массив сразу. Для обращения, как это видно выше, в квадратных скобках через запятую указываются координаты элемента по вертикали и горизонтали.

В некоторых языках программирования счет индексам начинается не с 1 а с 0. В *Pascal* работа с массивами организованна достаточно просто, и можно задавать диапазон изменения индексов в любых границах (даже отрицательных). Но для удобства и однозначности впредь все индексы будем начинать считать с 1.

Еще следует отметить, что наиболее естественными объектами, которые принято хранить в двумерных массивах являются числа. Такие массивы будем называть *матрицами*, так же как и в математике. И именно на их примере рассмотрим основные алгоритмы обработки этих структур.

5.2 . Поэлементная обработка двумерных массивов

Прямая безусловная поэлементная обработка двумерного массива

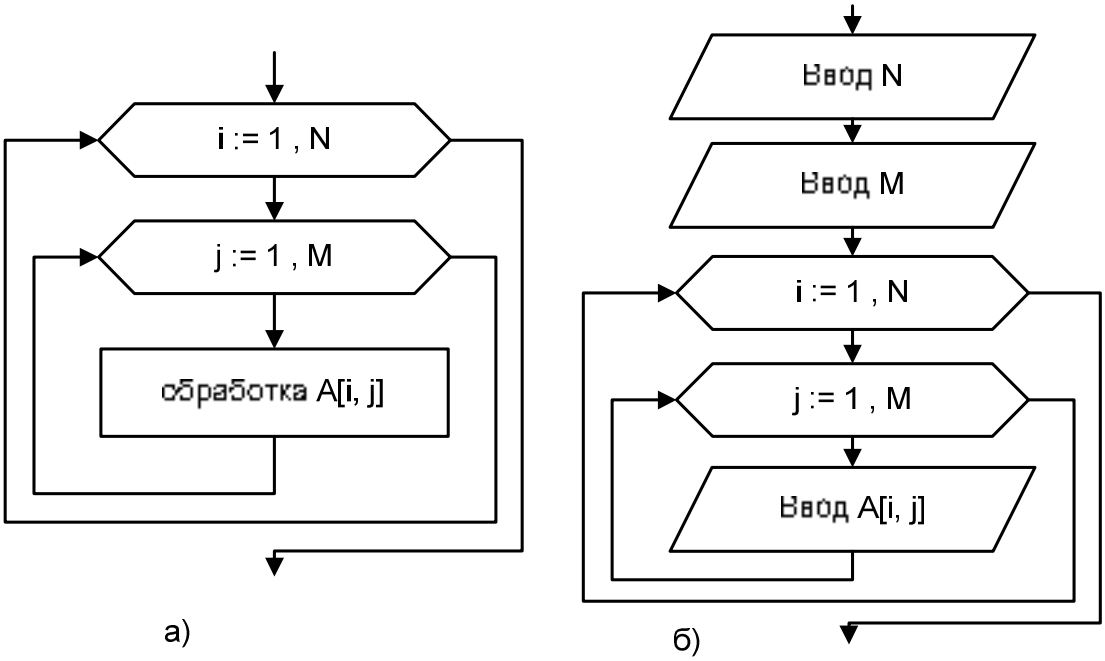


Рис. 5.2. Поэлементная обработка двумерного массива (*а*) и ввод массива (*б*)

предполагает такую обработку, при которой все его элементы безусловно просматриваются в порядке возрастания индексов. Индексы можно увеличивать, рассматривая массив по строкам или по столбцам. Блок-схема этого процесса (построчная реализация) представлена на рис. 5.2, *а*.

Самые простые алгоритмы поэлементной обработки массива – это алгоритмы *ввода* и *вывода*. Алгоритм ввода или *инициализация* пользователем представлен на рис. 5.2, *б*. Здесь, поскольку массивы мы объявляем с запасом размерности, следует сначала указать число строк ***N*** и количество столбцов ***M***. Массив рассматриваем, согласно соглашению, построчно, т. е. во внешнем цикле меняется индекс строки, а во внутреннем – индекс столбца.

```
writeLn('введите число строк');
readLn(N);
```

```
      writeLn('введите число строк');
      readLn(M);
      for i:=1 to N do
        for j:=1 to M do
          begin
            write('A[',  i,  ',',
j, ']=');
            readLn(A[i,j]);
          end;
```

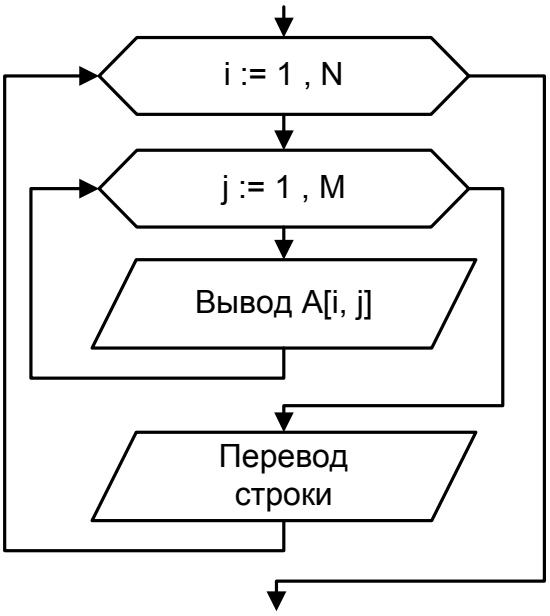


Рис. 5.3. Вывод двумерного массива

Вывод массива аналогичен вводу, только если будем выводить все элементы подряд, разные строки сольются между собой. Невозможно будет определить, где кончается одна строка и начинается следующая. Для этого нужно после вывода каждой строки принудительно переводить курсор на следующую. Делается это очень просто, методом вставки во внешний цикл алгоритма процедуры ***writeLn*** без параметров:

```
      for i:=1 to N do
        begin
          for j:=1 to M do
            write(A[i, j]:5);
          writeLn;
        end;
```

На блок-схеме (рис. 5.3) показана операция принудительного перевода курсора на следующую строку. В дальнейшем при оформлении блок-схем ее показывать не будем, негласно предполагая ее наличие. Вообще, такая обработка называется обработкой по строкам или столбцам и более детально будет рассмотрена ниже. Здесь же вывод массива приведен для того, чтобы не терять логическую связь со вводом.

В качестве примеров прямой обработки всех элементов массива можно привести алгоритм подсчета сумы, произведения, а также

изменение всех элементов. Скажем, рассмотрим увеличение всех элементов двумерного массива на некоторую константу ***x***:

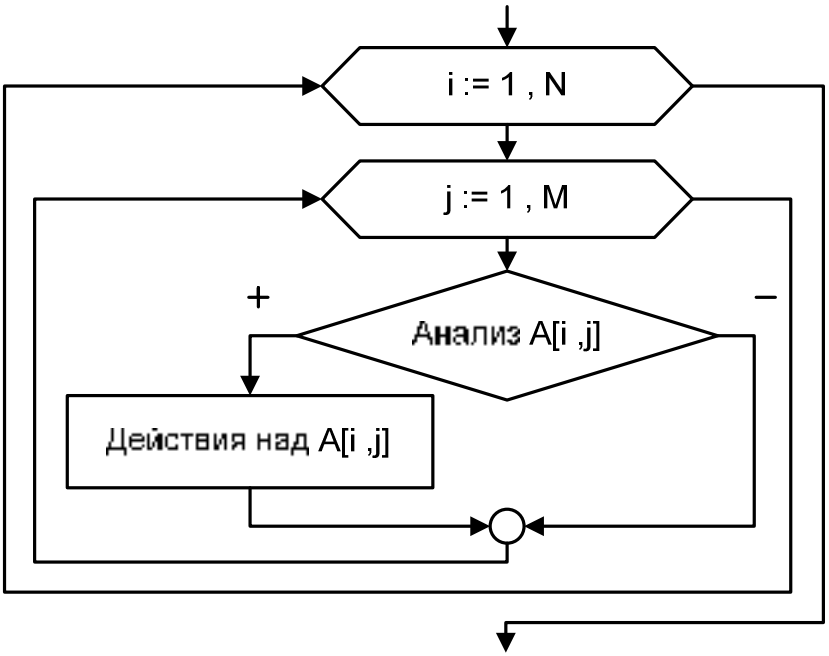


Рис. 5.4. Алгоритм анализа элементов массива

```
for i:=1 to N do
  for j:=1 to M do
    A[i, j] := A[i, j] + x;
```

Далее можно рассмотреть поэлементную обработку всего массива предполагающую анализ элементов. Таковая обработка выражается блок-схемой рис. 5.4. В теле внутреннего цикла помещено условие, в случае выполнения которого, происходит действие над элементом ***A[i,j]***.

Вот типичная задача на обработку всего двумерного массива с анализом элементов.

ПРИМЕР

Возвести в квадрат все нечетные элементы двумерного массива ***A***. Решение таково (рис. 5.5):

```
for i:=1 to N do
  for j:=1 to M do
    if odd(A[i,j]) then
      A[i,j] := sqr(A[i,j]);
```

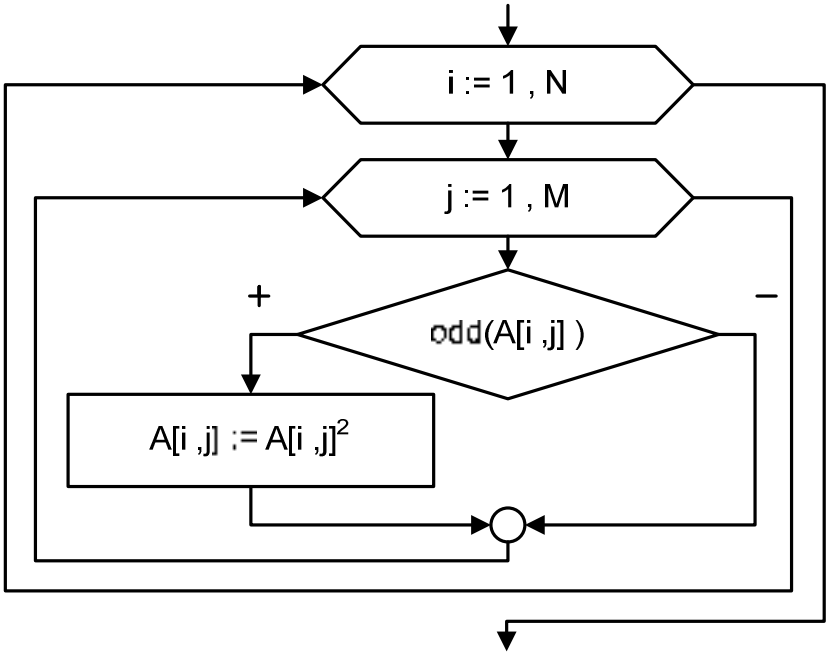


Рис. 5.5. Возведение в квадрат нечетных элементов

Напомним, что ***odd*** – логическая функция проверки нечетности.

Рассмотрим полностью задачу, которая в предыдущей главе решалась для одномерных массивов, а именно:

ПРИМЕР

Найти среднее арифметическое четных элементов массива.

Для начала составим тестовый пример:

вход: | 2 | 1 | 2 |
| 8 | 4 | 5 |; *выход*: ***SrA***=4.

Сам алгоритм практически такой же, как и в задаче для одномерных массивов отличие здесь в том, что дополнительно добавляется цикл по столбцам (по ***j***). Блок схема алгоритма представлена на (рис. 5.6).

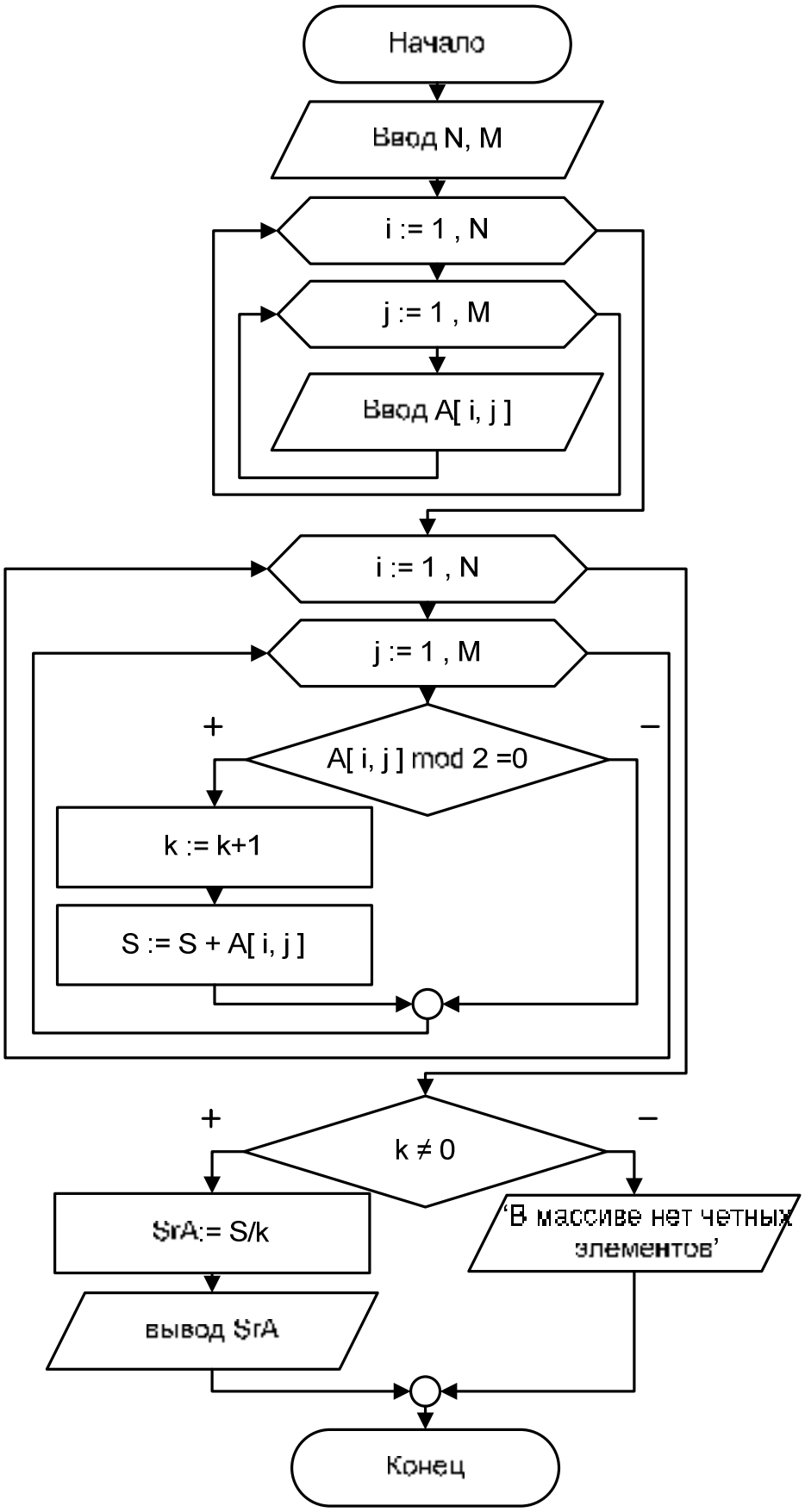


Рис. 5.6. Среднее арифметическое четных элементов двумерного массива

Программа будет такова:

```
program massiv2m;
var A:array[1..10,1..10] of integer;
    i,j,k,N,M:byte;
    S:integer;
    SrA:real;
begin
writeLn('Введите количество элементов в массиве');
readLn(N,M);
for i:=1 to N do
  for j:=1 to M do
    begin
      write('A[',i,',',j,']=');
      readLn(A[i,j]);
    end;
S:=0;
k:=0;
for i:=1 to N do
  for j:=1 to M do
    if (A[i,j] mod 2) = 0 then
      begin
        S:=S+A[i,j];
        k:=k+1;
      end;
if k<>0 then
  begin
    SrA := S/k;
    writeLn('Среднее арифметическое: ', SrA:8:2);
  end
else
  writeLn('В массиве нет четных элементов');
end.
```

Поэлементная обработка массива с анализом может быть представлена некоторыми классическими алгоритмами. Наиболее известные и часто встречающиеся – алгоритмы поиска экстремальных по значению (максимум и минимум). Итак, для примера рассмотрим поиск максимума в двумерном массиве. Алгоритм точно такой же, как и для одномерного массива. Стоит только не забывать, что при просмотре меняются как строки, так и столбцы. Кроме того, необходима пара индексов для выяснения точного местоположения элемента с

максимальным значением. За координаты отвечают переменные ***IMax***, ***JMax***, а за сам максимум переменная ***Max*** (рис. 5.7):

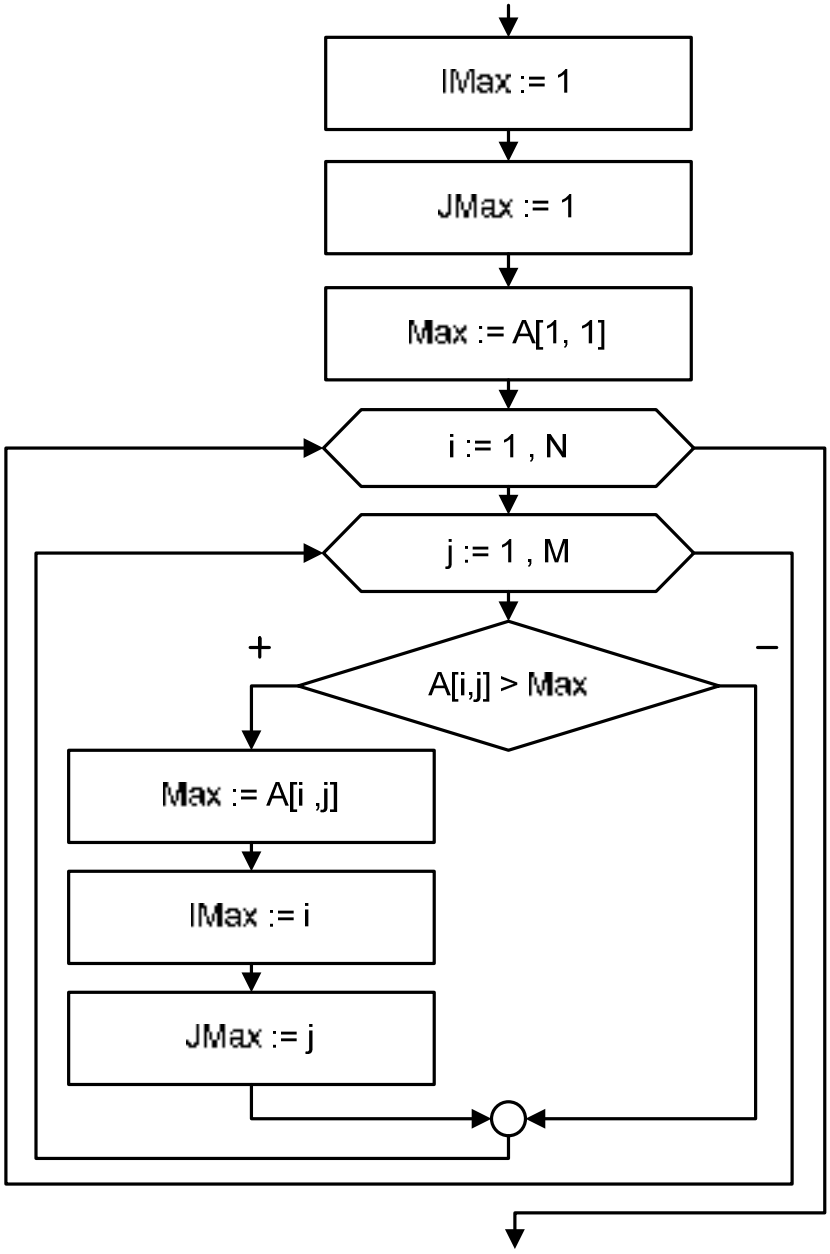


Рис. 5.7 Поиск максимума

```
Imax :=1;
Jmax :=1;
Max := A[1,1];
for i:=1 to N do
  for j:=1 to M do
    if A[i,j]>Max then
      begin
        Max := A[i,j];
```

```
        IMax := i;
        JMax := j;
    end;
```

Очевидно, что для поиска минимального элемента потребуется изменить знак в условии с «>» на «<». Да и имена переменных в которых будут храниться искомые значения следует заменить на ***IMin***, ***JMin*** и ***Min***.

Алгоритмы поиска элементов не всегда могут быть такими простыми. В качестве примера рассмотрим еще одну задачу, которая решалась для одномерного массива, а именно:

ПРИМЕР

Найти наименьший среди нечетных элементов двумерного массива.

Для этой задачи составим тестовый пример:

вход: $A =$

−8	−1	3	−1
−5	0	7	6
2	1	9	4

выход: ***Min*** = −5; ***IMin*** = 2; ***JMin*** = 1;

Здесь при решении есть некоторые особенности, связанные с тем, что обрабатываемый массив двумерный. А так алгоритм практически такой же, как и в предыдущей задаче, и распадается на две части: поиск первого нечетного элемента и поиск минимума, при условии нечетности, начиная с найденного. Тут тоже воспользуемся для первой части задачи циклом с предусловием. Теперь нужно производить инкрементацию не только индекса столбца ***JMin***, но и следить за своевременной инкрементацией индекса столбца ***IMin*** (рис. 5.8):

```
JMin := 1;
while (not odd(A[IMin,JMin])) and (IMin<N) do
```

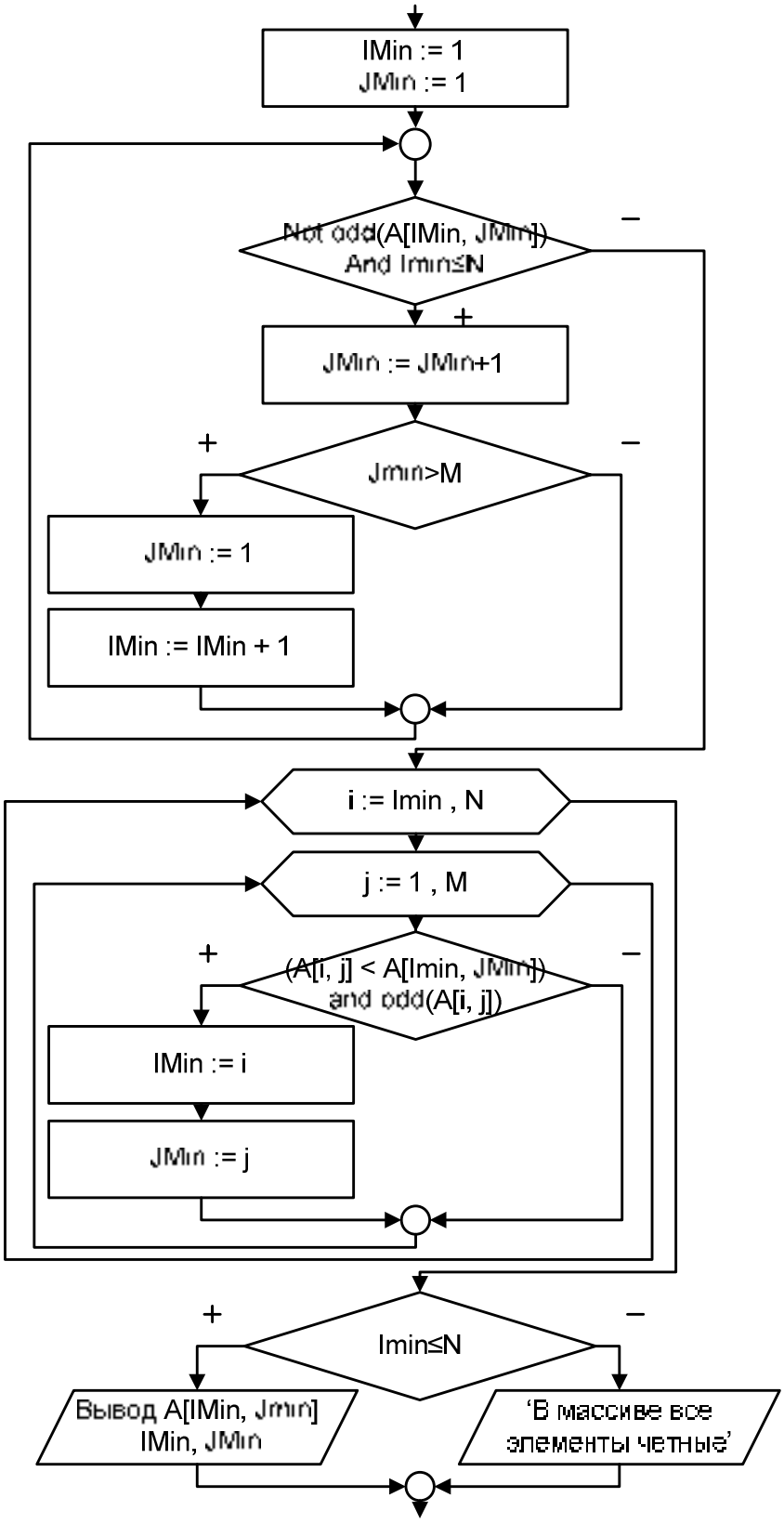


Рис. 5.8. Поиск минимального среди нечетных

IMin := 1;

```
  begin
    JMin := JMin+1;
    if JMin>M then
      begin
        JMin := 1;
        IMin := IMin+1;
      end;
  end;
for i:= IMin to N do
  for j:= 1 to M do
    if (A[i,j]< A[IMin,JMin]) and (odd(A[i,j]) then
    begin
      Min := A[i,j];
      IMin := i;
      JMin := j;
    end;
if IMin<=N then
  writeLn('A[',IMin,',',JMin,']=',A[IMin,JMin]);
else
  writeLn('в массиве все элементы четные');
```

Если в массиве все элементы четные, то будет выведено соответствующее сообщение. Рассмотренный пример является одним из вариантов решения задачи, однако, он не единственный. Есть иная реализация алгоритма поиска с использованием логической переменной ***Flag***, аналогично тому как это делалось для одномерного массива. Здесь будет все аналогично, за исключением добавления цикла по строкам и включения как индексов строк, так и столбцов (алгоритм рассматривать не будем).

Вот еще одна типовая задача на максимумы и минимумы для двумерных массивов.

ПРИМЕР

В двумерном массиве поменять местами максимальный и минимальный элементы.

Тут все просто. Ищем индексы максимума и минимума, а далее производим обмен в три действия (рис. 5.9).

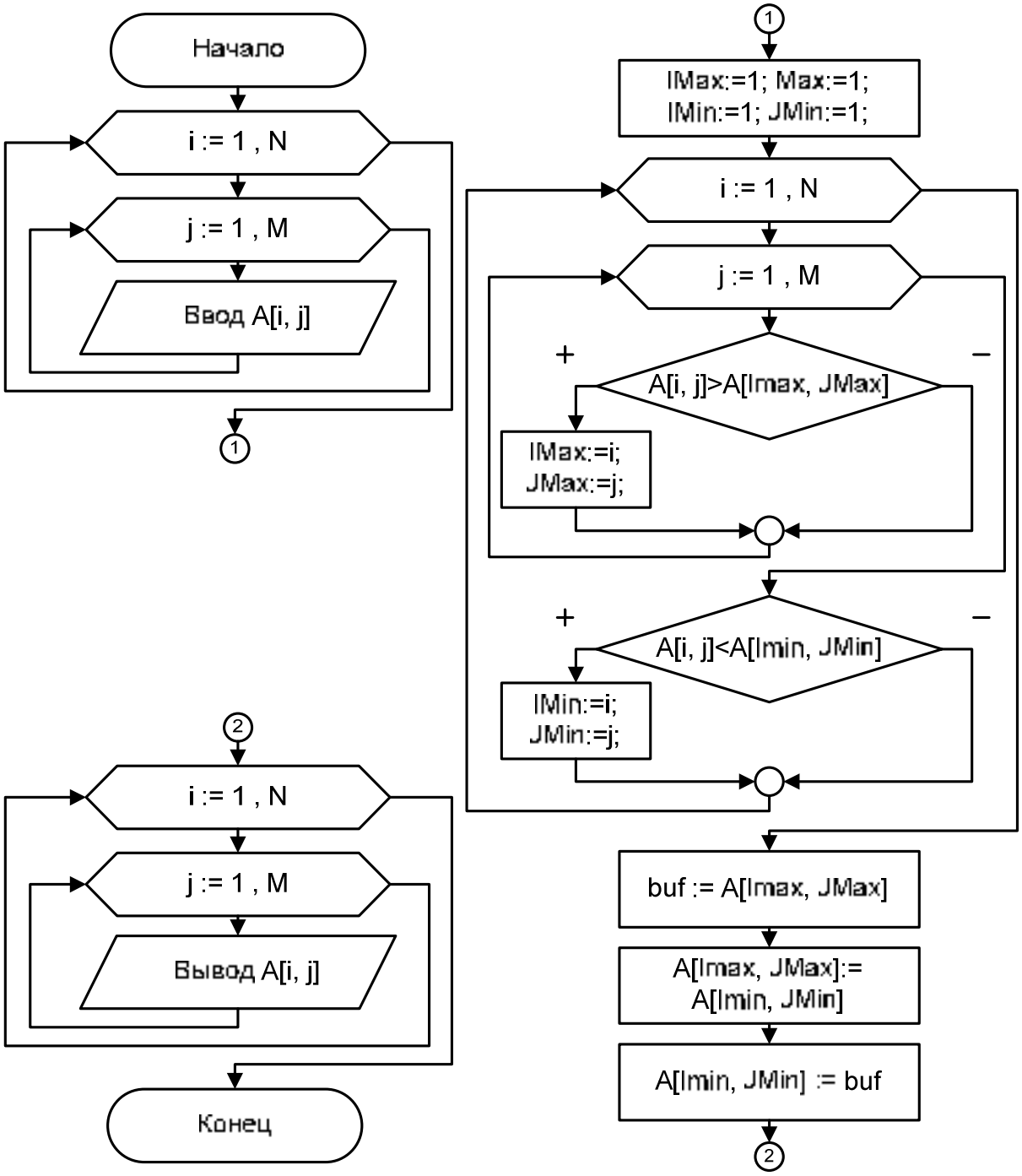


Рисунок 5.9 Обмен максимума и минимума в двумерном массиве

```
program MaxMInExch;
const L=10;
type T2Mx = array[1.. L,1..L] of integer;
var A: T2Mx;
    i,j,Imax,Jmax,IMin,JMin,N,M:byte;
    buf:integer;
begin
  writeLn('Введите размерность матрицы:');
```

```
  readLn(N,M);
  for i:=1 to N do
    for j:=1 to M do
      begin
        write('A[',i,',',j,']=');
        readLn(A[i,j]);
      end;
  IMax:=1; JMax:=1;
  IMin:=1; JMin:=1;
  for i:=1 to N do
    for j:=1 to M do
      begin
        if A[i,j]>A[IMax,JMax] then
          begin
            IMax:=i; JMax:=j;
          end;
        if A[i,j]<A[IMin,JMin] then
          begin
            IMin:=i; JMin:=j;
          end;
      end;
  buf:=A[IMax,JMax];
  A[IMax,JMax]:=A[IMin,JMin];
  A[IMin,JMin]:=buf;
  writeLn('Матрица после преобразования:');
  for i:=1 to N do
    begin
      for j:=1 to N do
        write(A[i,j]:4);
    writeLn;
    end;
end.
```

ПРИМЕР

Далее можно рассмотреть задачу формирования из заданного массива нового.

*Из матрицы **A** получить новые одномерные массивы **C** и **D**. В **C** содержатся положительные компоненты матрицы **A**, а в **D** – отрицательные. Длины получившихся массивов сохраняются в переменных **Nc** и **Nd**, соответственно.*

Тестовый пример может выглядеть так:

вход: $A =$

1	–5	4	2
–3	3	0	6
7	0	8	0

выход: $B =$

1	4	2	3	6	7	8

$D =$

–5	–3

Листинг программы для блок-схемы (рис. 5.10):

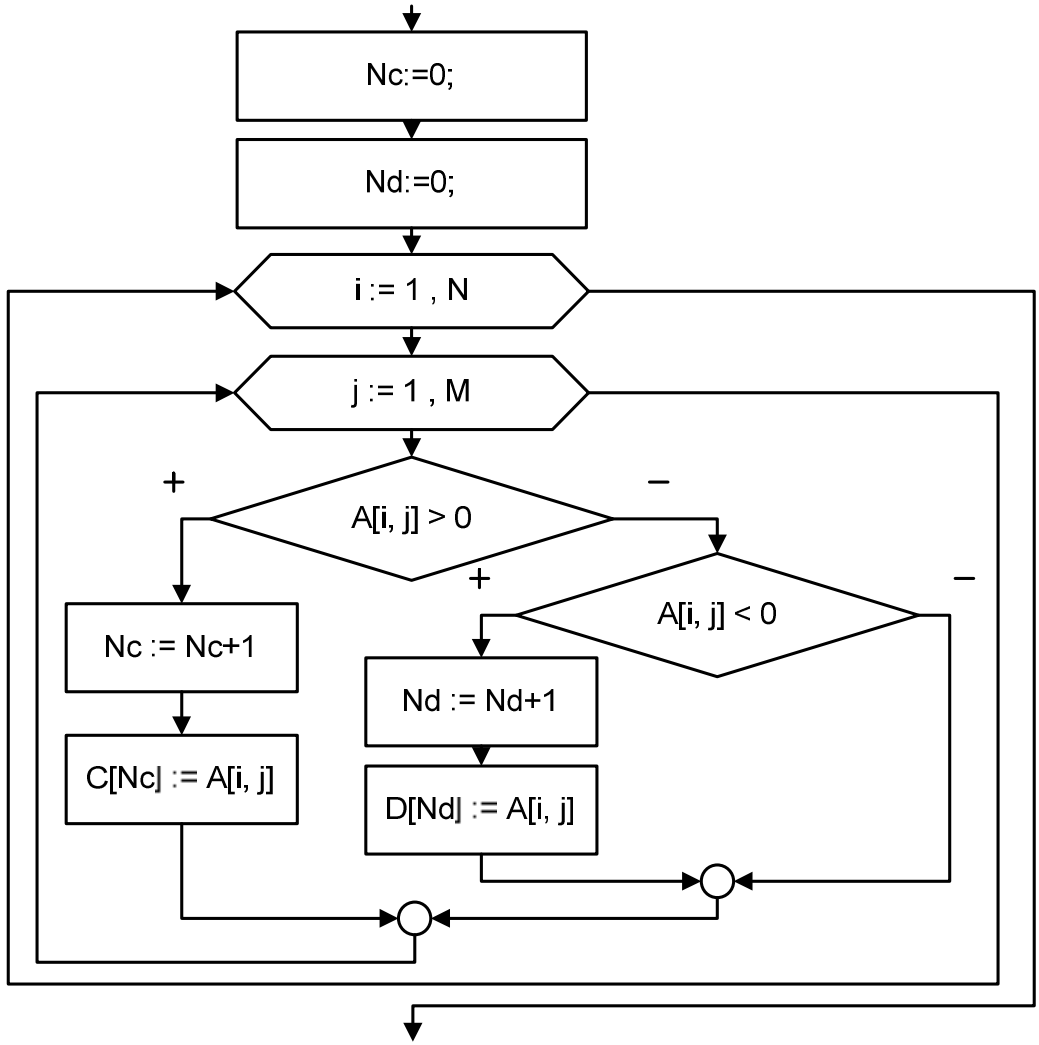


Рис. 5.10. Формирование из двумерного массива пары одномерных

```
Nc := 0; Nd := 0;
for i:=1 to N do
  for j:=1 to M do
    if A[i,j]>0 then
      begin
        Nc:=Nc+1; C[Nc]:=A[i,j];
      end
    else
      if A[i,j]<0 then
        begin
          Nd := Nd+1; C[Nd]:=A[i,j];
        end;
```

5.3 Обработка отдельных строк или столбцов матрицы

Важным классом алгоритмов обработки двумерных массивов является построчная или постолбцовая обработка. Если вспомнить одно из определений двумерного массива, которое говорит что это «массив одномерных массивов», то подход к поставленной задаче упрощается.

Для решения таких задач можно воспользоваться алгоритмами, показанными на рис. 5.11. Суть их сводится к тому, что внутри внешнего цикла помещаются действия, которые можно представить в виде алгоритма на одномерном массиве, если положить неизменным индекс строки ***i*** при построчном, или индекс ***j*** при постолбцовом проходе.

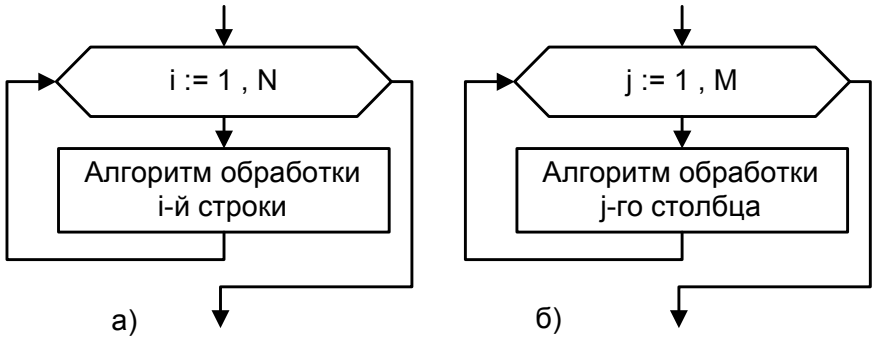


Рис. 5.11. Построчная (*а*) и постолбцовая (*б*) обработка двумерного массива

В качестве примера можно решить, скажем, такую задачу:

ПРИМЕР

Найти сумму положительных элементов в каждой строке матрицы.

Алгоритм решения состоит в следующем. Во внешнем цикле меняем индекс строки (рис. 5.12, *а*), а во внутреннем решаем задачу поиска суммы элементов одномерного массива с последующим выводом результатов на экран (рис. 5.12, *б*). Индекс строки ***i*** фиксируем во внутреннем цикле. Поиск суммы обычный, по рекуррентной формуле: «сумма текущего равна накопленной сумме на предыдущем, плюс текущее». На каждом новом проходе (при изменении индекса строки ***i***) происходит обнуление переменной хранящей текущее значение суммы ***S***.

Для иллюстрации задачи предоставим тестовый пример:

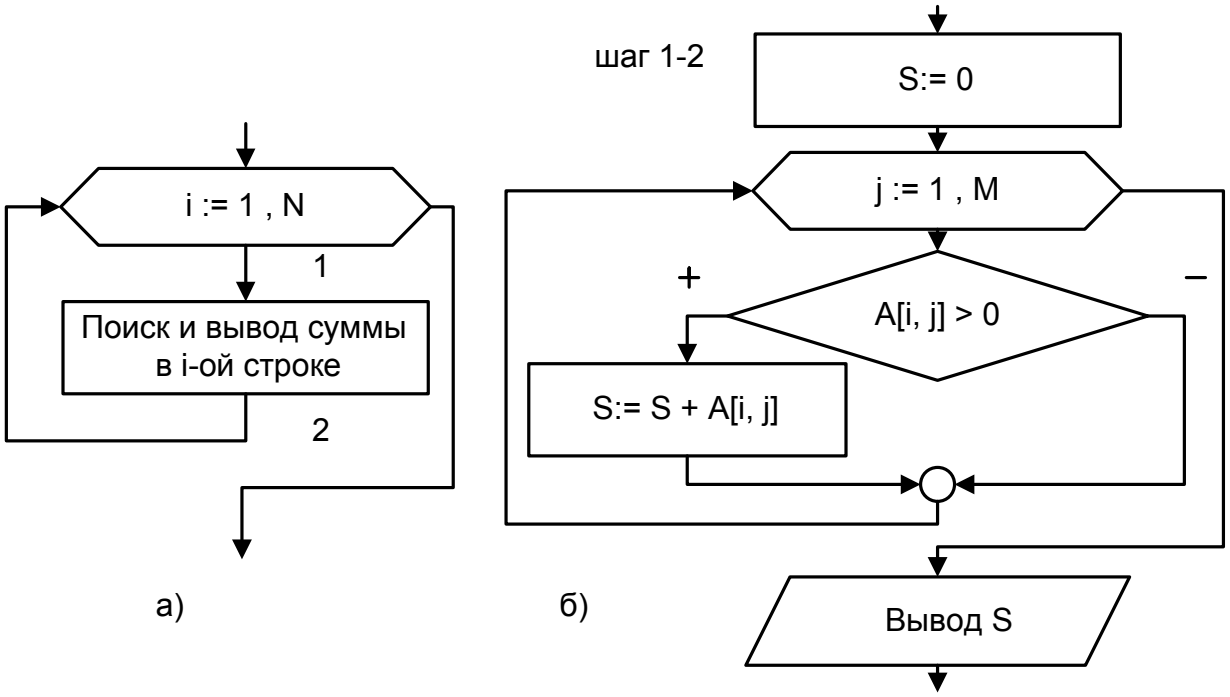


Рис. 5.12. Поиск суммы положительных элементов в каждой строке

вход: $A = \begin{array}{|c|c|c|c|c|}\hline -1 & -4 & -8 & 0 & -2 \\ \hline 1 & 0 & 4 & -5 & 3 \\ \hline 2 & -7 & -1 & 0 & 8 \\ \hline \end{array}$ *выход*: $S_1 = 0$, $S_2 = 8$, $S_3 = 10$

Программная часть на языке *Pascal* следующая:

```
for i:=1 to N do
  begin
    S := 0;
    for j:=1 to M do
      if  A[i,j]>0 then
        S:=S+A[i,j];
    writeLn('сумма ', i, '-той строки равна', S);
  end;
```

А теперь рассмотрим пример с обработкой элементов по столбцам.

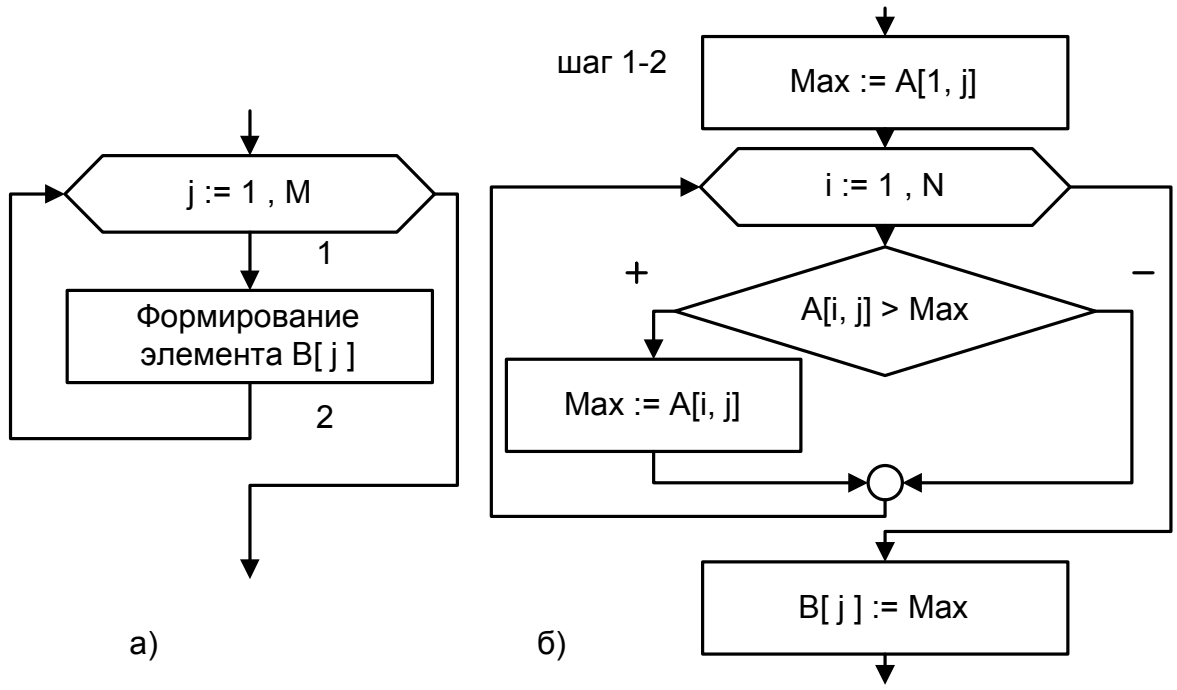


Рис. 5.13. Поиск максимума в каждом столбце

ПРИМЕР

Переписать максимальные элементы каждого столбца двумерного массива ***A*** *в одномерный массив* ***B***.

Тестовый пример выглядит так:

вход: $A =$

–1	–4	–8	0	–2
1	0	4	–5	3
2	–7	–1	0	8

выход: $B =$

2	0	4	0	8

,

а текст программы:

```
for j:=1 to M do
  begin
    max:= A[1, j];
    for i:=1 to N do
      if A[i,j]>max then
        max:=A[i,j];
    B[j]:=max;
  end;
```

Можно рассмотреть еще одну подобную задачу, а именно:

ПРИМЕР

Отсортировать по возрастанию каждую строку матрицы, т.е.:

вход: $A =$

–1	–4	–8	0	–2
1	0	4	–5	3
2	–7	–1	0	8

выход: $A =$

–8	–4	–2	–1	0
–5	0	1	3	4
–7	–1	0	2	8

Для решения нужно вспомнить, как происходила сортировка одномерного массива. Ведь каждая строка в матрице, по сути,– одномерный массив. Потому, если опять воспользоваться разбивкой алгоритма на детали, то решение достаточно простое. Во внешнем цикле меняется индекс строки (рис. 5.14 а), а во внутреннем – происходит сортировка строки по индексу столбца ***j***. В качестве метода сортировки возьмем пузырьковый.

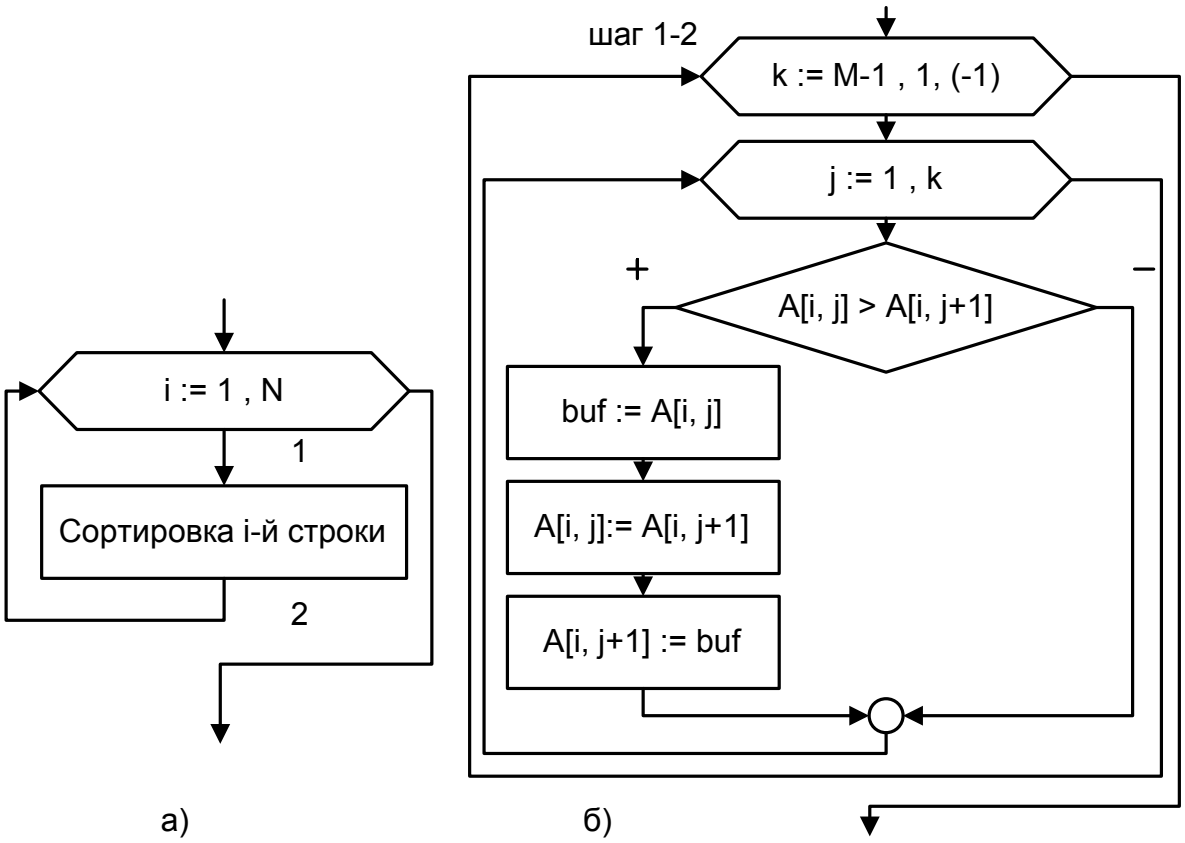


Рис. 5.14. Сортировка каждой строки матрицы

Видно, что алгоритм практически точно повторяет тот, который мы использовали для одномерного массива (рис. 5.14, *б*). Стоит обратить внимание на тот факт, что при решении этой задачи возникают циклы двойной степени вложенности, т.е. задача решается в три цикла. Вот текст алгоритма на *Pascal*:

```
for i:=1 to N do
  for k:=M-1 downTo 1 do
    for j:=1 to k do
      if A[i,j]>A[i,j+1] then
        begin
          buf := A[i,j];
          A[i,j] := A[i,j+1];
          A[i,j+1] := buf;
        end;
```

Для обработки элементов двумерного массива, на строки которого накладываются некоторые условия, нужно при просмотре этого массива внутрь циклов ставить условие не на элемент, а на индекс строки или столбца (в зависимости от условия задачи).

ПРИМЕР

Заполнить единицами каждый второй столбец матрицы, т.е.:

вход: $A =$

–1	–4	–8	0	–2
1	0	4	–5	3
2	–7	–1	0	8

выход: $A =$

–4	1	0	1	–2
0	1	–5	1	3
–7	1	0	1	8

Вот алгоритм с условием на индекс столбца (Рис. 5.**15** а)):

```
for j:=1 to M do
  if j mod 2 = 0 then
    for i:=1 to N do
      A[i,j]:=1;
```

Очевидно, что приведенный способ решения обладает недостатком, заключающимся в том, что число проходов по массиву вдвое больше чем четных столбцов. Число проходов можно сократить в 2 раза, если воспользоваться циклом с предусловием и без всяких условий рассматривать только четные столбцы. Это достигается путем «ручной»

инкрементации индекса столбца ***j*** не на 1, как это делает цикл ***for***, а на 2 (рис. 5.15, *б*):

```
j:=2;
while j<=M do
  begin
    for i:=1 to N do
      A[i,j]:=1;
    j:=j+2;
  end;
```

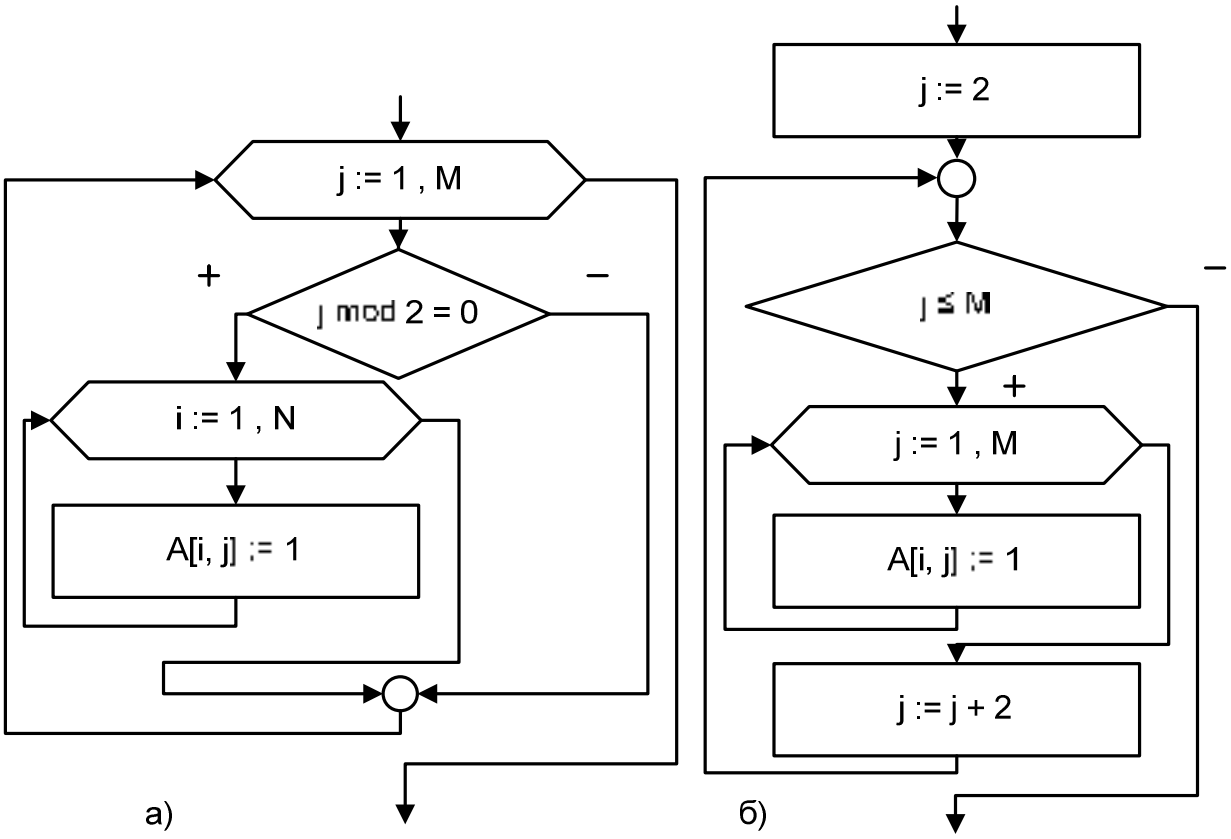


Рис. 5.15. Обработка четных столбцов:
а – при помощи цикла for; *б* – при помощи цикла while

Вообще, там, где происходит заведомо меньшее число проходов чем строк или столбцов в матрице, нет необходимости проводить полный перебор. Гораздо логичнее будет использовать или цикл с предусловием с нужным шагом по строкам/столбцам, или цикл ***for*** с меньшим числом проходов и с формулой прямого вычисления нужного индекса.

ПРИМЕР

Переставить местами соседние столбцы матрицы. Это означает следующее:

вход: $A =$

–1	–4	–8	0	–2
1	0	4	–5	3
2	–7	–1	0	8

выход: $A =$

–4	–1	0	–8	–2
0	1	–5	4	3
–7	2	0	–1	8

Последний столбец остался без изменения, поскольку для него не нашлось пары. Очевидно, что число парных перестановок столбцов в два раза меньше чем их количество в матрице.

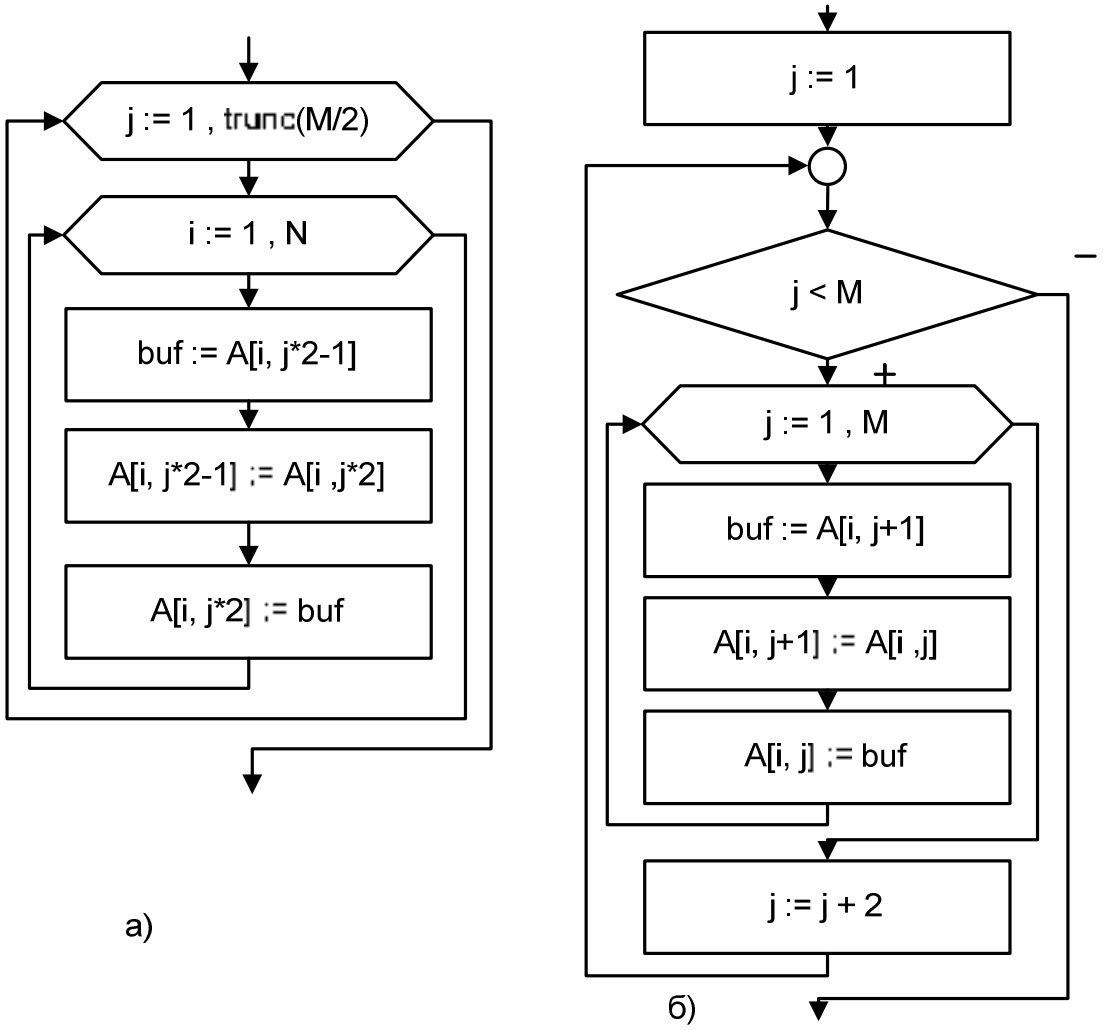


Рис. 5.16. Перестановка столбцов в матрице:

а – при помощи цикла for; *б* – при помощи цикла while

Алгоритм решения при помощи цикла ***for*** следующий (рис. 5.16, *а*):

```
for j:=1 to trunc(M/2) do
  for i:=1 to N do
    begin
      buf := A[i,j*2-1];
      A[i,j*2-1] := A[i,j*2];
      A[i,j*2] := buf;
    end;
```

А если воспользоваться циклом ***while***, то выглядеть это будет так (рис. 5.16 б):

```
j:=1;
while j<M do
  begin
    for i:=1 to N do
      begin
        buf := A[i,j+1];
        A[i,j+1] := A[i,j];
        A[i,j] := buf;
      end;
    j:=j+2;
  end;
```

5.4 Квадратные матрицы

Достаточно интересным и важным классом двумерных массивов являются квадратные. Квадратные матрицы – это такие матрицы у которых число элементов в строке равно числу элементов в столбце, т.е. ***M=N***.

В квадратных матрицах выделяют некоторые особенные группы элементов. Это, прежде всего, главная и побочная диагонали.

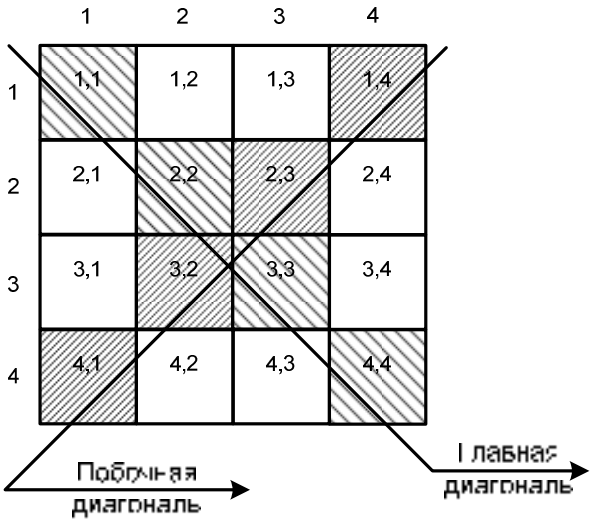


Рис. 5.17 Диагонали в матрице

Особенностью элементов на главной диагонали является тот факт, что для каждого из них индекс строки равен индексу столбца, т.е. ***i=j***. Если внимательно посмотреть на (рис. 5.17), то можно вывести правило принадлежности элемента к побочной диагонали. Это правило можно выразить формулой для индексов ***j=N-i+1***.

Для матрицы изображенной на (рис. 5.17) это действительно так. Покажем это:

```
N = 4 =>
A[1,4](i=1, j=4) j=4-1+1=4
A[2,3](i=1, j=4) j=4-2+1=3
A[3,2](i=1, j=4) j=4-3+1=2
A[4,1](i=1, j=4) j=4-4+1=1
```

Если смотреть на элементы диагоналей, то видно что они похожи на одномерные массивы, расположившиеся вдоль диагоналей. Потому для их обработки достаточно одного цикла.

ПРИМЕР

Найти среднее арифметическое отрицательных элементов главной диагонали.

Решение таково (рис. 5.18):

```
k:=0; S:=0;
for i:=1 to N do
  if A[i,i]<0 then
    begin
      k:=k+1;
      S:=S+ A[i,i];
    end;
if k<>0 then
  begin
    SrA:=S/k; writeLn('SrA =', SrA:8:2);
  end
else
  writeLn('На гл. диагонали нет отриц. элементов');
```

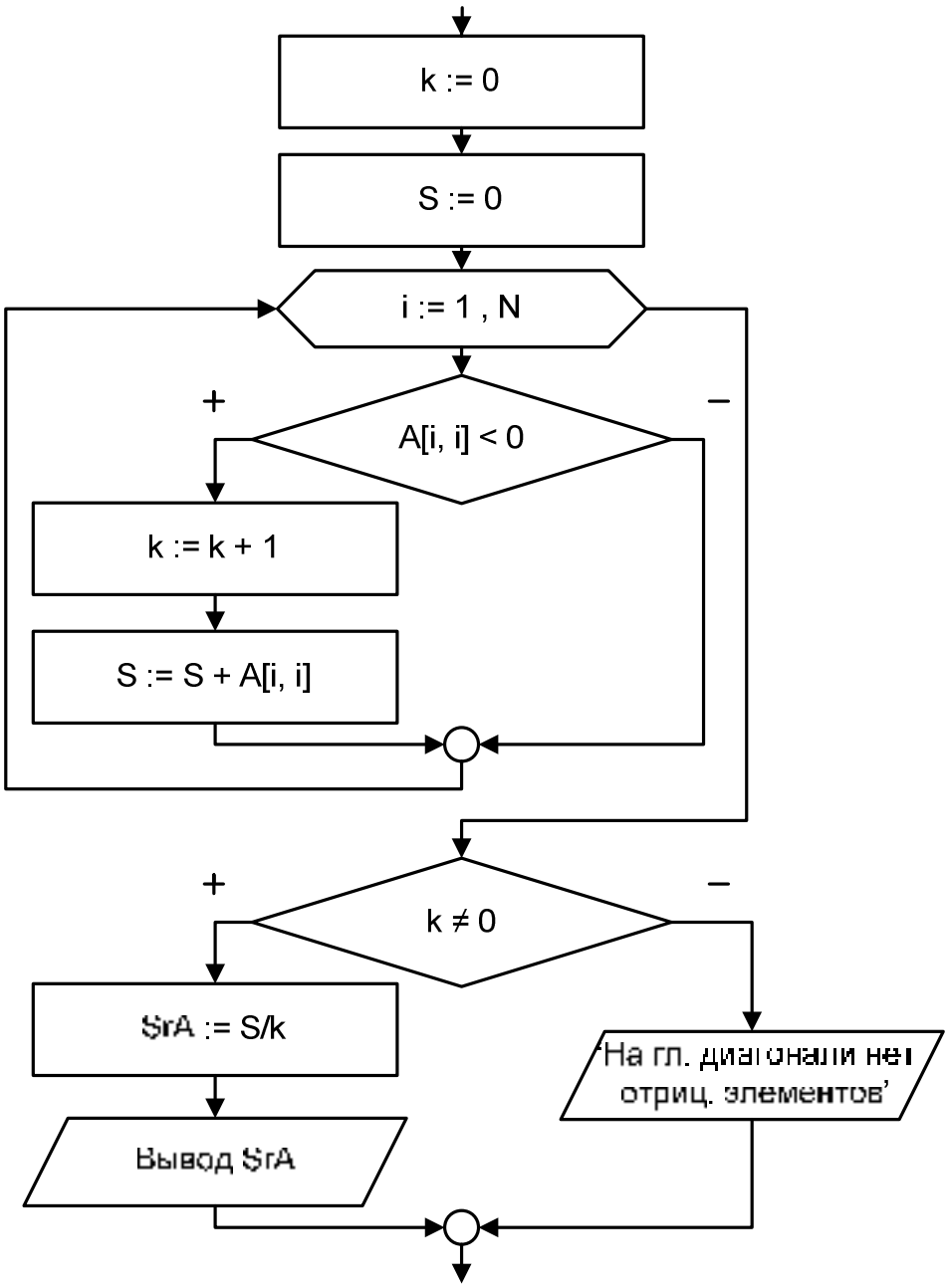


Рис. 5.18 Среднее арифметическое отрицательных элементов гл. диагонали

А теперь рассмотрим такую задачу:

ПРИМЕР

Обменять элементы главной и побочной диагоналей местами. Обмен происходит следующим образом:

вход: $A=$

4	2	8	9
3	6	0	1
6	7	8	0
5	3	2	1

выход: $A=$

9	2	8	4
3	0	6	1
6	8	7	0
1	3	2	5

Алгоритм простой (рис. 5.19):

```
for i:=1 to N do
  begin
    buf := A[i,i];
    A[i,i] := A[i ,N-i+1];
    A[i ,N-i+1] := buf;
  end;
```

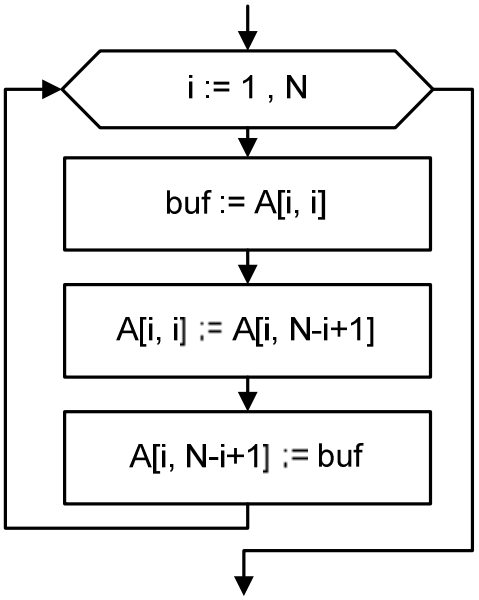


Рис. 5.19. Обмен диагоналей

Далее к особенным элементам стоит отнести верхний и нижний треугольники. Нижним треугольником называют главную диагональ и элементы под нею. Верхним треугольником называют главную диагональ и элементы над нею. Треугольники в матрице показаны на рис. 5.20.

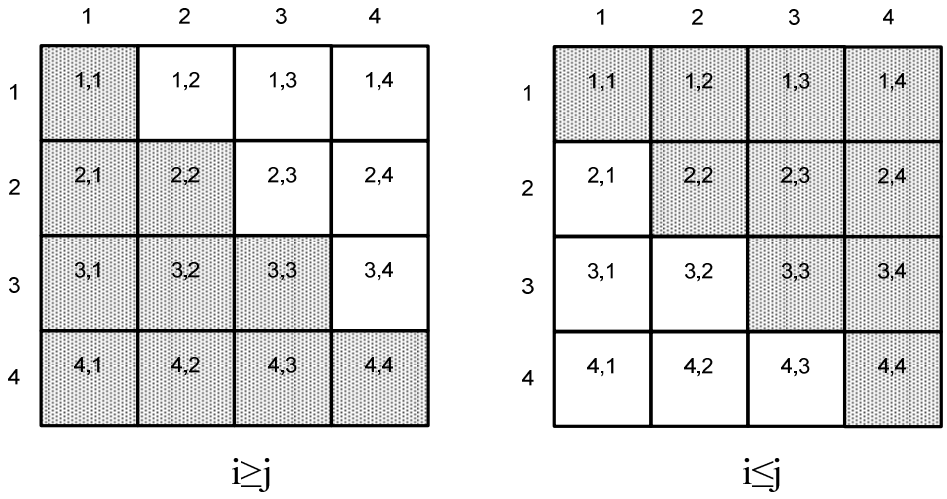


Рис. 5.20. Нижний и верхний треугольники в матрице

Условие нахождения элемента в нижнем треугольнике такое: ***i≥j***. Для верхнего треугольника неравенство обратное: ***i≤j***.

Кроме элементов над и под главной, выделяют также элементы над и под побочной диагональю. Условие нахождения над побочной диагональю такое: ***j<N-i+1***, а под побочной такое: ***j>N-i+1***.

Для обработки элементов из треугольников матриц можно пользоваться двумя способами. Первый заключается в просмотре всех элементов с последующей проверкой условия принадлежности индексов элемента тому или иному треугольнику. Такой способ более надежен, однако зря расходует системные ресурсы, просматривая всю матрицу целиком. Треугольник – это примерно половина матрицы. Для более рационального использования ресурсов можно иначе задать границы изменения циклов. Однако, в этом случае больше вероятность риска ошибиться.

ПРИМЕР

Посчитать количество нулей в нижнем треугольнике матрицы.

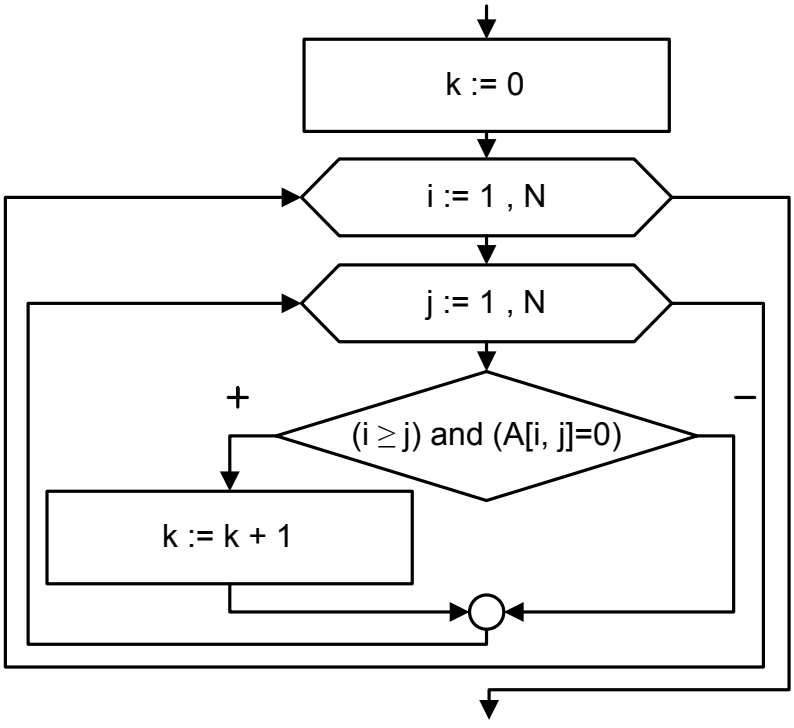


Рис. 5.21. Обработка нижнего треугольника с условием на индексах

Способ с анализом условия на индексы (рис. 5.21):

```
k := 0;
for i:=1 to N do
  for j:=1 to N do
    if (i>=j) and (A[i, j]=0) then
      k := k+1;
```

Способ с изменением границ циклов (рис. 5.22):

```
k := 0;
for i:=1 to N do
  for j:=1 to i do
    if A[i, j]=0 then
      k := k+1;
```

Далее рассмотрим более сложную задачу на двумерные массивы целиком (от и до).

ПРИМЕР

Заменить все нулевые элементы квадратной матрицы значением максимума среди элементов над побочной диагональю.

вход: $A = \begin{pmatrix} 4 & 2 & 8 & 9 \\ 3 & 0 & 0 & 1 \\ 6 & 7 & 8 & 0 \\ 5 & 3 & 2 & 0 \end{pmatrix}$ *выход*: $A = \begin{pmatrix} 4 & 2 & 8 & 9 \\ 3 & 6 & 6 & 1 \\ 6 & 7 & 8 & 6 \\ 5 & 3 & 2 & 6 \end{pmatrix}$

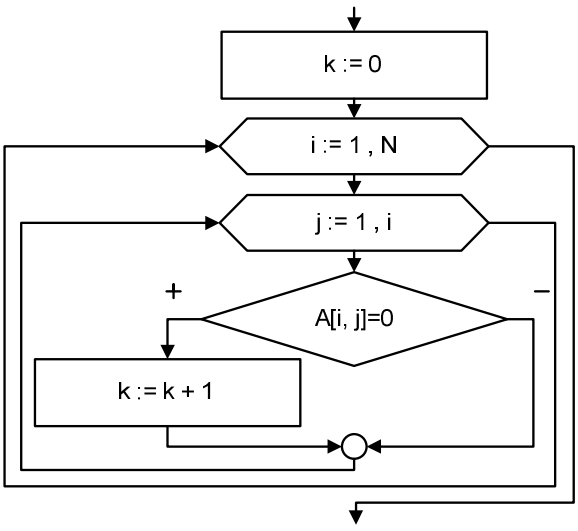


Рис. 5.22. Обработка нижнего треугольника с изменением границ циклов

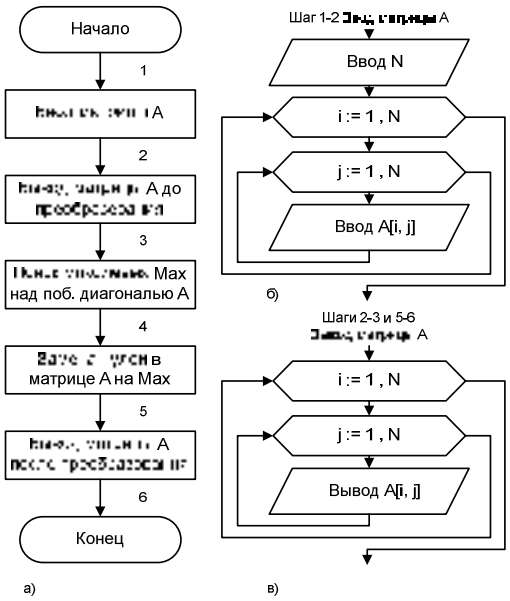


Рис. 5.23. Общая блок-схема с пошаговой детализацией отдельных действий

```
program ZeroMax;
const L=10;
type T2Mx = array[1.. L,1..L] of integer;
var A: T2Mx;
    i,j,N,M:byte;
    Max:integer;
begin
  {шаг 1-2: ввод матрицы}
  writeLn('Введите размерность матрицы:');
  readLn(N);
  for i:=1 to N do
    for j:=1 to N do
      begin
        write('A[',i,',',j,']=');
        readLn(A[i,j]);
      end;

  {шаг 2-3: вывод матрицы до преобразования}
  writeLn('Матрица до преобразования:');
  for i:=1 to N do
    begin
      for j:=1 to N do
        write(A[i,j]:4);
    writeLn;
    end;

  {шаг 3-4: поиск максимума}
  Max:=A[1,1];
  for i:=1 to N-1 do
    for j:=1 to N-i do
      if A[i,j]>Max then
        Max:=A[i,j];
  writeLn('Max =', Max);

  {шаг 4-5: замена}
  for i:=1 to N do
    for j:=1 to N do
      if A[i,j]=0 then
        A[i,j] := Max;

  {шаг 5-6: вывод матрицы после преобразования}
  writeLn('Матрица после преобразования:');
  for i:=1 to N do
    begin
      for j:=1 to N do
       write(A[i,j]:4);
      writeLn;
    end;
end.
```

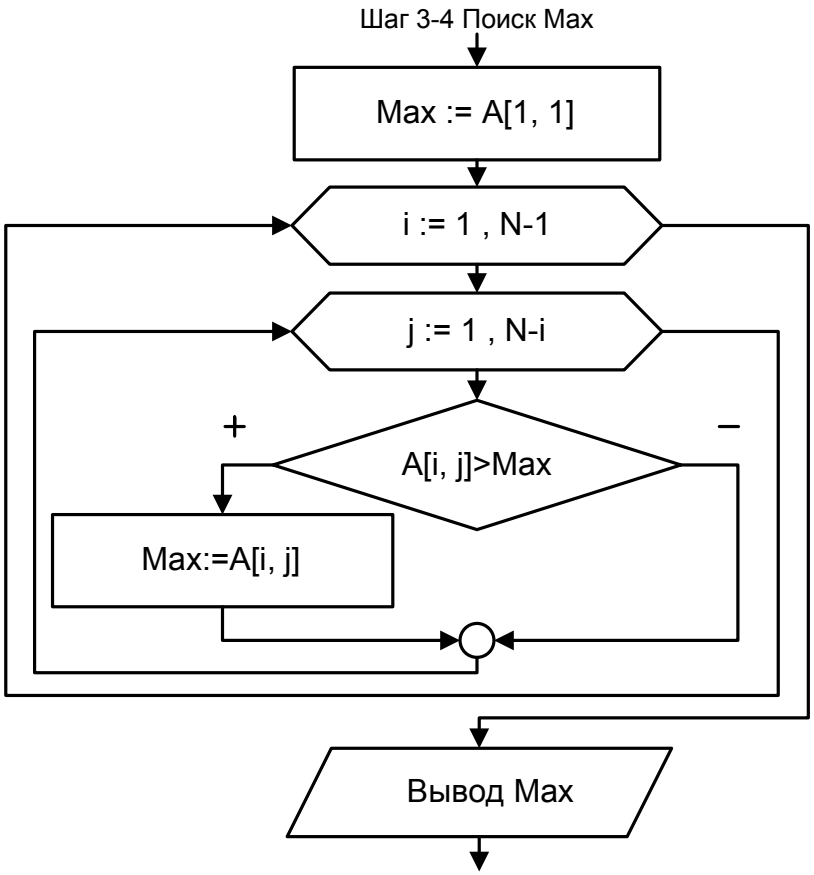


Рис. 5.22 Поиск максимума над побочной диагональю

Стоит отметить, что помимо квадратных матриц достаточно часто используют другие, более экзотические их виды. Это, например, треугольные (т. е. такие матрицы у которых число элементов в строке/столбце зависит от того в каком столбце/строке оно содержится). Есть еще разреженные матрицы, т. е. такие, у которых не все ячейки заполнены элементами и т. д.

Среди прочего, может быть интересен алгоритм для решения такой задачи (хотя он и не относится к квадратным матрицам):

ПРИМЕР

Удалить из матрицы строку и столбец содержащие максимум всей матрицы.

Вот что требуется сделать:

вход :

1	2	3	4	5
6	7	8	30	9
10	11	12	13	14
15	16	17	18	19

, *Imax* = 2, *Jmax* = 4;

выход:

1	2	3	5
10	11	12	14
15	16	17	19

Блок-схема алгоритма без программной реализации показана на рис. 5.26. Здесь не приводится стандартный поиск максимума. Считается, что координаты максимума ***Imax*** и ***Jmax*** были найдены ранее.

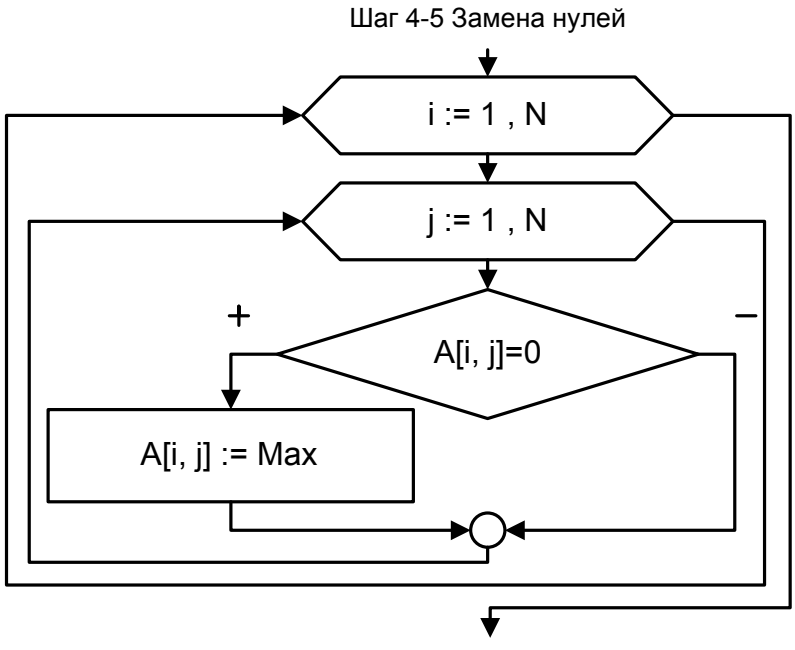


Рис. 5.23. Замена нулей найденным максимумом

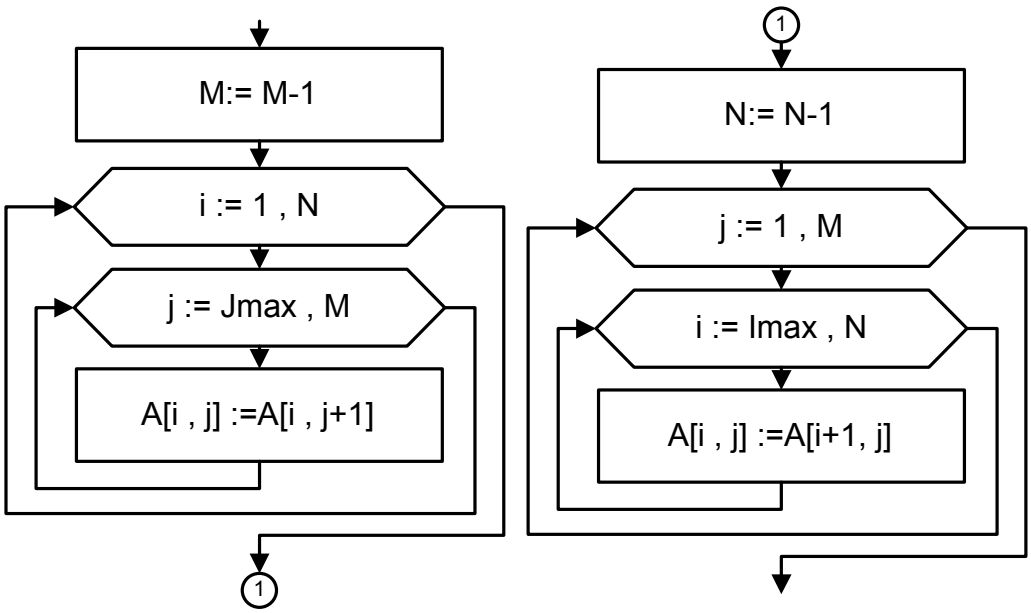


Рис. 5.24. Удаление строки и столбца из матрицы

5.5 Линейная алгебра и матрицы

Матрицы широко применяются в математике и технике, а потому автоматизация процесса основных алгоритмов применяемым к матрицам будет весьма полезна. Подобно тому, как мы рассматривали основы линейной алгебры для одномерных массивов (векторов), рассмотрим основные операции для двумерных (матриц). В основе работы всех алгоритмов на матрицах лежат поэлементные операции.

Сложение двух матриц

По определению, складывать можно только матрицы одного размера, в результате получается новая матрица такого же размера, т. е.:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1M} \\ b_{21} & b_{22} & \cdots & b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N1} & \cdots & b_{NM} \end{bmatrix} =$$

$$= \left[\begin{array}{c|c|c|c} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1M}+b_{1M} \\ \hline a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2M}+b_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline a_{N1}+b_{N1} & a_{N1}+b_{N2} & \cdots & a_{NM}+b_{NM} \end{array}\right] = \left[\begin{array}{c|c|c|c} c_{11} & c_{12} & \cdots & c_{1M} \\ \hline c_{21} & c_{22} & \cdots & c_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline c_{N1} & c_{N1} & \cdots & c_{NM} \end{array}\right]$$

Здесь ситуация аналогична ситуации с одномерными массивами, т.е. прямое копирование объекта при помощи оператора присваивания «:=» возможно только при полном совпадении типов исходного объекта и объекта-приемника.

Соответственно, в коде языка *Pascal* это выглядит следующим образом:

```
for i:=1 to N do
  for j:=1 to M do
     C[i,j] := A[i,j]+B[i,j];
```

Умножение на скаляр

Умножение на скаляр (на число) происходит поэлементно:

$$k\mathbf{A} = k\left[\begin{array}{c|c|c|c} a_{11} & a_{12} & \cdots & a_{1M} \\ \hline a_{21} & a_{22} & \cdots & a_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline a_{N1} & a_{N1} & \cdots & a_{NM} \end{array}\right] = \left[\begin{array}{c|c|c|c} ka_{11} & ka_{12} & \cdots & ka_{1M} \\ \hline ka_{21} & ka_{22} & \cdots & ka_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline ka_{N1} & ka_{N1} & \cdots & ka_{NM} \end{array}\right],$$

что алгоритмически выглядит так:

```
for i:=1 to N do
  for j:=1 to M do
     A[i,j]:=k*A[i,j];
```

Нормировка матрицы

Что касается процедур нормировки, то для матриц их существует достаточно много. Самая простая – нормировка на максимум:

$$\mathbf{A}_{norm} = \frac{\mathbf{A}}{\max A} = \begin{bmatrix} \frac{a_{11}}{\max A} & \frac{a_{12}}{\max A} & \cdots & \frac{a_{1M}}{\max A} \\ \frac{a_{21}}{\max A} & \frac{a_{22}}{\max A} & \cdots & \frac{a_{2M}}{\max A} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{a_{N1}}{\max A} & \frac{a_{N2}}{\max A} & \cdots & \frac{a_{NM}}{\max A} \end{bmatrix},$$

на языке *Pascal* алгоритм такой:

```
Max := A[1,1];
for i:=1 to N do
  for j:=1 to M do
    if A[i,j]>Max then
      Max:=A[i,j];
for i:=1 to N do
  for j:=1 to M do
    A[i,j]:=A[i,j]/Max;
```

Транспонирование

Транспонирование – процесс замены строк в матрице столбцами. Т.е., например, если исходная матрица имеет вид:

$\mathbf{A} = \begin{bmatrix} 1 & 3 & 8 \\ 0 & 4 & 7 \end{bmatrix}$, то транспонированная будет: $\text{Б}^T = \begin{bmatrix} 1 & 0 \\ 3 & 4 \\ 8 & 7 \end{bmatrix}$.

В общем виде это можно записать так:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix}, \Rightarrow \mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{N1} \\ a_{12} & a_{22} & \cdots & a_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1M} & a_{2M} & \cdots & a_{NM} \end{bmatrix}$$

Алгоритм обмена таков:

```
for i:=1 to N do
  for j:=1 to i do
    begin
      buf := A[i,j];
      A[i,j] := A[j,i];
      A[j,i] := buf;
    end;
```

Матричное умножение

Матричное умножение важная операция, при перемножении матриц имеет значение порядок следования, т.е. $\mathbf{AB} \neq \mathbf{BA}$. Перемножение матриц происходит по принципу «*строка на столбец*», для этого необходимо, чтобы число столбцов *Ma* в первой матрице равнялось числу строк *Nb* во второй:

$$\mathbf{C} = \mathbf{AB} = \left[\begin{array}{c|c|c|c} a_{11} & a_{12} & \cdots & a_{1M} \\ \hline a_{21} & a_{22} & \cdots & a_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline a_{N1} & a_{N1} & \cdots & a_{NM} \end{array}\right] \left[\begin{array}{c|c|c|c} b_{11} & b_{12} & \cdots & b_{1M} \\ \hline b_{21} & b_{22} & \cdots & b_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline b_{N1} & b_{N1} & \cdots & b_{NM} \end{array}\right] =$$

$$= \left[\begin{array}{c|c|c|c} \sum_{j=1}^{MaNb} a_{1j}b_{j1} & \sum_{j=1}^{MaNb} a_{1j}b_{j2} & \cdots & \sum_{j=1}^{MaNb} a_{1j}b_{jMb} \\ \hline \sum_{j=1}^{MaNb} a_{2j}b_{j1} & \sum_{j=1}^{MaNb} a_{2j}b_{j2} & \cdots & \sum_{j=1}^{MaNb} a_{2j}b_{jMb} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline \sum_{j=1}^{MaNb} a_{Naj}b_{j1} & \sum_{j=1}^{MaNb} a_{Naj}b_{j2} & \cdots & \sum_{j=1}^{MaNb} a_{Naj}b_{jMb} \end{array}\right]$$

частный случай перемножения:

$$\mathbf{A} = \left[\begin{array}{c|c|c} 1 & 1 & 1 \\ \hline 2 & 2 & 2 \end{array}\right] \quad \mathbf{B} = \left[\begin{array}{c|c|c|c} 2 & 3 & 1 & 1 \\ \hline 4 & 5 & 1 & 1 \\ \hline 6 & 7 & 1 & 1 \end{array}\right] \quad \mathbf{C} = \mathbf{AB} = \left[\begin{array}{c|c|c|c} 12 & 15 & 3 & 3 \\ \hline 24 & 30 & 6 & 6 \end{array}\right]$$

На языке *Pascal* алгоритм таков:

```
for i:=1 to Na do
    for j:=1 to Mb do
      begin
        C[i,j]:=0;
        for k:=1 to MaNb do
            C[i,j]:=C[i,j]+A[i,k]*B[k,j];
      end;
```

Здесь *MaNb* – размерность матриц, вдоль которых происходит свертка (в результирующей матрице этой размерности нет). Важное

условие возможности свертки матриц, как отмечалось выше: *MaNb*= *Ma=Nb*.

Определитель матрицы

Один из наиболее простых для реализации методов расчета определителя матрицы основан на *методе исключения Гаусса*. Суть его сводится к тому, что исходная матрица преобразуется к диагональному виду. Т.е., например, к виду верхней треугольной, что означает равенство нулю всех элементов под главной диагональю. Для треугольной матрицы определитель считается очень просто: он равен произведению элементов стоящих на главной диагонали.

Метод исключения Гаусса основывается на факте что любые строки или столбцы в матрице можно складывать между собой, умножая на произвольный коэффициент.

Сначала домножаем первую строку на соответствующий коэффициент для каждой строчки ниже, вычитая полученные значения из текущей. Таким образом, обнуляем все элементы под элементом a_{11}. Далее домножаем вторую строку на нужные коэффициенты для каждой строчки ниже второй для последующего ее вычитания. Этим добиваемся, чтобы под элементом a_{22} стояли все нули. Так продолжаем до тех пор, пока все элементы под главной диагональю не будут обнулены.

$$\det\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix} =$$

$$= \det \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} - a_{11}\frac{a_{21}}{a_{11}} & a_{22} - a_{12}\frac{a_{21}}{a_{11}} & \cdots & a_{2M} - a_{1M}\frac{a_{21}}{a_{11}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} - a_{11}\frac{a_{N1}}{a_{11}} & a_{N1} - a_{12}\frac{a_{N1}}{a_{11}} & \cdots & a_{NM} - a_{1M}\frac{a_{N1}}{a_{11}} \end{bmatrix} =$$

$$= \det \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ 0 & a_{22} - a_{12}\frac{a_{21}}{a_{11}} & \cdots & a_{2M} - a_{1M}\frac{a_{21}}{a_{11}} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{N1} - a_{12}\frac{a_{N1}}{a_{11}} & \cdots & a_{NM} - a_{1M}\frac{a_{N1}}{a_{11}} \end{bmatrix} =$$

и так далее, пока не будет получена верхняя треугольная матрица.

Представленный алгоритм подразумевает, что на главной диагонали элементы не равны нулю. Если будем составлять алгоритм для более общего случая, то это исключение придется учитывать. Делается это очень просто. Нужно попытаться переставить строку с нулем на главной диагонали с любой строкой ниже, которая, встав на место текущей, не будет обладать данным недостатком. Следует напомнить, что перестановка строк эквивалентна умножению определителя на –1.

Итак, алгоритм расчета определителя, предполагающий отсутствие нулей на главной диагонали, следующий:

```
for k:=1 to N-1 do
  for i:=k+1 to N do
    begin
      buf:=A[i,k];
      for j:=k to N do
        A[i,j]:=A[i,j]-A[k,j]*buf/A[k,k];
    end;
det:=1;
for k:=1 to N do
  det:=det*A[k,k];
```

6. ПОДПРОГРАММЫ

6.1 Иерархия. Черный ящик. Подпрограмма

Сознание человека устроено таким образом, что восприятие окружающей действительности происходит по принципам подобия. Это значит, например, что научившись некоторым базовым операциям, мы впоследствии опираемся на полученный опыт, пытаясь применить его к новой ситуации. Человек воспринимает мир *иерархически*. Отчасти это связано с особенностью способности мыслить методами формальной логики используя язык, с помощью которого человек общается с другими людьми.

Законы иерархии предполагают наличие в воспринимаемом объекте различных уровней с четкими законами подчинения. Именно по законам иерархии наиболее часто строятся схемы управления людскими коллективами. Т. е. такие схемы, которые предполагают наличие начальника сверху, нескольких начальников чуть ниже, подчиняющихся главному. У каждого из них, соответственно, есть свои подчиненные и т.д. (рис. 6.1). Одна из самых жестких систем такого типа – это система военной субординации. Важным свойством систем подобных изображенной на рис. 6.1 является *самоподобие*. Самоподобие позволяет для задания экземпляра системы описать лишь порядок взаимодействия между родительским и подчиненным модулем, а далее эти свойства распространить на все урони иерархии.

Программирование, как известно, тоже является средством управления. Только здесь в качестве подчиненных выступают не люди, а информация. Иерархия при программировании строится по схеме аналогичной рис. 6.1. Следует обратить внимание на отсутствие связей между блоками на одном иерархическом уровне.

Еще одним важным понятием на пути к определению подпрограмм является понятие *черного ящика*. ***Кибернетический черный ящик*** – такое

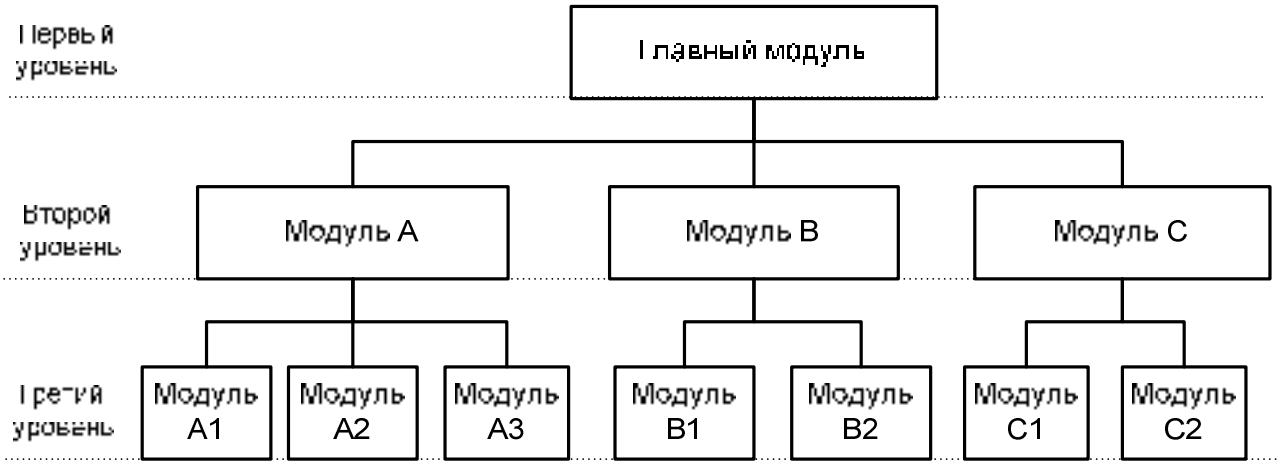


Рис. 6.1. Иерархическая организация

средство обработки информации, которое скрывает структуру своего внутреннего устройства от окружающих его объектов, не являющихся непосредственно подчиненными ему. Все что известно про черный ящик управляющему модулю – это как правильно подать входную и как забрать обработанную информацию. Если рассматривать схему, на Рис. 6.1, то главный модуль ничего не будет знать об устройстве своих подчиненных модулей A, B и C. И уж естественно ничего про A1, A2, A3, B1, B2, C1, C2. Более того, он даже не будет знать об их существовании. Далее, например, модуль A будет находиться в неведении относительно устройства A1, A2, A3, он будет знать только о факте их существования. Про соседние B, B1, B2 и C, C1, C2 ему ничего не будет известно. При таких существенных ограничениях, накладываемых на модули, возникает резонный вопрос о методах их взаимодействия между собой.

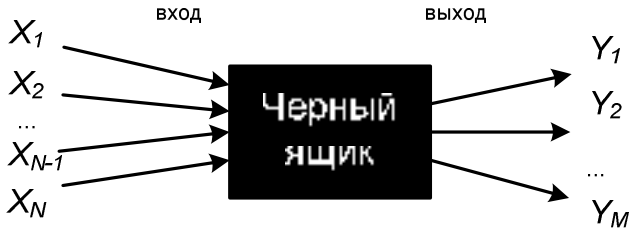


Рис. 6.2. Черный ящик

Для взаимодействия пары модулей находящихся на соседних уровнях иерархии существует так называемый *интерфейс*. ***Интерфейс*** – набор правил позволяющих организовать взаимодействие между парой систем.

Нужно иметь в виду, что правила взаимодействия между системами могут быть не только алгоритмическими, но выраженными в иной материальной или нематериальной форме. Чаще всего это некоторая условная конструкция и *протокол* ее работы[22]. Достаточно распространенной практикой является выделение в информационных потоках интерфейса *входных данных* и *данных на выходе*. Иногда черный ящик называют системой типа «*вход-выход*», в этом случае его изображение может таким как на рис. 6.2. Множество X называют входом черного ящика, а множество Y его выходом. Две системы называют *согласованными по интерфейсу*, если выходы первой можно совместить со входами второй.

При программировании можно столкнуться с ситуацией, когда одни и те же действия необходимо производить несколько раз над однотипными объектами. Например, ввести две матрицы и найти в них максимальный элемент. Для того, чтобы не писать один и тот же алгоритм несколько раз, используют *подпрограммы*.

Подпрограмма – это снабженный заголовком внутренний программный блок, расположенный в разделе описаний внешнего программного блока или программы. Назначение подпрограмм – изменение внешней по отношению к ним программной обстановки.

Подпрограмма описывается один раз и может быть затем неоднократно *вызвана* в разделе операторов программы или другой

[22] Простыми примерами такой конструкции могут быть интерфейсы USB в компьютере, интерфейсы силовой сети зданий (вилка и розетка + протокол (синусоидальное напряжение 50 Гц со среднеквадратичным значением 220 В)) и т. д.

подпрограммы. *Вызов* подпрограммы (т. е. ее реальное использование) приводит к выполнению входящих в нее операторов. После их выполнения работа программы продолжается с оператора, который следует непосредственно за вызовом подпрограммы, в вызывающем блоке.

При вызове подпрограмма может получать исходные данные от вызывающего блока и, при необходимости, возвращать ему результат работы.

В *Pascal* подпрограммы бывают двух видов: *процедуры* и *функции*. Разница между ними достаточно условна[23]. Суть этой разницы сводится к различию методов работы с ними.

Использование подпрограмм позволяет вывести процесс программирования на качественно иной уровень. Так как подпрограммы, по сути, являются независимыми блоками, то их разработку можно вести последовательно, шаг за шагом. Более того, различные участки кода, объединенные в общие библиотеки можно поручать разным программистам. Это позволяет более четко выделить сферы ответственности и разбить процесс программирования во времени и по степени сложности. Кроме того, выделив в подпрограммы многократно повторяющиеся действия, можно получить существенное сокращение программного текста, чем достигается более высокая наглядность и меньшая загруженность памяти ЭВМ.

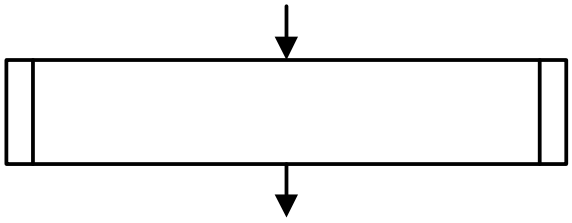


Рис. 6.3. Графическое обозначение
блока вызова подпрограммы

[23] Более того, в большинстве современных языков программирования вообще нет деления на процедуры и функции (например, в *C* существуют только функции, а процедурой можно назвать функцию не возвращающую значения (void)). Даже в *Pascal* имеется возможность настройки компилятора таким образом, чтобы не существовало различий между процедурами и функциями.

Как процедуры, так и функции, являясь алгоритмами написанными пользователем языка программирования *Pascal*, и имеющими уникальное имя на своем уровне иерархии в программе, на блок-схемах изображаются блоком предопределенного процесса, имеющего вид рис. 6.3.

6.2 Подпрограммы в языке Pascal

Использование подпрограмм вносит иерархическую зависимость в алгоритм решения задачи, т.е. одна часть программы подчиняется другой, также соблюдается принцип вложенности, то есть любой блок может содержать внутренние блоки. Поэтому рекомендуется придерживаться следующей последовательности описаний для каждого блока:

- заголовок блока;
- описание констант;
- описание типов;
- описание переменных;
- внутренние блоки;
- тело алгоритма блока.

Таким образом, структура у подпрограмм такая же, как и у основной программы, за исключением того, что в подпрограмме нельзя описывать список используемых библиотек, а после операторов подпрограммы стоит ***end*** с точкой с запятой, тогда как программа заканчивается ***end*** с точкой.

Как отмечалось выше, в языке *Pascal* два вида подпрограмм: *процедуры* и *функции*.

Функция – подпрограмма языка *Pascal*, реализующая некоторый алгоритм, результатом которого является формирование некоторого единственного значения. Обращение к функции происходит через ее имя. Функция всегда возвращает, как минимум, один параметр, причем

возвращение параметра происходит через операцию присваивания, путем помещения имени функции справа от этой операции. Функции *Pascal* очень похожи на обычные функции, используемые в математике. Часть функций уже изначально встроена в базовый набор языка и рассматривалась ранее. Это, например, ***sin(x)***, ***cos(x)***, ***ln(x)***, ***pi***, ***frac(x)***, ***trunc(x)***. Видно, что у перечисленных функций есть аргумент, но не у всех. Так, функция ***pi*** его не имеет, поскольку число *Пи* является фундаментальной константой и ни от чего не зависит. Хоть здесь они и не представлены, однако существуют функции с числом аргументов более одного.

Стандартные функции языка *Pascal* представляют собой некоторый алгоритм, реализующий последовательность действий с максимальной оптимальностью. Так, например, арифметико-логическое устройство микропроцессора не может напрямую реализовать вычисление функции ***sin***, потому, на первом этапе ***sin*** раскладывается в ряд, что позволяет, используя только базовые арифметические операторы (+, –, *, /), вычислять сложные тригонометрические функции. Естественно, что не нужно заботиться о программировании этих рядов, поскольку, в виду частоты использования, их уже описали авторы базовых библиотек *Pascal*. Потому работа с этими функциями достаточно проста. Однако, если бы в базовый набор не был встроен ***sin***, то, вспомнив, что разложение синуса в ряд имеет вид:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \ldots$$

можно было бы написать функцию:

```
function sin1(x:real):real;
var xx,S:real;
    i:byte;
```

```
    f:longInt;
begin
  S:=x;
  f:=1;
  xx:=x;
  i:=2;
  while i<20 do
  begin
    f:=f*i*(i+1);
    xx:=xx*x*x;
    if i mod 4 = 0 then
      S:=S+xx/f {добавл. в сумму неч. члена с плюсом}
    else
      S:=S-xx/f;{добавл. в сумму неч. члена с минусом}
    i:=i+2;
  end;
  sin1:=S;
end;
```

Здесь для простоты понимания принципа работы функции, мы ограничились двадцатой степенью переменной. Кроме того, здесь учтено, что период повтора знака у членов равен четырем (четные члены в сумму не включаются, однако используются для вычисления нечетных).

Далее рассмотрим порядок обращения к функциям, или, как принято говорить «*вызов*». Совершенно естественной выглядит запись обращения:

y:=sin(x);

z:=exp(y);

в то время, как такая запись бессмысленна:

sin(x) := y; (! не верно!)

exp(y) := z; (! не верно!)

Синтаксис *Pascal* не допускает подобных общений. Всегда имя функции ставится справа от операции присваивания. Единственным исключением является определение значения функции при ее описании в собственном *теле*. Т.е. при описании для задания значения имя функции без параметров ставится слева от операции присваивания.

Описание состоит из *заголовка* и *тела*.

Заголовок выглядит следующим образом:

function *имя_функции* ***(****список формальных параметров****) :*** *тип результата*;

Здесь:

имя_функции – идентификатор пользователя, используемый затем для ее вызова;

список формальных параметров – набор параметров, состоящий для функции, как правило, только из входных переменных. Для каждого формального параметра указывается его тип и способ передачи. Параметры одного типа и с одинаковым способом передачи, перечисляются через запятую, все остальные через точку с запятой.

Результат работы функции *возвращается* в вызывающий модуль через ее имя. Тип результата указывается в заголовке. Чтобы вернуть результат, необходимо чтобы в разделе операторов (теле) функции присутствовал хотя бы один оператор присваивания, который ставит в соответствие имени функции полученное в результате работы значение:

Имя_функции ***:=*** *результат*;

Пример описания заголовка функции может быть таким:

```
function Summa(Const X:T2mx; Const N,M:byte): integer;
```

То, что касается вызова функции, то стоит сказать, что она не является отдельным оператором и может быть использована только:

- в выражениях, в правой части оператора присваивания,
- в составе булевского выражения,
- в списке вывода процедур ***write*** или ***writeLn***.

- в списке фактических параметров любой подпрограммы, если только способ их передачи позволяет это (передача должна быть либо по значению, либо по константной ссылке[24]).

ПРИМЕР

```
S:=Summa(A,Na,Ma) + Summa(B,Na,Ma);
```

или так:

```
If Summa(A,N,M) > Summa(B,K,L) then
  begin
    writeLn('Сумма положительных элементов матрицы A
    больше суммы положительных элементов матрицы B');
    writeLn ('эта сумма равна: ',Summa(A,N,M));
  end;
```

Само описание функции поиска положительных элементов в двумерном массиве может быть таким:

```
function Summa(Const X:T2mx; Const N,M:byte): integer;
var i,j:byte;
    S:integer;
begin
  S:=0;
  for i:=1 to N do
    for j:=1 to M do
      if X[i,j]>0 then
         S:=S+X[i,j];
  Summa:=S;
end;
```

Блок-схема алгоритма этой функции показана на (рис. 6.4).

Необходимо отметить, что обычно функции применяются для выполнения каких-либо математических вычислений, в результате которых получается одно число. Если в результате работы функции происходит изменение каких-либо данных (преобразование массивов и т. д.), то это называется побочным действием функции и таких ситуаций по возможности нужно избегать, т.к. это приводит к нарушению логики использования средств языка *Pascal* (раз есть процедуры, то и пользоваться лучше ими в подходящем случае, чем применять нечто не

[24] Подробнее о способах передачи будет рассказано ниже.

совсем подходящие под ситуацию). Далее подробно рассмотрим процедуры.

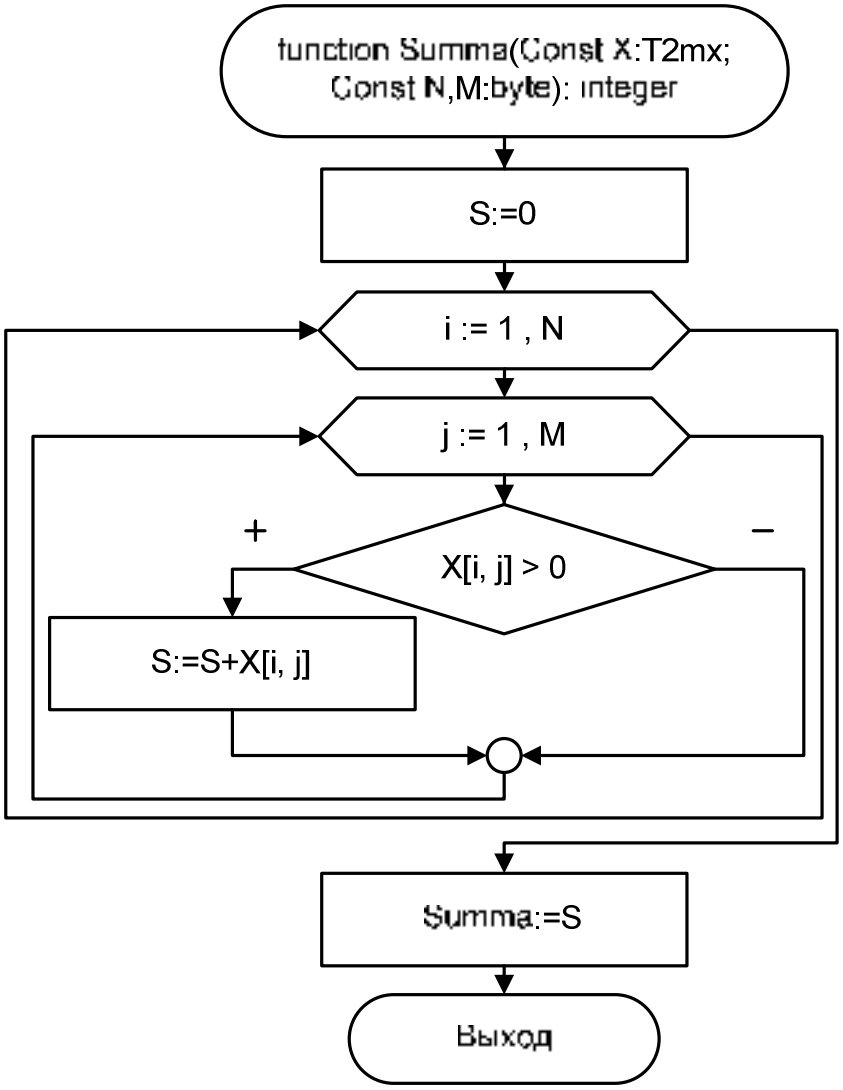


Рис. 6.4. Функция суммы положительных элементов

Процедура – подпрограмма языка *Pascal*, предназначенная для формирования нескольких значений и/или выполнения некоторых действий не связанных напрямую с изменением значений параметров. Процедура описывается в разделе описаний вызывающего ее блока, т.е. в программе или подпрограмме более высокого уровня, и состоит из заголовка, раздела описаний и раздела операторов (тела) процедуры.

Заголовок выглядит следующим образом:

procedure *имя_процедуры* ***(****список формальных параметров****);***

Здесь:

имя_процедуры – идентификатор пользователя, используемый затем для вызова процедуры;

список формальных параметров – набор параметров, состоящий из входных и выходных данных. Для каждого формального параметра указывается его тип и способ передачи. Параметры одного типа и с одинаковым способом передачи, перечисляются через запятую, все остальные через точку с запятой.

ПРИМЕР

```
procedure Vvod(var X:T2mx; Var N,M:byte; Name:char);
```

или так:

```
procedure SqrtMass(const X:T2mx; const N,M:byte);
```

Вызов процедуры – это отдельный оператор, в котором указывается

Имя_процедуры (*список фактических параметров****);***

Опишем полностью процедуры объявленные выше. Процедура ввода двумерного массива (рис. 6.5, *а*):

```
procedure Vvod(var X:T2mx; var N,M:byte; Name:char);
var i,j: byte;
begin
  writeLn('вводим массив ', Name, ', введите N и M');
  readLn(N,M);
  for i:=1 to N do
    for j:=1 to M do
      begin
        write(Name,'[',i,',',j,']=');
        readLn(X[i,j]);
      end;
end;
```

Возведение в квадрат элементов двумерного массива будет таким (рис. 6.5, *б*):

```
procedure SqrMass(const X:T2mx; const N,M:byte);
var i,j: byte;
begin
  for i:=1 to N do
```

```
    for j:=1 to M do
      X[i,j]:=sqr(X[i,j]);
end;
```

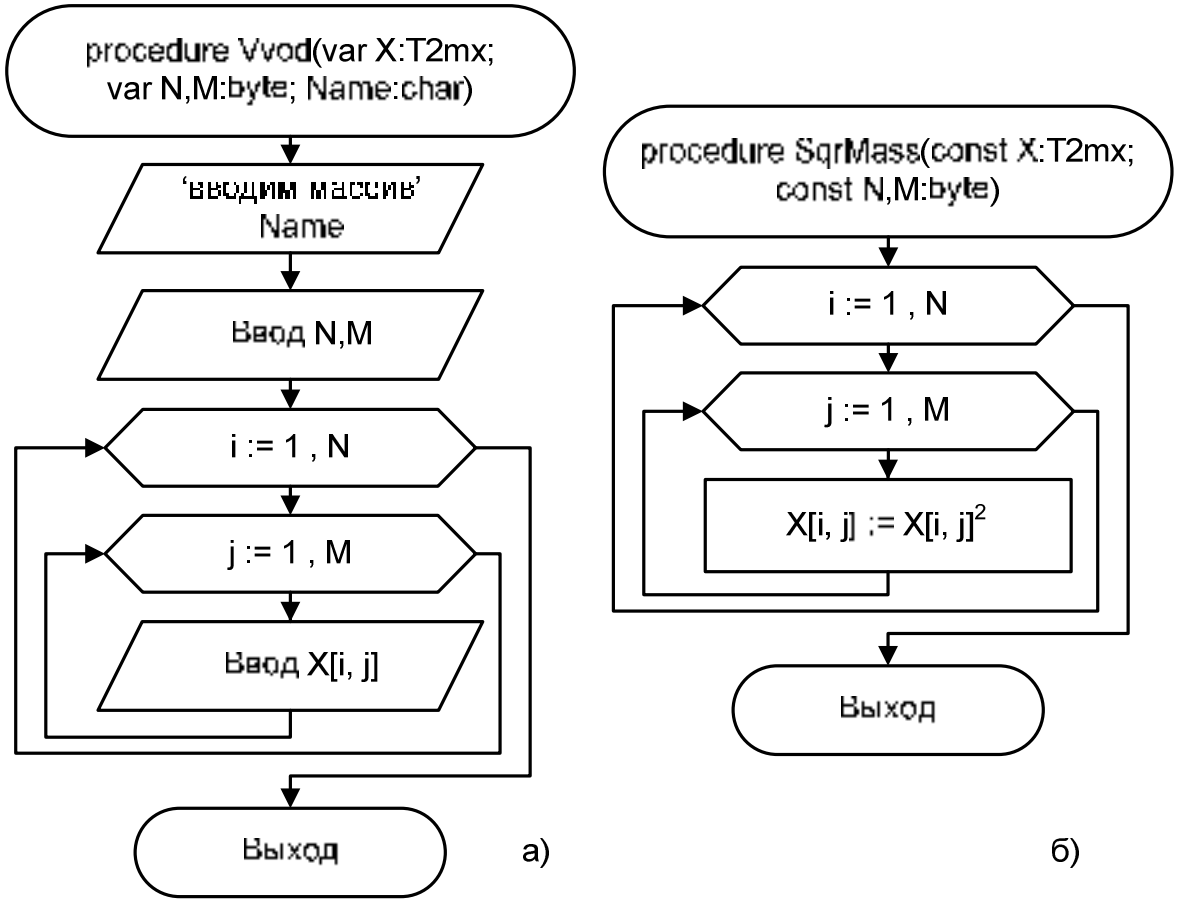


Рис. 6.5 Ввод двумерного массива (*а*) и возведение его элементов в квадрат (*б*)

6.3 Локальные и глобальные идентификаторы

Ранее отмечалось, что главная программа и входящие в ее состав подпрограммы имеют свои разделы описаний, а потому объявленные в этих разделах идентификаторы (константы, типы, переменные) обладают разными свойствами.

Идентификаторы, описанные в подпрограмме, являются ***локальными*** для нее, т. е. работа с ними возможна только внутри этой подпрограммы и внутри вложенных в нее блоков.

Имена, описанные в модулях более высокого уровня, являются ***глобальными*** для всех своих подчиненных. Эти имена могут быть использованы в любом модуле стоящем ниже на иерархической лестнице, а также в исполнительной части самого модуля.

Если объявление глобальных переменных[25] происходит в основной программе, то во время ее работы значения глобальных переменных записываются в область памяти, называемую *сегментом данных* (*статический сегмент*) и доступны постоянно на протяжении всей работы программы. Локальные данные записываются в иную специальную область памяти, называемую *стеком* и доступны только во время работы подпрограммы, в которой они описаны, по завершении работы подпрограммы эти данные стираются.

Основные правила работы с глобальными и локальными переменными:

– локальные переменные доступны внутри блока, в котором они описаны, и во вложенных в него блоках;

– имя, описанное в локальном блоке «закрывает» совпадающее с ним имя из блока более высокого уровня. То есть, если при обработке подпрограммы возникла *коллизия имен* (имена глобальной и локальной переменных совпадают), то обрабатываться будет локальная переменная, до тех пор, пока работа с подпрограммой не закончится. Однако, как говорилось ранее, все глобальные переменные доступны в подпрограмме. Если возникает потребность в обращении к переменной при коллизии имен, то следует полностью указывать ее имя вместе с названием модуля. Делается это так: вначале указывается название модуля (модуль основной программы – это, собственно, название программы, указанное после слова

[25] Здесь под переменной подразумеваются данные, которые могут храниться за любым идентификатором (константой, типом и т. д.); дело в том, что именно переменные являются основным объектом работы большинства алгоритмов.

program), а далее через точку имя переменной (или иной идентификатор), к которому нужно обратиться. Для примера, изображенного на рис. 6.6, полные обращения к переменным такие:

G.X, ***G.Y***, ***G.A.A1.XA1***, ***G.C.YC*** и т.д.

На рис. 6.6 изображена иерархическая структура некоторой условной программы. Эта программа имеет основной глобальный модуль обозначенный как ***G***. В ***G*** объявлены переменные ***X*** и ***Y***, которые являются глобальными по отношению ко всем подчиненным модулям.

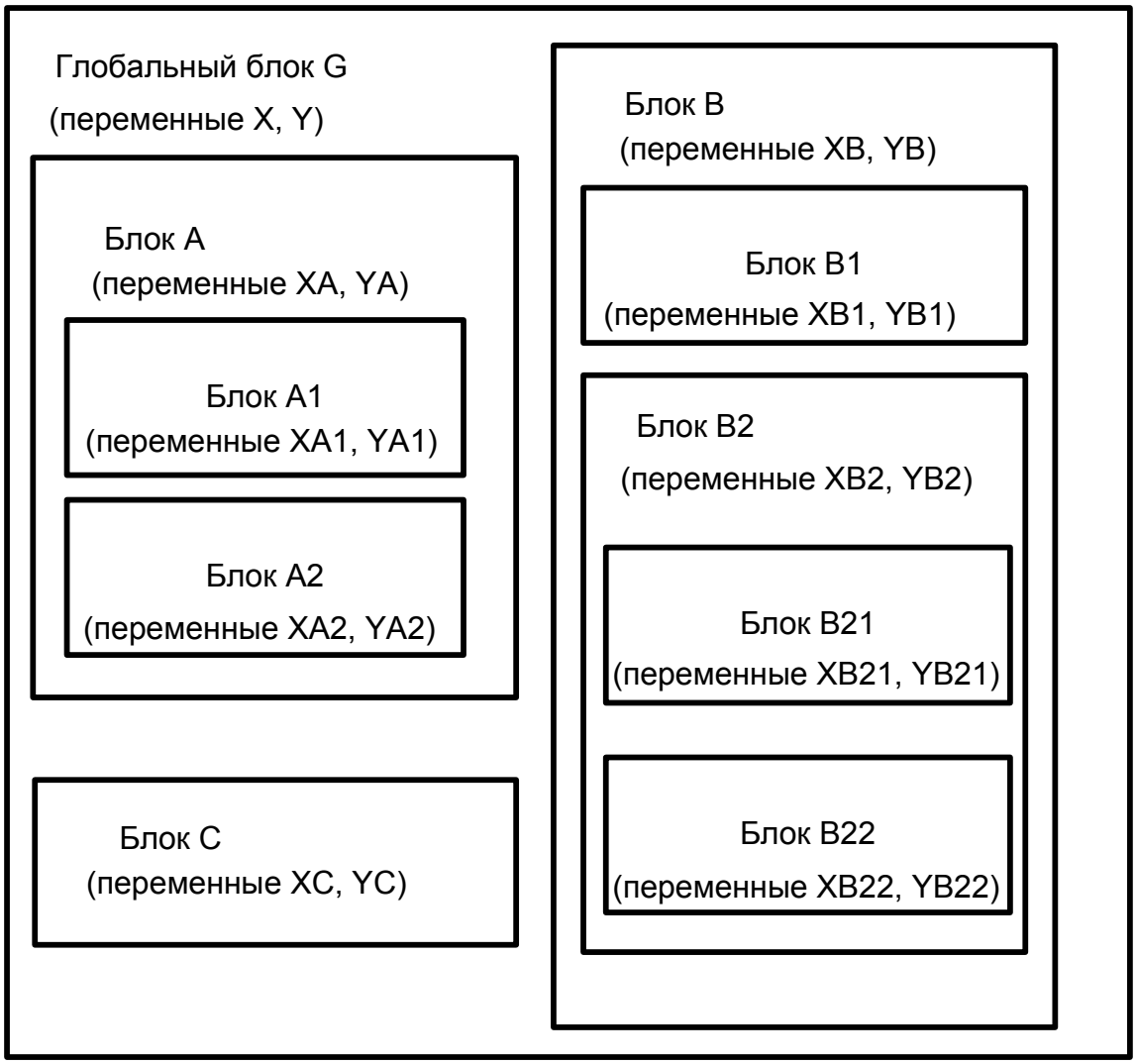


Рис. 6.6. Пример иерархии подпрограмм и переменных

Область видимости этих переменных абсолютная, т.е. в данной программе они доступны из любой точки. А переменные ***XA*** и ***YA*** из блока ***A*** будут видны помимо, собственно блока ***A*** еще и в блоках ***A1*** и ***A2***. Ну а, например, переменные ***XB22*** и ***YB22*** нигде кроме ***B22*** не будут видны, они для ***B22*** являются локальными идентификаторами.

Можно определить локальные идентификаторы процедур как те, которые описываются в разделах ***var***, ***const***, ***type*** и в скобках с параметрами.

Глобальными можно назвать все те идентификаторы, которые используются в основной программе, в том числе и в скобках при вызове процедур.

6.4 Параметры подпрограмм

Все то, что, записывается в скобках сразу после названия подпрограммы, называется ее параметрами. Параметры, по сути, являются тем интерфейсом, с помощью которого данная подпрограмма «общается с внешним миром».

При обмене данными между программой и подпрограммами используется механизм передачи входных и выходных параметров. *Входные параметры* – это исходные для подпрограммы данные, а *выходные* – результат ее работы.

Для того чтобы подпрограмма могла быть использована многократно для разных наборов входных и выходных параметров, используют наборы формальных и фактических параметров.

Формальные параметры – это локальные переменные необходимые для описания алгоритма подпрограммы, они описываются в ее заголовке и используются в собственном разделе операторов. Выше говорилось, что

формальные параметры – все то, что указывается в скобках справа от названия подпрограммы *при ее описании*. Потому описание формальных параметров происходит только один раз.

Вообще говоря, список формальных параметров является необязательной частью заголовка, его наличие зависит от способа обмена информацией процедуры с вызывающим блоком, т.е., например, возможен такой вариант заголовка:

```
Procedure Poisk;
```

В этом случае процедура вызывается просто по имени:

```
Poisk;
```

Однако использование процедур и функций без параметров, как правило, предполагает либо независимость исходных данных от данных глобального модуля, либо использование глобальных переменных (или иных идентификаторов). Если есть возможность не пользоваться глобальными переменными, то лучше ей воспользоваться, поскольку это позволяет повысить автономность подпрограммы и надежность ее алгоритма.

Фактические параметры – это набор данных, в обработке которых и заключается предназначение алгоритма. В момент вызова формальные параметры связываются с фактическими во всей подпрограмме. Другими словами фактические параметры – все то, что указывается в скобках справа от названия процедуры или функции *при ее вызове*. Фактические параметры у подпрограммы могут меняться при каждом вызове, а формальные нет.

Имена формальных и фактических параметров могут совпадать, это не отразится на выполнении программы, но может привести к проблемам при понимании алгоритма работы, поэтому рекомендуется использовать для формальных и фактических переменных разные имена.

Следует отметить, что поскольку параметры представляют собой интерфейс связи между главным модулем и процедурами, то параметры заявленные (формальные) должны соответствовать параметрам фактическим. Критериев такого соответствия принято выделять всего четыре:

- *по количеству*, т.е. количество заявленных и реально используемых переменных должно совпадать;

- *по типу*, т.е. тип заявленных и реально используемых переменных должен совпадать;

- *по порядку следования*, т.е. переменные в описании подпрограммы и при ее вызове должны быть перечислены в одинаковом порядке;

- *по способу передачи*, т.е. статус параметров в главной программе должен быть совместимым с заявленным статусом параметров подпрограммы.

На способах передачи параметров стоит остановиться подробнее.

В зависимости от того, является передаваемый параметр входным или выходным, различают и способ его передачи. Для языка *Pascal* принято выделять три способа передачи:

- *по значению*;

- *по ссылке с правом изменения*;

- *по ссылке без права изменения*.

Основным моментом важным для понимания является усвоение принципов лежащих в основе передачи по значению или по ссылке. Условно разницу этих двух видов передач можно изобразить так, как показано на рис. 6.7.

По значению передаются параметры-значения, являющиеся простыми входными данными, т.е. константами, именами переменных и простыми выражениями. При этом значение передаваемого фактического

параметра копируется в память, отводимую под подпрограмму (стек), и работа с ним осуществляется как с локальной переменной, т.е. его можно изменять, но результат изменений в вызывающий блок передан не будет, а будет удален при завершении работы подпрограммы. В качестве

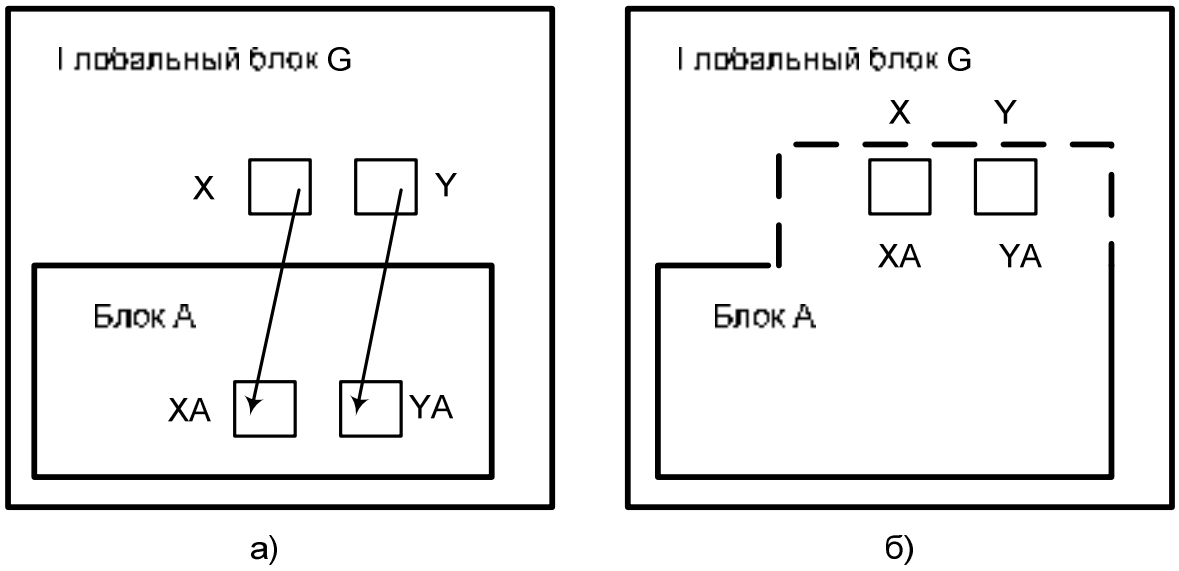


Рис. 6.7. Передача параметров по значению (*а*) и по ссылке (*б*)

начального значения формальный параметр получает текущее значение соответствующего фактического параметра.

Таким образом, параметры-значения можно использовать в тех случаях, когда, например, результат работы процедуры выводится непосредственно на экран и его не надо передавать вызывающему блоку.

Можно сказать, что при передаче по значению в локальном блоке организуется копия переданной переменной. Таким образом, мы имеем сразу две переменные, одна из них – глобальная, а вторая локальная. Это может привести к тому, что при достаточно большом размере этих переменных возникнет дефицит памяти. Кроме того, при завершении подпрограммы локальная копия параметра уничтожается, вместе со всеми остальными переменными подпрограммы. Это может являться причиной

ситуации, когда подпрограмма вроде не содержит синтаксических ошибок и производит модификацию параметра переданного по ссылке, при ее завершении вычисленное значение теряется, принимая значение бывшее при входе в нее. Подобного рода ошибки не всегда удается четко отследить, а потому рекомендуется как можно реже прибегать к передаче параметров по значению. Если нужно запретить подпрограмме модификацию передаваемого параметра, то нужно воспользоваться передачей по ссылке без права изменения.

При описании параметров передаваемых по значению в языке *Pascal* перед их именами в скобках никаких префиксов не ставится. Вот примеры передачи всех параметров по ссылке:

```
procedure Summa (X: Tmatrix; Y,Z: byte);
```

или так:

```
procedure Vivod (X: Tmatrix; Y,Z: byte; MName: char);
```

При передаче по ссылке передается ссылка на сегмент данных, т.е. на область памяти, в которой хранится фактический параметр. По ссылке можно передавать параметры с правом или без права модификации. В зависимости от разрешения на модификацию, различают *параметры-константы* и *параметры-переменные*.

Параметры константы – параметры, переданные по ссылке без права их изменения.

Параметры-константы используются, когда передаются входные данные, являющиеся сложными структурированными переменными (например, массивы). При таком способе передачи изменение формального параметра запрещено, если переданный параметр будет изменяться, компилятор выдаст ошибку. Для использования этого способа передачи, в списке формальных параметров перед параметром-константой ставится префикс ***const***. Вот примеры объявления параметров-констант:

```
procedure Vivod(const X:Tmatrix; const Y,Z:byte;
                const MName:char);
```

или так:

```
function Max(const X:Tmatrix; const Y,Z:byte);
```

Параметры-переменные – параметры, переданные по ссылке с правом их изменения. Параметры-переменные используются для передачи выходных значений процедур. При изменении параметров-переменных изменяется соответствующий фактический параметр, таким образом, изменения сохранятся и после завершения работы подпрограммы. Для использования этого способа передачи, в списке формальных параметров перед параметром-переменной ставится префикс ***var***. Не стоит передавать по ссылке с правом изменения параметры, о которых точно известно, что в данной процедуре они не меняются. Соблюдение этого правила поможет предотвратить возможные ошибки.

Примеры заголовков процедур с параметрами-переменными:

```
procedure Vvod (var X: Tmatrix; var Y,Z: byte);
```

или

```
procedure Resize(var A:T1mass; var N: byte);
```

Некоторые типы данных при передаче могут иметь только формат параметров-переменных. К таковым, например, относятся переменные файлового типа.

Как говорилось ранее, при передаче необходимо соблюдение соответствия между формальными и фактическими параметрами по способу передачи. Чтобы понять, что это такое, есть смысл рассмотреть пример в котором данное соответствие не выполняется.

формальный набор:

```
procedure Pr1(var A:integer; var B: integer);
```

фактический набор

```
Pr1(10, S);
```

Очевидно несоответствие по способу передачи, поскольку для переменной ***A*** заявлена передача по ссылке с правом изменения. Для ***A*** при вызове подпрограммы в стеке не будет резервироваться место и потому возникает неопределенность с местом хранения этой переменной, ведь при вызове процедуры вместо переменной указана константа ***10***.

Еще одно скрытое несоответствие может быть если, например, указанный фактический параметр ***S*** будет являться не переменной, а, скажем именованной константой, тогда при попытке изменения его при фактическом вызове программы опять возникнет проблема доступа к ***S***. В процедуре сказано, что на месте ***S*** находится переменная (***var B: integer***), а в основной программе – константа. Подобные несоответствия выявляются еще на этапе компиляции. В случае если перед заявленным параметром указать префикс ***const*** (параметр-константа), то *Pascal* в случае несоответствия не выдаст ошибку, а поместит значение в стек и продолжит работу.

При передаче структурированных типов (файлов, массивов, записей и т.д.) необходимо создавать новый тип в разделе описаний типов ***type***.

ПРИМЕР

Ввести пять квадратных матриц $A_{Na\times Na}$, $B_{Nb\times Nb}$, $C_{Nc\times Nc}$, $D_{Nd\times Nd}$, и $E_{Ne\times Ne}$.

Для этого можно использовать процедуру ввода матрицы с набором формальных параметров состоящего из матрицы ***X*** и размерности матрицы ***Y***, и переменной символьного типа, используемой для передачи имени матрицы в процедуру. При этом процедура ввода будет описана один раз, а вызываться пять раз (рис. 6.8).

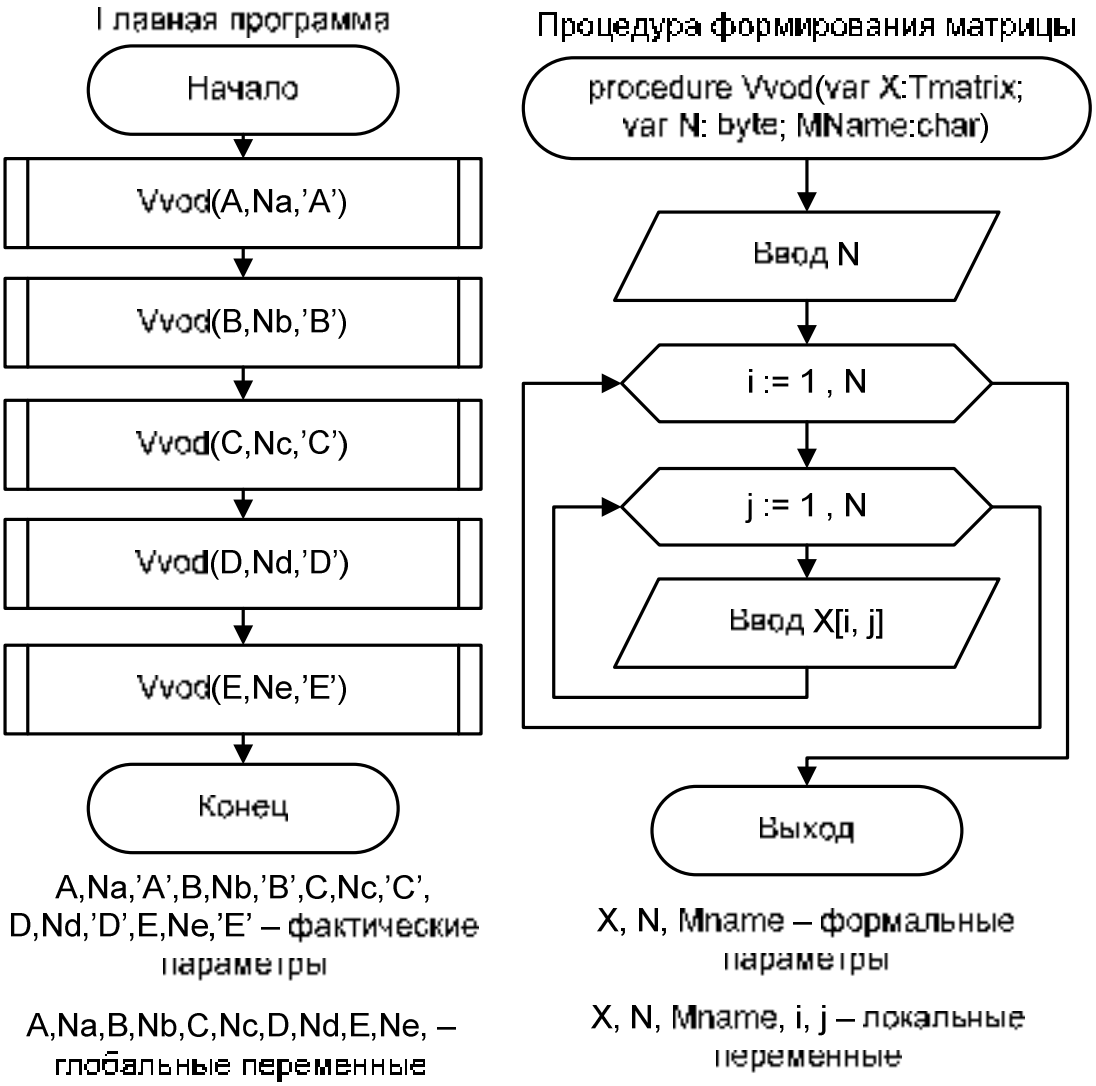


Рис. 6.8. Ввод пяти разных матриц при помощи одной процедуры

```
Program PR2;

Type Tmatrix = array [1..10,1..10] of integer;
Var A,B,C,D,E:Tmatrix; {Объявление глоб. переменных}
    Na,Nb,Nc,Nd,Ne:byte;
procedure Vvod(var X:Tmatrix; Var N: byte; MName:char);
          {X,Y,MName – формальные параметры}
          {X,Y,MName – они же локальные переменные}
var i,j: byte; {i,j - чисто локальные переменные}
begin
  writeLn('Введите размерность матрицы ',MName);
  readLn(N);
  for i:=1 to N do
  for j:=1 to N do
    begin
      writeLn(Mname,'[',i,',',j,']=');
      readLn(X[i,j]);
    end;
```

```
end;
{основная программа}
begin
  Vvod(A,Na,'A'); {вызов процедуры Vvod}
  Vvod(B,Nb,'B'); {с фактическими параметрами}
  Vvod(C,Nc,'C'); {A,Na,'A' , B,Nb,'B' и т.д.}
  Vvod(D,Nd,'D'); {A,B,C,D,E и Na,Nb,Nc,Nd,Ne - это}
  Vvod(E,Ne,'E'); {глобальные переменные}
end.
```

6.5 Примеры решения задач

Для иллюстрации возможности применения подпрограмм рассмотрим несколько задач.

ПРИМЕР

Даны три матрицы $A_{N\times M}$, $B_{K\times L}$ *и* $C_{R\times H}$ *найти произведение их максимальных элементов.*

Для начала, как обычно, составим тестовый пример:

$$\textit{Вход}:\ A=\begin{array}{|c|c|c|c|}\hline 1 & -3 & 5 & 3\\ \hline 8 & 1 & 0 & -2\\ \hline -8 & 11 & 2 & -10\\ \hline\end{array},\ B=\begin{array}{|c|c|c|}\hline 4 & -2 & 4\\ \hline 5 & 7 & 0\\ \hline -1 & 2 & 4\\ \hline 3 & 9 & 10\\ \hline\end{array},\ C=\begin{array}{|c|c|}\hline 2 & 5\\ \hline 1 & 4\\ \hline\end{array}$$

$$MaxA=11 \qquad MaxB=10 \qquad MaxC=5$$

$$\textit{Выход}:\ P=MaxA\cdot MaxB\cdot MaxC=550$$

Итак, для решения задачи необходимо ввести матрицу ***A***, ввести матрицы ***B*** и ***C***, найти максимальный элемент матрицы ***A***, найти максимальный элемент матрицы ***B***, найти максимальный элемент матрицы ***D***, вычислить произведение найденных максимумов ***P*** и вывести его на экран. Очевидно, что для решения понадобятся процедура ввода матрицы и процедура поиска максимального элемента матрицы.

Решение задачи начинается с составления блок-схем процедур. На этом этапе необходимо определить наборы формальных параметров и

способы их передачи для каждой процедуры, так как эти данные указываются в первом блоке блок-схемы.

Назовем процедуру ввода матрицы словом ***Vvod***. Для работы этой процедуры понадобится матрица и ее размерность (число строк и столбцов). При этом все три параметра должны передаваться как параметры-переменные. Для корректного ввода в процедуру передадим также буквенное обозначение матрицы, для чего воспользуемся переменной ***Mname*** символьного типа. Чтобы не было коллизии имен между формальными и фактическими параметрами, назовем формальную матрицу ***X***, а ее размерности ***Y***, ***Z***.

Процедура поиска максимального элемента пусть будет ***Maximum***. Для процедуры, собственно, матрица и ее размерность будут входными параметрами, а значение максимального элемента – выходным. Таким образом, здесь будет использован набор из четырех формальных параметров, три из которых будут передаваться как параметры-константы, а один – как параметр-переменная.

Переменные ***i***, ***j***, используемые в качестве переменных в циклах-счетчиках – чисто локальные, так как их значения носят вспомогательный характер и нужны только в пределах процедур для перебора элементов матрицы.

Затем составляется блок-схема основной программы (рис. 6.9, *а*), процедур ввода (рис. 6.9, *б*) и поиска максимума (рис. 6.10).

При вызове процедур используется три набора фактических параметров, первый для матрицы ***A***, второй для матрицы ***B***, и третий – для

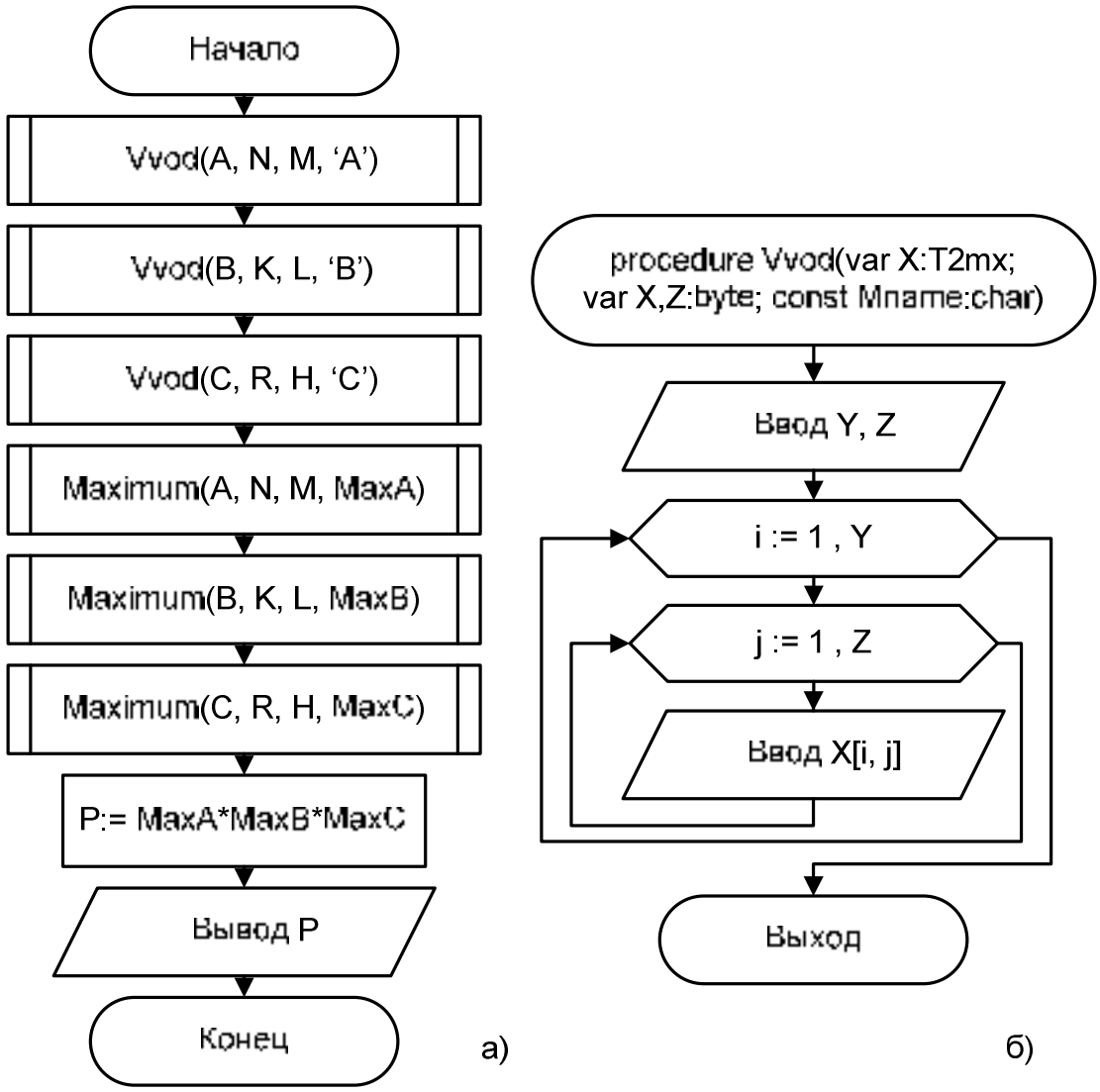


Рис. 6.9. Основная программа и процедура ввода матрицы

матрицы ***C***. Наборы фактических параметров соответствуют набору формальных параметров по количеству, типу, порядку следования и способу передачи.

В процедуру ***Vvod*** сначала передается матрица ***A***, ее размерность ***N***, ***M***, и буква ***'A'***, затем матрица ***B***, ее размеры ***K***, ***L*** и буква ***'B'***, после этого идет третий вызов процедуры, на этот раз уже с матрицей ***C***, ее размерностью ***R***, ***H*** и названием ***'C'*** в качестве фактических параметров.

Распространенная ошибка передавать все двенадцать параметров в одном блоке.

Процедура ***Maximum*** также должна вызываться трижды, но чтобы не потерять значение максимального элемента матрицы ***A***, нужно при вызове

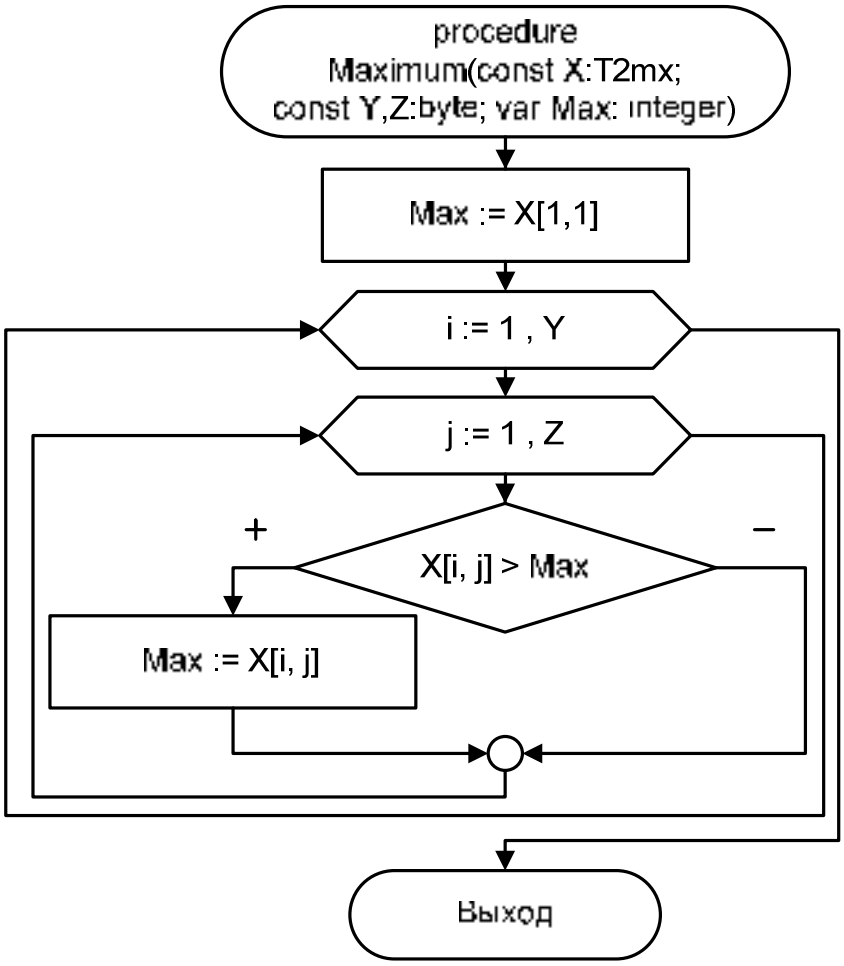


Рис. 6.10. Процедура поиска максимума в матрице

процедуры взять три различные переменные: ***MaxA***, ***MaxB*** и ***MaxC*** для максимальных элементов матриц ***A***, ***B*** и ***C*** соответственно.

Для того чтобы найти произведение максимальных элементов, дополнительную процедуру использовать не нужно, так как действие будет производиться один раз. При этом переменная ***P*** будет глобальной, поскольку она используется в основной программе.

Последний этап решения задачи – составление программы по полученной блок-схеме. Следует учесть, что поскольку по условию

матрицы разного размера, то для описания нового типа данных ***T2mx*** необходимо использовать число строк и столбцов, которое удовлетворило бы размеры всех трех матриц. Можно взять, например, ***T2mx=array[1..15, 1..15] of integer***.

```
program PP_1;
type T2mx = array[1..15, 1..15] of integer;
var  A,B,C: T2mx;
     N,M,K,L,R,H: byte;
     P, MaxA, MaxB, MaxC: integer;

{******Процедура ввода матрицы*******}
procedure Vvod(var X:T2mx; var Y,Z:byte;
           const MName:char);
var i,j: byte;
begin
  WriteLn('Введите размерность матрицы ',MName);
  ReadLn(Y, Z);
  for i:=1 to Y do
    for j:=1 to Z do
      begin
        Write(Mname,'[',i,',',j,']=');
        ReadLn(X[i,j]);
      end;
  end;

{****** Процедура поиска максимума ******}
procedure Maximum(const X: Tmatrix; const Y,Z: byte;
           var Max: integer);
var i,j: byte;
begin
  max:=x[1,1];
  for i:=1 to Y do
    for j:=1 to Z do
      if x[i,j]>max then
        max:=x[i,j];
end;

{****** Основная программа ******}
Begin
  Vvod (A,N,M,'A');
  Vvod (B,K,L,'B');
  Vvod (C,R,H,'C');
  Maximum (A,N,M,MaxA);
  Maximum (B,K,L,MaxB);
  Maximum (C,R,H,MaxC);
  P:=MaxA*MaxB*MaxC;
```

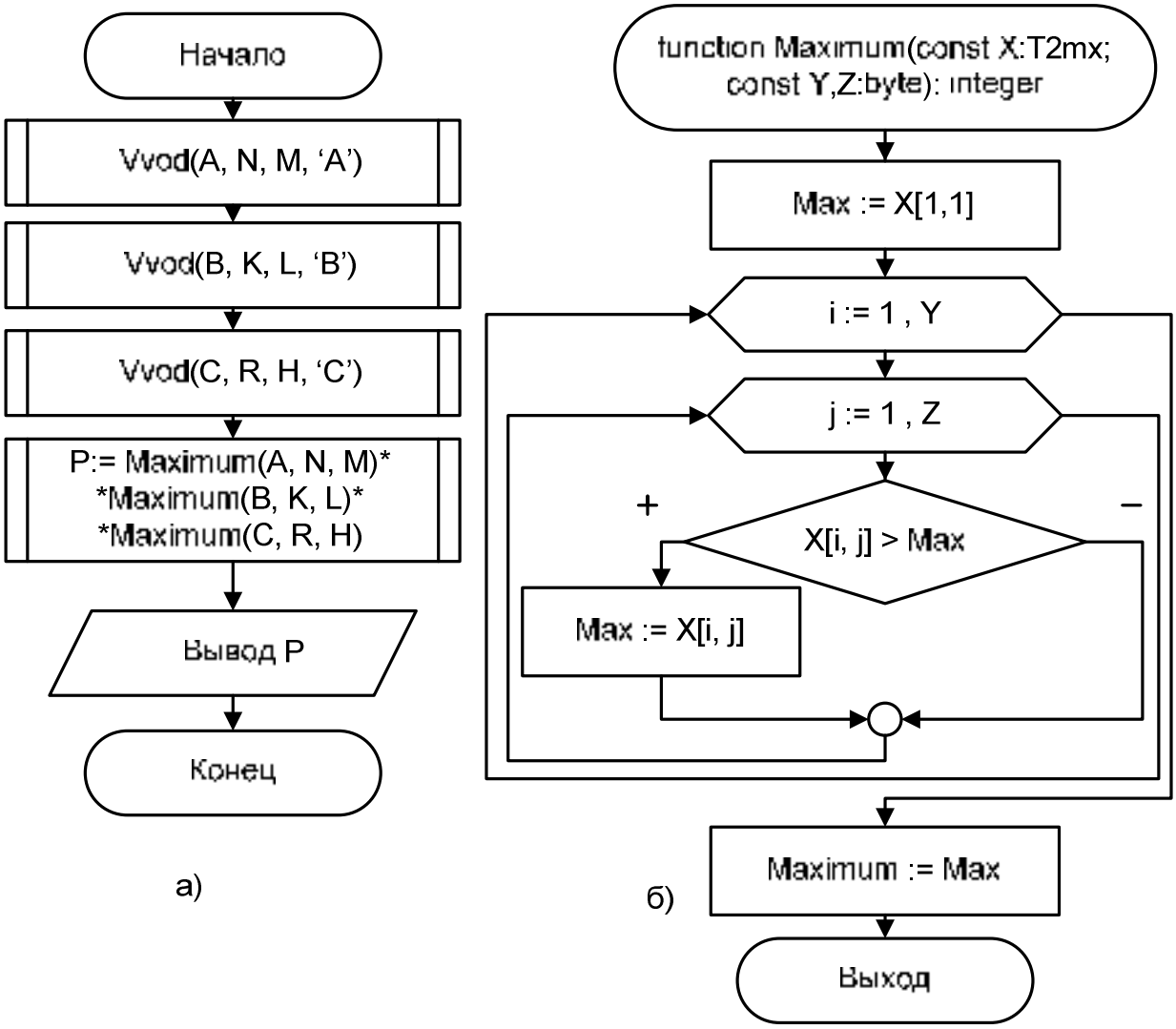


Рис. 6.11. Решение задачи с помощью функции

```
  writeln ('P=',P);
End.
```

Теперь решим ту же задачу, но только используя для поиска максимума не процедуру, а функцию, поскольку именно функция лучше подходит для решения, так как от каждой матрицы надо получить, по сути, одно единственное число, а функция, как раз, возвращает единственное значение. Блок-схема несколько изменится. Поменяется основная программа (рис. 6.11, *а*) и процедура поиска максимума преобразуется в функцию (рис. 6.11, *б*). Блок-схема ввода матрицы останется прежней (рис. 6.10, *а*). Соответственно, программа может быть записана таким образом:

```
program PP_2;
type
     T2mx = array[1..15, 1..15] of integer;
var
     A,B,C: T2mx;
     N,M,K,L R,H: byte;
     P: integer;
{******Процедура ввода матрицы*******}
procedure Vvod(var X: T2mx; var Y,Z: byte; MName:char);
var i,j: byte;
begin
  writeLn('Введите размерность матрицы ',MName);
  readLn(Y, Z);
  for i:=1 to Y do
    for j:=1 to Z do
      begin
        write(Mname,'[',i,',',j,']=');
        readLn(X[i,j]);
      end;
end;
{****** Функция поиска максимума ******}
function Maximum(const X:T2mx;
                    const Y,Z:byte): integer;
var i,j: byte;
     Max:integer;
begin
  Max:=x[1,1];
  for i:=1 to Y do
    for j:=1 to Z do
      if x[i,j]>Max then
        max:=x[i,j];
  Maximum:=max;
end;
{****** Основная программа ******}
begin
  Vvod (A,N,M,'A');
  Vvod (B,K,L,'B');
  Vvod (C,R,H,'C');
  P:=Maximum (A,N,M)*Maximum (B,K,L)*Maximum (C,R,H);
  writeln ('P=',P);
end.
```

Далее решим такую задачу:

ПРИМЕР

Даны две матрицы $\boldsymbol{A}_{Na\times Na}$ *и* $\boldsymbol{B}_{Nb\times Nb}$, *найти сумму отрицательных элементов в каждой из матриц и заменить ей минимальный элемент на главной диагонали в противоположной матрице.*

Входные данные: $A_{3\times3} =$

2	−6	5
−3	3	1
−4	2	4

, $B_{4\times4} =$

2	0	−1	3
−5	−6	7	4
8	−1	2	0
11	−3	−6	10

Промежуточные данные:

SA=–13, ***SB= – 22***

IminA = JminA=1; ***IminB= JminB=2***.

Выходные данные:

$A =$

–22	–6	5
–3	3	1
–4	2	4

,

$B =$

2	0	–1	3
–5	–13	7	4
8	–1	2	0
11	–3	–6	10

Для решения задачи вводим матрицы ***A*** и ***B***, далее находим сумму отрицательных элементов ***SA*** и ***SB*** в каждой из них, находим координаты минимальных элементов на главной диагонали (***IminA*** и ***IminB***) и далее, зная эти координаты, заменяем минимальные элементы полученными суммами в перекрестном порядке.

Решение задачи начинается с составления блок-схемы основной программы и блок-схем процедур. Сначала мы формируем общую последовательность выполнения алгоритма, набор и порядок вызова необходимых процедур.

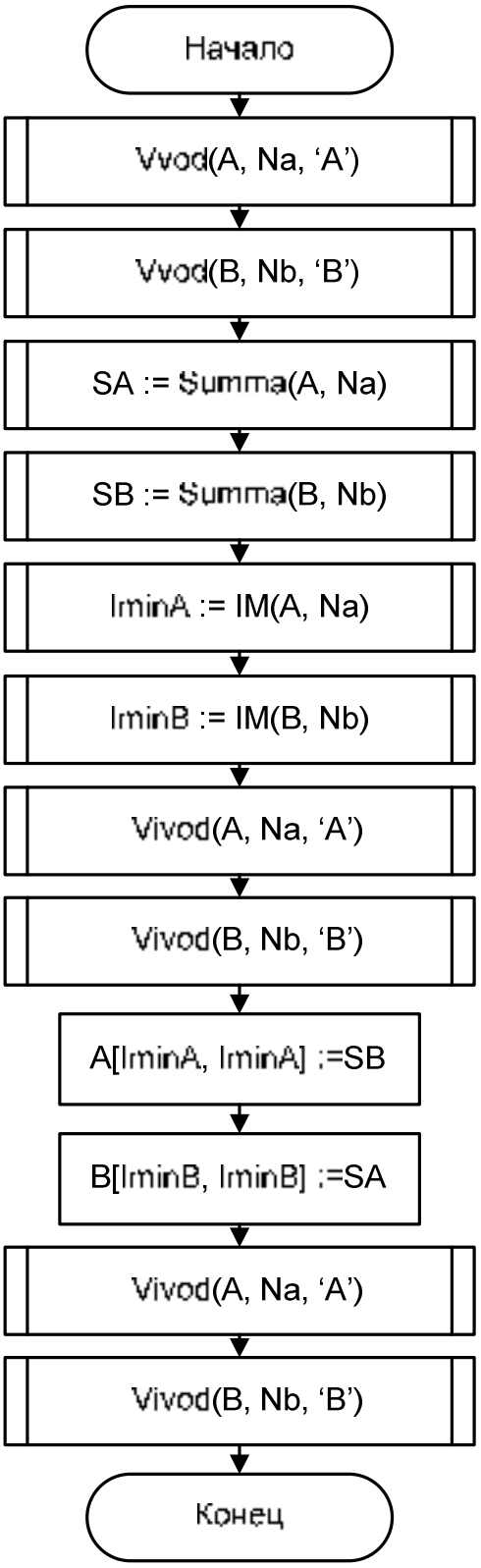


Рис. 6.12 Основная программа

Нужно определиться с набором входных и выходных данных для каждой подпрограммы. На основе этого выбираем формальные параметры и способы их передачи. Далее детализируем алгоритм работы каждой подпрограммы в отдельности.

Полученная блок-схема основной программы показана на рис. 6.12. Следует отметить, что перед модификацией матриц выводим их на экран для того, чтобы иметь возможность проверить корректность введенных данных. Кроме того, дополнительный вывод позволяет наглядно представить исходные значения и потому достаточно легко самостоятельно убедиться в правильности промежуточных и выходных данных. Использование подпрограмм для решения задачи позволяет производить достаточно сложное действие по выводу двумерного массива всего одной строкой.

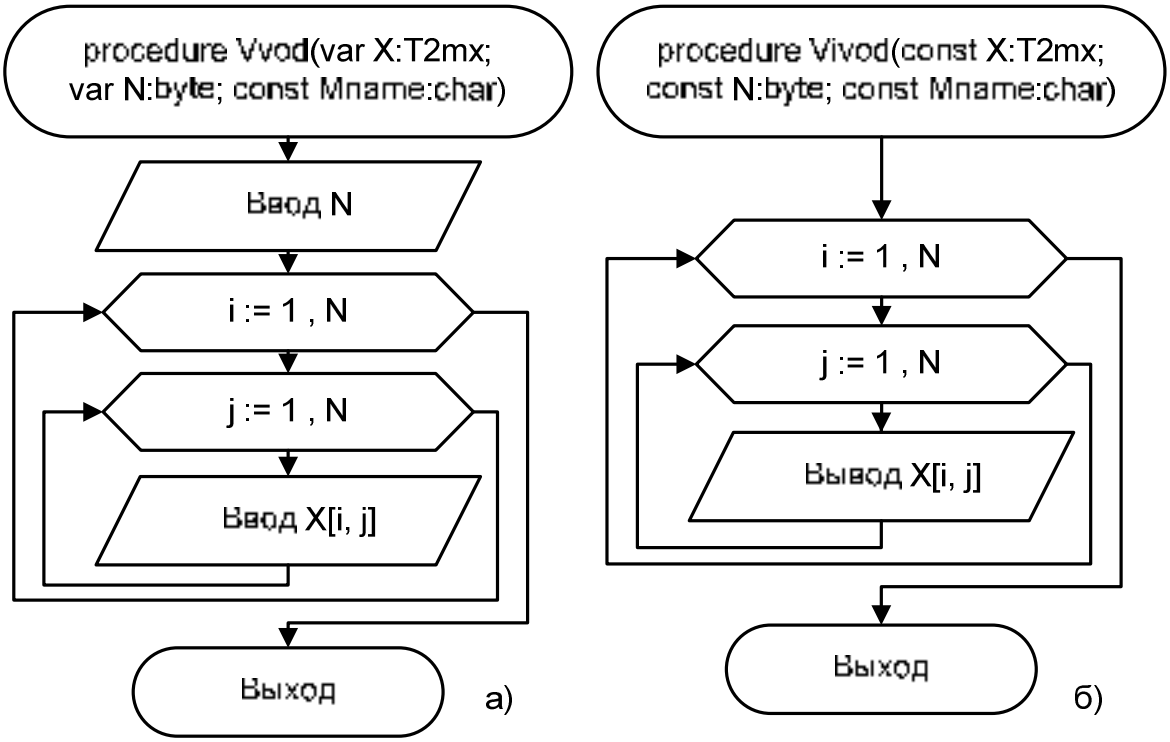


Рис. 6.13. Ввод и вывод квадратной матрицы

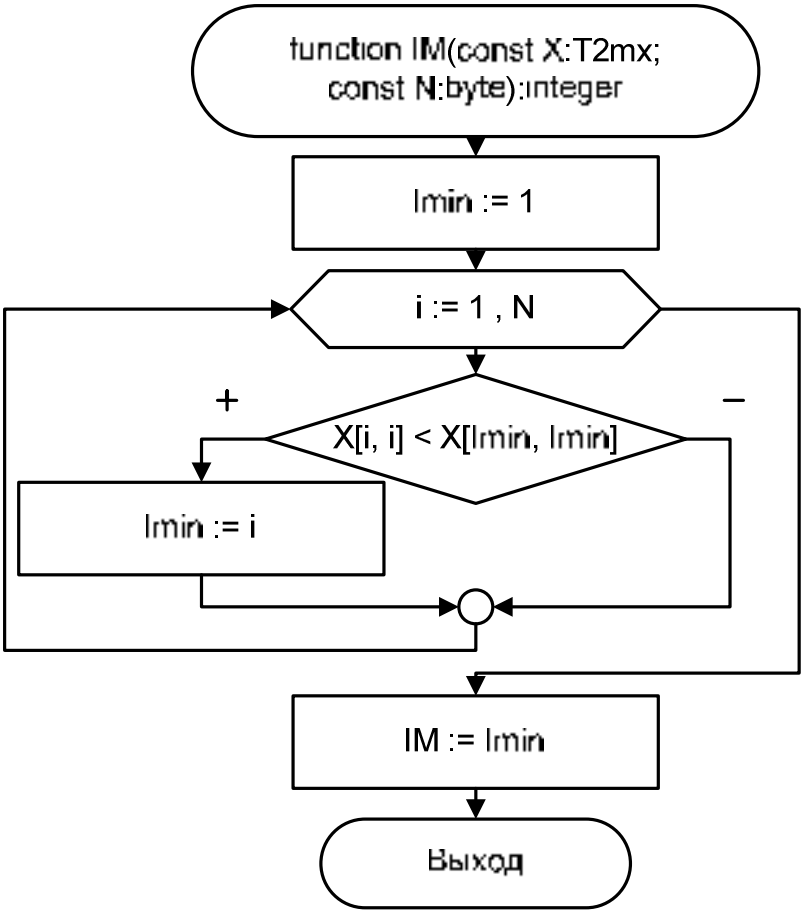


Рис. 6.14 Поиск минимума в квадратной матрице

Замена минимального элемента – одно действие, поэтому для него дополнительную процедуру не создаем.

Процедуры ввода и вывода двумерного массива достаточно стандартны. Единственное отличие будет в том, что матрицы в данной задаче квадратные, значит, в качестве размерности достаточно передавать количество строк.

Для поиска суммы отрицательных элементов воспользуемся функцией ***Summa*** (рис. 6.15), передадим в нее матрицу и ее размерность, так как они изменяться не будут, то передавать их будем как параметры-константы.

Для поиска индекса минимального элемента воспользуемся функцией ***IM*** (рис. 6.14), передадим в нее матрицу и размерность, как параметры-константы. Результат работы функции ***Imin*** – значение индекса минимального элемента главной диагонали матрицы. Для поиска нужен только один цикл, так как индекс строки элементов главной диагонали совпадает с номером столбца.

Теперь составим программу:

```
program PP_3;
type
     T2mx = array[1..15, 1..15] of integer;
var
     A,B: T2mx;
Na,Nb: byte;
     SA,SB,IminA,IminB: integer;

{******Процедура ввода матрицы*******}
procedure Vvod (var X:T2mx; Var N:byte;const MName:char);
var i,j: byte;
begin
  writeLn('Введите размерность матрицы ',MName);
  readLn(N);
  for i:=1 to N do
  for j:=1 to N do
    begin
      write(Mname,'[',i,',',j,']=');
```

```
      readLn(X[i,j]);
    end;
end;


{****** Функция поиска суммы ******}
function Summa(const X:T2mx; const N: byte):integer;
var i,j: byte;
    S:integer;
begin
  S:=0;
  for i:=1 to N do
    for j:=1 to N do
      if x[i,j]<0 then
        S:=S+x[i,j];
end;
{*** Функция поиска индекса минимального элемента ***}
```

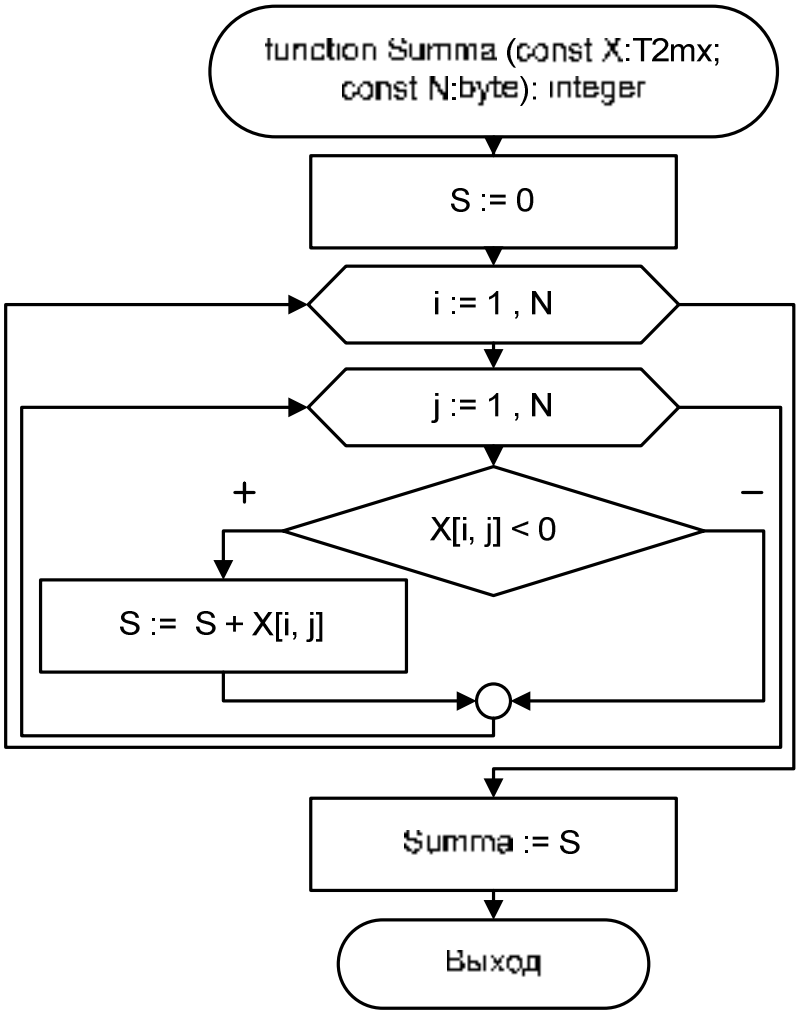


Рис. 6.15. Сумма отрицательных элементов

```
function IM(const X:T2mx; const N:byte): integer;
var i,j,Imin: byte;
begin
  Imin:=1;
  for i:=1 to N do
    if x[i,i]<X[Imin,Imin] then
      imin:=i;
end;
{******Процедура вывода матрицы*******}
procedure Vivod(const X:T2mx; const N:byte;
                    const MName: char);
var i,j: byte;
begin
writeLn ('Вывод матрицы ',MName);
  for i:=1 to N do
    begin
      for j:=1 to N do
        Write(X[i,j]:4);
      writeLn;
    end;
end;
{****** Основная программа ******}
begin
  Clrscr;
  Vvod(A,Na,'A');
  Vvod(B,Nb,'B');
  SA := Summa(A,Na);
  SB := Summa(B,Na);
  IminA := IM(A,Na);
  IminA := IM(B,Nb);
  writeLn('матрицы до преобразования');
  Vivod (A,Na,'A');
  Vivod (B,Nb,'B');
  A[IminA, IminA]:= SB;
  B[IminB, IminB]:= SA;
  writeLn('матрицы после преобразования');
  Vivod (A,Na,'A');
  Vivod (B,Nb,'B');
end.
```

7. ФАЙЛЫ

7.1 Основные определения и объявление файла

Достаточно часто при решении тех или иных задач средствами ЭВМ возникает потребность в хранении и обработке больших объемов однотипных данных. Можно попытаться воспользоваться массивами. Однако это не всегда удается в силу того, что при обработке массивы хранятся в оперативной памяти, которая имеет относительно небольшой объем, а потому бывает нецелесообразно помещать их туда целиком. Кроме того, оперативная память в большинстве ЭВМ не обладает свойством энергонезависимости. Это означает, что при выключении питания компьютера несохраненная информация теряется.

В качестве одного из методов решения вышеобозначенной проблемы были предложены так называемые *файлы*.

Файл – именованная область на внешнем информационном носителе (диске), содержащая данные.

Язык *Pascal* содержит ряд достаточно продвинутых средств для работы с файлами. В *Pascal* файлы принято условно делить на *физические* и *логические*.

Физический файл – собственно файл, который содержится на носителе информации. Одной из важнейших его характеристик является наличие имени в файловой системе, т. е. адреса. Например, ***C:\TP7\EXAMPLE.PAS*** (имя в абсолютной адресации) или ***FF\DDD.DAT*** (имя в относительной адресации).

Логический файл – файловая переменная языка *Pascal*, смысл его в том, чтобы компилятор «знал» какого типа файл записан по определенному адресу и как с ним следует обращаться. Это, по сути, ссылка на физический файл. Кроме того, наличие файловой переменной

облегчает запись программ, делая их более наглядными и выразительными.

Прежде чем начать работать с файлом в *Pascal*, необходимо указать его физическое местоположение. Для связи логического и физического файлов существует процедура ***assign***. Формат у нее следующий:

assign(*Файловая_переменная****,*** *путь к файлу* ***);***

Путь к файлу – выражение строкового типа. Например, для связи файла ***F*** с именем ***C:\Lab_1.dat*** следует ввести следующую команду:

```
assign(F,' C:\Lab_1.dat');
```

Для связи, например, логического файла ***MyFile*** и физического ***MyFile.cas*** следует написать

```
assign (MyFile, 'MyFile.cas');
```

Файлы в *Pascal* бывают нескольких видов, а именно: *компонентные*, *текстовые*, *бинарные (блочные)*.

Бинарным называется файл состоящий из последовательности нулей и единиц без определенной структуры.

Текстовый файл – файл содержащий набор символов таблицы *ASCII*. Редактирование этого файла возможно с помощью простого текстового редактора типа *NotePad*, встроенного в ОС *Windows*.

Компонентный файл – файл состоящий из однотипных ячеек. По своей структуре очень похож на одномерный массив. Формат записи данных является проприетарным[26]. Именно такие файлы нами будут рассмотрены далее. Основные отличия от массивов показаны в таблице.

Пример объявления файловых типов средствами языка *Pascal*:

```
type
  TTxt = text;
  TBinare = file;
```

[26] То есть порядок организации информационных блоков в файле скрыт от всеобщего обозрения.

```
TCompInt = file of integer;
TCompChar = file of char;
TCompBool = file of Boolean;
```

Здесь представлены следующие типы: ***TTxt*** – текстовый файл; ***TBinare*** – бинарный файл; ***TCompInt*** – файл целочисленных значений; ***TCompChar*** – файл символов; ***TCompBool*** – файл с компонентами логического типа.

Сравнение компонентных файлов и массивов

	Скорость обработки	Возможный размер	Энергонезависимость
Файлы	Медленно	Огромный	Есть
Массивы	Быстро	Малый	Отсутствует

Компонентный файл при объявлении выглядит следующим образом:

file of *тип компонент****;***

Тип компонент в файле может быть любым, кроме файлового.

Мы объявили только типы данных, теперь можно объявить и сами переменные. Например, следующим образом:

```
var
  F,G,S: TCompBool;
  A,B: TBinare;
  C: TCompChar;
  D,E : TCompBool;
  R: TCompInt;
  T,T1,T2 : TTxt;
  RRR: file of real;
```

Здесь объявлены *переменные-файлы*, причем, как видно, такая переменная как ***RRR***, является файлом вещественного типа, объявленным напрямую.

Для того чтобы начать использовать файл в программе нужно сначала его объявить, далее связать объявленную переменную с физической областью на диске (процедура ***assign***). После этого открываем его

для записи или чтения. Если файл вновь создаваемый, то следует использовать процедуру ***rewrite***. Формат у нее такой:

rewrite (*Файловая_переменная****);***

Если ранее в файле, на который ссылается файловая переменная, уже существовала какая-либо информация, то она автоматически стирается. Для работы с существующим файлом следует использовать процедуру ***reset***. При этом указатель автоматически сбрасывается на начало (*reset – сброс (англ.)*). Формат у этой процедуры таков:

reset (*Файловая_переменная****);***

После того как работа с файлом закончена, его следует закрыть при помощи процедуры ***close***. Это команда операционной системе на освобождение системных ресурсов, а также предотвращение ошибок некорректного чтения/записи в то время, пока работа с ним не ведется. Формат следующий:

close (*Файловая_переменная****);***

7.2 Компонентные файлы

Отвлекаясь от текстовых и бинарных файлов, займемся более подробно рассмотрением компонентных файлов.

Как отмечалось, особенностью компонентного файла является формальная схожесть с такой структурой как одномерный массив. Файл также как и массив имеет компоненты заданного типа, занимающие определенное место в памяти. Существенным отличием файла от массива является то, что под памятью следует понимать не оперативную память компьютера, а память статическую, расположенную на внешнем носителе. Объем информации на внешнем носителе существенно превосходит таковой в оперативной памяти, потому на размер файлов не накладывается

ограничение по длине, свойственное массивам (например, в *Turbo Pascal TP 7.0*, длина массива не должна превышать 64 КБ). Таким образом, для файла находится место там, где возникает задача обработки информации больших объемов, причем данные не теряются после завершения работы программы или компьютера. То есть, работая сегодня с информацией, мы можем спокойно сделать перерыв и вернуться к ее обработке через несколько дней. В настоящее время общее понятие о файлах у большинства людей имеющих опыт работы на компьютере не вызывает сложностей. Далее рассмотрим более подробно структуру файла, и порядок работы с ним.

Если в массиве за обращение к определенному компоненту отвечал индекс, то в компонентном файле принято говорить об ***указателе*** (или *файловом курсоре*), который показывает с начала какой позиции будет происходить чтение или запись в файл при ближайшем обращении к нему. Указатель можно представить в условно-графическом виде, как, например, на рис. 7.2, где он обозначен черной стрелочкой. Следует отметить, что счет позиции указателя начинается с нулевой. Это значит, что находясь в начале файла, как, например, в целочисленном ***F1***, указатель имеет позицию 0. В файле ***F2*** указатель находится на пятой позиции (5), в ***F3*** – в конце (9), в ***F4*** – на третьей позиции (3).

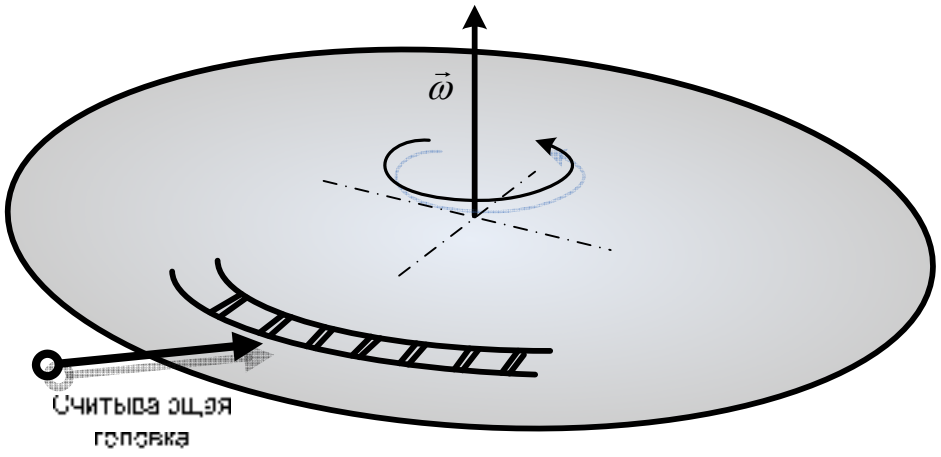


Рис. 7.1. Модель диска и файла на нем

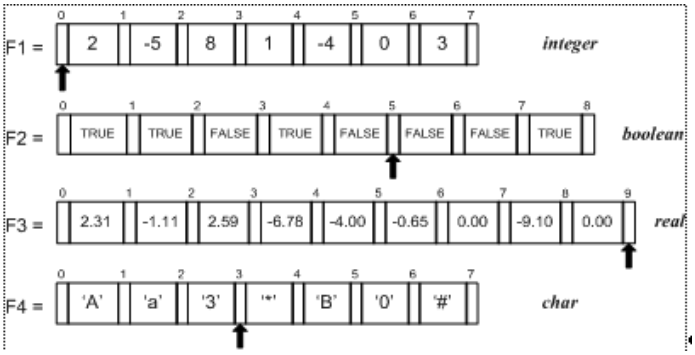


Рис. 7.2. Примеры различных файлов: F1 – целочисленный; F2 – логический; F3 – вещественный; F4 – символьный

Еще одним отличием в графическом представлении файлов являются двойные створки между элементами и на границах. Это обозначение принято для того, чтобы четко видеть, где в данный момент находится курсор. Чтение или запись в файловую ячейку происходит путем перемещение курсора вправо. Данное обозначение связано с исторической особенностью хранения файлов на диске.

Как известно, диск вращается в одну сторону, а считывающая головка при этом меняет свои координаты, приближаясь к центру, либо перемещаясь к периферии. Информация на диске записана последовательно концентрическими кольцами. То есть получается, что считывающий элемент находится на месте, в то время как файл проходит мимо него в одном направлении. С некоторой долей условности можно представить указатель в файле в виде считывающей головки диска. При чтении или записи некоторой информации происходит его перемещение в определенном направлении относительно головки. Это направление фиксировано и связано с конструктивной особенностью конкретной модели диска. Очень простая иллюстрация описанного процесса изображена на рис. 7.1.

Указатель можно переместить на произвольную файловую позицию. Делается это при помощи процедуры ***seek***. Формат ее записи таков:

seek(*Файловая_переменная****, номер позиции****);***

Здесь помимо, собственно, *файловой_переменной*, над которой производятся действия, указан еще и *номер позиции* на которую будет установлен указатель при вызове данной процедуры. Следует иметь в виду, что первая файловая позиция имеет нулевой номер. А последняя

файловая позиция имеет номер, равный количеству ячеек в файле, однако при установке указателя в конец файла мы ничего не сможем прочитать (поскольку файл закончился), но легко запишем туда нужную нам информацию.

Часто в задачах требуется узнать, на какой позиции в данный момент находится указатель. Для этого существует функция ***filePos***. Формат у нее следующий:

filePos(*Файловая_переменная****);***

Эта функция через свое имя возвращает значение позиции, на которой в данный момент находится указатель. Если требуется узнать длину файла, т.е. число компонент в нем, то следует применять функцию ***fileSize***. У нее такой формат:

fileSize(*Файловая_переменная****);***

Отметим, что как ***filePos***, так и ***fileSize***. возвращают целые значения типа ***longInt***. Т.е., если, например, требуется установить курсор в начало файла, то следует вызвать процедуру ***seek(F1,0)***, или ***reset(F1)***. Это проиллюстрировано на рис. 7.2. Среди других файлов, изображенных на рисунке можно провести серию вызовов процедур, которые приведут к распределению, которое там, собственно, имеется:

```
seek(F1,0);
seek(F2,5);
seek(F3,9);
seek(F4,3);
```

Или, что равнозначно, такую серию:

```
reset(F1);
seek(F2, fileSize(F2)-3);
seek(F3, fileSize(F3));
seek(F4, fileSize(F4)-4);
```

Видно, что для файлов ***F2*** и ***F4*** процедура установки получилась весьма своеобразной: от конца файла мы отняли три пункта в одном случае

и четыре в другом. Теперь можем вызвать, например, следующую последовательность операций:

```
a1:= filePos(F1);
a2:= filePos(F2);
a3:= filePos(F3);
a4:= filePos(F4);
```

в результате переменные ***a1***, ***a2***, ***a3***, ***a4*** получат соответственно, значения 0, 5, 9, 3.

Помимо установки указателя на произвольную позицию, существует еще и задача, собственно, чтения/записи компонент файла. Для этого применяют стандартные процедуры ***read*** и ***write***.

read(*Файловая_переменная****,*** *Читаемая_переменная_1****,*** *Читаемая_переменная_2****,*** *…****);***

write(*Файловая_переменная****,*** *Записываемая_переменная_1****,*** *Записываемая_переменная_2****,*** *…****);***

В первом случае мы имеем дело с чтением, а во втором с записью информации. Причем, компилятор самостоятельно определяет тот факт, что чтение или запись происходит именно в файл – по типу первой переменной в списке параметров (в скобках). Это означает, что если первая переменная среди параметров процедуры будет, например, числового или строкового типа, то они будут переданы на стандартное устройство ввода-вывода (на экран монитора). В этом, собственно и состоит их отличие от простых ***write*** и ***read***, рассматриваемых ранее. Нужно иметь в виду, что как чтение, так и запись происходят исходя из позиции указателя. После чтения или записи переменной автоматически происходит перемещение указателя на единицу вправо. Об этом всегда следует помнить!

Графически действия процедур ***write*** и ***read*** можно представить так, как это сделано на рис. 7.3. Здесь указатель изначально стоит на позиции 2,

а потому при чтении файла ***F1*** в переменную ***buf1*** происходит копирование ячейки справа от указателя (в ней находится число 8) в ячейку памяти, в которой хранится значение переменной ***buf1*** (теперь и в ***buf1*** тоже число 8). И после этого указатель перемещается на следующую позицию вправо. При дальнейшем вызове процедуры записи ***write(F1,buf2)*** происходит копирование содержимого ячейки памяти содержащей переменную ***buf2*** в ячейку файла под номером 3. Указатель передвигается на еще одну позицию вправо.

Теперь рассмотрим алгоритм клавиатурного ввода и вывода файла. Начнем с клавиатурного ввода. Для ввода файла можно пользоваться подобно массивам сведения о количестве компонент. Блок-схема типового алгоритма ввода представлена на рис. 7.4. Вариант а) использует число компонент ***N***, а далее идет последовательный ввод и соответствующая запись элемента. Здесь ввод аналогичен массиву. Однако, далеко не всегда известно точное число компонент наперед. Как правило, файлы содержат достаточно большой объем информации и потому следует прекращать их

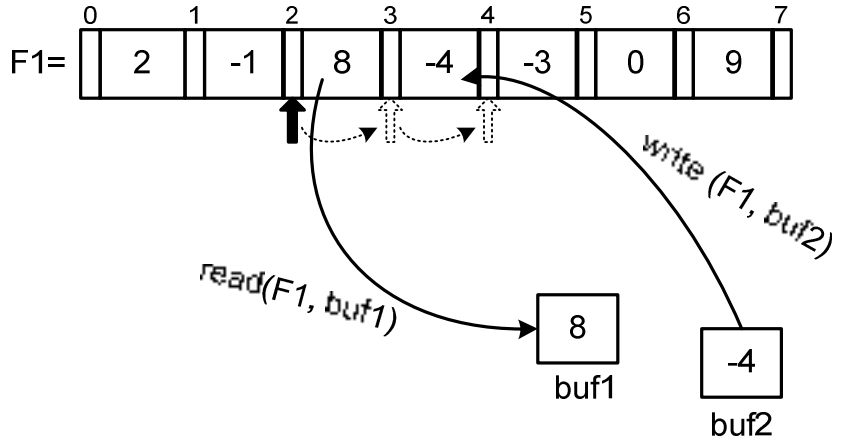


Рис. 7.3. Иллюстрация чтения компоненты файла F1
в переменную buf1 и записи значения переменной buf2 в файл

ввод не исходя из предполагаемого объема, а пользуясь некоторым *стоп-*

событием. Вообще, алгоритм изображенный на рис. 7.4, *а* является нерекомендуемым и по возможности его следует избегать.

В варианте рис. 7.4, *б* мы не знаем заранее, сколько всего будет введено компонент и потому можем ввести их произвольное количество. Точка остановки ввода (*стоп-событие*) будет при появлении признака конца файла. В данном случае в качестве такового выбрано число 999. Как только мы вводим число 999, выполняется условие выхода из цикла и запись компонент прекращается. Однако в данном случае мы не сможем вписать в файл, собственно, 999.

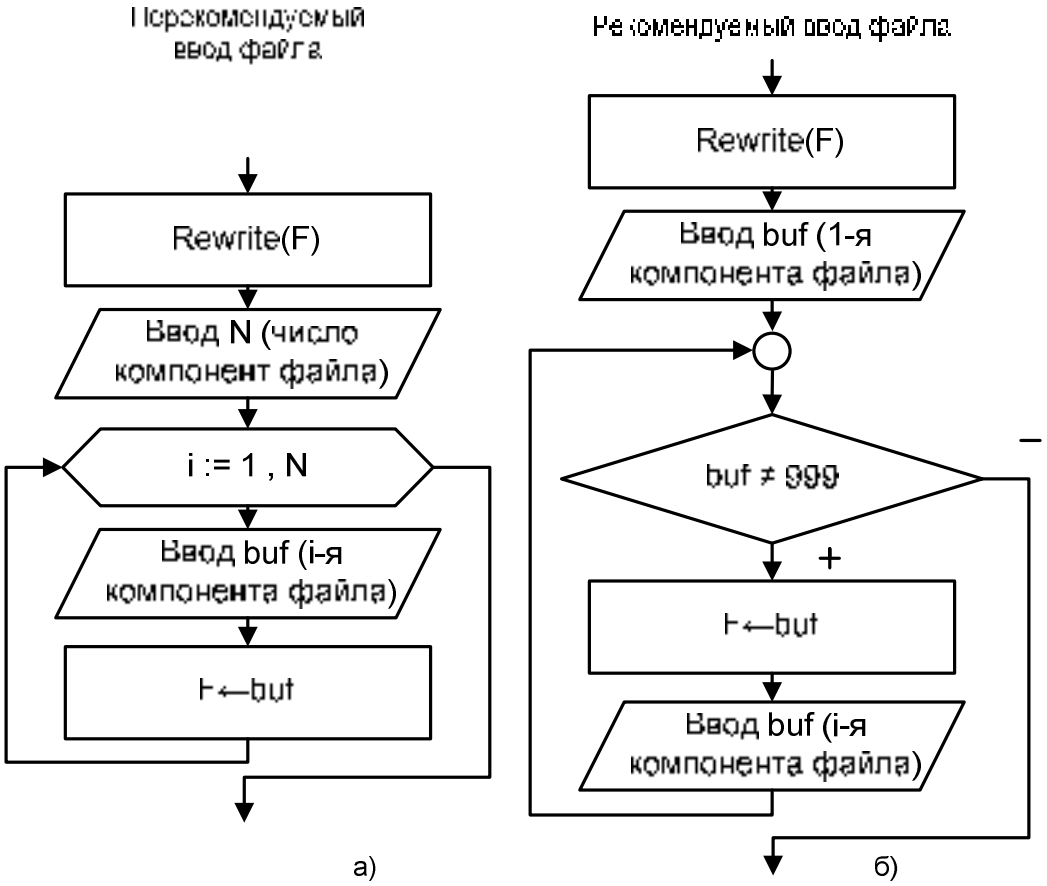


Рис. 7.4. Типовые алгоритмы ввода файлов:
а – по известному числу компонент (не рекомендуется к использованию); *б* – по признаку конца ввода

Далее рассмотрим последовательный вывод файла. Собственно, как и при вводе, можно предложить два вида вывода: по заранее известному числу компонент и по достижению конца файла. При выводе можно использовать тот факт, что количество компонент в файле известно, однако такой подход является нерекомендуемым. Рекомендуется производить вывод всех компонент подряд вплоть до достижения конца файла. При этом на каждой итерации производится проверка на достижение указателем конца файла. Признак конца можно задать при помощи функции ***EOF***. Она имеет формат:

EOF(*Файловая переменная****);***

Данная функция возвращает логическое значение (***TRUE*** или ***FALSE***), в зависимости от того, достигнут конец файла или нет. Название *EOF* является акронимом от английского *End Of File*, что по-русски означает *конец файла*. Соответствующие блок-схемы ввода представлены на рис. 7.5.

Поскольку мы не пользуемся процедурой установки курсора на произвольную позицию ***seek***, а только лишь последовательно записываем в файл компоненты, то такой метод работы с файлами называется *последовательным доступом*. Если же используется процедура ***seek***, то доступ к файлу принято называть *произвольным*.

Очень часто при работе с файлами возникает ситуация, когда требуется произвести его усечение. Усечение файла, т.е. отбрасывание компонент следующих за текущей позицией применяется для того, чтобы не хранить заведомо ненужные данные. Для усечения файла с текущей позиции применяется процедура ***truncate***, она имеет формат:

truncate(*Файловая_переменная****);***

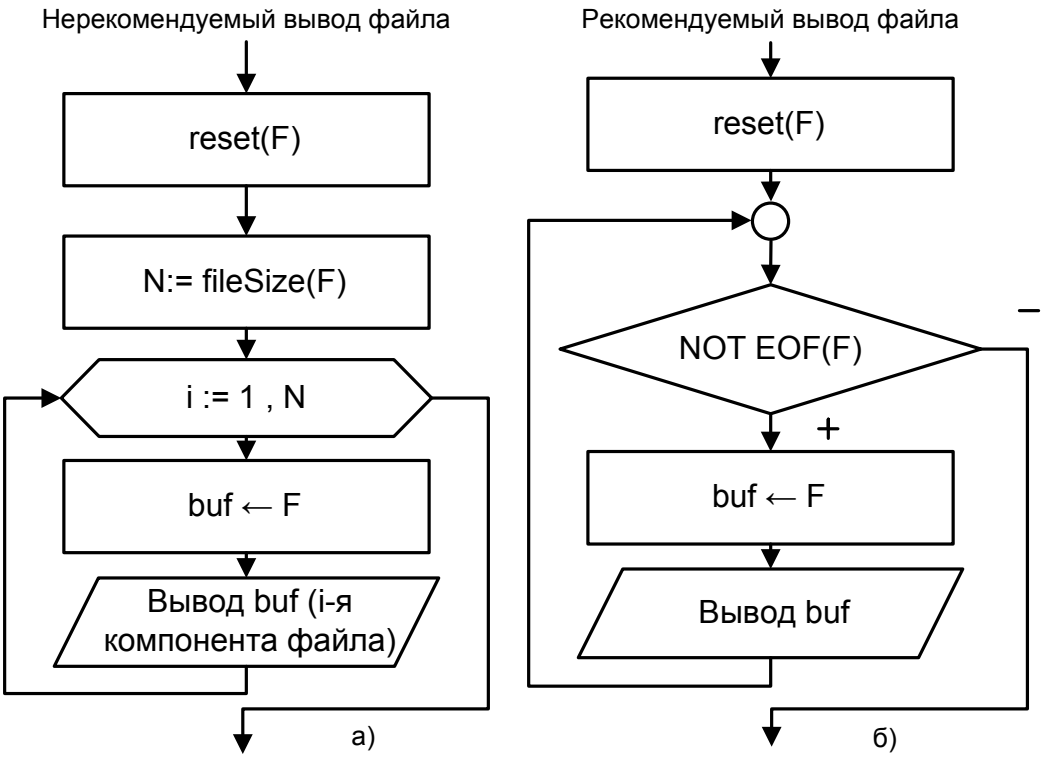


Рис. 7.5. Вывод файла:
а – по известному числу компонент (не рекомендуется к использованию);
б – по признаку конца файла

7.3 Файлы последовательного доступа

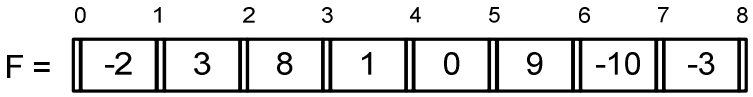
Если обработка элементов файла ведется от начала и до конца последовательно, без использования процедуры принудительной перестановки курсора, то такая обработка называется последовательным доступом. Если обработке подвергаются все без исключения элементы, то блок-схема будет такой, как на рис. 7.6, *а*. Если на элемент накладываются некоторые условия, то в тело цикла добавляется дополнительная развилка (рис. 7.6, *б*). Среди ранее рассмотренных алгоритмов последовательным доступом стоит считать вывод файла.

Решим несколько типовых задач на последовательный доступ.

ПРИМЕР

Найти сумму всех элементов файла. Составим тестовый пример:

вход:



выход:

S = (–2)+3+8+1+0+9+(–10)+(–3) = 6.

Решение задачи целиком показано на рис. 7.7.

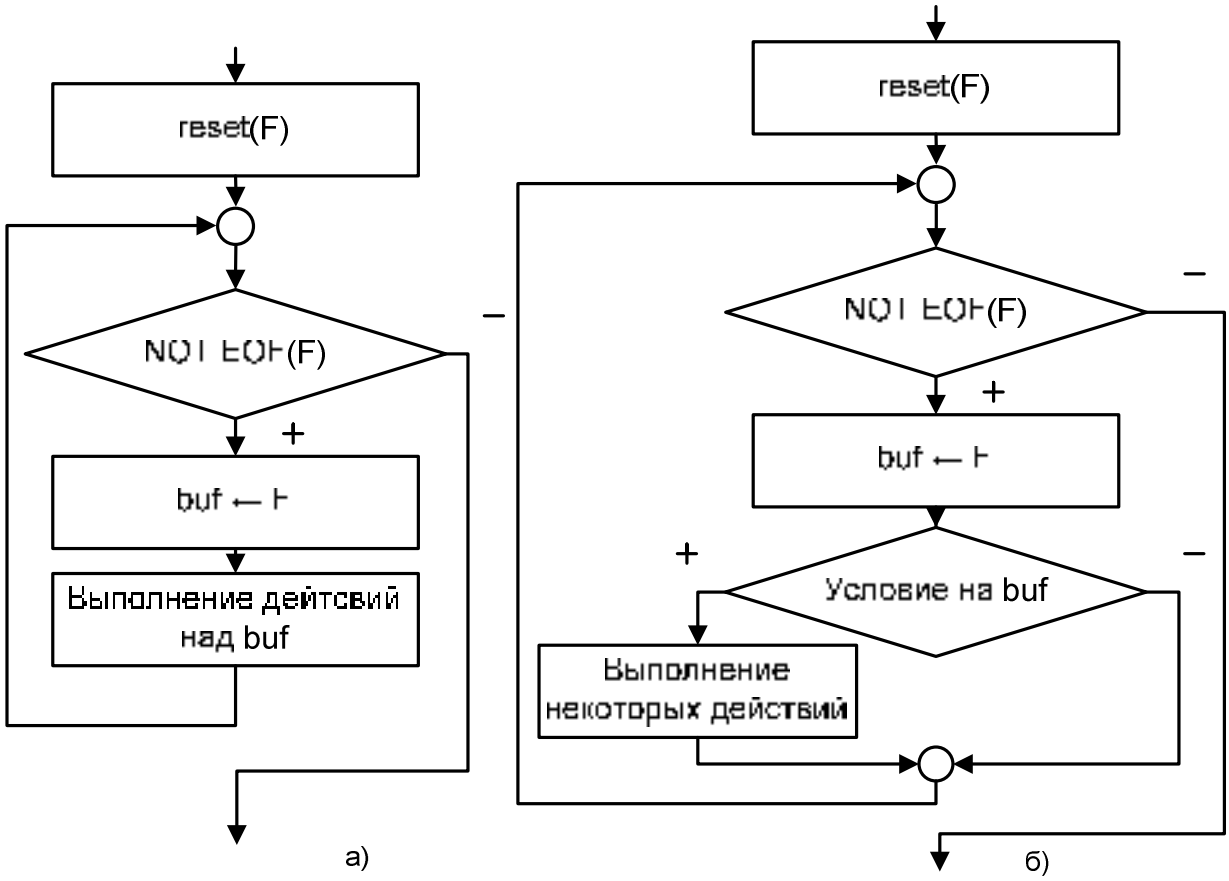


Рис. 7.6. Последовательная безусловная обработка файла (*а*) и последовательная условная обработка файла (*б*)

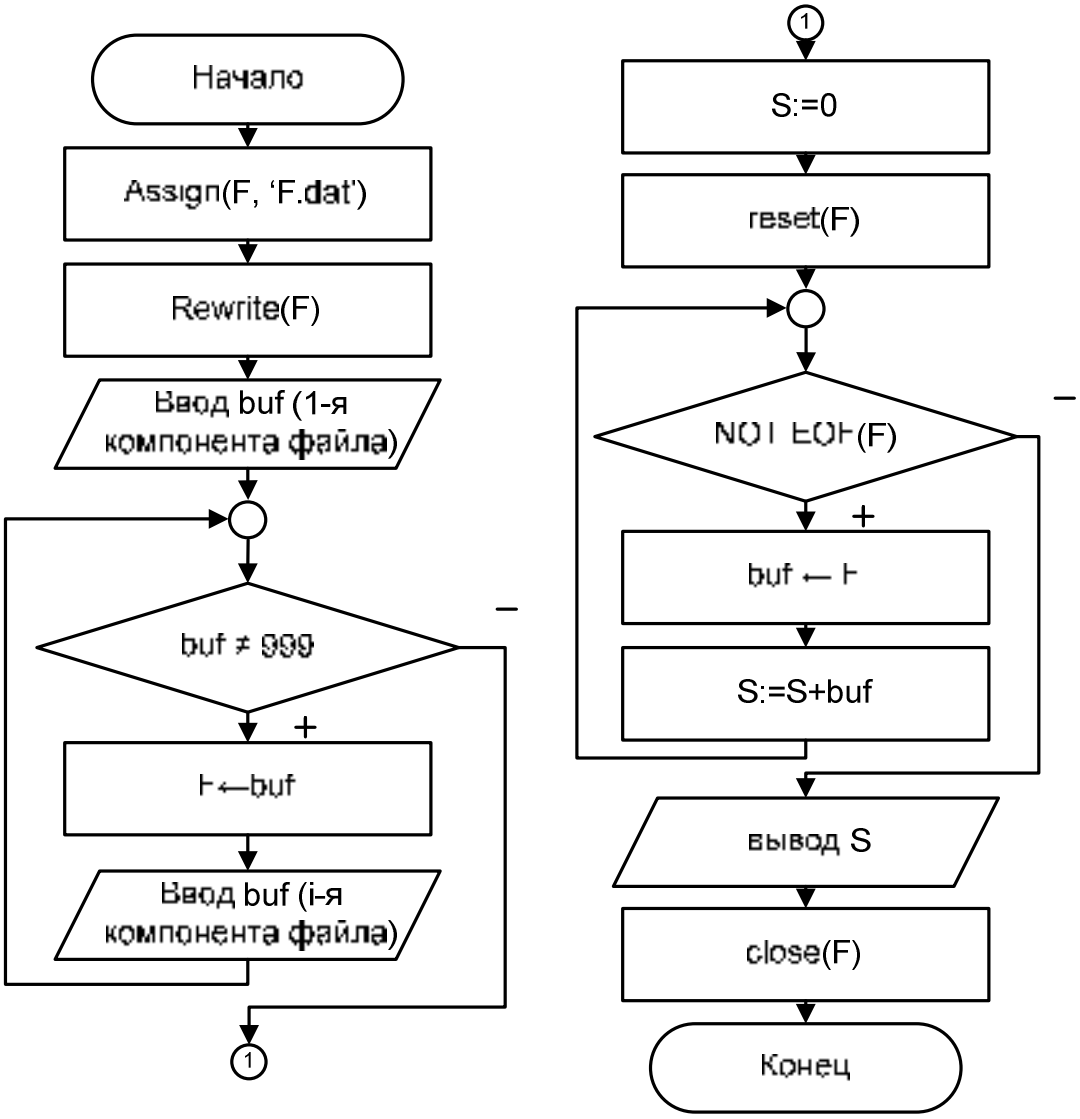


Рис. 7.7 Поиск суммы элементов файла

Листинг программы будет таков:

```
program Files_0;
var
     F: file of integer;
     S,buf:integer;
begin
  Assign(F,'F.dat');
  rewrite(F);
  writeLn('введите первую компоненту файла');
  readLn(buf);
  while buf<>999 do
    begin
      write(F,buf);
      writeLn('введите следующую компоненту:');
      readLn(buf);
    end;
  S:=0;
  reset(F);
```

```
  while not EOF(F) do
    begin
      read(F,buf);
      S:=S+buf;
    end;
  writeLn('найденная сумма равна:',S);
  close(F);
end.
```

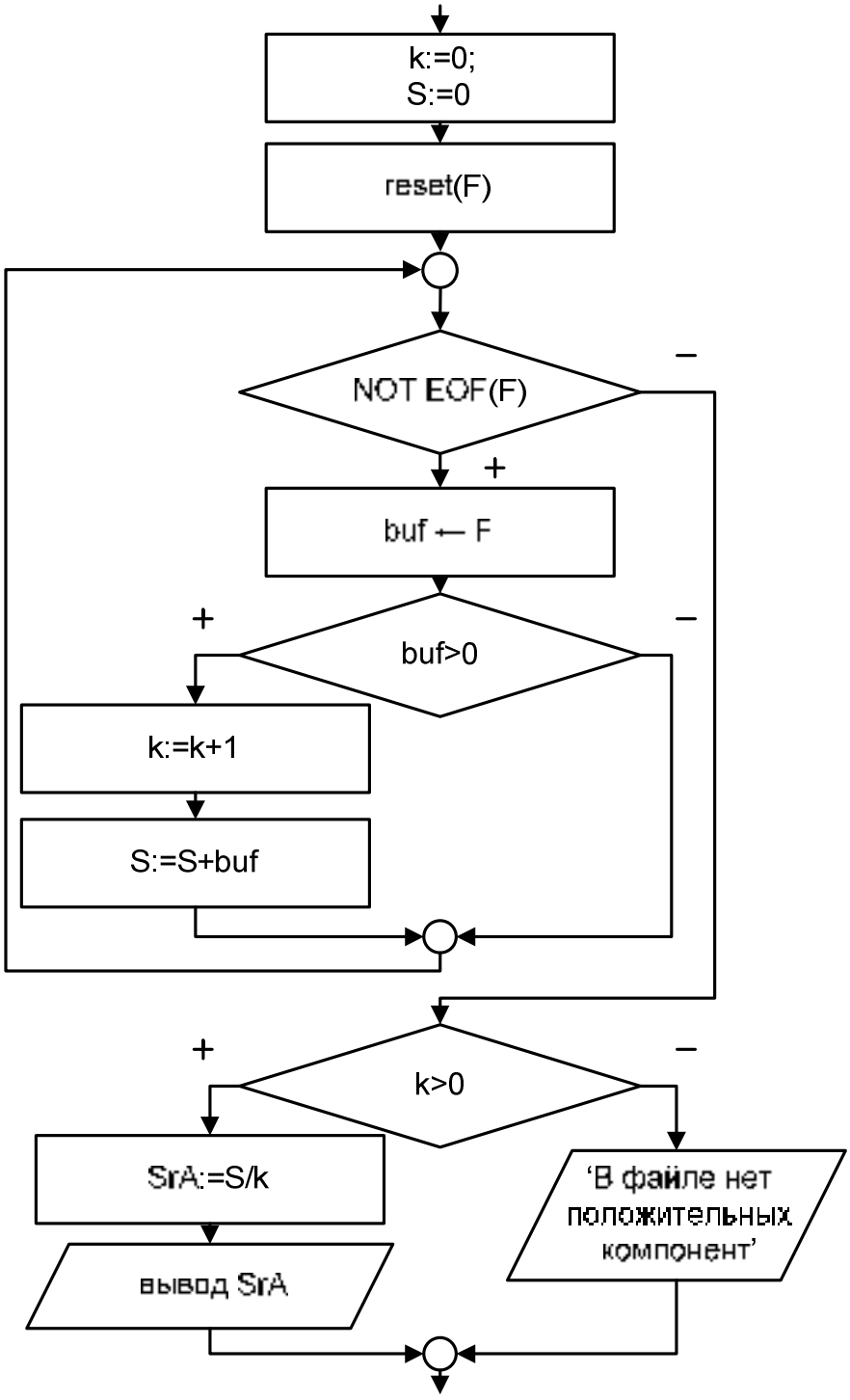


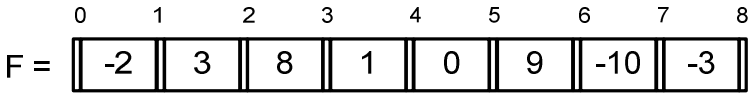
Рис. 7.8 Среднее арифметическое положительных компонент

Следующий пример – на последовательную обработку с анализом элементов.

ПРИМЕР

Найти среднее арифметическое положительных компонент файла.

вход:



выход: SrA = (3+8+1+9)/4= 21/4 = 5,25.

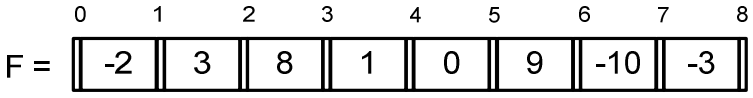
Приведем здесь блок-схему только основного алгоритма без ввода файла, который был расписан ранее. Файл читается от начала до конца и анализируются прочитанные компоненты. Если компоненты оказываются большими нуля, то они добавляются в сумму ***S***, при этом увеличивается счетчик ***k***. Среднее арифметическое ***SrA*** существует лишь тогда, когда в файле есть положительные элементы. Блок-схема описанного процесса показана на рис. 7.8.

Далее еще один пример на последовательную обработку.

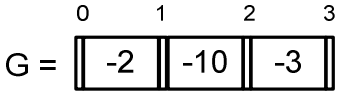
ПРИМЕР

Переписать отрицательные компоненты файла F в файл G.

вход:



выход:



Сначала вручную вводится файл ***F***, далее файл проходится в прямом направлении и встретившиеся отрицательные компоненты копируются последовательно в файл ***G***. Здесь следует обратить внимание на процедуру связки файла ***G***: ***Assign(G, ‘G.dat’)***, и не написать случайно в качестве параметра отвечающего за имя физического файла файл ***F***, т.е. ***‘F.dat’***, а то

получится, что переменные ***F*** и ***G***, будут ссылаться на оду и ту же область диска и, соответственно, никакого копирования не произойдет. Эта задача решена полностью (рис. 7.9).

Такова реализация решения средствами *Pascal*:

```
program FtoG;
var  F,G: file of integer;
     buf:integer;
begin
  Assign(F,'F.dat');
  rewrite(F);
  writeLn('введите первую компоненту файла');
  readLn(buf);
  while buf<>999 do
    begin
      write(F,buf);
      writeLn('введите следующую компоненту:');
      readLn(buf);
    end;
  reset(F);
  Assign(G,'G.dat');
  rewrite(G);
  while not EOF(F) do
    begin
      read(F,buf);
      if buf<0 then
write(G,buf);
    end;
  writeLn('полученный файл G:');
  reset(G);
  while not EOF(G) do
    begin
      read(G,buf);
      write(buf:5);
    end;
  close(F);
close(G);
end.
```

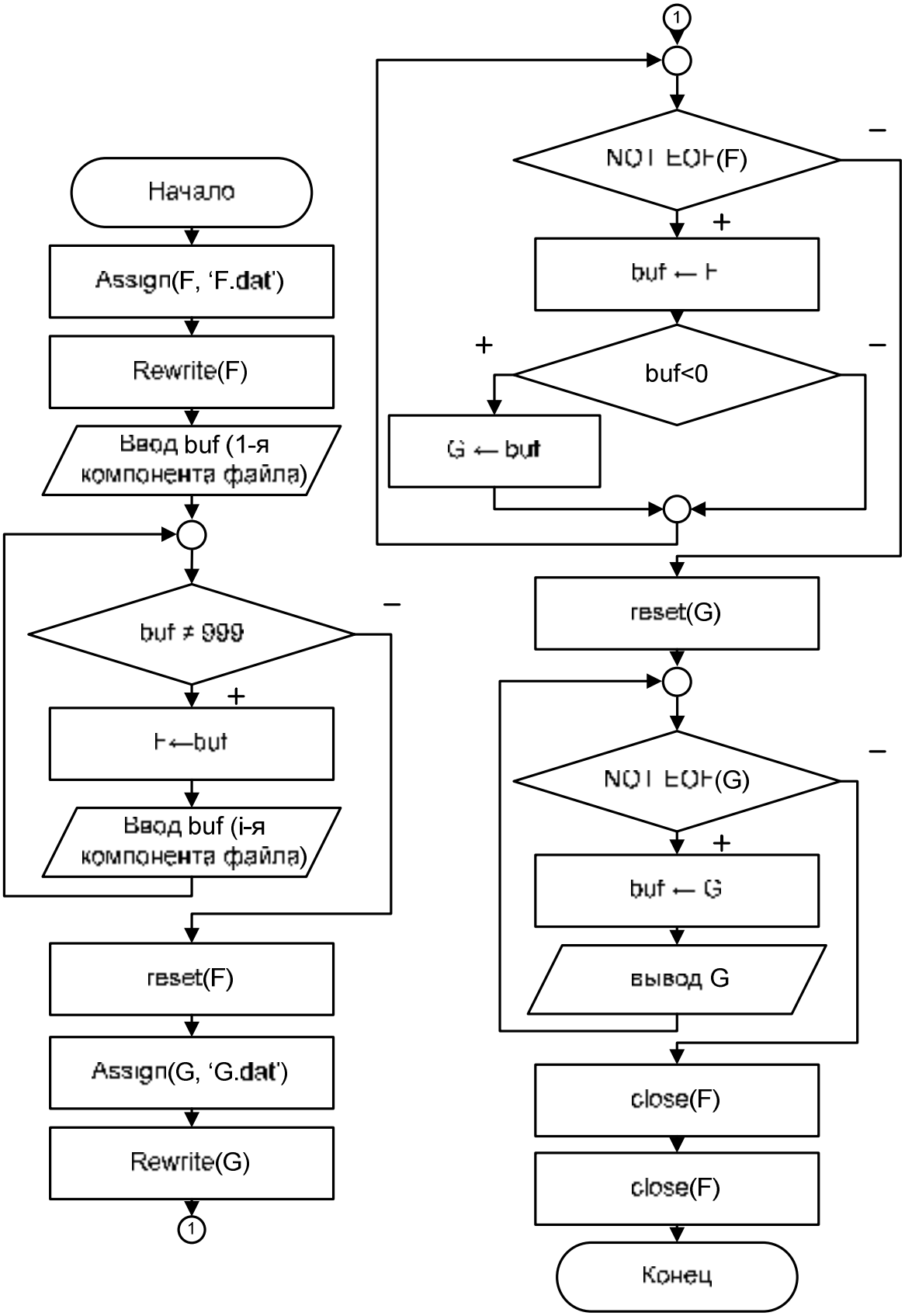


Рис. 7.9. Переписывание отрицательных компонент одного файла в другой файл

7.4 Файлы произвольного доступа

Рассмотрим несколько примеров алгоритмов работы с файлами произвольного доступа. Под произвольным доступом понимается работа с файлом с возможностью произвольного перемещения указателя. Как правило, произвольный доступ обеспечивается процедурой ***seek***.

Среди основных алгоритмов обработки файлов стоит особо отметить алгоритм чтения компонент и записи на это место нового значения. Поскольку указатель перемещается вправо как при чтении, так и при записи, то сразу после записи, его следует вернуть на одну позицию назад. Приведем пример последовательного чтения компонент файла с последующей записью новых на то же место. Для этого обычно применяют алгоритм, подобный изображенному на рис. 7.10.

Далее рассмотрим задачу, использующую среди прочего описанный алгоритм.

ПРИМЕР

Ввести файл действительных чисел. Найти в нем среднее арифметическое каждой второй компоненты. Далее все компоненты, чей модуль меньше модуля найденного среднего арифметического, удвоить. Файл до и после преобразования вывести на экран.

Начинаем, как обычно, с тестового примера:

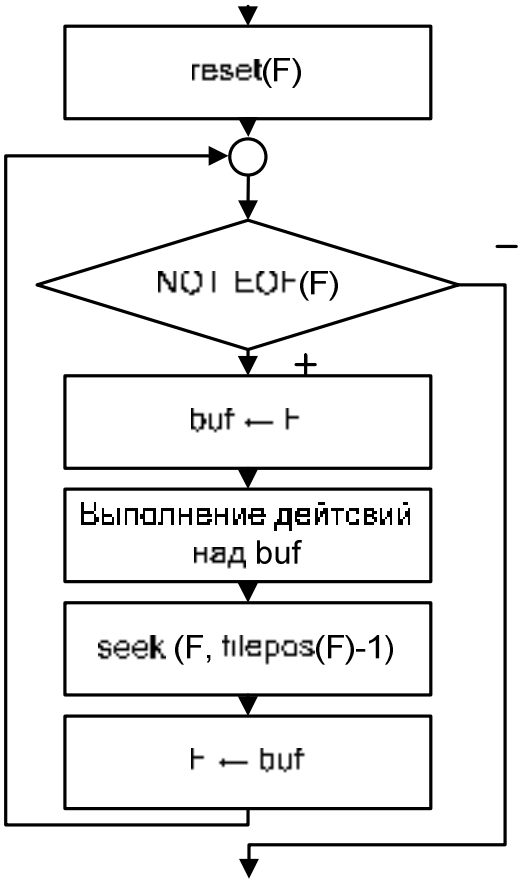
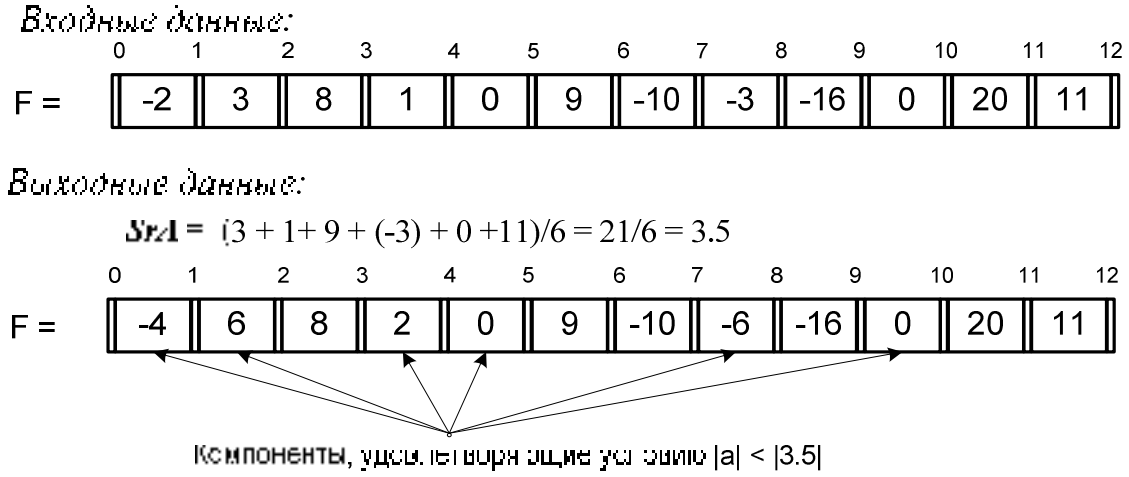


Рисунок 7.10. Формирование новых компонент на основе прочитанных



Следующий этап – это составление блок-схемы алгоритма. Можно сначала записать общий алгоритм, а далее каждый шаг детализировать. Общая последовательность действий решения представлена на рис. 7.11, *а*. Здесь шаги 1-2, 2-3 и 5-6 уже были описаны ранее (см. рис. 7.4, рис. 7.5). Потому есть смысл детализировать шаги 3-4 и 4-5. Их детализация изображена на рис. 7.11, *б* и рис. 7.12. Следует отметить, что доступ к каждой второй компоненте файла осуществляется прямым обращением. Из детализации шага 3-4 видно, что есть возможность «проскочить» конец файла и поставить указатель на несуществующую позицию. Однако это не вызовет ошибки, поскольку такие действия возможны. Даже можно записать значение в компоненту далеко за границей файла. В этом случае компоненты между последней компонентой и вновь введенной будут не определены (скорее всего, они будут заняты нулями, однако не факт, так как это зависит от версии компилятора *Pascal*). Самое главное не пытаться читать с позиции указателя выходящей за границы файла, поскольку это приведет к ошибке.

По составленным блок-схемам можно написать программу на языке *Pasccal*:

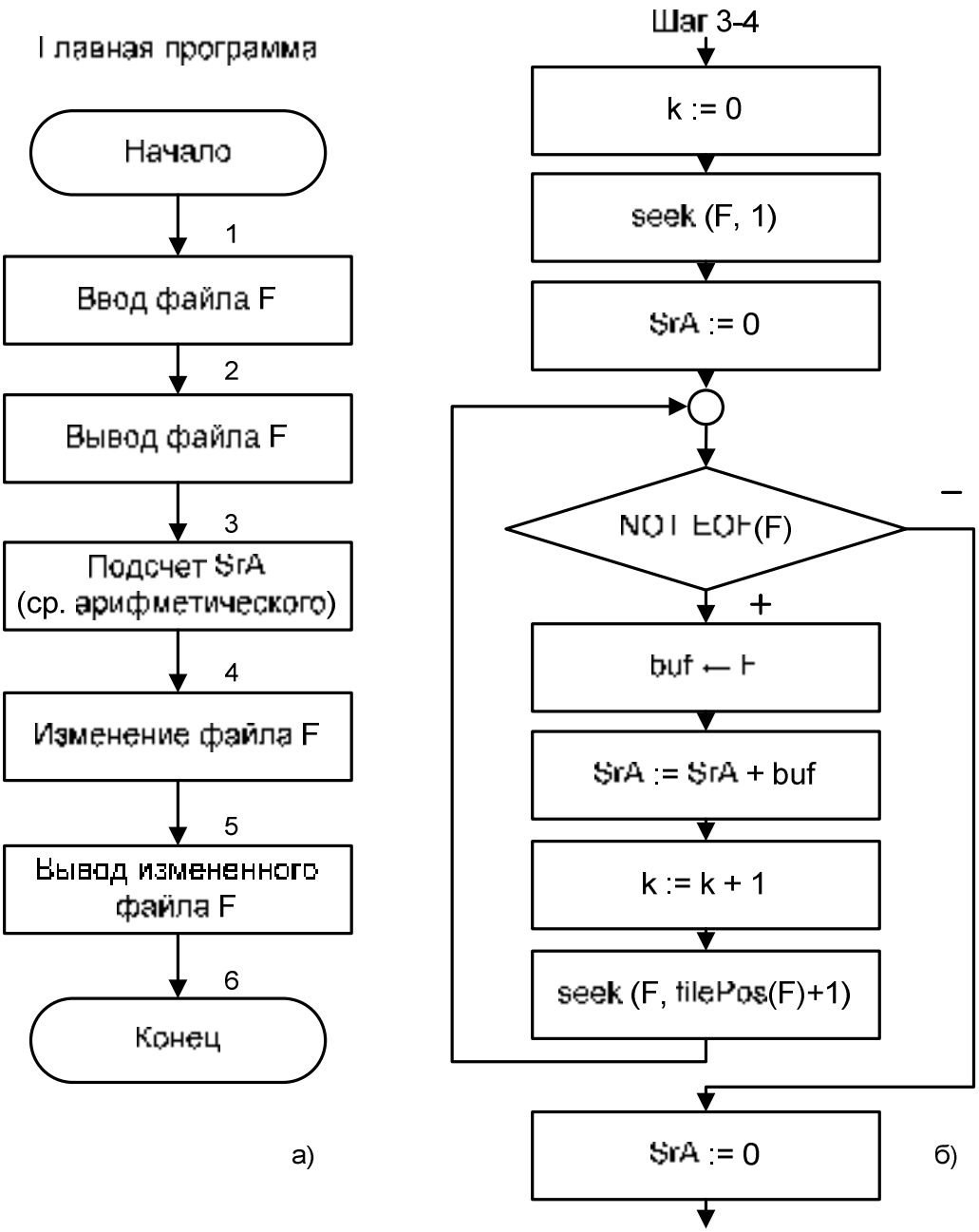


Рис. 7.11. Главная программа (*а*) и поиск среднего арифметического каждой второй компоненты – б)

```
program Files_1;
type Tfl = file of real;
var F : Tfl;
    buf, SrA : real;
    k : byte;
begin
  {начало ввода файла (шаг 1-2)}
  assign(F,'GoodFile1.dt');
  rewrite(F); k:=0;
```

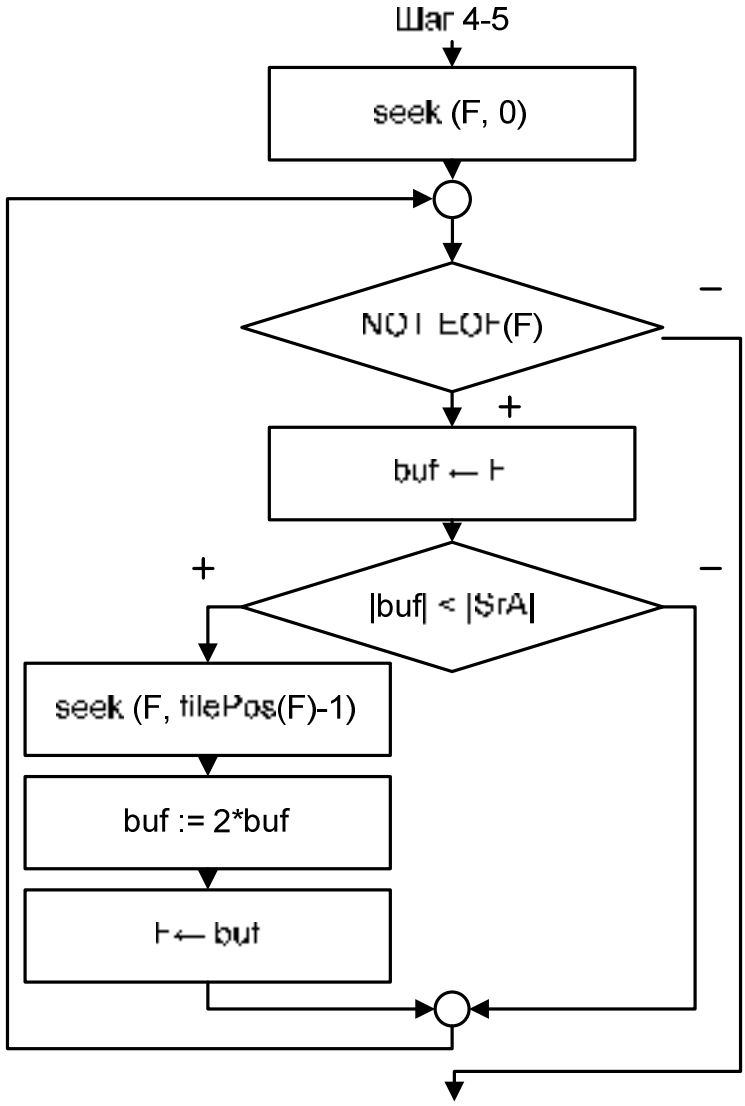


Рис. 7.12. Удвоение компонент, удовлетворяющих условию

```
writeLn('999 – окончание ввода');
  writeLn('Вводим файл F:');
  write('1-я комп. файла = ');
  readLn(buf);
  while buf <> 999 do
    begin
      write(F, buf);
      k:=k+1;
      write(k,'-я комп. файла = ');
      readLn(buf);
    end;
  {Вывод файла (шаг 2-3)}
  reset(F);
```

```
  writeLn('Исходный файл F:');
  while not EOF(F) do
    begin
      read(F, buf);
      write(buf:7:2);
    end;
  {подсчет среднего арифметического (шаг 3-4)}
  k:=0;
  seek(F,1);
  SrA:=0;
  while not EOF(F) do
    begin
      read(F,buf);
      SrA:=SrA+buf;
      k:=k+1;
      seek(F,filePos(F)+1);
    end;
  SrA := SrA / k;
  {Замена компонент, чей модуль меньше среднего}
  {арифметического (шаг 4-5)}
  seek(F,0);
  while not EOF(F) do
    begin
      read(F,buf);
      if abs(buf)<abs(SrA) then
        begin
          seek(F,filePos(F)-1);
          buf:=2*buf;
          write(F,buf);
        end;
    end;
  {Вывод измененного файла (шаг 5-6)}
  seek(F,0);
  writeLn('Измененный файл F:');
  while not EOF(F) do
    begin
      read(F, buf);
      write(buf:7:2);
    end;
  close(F);
end.
```

Поскольку файлы своей структурой напоминают одномерные массивы, то и задачи при их обработке возникают схожие. При решении этих задач практически всегда идейно используются одинаковые алгоритмы как для файлов, так и для массивов. Несмотря на схожесть,

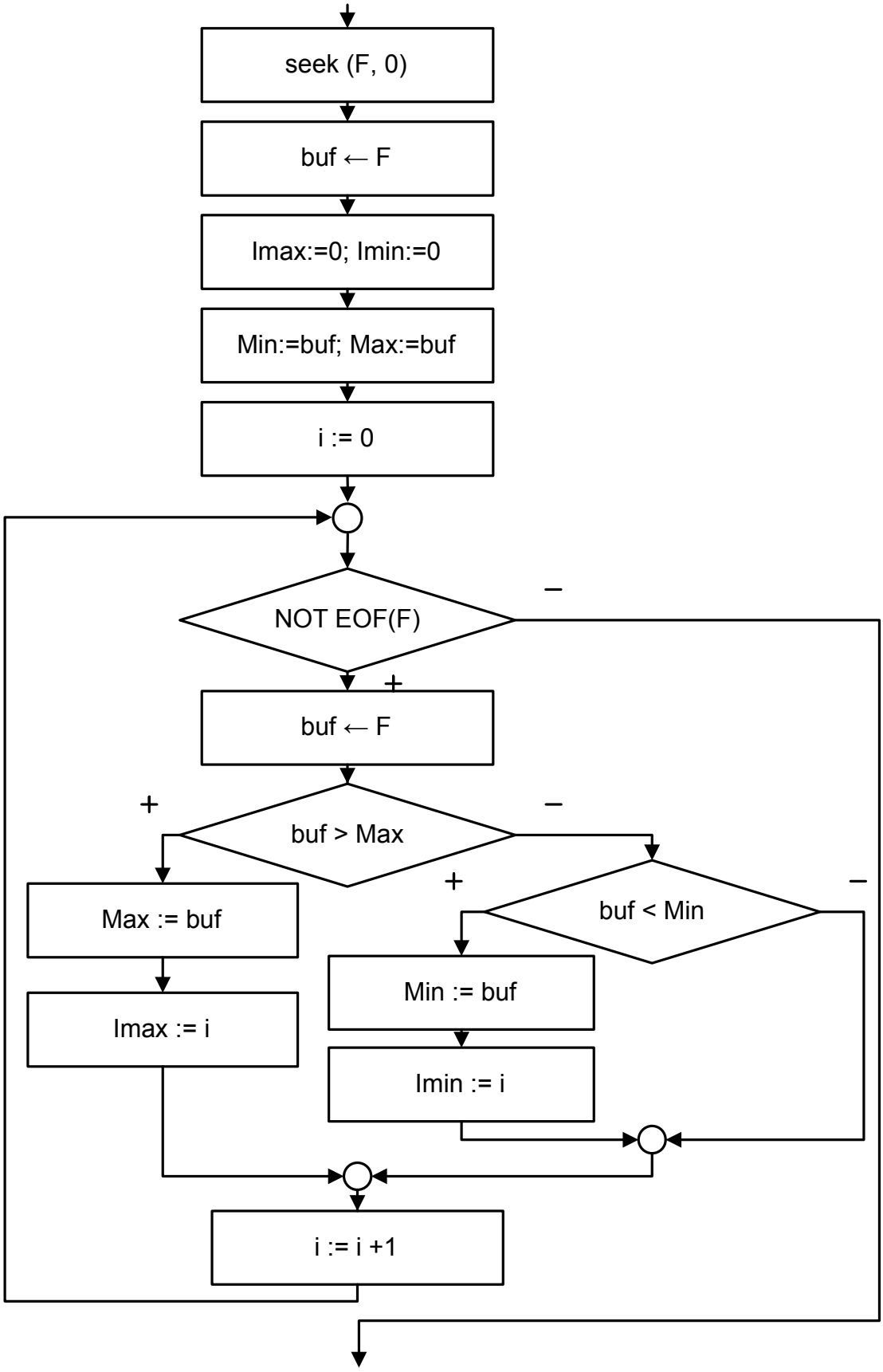


Рис. 7.13. Поиск экстремальных компонент в файле и их позиций

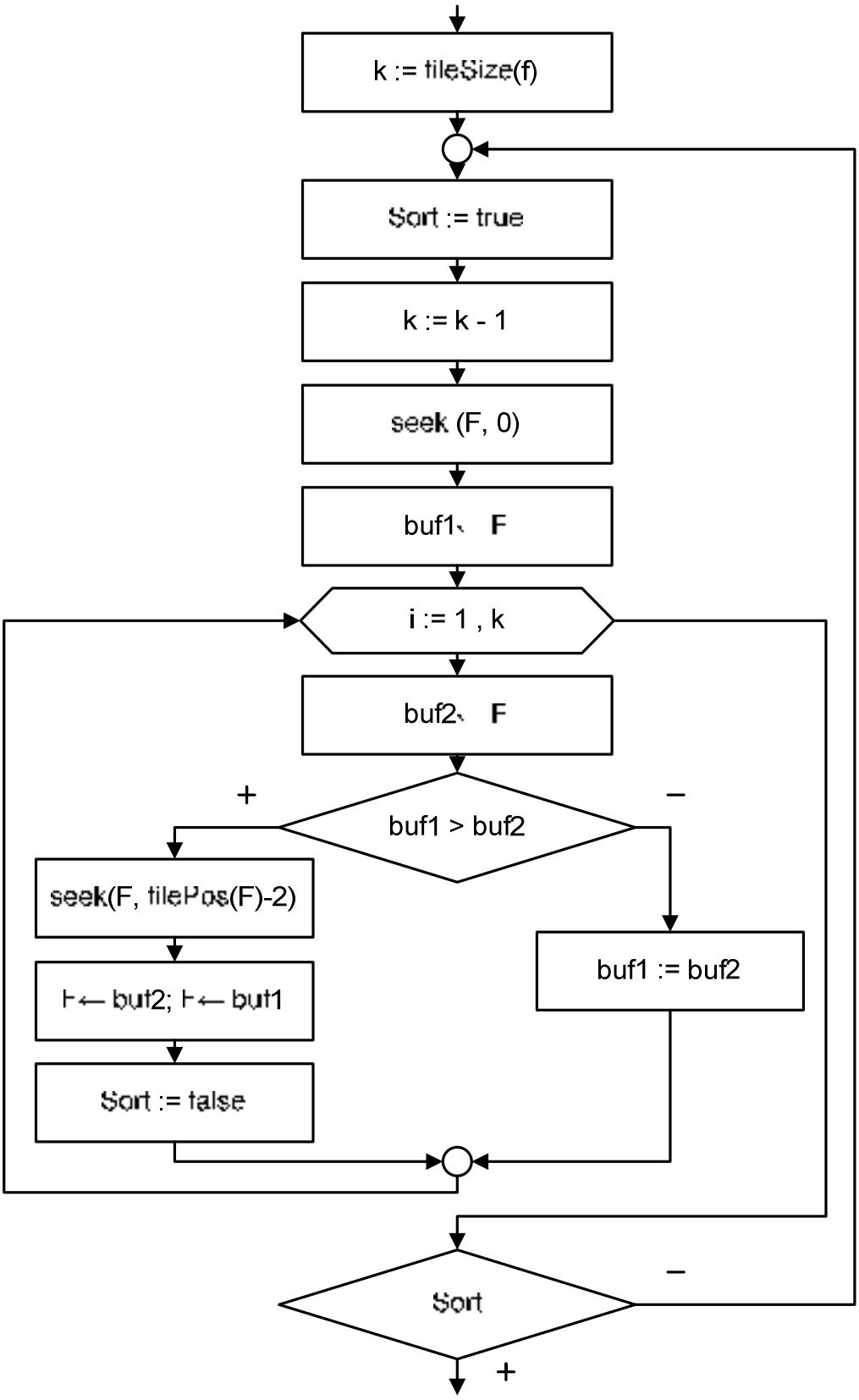


Рис. 7.14. Сортировка файла по возрастанию методом «пузырька»

все-таки имеются различия в реализации. Самое главное отличие, как отмечалось ранее, в том, что для доступа к компонентам файла используется указатель, в отличие от индекса в массиве.

Задачи могут быть на сортировку, циклический сдвиг, различные перестановки, поиск определенных элементов и т.д. Далее приведем без пояснений возможные алгоритмы поиска максимальной и минимальной компоненты в файле и их позиций (рис. 7.13), а также алгоритм сортировки элементов по возрастанию (рис. 7.14).

7.5. Файлы и подпрограммы

Использование файлов в качестве формальных параметров подпрограмм допускается только как параметров-переменных (с префиксом ***var***), т. е. передача файла-параметра происходит по ссылке с правом изменения. Передача по значению не возможна ввиду возможной относительной неограниченности размера файла. Стоит напомнить, что при передаче параметра по значению происходит создание копии объекта в оперативной памяти компьютера, из-за чего возможна ситуация элементарной нехватки памяти.

Поскольку файловый тип – сложный, то, подобно массивам, он должен быть изначально описан в разделе типов. Рассмотрим работу с файлами и подпрограммами на примере решения следующей задачи.

ПРИМЕР

Создать файл F. Читая файл с конца, переписать компоненты с четных позиций в файл G, а с нечетных – в файл H.

Начнем с тестового примера:

Входные данные:

F =

0	1	2	3	4	5	6	7	8	9	10
1	3	0	-5	2	9	-8	1	31	0	

Выходные данные:

H =

0	1	2	3	4	5
0	1	9	-5	3	

G =

0	1	2	3	4	5
31	-8	2	0	1	

Задачу решим, используя процедуры и функции. Блок-схема алгоритма основной программы представлена на рис. 7.15.

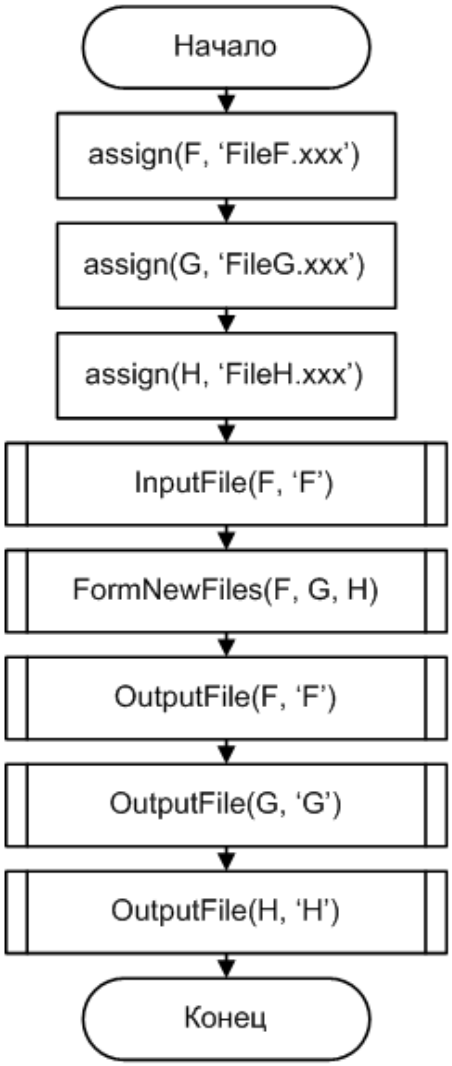


Рисунок 7.15 Основная программа
к задаче на чтение файла в обратном порядке

Теперь опишем все процедуры, входящие в данный алгоритм. Это, прежде всего, ввод и вывод (***InputFile*** и ***OutputFile***) – рис. 7.16, а также процедура ***FormNewFile*** (рис. 7.17). В процедуре ***FormNewFile*** следует обратить внимание на то, что проход происходит в обратном направлении. Факт достижения начала файла проверяется постусловием стандартной функцией ***filePos***. Цикл прекращается при равенстве нулю текущей позиции файла.

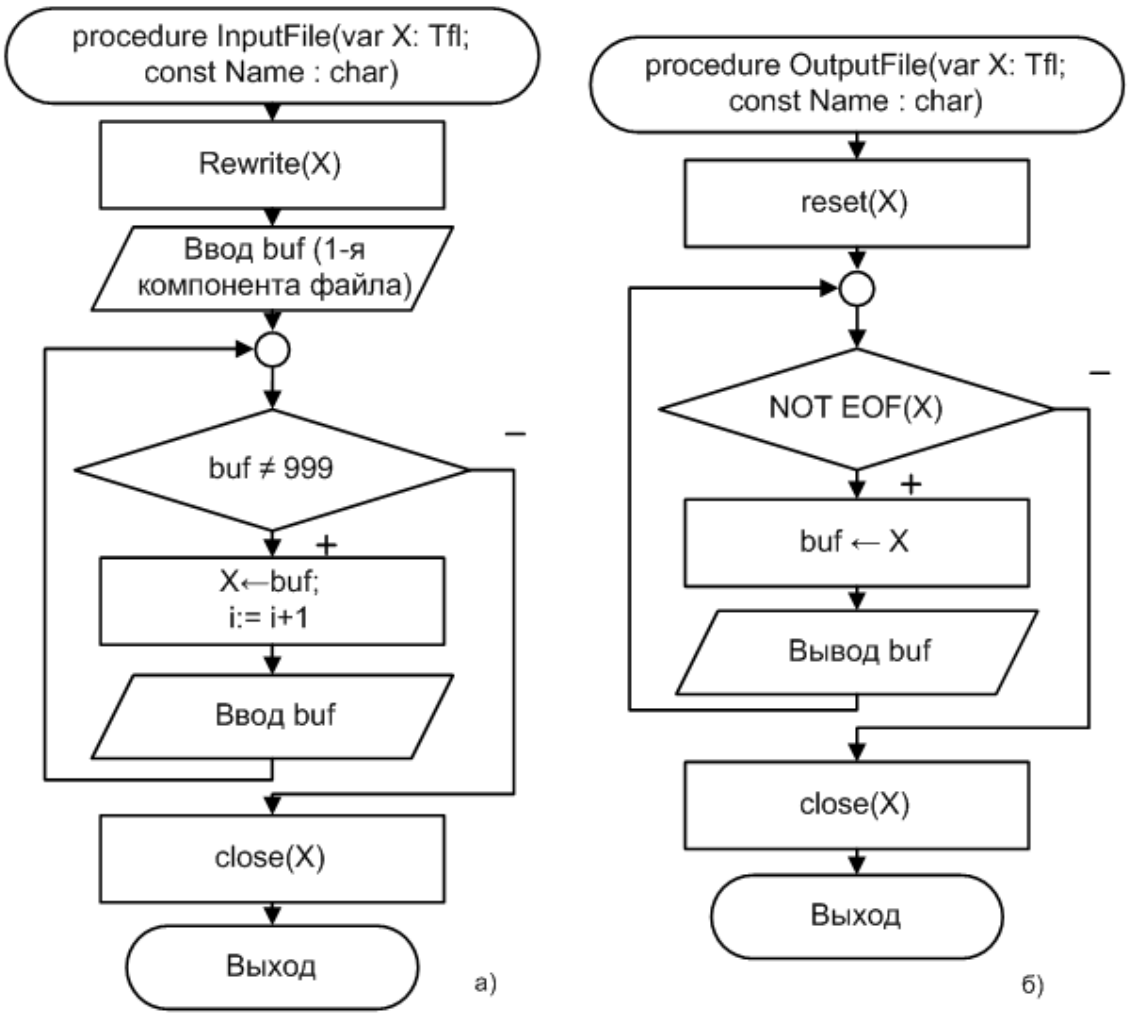


Рис. 7.16. Процедуры ввода и вывода файла

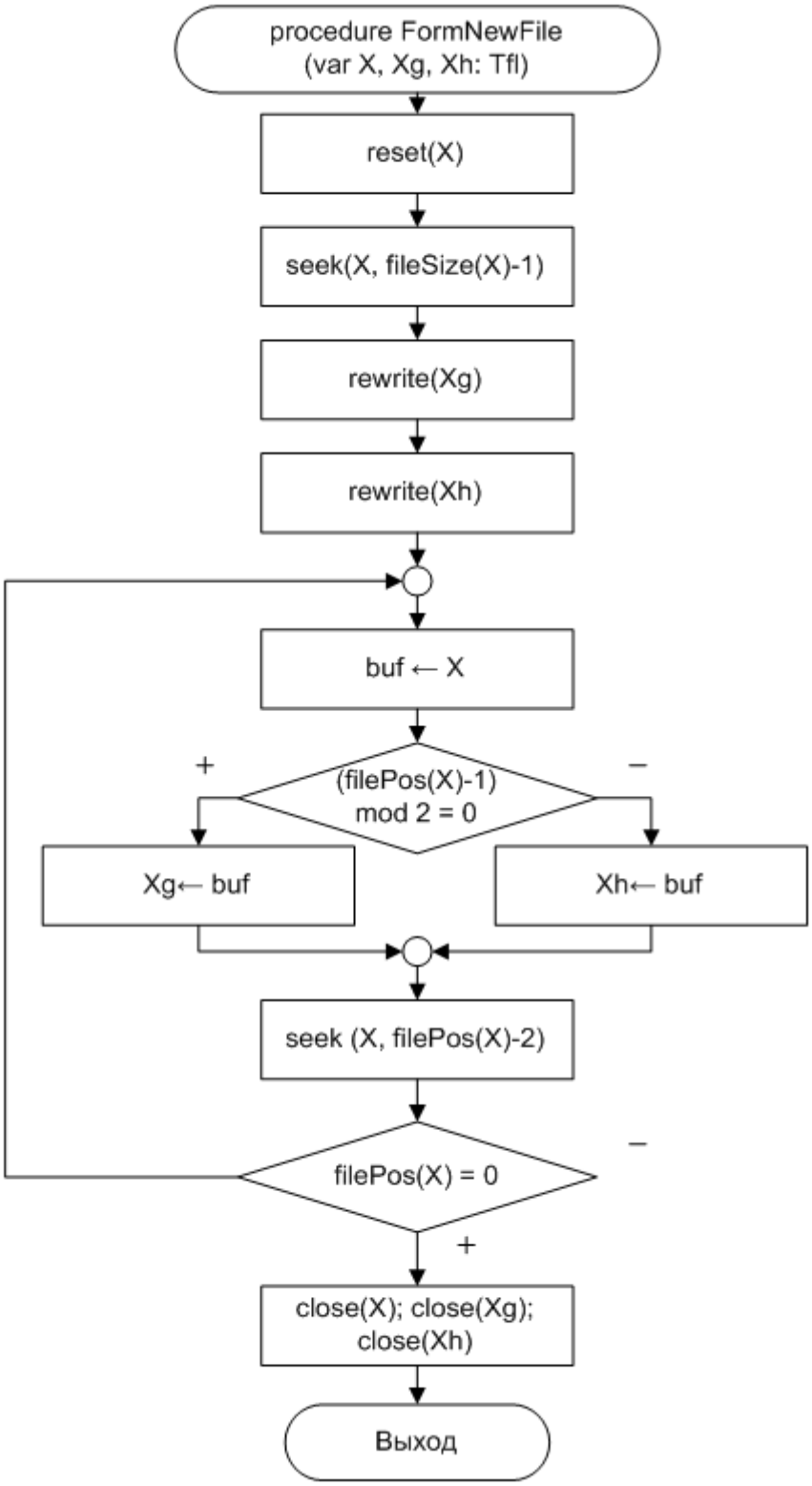


Рис. 7.17. Процедура чтения файла в обратном порядке
и записи результатов в пару новых файлов

Имея блок-схему, достаточно просто составить программу.

```
program FilePodprogr;
type Tfl = file of real;
var F,G,H: Tfl;
{процедура ввода файла}
procedure InputFile(var X: Tfl; const Name: char);
var i: byte; buf: real;
begin
  rewrite(X);
  writeLn('Вводим файл ',Name,' 999- окончание ввода');
  write('первая компонента:');
  readLn(buf); i:=1;
  while buf<>999 do
    begin
      write(X,buf); inc(i);
      write(i,'-я комп. файла = ');
      readLn(buf);
    end;
  close(X);
end;
{процедура вывода файла}
procedure OutputFile(var X:Tfl; const Name:char);
var buf: real;
begin
  reset(X);
  writeLn;
  writeLn('Файл', Name, ':');
  while not EOF(X) do
    begin
      read(X, buf);
      write(buf:7:2);
    end;
  close(X);
end;
{процедура формирования новых файлов}
procedure FormNewFile(var X,Xg,Xh:Tfl);
var buf: real;
begin
  reset(X);
  seek(X, FileSize(X)-1);
  rewrite(Xg);
  rewrite(Xh);
  repeat
    Read(X, buf);
    if (filePos(X)-1) mod 2=0 then
      write(Xg,buf)
    else
      write(Xh,buf);
    seek(X, filePos(X)-2);
```

```
  until filePos(X)=0;
end;
{главная программа}
begin
  assign(F,'FileF.xxx');
  assign(G,'FileG.xxx');
  assign(H,'FileH.xxx');
  InputFile(F,'F');
  FormNewFile(F,G,H);
  OutputFile(F,'F');
  OutputFile(G,'G');
  OutputFile(H,'H');
end.
```

7.6. Компонентные файлы и массивы

Использование файлов является универсальным инструментом для хранения в энергонезависимой памяти информации любого вида. Как наиболее простой и в тоже время наглядный пример этой информации можно рассмотреть массивы. Важно придумать правило, по которому будет вестись запись компонент в файл и, соответственно, правило извлечения данных в таком порядке, чтобы на выходе получалась структура идентичная структуре на входе.

Для начала возьмем одномерный массив. Вспомним, что компонентный файл является весьма схожим с одномерным массивом практически по всем параметрам, потому самый простой вариант действий в данном случае – ничего особо не менять. Просто следует поставить в соответствие компонентам файла элементы массива. Нужно указать в файле начальную позицию с которой будет вестись чтение/запись массива и число элементов в массиве. Фрагмент программы полностью копирующий файл ***F*** в одномерный массив ***X*** может быть таковым:

```
reset(F);
i:=1;
while not EOF(F) do
begin
   read(F,buf);
```

```
   X[i]:=buf;
   i:=i+1;
end;
```

Или наоборот, копирование элементов массива в файл:

```
reset(F);
for i:=1 to N do
  begin
    buf:=X[i];
    write(F,buf);
  end;
```

Для двумерных массивов можно придумать разные способы переноса элементов в файл. Самый простой – чтение матрицы по столбцам или по строкам. Например, так можно копировать элементы двумерного массива ***A*** в файл ***F*** построчно:

```
reset(F);
for i:=1 to N do
  for j:=1 to M do
    begin
      buf:= A[i,j]
      write(F,buf);
    end;
```

Обратная операция представляется более сложной ввиду гипотетической невозможности формирования двумерного массива из элементов файла. Если длина файла не равна произведению строк и столбцов матрицы, то при ее формировании возникает множество вопросов: «А действительно ли в файле содержится матрица?», «В какой именно части файла она содержится?», «Может можно сформировать матрицу только из начальных элементов матрицы?». Рассмотрим случай формирования матрицы ***A*** размером ***N***×***M*** из файла ***F***:

```
reset(F);
if fileSize(F) >= N*M then
  begin
    for i:=1 to N do
      for j:=1 to M do
        begin
          read(F,buf);
          A[i,j]:=buf;
        end;
```

```
  end
else
  begin
    writeln('В файле F недостаточно компонент');
    writeln('в файле: ', FileSize(F),' компонент');
    writeln('в матрице: ', N*M ,' 'элементов');
  end;
```

Для иллюстрации одновременной работы с файлами и массивами рассмотрим решение нескольких задач. Пусть, например, нужно переписать из двумерного массива ***A*** в файл ***F*** элементы в последовательности изображенной ниже, т. е. «по змейке».

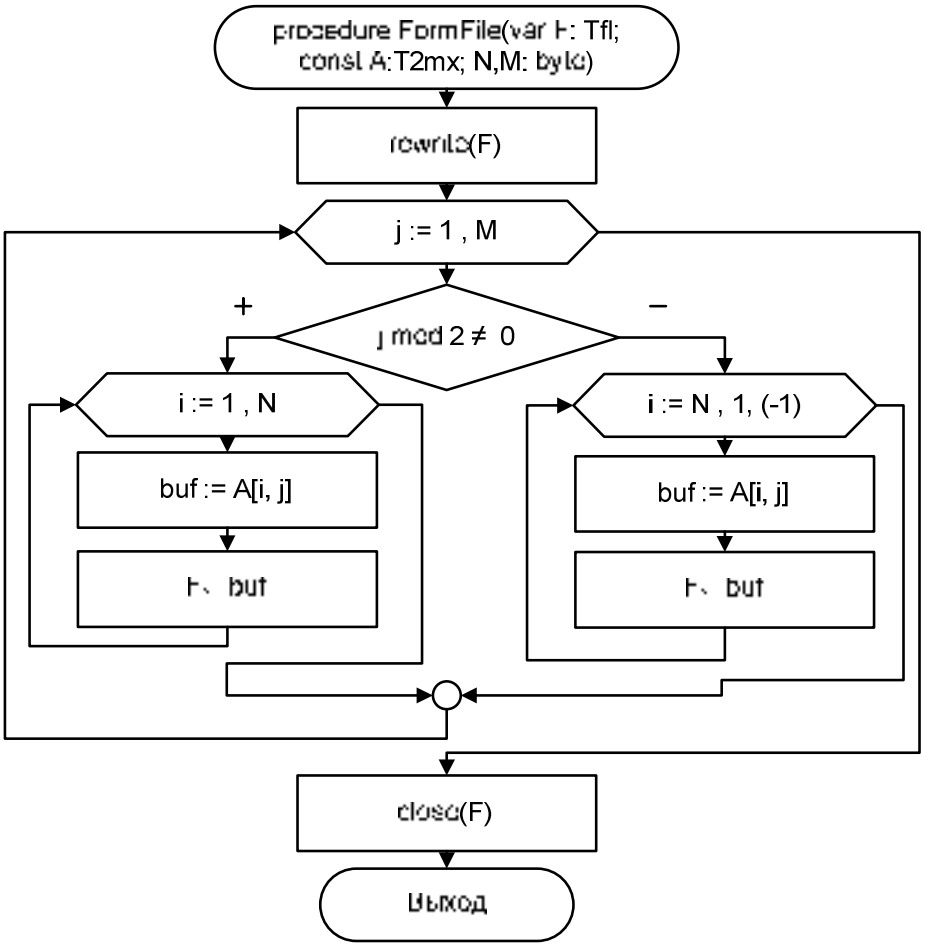
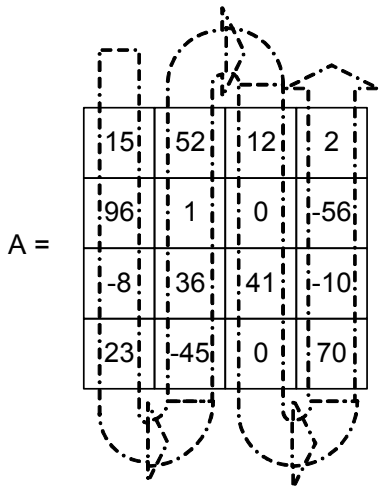


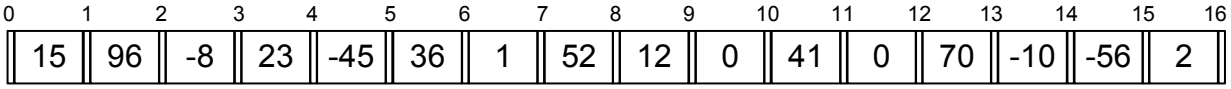
Рис. 7.18. Формирование файла из матрицы «по змейке»

Не приводя стандартных процедур ввода/вывода матрицы и вывода файла, продемонстрируем процедуру собственно формирования файла, изображенную на рис. 7.18.

Рассмотрим полное решение достаточно сложной задачи на совместное использование файлов и массивов.



В результате получается файл F:



ПРИМЕР

*Задан файл **F**, в котором содержатся элементы матрицы. Причем известно, что сначала записано число строк, далее число столбцов, а далее элементы матрицы построчно. Нужно: 1) восстановить матрицу; 2) отнормировать ее (разделить все элементы на значение максимального); 3) занести матрицу обратно в файл; 4) переписать все положительные элементы из файла в одномерный массив **B**.*

Тестовый пример показан на рис. 7.19.

Блок-схемы основной программы и всех процедур представлены на рис. 7.20 – 7.23.

По блок-схеме составим программу.

```
program Files_3;
type Tfl = file of real;
     T2mx = array[1..10, 1..10] of real;
     T1mas = array[1..100] of real;
var  F: Tfl;
```

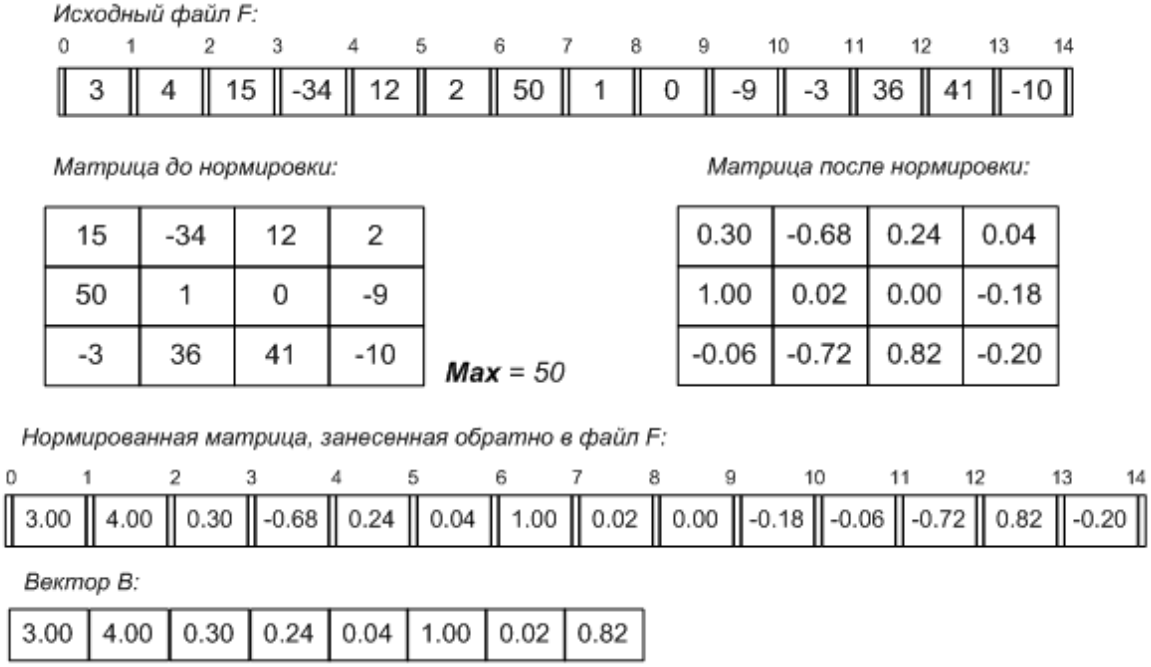


Рис. 7.19. Тестовый пример к задаче

```
 A: T2mx;
     B: T1mas;
     Na,Ma,K: byte;


{******Процедура ввода файла*******}
procedure InputFile(var X:Tfl; const Name:char);
var i,N,M: byte;
    buf: real;
begin
  rewrite(X);
  writeLn('N, M равны :');
  readLn(N,M);
  while (N<1) or (M<1) do
    begin
      writeLn('N и M должны быть положительными');
      writeLn('N, M равны :');
      readLn(N,M);
    end;
  write(X,N,M);
  for i:=1 to N*M do
    begin
      write('комп. файла ',Name,' равна: ');
```

```
      readLn(buf);
      write(X,buf);
    end;
  close(X);
end;
{****** Процедура вывода файла ******}
procedure OutputFile(var X:Tfl;const Name:char);
var buf:real;
begin
  writeLn('Вывод файла ', Name);
  reset(X);
  while not EOF(X) do
    begin
      read(X, buf); write(buf:7:2);
    end;
  close(X);
end;

{****** Процедура формирования матрицы ******}
procedure FormMatrix(var X: Tfl; var Matr: T2mx;
                     var N,M: byte);
var i,j: byte;
    buf1, buf2, buf: real;
begin
  reset(X);
  read(X,buf1,buf2);
  N:= trunc(buf1);
  M:= trunc(buf2);

for i:= 1 to N do
    for j:= 1 to M do
      begin
        read(X,buf);
        Matr[i,j]:=buf;
      end;
  close(X);
end;
{****** Процедура вывода матрицы ******}
procedure OutputMatrix(const Matr:T2mx;
                       const N,M: byte);
var i,j : byte;
begin
  writeLn('Матрица:');
```

```
  for i:= 1 to N do
    begin
      for j:= 1 to M do
        write(Matr[i,j]:7:2);
      writeLn;
    end;
end;
{****** Процедура нормировки матрицы ******}
procedure NormMatrix(var Matr:T2mx; const N,M: byte);
var i,j : byte;
    Max: real;
begin
  max:= Matr[1,1];
  for i:= 1 to N do
    for j:= 1 to M do
      if Matr[i,j] > Max then
        Max:= Matr[i,j];
  for i:= 1 to N do
    for j:= 1 to M do
      Matr[i,j]:= Matr[i,j]/Max;
end;
{**** Процедура записи матрицы обратно в файл ****}
procedure FormFile(var X:Tfl; const Matr:T2mx;
                   const N,M:byte);
var i,j : byte; buf: real;
begin
  rewrite(X);  write(X,N,M);
  for i:= 1 to N do
    for j:= 1 to M do
      begin
        buf:=Matr[i,j];
        write(X,buf);
      end;
  close(X);
end;
{****** Процедура формирования вектора ******}
Procedure FormVector(var X: Tfl; var Vect:T1mas;
                     var I:byte);
var buf:real;
begin
  reset(X);
  I:= 0;
  while not EOF(X) do
```

```
    begin
      read(X, buf);
      if buf > 0 then
        begin
          I:= I + 1;
          B[I] := buf;
        end;
    end;
  close(X);
end;

{**** Процедура вывода полученного вектора ****}
procedure OutputVector(const Vect:T1mas; const N:byte);
var i: byte;
begin
  writeLn('Полученный вектор:');
  for i:= 1 to N do
    write (B[i]:7:2);
end;

{****** Основная программа ******}
begin
  assign(F,'fileF.xxx');
  InputFile(F,'F');
  OutputFile(F,'F');
  FormMatrix(F,A,Na,Ma);
  OutputMatrix(A,Na,Ma);
  NormMatrix(A,Na,Ma);
  OutputMatrix(A,Na,Ma);
  FormFile(F,A,Na,Ma);
  OutputFile(F,'F');
  FormVector(F,B,K);
  OutputVector(B,K);
end.
```

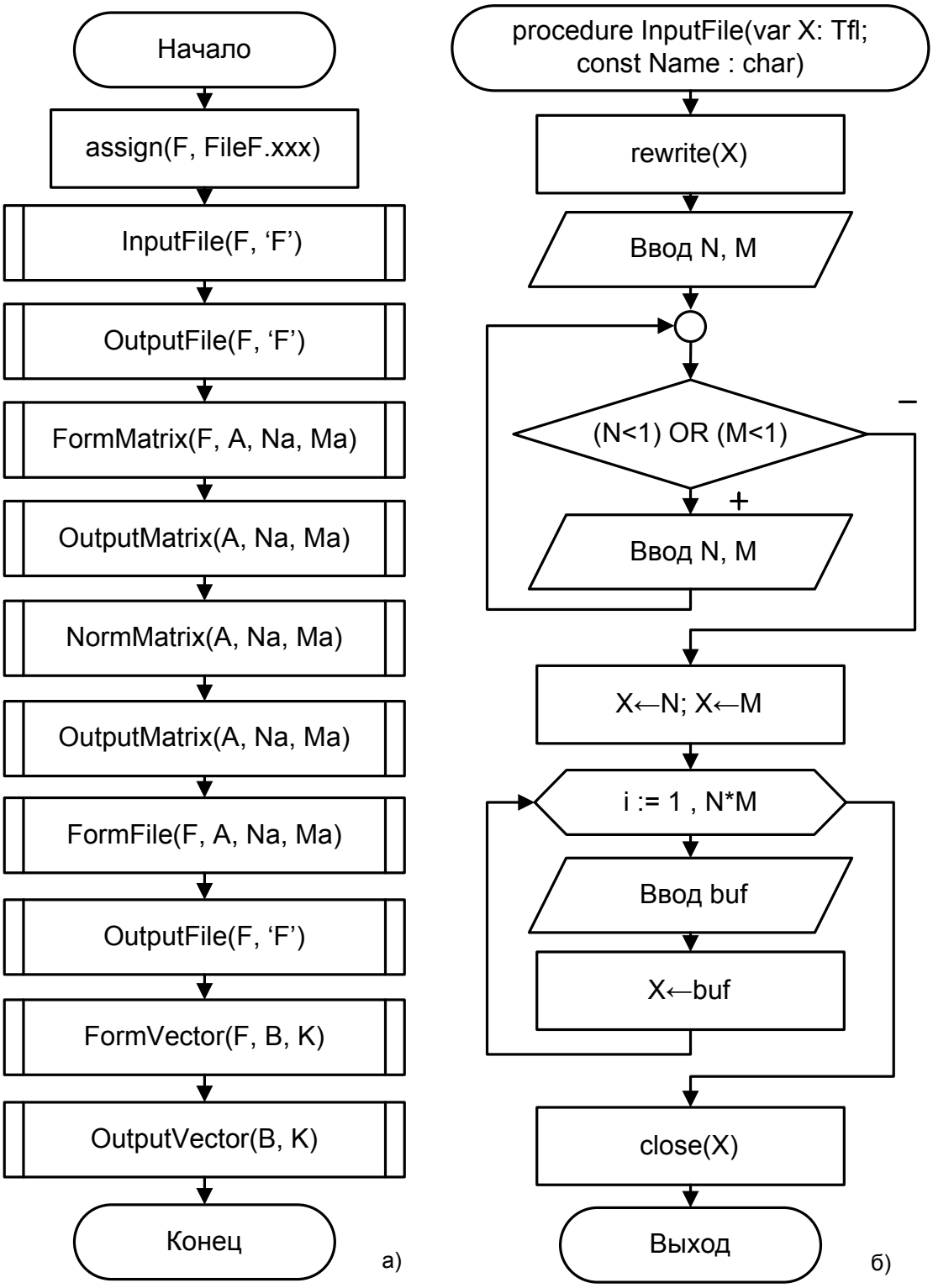


Рис. 7.20. Главная программа (*а*) и формирование файла (*б*)

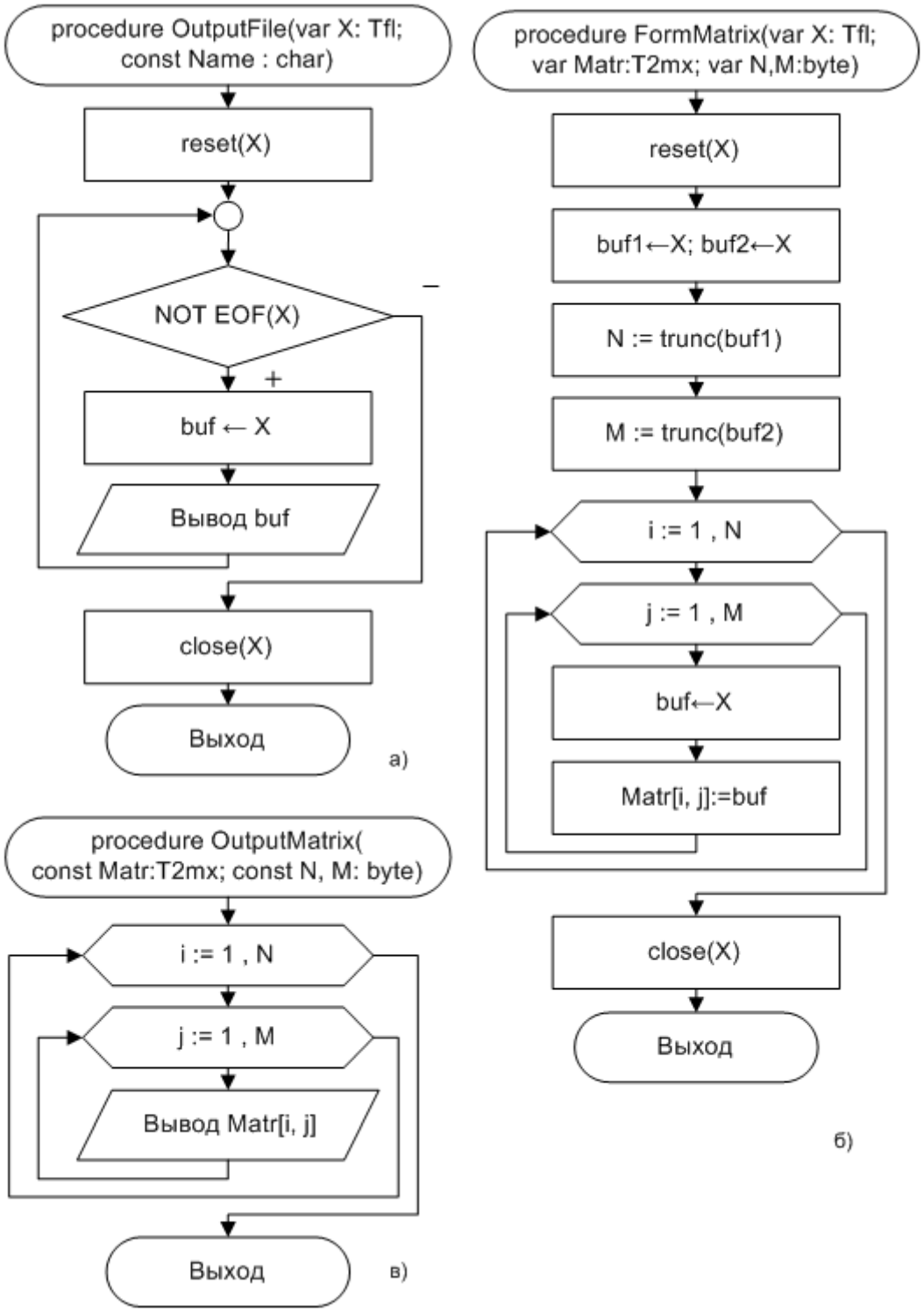


Рис. 7.21. Процедура вывода файла (а); процедура формирования матрицы из файла (*б*); процедура вывода матрицы (*в*)

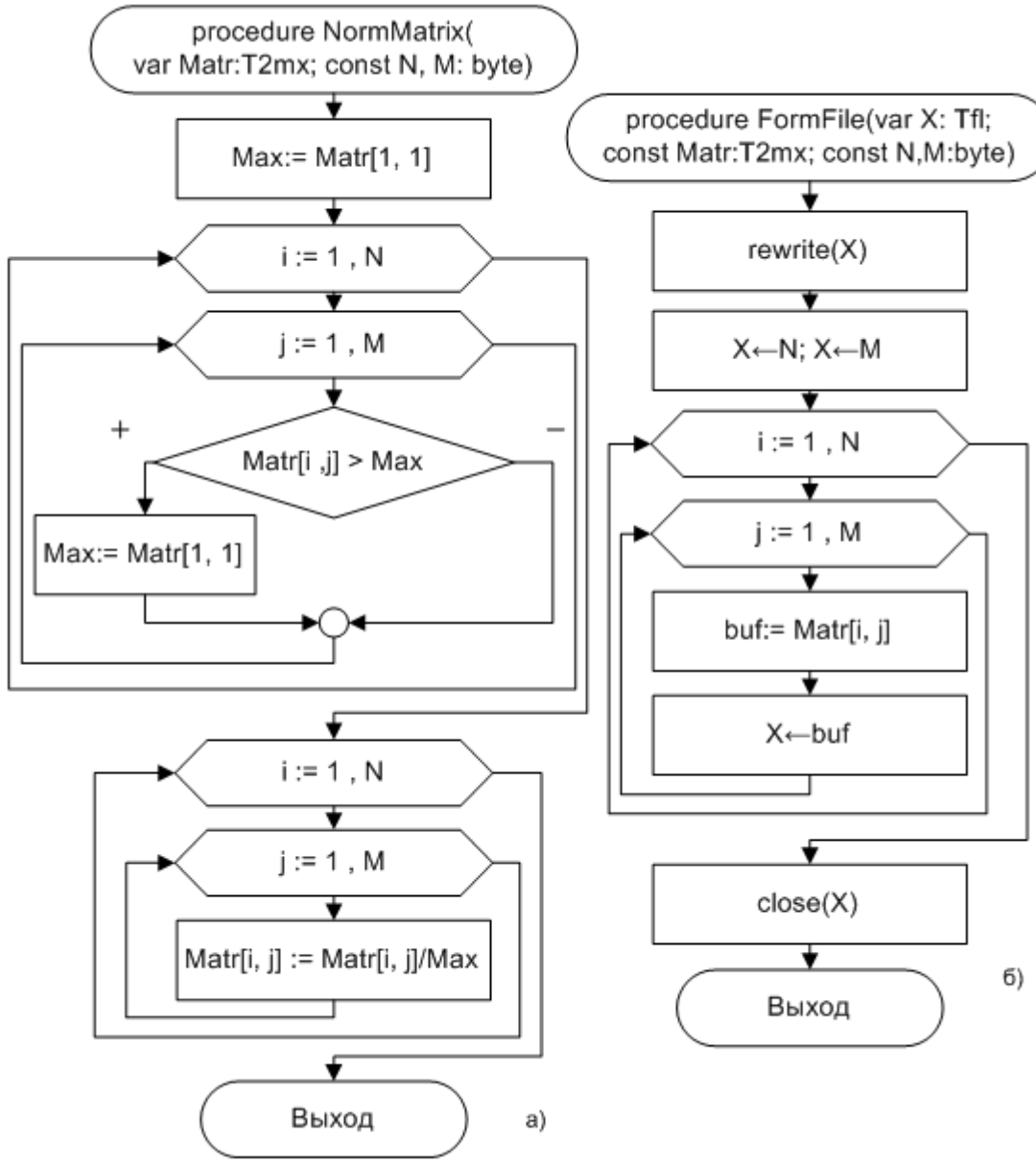


Рис. 7.22. Нормировка матрицы (*а*); формирование файла из матрицы (*б*)

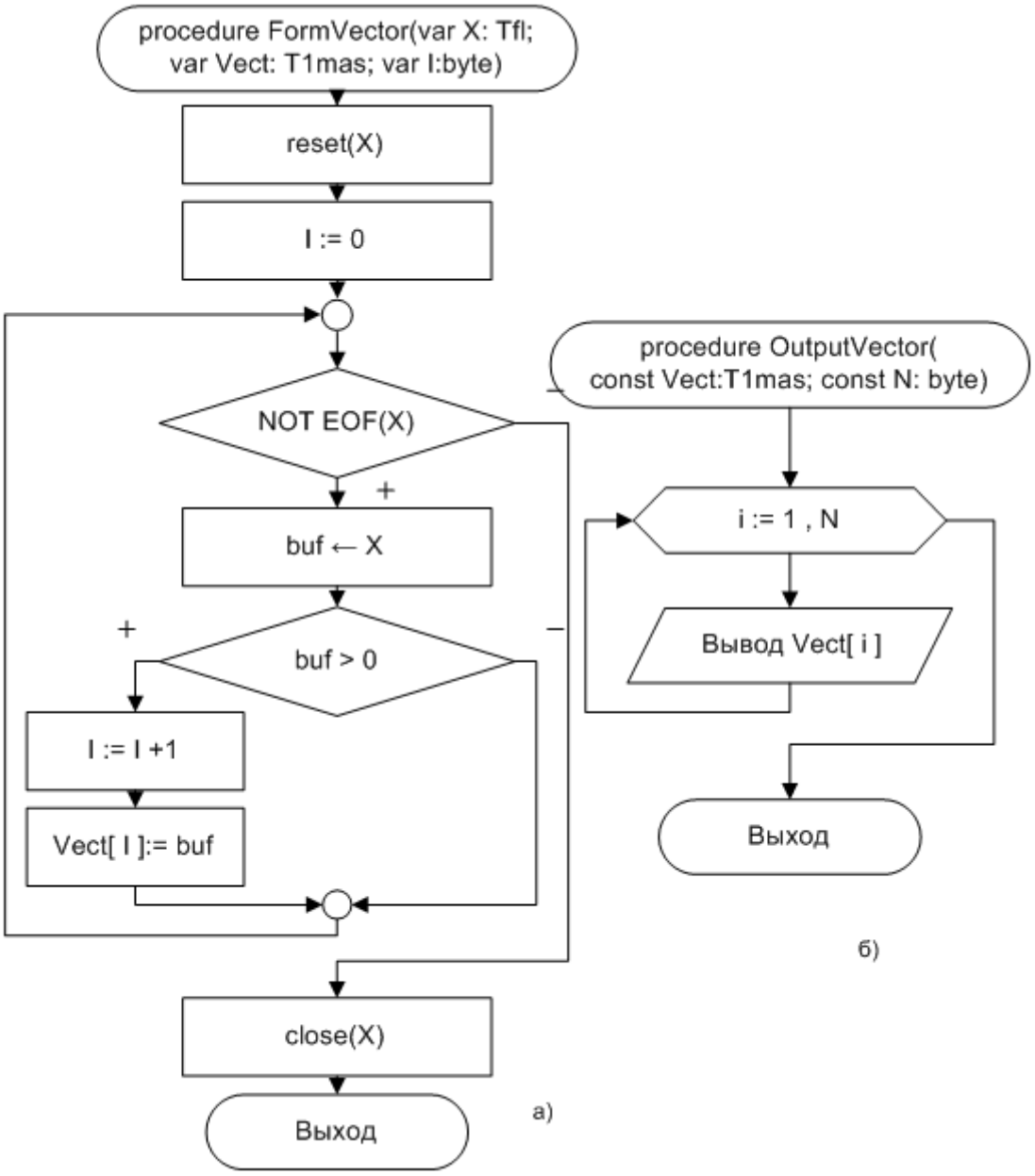


Рис. 7.23. Формирование вектора (*а*); вывод вектора (*б*)

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Обработка записей средствами языка Паскаль : учеб. пособие / Р. С. Богатырев, О. В. Гостевская, И. Г. Лемешкина, В. Ю. Наумов, Е. С. Павлова ; ВолгГТУ. – Волгоград, 2008. – 54 с.

2. Сложные структурированные типы данных в языке ТР: массивы : учеб. пособие / О. А. Авдеюк, О. В. Гостевская, С. Р. Калмыкова, Е. С. Павлова ; ВолгГТУ. – Волгоград, 2006. – 64 с.

3. *Златопольский, Д. М.* Сборник задач по программированию / Д. М. Златопольский. – 2-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2007. – 240 с.

4. Программирование на языке Паскаль : Задачник / под ред. О. Ф. Усковой. – СПб : Питер, 2003. – 336 с.

5. *Хармут, Х.* Применение методов теории информации в физике : пер. с англ. / Х. Хармут. – М.: Мир, 1989. – 344 с.

6. *Фаронов, В. В.* Turbo Pascal 7,0. Начальный курс : учеб. пособие / В. В. Фаронов. – М.: Нолидж, 2001. – 576 с.

7. *Андерсон, Д.* Дискретная математика и комбинаторика : пер. с англ. / Д. Андерсон. – М.: Вильямс, 2004. – 960 с.

8. Информатика и программирование: руководство к лабораторным и практическим занятиям. Ч. 1 : учеб. пособие / Авдеюк О.А., Акулов Л.Г., Гостевская О.В., Лемешкина И.Г., Павлова Е.С., Наумов В.Ю.; ВолгГТУ. – Волгоград, 2014. – 80 с.

9**.** Информатика и программирование: руководство к лабораторным и практическим занятиям. Ч. 2 : учеб. пособие / Авдеюк О.А., Акулов Л.Г., Гостевская О.В., Лемешкина И.Г., Павлова Е.С., Наумов В.Ю.; ВолгГТУ. – Волгоград, 2014. – 60 с.

10. *Муха, Ю.П.* Информатика. Ч. 1. Теория информации и кодирования : Конспект лекций по дисциплине «Информатика» : учеб. пособие / Муха Ю.П., Авдеюк О.А., Новицкий А.С. ; ВолгГТУ. – Волгоград, 2004. – 76 с.

11. Информационные технологии в профессиональной деятельности : Конспект лекций : учеб. пособие / Авдеюк О.А., Акулов Л.Г., Гостевская О.В., Королева И.Ю., Наумов В.Ю., Скворцов М.Г. ; ВолгГТУ. – Волгоград, 2014. – 80 с.

12. *Акулов, Л.Г.* Информатика: основы программирования на языке Паскаль : учеб. пособие / Л.Г. Акулов, В.Ю. Наумов, О.А. Авдеюк ; ВолгГТУ. – Волгоград, 2017. – 268 с.

Учебное издание

Оксана Алексеевна **Авдеюк**
Леонид Геннадьевич **Акулов**
Вадим Юрьевич **Наумов**

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ:
ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПАСКАЛЬ

Учебно-методическое пособие

2-е издание, исправленное

Выпускающий редактор *Л. Н. Рыжих*

Темплан 2018 г. (учебники и учебные пособия). Поз. № 104.
Подписано в печать 22.03.2018. Формат 60x84 1/16. Бумага газетная.
Гарнитура Times. Печать офсетная. Усл. печ. л. 15,58. Уч.-изд. л. 11,65.
Тираж 200 экз. Заказ

Волгоградский государственный технический университет.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 1.

Отпечатано в типографии ИУНЛ ВолгГТУ.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 7.