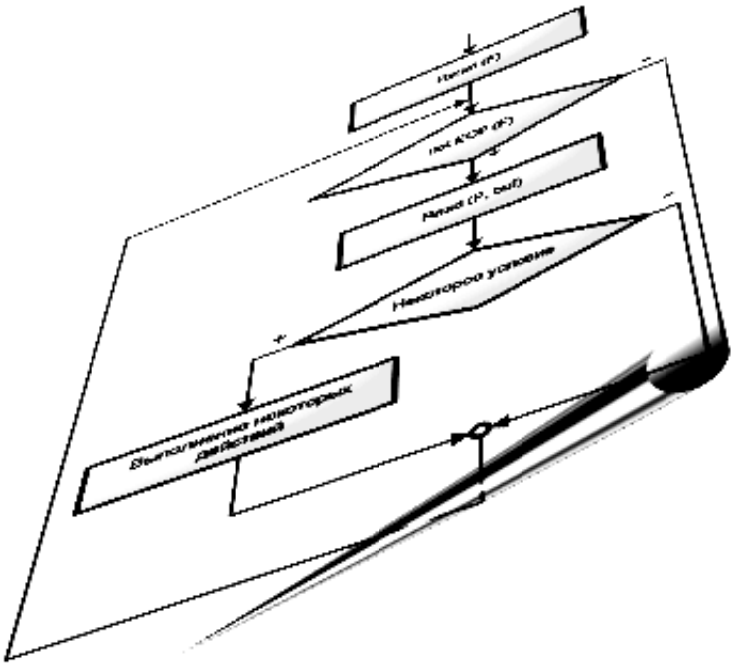
Волгоградский государственный технический университет

Л. Г. Акулов, Р. С. Богатырёв, В. Ю. Наумов

Введение в информатику. Основы программирования на языке Pascal



Волгоград 2008

СОДЕРЖАНИЕ

ОТ АВТОРОВ

Конец двадцатого века по праву называется эпохой информационных технологий. Благодаря прогрессу в области электронной техники общество совершило качественный скачок по кривой своего развития.

За последние десятилетия персональный компьютер превратился из экзотики в достаточно заурядную вещь, встречающуюся дома практически у каждого. Более того, современные тенденции разработки техники таковы, что для управления ею все чаще применяют сложные конструкции, в основе работы которых лежит цифровое электронное устройство. Мобильные телефоны, плееры, часы и даже стиральные машины и в настоящее время находятся под управлением небольших встроенных вычислительных устройств. Микропроцессор управляет подачей топлива в двигатель автомобиля, антиблокировочной системой, распределяет нагрузку на электрические сети в электроэнергетике, выбирает режимы работы станка, иные большие и малые задачи возложены сегодня на электронику.

Цифровые технологии настолько прочно вошли в нашу жизнь, что мы перестали задумываться над тем какие принципы в них заложены. Этот факт, с одной стороны, может считаться положительным, поскольку позволяет не размышлять об устройстве инструмента, а сосредоточиться на творческом процессе, а с другой, весьма печален. Печаль вызвана тем, что любая техника априори несовершенна и потому имеет достаточно четкие границы своего применения. Незнание принципов заложенных в основу ее работы приводит к завышенным требованиям, к необоснованным надеждам на простое решение, или просто к неэффективному использованию.

В основе любого цифрового устройства лежат достаточно простые законы. Эти законы – законы машинной логики. Логика машины отчасти похожа на логику человека, однако, есть масса отличий. Знание этих законов позволяет эффективно ставить задачу устройсву-исполнителю и принимать результат в ожидаемой форме.

Для постановки задачи нужно иметь представление об алгоритмах (о последовательности действий), об их свойствах. Правильно составленный алгоритм дает надежду на достаточно быстрое и эффективное решение задачи.

В данном пособии рассмотрение задач на составление алгоритмов ведется на примере языка высокого уровня *Pascal*. В настоящее время язык *Pascal* имеет большое число различных реализаций. Самая популярная версия в нашей стране, да и, наверное, в мире – это система программирования *Turbo Pascal 7.0* фирмы *Borland*. Именно на этой версии языка основывались и продолжают основываться многие учебные планы программ по основам программирования.

Конечно, *Pascal 7.0* как система серьезных программных разработок сейчас не используется, более того, для начального обучения существуют

более простые языки. Однако, былая популярность и некоторая инертность не позволяют вычеркнуть его из учебных планов. *Pascal* существует как язык уже не одно десятилетие и не думает прекращать свое развитие. В настоящее время фирма *Borland* основываясь на синтаксисе *Pascal* поддерживает *Delphi* – современный этап развития *Pascal*. Как в нашей стране, так и в мире *Delphi* является одной из наиболее популярных систем программирования как на любительском, так и на профессиональном уровне. Потому разговоры о бесперспективности изучения *Pascal* на начальном этапе не выдерживают никакой критики.

Более того, в этом пособии внимание акцентируется не на конкретном языке, а на алгоритмике процесса. Мы пытались составлять решения задач вообще опосредованно к языку программирования. Для этой цели, как нельзя кстати, пришелся аппарат представления алгоритмов в виде блок-схем. Блок-схемы позволяют «нарисовать» алгоритм, увидеть его целиком, а уж как и на чем он будет реализован – это отдельная задача, причем достаточно часто тривиальная и в большинстве случаев неинтересная.

Именно умение корректно составить алгоритм определяет успех при управлении той или иной системой. Причем речь идет не только о составлении программ для компьютера, но и при планировании прочих действий, включая управление людьми. Неправильно поставленная, некорректная, неоптимальная, недостаточно или наоборот излишне детализированная задача по выполнению последовательности каких-либо действий может привести к ее неисполнению.

Цель при решении задач на программирование состоит не в том, чтобы получить конечное значение какой-либо величины, а в том, чтобы научить компьютер самостоятельно искать эту величину, тем самым избавляя себя и других людей от выполнения рутинных действий.

Материал изложенный в настоящем пособии читается уже многие годы для студентов-первокурсников ВолгГТУ в рамках учебной дисциплины «информатика». Подход к изучению и последовательность изложения материала показали свою жизнеспособность.

Основанием создания учебного пособия послужило отсутствие литературы отражающей особенности преподавания информатики как общетехнической дисциплины на кафедре Вычислительная техника ВолгГТУ. Несмотря на обилие книг по программированию, подход основанный на преимуществе блок-схемы перед программным кодом встречается достаточно редко. Причиной тому служит ориентированность на подготовленного читателя, на читателя желающего самостоятельно обрести навыки программирования в той или иной системе программирования, на том или ином языке.

Целью данной книги является изложение учебного материала по информатике в достаточно компактной форме. Даже если студент что-то не понял на лекции, то он всегда сможет обратиться к книге для уточнения пропущенного материала, материала плохо изложенного в его собственном конспекте.

Изложение материала ведется с максимально возможным упрощением. Причина этому – ухудшающийся год от года уровень знаний вчерашних школьников. Демографическая обстановка, социально-экономический кризис 90-х годов прошлого века не могли не отразиться на качестве подготовки абитуриентов. Курс информатики читается таким образом, как будто ранее он не изучался. Даже несмотря на упрощение, опыт последних лет показывает, что и с этой программой могут справиться далеко не все обучаемые. Причины и следствия образовательной деградации оставим за рамками книги.

При создании пособия ценные замечания высказывали *ИМЯРЕК*, за что им отдельная благодарность.

Книга ограничена по объему, потому здесь не изложены вопросы создания пользовательских библиотек, не рассмотрено объектно-ориентированное программирование, нет задач на графы, на связанные списки, на рекурсию, на специальные алгоритмы обработки нестандартных массивов. Все это здесь не уместилось, да и стандартный объем часов по информатике достаточно небольшой, однако есть надежда на скорый выход в свет продолжения, в котором все эти вещи по-возможности будут освещены.

март 2008 г.

1. БАЗОВЫЕ СВЕДЕНИЯ ОБ ИНФОРМАТИКЕ, ПРОГРАММАХ И ЭВМ

1.1 Общие сведения

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

Информация – способность одной системы оценивать поведение другой системы. Способность различать что-либо.

Информатика – наука о средствах сбора, хранения, передачи и обработки информации выраженной в количественном виде.

Информационные процессы – действия выполняемые над информацией, т.е. процессы получения, передачи, преобразования, хранения и использования информации.

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 объём сообщения – количество бит в этом сообщении.

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

1 кбит/с = 1000 бит/с
1 Мбит/с = 1000 Кбит/с
1 Гбит/с = 1000 Мбит/с
1 Тбит/с = 1000 Гбит/с

Вообще, процессы, происходящие в физическом мире, принято считать непрерывными. Для того, чтобы можно было говорить об их информационной обработке, необходимо чтобы физическому процессу было сопоставлено число. Процесс сопоставления чисел непрерывным физическим процессам называется *оцифровкой*. Очевидно, что оцифровка не полностью

[1] Помимо 8-ми 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.

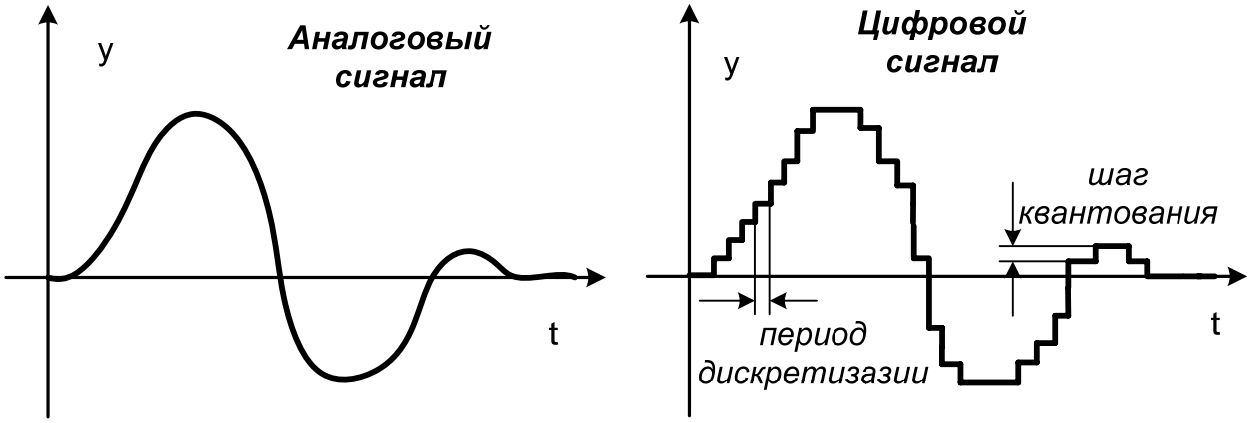


Рисунок 1.1 Аналого-цифровое преобразование сигнала

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 можно ограничиться малым разрешением.

Оцифровку можно наглядно представить, например, при переводе аналоговой записи (магнитофонной кассеты) в цифровую форму (Рисунок 1.1). 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 называется *периодом дискретизации*.

1.2 Двоичная система исчисления

Исторически сложилось, что люди используют десятичную систему исчисления, т.е. систему исчисления, 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 исчисления. 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 нём весь цифровой алфавит по тем же правилам, которые применялись для формирования двузначных чисел.

Аналогичным образом формируются натуральные числа в любой другой системе исчисления, например, в двоичной (Таблица 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

Компьютерные микропроцессоры представляют собой схему из миллионов соединённых транзисторных ключей, которые могут находиться лишь в двух состояниях – открыто или закрыто. Если соотнести 0 с состоянием, например, открытого транзисторного ключа, а 1 – с состоянием закрытого, то двоичная система исчисления может быть удобной для представления числовой информации в компьютерах.

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.

Методы кодировок основаны на свойствах человеческого зрения и слуха. Например, чтобы закодировать звук так, чтобы среднестатистическое человеческое ухо не уловило потери качества, достаточно измерять и записывать амплитуду акустического колебания каждую 1/44000 секунды. Далее достаточно воспроизвести последовательность записанных звуковых амплитуд с частотой 44 КГц для того, чтобы услышать неискаженную музыку или речь. Потеря качества в действительности существует, но мы не ощущаем ее. Это связано с тем, что естественный способ воспроизведения звуков непрерывен, то есть, можно сказать, что природа воспроизводит звуки с бесконечной частотой дискретизации, а не 44 КГц, но человеческое ухо не

сможет отличить разницы.[1] Это является примером дискретизации по времени. Кроме того, для перевода звука в цифру необходимо произвести квантование по амплитуде. Наиболее распространенным является 16-битное квантование, т.е. мы можем отчетливо различить 2^{16} = 65536 различных позиций амплитуды звукового колебания (см. Рисунок 1.1).

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-ми битная таблица *ASCII* (говорят «*АСКИ*»), в которой описано 256 кодов соответствующих, в том числе, латинским и русским буквам. Или 16-ти 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.

Подробнее рассмотрим Таблица 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

[1] Это связано с механическими особенностями строения слухового аппарата человека. В частности, этот диапазон совпадает со спектром собственных частот звуковоспринимающей и звукопроводящей систем уха. Кроме частотной восприимчивости существует еще и амплитудная восприимчивость, показывающая верхнюю и нижнюю границу чувствительности уровня акустического давления.

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.

Для начала составим таблицу натуральных степеней двойки (Таблица 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 Таблица 1.2 число ближайшее к 1247 снизу, таким числом является 1024.

1247=1024+223.

Теперь найдем в Таблица 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.

1.3 Понятие информации

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.

Подход описанный далее впервые был осуществлен Хартли[1] 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 Рисунок 1.2. Допустим, 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 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

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

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

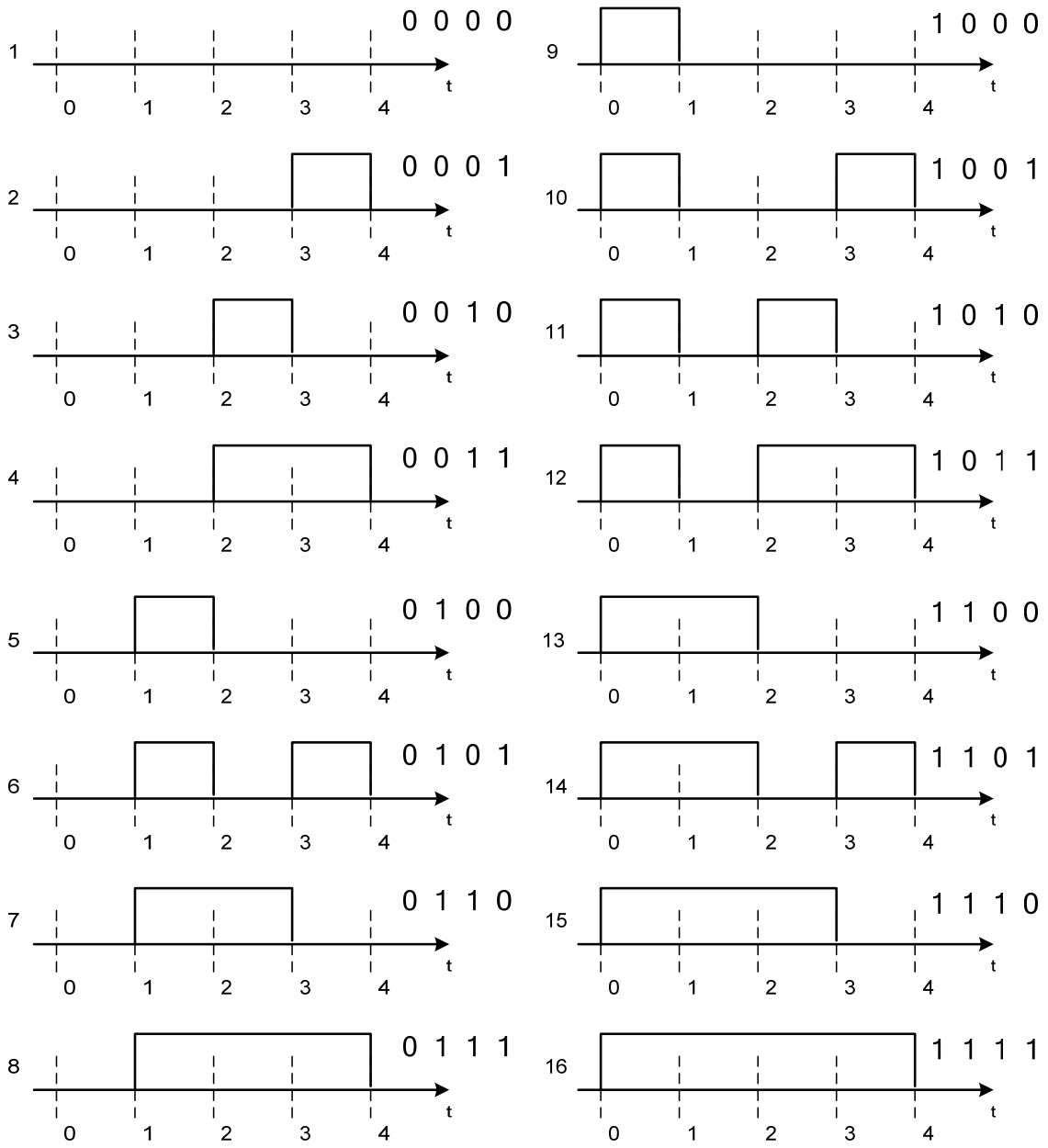


Рисунок 1.2 Кодирование всех возможных вариантов 4-х битного сообщения

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 $H = \log_2\left(2^4\right) = 4$ (см. Рисунок 1.2).

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.[1] 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, что $p_{"1"} = p_{"0"} = 1/2$, т.к. полная вероятность единица, т.е. $p_{"1"} + p_{"0"} = 1$.

Обобщение для случая $p_{"1"} \neq p_{"0"}$ было введено в 1948 г. К. Шенноном[2]:

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

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***.

1.4 Программное обеспечение

Следует отметить, что в современных вычислительных системах выделяют два основных компонента: ***программная часть*** (*software*) и

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

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

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 3 группы (см. Рисунок 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

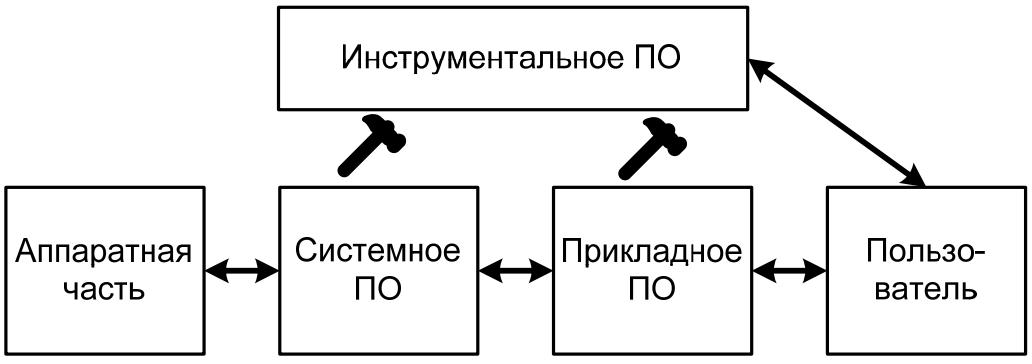


Рисунок 1.3 Взаимодействие между различными видами программного обеспечения и человеком

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.

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 другие. 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 настоящей книги 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 (см. приложение). 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.

1.5 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 память, внешняя память и различные периферийные устройства (см. Рисунок 1.4).

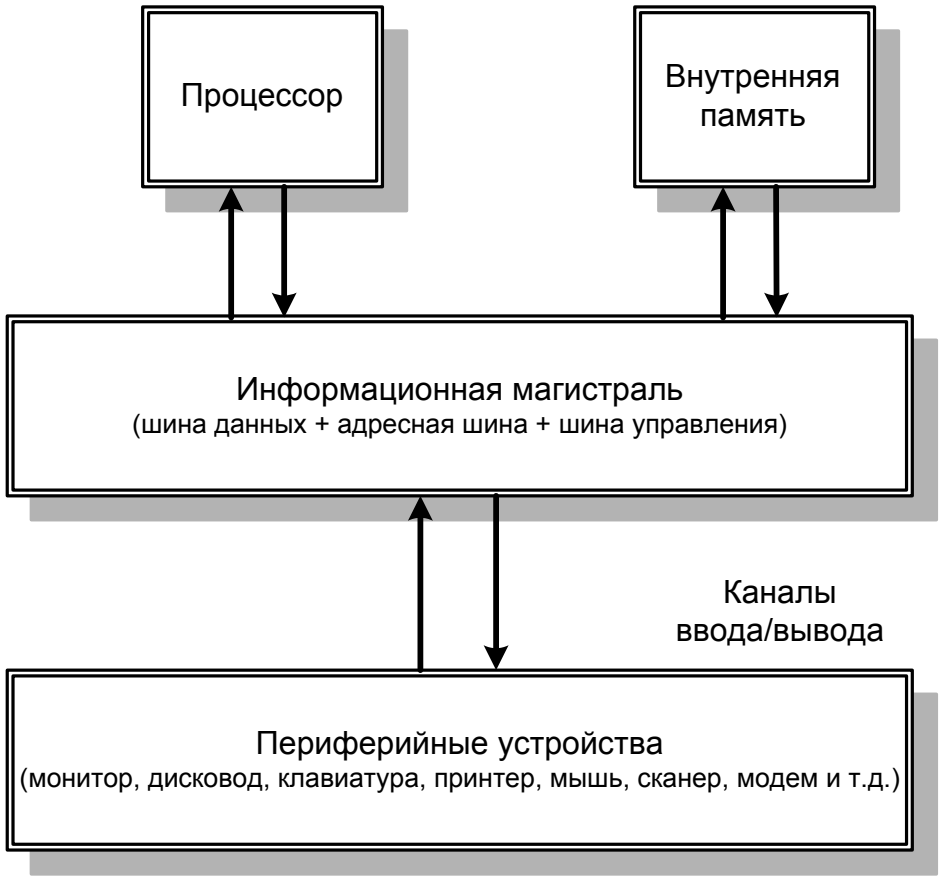


Рисунок 1.4 Магистрально-модульный принцип построения компьютера

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*[1]. 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 жёсткие 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 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 обмен данными с периферийными устройствами.

Жёсткий 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*).

Жесткий диск – самый главный носитель информации в системе. На нем хранятся все программы и данные, которые в настоящий момент не находятся в оперативной памяти. Накопитель на жестких дисках часто просто называется жестким диском, т.к. он состоит из вращающихся алюминиевых или керамических пластин, покрытых слоем ферромагнетика. Размеры винчестеров могут быть разные. Емкость памяти дисковода жестких дисков зависит от плотности, размеров и количества пластин. В большинстве настольных компьютеров сегодня используются дисководы с жесткими дисками, рассчитанными на *формфактор* 3,5”, в то время как в большинстве

[1] 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.

портативных компьютеров (в лэптопах и ноутбуках) используются жесткие диски с формфактором 2,5” и даже 1”.

Накопители на жестких дисках обычно называют *винчестерами*. Этот термин появился в 60-е годы, когда фирма IBM выпустила высокоскоростной накопитель с одним несъемным и одним сменным дисками емкостью по 30 Мбайт. Этот накопитель состоял из пластин, которые вращались с высокой скоростью, и «парящих» над ними головок, а номер его разработки – 30-30. Такое цифровое обозначение (30-30) совпало с обозначением популярного нарезного оружия *Winchester*, поэтому сам термин вскоре стал применяться в отношении любого стационарно закрепленного жесткого диска.

Сегодня имеется тенденция назвать винчестером любое внутреннее устройство долговременного хранения больших объемов информации. Последние модели вообще являются полностью твердотельными (полупроводниковыми) и не имеют в своем составе механических подвижных частей.

Монитор – устройство для отображения визуальной информации.

В большинстве систем монитор помещен в собственный корпус, отдельно от системного блока. В портативных компьютерах монитор встраивается в корпус системного блока компьютера. Мониторы обычно классифицируются по трем параметрам: по размеру диагонали (в дюймах), разрешающей способности (в пикселях) и частоте регенерации изображения в герцах (Гц). Разрешающая способность может быть равна от 640x480 пикселей (сначала размер по горизонтали, а затем по вертикали) до 1600x1200 пикселей, а последнее время и больше. Каждый пиксель в мониторе состоит из трех элементов-точек, по одной для каждого цвета – красного, синего и зеленого. Средний монитор способен регенерировать изображение 60-100 раз в секунду (частота 60-100 Гц), в то время как частота регенерации в более качественных мониторах превосходит эти значения. Частота регенерации показывает, как часто дисплей повторно отображает на экране содержимое видеопамяти. Сегодня мониторы на электронно-лучевых трубках становятся частью истории. В пришедших им на смену ЖК панелях (*LCD – Liquid Crystal Display* – дисплей на жидких кристаллах) частота регенерации изображения уже не играет такой роли как раньше[1], поскольку физические принципы его формирования несколько иные. К динамическим характеристикам можно отнести время переключения пикселя из одного состояния в другое. Кроме этого существует еще масса разных характеристик, влияющих на качество отображаемой монитором информации.

[1] Частота регенерации ЭЛТ мониторов и утомляемость глаз находятся в тесной зависимости, поскольку при частотах меньших либо равных 60 Гц глаз ощущает мерцание, что является негативным раздражающим фактором для нервной системы. Особенно чувствительно к динамике периферическое зрение. Ради интереса можете выставить 60 Гц на ЭЛТ мониторе и направить взгляд чуть выше. Такие физиологические особенности являются следствием эволюционного развития зрения человека, поскольку помогают быстрее реагировать на опасность. Чтобы не было «страшно» рекомендуется ставить частоту развертки в 100 Гц и выше.

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, которых 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, то есть 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, то есть 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[1] – 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 совет того, 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, эвристическое 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.

[1] 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.

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 *синтаксические диаграммы*, предложенные Н. Виртом[1] для описания синтаксиса создаваемого им языка программирования *Pascal*.

Естественный язык используется для описания, например, алгоритма приготовления кулинарных блюд (рецепт), алгоритма действий рабочих в тех или иных ситуациях на производстве (инструкции), алгоритма 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.

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

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:

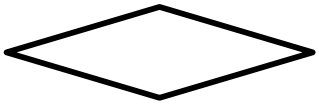
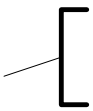
1. задать значения коэффициентов a, b, c;
2. если $a = 0$, то выполнять *шаг 8*;
3. вычислить дискриминант $D = b^2 - 4ac$;
4. если $D < 0$, то выполнять *шаг 17*;
5. вычислить действительные корни

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

6. распечатать значения x_1, x_2;
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 (Таблица 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 мы дали, 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 данном пособии принято обозначение «:=», т.е. так как в языке *Pascal*. Именно такое обозначение может быть более предпочтительным для людей раннее 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 дадим пример неправильного использования операции присваивания.

ПРИМЕР

2 := A // Ошибка!!! (нельзя изменять системную константу)

A+B := 6 //Ошибка!!! (слева записана не переменная, а выражение)

$5 - 4 := 3 + B^2$ // Ошибка!!! (слева константное выражение)

$\sqrt{A} := 1$ // Ошибка!!! (слева записано выражение)

2.4 Основы алгебры логики

Любая машина для решения алгоритмических задач, как это ни странно, выполняет некий мыслительный процесс, который называют машинной логикой. Только в отличии 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*[1] *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 предикатов выступают две дефиниции: *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 6 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, и ***FALSE*** в противном случае. Например, пусть $x = 7$, тогда операция отношения $x > 0$ даст результат ***TRUE***, то есть истинно, что $7 > 0$. А при том же значении x операция $x < 5$ даст ответ ***FALSE***, что означает ложность, утверждения $7 < 5$.

Для простоты понимания, особенно студентам ранее не изучавшим основ алгебры логики, нужно читать операции отношения с вопросительной интонацией и пытаться дать на заданный вопрос ответ *ДА* (***TRUE***) или *НЕТ*

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

(***FALSE***). То есть, 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 не просто так, а по определенным правилам. Для предикатных 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, то есть 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, т.к. 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 «$\wedge$», ***OR*** – *логическое ИЛИ* (*логическое сложение* или *дизъюнкция*, обозначается «$\vee$»), ***XOR*** – *исключающее ИЛИ* (*сложение по модулю 2*, обозначается – «$\oplus$») являются бинарными, то есть 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 «$\neg$») 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, то есть 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 операндом ***A*** и над операндом ***B*** (Таблица 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***, то есть

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

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

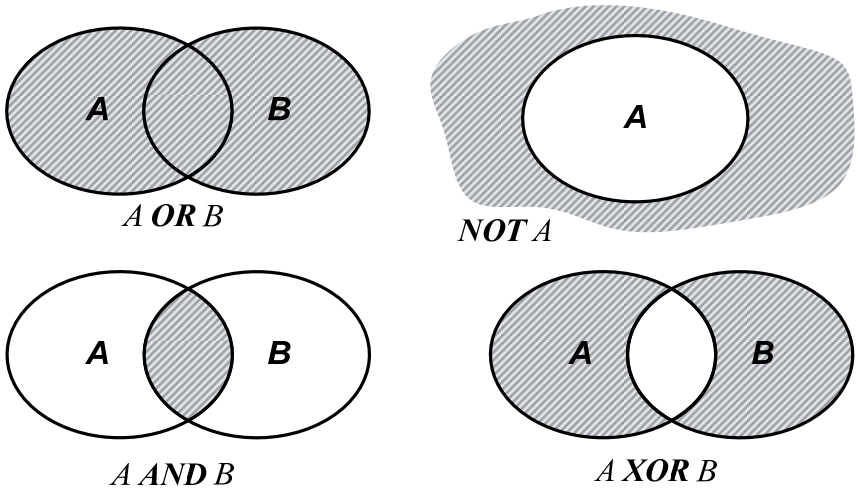


Рисунок 2.1 Связь теории множеств и булевой алгебры

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*:
   $\overline{(\text{A}\ \textbf{OR}\ \text{B})} = (\overline{\text{A}}\ \textbf{AND}\ \overline{\text{B}})$,
   $\overline{(\text{A}\ \textbf{AND}\ \text{B})} = (\overline{\text{A}}\ \textbf{OR}\ \overline{\text{B}})$;
8. правила *склеивания*:
   $(\text{A}\ \textbf{OR}\ \overline{\text{B}})\ \textbf{AND}\ (\text{A}\ \textbf{OR}\ \text{B}) = \text{A}$,
   $(\text{A}\ \textbf{AND}\ \overline{\text{B}})\ \textbf{OR}\ (\text{A}\ \textbf{AND}\ \text{B}) = \text{A}$.

2.5 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[1] в 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.

Линейные вычислительные процессы

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 Рисунок 2.2.

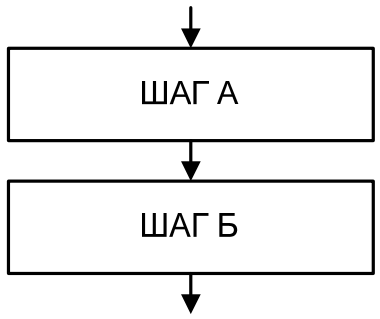


Рисунок 2.2 Управляющая структура "следование"

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

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 (Рисунок 2.3). Причем выходы направляют

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

влево и вправо, а ни в коем случае не вниз (чтобы иметь возможность отличать развилки от циклов).

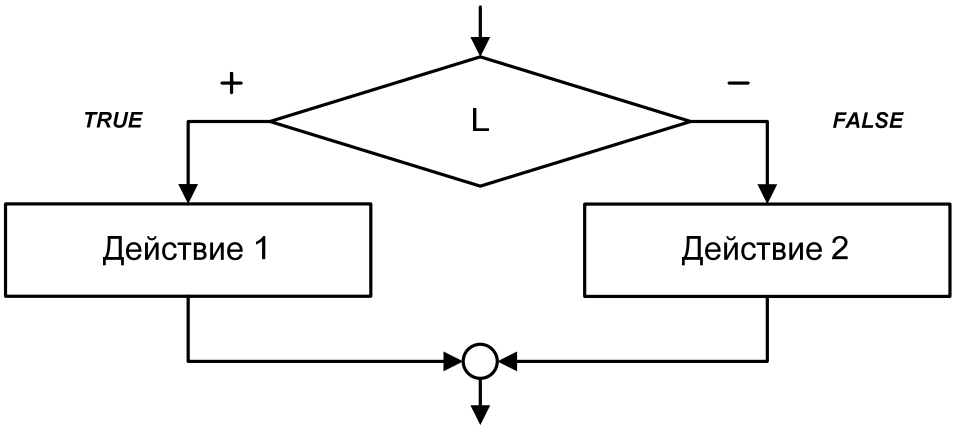


Рисунок 2.3 Полная развилка

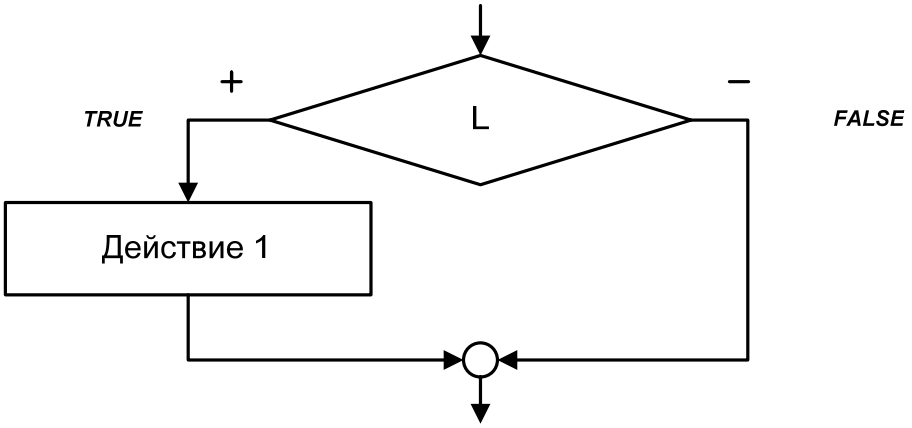


Рисунок 2.4 Неполная развилка

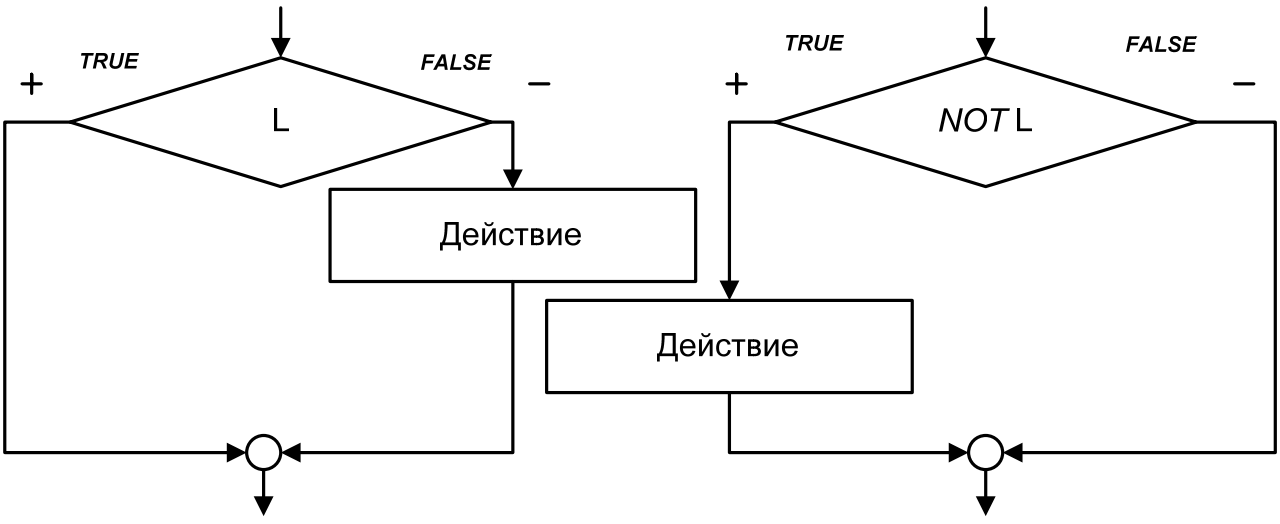


Рисунок 2.5 Преобразование развилки с пустой положительной ветвью в развилку с непутой положительной ветвью

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

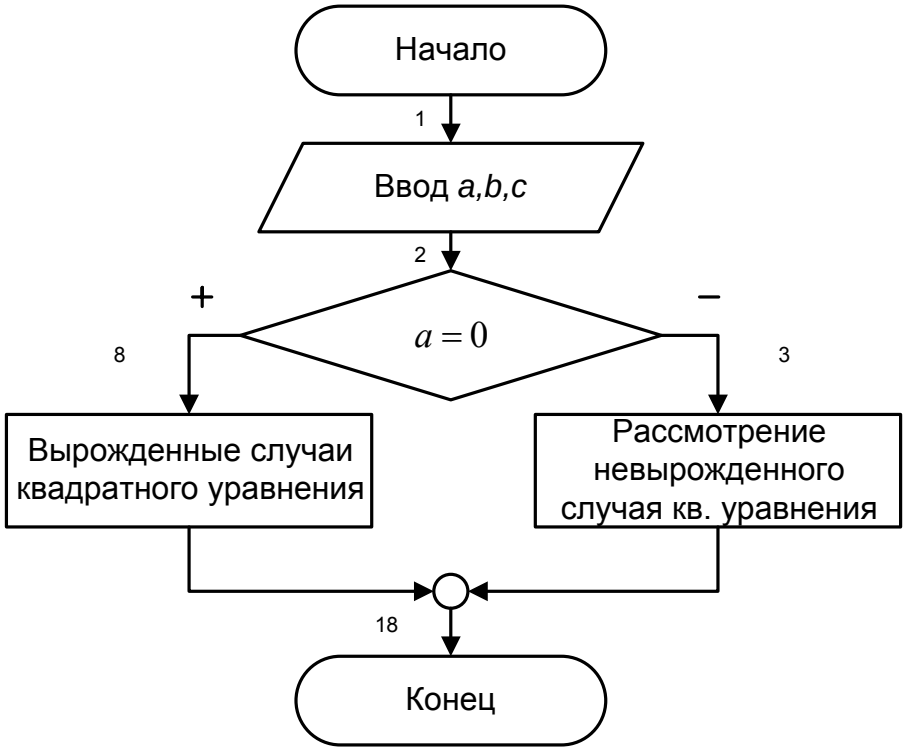


Рисунок 2.6 Общая блок-схема решения квадратного уравнения

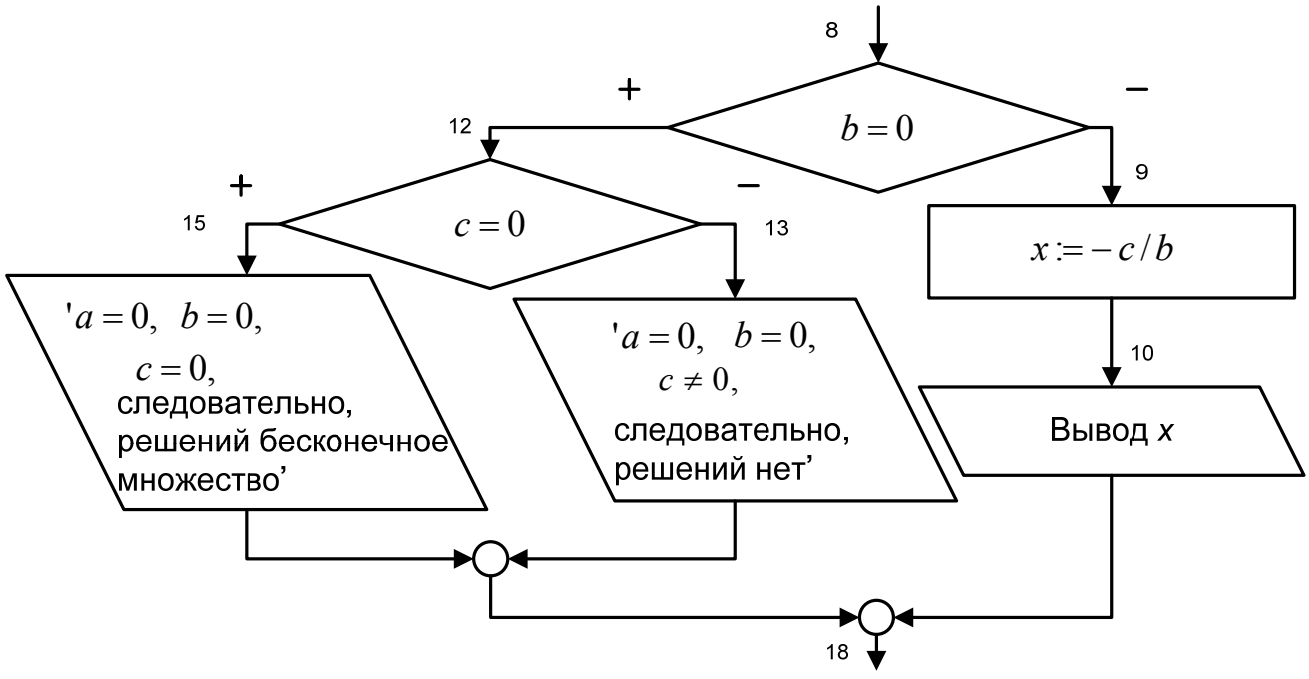


Рисунок 2.7 Шаг 2-18 алгоритм решения вырожденного случая квадратного уравнения

том, что для ее реализации достаточно инвертировать логическое выражение *L* (применить операцию логического отрицания ***NOT***) и мы придем к обычной структуре с непустой положительной ветвью (Рисунок 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.

ПРИМЕР

Рассмотрим более сложный пример. Построим блок-схему алгоритма с использованием развилок (Рисунок 2.6). Пусть это будет блок-схема алгоритма решения квадратного уравнения $ax^2 + bx + c = 0$, описанного выше, в разделе «*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 пунктах описано стрелками, направленными к *шагу 18*, соответственно, от *6*-го, *10*-го, *13*-го и *15*-го шагов.

Еще одной особенностью изображения блок-схемы (Рисунок 2.7) 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, мы сначала рассматриваем проблему в общем (говорим что нужно ввести коэффициенты уравнения a,b,c и далее в зависимости от введенных данных решаем один

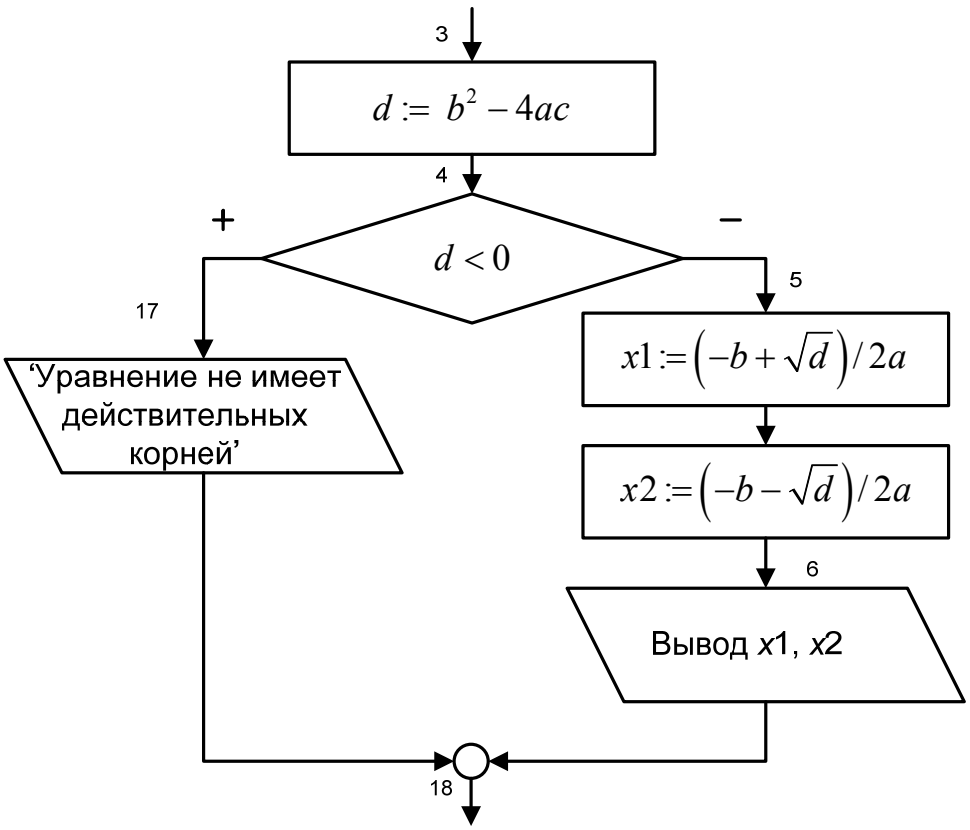


Рисунок 2.8 Шаг 3-18: алгоритм решения невырожденного квадратного уравнения

или другой вид уравнения (Рисунок 2.6 и Рисунок 2.8)). После этого каждый из шагов мы 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).

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*.

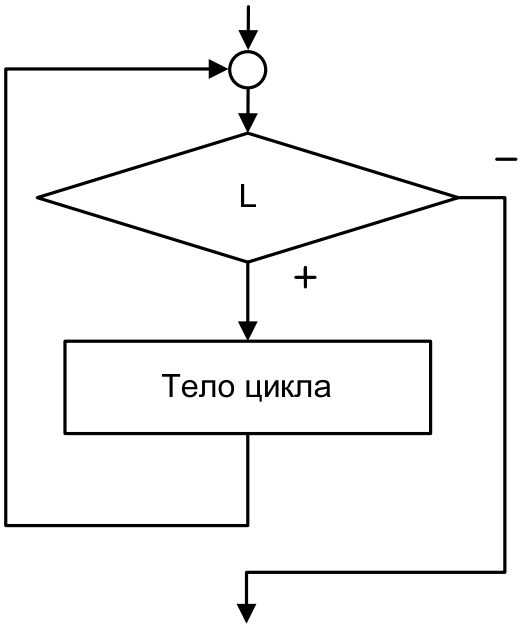


Рисунок 2.9 Цикл с предусловием

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.

Цикл с предусловием или цикл «***Пока***» (цикл выполняется, пока логическое выражение *L* дает результат ***TRUE***). Блок-схема изображена на Рисунок 2.9. В *Pascal* этот цикл называется «***while*** … ***do***». Как следует из названия цикла, его тело будет выполняться пока верно логическое выражение (*L*=***TRUE***)[1]. Как только результат 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.

[1] Очень часто в предикатных узлах для проверки логической переменной сравнивают ее 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.

$F(N) = N!$

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), т.к. число I умноженное на 1 останется равным I, т.е. после первого шага переменная F станет $F = I = 1$. Дальнейшее умножение на 2, на 3 и т.д. до N даст факториал числа $F(N) = N!$.

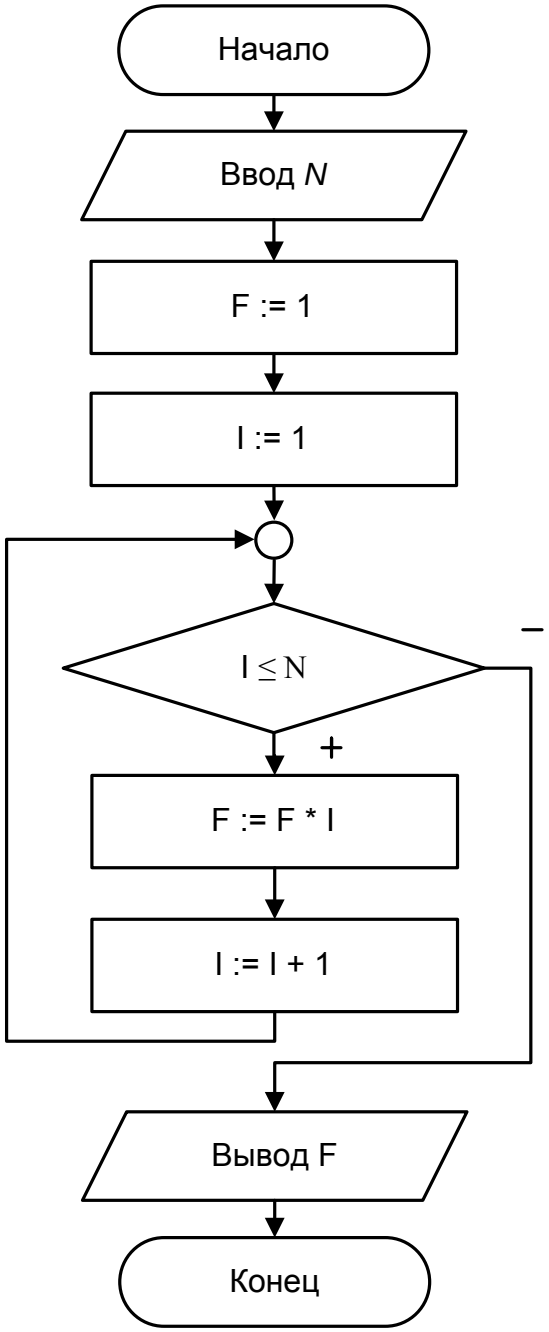


Рисунок 2.10 Алгоритм расчета факториала (цикл с предусловием)

Рассмотрим, как работает данный алгоритм. 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. При входе в цикл анализируется логическое выражение «*I*≤*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

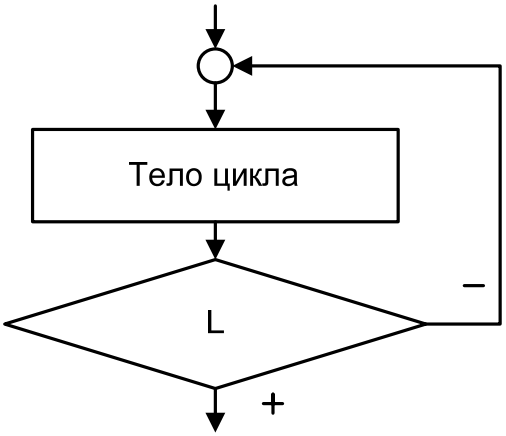


Рисунок 2.11 Цикл с постусловием

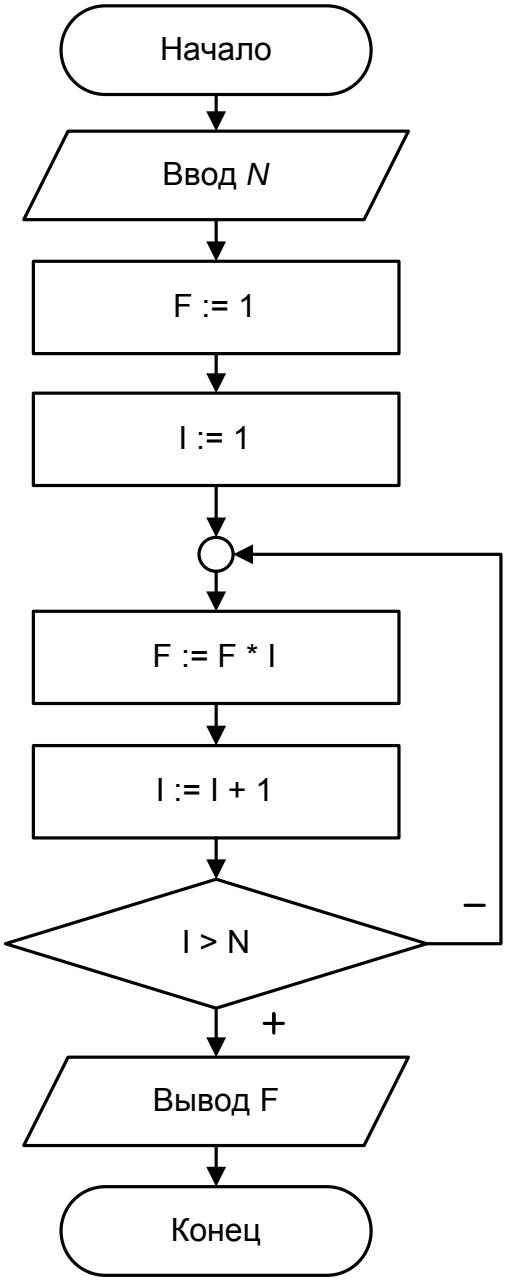


Рисунок 2.12 Алгоритм расчета факториала (цикл с постусловием)

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 Рисунок 2.11. В отличие от цикла с предусловием, его тело повторяется пока L=***FALSE***, как только L становится равным ***TRUE***, выполнение тела завершается.

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. 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 Рисунок 2.12 представлена блок-схема алгоритма вычисления факториала числа N с помощью цикла с постусловием.

Третий цикл – ***цикл с параметром***, или, его еще называют «***цикл For***». Этот цикл работает следующим образом. Параметру I присваивается его начальное значение K, далее выполняется тело цикла, в

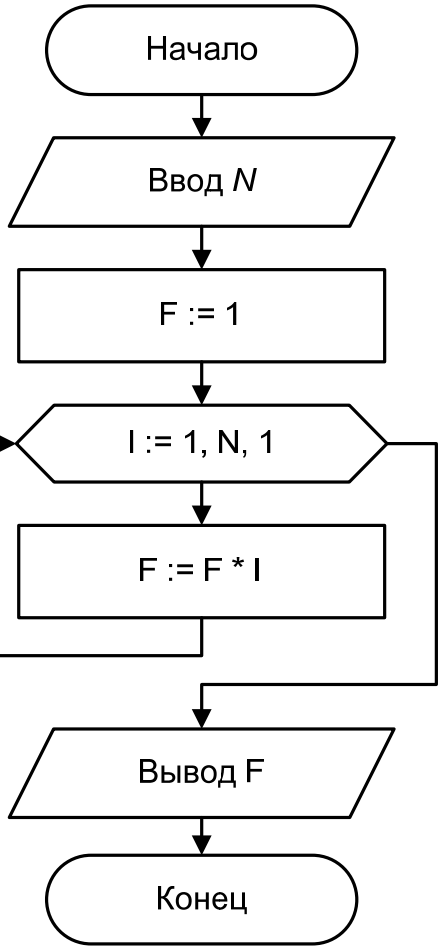


Рисунок 2.14 Вычисление факториала (цикл с параметром)

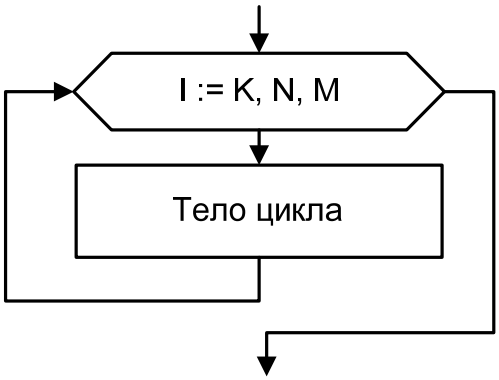


Рисунок 2.13 Цикл с параметром

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 (Рисунок 2.13).

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 (Рисунок 2.14).

Замечания по оформлению блок-схем

Как уже отмечалось ранее, блок-схемы представляют собой одну из форм записи последовательности действий. Поскольку мы договорились 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 (Рисунок 2.15).

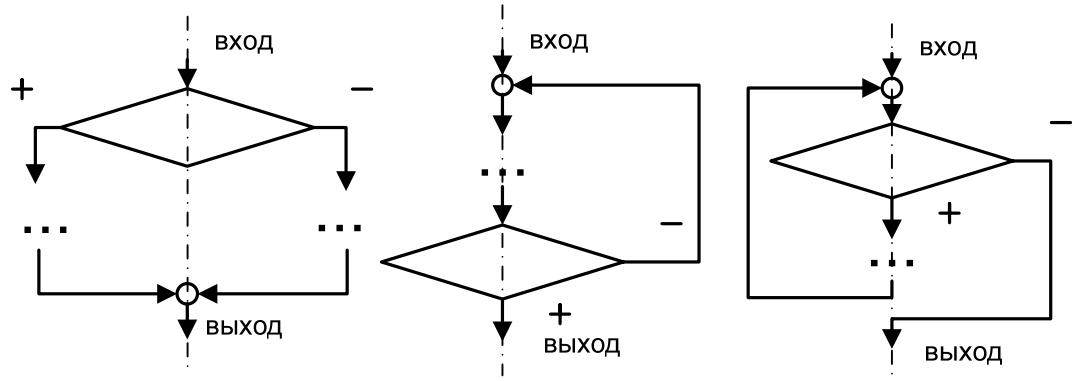


Рисунок 2.15 Симметрия в обозначении базовых алгоритмических конструкций

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 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.

Структурное программирование отличается тем, что внутри каждой элементарной конструкции существует блок 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 Рисунок 2.7.

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). 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 уж, тем более, не с низу наверх.

Примеры использования базовых алгоритмических конструкций показаны на (Рисунок 2.16), где четко можно различить все их типы, а также входы и выходы.

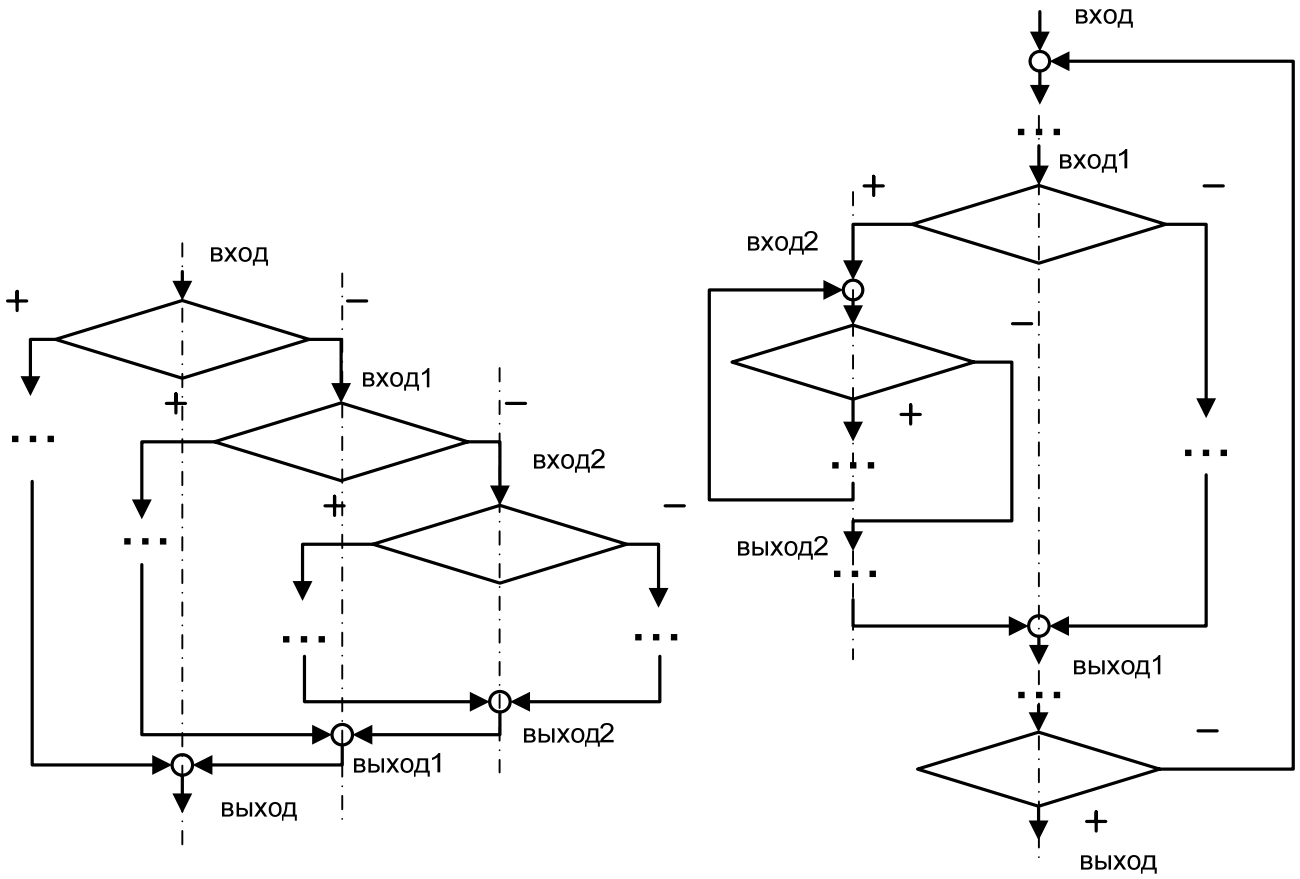


Рисунок 2.16 Вложенные алгоритмические конструкции

3. ОСНОВНЫЕ СВЕДЕНИЯ ОБ 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 такие 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 буквах будут восприняты компилятором как одно и то же.

Приведем примеры идентификаторов.

ПРИМЕР

сначала правильные:

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

теперь 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 8 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, мы захотели экран очистить. Для этого нам потребуется процедура ***clrscr***, содержащаяся в библиотеке *CRT*. Если использовать ее без ***Uses CRT***, то откомпилировать программу будет невозможно, т.к. 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 данного 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, которые будут использованы 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***.

Вот 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, то 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 «=»[1], 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 такие 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[2].

[1] 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».

[2] 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.

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;
⋮
```

В следующем разделе ***описываются переменные***. Это раздел ***var*** (*англ. Variable – переменная*). Переменной можно назвать поименованную область памяти, содержащую информацию заданного типа (см. раздел 2). *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 храниться в двоичном виде. Чтобы отделить, например, текст от чисел необходимо к каждой переменной привязывать способ перекодирования, то есть 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 подробно останавливаться на этой теме, т.к. она будет подробно раскрыта позже (см. раздел 7).

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 имеют 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 – переход на новую строчку. К сожалению, старые *IDE* языка *Pascal* этот тип комментариев не поддерживают. Это относится, прежде всего, к устаревшей *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 (см. подпункт глава 1). 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. Т.к. диапазон начинается с 0, то последнее число не 256, а 255.

Все целочисленные типы данных *Turbo Pascal* описаны в следующей таблице (Таблица 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[1]	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)

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	$\pm1.5\cdot10^{-45}..\pm1.5\cdot10^{38}$
Real	48	11-12	$\pm2.9\cdot10^{-39}..\pm1.7\cdot10^{38}$
Double	64	15-16	$\pm5.0\cdot10^{-324}..\pm1.7\cdot10^{308}$
Extended	80	19-20	$\pm1.9\cdot10^{-4951}..\pm1.1\cdot10^{4932}$
Comp	64	19-20	$-9.2\cdot10^{18}..9.2\cdot10^{18}$

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[2] 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.

[1] 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-ти 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.

[2] 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.

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

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[1], в которую возводят число 10 при домножении на *мантиссу* числа.

Для большей наглядности поставим в соответствие каждому числу из примера обычную его математическую запись (Таблица 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. 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[2] 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';
```

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

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

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, который рассмотрим здесь – ***boolean***. Это логический тип, описание которого достаточно подробно приведено в разделе 2.4, 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;
```

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

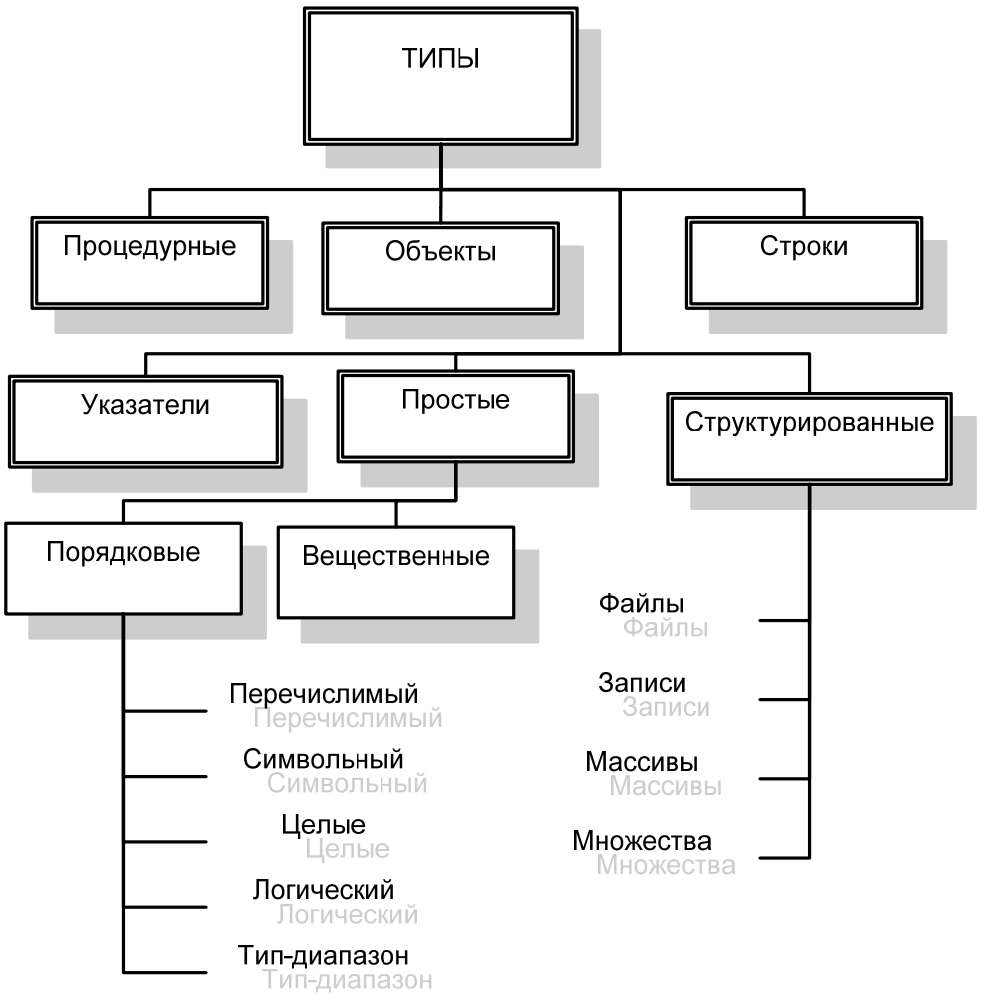


Рисунок 3.1 Генеалогия типов языка Pascal

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 (Рисунок 3.1).

3.4 Математические операции и функции

Напомним, что оператор присваивания в *Pascal* записывается «:=». В математике мы привыкли присваивать переменной значение с помощью знака «=», причем неважно с какой стороны от знака записана переменная, например, «*X*=5» или «5=*X*», для нас 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, т.к. 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 *X* в результате присвоения будет значение ***TRUE***, иначе – ***FALSE***.

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 (Таблица 3.4).

Таблица 3.4 Стандартные математические функции Pascal

Математическая запись функции	Запись функции в ТР
|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 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 $\mathrm{tg}(x)=\dfrac{\sin(x)}{\cos(x)}$. Вычисление, например, арксинуса таково: $\arcsin(x)=arctg\left(\dfrac{x}{\sqrt{1-x^2}}\right)$

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 следующей таблице (Таблица 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, которая «встряхивает» генератор случайных чисел
odd(x)	***longint***	***boolean***	Дает ***TRUE***, если *x* – нечетное, ***FALSE***, если *x* – четное
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
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
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)***
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[1] режиме, служат две процедуры ***write*** и ***writeLn***. Они имеют формат

write(*a, b, c, …****);***

writeLn(*a, b, c, …****);***

где *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 из них.

Чтобы вывести некоторый текст, можно воспользоваться этой процедурой так:

```
write('текст');
```

Содержимое экрана после выполнения этой процедуры:

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

текст

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

```
var
  x:byte;
⋮
x:=5;
write(x);
⋮
```

Содержимое экрана:

5

Эти два способа вывода можно комбинировать так:

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

Содержимое экрана:

x=5

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:

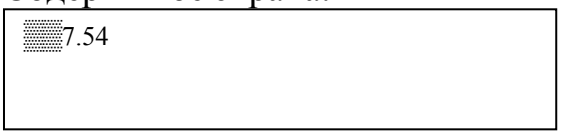
5.00100000000000E+0000

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 6 знакомест, но т.к. оно заняло лишь 4, то оставшиеся два знакоместа слева были заполнены пробелами.

Процедура ***writeLn*** выполняет те же функции, что и ***write***, но в отличие от нее 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 тоже осуществляется непосредственно.

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. Далее после следующего ввода набранное значение помещается в следующую переменную и так далее. 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 сама по себе, поскольку 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 (***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 большие 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 ***str2*** имеют одинаковую размерность (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 Таблица 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*) и записывает результат в строку *s*
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 Киррилицей)

3.7 Программирование развилок

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

if *условие* ***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 *условие* ***then***
*управляющий оператор положительной ветви****;***

ПРИМЕР

Ввести 3 заведомо неравных числа (A, B, C). Наибольшее из них возвести в квадрат.

```
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.
```

Рассмотренный пример показывает, как именно можно пользоваться разветвляющимися процессами. Блок-схема программы представлена на Рисунок 3.2.

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

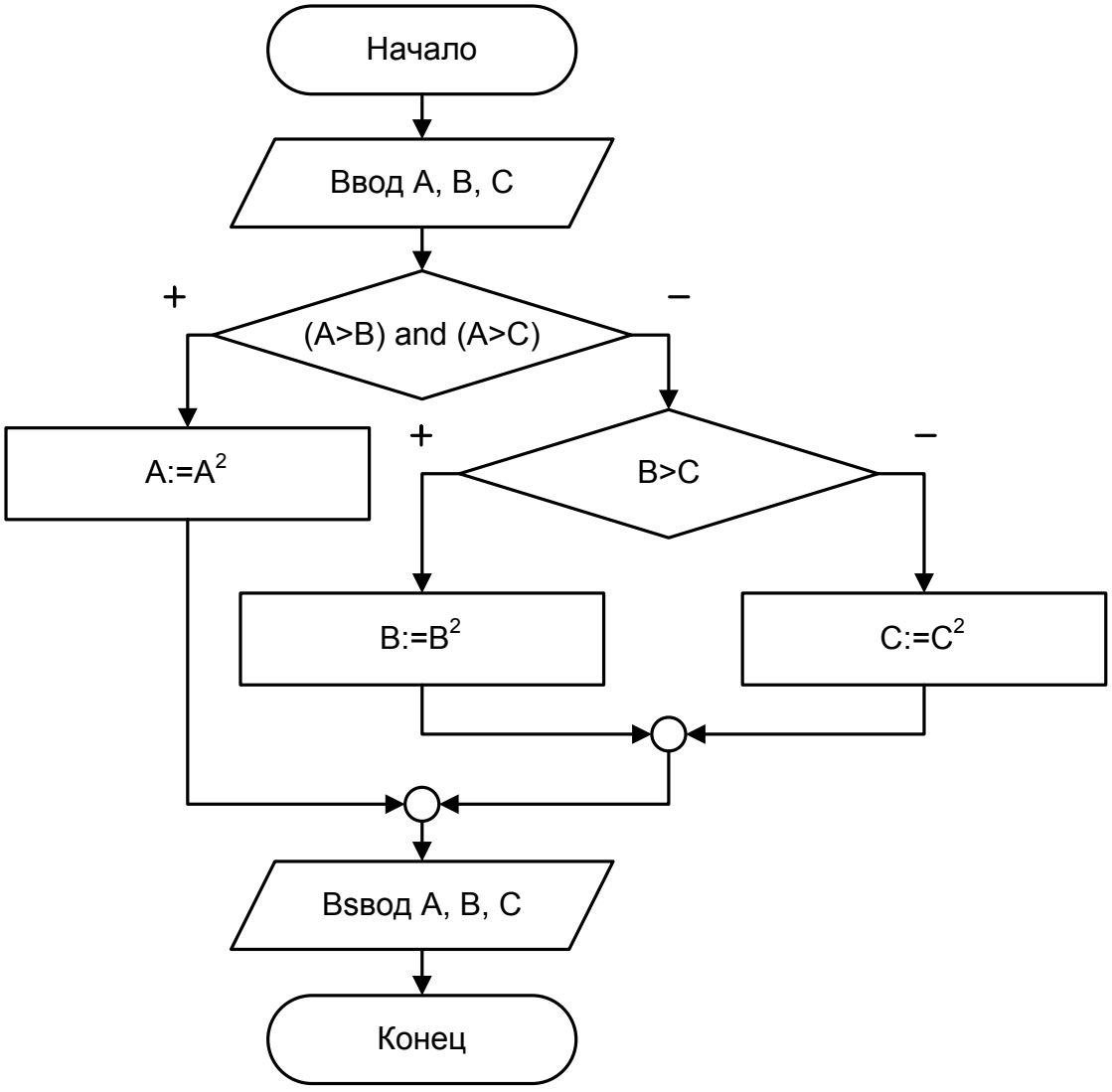


Рисунок 3.2 Возведение в квадрат наибольшего из трех неравных чисел

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». Алгоритмические управляющие структуры развилкой с двумя вариантами дальнейшего движения были рассмотрены ранее. Однако существует так называемая, развилка множественного выбора. Т.е. когда мы можем двигаться далее по алгоритму не в двух возможных направлениях, а в гораздо большем их количестве.

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

case *значение* ***of***
*первый вариант**:** управляющий оператор для 1-го варианта**;***
*второй вариант**:** управляющий оператор для 2-го варианта**;***
*третий вариант**:** 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 на Рисунок 3.3.

ПРИМЕР

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

```
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 ', H, ' соответствует ', S);
end.
```

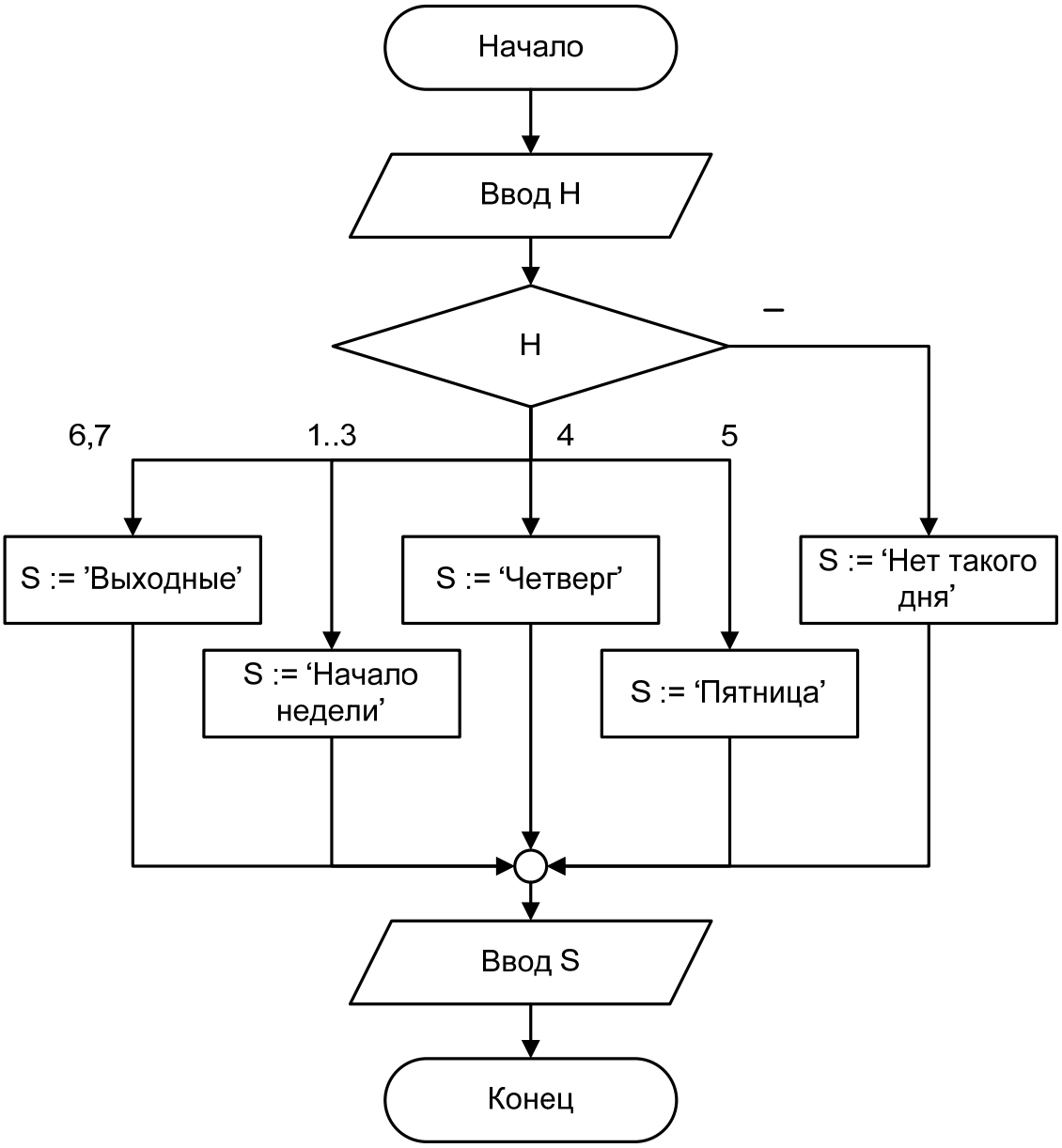


Рисунок 3.3 Пример работы Case

3.8 Программирование циклов

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

while *условие* ***do***
*тело цикла****;***

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

Цикл с постусловием записывается как

repeat
*тело цикла****;***
until *условие выхода****;***

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

Цикл с параметром и с шагом «+1» записывается следующим образом:

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**;***

Для понимания рассмотрим работу всех трех циклов на примере одной задачи.

ПРИМЕР

Протабулировать функцию $y=x^2$ на промежутке [-2, 1].

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

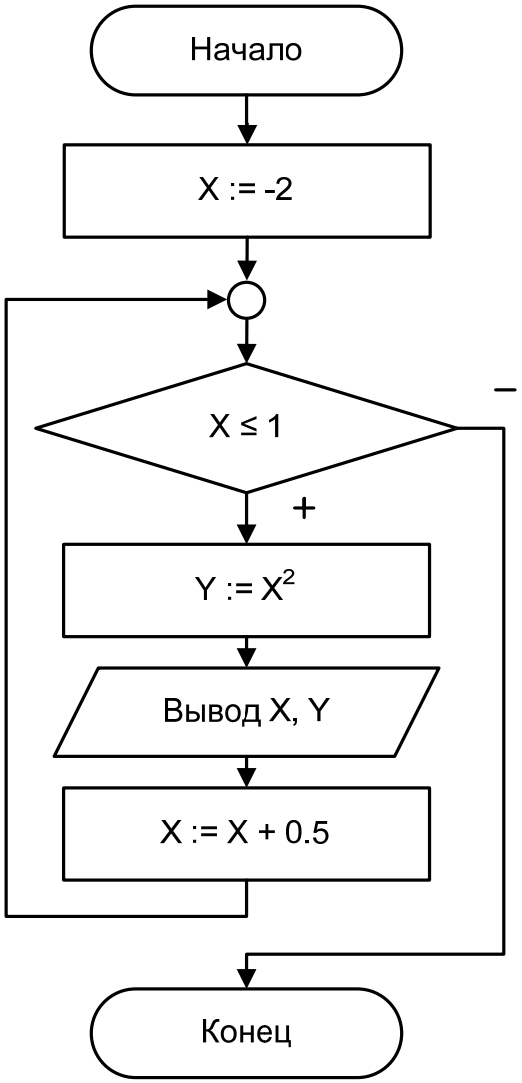


Рисунок 3.4 Табулирование функции циклом while

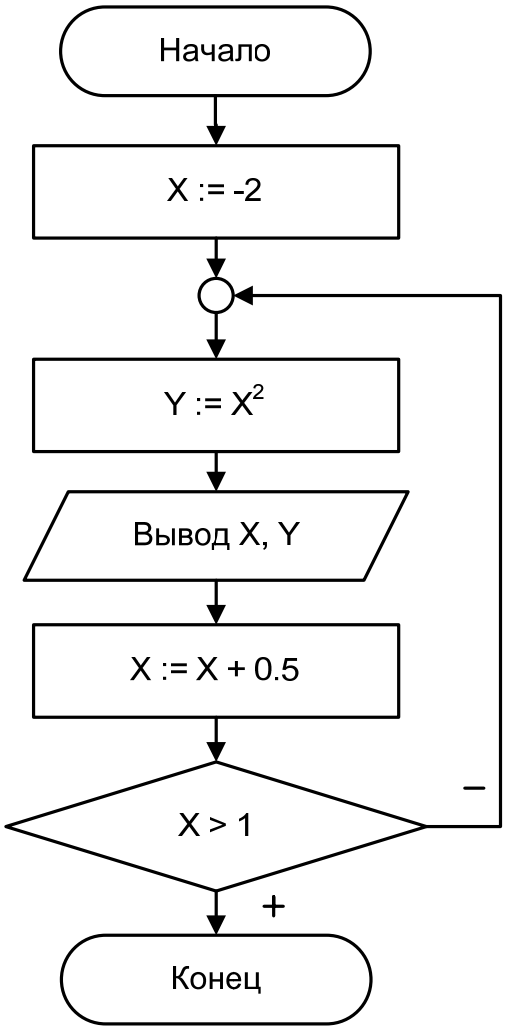


Рисунок 3.5 Табулирование функции циклом repeat ... until

```
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 Рисунок 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.
```

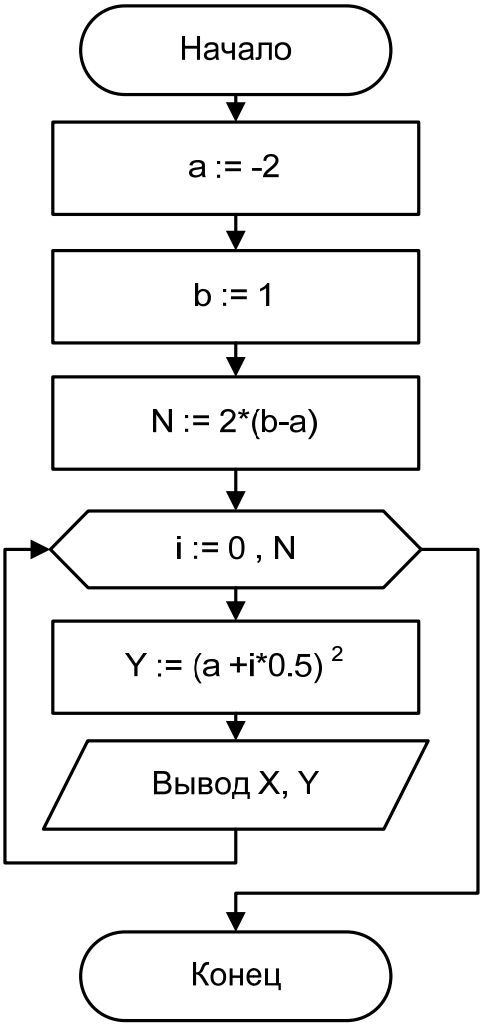


Рисунок 3.6 Табулирование функции циклом for

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

```
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.
```

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

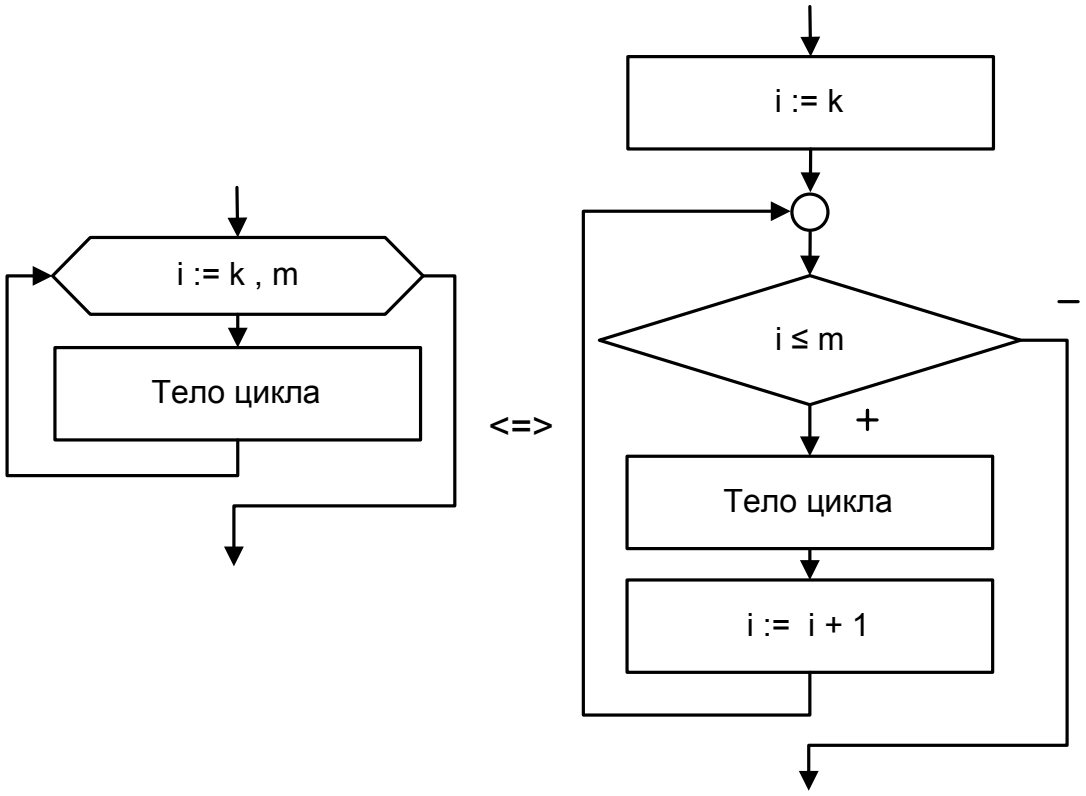


Рисунок 3.7 Эквивалентное преобразование цикла for в цикл while

```
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 Рисунок 3.7.

3.9 Составной оператор

Итак, выше мы описали 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.

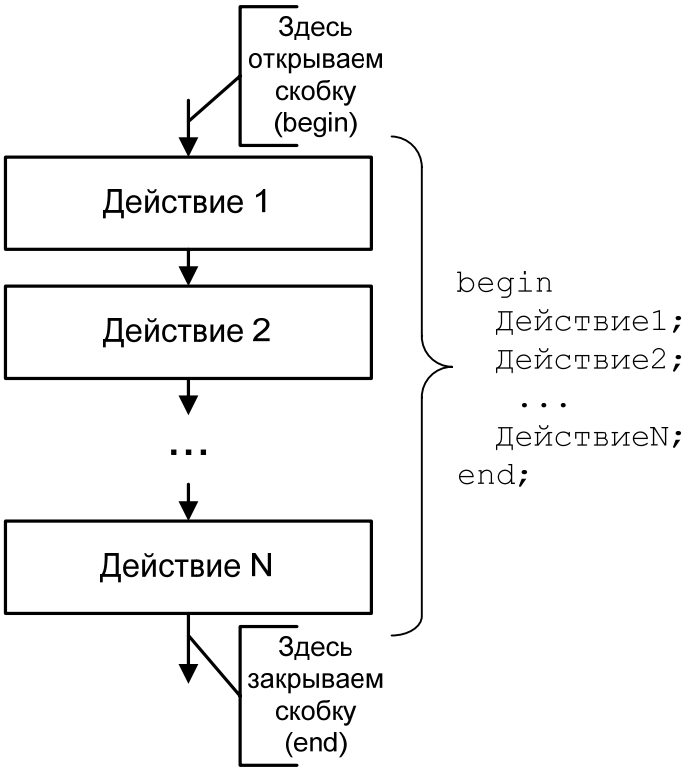


Рисунок 3.9 Сборка составного оператора

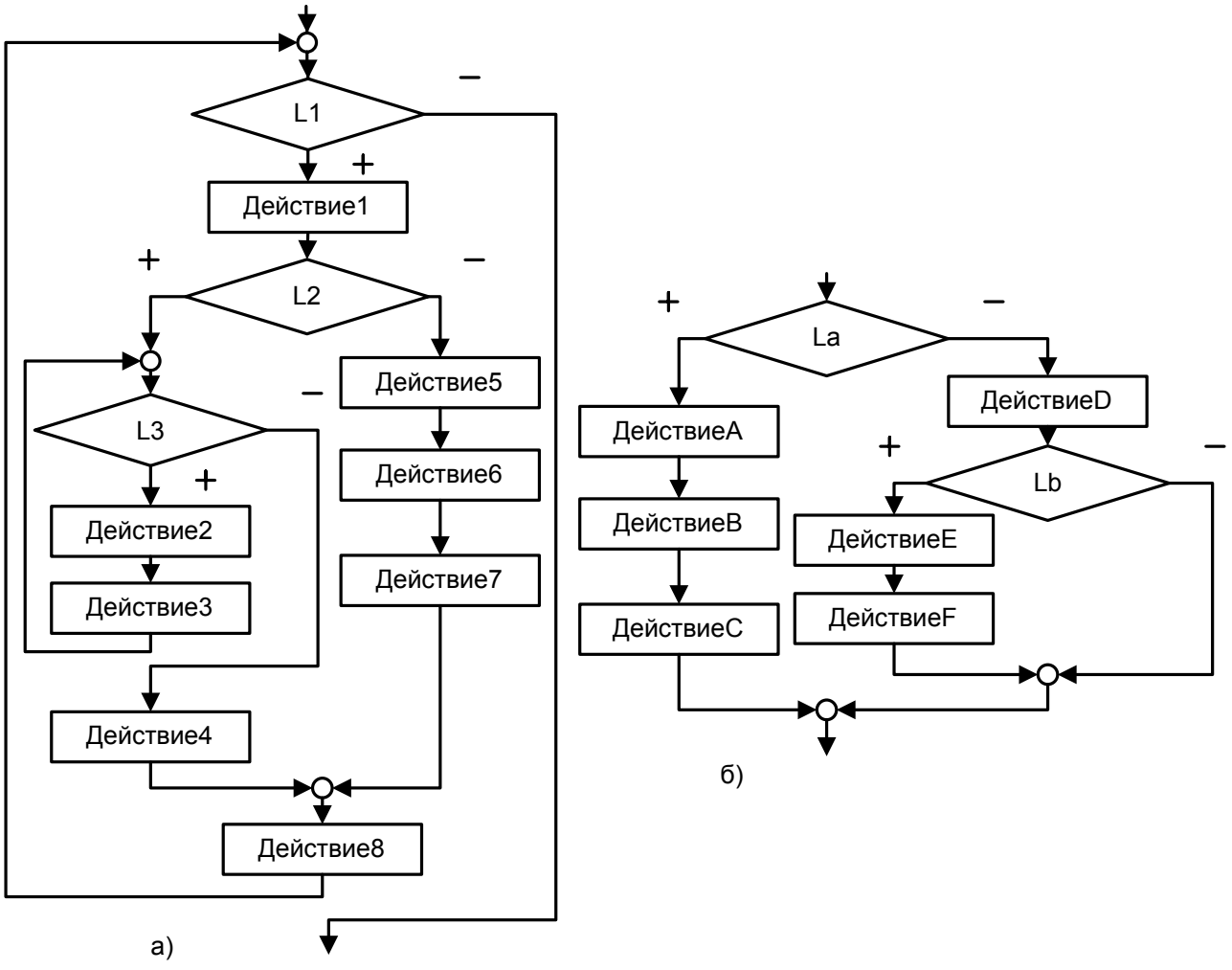


Рисунок 3.8 Примеры алгоритмов, где используется составной оператор

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 Рисунок 3.9. 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 (Рисунок 3.8). Для случая (Рисунок 3.8) 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 (Рисунок 3.8) б) *Pascal*-код таков:

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

4. РЕШЕНИЕ ТИПОВЫХ ЗАДАЧ НА РАЗВИЛКИ И ЦИКЛЫ

4.1 Задачи на развилки

При решении задач на развилки используется ветвление алгоритма. Как мы помним, направление ветвления зависит от истинности предиката стоящего в точке развилки. Следует иметь в виду, что множество вариантов ветвления конечно. Составление алгоритма должно полностью охватывать все существующие варианты решения задачи. При этом желательно направления движения по алгоритму, приводящие к одинаковым результатам не дублировать.

Большинство задач на развилки по своей сути являются задачами на правильное составление предиката. Предикат, как мы помним, является выражением, принимающим всего 2 значения ***TRUE*** или ***FALSE***. Если у нас несколько предикатов, то мы их комбинируем, используя основы алгебры логики.

Далее рассмотрим наиболее показательные в этом отношении примеры.

ПРИМЕР

Выяснить попадает ли точка с координатами (x, y) *в окружность радиуса* R *с центром в начале координат*.

Задача достаточно проста, если воспользоваться иллюстрацией (Рисунок 4.1 а)). Точка удалена от начала координат на расстояние, вычисляемое по теореме Пифагора: $r1 = \sqrt{x^2 + y^2}$. Если радиус окружности R , то в зависимости от того больше, меньше или рано $r1$ радиусу R , мы будем

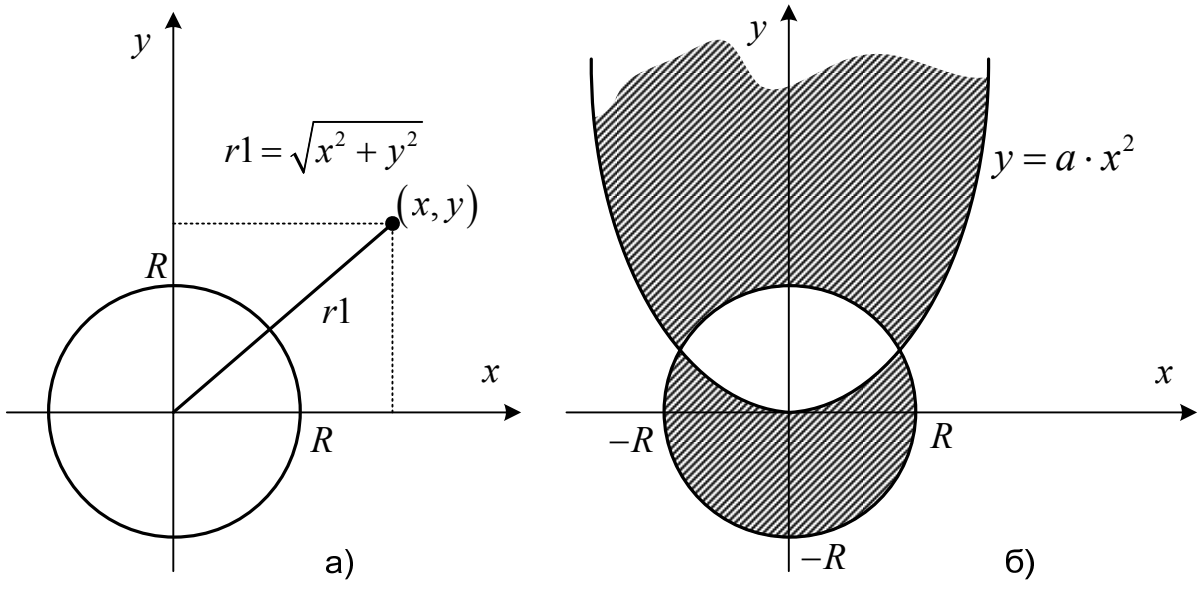


Рисунок 4.1 Попадание точки в окружность – а) и попадание точки в заштрихованную область – б)

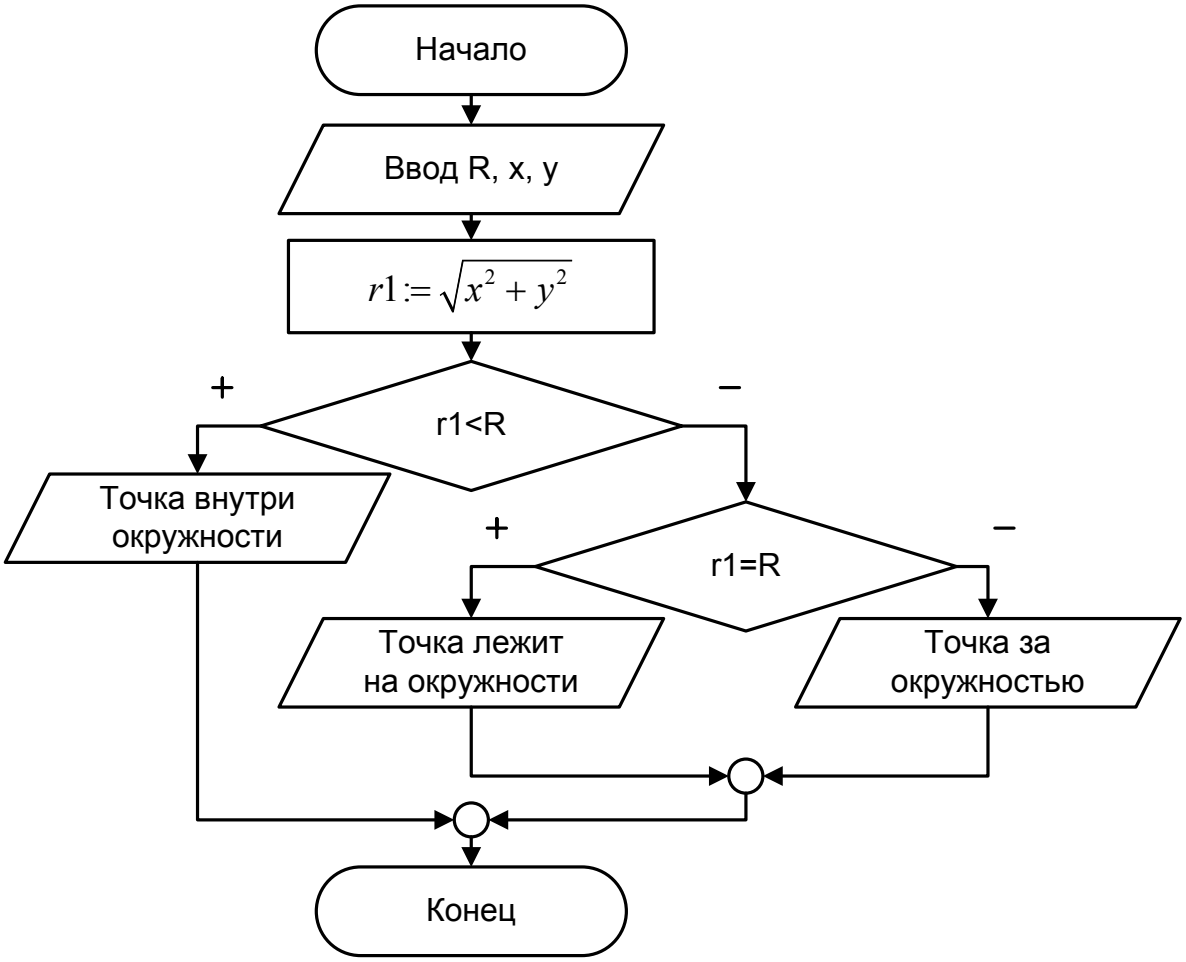


Рисунок 4.2 Проверка попадания точки в окружность

иметь три варианта расположения точки: «*внутри окружности*», «*на окружности*» или «*за окружностью*». Всего три варианта, задача решается в два предиката. Использование двух предикатов, как раз, дает три маршрута движения от точки начала алгоритма к его концу (Рисунок 4.2).

Программа на языке *Pascal* такова:

```
program Radius;
var x,y,R,r1:real;
begin
  writeLn('введите радиус R');
  readLn(R);
  writeLn('введите координаты x, y');
  readLn(x,y);
  r1:=sqrt(sqr(x)+sqr(y));
  if r1<R then
    writeLn('точка внутри окружности')
  else
    if r1=R then
      writeLn('точка лежит на окружности')
    else
      writeLn('точка за окружностью');
end.
```

Тестовый пример может быть, скажем, таким:

вход: $R = 5$, $x = 3$, $y = 4$

выход: «точка лежит на окружности»

Далее рассмотрим модификацию предыдущей задачи, которую сформулируем следующим образом:

ПРИМЕР

Выяснить попадает ли точка с координатами (x, y) *в заштрихованную область (Рисунок 4.1 б)).*

Для решения этой задачи нужно внимательно посмотреть на иллюстрацию и написать условия попадания точки в область круга радиуса R

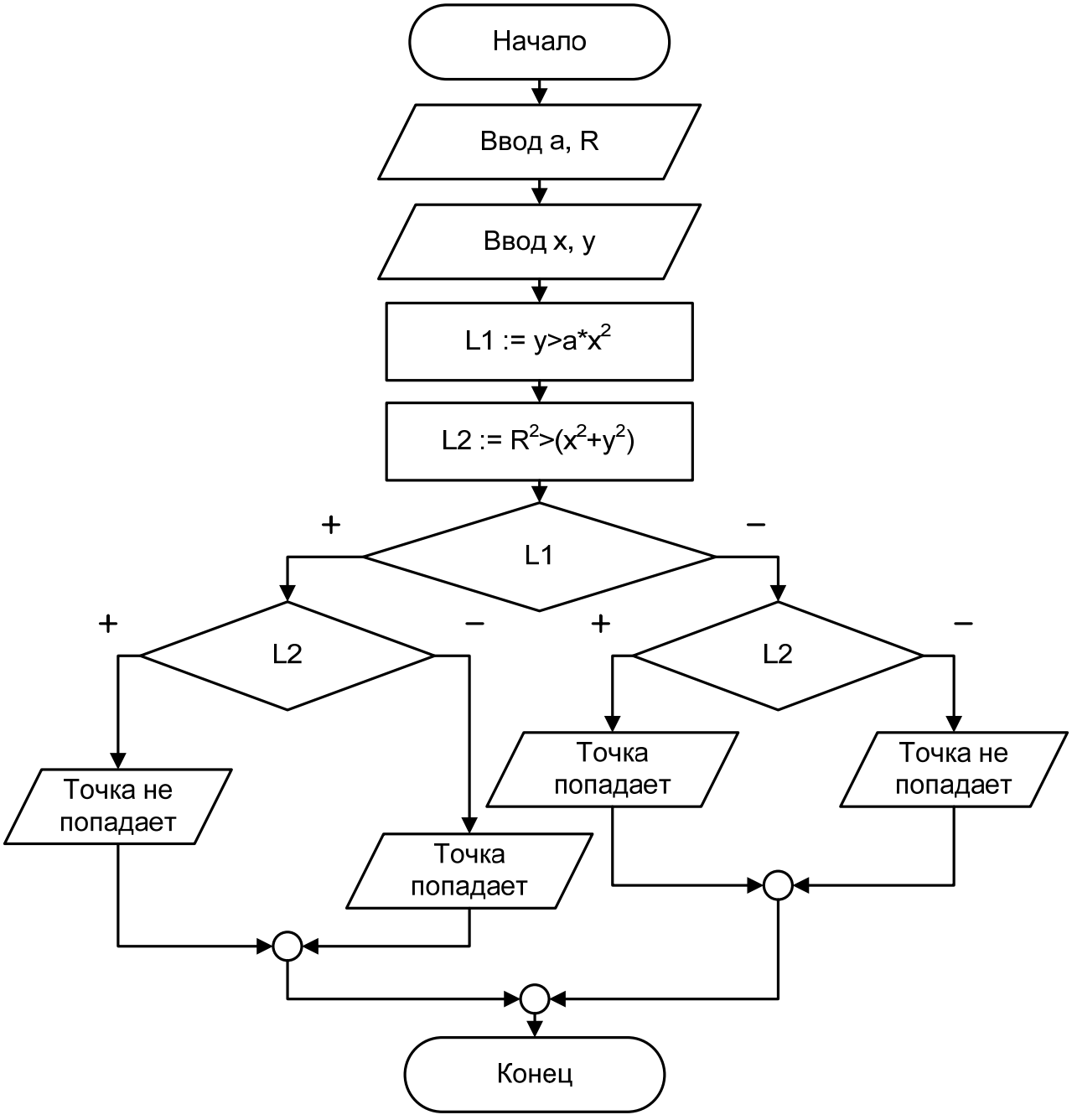


Рисунок 4.3 Нерациональный алгоритм проверки попадания точки в заштрихованную область

и в область расположенную над параболой. Условие попадания точки в круговую область: $R^2 > x^2 + y^2$. При решении задачи удобно пользоваться переменными типа ***boolean***, поскольку увеличивается наглядность алгоритма. Для круговой области мы воспользовались переменной *L2*, а для области над параболой мы взяли переменную *L1* и поместили в нее выражение $y > a \cdot x^2$. Итак, если точка попадает в область над параболой, то выполнено условие *L1*, однако в этом случае точка не должна быть внутри круга, т.е. условие *L2* должно быть равно ***FALSE***. Ну а если условие *L1* не выполняется, то для попадания точки в заштрихованную область необходимо чтобы она находилась внутри круга, т.е. чтобы *L2* было равно ***TRUE***. Описанные условия представлены в виде алгоритма на (Рисунок 4.3).

В качестве тестового примера можем записать такой набор входных и выходных данных:

входные: $a = 1,\ R = 1,\ x = 0,\ y = 0{,}5$

выходные: «**точка не попадает в область**»

Далее идет листинг программы.

```
Program PRxor1;
var x,y,R,a:real;
    L1,L2:boolean;
begin
  writeLn('введите коэффициент a');
  readLn(a);
  writeLn('введите радиус R');
  readLn(R);
  writeLn('введите координаты x, y');
  readLn(x,y);
  L1:=y>a*sqr(x);
  L2:=sqr(R)>sqrt(sqr(x)+sqr(y));
  if L1 then
    if L2 then
      writeLn('точка не попадает')
    else
      writeLn('точка попадает')
  else
    if L2 then
      writeLn('точка попадает')
    else
      writeLn('точка не попадает');
end.
```

Однако, задачу можно решить гораздо быстрее, если воспользоваться алгеброй логики при составлении предикатов. Ранее рассматривались разные операции и давались иллюстрации из теории множеств для каждой из них. Здесь, очевидно, наиболее подходит операция ***исключающего ИЛИ***, т.е. ***XOR***. В этом случае задача решается в один предикат (Рисунок 4.4). Программа на *Pascal* такова:

```
program PRxor1;
var x,y,R,a:real;
L1,L2:boolean;
begin
  writeLn('введите коэффициент a и радиус R ');
  readLn(a,R);
  writeLn('введите координаты x, y');
  readLn(x,y);
  L1:=y>a*sqr(x);
  L2:=sqr(R)>sqrt(sqr(x)+sqr(y));
  if L1 xor L2 then
    writeLn('точка попадает')
  else
    writeLn('точка не попадает');
end.
```

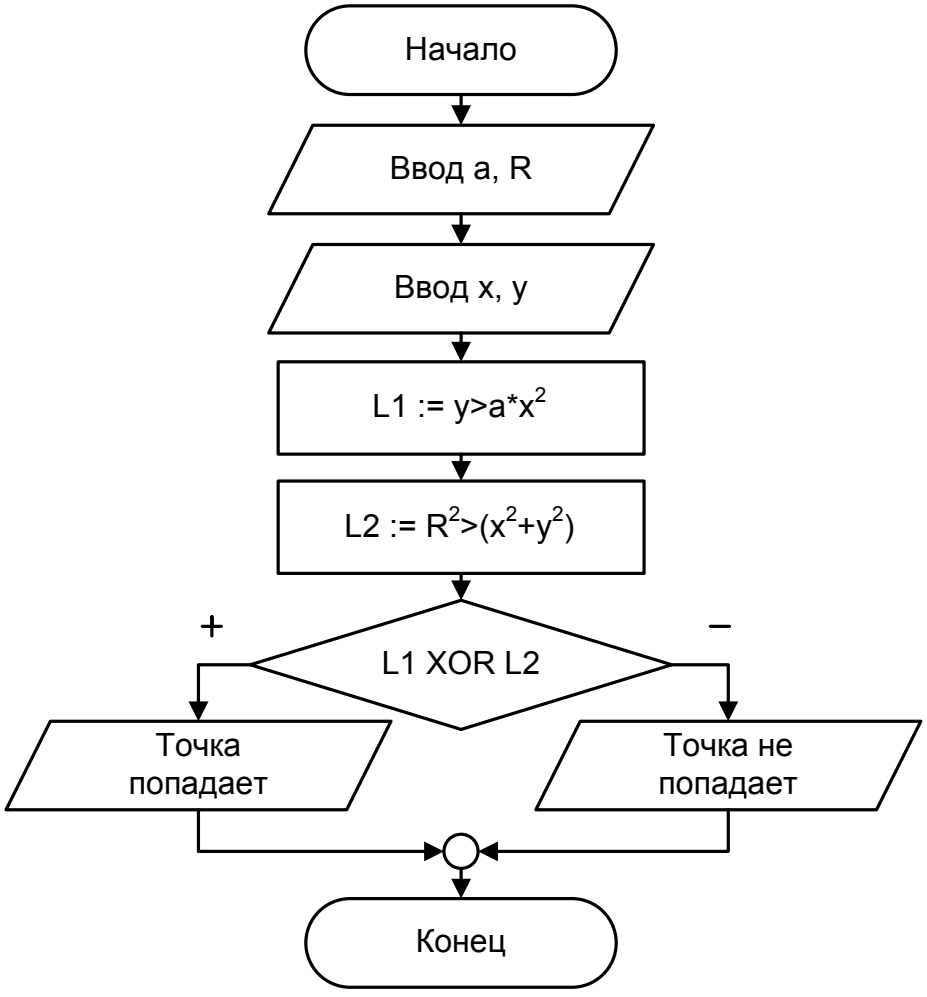


Рисунок 4.4 Проверка на попадание точки в заштрихованную область с использованием операции XOR

Теперь рассмотрим такую задачу:

ПРИМЕР

Даны три неравных числа a,b,c. *Выстроить их в порядке возрастания.*

Если взять три числа и попытаться выстроить в порядке возрастания, то методом перебора можно прийти к заключению, что всего существует 6 вариантов порядка трех чисел. Вот эти варианты: «a,b,c», «a,c,b», «b,a,c», «b,c,a», «c,a,b», «c,b,a». Вообще, если вспомнить комбинаторику, то для N объектов существует $N!$ перестановок. В нашем случае $3!=6$, что было показано выше.

Самый простой способ решения состоит в составлении сложных предикатов на проверку отношения сразу между всеми числами. В этом случае алгоритм состоит в последовательном переборе всех вариантов (Рисунок 4.5). Программа для этого случая такая:

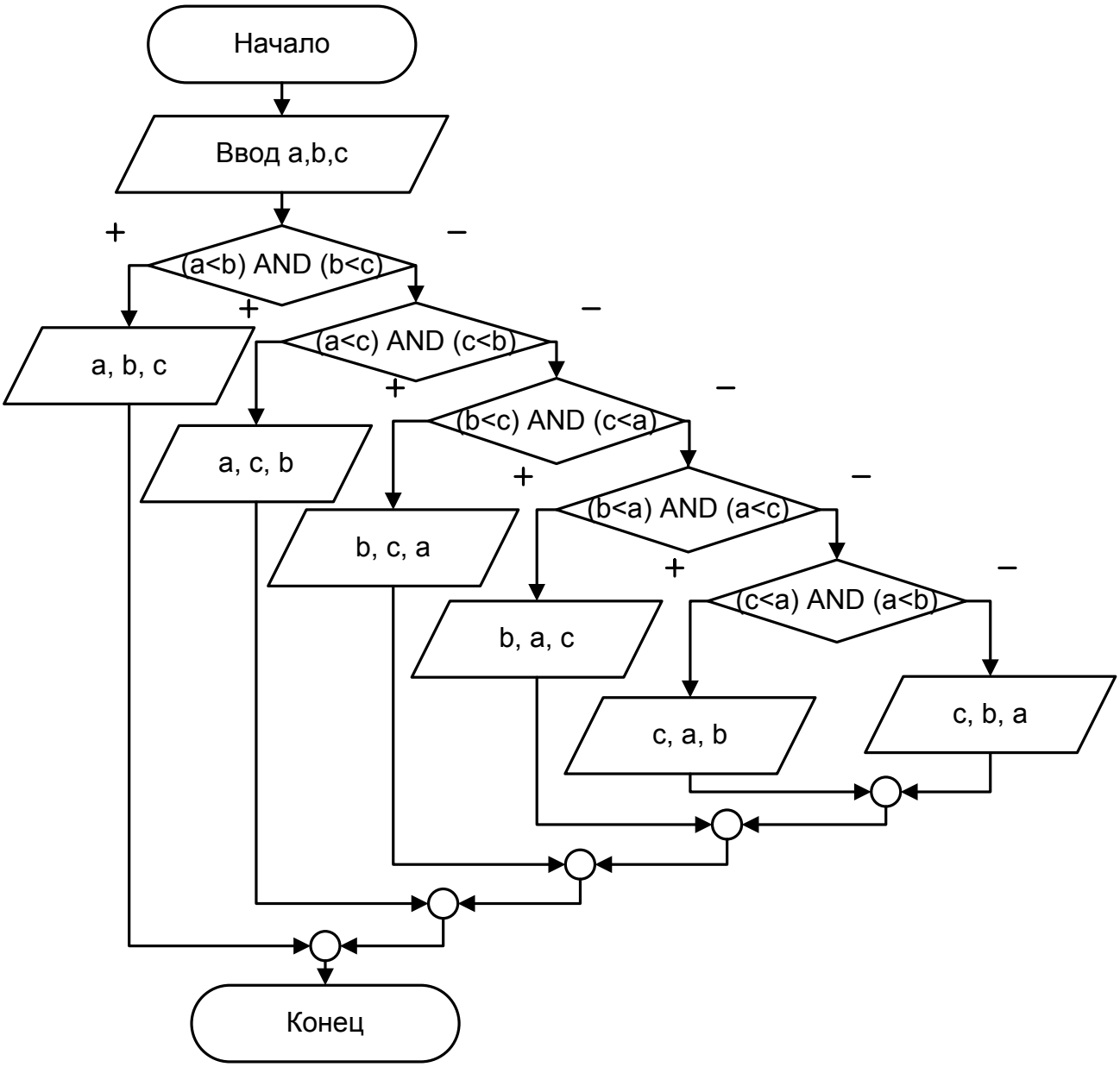


Рисунок 4.5 Выстраивание трех чисел по возрастанию наглядным способом

```
program UporABC1;
var a,b,c:integer;
begin
  writeLn('введите три числа a, b, c');
  readLn(a,b,c);
  if (a<b) and (b<c) then
    writeLn('a=',a,'  b=',b,'  c=',c)
  else if (a<c) and (c<b) then
    writeLn('a=',a,'  c=',c,'  b=',b)
  else if (b<c) and (c<a) then
    writeLn('b=',b,'  c=',c,'  a=',a)
  else if (b<a) and (a<c) then
    writeLn('b=',b,'  a=',a,'  c=',c)
  else if (c<a) and (a<b) then
    writeLn('c=',c,'  a=',a,'  b=',b)
  else
    writeLn('c=',c,'  b=',b,'  a=',a);
end.
```

Приведенный выше вариант обладает как достоинствами, так и недостатками. Среди достоинств следует отметить высокую наглядность решения, очень трудно что-то перепутать. Все условия для определенной последовательности проверяются сразу. Все последовательности идут друг за другом. Однако, есть и недостаток. Этот недостаток – время исполнения программы. Так, например, если мы будем иметь истинным только последнее сочетание (c,b,a), то для того, чтобы добраться до него, придется проверить на истинность все вышестоящие условия. Глубина вложенности самого последнего варианта равна пяти. Если мы захотим, чтобы вложенность самого дальнего варианта была меньше, то придется изменить алгоритм. Для этого придется проверять все условия раздельно. Алгоритм при этом усложнится, однако, среднее время его исполнения уменьшится. На такой простой задаче, какую мы сейчас рассматриваем это время засечь практически невозможно, однако важно понять сам принцип оптимизации алгоритмов. Ведь, во-первых, условий может быть намного больше, во-вторых, эти условия могут выполняться многократно, в третьих, в качестве условий могут выступать реальные физические процессы с большим характерным временем их протекания.

Более быстрый (но менее застрахованный от случайных ошибок) алгоритм может быть, например, таким как изображено на (Рисунок 4.6). Здесь самый глубокий вариант спрятан максимум за три предиката.

Программная реализация будет такая:

```
program UporABC2;
var a,b,c:integer;
```

```
begin
  writeLn('введите три числа a, b, c');
  readLn(a,b,c);
  if a<b then
    if b<c then
      writeLn('a=',a,'  b=',b,'  c=',c)
    else
      if a>c then
        writeLn('c=',c,'  a=',a,'  b=',b)
      else
        writeLn('a=',a,'  c=',c,'  b=',b)
  else
    if a<c then
      writeLn('b=',b,'  a=',a,'  c=',c)
    else
      if b>c then
        writeLn('c=',c,'  b=',b,'  a=',a)
      else
        writeLn('b=',b,'  c=',c,'  a=',a);
end.
```

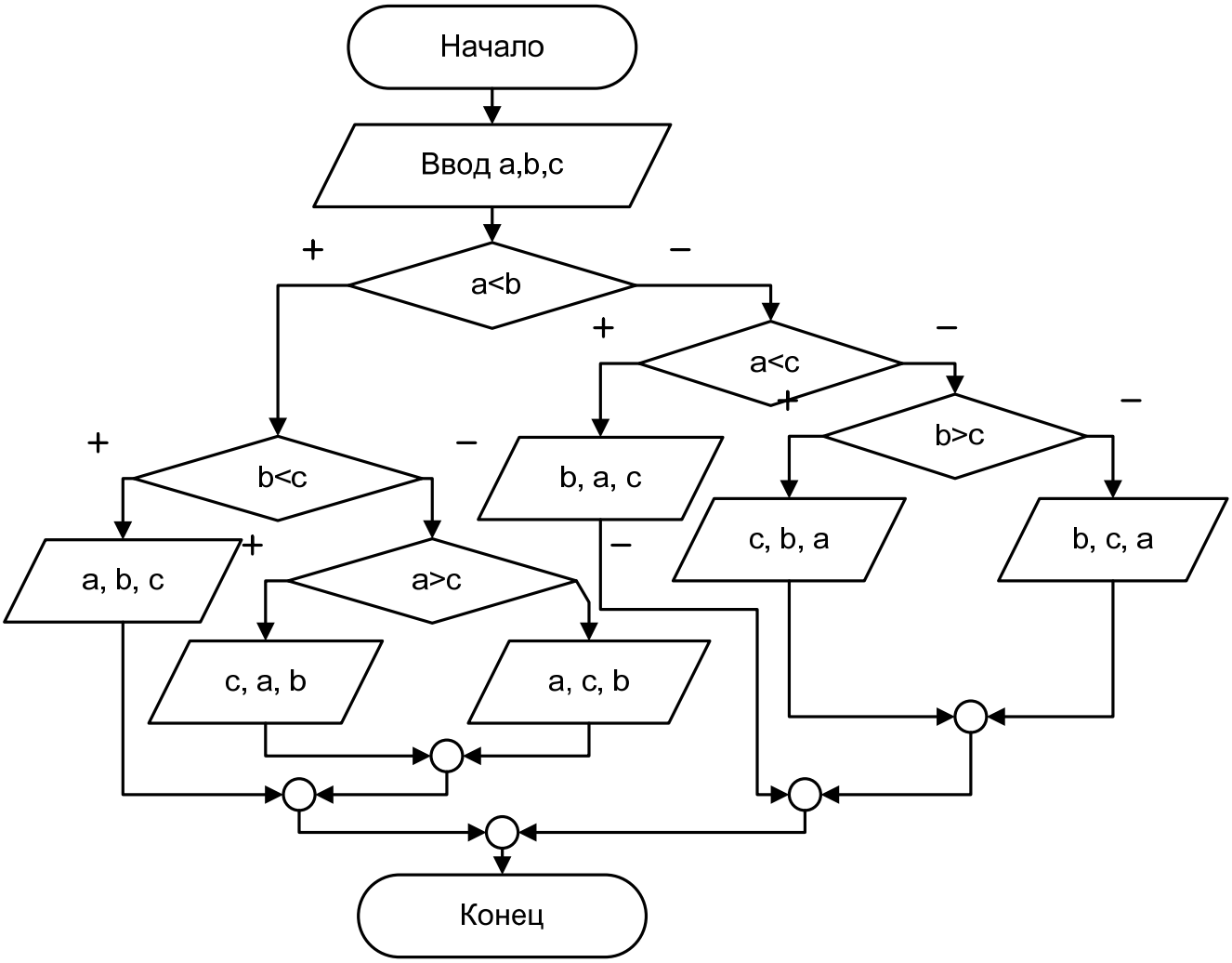


Рисунок 4.6 Упорядочивание трех чисел (более быстрый вариант)

4.2 Задачи на использование циклов

Задачи на использование циклов являются очень важными, поскольку практически в любой современной программе они присутствуют. Можно даже сказать, что именно использование циклов делает использование компьютера эффективным.

В предыдущей главе уже рассматривалась задача про табуляцию функции, для ее решения использовался один цикл. Можно условие той задачи немного модифицировать.

ПРИМЕР

Найти максимум среди значений функции $y = -2x^3 + 5x^2 + 3x - 1$ *при изменении* x *от* -1 *до* 4 *с шагом* $0{,}1$.

Т.е, фактически нам нужно протабулировать функцию, и из рассчитанных значений выбрать то, которое наибольшее. Плюс к этому, нужно запомнить значение x, при котором достигается этот максимум. Вообще, функция, о которой идет речь имеет график изображенный на (Рисунок 4.7). Естественно, что при расчете компьютер нам не нарисует график, однако это мы сможем сделать сами, если соединим точки, которые получаются при табуляции. Видно, что искомый максимум в районе точки $x = 2$. Вот только то, что мы видим и сразу оцениваем нам пока мало что дает, поскольку необходимо написать программу, которая бы самостоятельно среди множества чисел находила наибольшее значение.

Составить такую программу, на самом деле, не очень сложно. Для этого нужно взять за максимум первую точку, а далее просматривать в цикле все оставшиеся точки и сравнивать их. Если последующая точка окажется больше предыдущей, то ее мы принимаем за максимум. Далее продолжаем эти

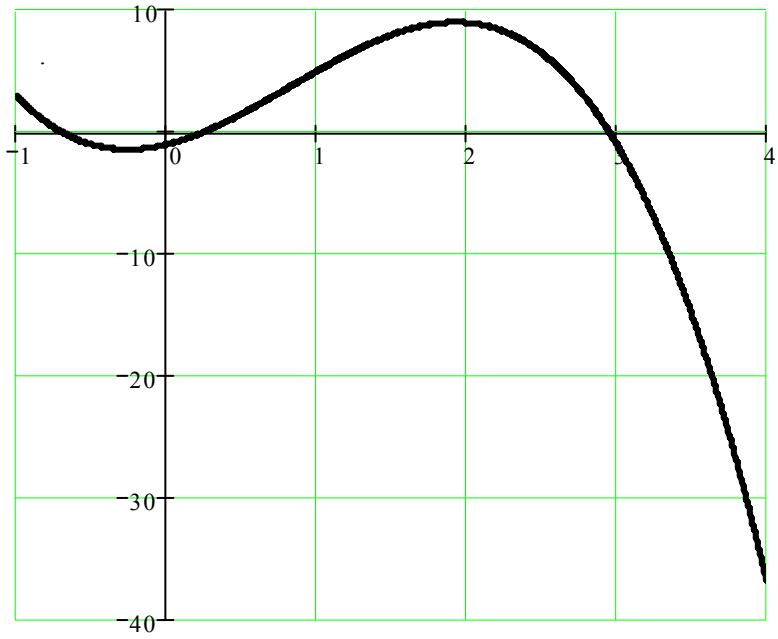


Рисунок 4.7 График кубического полинома

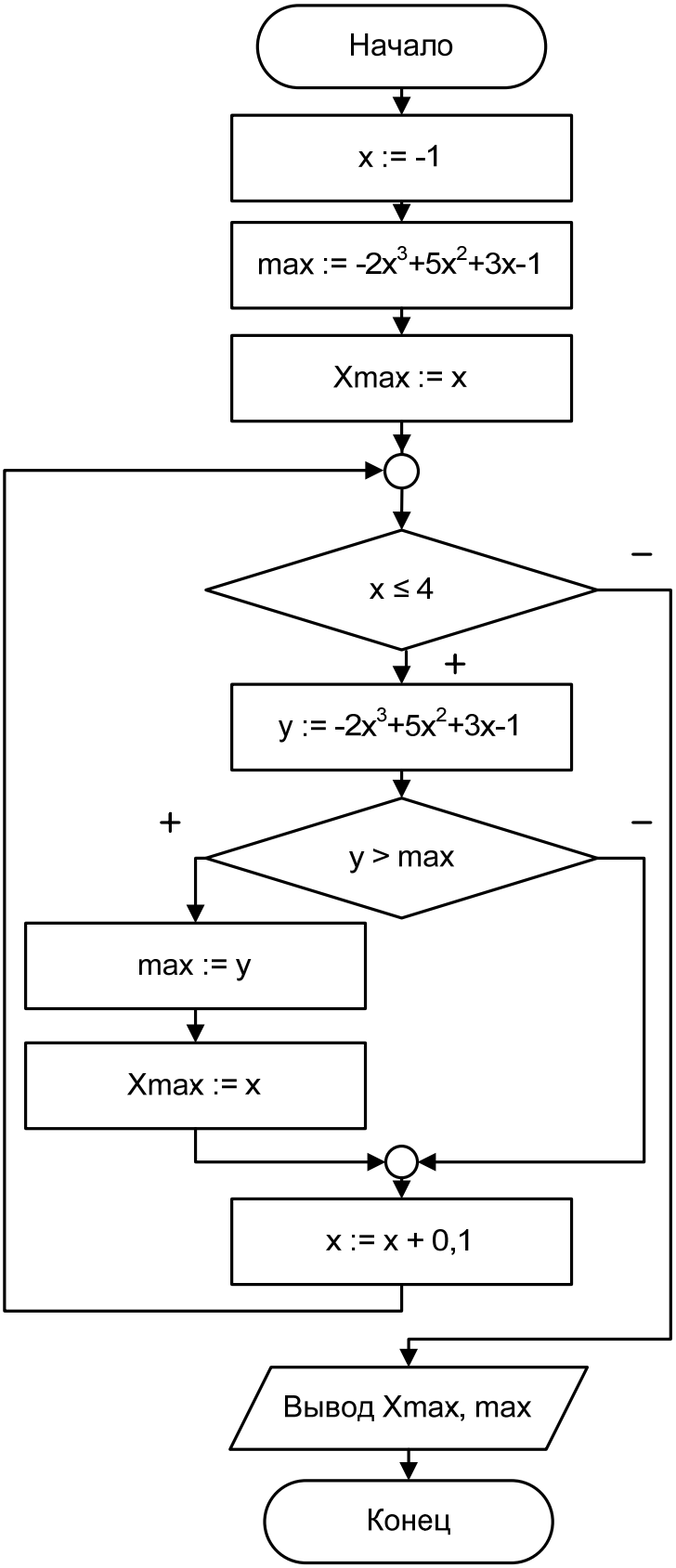


Рисунок 4.8 Поиск максимума среди значений функции

действия до тех пор, пока все точки не будут рассмотрены. Описанный алгоритм представлен на (Рисунок 4.8).

Программа такова:

```
program maxF;
var x,y,max,Xmax:real;
begin
  cls;
  x:=-1;
  max:=-2*x*x*x+5*x*x+3*x-1;
  Xmax:=x;
  while x<=4 do
    begin
      y:=-2*x*x*x+5*x*x+3*x-1;
      if y>max then
        begin
          max:=y;
          Xmax:=x;
        end;
      x:=x+0.1;
    end;
  writeLn('максимум достигается в точке ', x:6:2);
  writeLn('и он равен ',max:10:5);
end.
```

В результате получается, что программа нам выведет такой результат:

```
максимум достигается в точке   4.00
и он равен    9.03200
```

что, судя по графику, соответствует истине.

А теперь усложним задачу таким образом, чтобы функция, которую придется табулировать, имела пару аргументов. Итак,

ПРИМЕР

Протабулировать функцию $z = \cos\left(y \cdot \sin\left(x^3\right)\right)$, *причем вывести только те точки, которые больше чем* $0{,}6$. *Переменная* y *меняется на интервале* $[-2;1]$ *с шагом* $\Delta y = 1$, *переменная* x *меняется на интервале* $[13;15]$ *с шагом* $\Delta x = 0{,}8$.

Эта задача решается в два цикла, причем один будет вложен в другой. Пусть цикл по y будет внешним, а цикл по x – внутренним. Это означает, что мысленно зафиксировав значение y мы изменяем весь диапазон значений x. По своей сути изменение внутреннего цикла представляет простую табуляцию функции с одной переменной. Когда перебор всех значений переменных во внутреннем цикле окончен, мы меняем на значение шага переменную внешнего цикла и снова проходим по всем значениям цикла внутреннего. Действия повторяем до тех пор, пока не будет пройден весь диапазон изменения внешней переменной. Описанный алгоритм зафиксирован на (Рисунок 4.9).

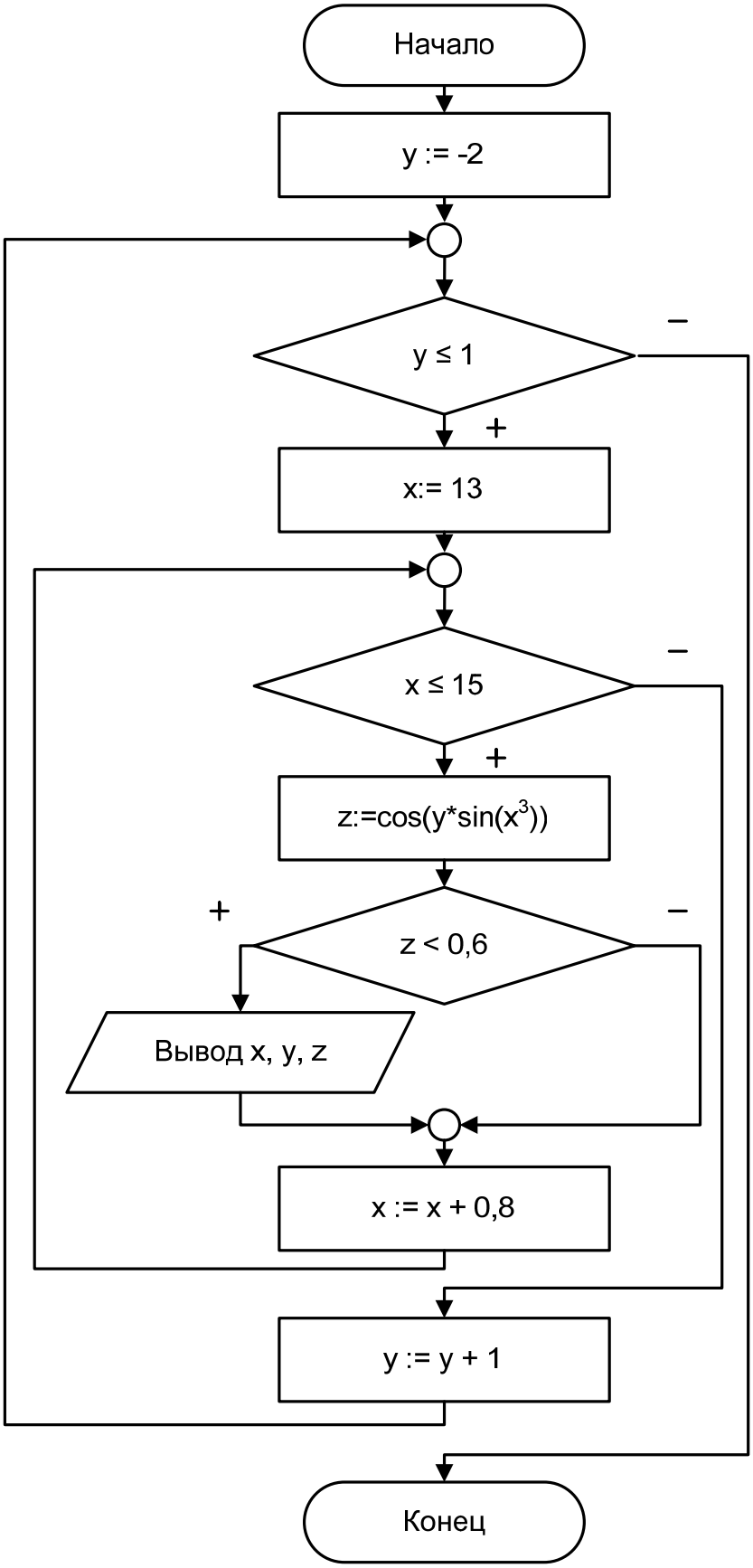


Рисунок 4.9 Табуляция функции с двумя переменными и вывод только тех значений, которые удовлетворяют условию

Программная реализация такая:

```
program ddd;
var x,y,z:real;
begin
  writeLn('табуляция функции cos(y*sin(x*x*x)):');
  y:=-2;
  while y<=1 do
    begin
      x:=13;
      while x<=15 do
        begin
          z:=cos(y*sin(x*x*x));
          if z<0.6 then
            writeLn('z(',x:6:3,', ',y:6:3,') =',z:6:2);
          x:=x+0.8;
        end;
      y:=y+1;
    end;
end.
```

В результате на консоли появится сообщение:

```
табуляция функции cos(y*sin(x*x*x)):
z(13.000, -2.000) = -0.14
z(13.800, -2.000) = -0.40
z(14.600, -2.000) = -0.28
z(13.800, -1.000) =  0.55
z(13.800,  1.000) =  0.55
```

Рассмотрим еще одну интересную задачу.

ПРИМЕР

Для функции $y = e^x / x^2$ *где* x *изменяется от точки* $x = 1$ *с шагом* $\Delta x = 0{,}5$ *в положительном направлении, определить на каком шаге будет выполнено условие* $y > 10^4$.

Решение задачи достаточно простое и представлено на блок-схеме (Рисунок 4.10).

Программная реализация:

```
program ddd;
var x,y:real;
    k:integer;
begin
  x:=1; k:=0;
  repeat
    y:=exp(x)/sqr(x);
    inc(k);
    x:=x+0.5;
  until y>1e4;
  writeLn('условие достигается в точке ', x:8:1);
  writeLn('на ',k,'-м шаге' );
end.
```

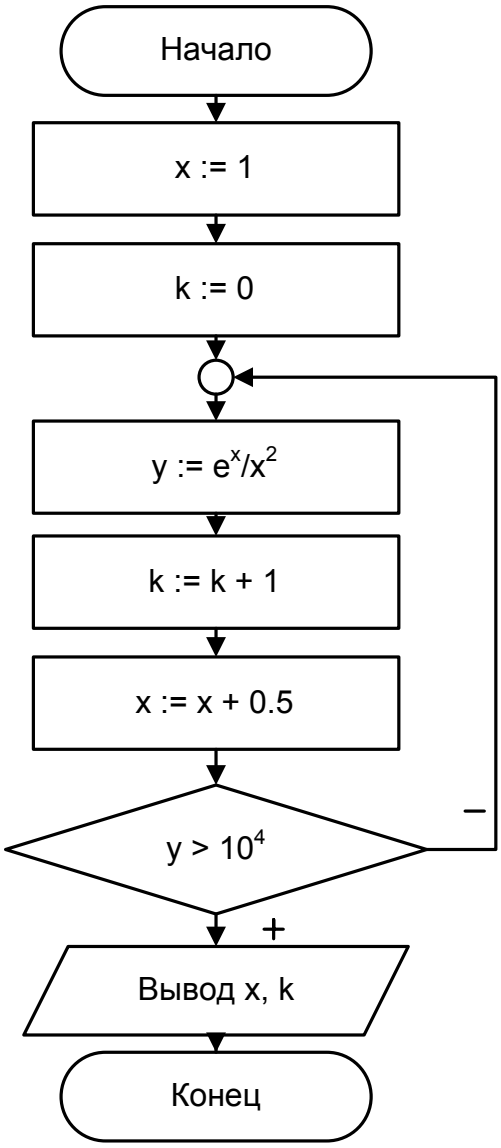


Рисунок 4.10 Поиск шага на котором монотонная функция превысит некоторый порог

В результате получим такой ответ:

```
условие достигается в точке
15.5
на 29-м шаге
```

Как уже отмечалось ранее, при программировании циклов важную роль играют рекуррентные формулы, которые достаточно просто позволяют получить значение элемента некоторой последовательности методом вычисления всех ее членов начиная с первого и заканчивая тем, который нам необходим в конкретной задаче. Эти формулы использовались абсолютно во всех задачах на циклы, которые здесь были рассмотрены. Напомним, что рекуррентной называется такая последовательность, у которой каждый последующий ее член основан на значении предыдущего. Это означает, что рекуррентными будут, в частности, все те операции присваивания, у которых слева от операции стоит переменная встречающаяся справа. Наиболее часто используемая форма рекуррентной записи – это форма рекуррентного подсчета суммы. В задачах на табуляцию таковой формой обладают действия по изменению аргумента ($x := x + \Delta x$). Изменение значения переменной по такой формуле называют *инкрементацией*. Если переменная не увеличивается, а уменьшается (используется формула $x := x - \Delta x$), то действие зазывается *декрементацией*. Стоит отметить, что даже поиск максимума, по своей сути рекуррентен, поскольку каждое последующее его значение основано на предыдущем (хотя и не так явно, как в операциях присваивания).

Практически всегда, когда производят расчет суммы, за начальное значение принимают ноль. Рассмотрим, например, такую задачу:

ПРИМЕР

Найти значение суммы $S = \sum_{k=1}^{6} k^2$. Расписывается эта сумма достаточно просто $S = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2$. В результате будем иметь 91. Блок схема изображена на (Рисунок 4.11 а)).

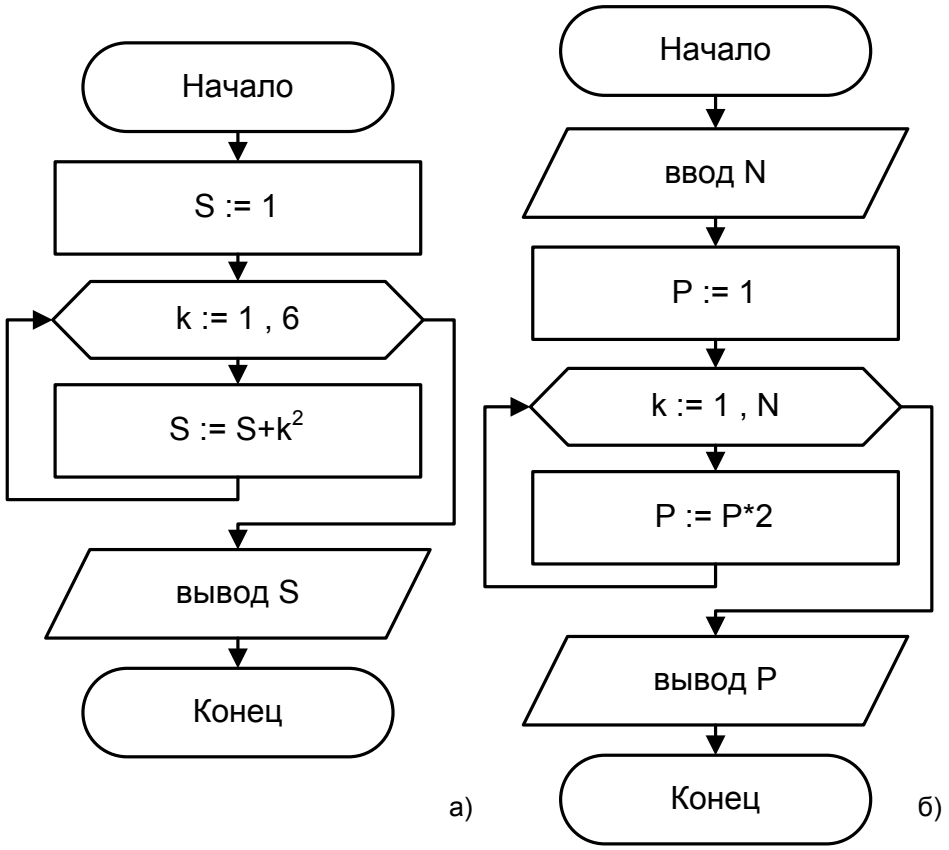


Рисунок 4.11 Задачи на поиск суммы (а)) и произведения (б))

```
program SumS;
var s:real;
    k,N:integer;
begin
  s:=0;
  for k:=1 to 6 do
    s:=s+sqr(k);
  writeLn('сумма равна ', s:8:2);
end.
```

При решении задач на произведения в качестве начального значения принимают 1.

ПРИМЕР

Найти значение выражения $P = 2^N$ *не используя операцию возведения в степень*. Эта задача эквивалентна расчету такой формулы: $\prod_{k=1}^{N} 2 = 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2$ (всего N двоек). Решение показано на (Рисунок 4.11 б)).

```
program PrP;
var k,N:integer;
    P:longInt;
begin
  writeLn('введите N');
  readLn(N);
  p:=1;
```

```
  for k:=1 to N do
    p:=p*2;
  writeLn('2 в степени ',N,' равно ', p);
end.
```

Более интересной представляется задача на вложенные циклы, где прежде чем рассчитать значение по внешней рекуррентной формуле, приходится использовать внутреннюю.

ПРИМЕР

Найти значение суммы $S = \sum_{k=1}^{N}\left(k \cdot \sum_{j=k}^{2k} \cos(j) \right)$.

Здесь для подсчета внешней суммы каждый раз придется пересчитывать внутреннюю. При решении внешнюю обозначим переменной S_ext, а внутреннюю, соответственно, S_int. Структурно получается «*сумма в сумме*», что отражено на блок-схеме (Рисунок 4.12).

Программа выглядит так:

```
program ddd;
var S_ext,S_int:real;
    k,N,j:integer;
begin
  writeLn('введите N');
```

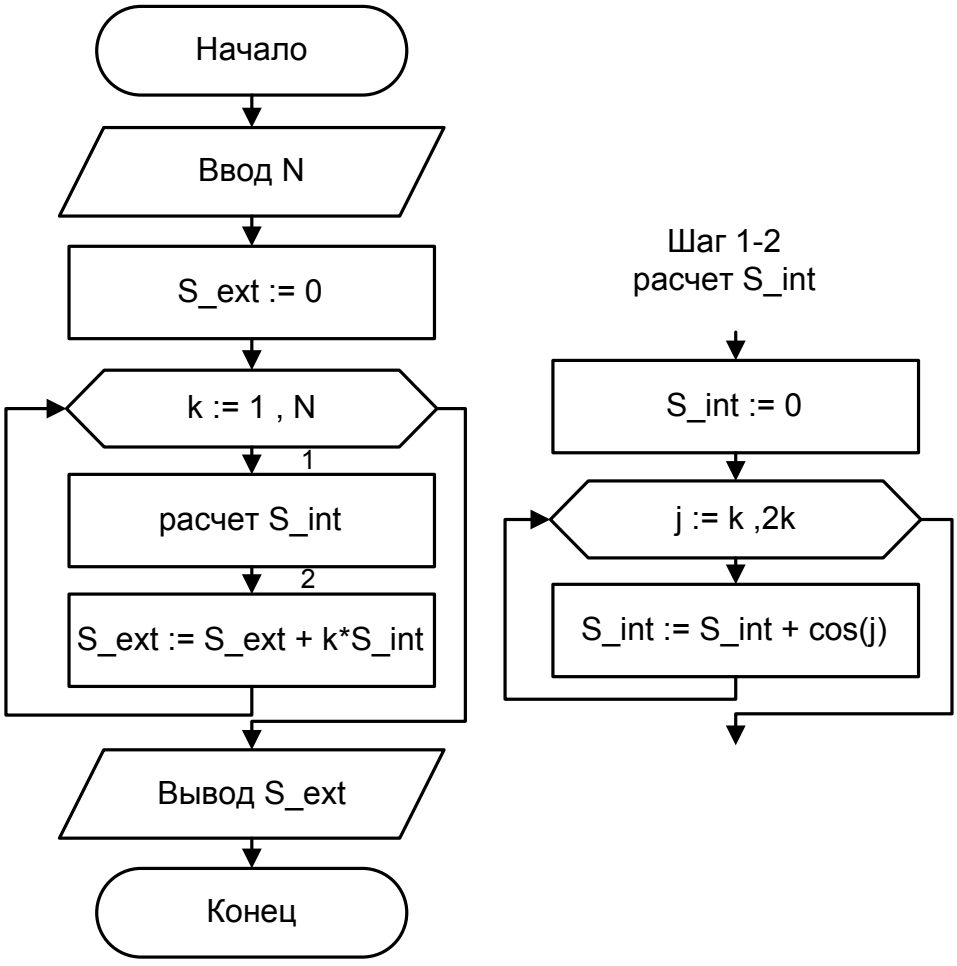


Рисунок 4.12 Задача на вложенные суммы

```
  readLn(N);
  S_ext:=0;
  for k:=1 to N do
    begin
      S_int:=0;
      for j:=k to 2*k do
        S_int:=S_int+cos(j);
      S_ext:=S_ext+k*S_int;
    end;
  writeLn('сумма равна ', S_ext:8:2);
end.
```

Иногда, несмотря на кажущуюся сложность, задача решается несколько проще, чем полагалось изначально.

ПРИМЕР

Найти значение суммы $S = \sum_{k=1}^{N} \frac{k^2}{k!}$.

Здесь для подсчета внешней суммы потребуется на каждой новой итерации вычислять факториал для текущего значения k. На первый взгляд, без вложенного цикла на произведение не обойтись. Однако, если посмотреть внимательно на формулу и вспомнить замечательное свойство факториала заключающееся в том, что для вычисления последующего значения достаточно знать предыдущее, то алгоритм запишется намного короче (Рисунок 4.13 а)). И, что самое главное, без вложенных циклов.

На комбинацию

```
program ddd;
var s:real;
    f,k,N:integer;
begin
  cls;
  writeLn('введите N');
  readLn(N);
  s:=0;
  f:=1;
  for k:=1 to N do
    begin
      f:=f*k;
      s:=s+sqr(k)/f;
    end;
  writeLn('сумма равна ', s:8:2);
end.
```

ПРИМЕР

Найти значение выражения $\sqrt{x+\sqrt{x+\sqrt{x+...}}}$ *(всего 20 корней).*

Для вычисления этого выражения необходимо самостоятельно придумать рекуррентную формулу. Это уже будет и не сумма и не произведение, а нечто более сложное. Чтобы понять, какая формула здесь

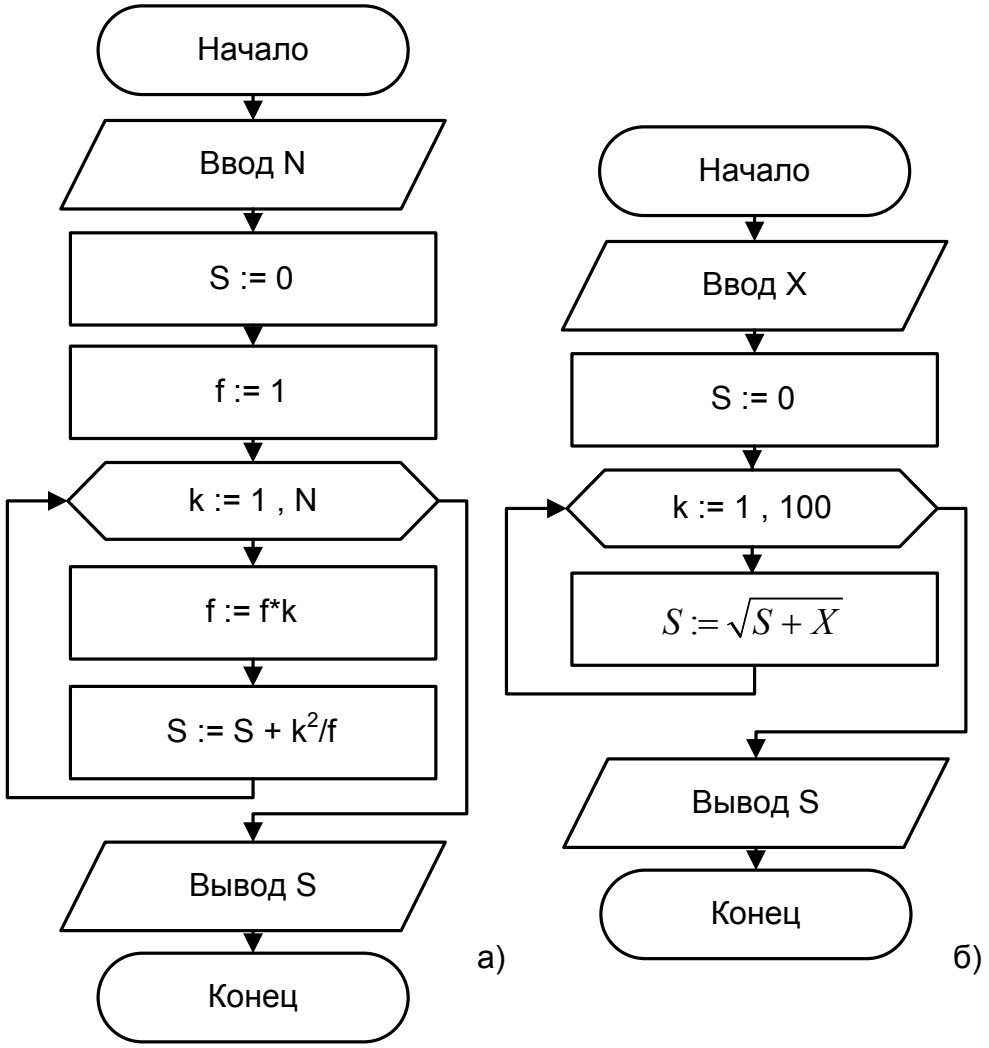


Рисунок 4.13 Сложная сумма - а) и нестандартная рекуррентная формула - б)

применяется, можно последовательно начать расписывать вычисляемое значение на каждом шаге. Итак,

i=1, $S_1 = \sqrt{x}$;

i=2, $S_2 = \sqrt{x + \sqrt{x}} = \sqrt{x + S_1}$;

i=3, $S_3 = \sqrt{x + \sqrt{x + \sqrt{x}}} = \sqrt{x + S_2}$;

…

$i=k$, $S_1 = \sqrt{x + S_{k-1}}$;

…

i=20, $S_{20} = \sqrt{x + S_{19}}$;

Видна рекуррентная зависимость $S := \sqrt{x + S}$. Осталось определить точку входа. Этой точкой, очевидно, будет $S_0 = 0$, поскольку $S_1 = \sqrt{x + S_0} = \sqrt{x}$. Теперь, зная вид рекуррентной зависимости и точку начальное значение, достаточно просто составить алгоритм для вычисления S (Рисунок 4.13 б)).

Программа представлена ниже:

```
program sqrtS;
var k:integer;
    x,s:real;
begin
  cls;
  writeLn('введите x');
  readLn(x);
  s:=0;
  for k:=1 to 20 do
      s:=sqrt(x+s);
  writeLn('s равно ', s);
end.
```

При x=30 сумма получается равной 6, а, например, при x=7, она становится такой: 3.19258240356725.

Весьма интересными являются задачи на *перекрестные рекуррентные зависимости*. Перекрестными рекуррентными зависимостями называют такие, у которых каждое последующее значение вычисляется на основе *нескольких* предыдущих значений. Самый простой пример – это *авторегрессионные* последовательности. Наиболее известная последовательность задаваемая рекуррентно на основе двух предыдущих значений является *последовательность Фибоначчи*. Последовательность задается таким образом: $a_1 = 1$, $a_2 = 1$, $a_3 = a_1 + a_2$, $a_4 = a_3 + a_2$, $a_k = a_{k-2} + a_{k-1}$.

ПРИМЕР

Вывести 10 первых членов последовательности Фибоначчи.

Решается задача в один цикл. Решение представлено на (Рисунок 4.14 а)).

Ниже идет программа:

```
program ddd;
var ak_3,ak_2,ak_1,ak:real;
    k:integer;
begin
  ak_2:=1;
  ak_1:=1;
  k:=2;
  writeln(1:5,'   z=',ak_1);
  writeln(2:5,'   z=',ak_2);
  repeat
    ak:=ak_1+ak_2;
    ak_2:=ak_1;
    ak_1:=ak;
    inc(k);
    writeln(k:5,'   z=',ak);
  until k=10;
end.
```

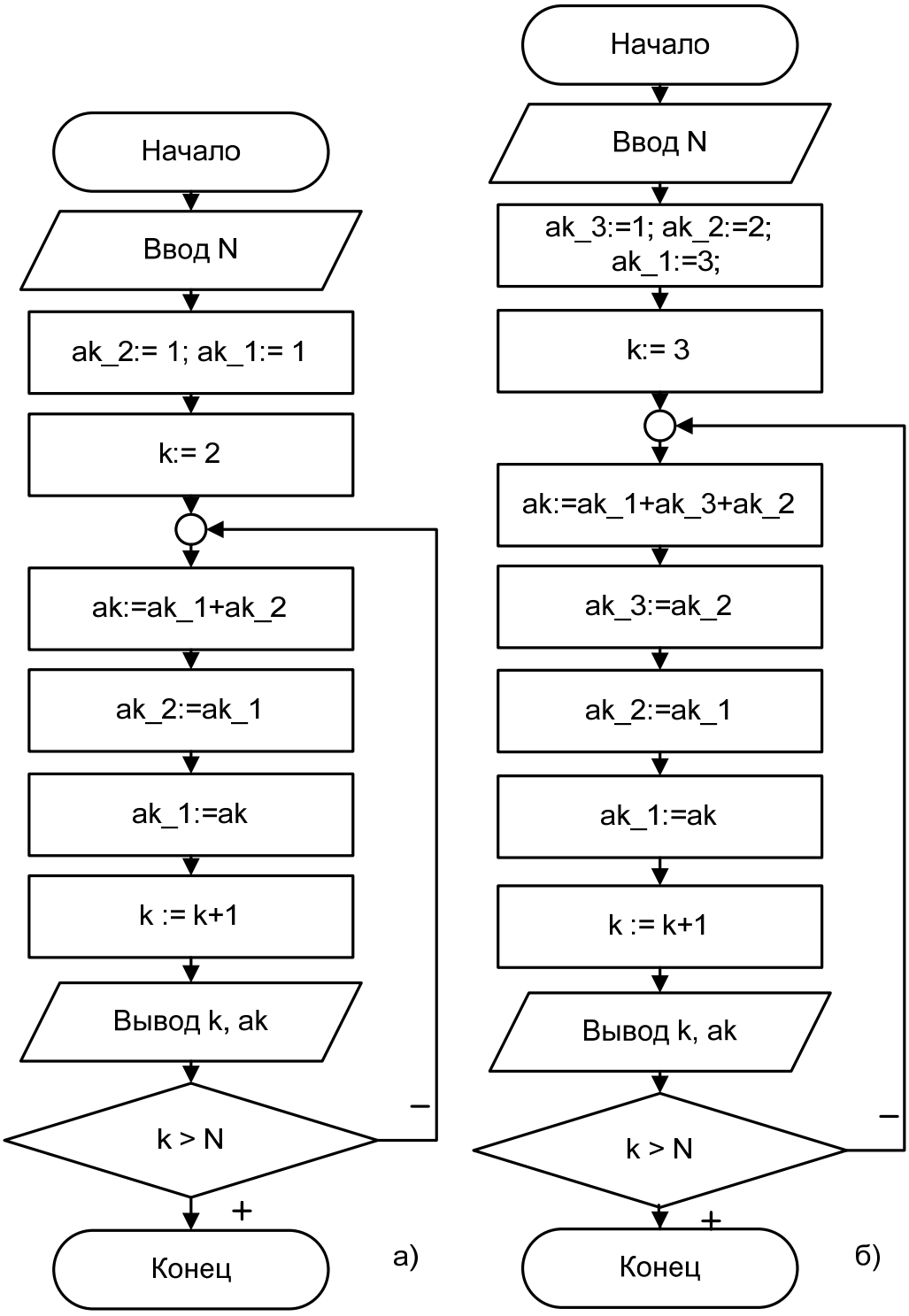


Рисунок 4.14 Последовательность Фибоначчи - а) и последовательность основанная на трех предыдущих членах

Результат таков:

```
1    z=1
2    z=1
3    z=2
4    z=3
5    z=5
6    z=8
7    z=13
```

```
 8    z=21
 9    z=34
10    z=55
```

Можно несколько модифицировать предыдущую последовательность и сделать так, чтобы в авторегрессионной формуле участвовало три предыдущих члена.

ПРИМЕР

Вывести N $(N \le 3)$ *первых членов последовательности заданной рекуррентно* $a_k = a_{k-1} + a_{k-2} + a_{k-3}$, $a_1 = 1$, $a_2 = 2$, $a_3 = 3$.

При решении этой задачи важно понять суть перехода от пары переменных к трем. В алгоритме на каждой итерации применяется так называемый *циклический сдвиг* переменных. Т.е. переменные принимают значения тех, которые им предшествуют. Ну а самая последняя, т.е. текущая переменная принимает свое значение согласно заданной рекуррентной формуле (Рисунок 4.14 б)). Имена переменным даны согласно порядку их расположения в последовательности, т.е. $a_k = ak$, $a_{k-1} = ak\_1$, $a_{k-2} = ak\_2$, $a_{k-3} = ak\_3$. Программа такова:

```
program ddd;
var ak_3,ak_2,ak_1,ak:real;
    k,N:integer;
begin
  writeLn('введите N');
  readLn(N);
  while N<=3 do
    begin
      writeLn('введите N');
      readLn(N);
    end;
  ak_3:=1;
  ak_2:=2;
  ak_1:=3;
  k:=3;
  writeln(1:5,'   z=',ak_3);
  writeln(2:5,'   z=',ak_2);
  writeln(3:5,'   z=',ak_1);
  repeat
    ak:=ak_1+ak_3+ak_2;
    ak_3:=ak_2;
    ak_2:=ak_1;
    ak_1:=ak;
    inc(k);
    writeln(k:5,'   z=',ak);
  until k=N;
end.
```

ПРИМЕР

Вывести числа от 1 *до N построчно, причем в первой строке число выводится* 1 *раз, во второй выводится двойка* 2 *раза*, *в третьей тройка* 3 *раза и т.д. В строке N число N выводится N раз.*

Для решения этой задачи потребуется использование вложенного цикла. Во внешнем цикле будет происходить изменение значения выводимого числа,

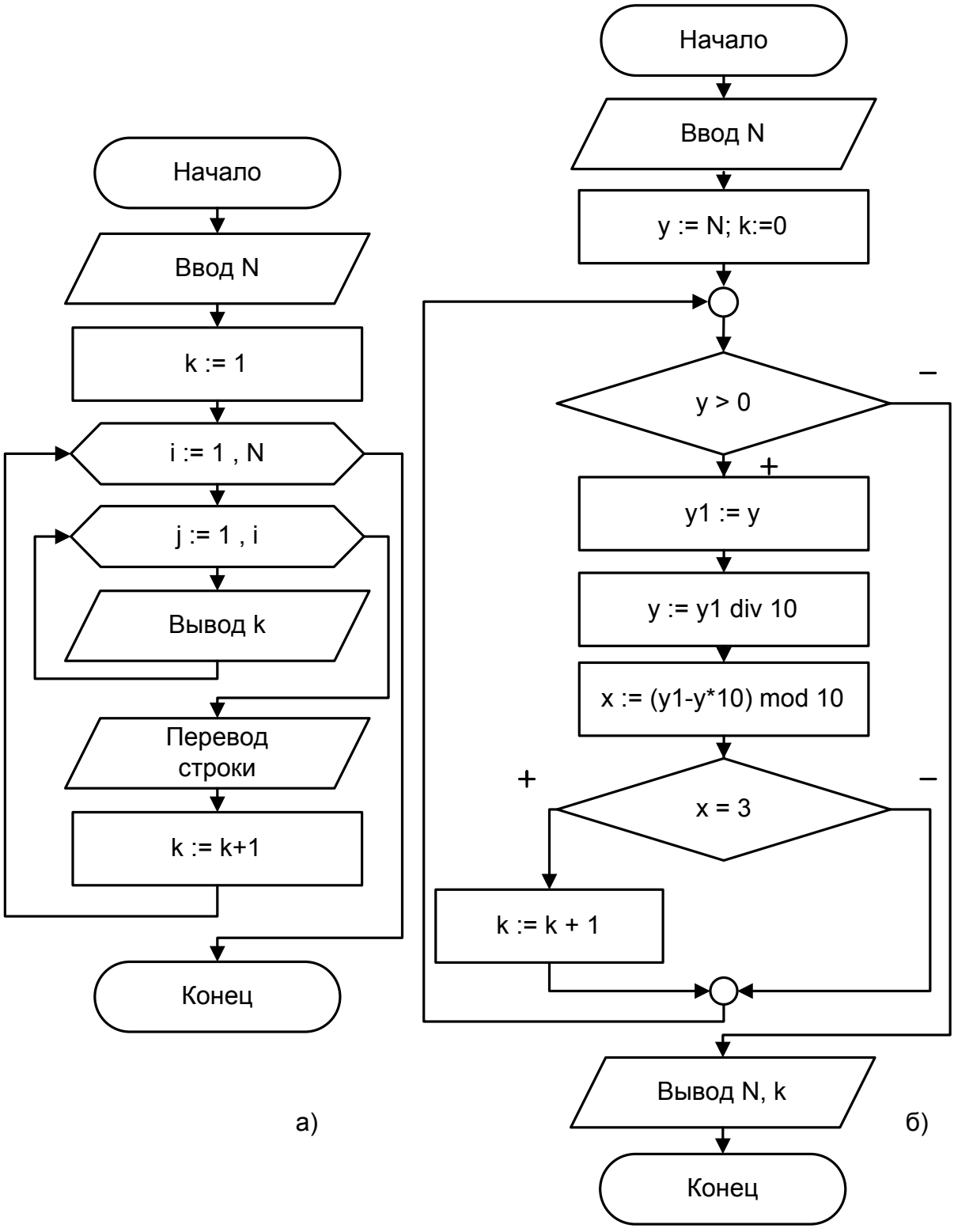


Рисунок 4.15 Вывод чисел "пирамидой" – а) и анализ числа на встречаемость в его записи цифр – б)

а во внутреннем – многократный вывод числа соответственно номеру строчки. Блок схема показана на Рисунок 4.15 а). Язык *Pascal* вот такой:

```
program BAC;
var i,j,N,k:integer;
begin
  writeLn('введите N');
  readLn(N);
  k:=1;
  for i:=1 to N do
    begin
      for j:=1 to i do
        write(k:5);
      writeLn;
      inc(k);
    end;
end.
```

В результате мы увидим на консоли:

```
1
2    2
3    3    3
4    4    4    4
5    5    5    5    5
```

ПРИМЕР

Ввести число N, выяснить сколько раз цифра 3 *входит в запись этого числа*.

Алгоритм основан на последовательном рассмотрении остатка от деления на 10 младшего разряда рассматриваемого числа. Алгоритм представлен на (Рисунок 4.15 б)). Здесь переменная *y* хранит число равное целой части от деления текущего числа на 10. Переменная *y1* – буферная переменная для хранения предыдущего (неделенного на 10) числа. Соответственно, остаток от деления проверяется для чисел *y**10 и *y1*. Если остаток равен 3, то мы производим инкрементацию переменной *k,* в которой хранится количество встреченных троек.

```
program cifra3;
var x,y,y1,k,N:integer;
begin
  writeLn('введите N');
  readLn(N);
  y:=N;
  while y>0 do
    begin
      y1:=y;
      y:=y1 div 10;
```

```
      x:= (y1-y*10) mod 10;
      if x=3 then
        inc(k);
    end;
    writeLn('цифра3 вх. в число ',N,'всего ',k,' раз');
end.
```

Вот что будет на консоли в результате работы программы:

```
введите N
3331333
цифра 3 вх. в число 3331333 всего 5 раз
```

5. ОДНОМЕРНЫЕ МАССИВЫ

5.1 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 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 (Рисунок 5.1).

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

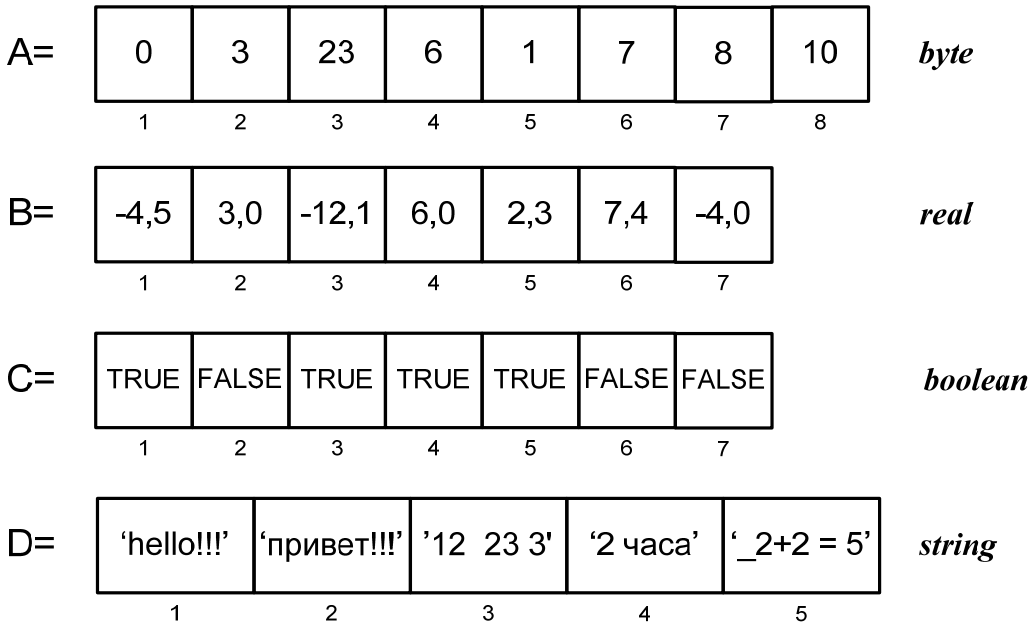


Рисунок 5.1 Примеры одномерных массивов разных типов. В ячейках записаны элементы, под каждой ячейкой записан индекс элемента в массиве.

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

Пример объявления массивов, изображенных на рисунке (Рисунок 5.1) 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 (Рисунок 5.1). Однако, объявление переменной есть процесс выделения под нее памяти. Еще до начала работы программы мы должны знать, какого размера массив нам потребуется в процессе ее исполнения. К сожалению, не всегда заранее известно о точной его длине. Потому приходится использовать динамические массивы (которых мы 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 нам 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. К сожалению, 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 рисунке (Рисунок 5.1).

```
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.

5.2 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 Рисунок 5.2.

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[1]. Для того, чтобы появился смысл, массив нужно заполнить. 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 Рисунок 5.3 а).

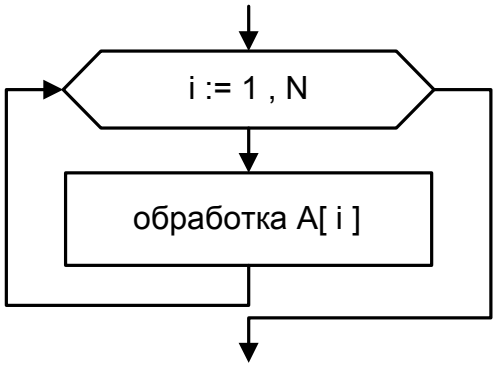


Рисунок 5.2 Обработка каждого элемента массива

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

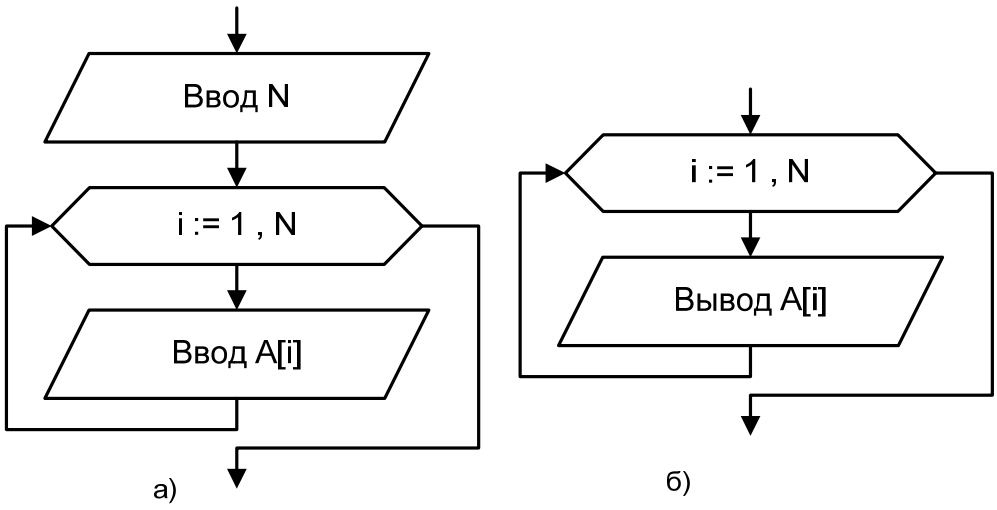


Рисунок 5.3 Алгоритмы ввода и вывода одномерного массива

```
writeLn('Введите количество элементов в массиве');
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 Рисунок 5.3 б)):

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

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;
…
```

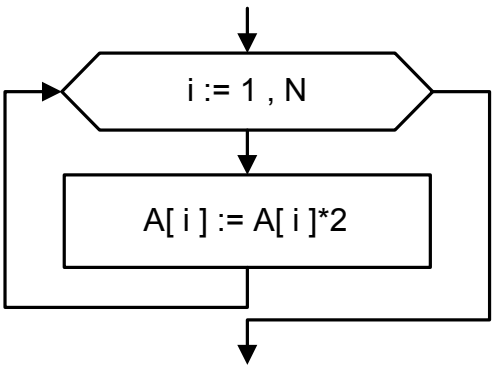


Рисунок 5.4 Умножение элементов массива на 2

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).

[1] 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);

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];
…
```

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

Пусть, например, нужно скопировать элементы массива ***A*** в массив ***B***. 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 мы 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];
```

5.3 Элементы, удовлетворяющие некоторому условию (поиск)

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 Рисунок 5.5. Условие может быть каким угодно, например, положительность, равенство чему-либо, четность и т.д.

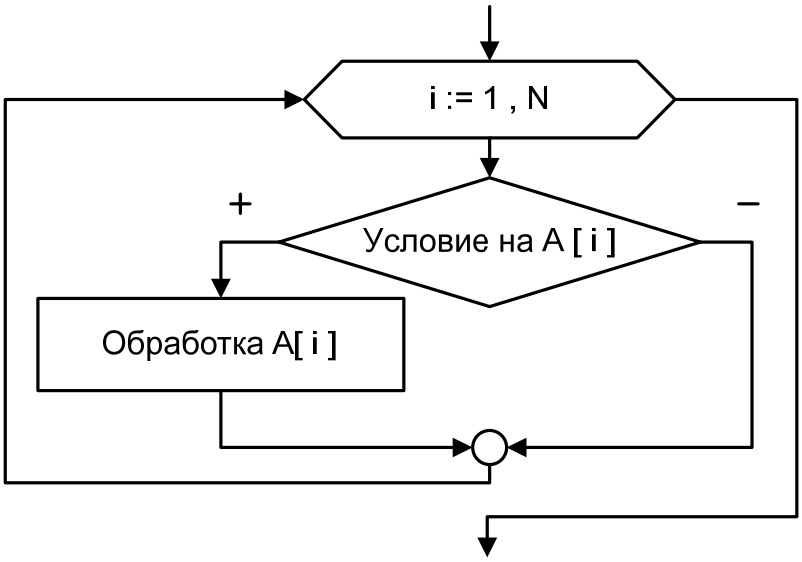


Рисунок 5.5 Обработка элементов массива, удовлетворяющих некоторому условию

ПРИМЕР

Рассмотрим задачу подсчета среднего арифметического четных элементов массива. Решением будет являться следующий алгоритм, записанный в виде блок-схемы (Рисунок 5.6) и в виде программы:

```
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.
```

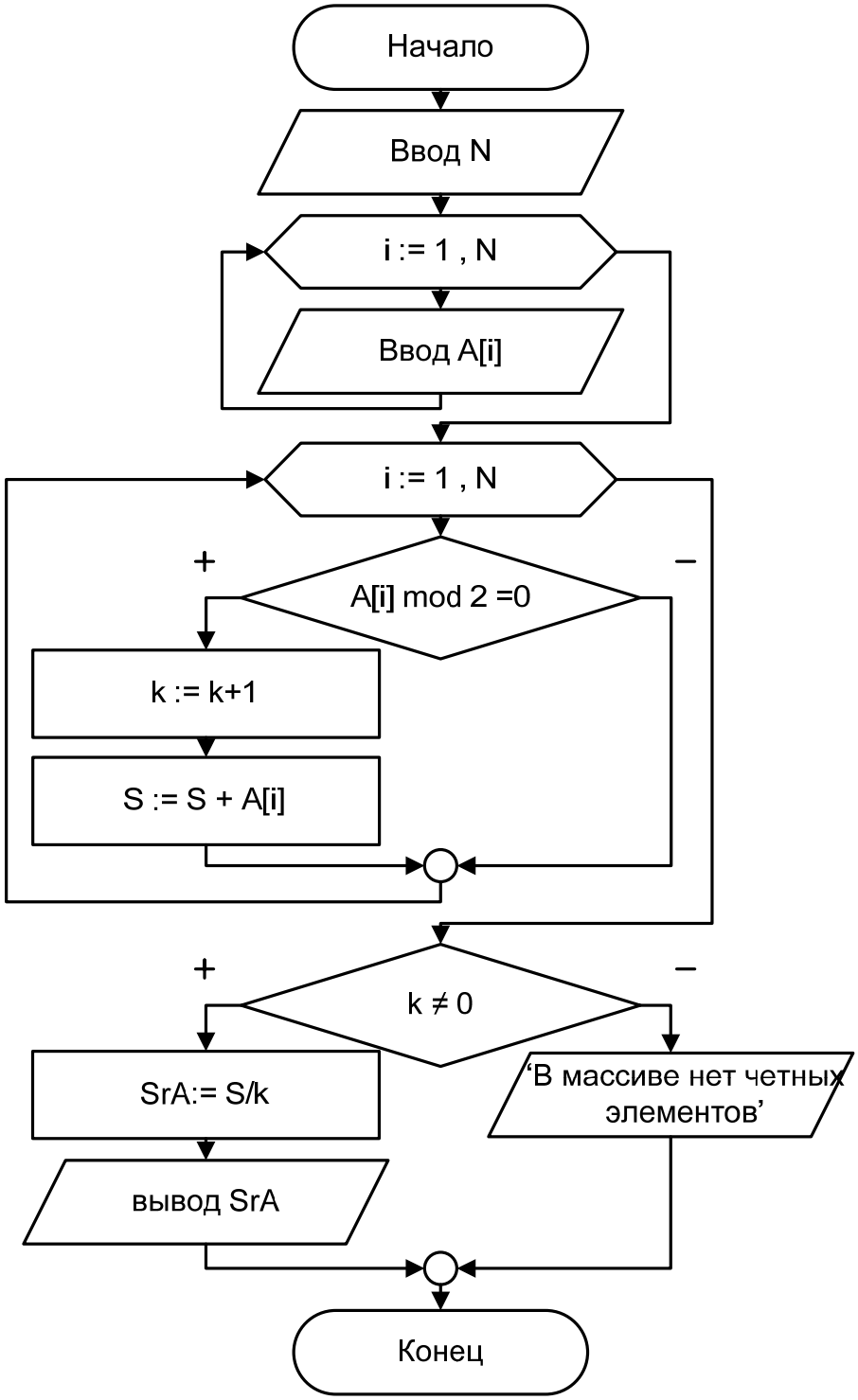


Рисунок 5.6 Подсчет среднего арифметического четных элементов массива

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 мы в алгоритм после цикла вставили развилку, 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***.

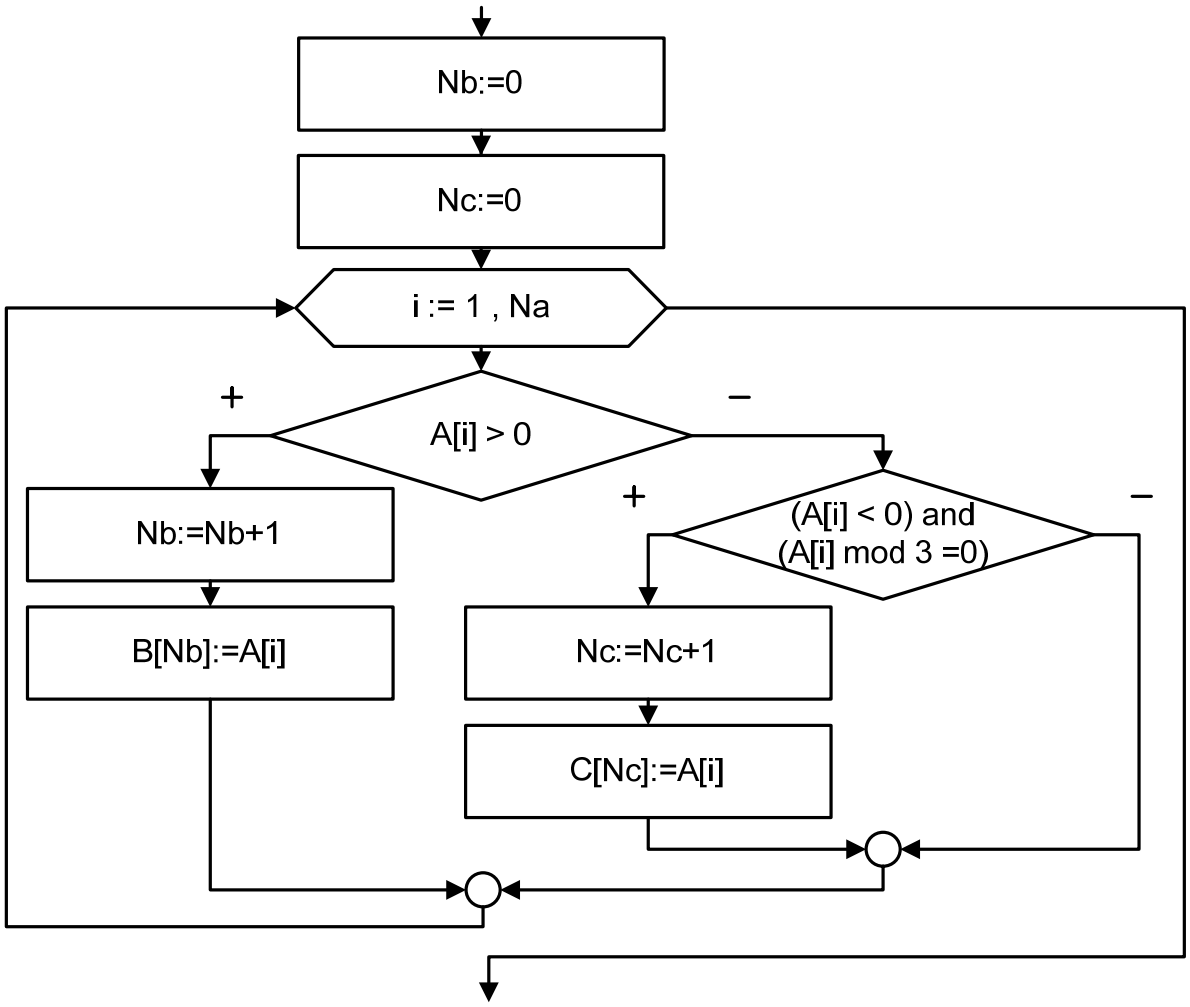


Рисунок 5.7 Формирование из одного массива пары новых

Алгоритм достаточно прост:

```
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;
```

Блок-схема изображена на (Рисунок 5.7). Здесь 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) достаточно прост. Упрощенно его можно представить следующим образом: Пусть у нас есть кучка камней и нам нужно найти наибольший среди них. Для этого мы 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 мы 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 (Рисунок 5.8):

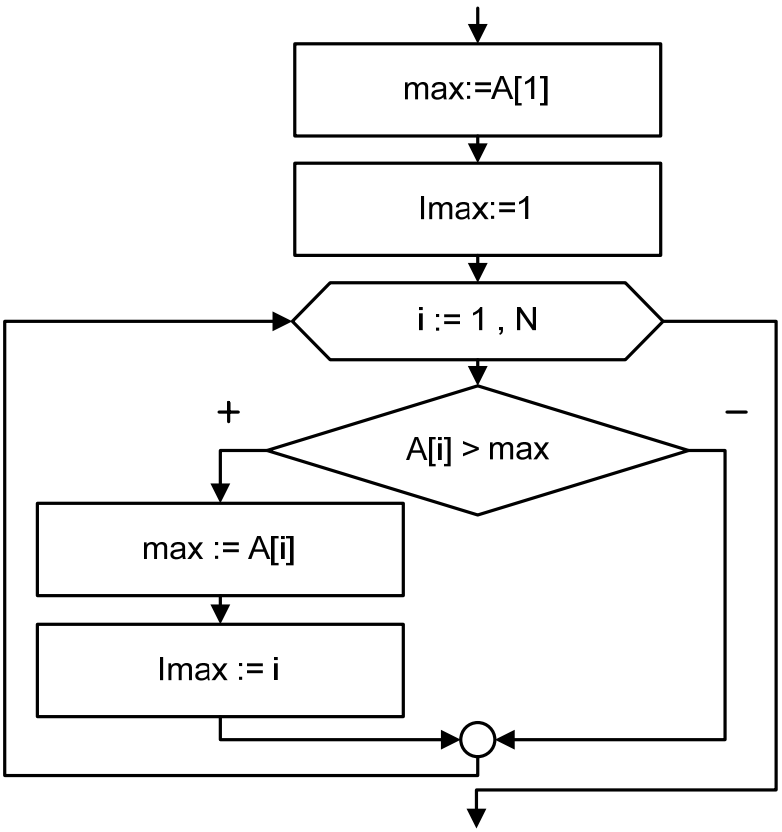


Рисунок 5.8 Поиск максимума и его индекса

```
…
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 мы и накладываем 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 (Рисунок 5.9):

```
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');
```

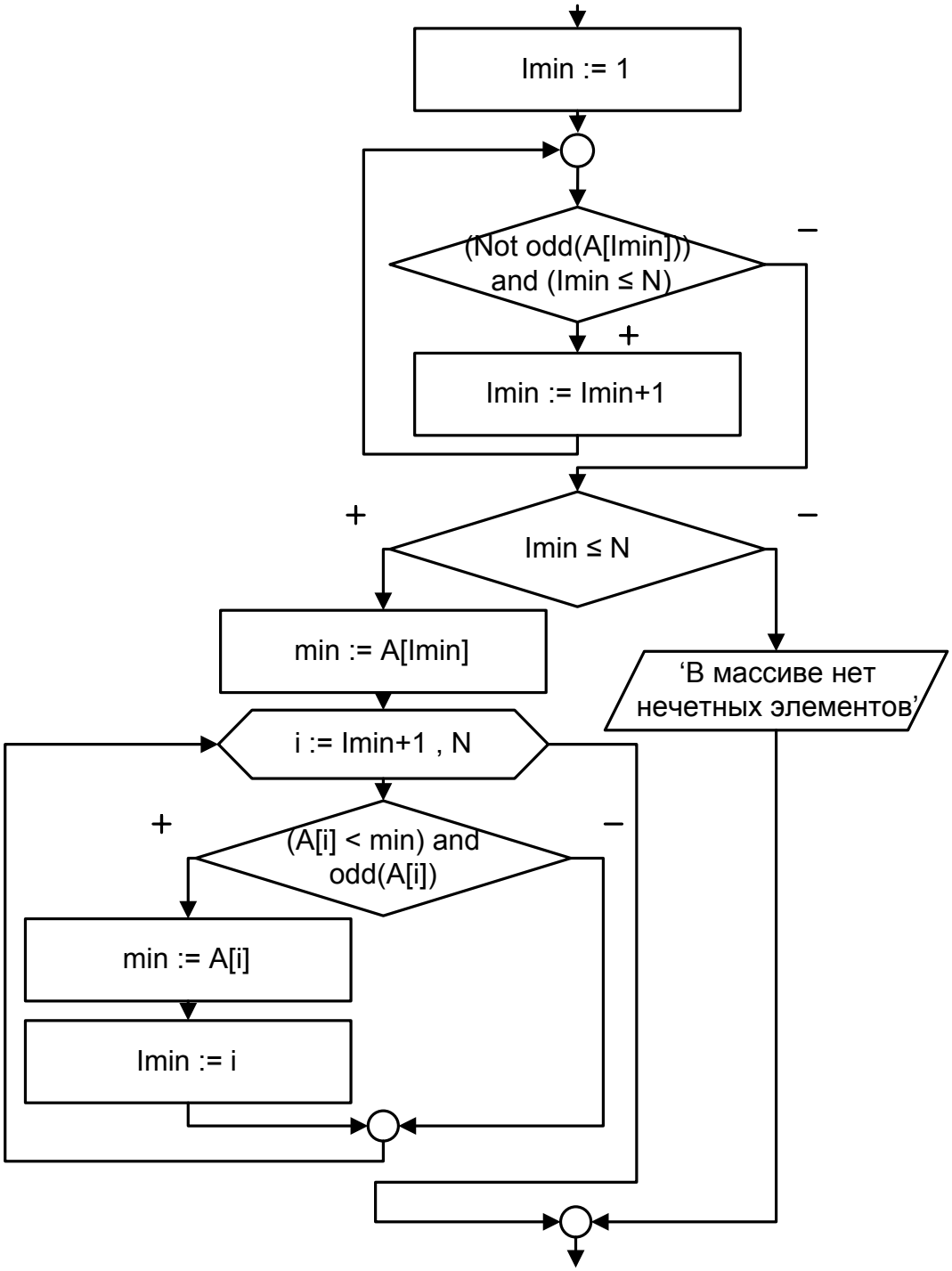


Рисунок 5.9 Поиск минимального среди нечетных в два последовательных цикла

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. 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 (Рисунок 5.10).

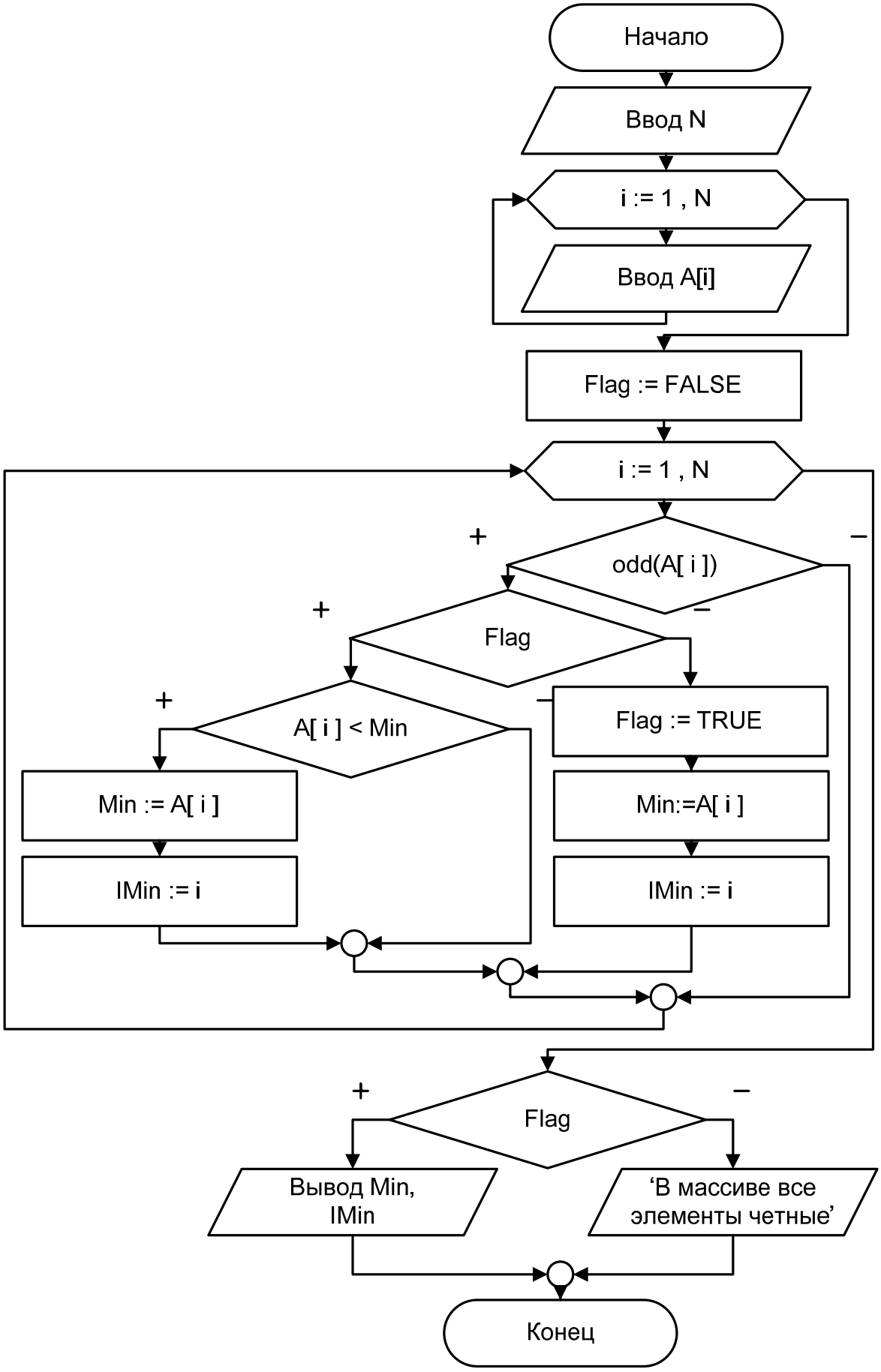


Рисунок 5.10 Поиск минимального среди нечетных в один цикл

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.
```

5.4 Обработка массивов по индексам. Перестановка элементов

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.

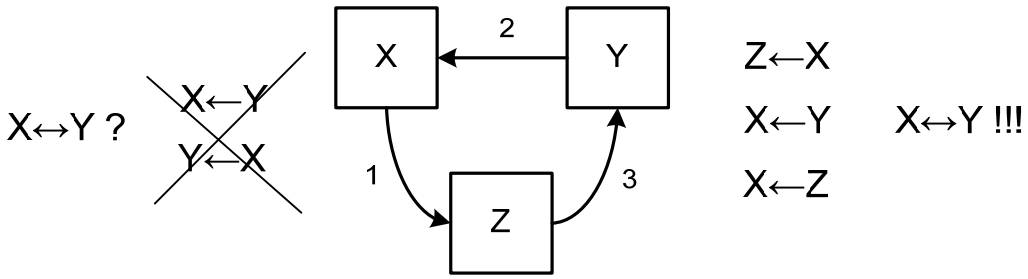


Рисунок 5.11 Обмен значениями пары переменных в три действия

Еще одной особенностью данной задачи является операция перестановки. Нам 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 (Рисунок 5.11). 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 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***.

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 (Рисунок 5.12 а)):

```
for i:=1 to N do
  if A[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 (Рисунок 5.12 б)).

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

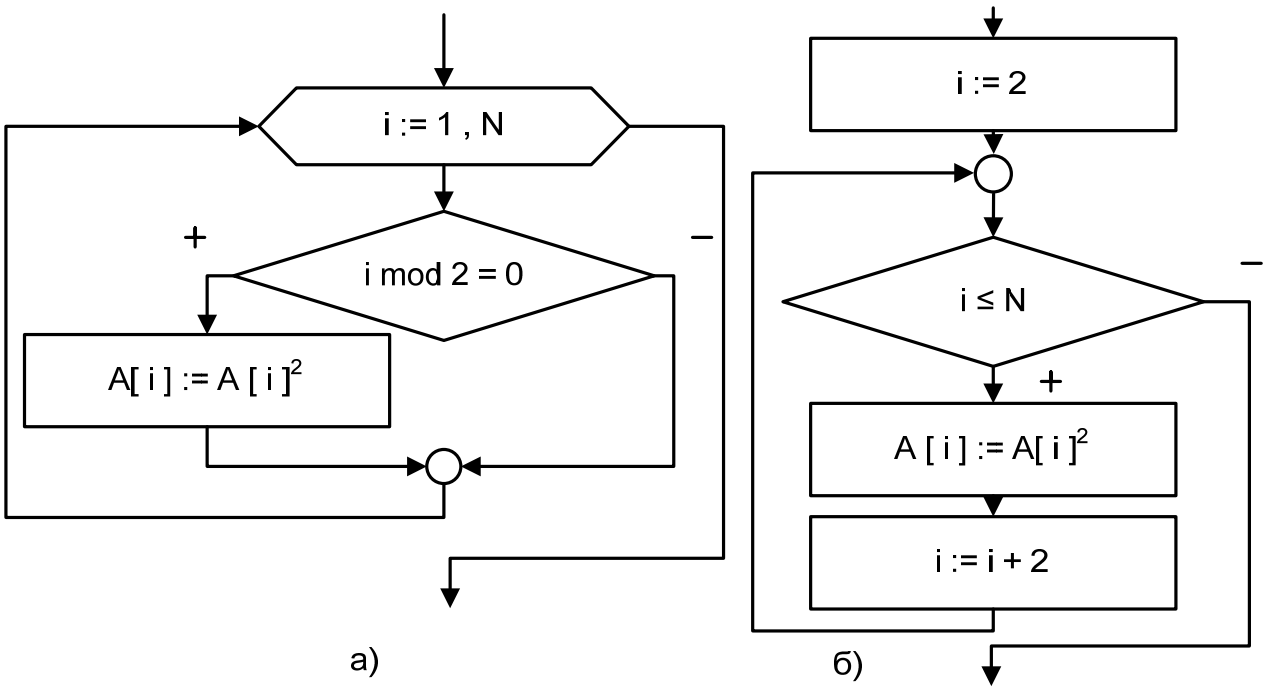


Рисунок 5.12 Перебор всех индексов а) и непосредственное вычисление б)

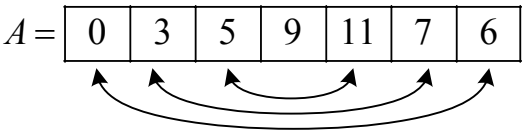
Теперь разберем некоторые более сложные алгоритмы перестановок в одномерном массиве.

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.

Мы поступим иначе. Будем читать массив одновременно с двух сторон, двигаясь к его центру, в процессе движения крайние элементы будем обменивать местами (если мы 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).

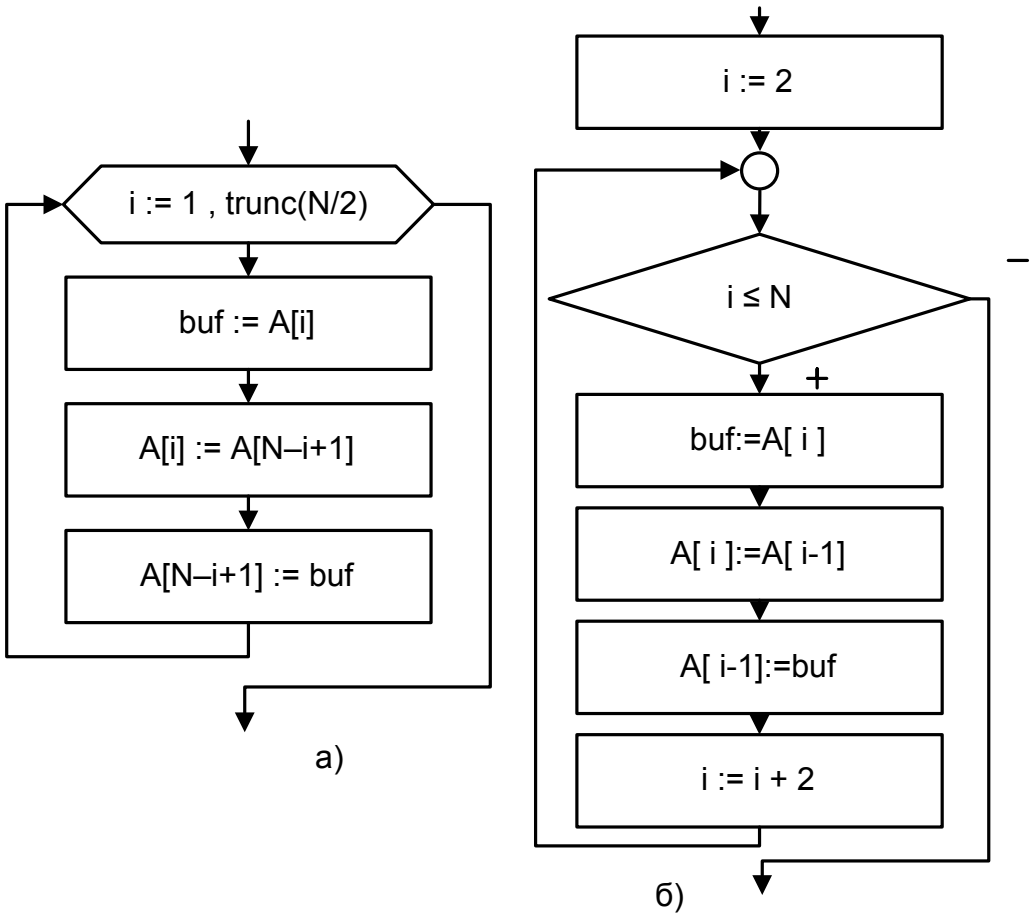


Рисунок 5.13 Инверсия массива а) и перестановка соседних элементов б)

Реализация перестановки элементов такова (Рисунок 5.13 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;
```

Все очень просто. Мы для движения в прямом направлении используем индекс ***i***, а для обратного прохода индекс вычисляется по формуле ***N–i+1***. Действительно, проведем *трассировку*[1] 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.

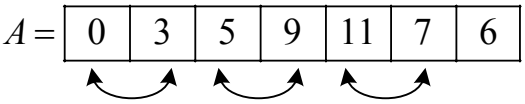
ПРИМЕР

Поменять местами соседние элементы массива.

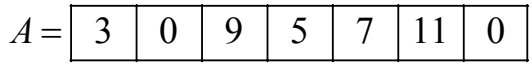
[1] Трассировка есть процесс записи значений переменных на каждом шаге работы программы

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 (Рисунок 5.13 б)):

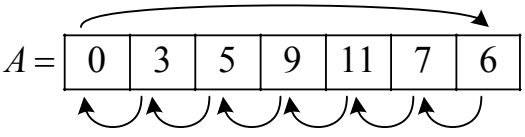
```
i:=2;
while i<=N do
  begin
    buf:=A[i];
    A[i]:=A[i-1];
    A[i-1]:=buf;
    i:=i+2;
  end;
```

ПРИМЕР

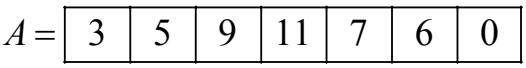
Произвести единичный *циклический сдвиг* элементов массива вправо. Под циклическим сдвигом понимается изменение положения каждого элемента на одну позицию (в данном случае вправо). Соответственно, последний элемент окажется за пределами массива, а на месте первого образуется вакансия. При циклическом 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:



Алгоритм этого процесса следующий (Рисунок 5.14):

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

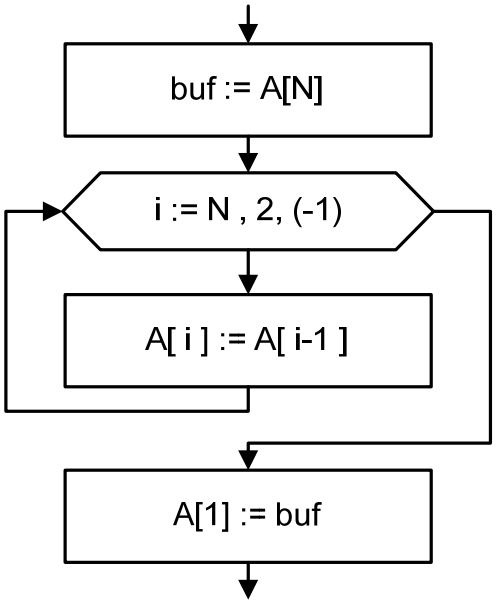


Рисунок 5.14 Циклический сдвиг вправо

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*** (Рисунок 5.15).

```
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;
```

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

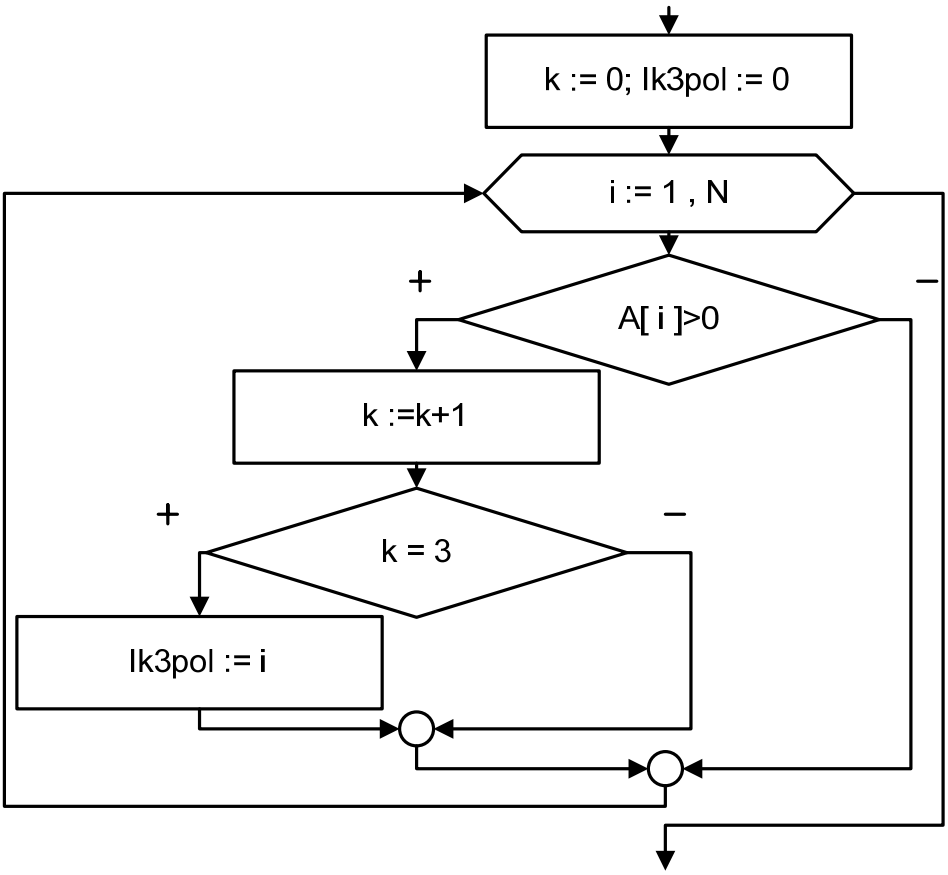


Рисунок 5.15 Поиск третьего положительного

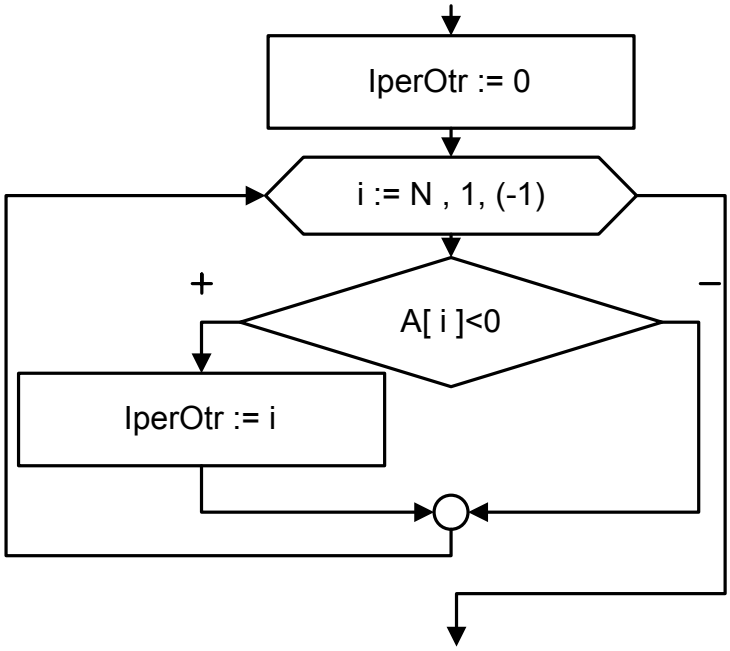


Рисунок 5.16 Поиск первого отрицательного элемента

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 пропустить цикл в обратном порядке, в результате получим последовательное 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 (Рисунок 5.16):

```
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 (Рисунок 5.17 а)).

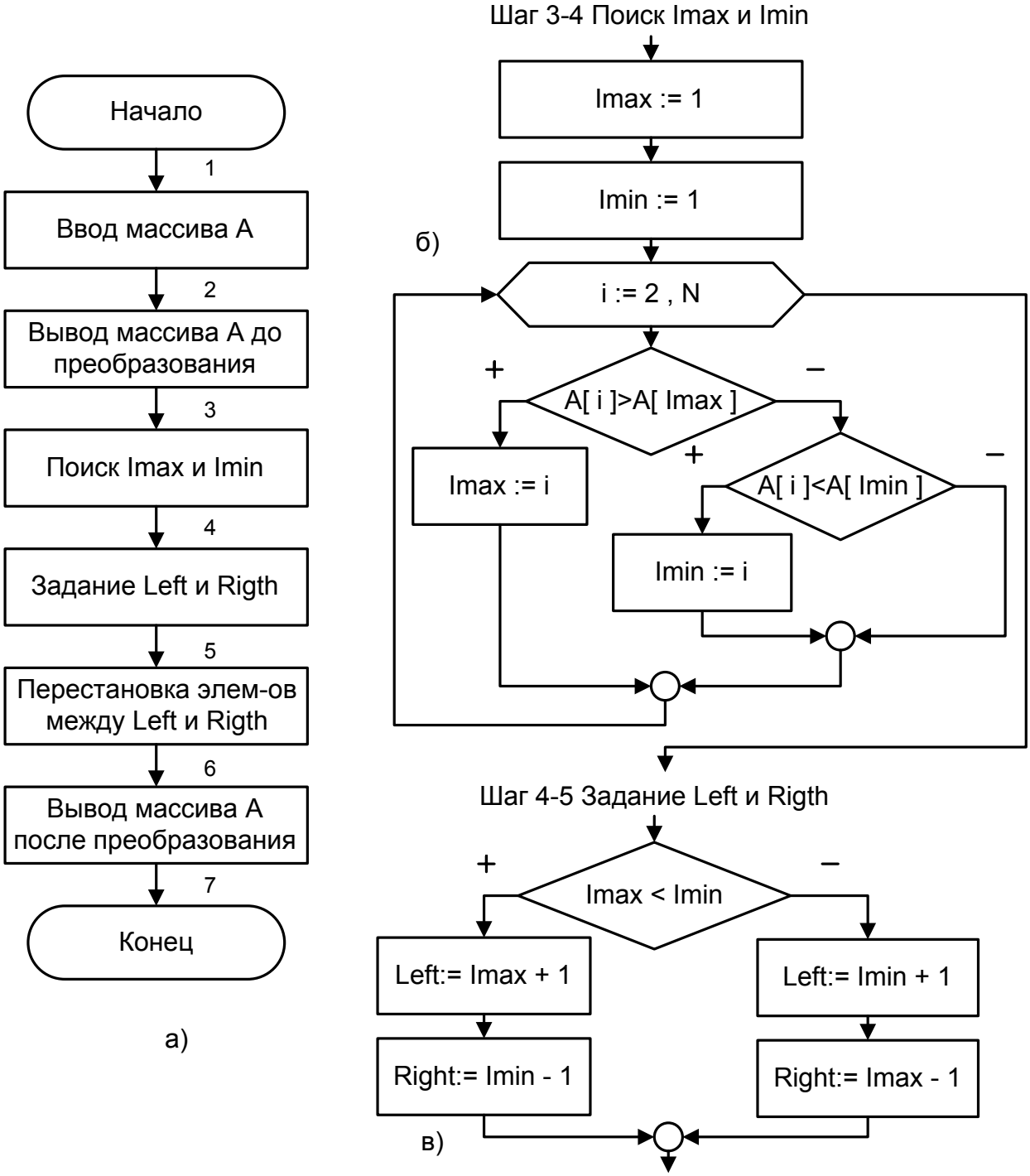


Рисунок 5.17 Перестановка в обратном порядке элементов расположенных между максимумом и минимумом. а) – общий алгоритм; б) – поиск Imin и Imax; с) – задание левой и правой границ

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

Чтобы решить эту задачу нам потребуется сразу после ввода массива найти положение максимума и минимума ***Imax*** и ***Imin*** (Рисунок 5.17 б)). Сам же ввод (шаг 1-2) и вывод (как исходного (шаг 2-3), так и преобразованного

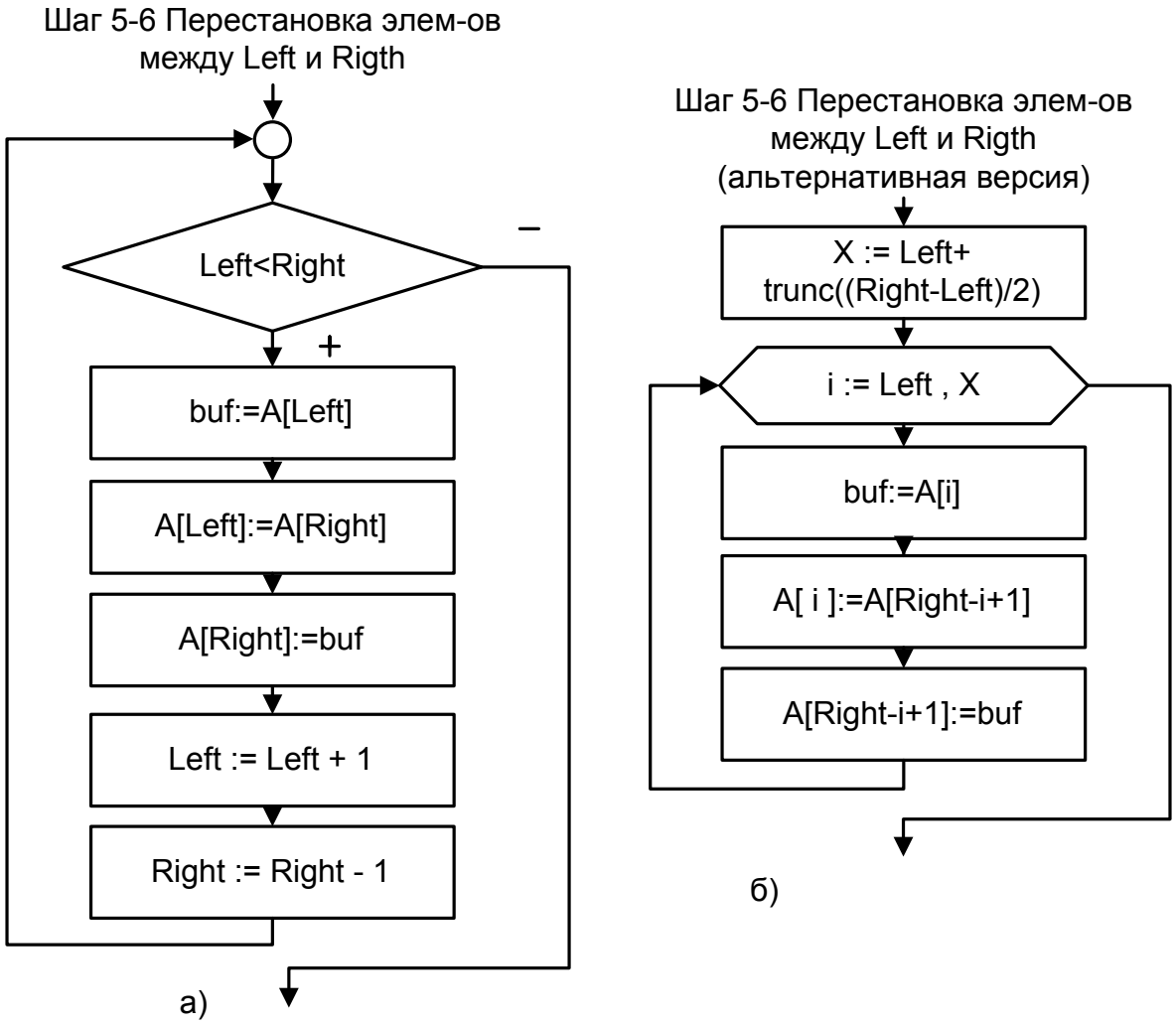


Рисунок 5.18 Перестановка в обратном порядке элементов между Left и Right а)– с изменением границ и б)– непосредственно вычисляя граничные индексы

(шаг 6-7)) здесь не расписывается, поскольку это стандартные алгоритмы и их блок-схемы изображены, например, на Рисунок 5.3.

Зная положение максимума и минимума теперь нужно определить, что из них встречается раньше для того, чтобы корректно задать границы изменения переменной цикла. Левая граница у нас получила название ***Left***, а правая ***Right*** (Рисунок 5.17 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 нам не нужно следить за корректностью формул правой (***Rigth***) и левой (***Left***) границы диапазона. Нам лишь 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 (Рисунок 5.18 а)). Альтернативный алгоритм мы тоже приводим, для того, чтобы видеть, что применяя его запутаться в индексах несколько проще (Рисунок 5.18 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.
```

5.5 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 у нас, 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 мы 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 у нас 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
```

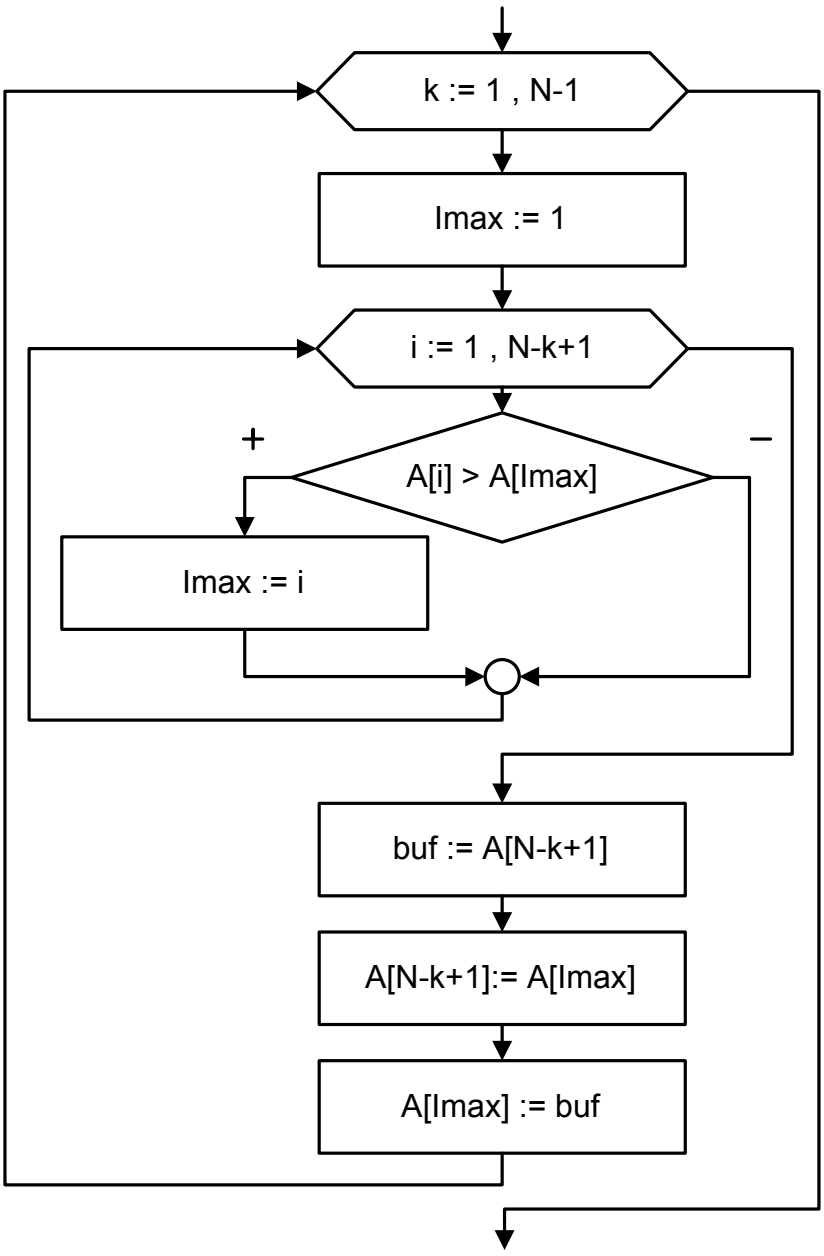


Рисунок 5.19 Сортировка методом линейного поиска

Описанный алгоритм представлен далее (Рисунок 5.19):

```
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 двигаясь по массиву мы выхватываем самый большой встретившийся элемент и далее двигаем его к кону 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.

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

```
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
```

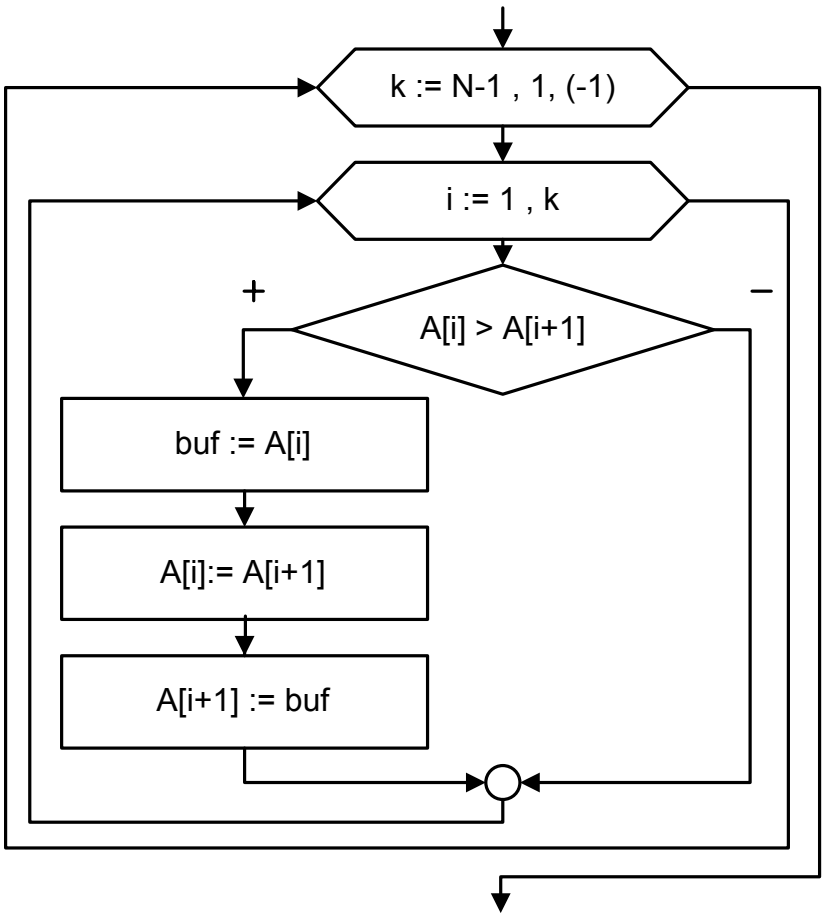


Рисунок 5.20 Сортировка пузырьковым методом

Данный алгоритм представлен ниже (Рисунок 5.20):

```
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 происходят просто так. 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 (Рисунок 5.21):

```
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

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

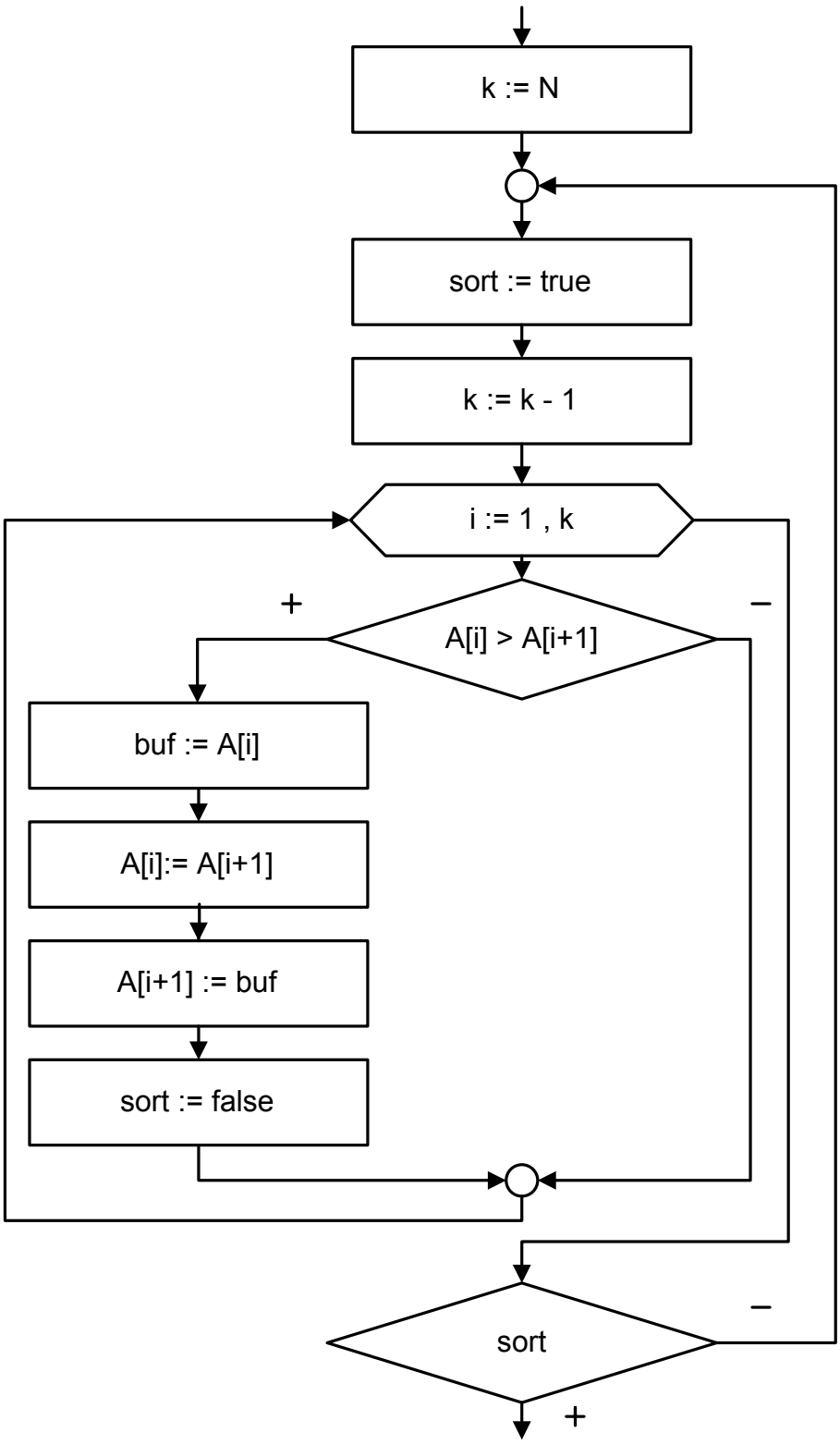


Рисунок 5.21 Усовершенствованный метод пузырька

Это сумма арифметической прогрессии, она равна

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

По данной формуле видно, что зависимость времени исполнения сортировки от размера массива квадратичная. Для этого мы ввели функцию $\Theta\left(N^2\right)$. Видно, что чем длиннее массив, тем больше времени требуется на его

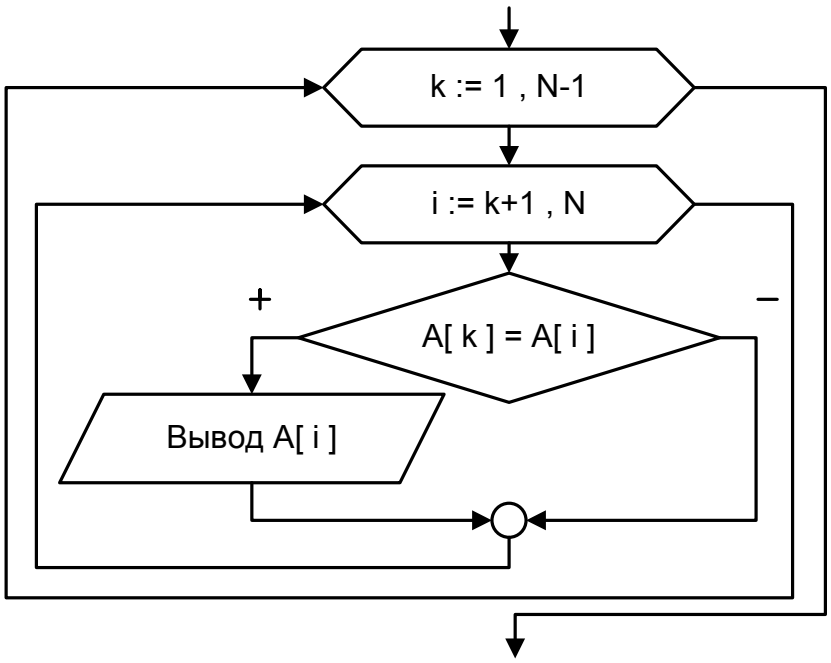


Рисунок 5.22 Поиск одинаковых элементов в одномерном массиве

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 мы в данной главе рассматривать не будем, поскольку они достаточно сложны для неподготовленного читателя. Следует отметить, что сложность $N\log_2(N)$ гораздо более приемлемая, чем N^2, поскольку, например если длина массива возрастает в 100 раз, то время простой сортировки увеличится в 10000 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 мы 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 (Рисунок 5.22):

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

5.6 Линейная алгебра и вектора

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 нам 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 меньше чем любая из его координат (можно привести теорему Пифагора, где гипотенуза всегда меньше любого из катетов).

В соответствии с формулой (4.1), имеем:

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

Это значит, что если нам 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

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

Векторное произведение

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

$$\vec{\mathbf{C}} = \vec{\mathbf{A}} \times \vec{\mathbf{B}} = \begin{vmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \vec{e}_1\left(a_2b_3 - a_3b_2\right) - \vec{e}_2\left(a_1b_3 - a_3b_1\right) + \vec{e}_3\left(a_1b_2 - a_2b_1\right) =$$

$$= \left\{\left(a_2b_3 - a_3b_2\right), -\left(a_1b_3 - a_3b_1\right), \left(a_1b_2 - a_2b_1\right)\right\}.$$

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];
```

6. ДВУМЕРНЫЕ МАССИВЫ

6.1 Понятие и объявление двумерного массива

Довольно часто при обработке больших объемов информации мы 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*.

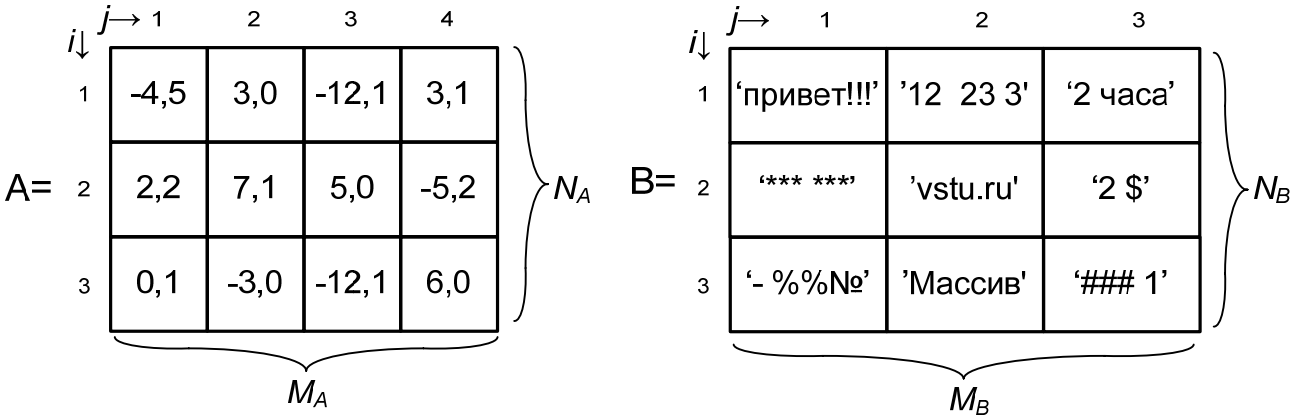


Рисунок 6.1 Примеры двумерных массивов

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 Рисунок 6.1.

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 мы 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 мы 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 именно мы организуем порядок их задания. Но далее всегда будем обозначать сначала строку, а затем столбец.

Для примеров на Рисунок 6.1 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 мы можем 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.

6.2 Поэлементная обработка двумерных массивов

Прямая безусловная поэлементная обработка двумерного массива предполагает такую обработку, при которой все его элементы безусловно

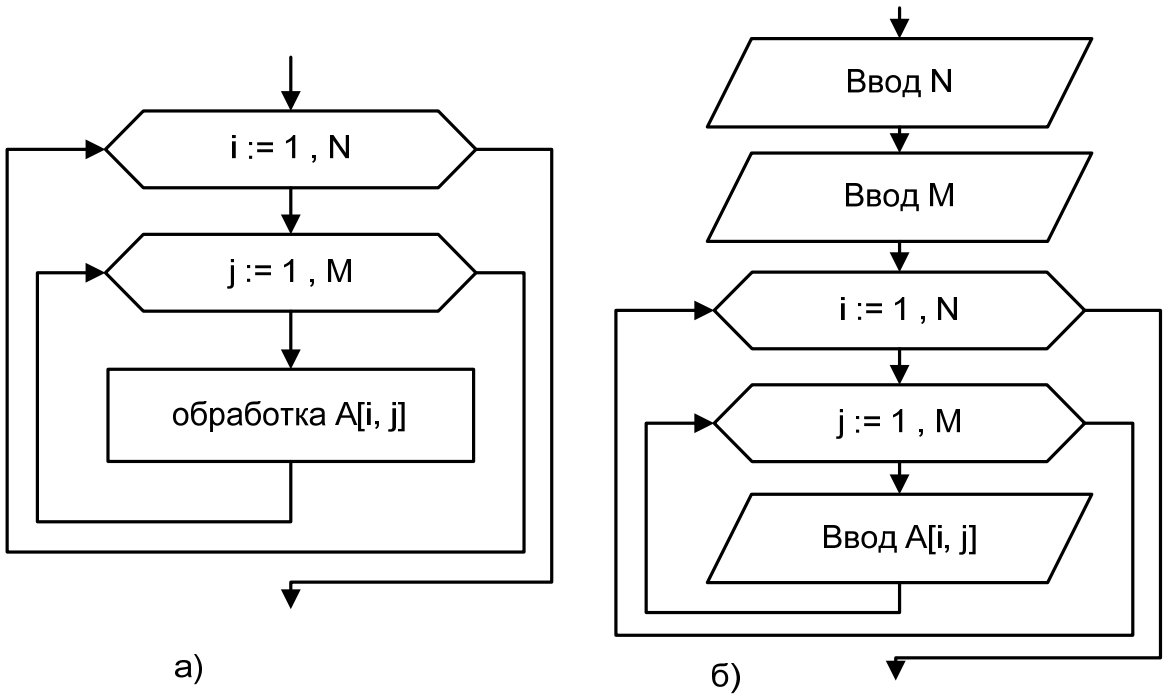


Рисунок 6.2 Поэлементная обработка двумерного массива а) и ввод массива б)

просматриваются в порядке возрастания индексов. Индексы можно увеличивать, рассматривая массив по строкам или по столбцам. Блок-схема этого процесса (построчная реализация) представлена на Рисунок 6.2 а).

Самые простые алгоритмы поэлементной обработки массива – это алгоритмы *ввода* и *вывода*. Алгоритм ввода или *инициализация* пользователем представлен на Рисунок 6.2 б). Здесь, поскольку массивы мы объявляем с запасом размерности, следует сначала указать число строк ***N*** и количество столбцов ***M***. Массив мы 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;
```

Вывод массива аналогичен вводу, только если мы будем выводить все элементы подряд, то разные строки сольются между собой. Мы не сможем 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 пустой процедуры ***writeLn***:

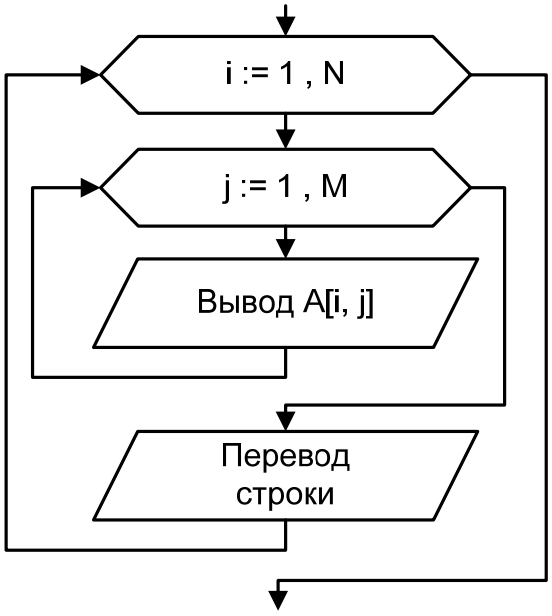


Рисунок 6.3 Вывод двумерного массива

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

На блок-схеме Рисунок 6.3 показана операция принудительного перевода курсора на следующую строку. В дальнейшем при оформлении блок-схем мы 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***:

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

Алгоритм прост, блок-схему приводить не будем.

Далее можно рассмотреть поэлементную обработку всего массива предполагающую анализ элементов. Таковая обработка выражается блок-схемой Рисунок 6.3. В теле внутреннего цикла помещено условие, в случае выполнения которого, происходит действие над элементом ***A[i,j]***.

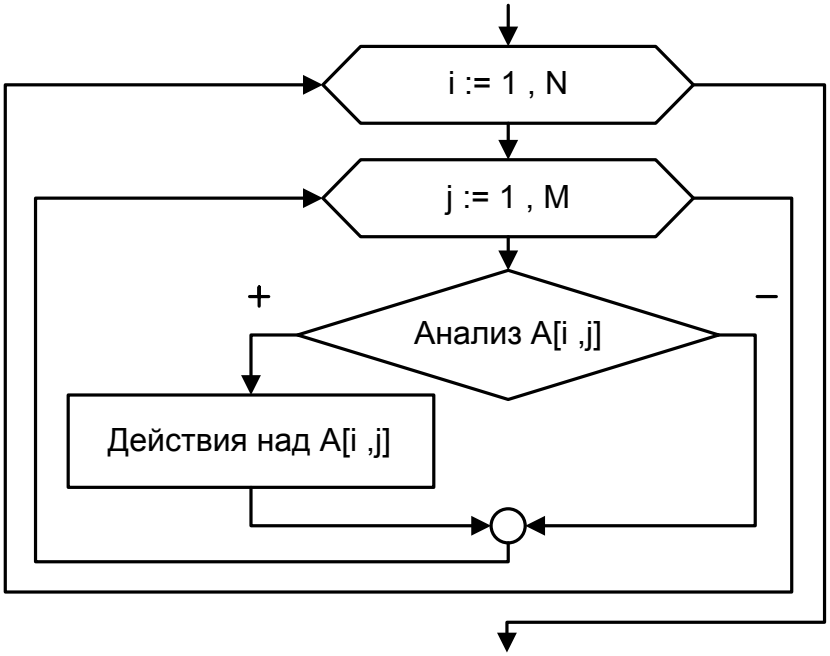


Рисунок 6.4 Алгоритм анализа элементов массива

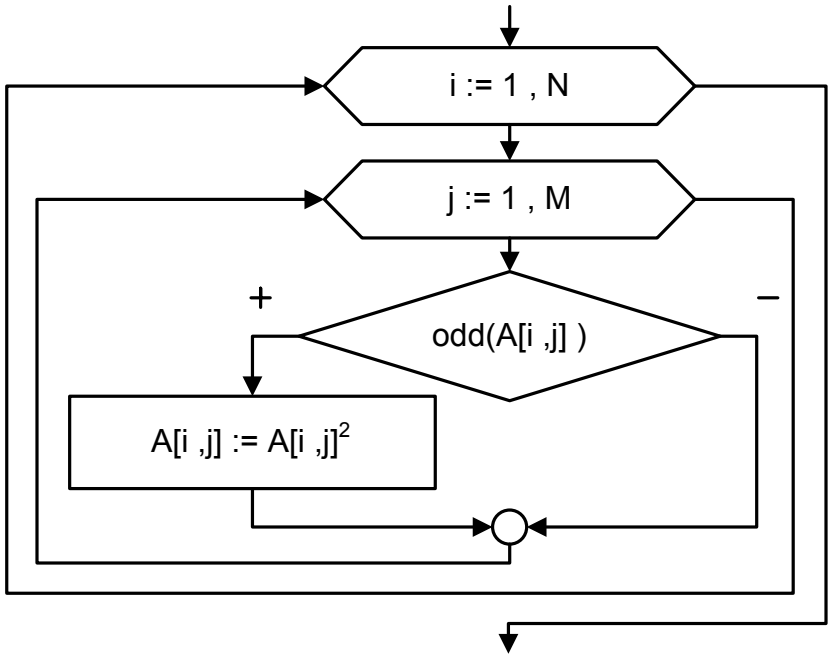


Рисунок 6.5 Возведение в квадрат нечетных элементов

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

ПРИМЕР

Возвести в квадрат все нечетные элементы двумерного массива ***A***. Решении таково (Рисунок 6.5):

```
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]);
```

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, как это принято, 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 (Рисунок 6.6).

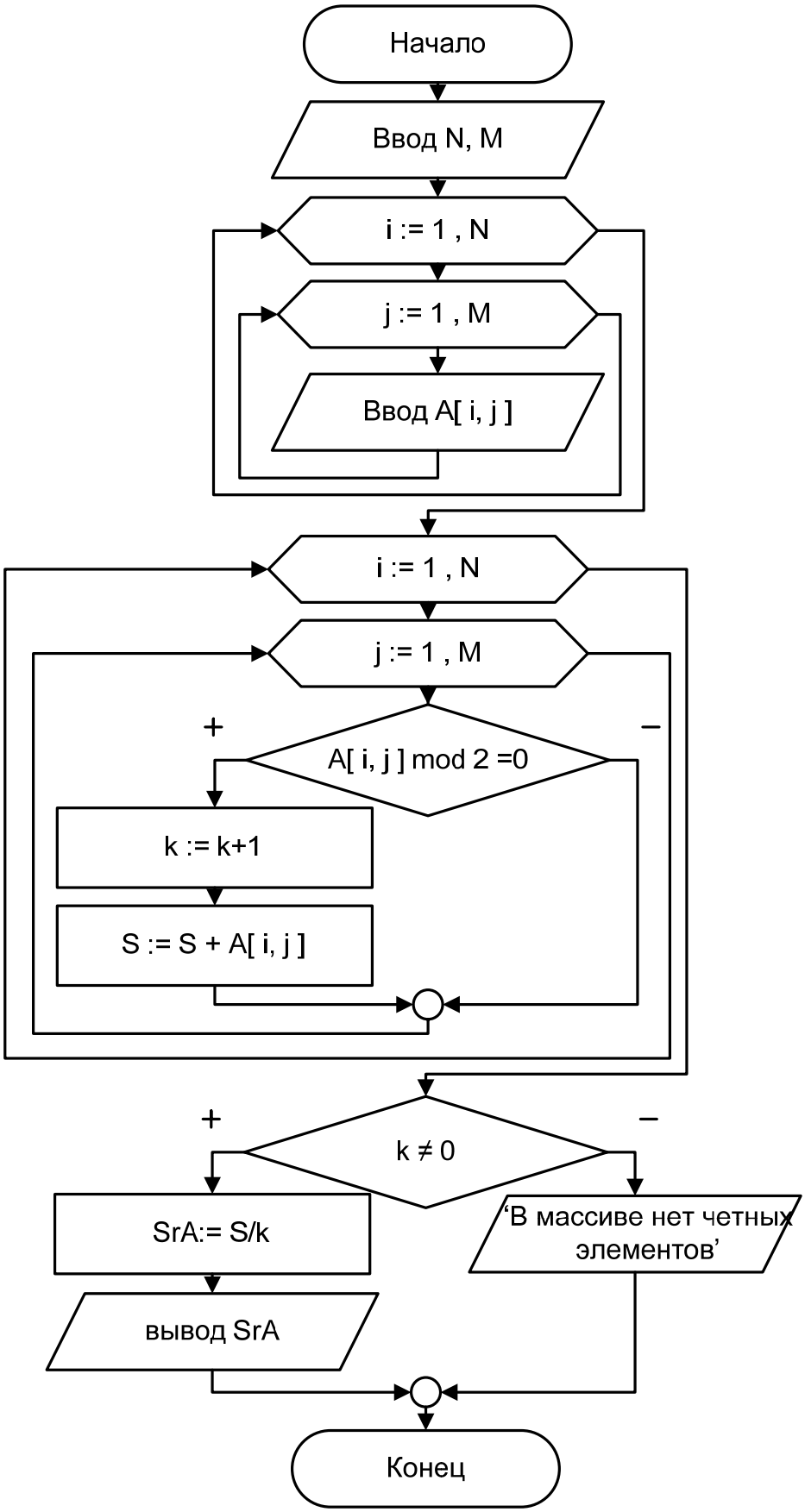


Рисунок 6.6 Среднее арифметическое четных элементов двумерного массива

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, нам необходима пара индексов для выяснения точного местоположения элемента с максимальным значением. За координаты отвечают переменные ***IMax***, ***JMax***, а за сам максимум переменная ***Max*** (Рисунок 6.8):

```
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 нам потребуется изменить знак в условии с «>» на «<». Да и имена переменных в которых будут храниться искомые значения следует заменить на ***IMin***, ***JMin*** и ***Min***.

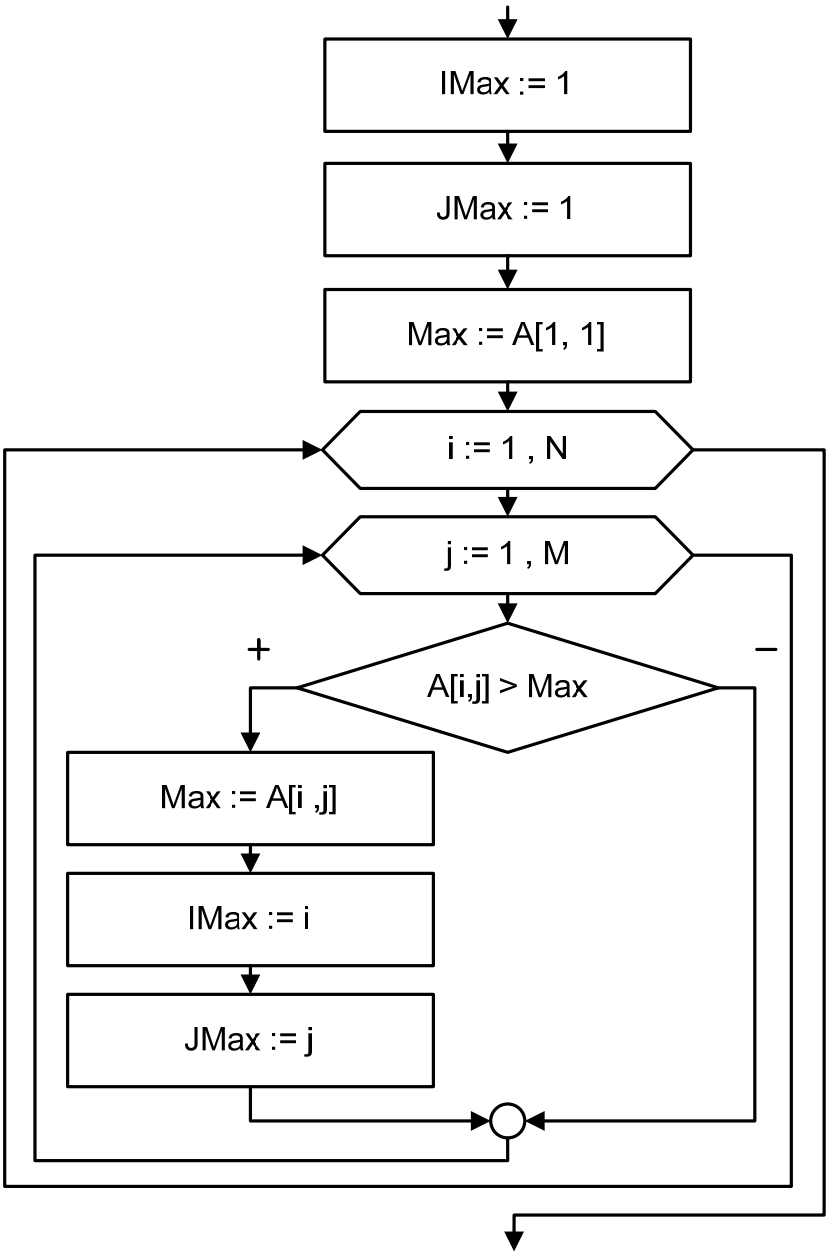


Рисунок 6.7 Поиск максимума

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, 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*** (Рисунок 6.8):

```
IMin := 1;
JMin := 1;
while (not odd(A[IMin,JMin])) and (IMin<N) do
  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<=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 всё аналогично, за исключением добавления цикла по строкам и включения как индексов строк, так и столбцов (алгоритм рассматривать не будем).

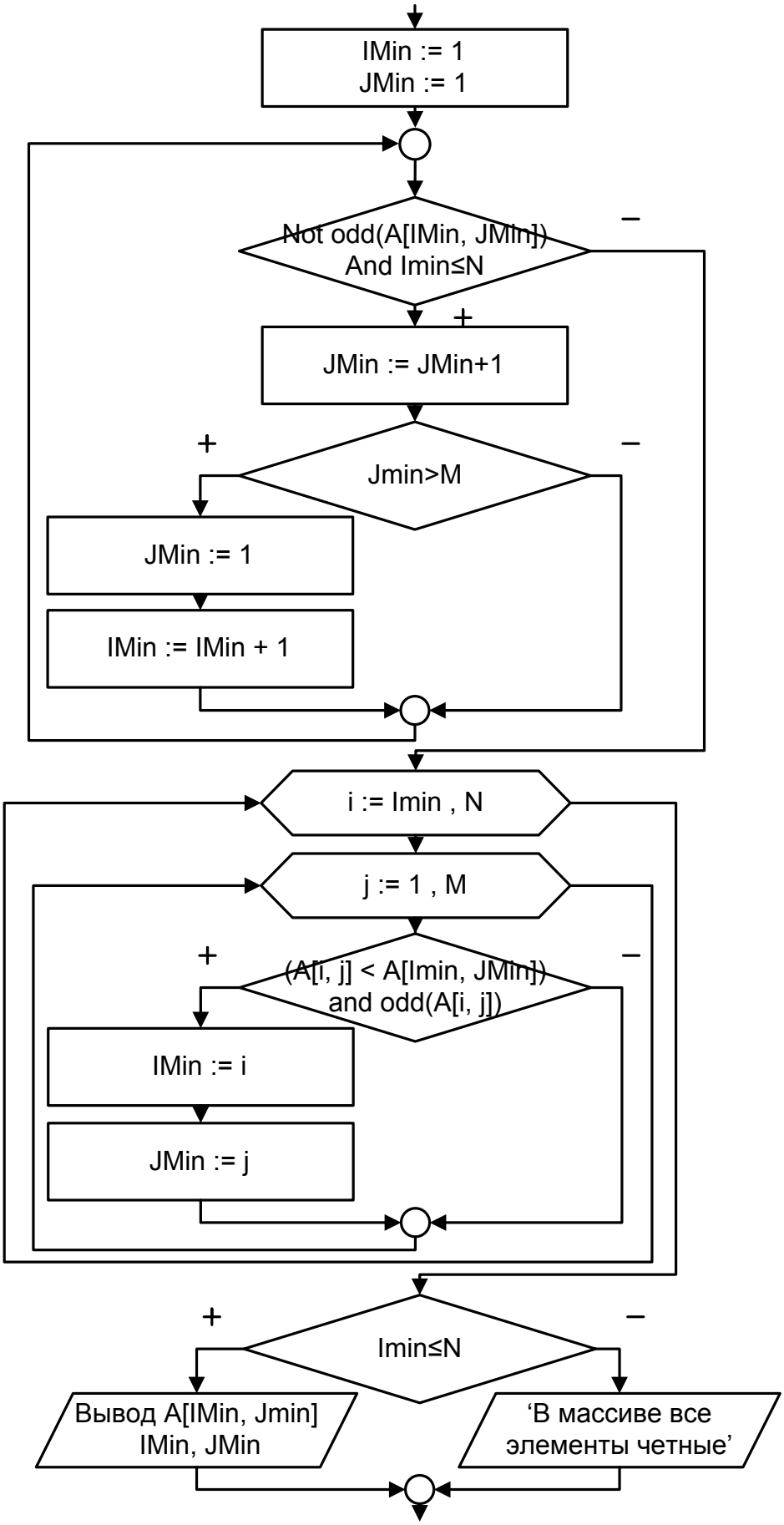


Рисунок 6.8 Поиск минимального среди нечетных

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 (Рисунок 6.9).

```
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.
```

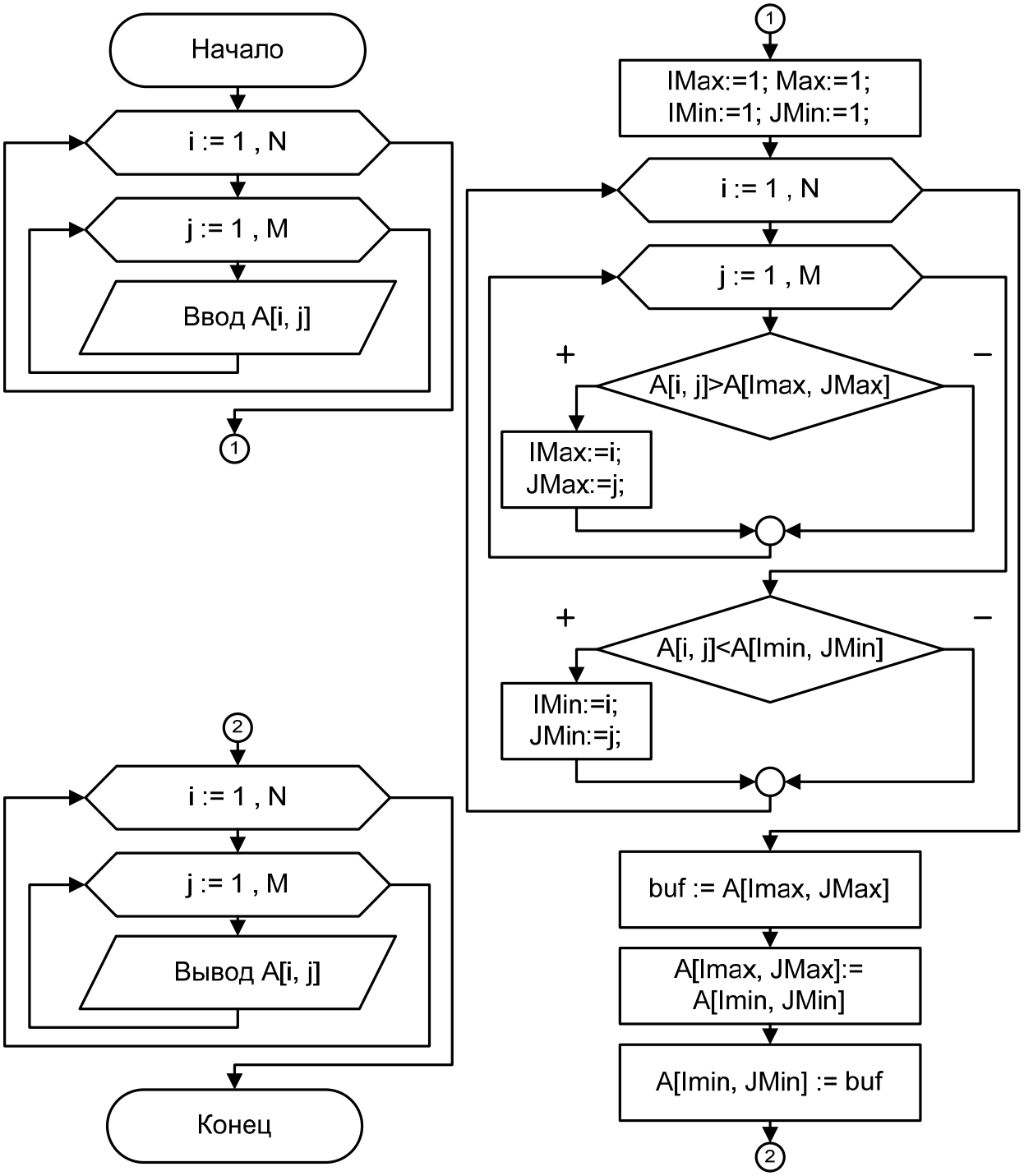


Рисунок 6.9 Обмен максимума и минимума в двумерном массиве

ПРИМЕР

Далее можно рассмотреть задачу формирования из заданного массива нового 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.*

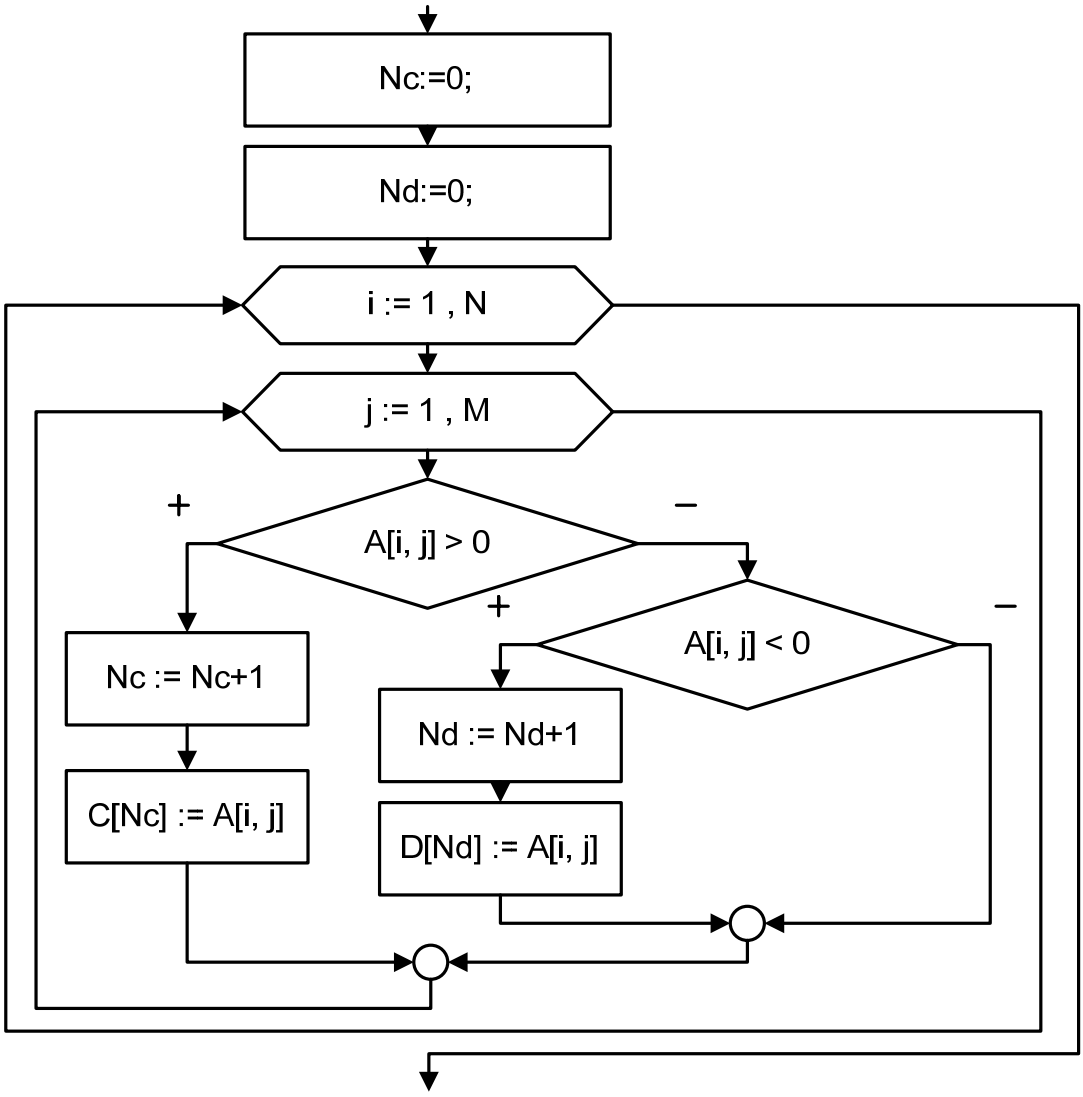


Рисунок 6.10 Формирование из двумерного массива пары одномерных

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 (Рисунок 6.10):

```
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;
```

6.3 Обработка отдельных строк или столбцов матрицы

Важным классом алгоритмов двумерных массив 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 Рисунок 6.11. 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.

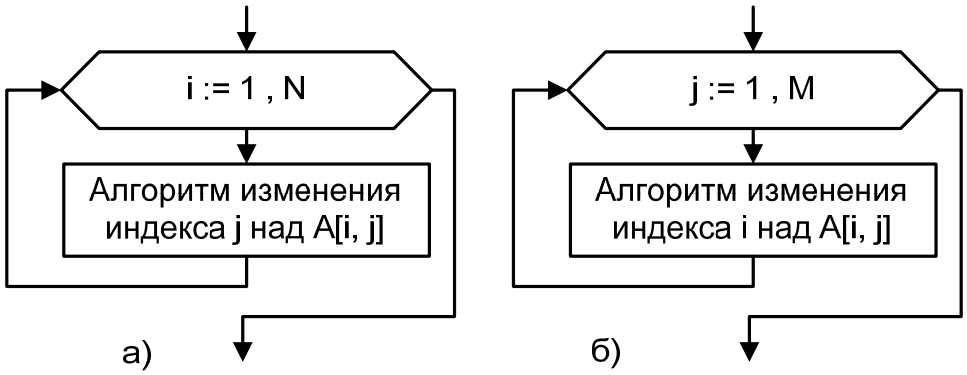


Рисунок 6.11 Построчная – а) и постолбцовая – б) обработка двумерного массива

В качестве примера можно решить, скажем, такую задачу:

ПРИМЕР

Найти сумму положительных элементов в каждой строке матрицы.

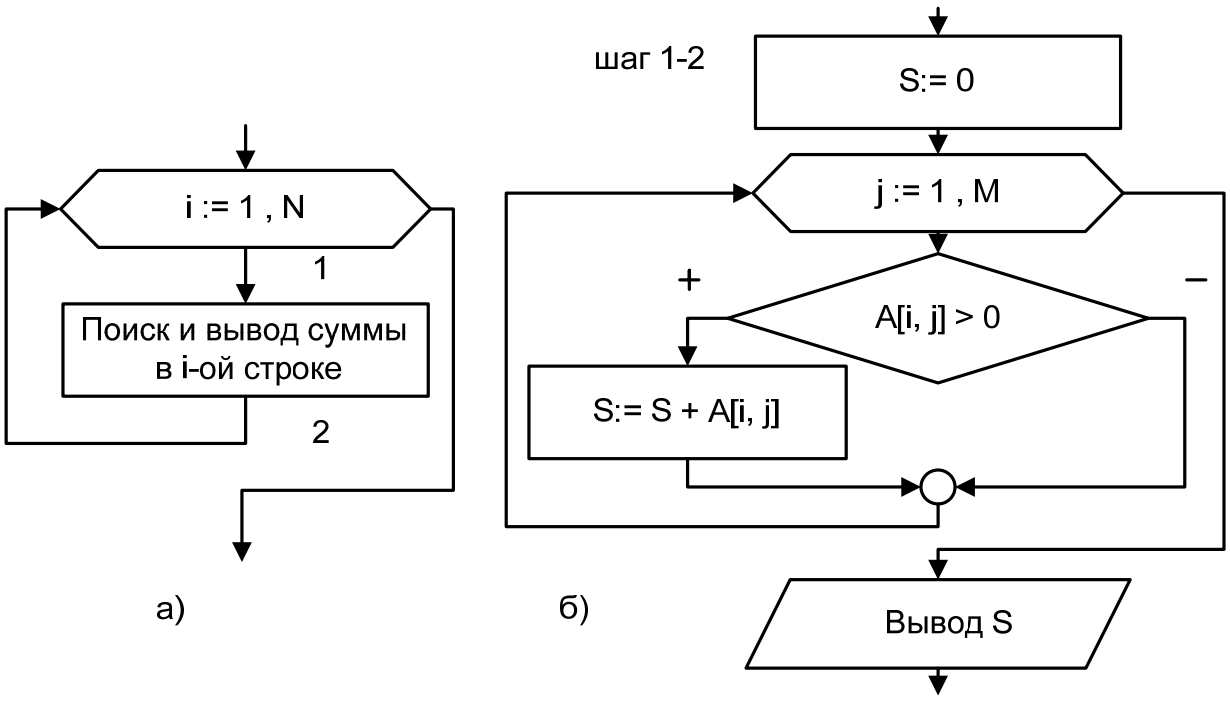


Рисунок 6.12 Поиск суммы положительных элементов в каждой строке

Алгоритм решения состоит в следующем. Во внешнем цикле меняем индекс строки (Рисунок 6.12 а)), а во внутреннем решаем задачу поиска суммы элементов одномерного массива с последующим выводом результатов на экран (Рисунок 6.12 б)). Индекс строки ***i*** фиксируем во внутреннем цикле. Поиск суммы обычный, по рекуррентной формуле: «сумма текущего равна сумме предыдущего 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:

вход: $A=$

–1	–4	–8	0	–2
1	0	4	–5	3
2	–7	–1	0	8

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. Пусть задача будет формулируется следующим образом:

ПРИМЕР

*Переписать максимальные элементы каждой строки 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 j:=1 to N do
      if A[i,j]>max then
        max:=A[i,j];
    B[j]:=max;
  end;
```

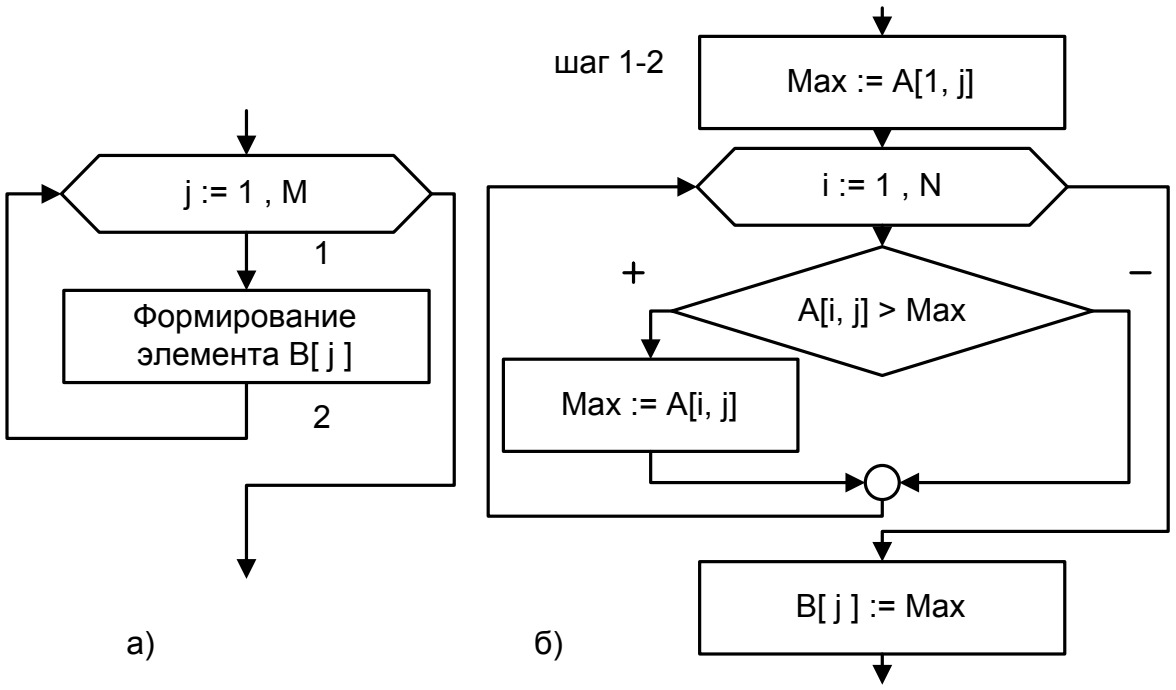


Рисунок 6.13 Поиск максимума в каждом столбце

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 (Рисунок 6.14 а)), а во внутреннем – происходит сортировка строки по индексу столбца ***j***. В качестве метода сортировки возьмем пузырьковый. Видно, что алгоритм практически точно повторяет тот, который мы использовали для одномерного массива (Рисунок 6.14 б)). Стоит обратить внимание на тот факт, что при решении этой задачи у нас 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;
```

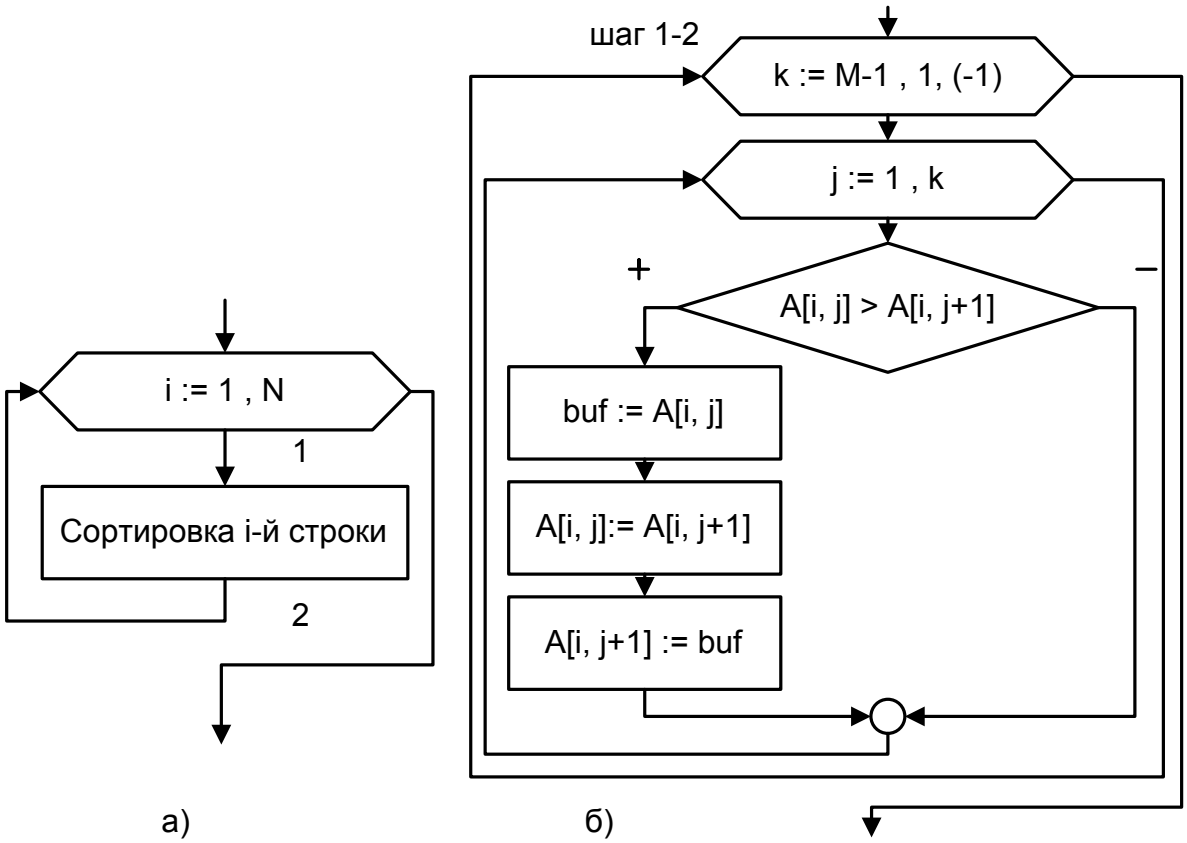


Рисунок 6.14 сортировка каждой строки матрицы

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 (Рисунок 6.15 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 у нас 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 (Рисунок 6.15 б)):

```
j:=2;
```

```
while j<=M do
  begin
    for i:=1 to N do
      A[i,j]:=1;
    j:=j+2;
  end;
```

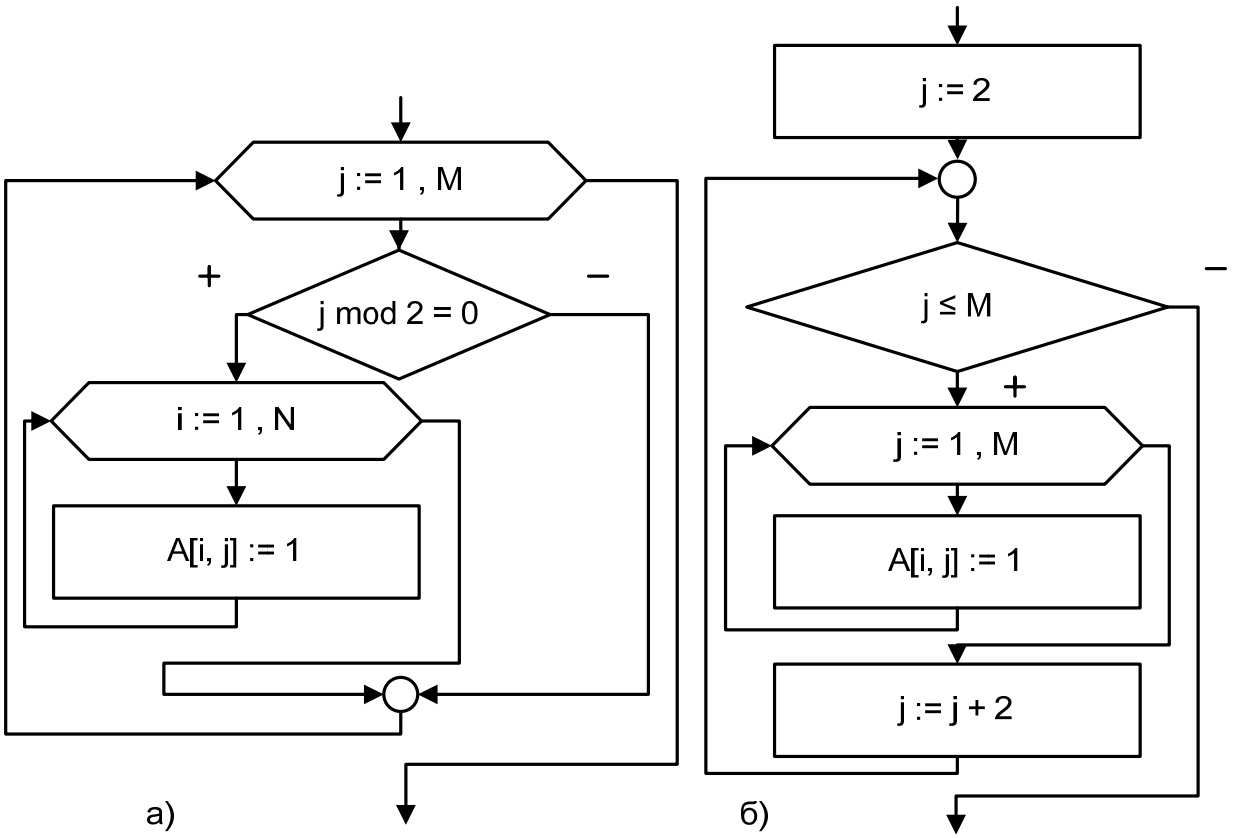


Рисунок 6.15 Обработка четных столбцов при помощи цикла for – а) и при помощи цикла while –б)

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 строки 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.

Алгоритм решения при помощи цикла ***for*** следующий (**Ошибка! Источник ссылки не найден.** 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 (**Ошибка! Источник ссылки не найден.** 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;
```

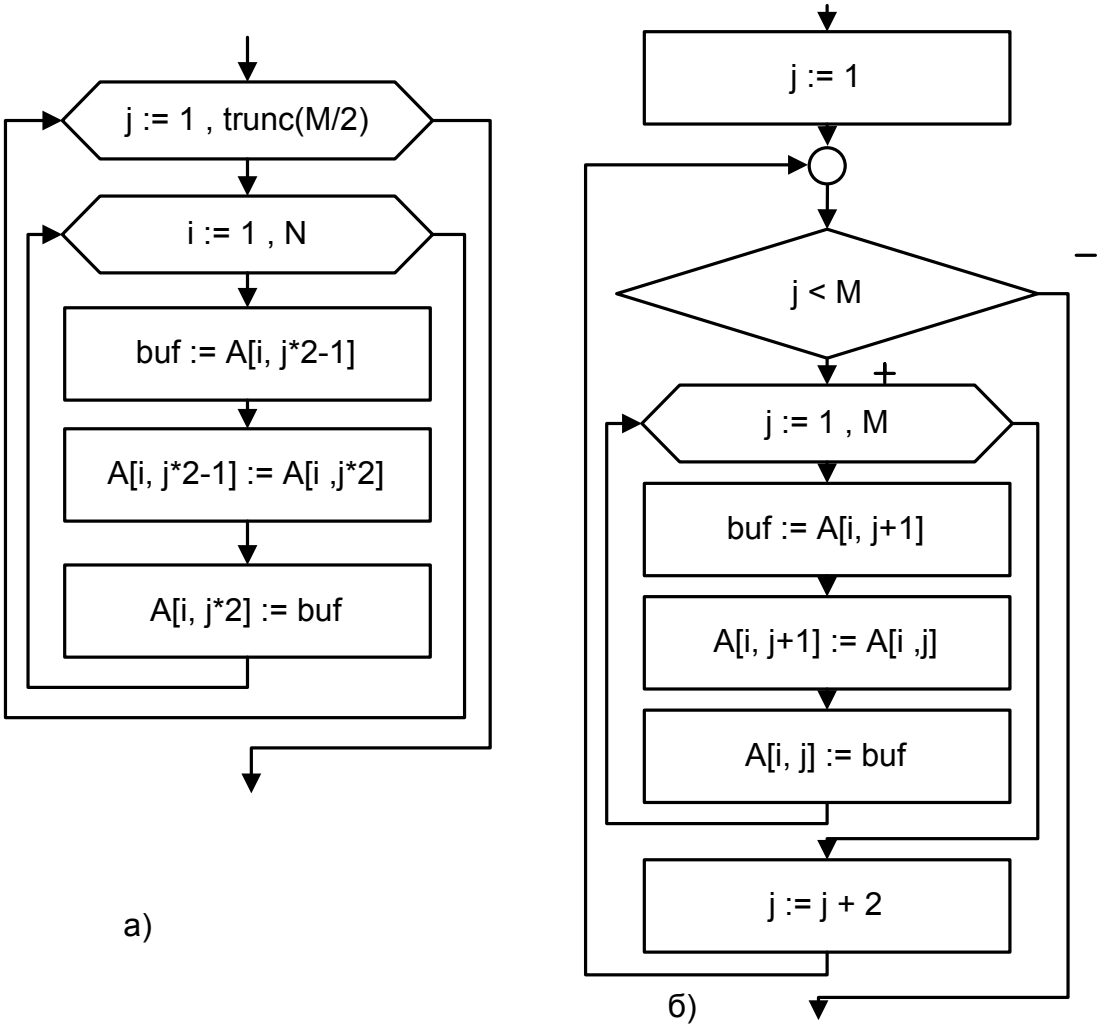


Рисунок 6.16 Перестановка столбцов в при помощи цикла for –а) и при помощи цикла while –б)

6.4 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.

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 (Рисунок 6.17), то можно вывести правило принадлежности элемента к побочной диагонали. Это правило можно выразить формулой для индексов ***j=N-i+1***.

Для матрицы изображенной на (Рисунок 6.17) 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.

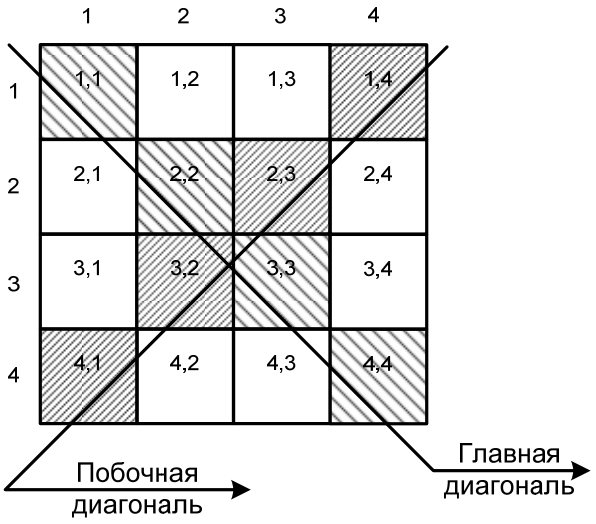


Рисунок 6.17 Диагонали в матрице

ПРИМЕР

Найти среднее арифметическое отрицательных элементов главной диагонали.

Решение таково (Рисунок 6.18):

```
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');
```

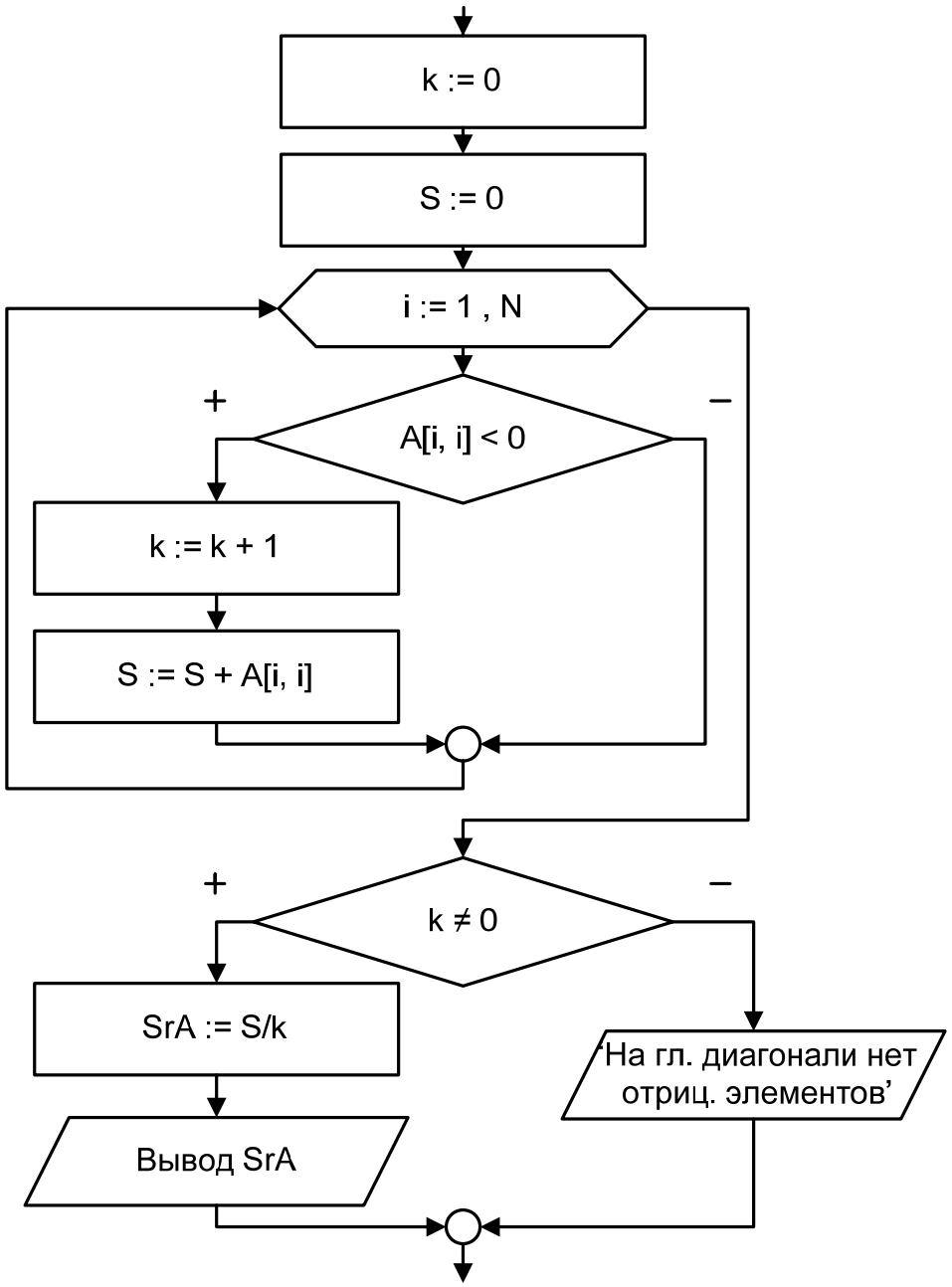


Рисунок 6.18 Среднее арифметическое отрицательных элементов гл. диагонали

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 (Рисунок 6.19):

```
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;
```

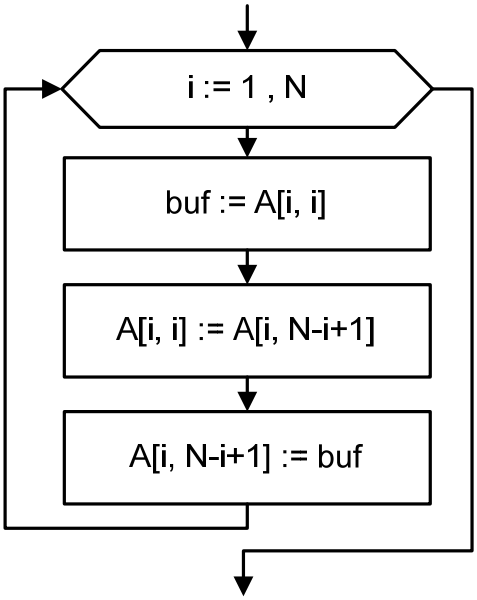


Рисунок 6.19 Обмен диагоналей

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 **Ошибка! Источник ссылки не найден.**.

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***.

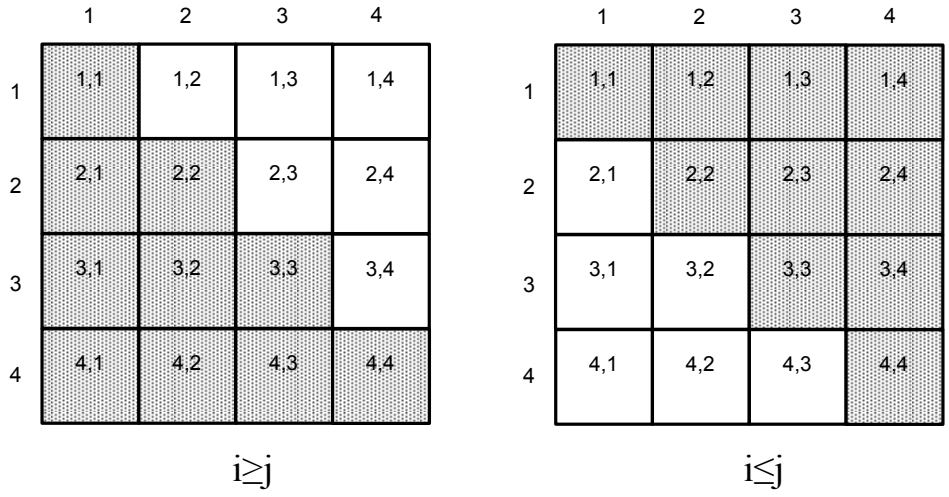


Рисунок 6.20 Верхний и нижний треугольники в матрице

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 есть риск ошибиться и тогда задача будет решена вообще неправильно.

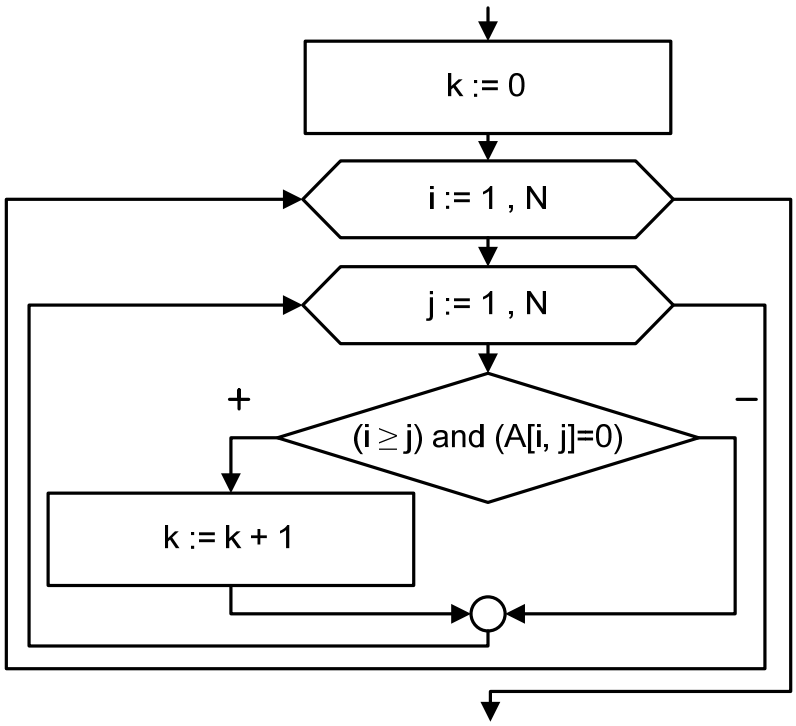


Рисунок 6.21 Обработка нижнего треугольника с условием на индексах

ПРИМЕР

Посчитать количество нулей в нижнем треугольнике матрицы.

Способ с анализом условия на индексы (Рисунок 6.21):

```
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;
```

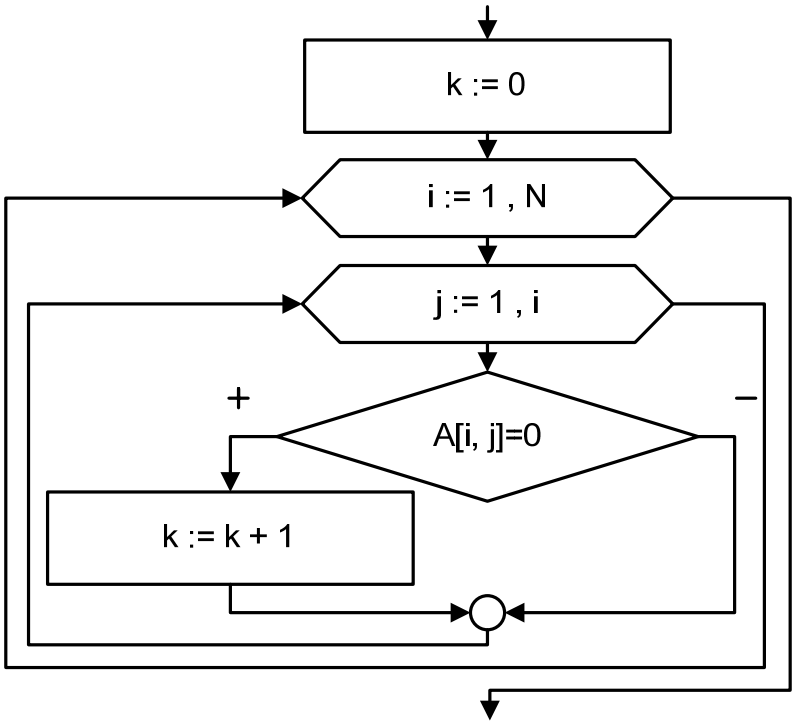


Рисунок 6.22 Обработка нижнего треугольника с изменением границ циклов

Способ с изменением границ циклов (Рисунок 6.22):

```
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: $A=$

4	2	8	9
3	0	0	1
6	7	8	0
5	3	2	0

выход: $A=$

4	2	8	9
3	6	6	1
6	7	8	6
5	3	2	6

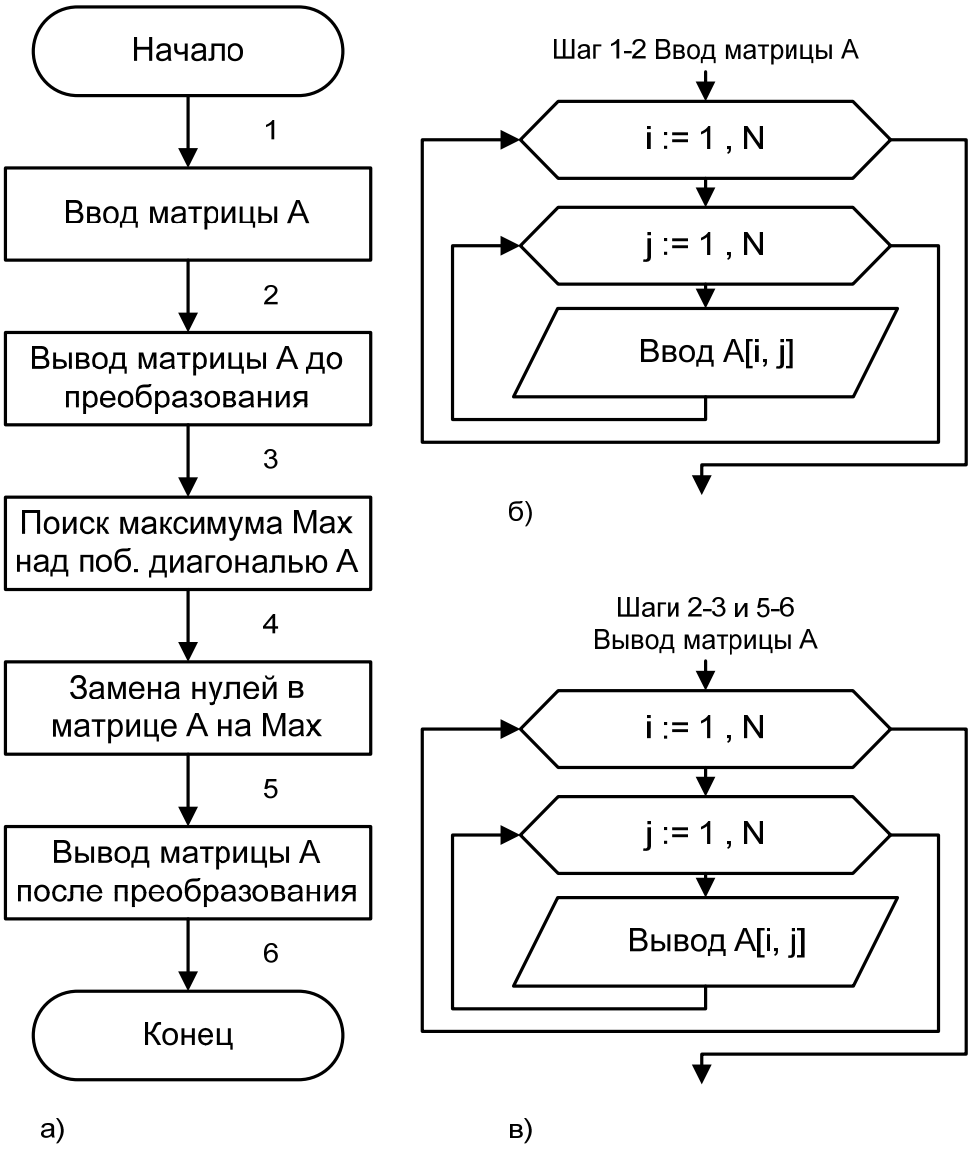


Рисунок 6.23 Общая блок-схема с пошаговой детализацией отдельных действий

```
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.
```

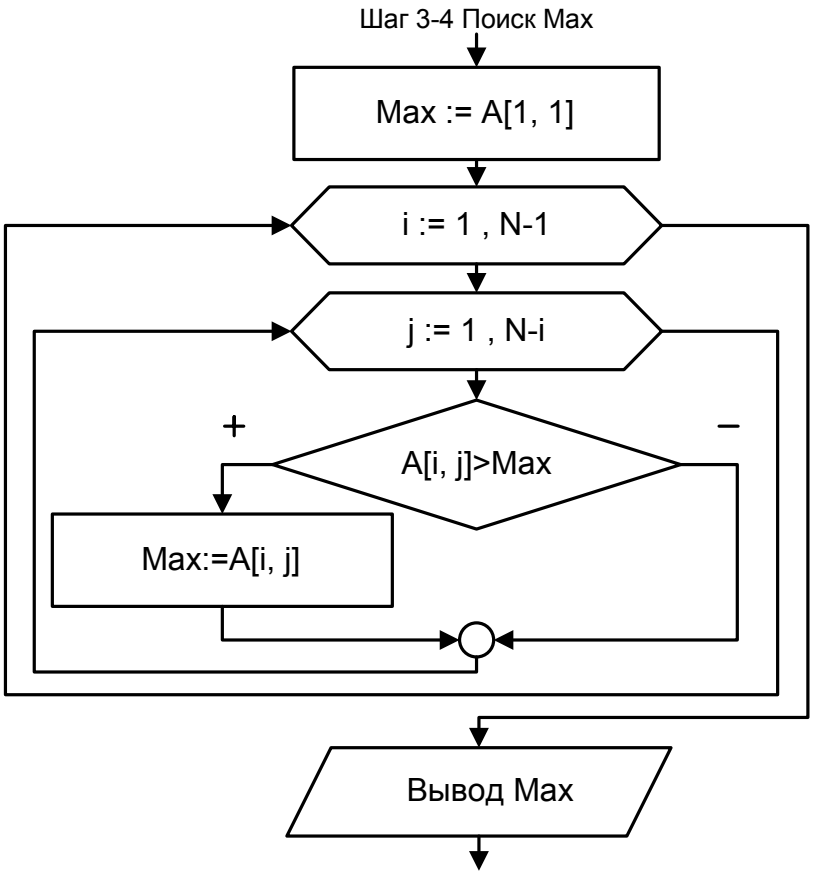


Рисунок 6.24 Поиск максимума над побочной диагональю

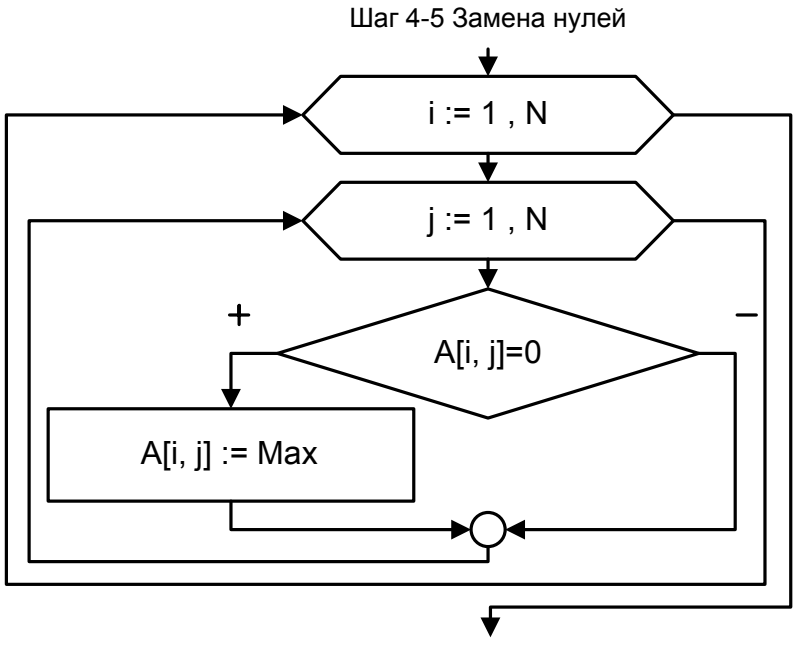


Рисунок 6.25 Замена нулей найденным максимумом

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.

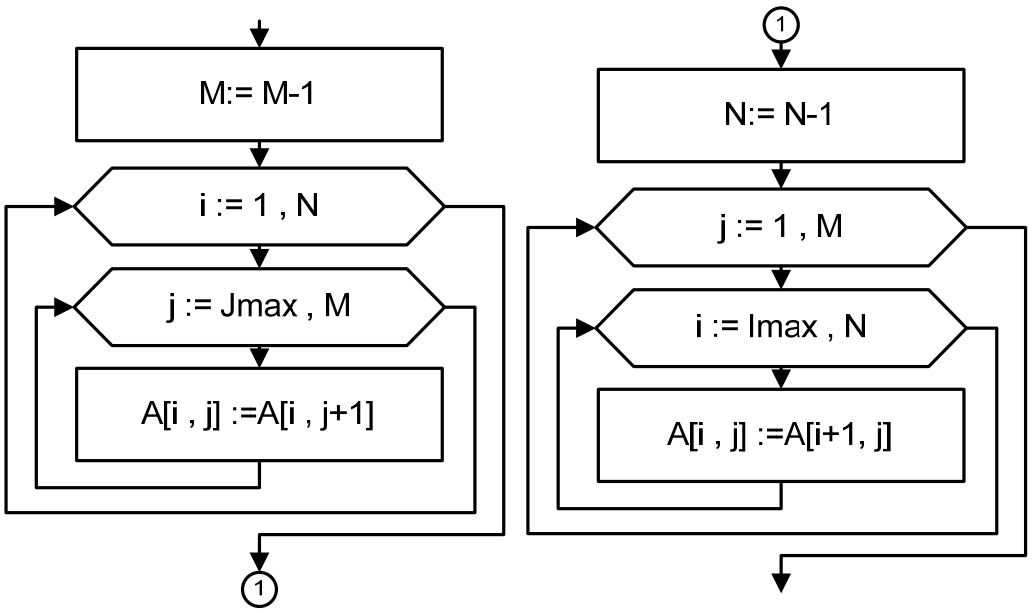


Рисунок 6.26 Удаление строки и столбца из матрицы

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***=3; ***Jmax***=4; *выход*:

1	2	3	5
6	7	8	9
15	16	17	19

Блок-схема алгоритма без программной реализации показана на Рисунок 6.26. Здесь не приводится стандартный поиск максимума. Считается, что координаты максимума ***Imax*** и ***Jmax*** были найдены ранее.

6.5 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.:

$$\mathbf{C}=\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix}+\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1M} \\ b_{21} & b_{22} & \cdots & b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N1} & \cdots & b_{NM} \end{bmatrix}=$$

$$=\begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1M}+b_{1M} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2M}+b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1}+b_{N1} & a_{N1}+b_{N2} & \cdots & a_{NM}+b_{NM} \end{bmatrix}=\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1M} \\ c_{21} & c_{22} & \cdots & c_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ c_{N1} & c_{N1} & \cdots & c_{NM} \end{bmatrix}$$

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:

$$k\mathbf{A}=k\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix}=\begin{bmatrix} ka_{11} & ka_{12} & \cdots & ka_{1M} \\ ka_{21} & ka_{22} & \cdots & ka_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{N1} & ka_{N1} & \cdots & ka_{NM} \end{bmatrix},$$

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:

Ошибка! Объект не может быть создан из кодов полей редактирования.,

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: $\mathbf{A}^T = \begin{bmatrix} 1 & 0 \\ 3 & 4 \\ 8 & 7 \end{bmatrix}$.

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. **Ошибка! Объект не может быть создан из кодов полей редактирования.**. Перемножение матриц происходит по принципу «*строка на столбец*», для этого необходимо, чтобы число столбцов в первой матрице равнялось числу строк во второй:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1M} \\ b_{21} & b_{22} & \cdots & b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N1} & \cdots & b_{NM} \end{bmatrix} =$$

$$= \begin{bmatrix} \sum_{j=1}^{MaNb} a_{1j}b_{j1} & \sum_{j=1}^{MaNb} a_{1j}b_{j2} & \cdots & \sum_{j=1}^{MaNb} a_{1j}b_{jMb} \\ \sum_{j=1}^{MaNb} a_{2j}b_{j1} & \sum_{j=1}^{MaNb} a_{2j}b_{j2} & \cdots & \sum_{j=1}^{MaNb} a_{2j}b_{jMb} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{MaNb} a_{Naj}b_{j1} & \sum_{j=1}^{MaNb} a_{Naj}b_{j2} & \cdots & \sum_{j=1}^{MaNb} a_{Naj}b_{jMb} \end{bmatrix}$$

частный случай перемножения:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 6 & 7 & 1 & 1 \end{bmatrix} \quad \mathbf{C} = \mathbf{AB} = \begin{bmatrix} 12 & 15 & 3 & 3 \\ 24 & 30 & 6 & 6 \end{bmatrix}$$

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;
```

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.

$$\det\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N1} & \cdots & a_{NM} \end{bmatrix} =$$

$$= \det\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} - a_{11}\dfrac{a_{21}}{a_{11}} & a_{22} - a_{12}\dfrac{a_{21}}{a_{11}} & \cdots & a_{2M} - a_{1M}\dfrac{a_{21}}{a_{11}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} - a_{11}\dfrac{a_{N1}}{a_{11}} & a_{N1} - a_{12}\dfrac{a_{N1}}{a_{11}} & \cdots & a_{NM} - a_{1M}\dfrac{a_{N1}}{a_{11}} \end{bmatrix} =$$

$$= \det\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ 0 & a_{22} - a_{12}\dfrac{a_{21}}{a_{11}} & \cdots & a_{2M} - a_{1M}\dfrac{a_{21}}{a_{11}} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{N1} - a_{12}\dfrac{a_{N1}}{a_{11}} & \cdots & a_{NM} - a_{1M}\dfrac{a_{N1}}{a_{11}} \end{bmatrix} =$$

и так далее, пока не будет получена верхняя треугольная матрица.

Представленный алгоритм подразумевает, что на главной диагонали элементы не равны нулю. Если же мы 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];
```

7. ПОДПРОГРАММЫ

7.1 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. (Рисунок 7.1). Одна из самых жестких систем такого типа – это система военной субординации. Важным свойством систем подобных изображенной на (Рисунок 7.1) 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 мы помним, тоже является средством управления. Только здесь в качестве подчиненных выступают не люди, а информация. Иерархия при программировании строится по схеме аналогичной Рисунок 7.1. Следует обратить внимание на отсутствие связей между блоками на одном иерархическом уровне.

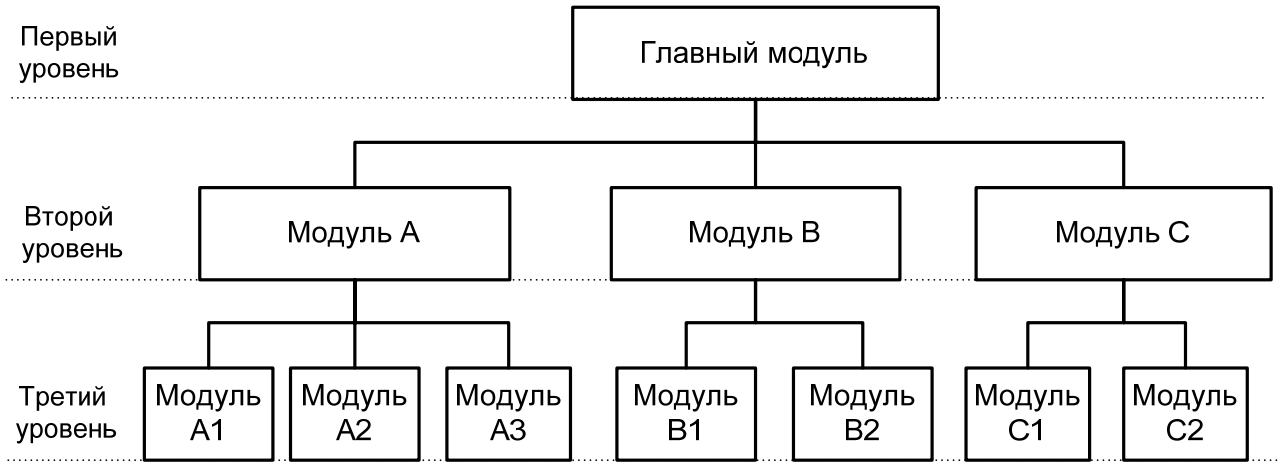


Рисунок 7.1 Иерархическая организация

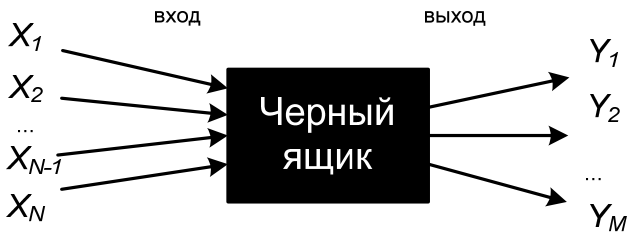


Рисунок 7.2 Черный ящик

Еще одним важным понятием на пути к определению подпрограмм является понятие *черного ящика*. ***Кибернетический черный ящик*** – такое 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 Рисунок 7.1, 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.

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[1]. 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 Рисунок 7.2. Множество X называют входом черного ящика, а множество Y его выходом. Две системы называют *согласованными по интерфейсу*, если выходы первой можно совместить с 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.

[1] 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.

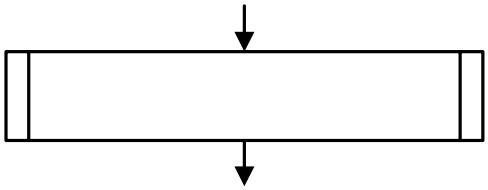


Рисунок 7.3 Графическое обозначение подпрограммы

Подпрограмма описывается один раз и может быть затем неоднократно *вызвана* в разделе операторов программы или другой 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[1]. Суть этой разницы сводится к различию методов работы с ними.

Использование подпрограмм позволяет вывести процесс программирования на качественно иной уровень. Т.к. 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.

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 Рисунок 7.3.

7.2 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;

[1] 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* существует возможность настройки компилятора таким образом, чтобы не существовало различий между процедурами и функциями.

- описание констант;
- описание типов;
- описание переменных;
- внутренние блоки.

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 микропроцессоров 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 нам 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:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \ldots$$

мы бы могли написать функцию:

```
function sin(x:real):real;
var xx,S:real;
    i:byte;
    f:longInt;
begin
  S:=0;
  f:=1;
  xx:=x;
```

```
    i:=1;
    while i<20 do
    begin
      S:=S+xx;  {добавление в сумму неч. члена с плюсом}
      i:=i+1;   {четное значение}
      xx:=xx*x; {четная степень}
      f:=f*i;   {четный факториал}
      i:=i+1;   {нечетное значение}
      xx:=xx*x; {нечетная степень}
      f:=f*i;   {нечетный факториал}
      S:=S-xx;  {добавление в сумму неч. члена с минусом}
    end;
    sin:=S;
  end;
```

Здесь для простоты понимания принципа работы функции, мы ограничились первой двадцаткой членов ряда. 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 мы имя функции без параметров ставим 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[1]).

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 (Рисунок 7.4).

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 совсем подходящие под ситуацию). Далее подробно рассмотрим процедуры.

[1] Подробнее о способах передачи будет рассказано ниже

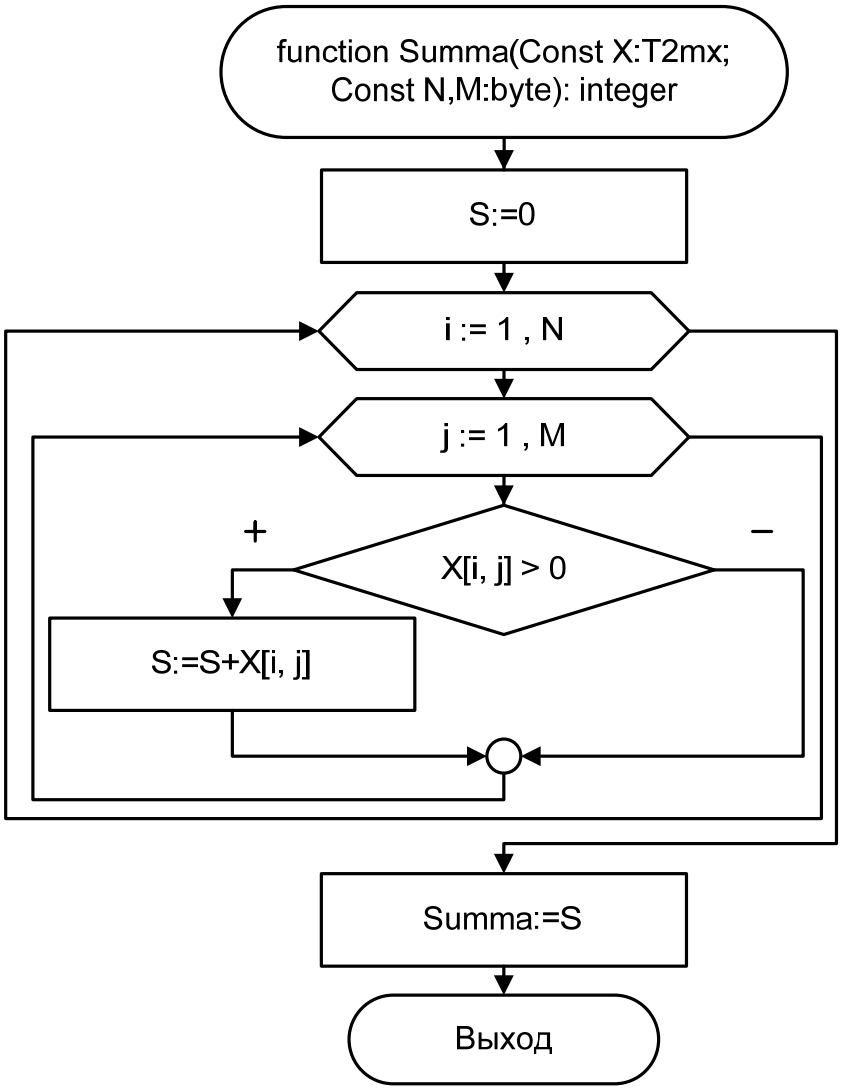


Рисунок 7.4 Функция суммы положительных элементов

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.

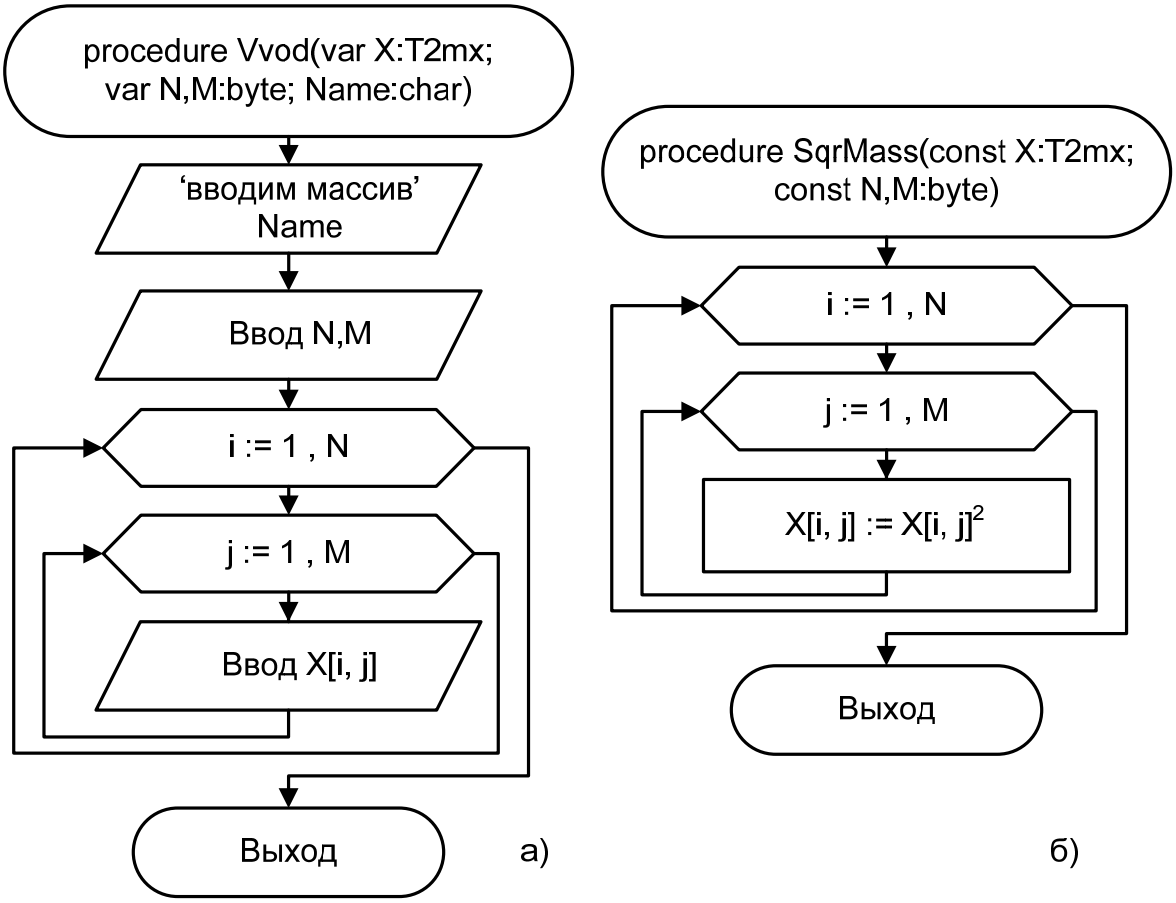


Рисунок 7.5 Ввод двумерного массива (а)) и возведение его элементов в квадрат (б))

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 (Рисунок 7.5 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 (Рисунок 7.5 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;
```

7.3 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[1] 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. Т.е. 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

[1] Здесь под переменной подразумеваются данные, которые могут храниться за любым идентификатором (константой, типом и т.д.), просто именно переменные являются основным объектом работы большинства алгоритмов.

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 ***program***), а далее через точку имя переменной (или иной идентификатор) к которому нужно обратиться. Для примера, изображенного на Рисунок 7.6 полные обращения к переменным такие:

G.X, ***G.Y***, ***G.A.A1.XA1***, ***G.C.YC*** и т.д.

На Рисунок 7.6 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. 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.

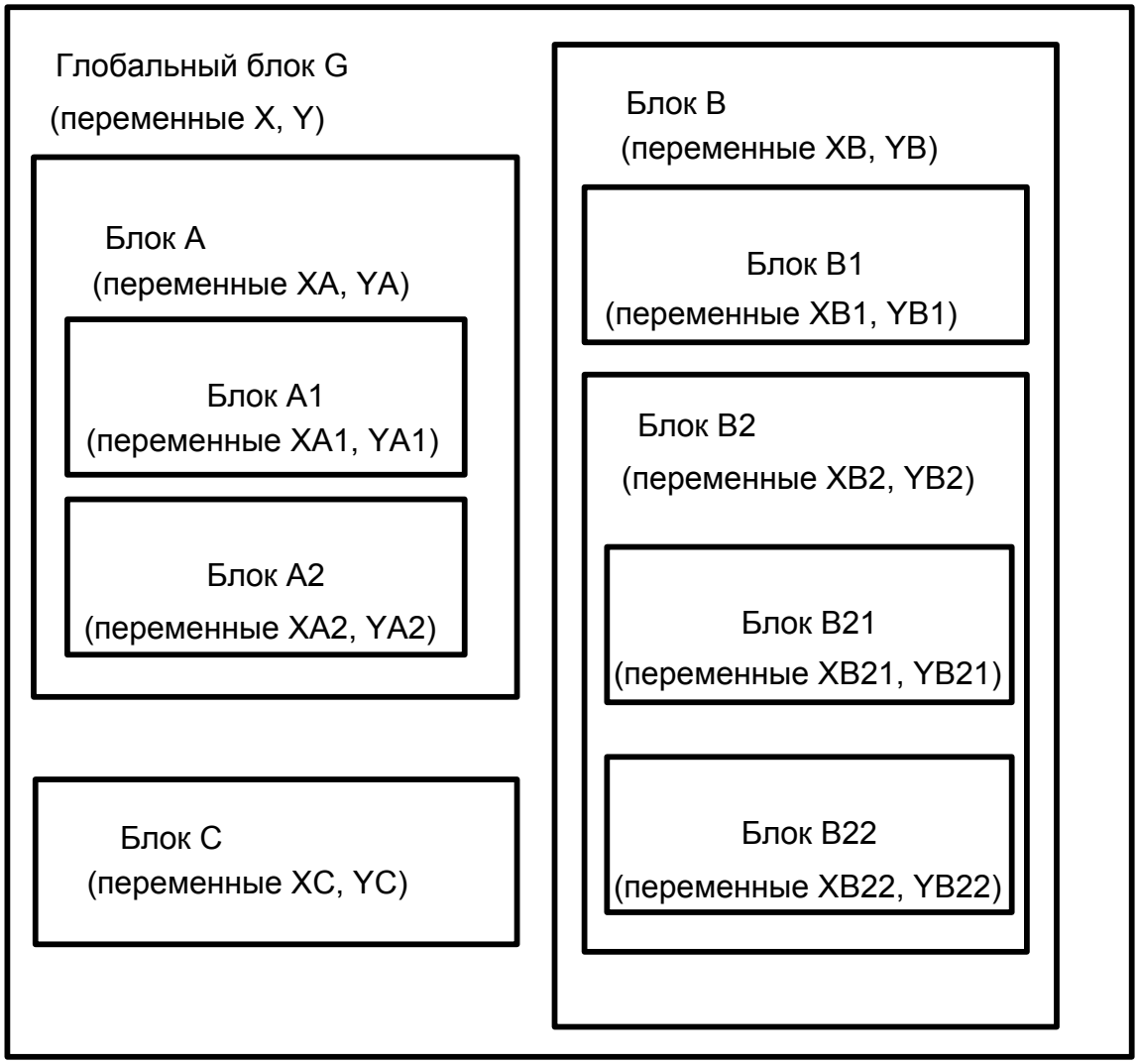


Рисунок 7.6 Пример иерархии подпрограмм и переменных

Глобальными можно назвать все те идентификаторы, которые используются в основной программе, в том числе и в скобках при вызове процедур.

7.4 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 заменяются 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 Рисунок 7.7.

По значению передаются параметры-значения, являющиеся простыми входными данными, т.е. константами, именами переменных и простыми выражениями. При этом значение передаваемого фактического параметра

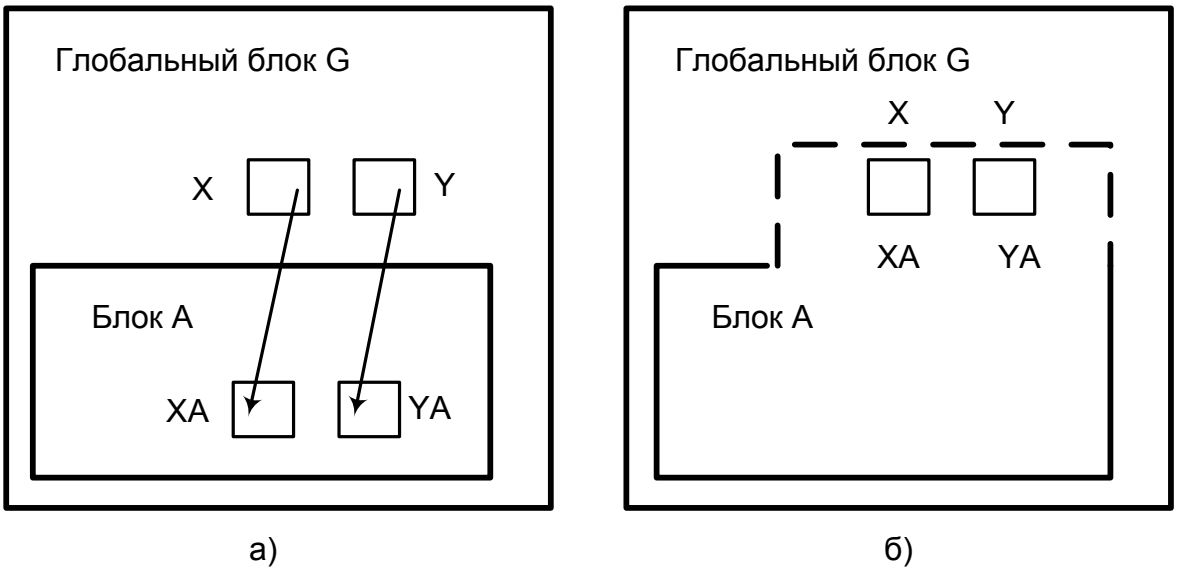


Рисунок 7.7 Передача параметров по значению а) и по ссылке б)

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 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***.

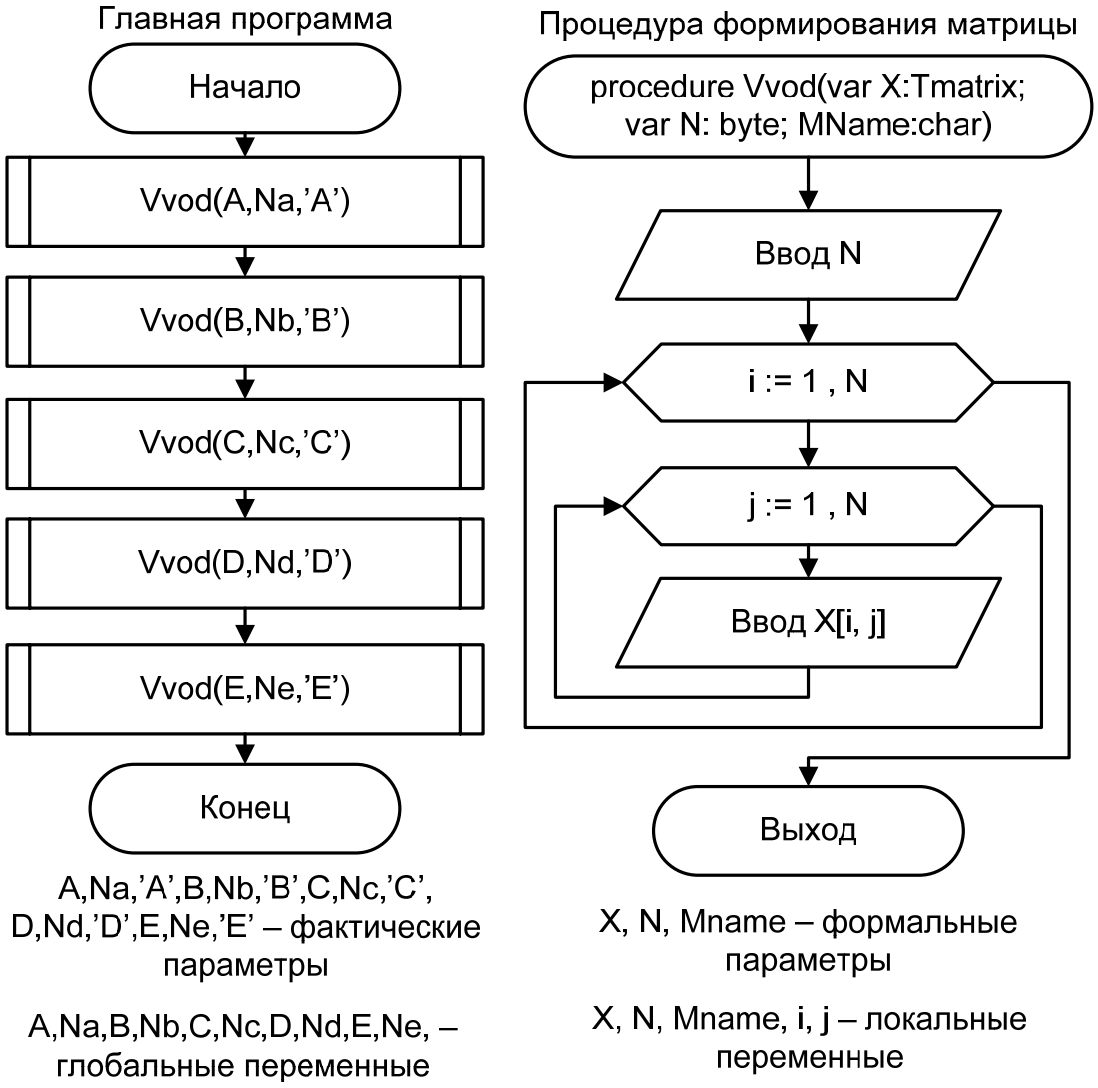


Рисунок 7.8 Ввод пяти разных матриц при помощи одной процедуры

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 (Рисунок 7.8).

```
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.
```

7.5 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:

Вход: $A=\begin{pmatrix}1&-3&5&3\\8&1&0&-2\\-8&11&2&-10\end{pmatrix}$, $B=\begin{pmatrix}4&-2&4\\5&7&0\\-1&2&4\\3&9&10\end{pmatrix}$, $C=\begin{pmatrix}2&5\\1&4\end{pmatrix}$

$MaxA=11$ $MaxB=10$ $MaxC=5$

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.

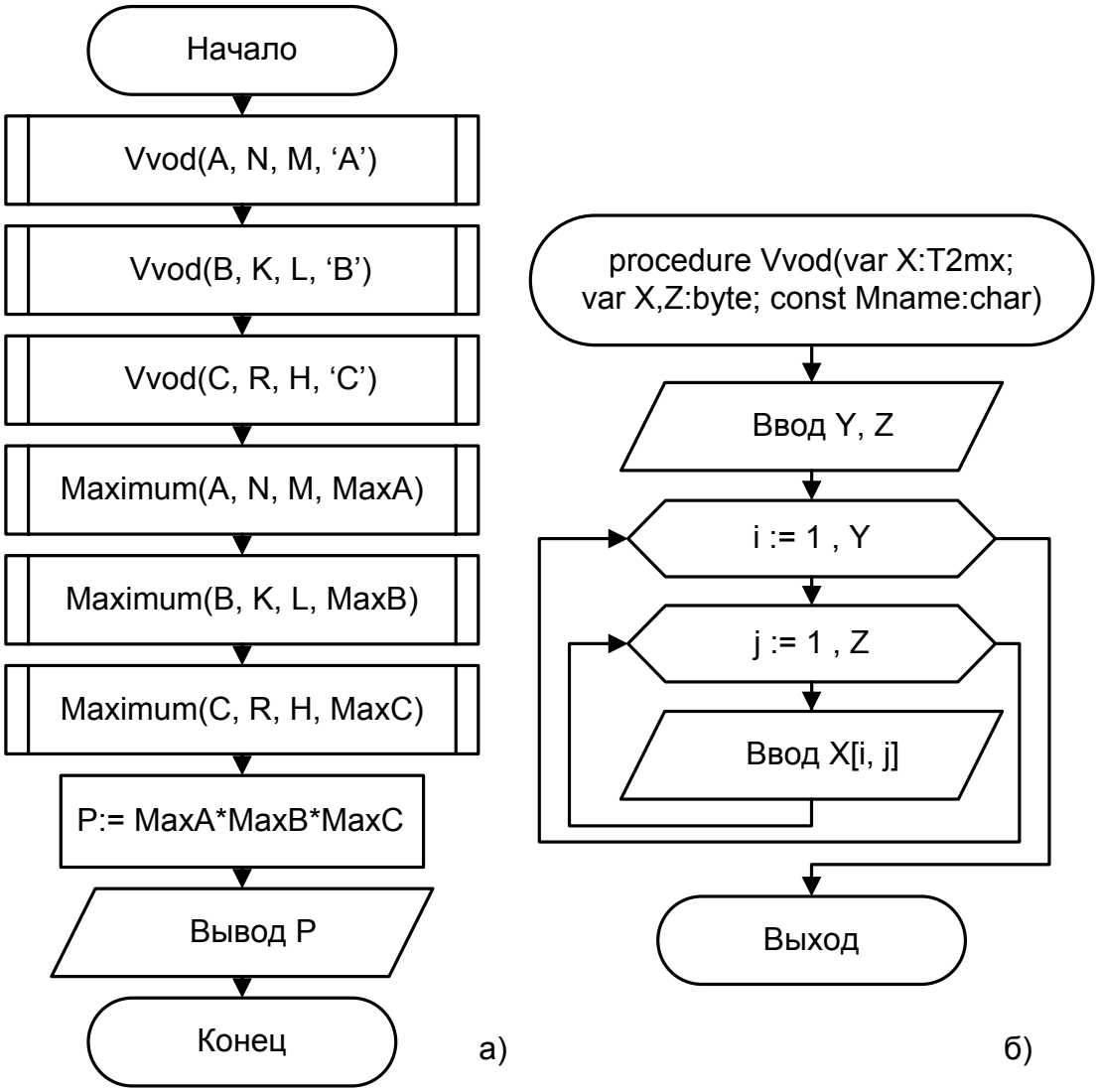


Рисунок 7.9 Основная программа и процедура ввода матрицы

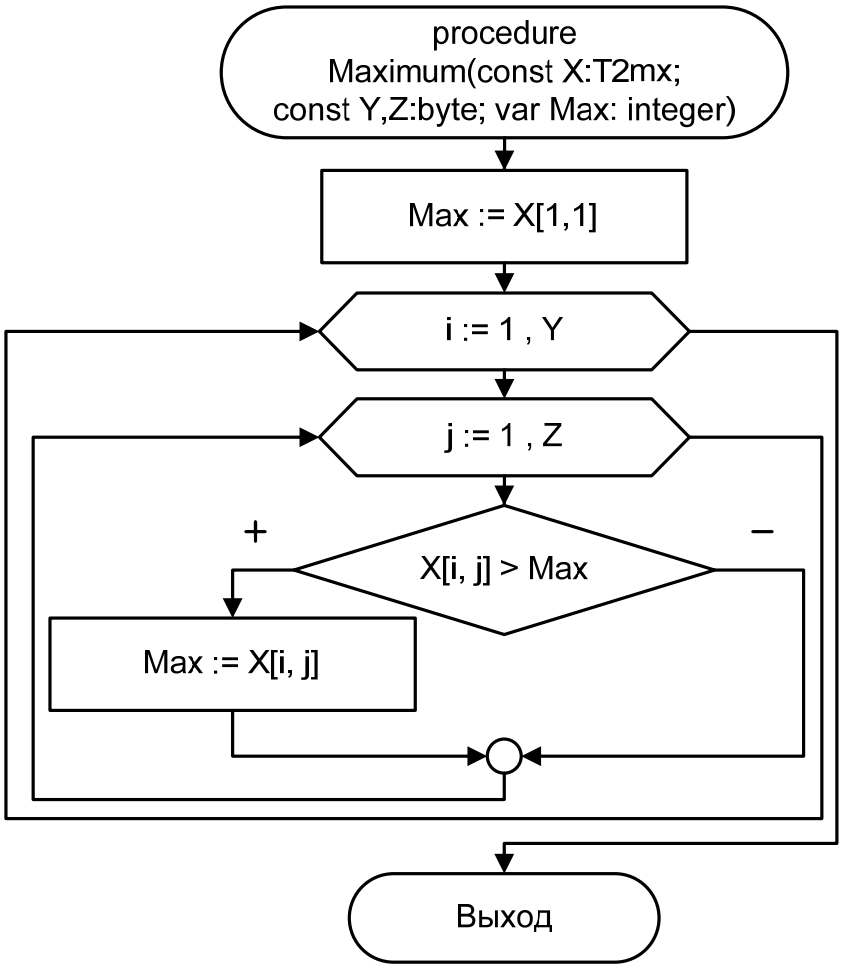


Рисунок 7.10 Процедура поиска максимума в матрице

Затем составляется блок-схема основной программы (Рисунок 7.9 а)), процедур ввода (Рисунок 7.9 б)) и поиска максимума (Рисунок 7.10).

При вызове процедур используется три набора фактических параметров, первый для матрицы ***A***, второй для матрицы ***B***, и третий – для 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 шесть параметров в одном блоке.

Процедура ***Maximum*** также должна вызываться трижды, но чтобы не потерять значение максимального элемента матрицы ***A***, нужно при вызове 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;
  writeln ('P=',P);
End.
```

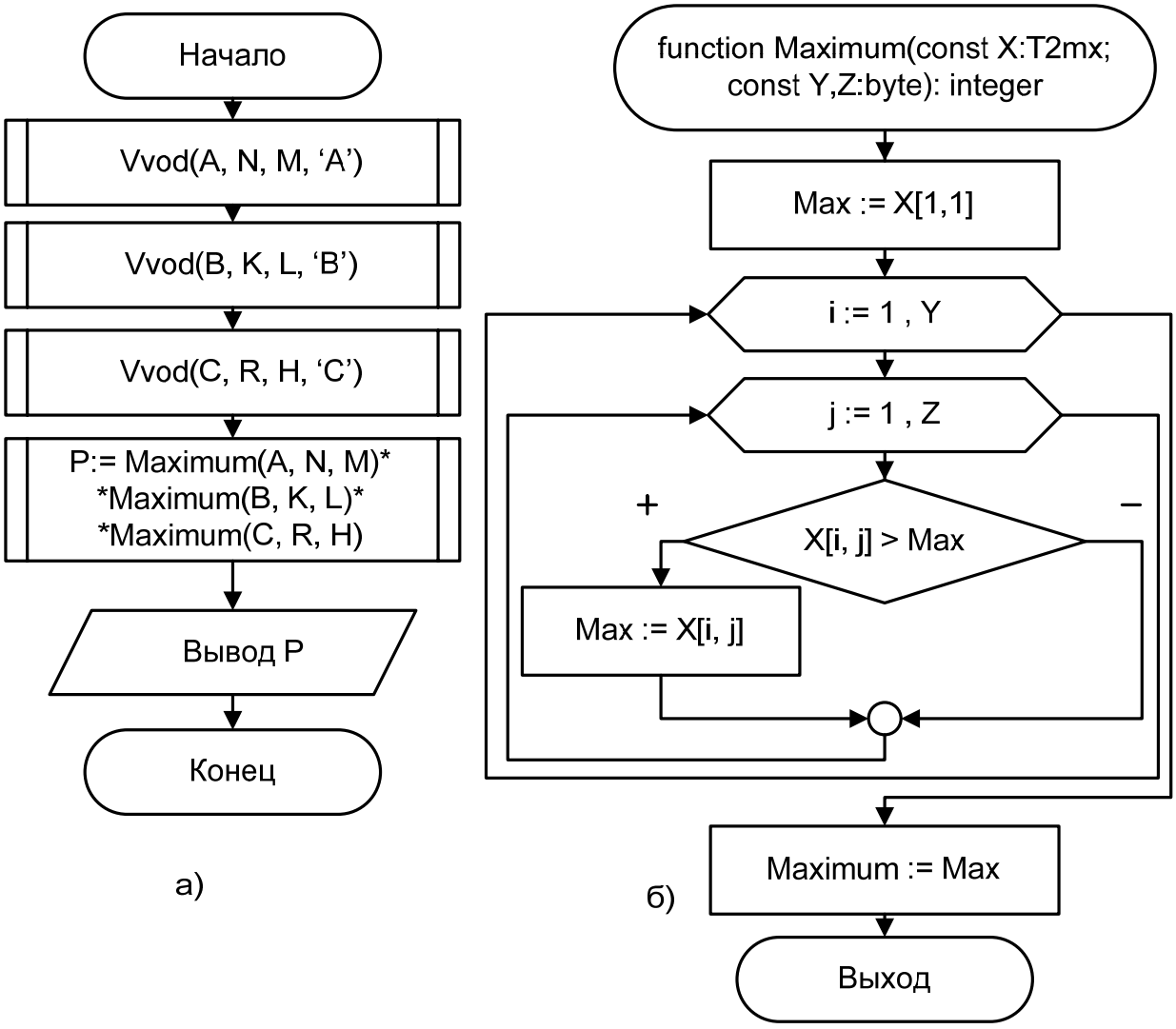


Рисунок 7.11 Решение задачи с помощью функции

Теперь решим ту же задачу, но только используя для поиска максимума не процедуру, а функцию, поскольку именно функция лучше подходит для решения, т.к. от каждой матрицы надо получить, по сути, одно единственное число, а функция, как раз, возвращает единственное значение. Блок-схема несколько изменится. Поменяется основная программа (Рисунок 7.11 а)) и процедура поиска максимума преобразуется в функцию (Рисунок 7.11 б)). Блок-схема ввода матрицы останется прежней (Рисунок 7.10 а)). Соответственно, программа может быть записана так:

```
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: $A_{3\times3} = \begin{pmatrix} 2 & -6 & 5 \\ -3 & 3 & 1 \\ -4 & 2 & 4 \end{pmatrix}$, $B_{4\times4} = \begin{pmatrix} 2 & 0 & -1 & 3 \\ -5 & -6 & 7 & 4 \\ 8 & -1 & 2 & 0 \\ 11 & -3 & -6 & 10 \end{pmatrix}$

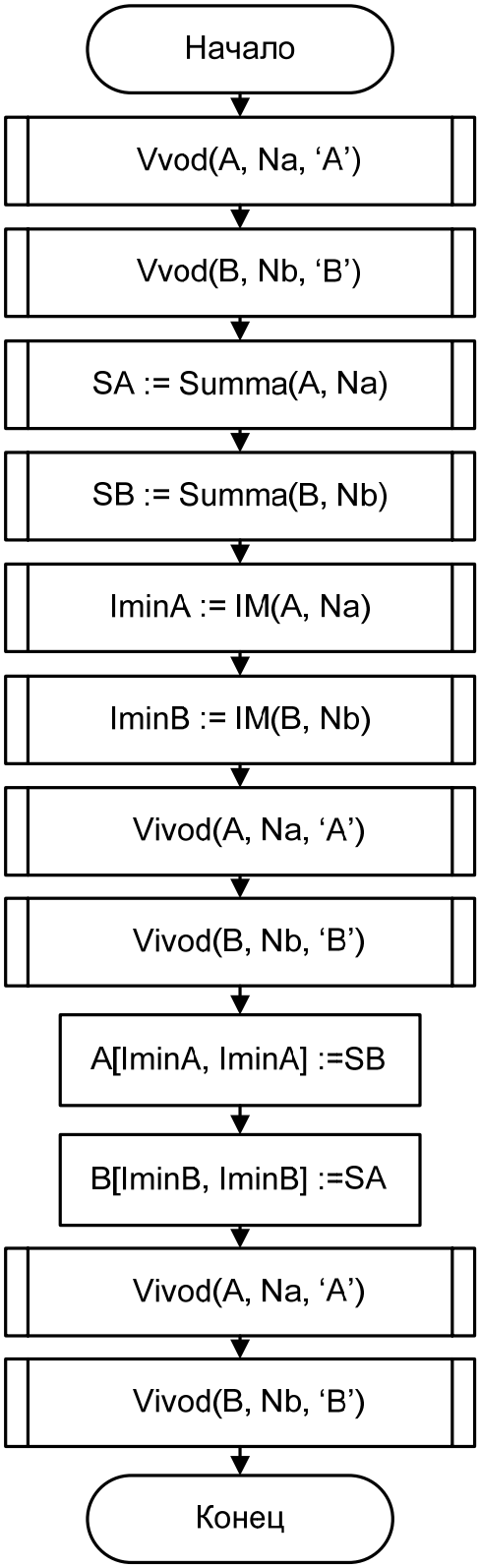


Рисунок 7.12 Основная программа

Промежуточные данные:
SA=–13, ***SB= – 22***
IminA = JminA=1; ***IminB= JminB=2***.
Выходные данные:

$$A = \begin{array}{|c|c|c|}\hline -13 & -6 & 5 \\ \hline -3 & 3 & 1 \\ \hline -4 & 2 & 4 \\ \hline \end{array},$$

$$B = \begin{array}{|c|c|c|c|}\hline 2 & 0 & -1 & 3 \\ \hline -5 & -22 & 7 & 4 \\ \hline 8 & -1 & 2 & 0 \\ \hline 11 & -3 & -6 & 10 \\ \hline \end{array}$$

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.

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 Рисунок 7.12. Следует отметить, что перед модификацией матриц мы их выводим 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.

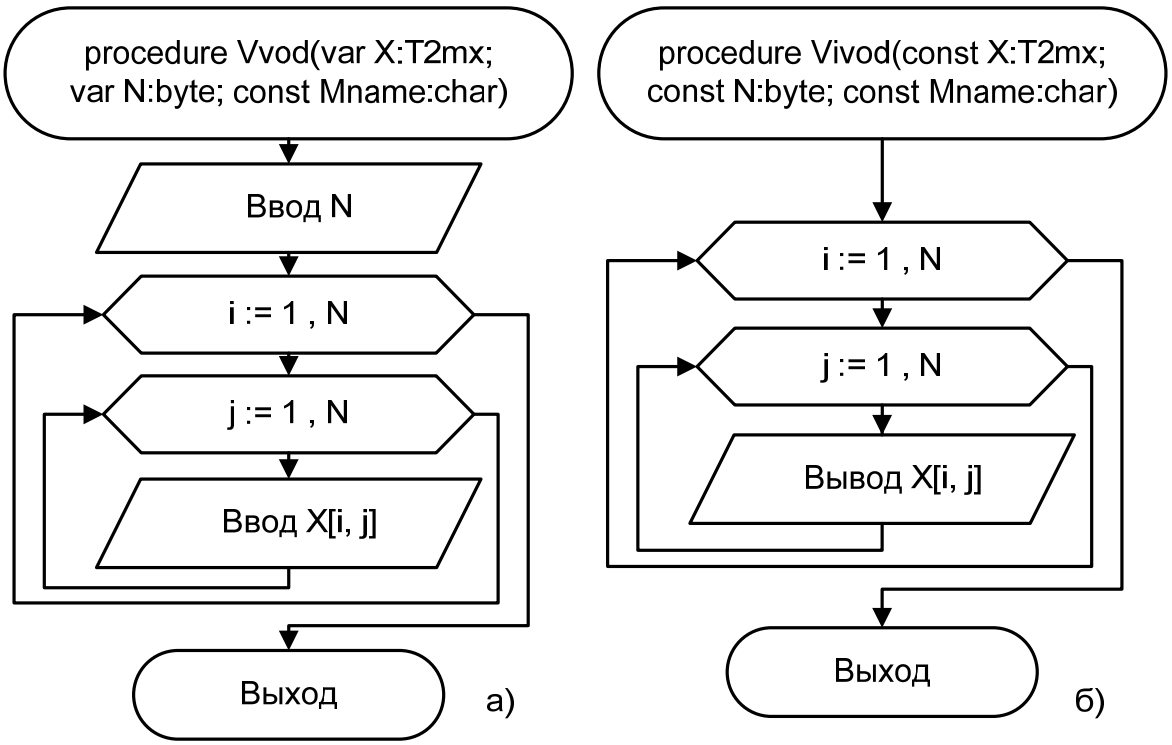


Рисунок 7.13 Ввод и вывод квадратной матрицы

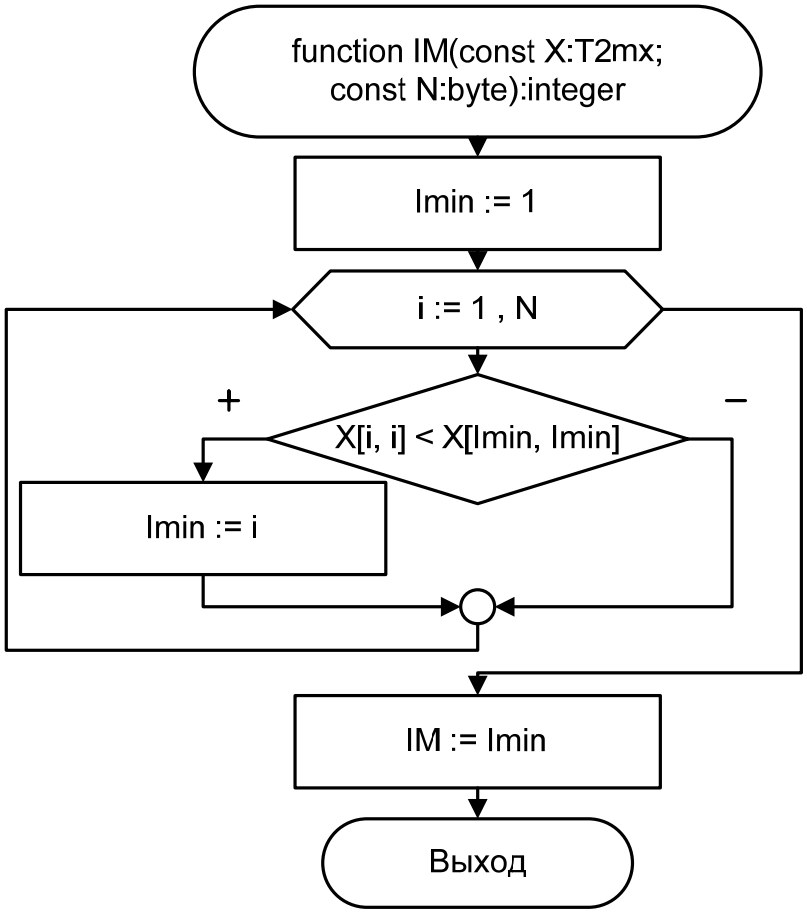


Рисунок 7.14 Поиск минимума в квадратной матрице

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*** (Рисунок 7.15), 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*** (Рисунок 7.14), 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.

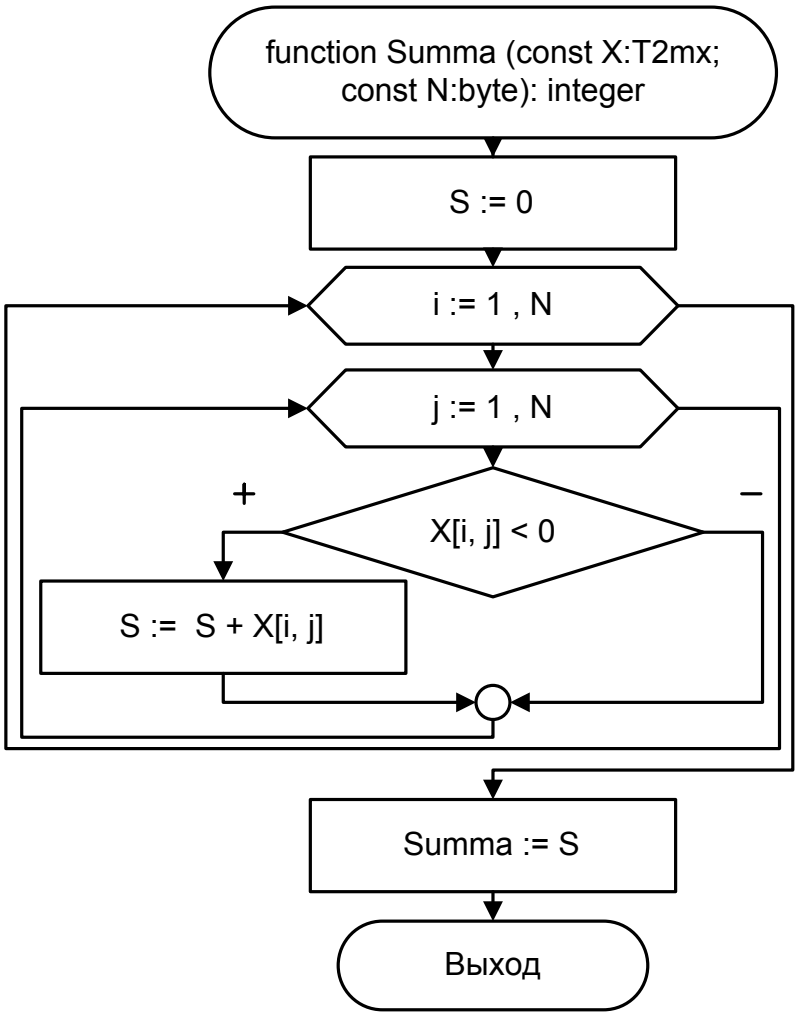


Рисунок 7.15 Сумма отрицательных элементов

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 ***}
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.
```

8. ФАЙЛЫ

8.1 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*.

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. Именно такие файлы нами будут рассмотрены далее. Основные отличия от массивов показаны в Таблица 8.1.

Таблица 8.1 «Сравнение компонентных файлов и массивов»

	скорость обработки	***возможный размер***	***энергонезависимость***
файлы	медленно	огромный	есть
массивы	быстро	малый	отсутствует

Пример объявления файловых типов средствами языка *Pascal*:

```
type
  TTxt = text;
  TBinare = file;
  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.

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****);***

8.2 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 понятие файла 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 Рисунок 8.2, где он обозначен черной стрелочкой. Следует отметить, что счет позиции указателя начинается с нулевой. Это значит, что находясь в начале

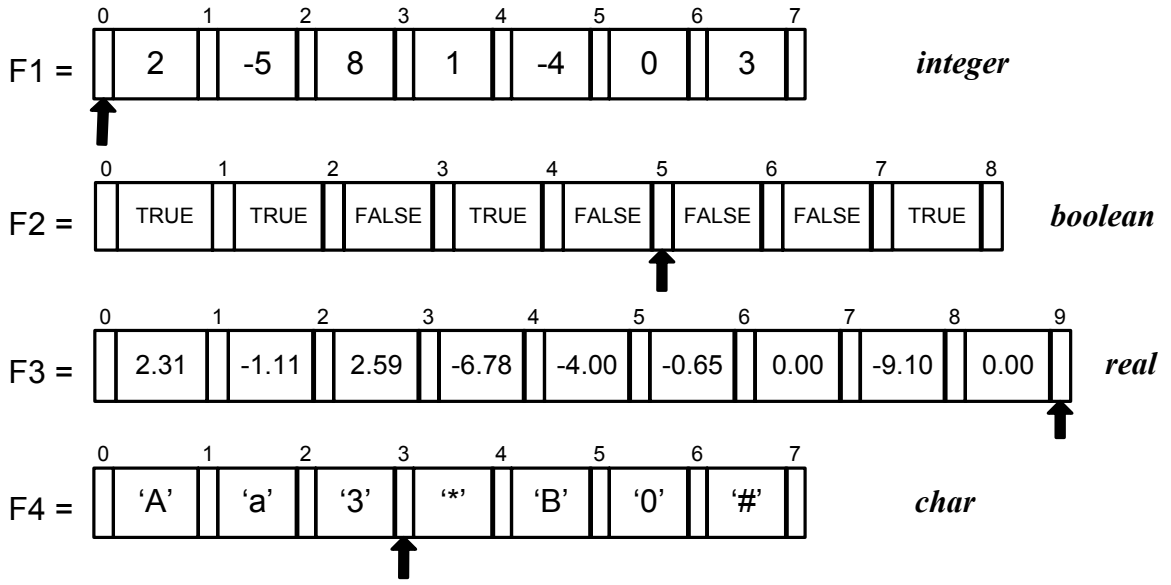


Рисунок 8.2 Примеры различных файлов: F1 – целочисленный; F2 – логический; F3 – вещественный; F4 – символьный

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).

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. Т.е. 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

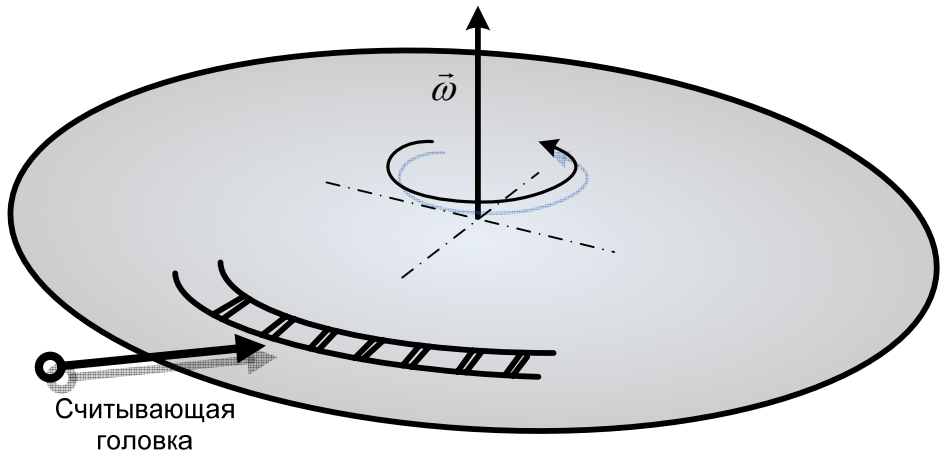


Рисунок 8.1 Модель диска и файла на нем

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 Рисунок 8.1.

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 Рисунок 8.2. 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 Рисунок 8.3. 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

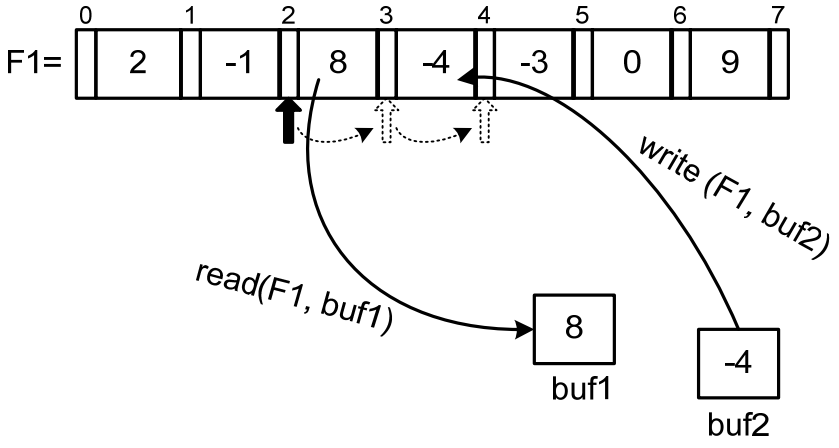


Рисунок 8.3 Иллюстрация чтения компоненты файла F1 в переменную buf1 и записи значения переменной buf2 в файл.

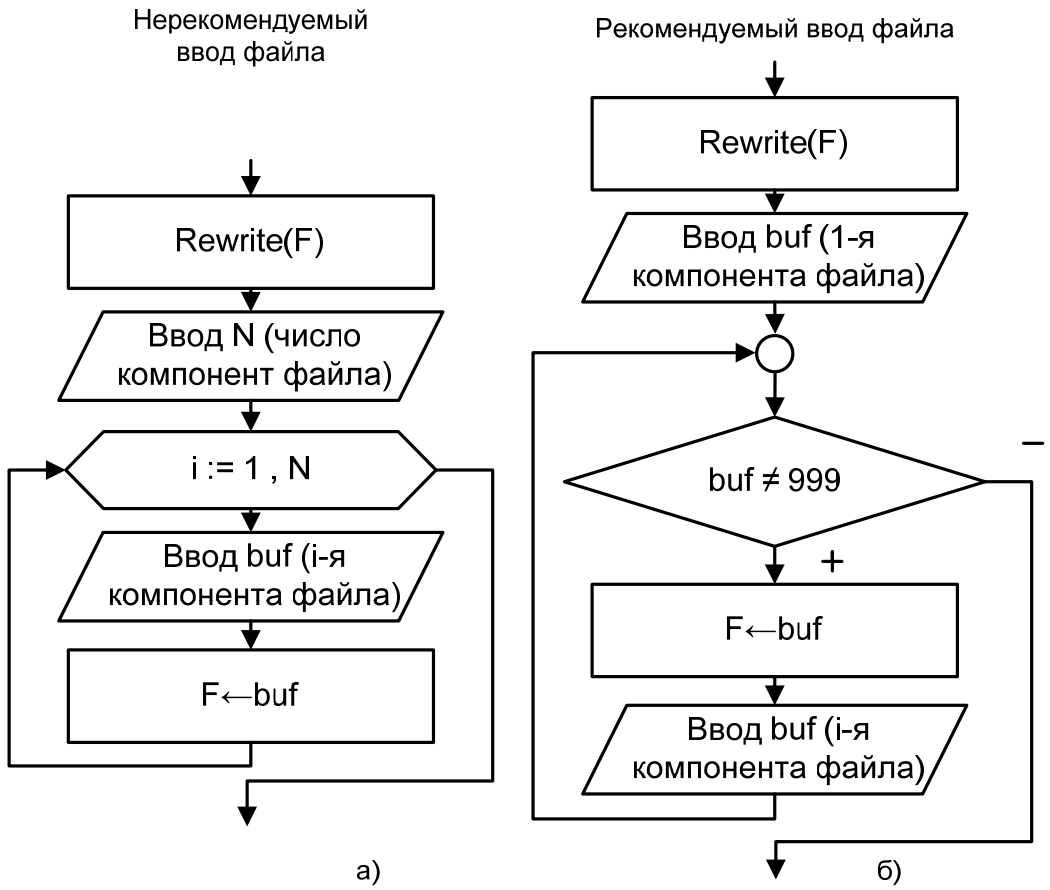


Рисунок 8.4 Типовые алгоритмы ввода файлов: а) – по известному числу компонент (не рекомендуется к использованию); б) – по признаку конца ввода

массивам сведения о количестве компонент. Блок-схема типового алгоритма ввода представлена на Рисунок 8.4. 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 ввод не исходя из предполагаемого объема, а пользуясь некоторым *стоп-событием*. Вообще, алгоритм изображенный на Рисунок 8.4 а) является нерекомендуемым и по возможности его следует избегать.

В варианте Рисунок 8.4 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.

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

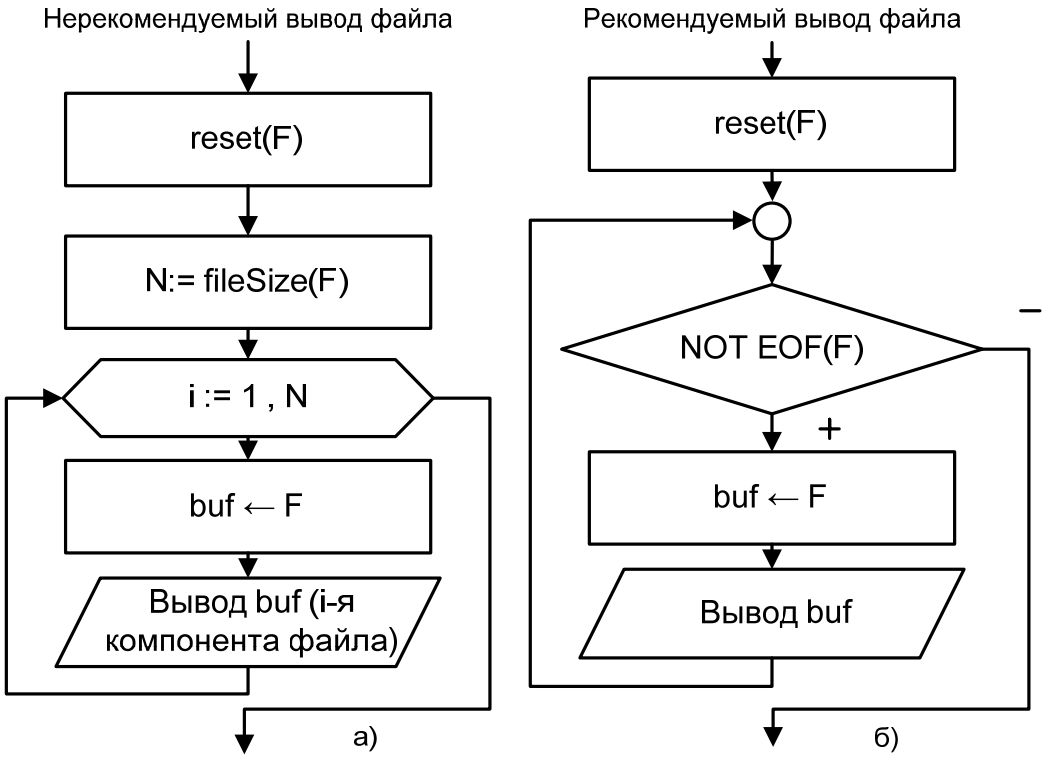


Рисунок 8.5 Вывод файла: а) – по известному числу компонент (не рекомендуется к использованию); б) – по признаку конца файла

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 Рисунок 8.5.

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****);***

8.3 Файлы последовательного доступа

Если обработка элементов файла ведется от начала и до конца последовательно, без использования процедуры принудительного преставления курсора, то такая обработка называется последовательным доступом. Если обработке подвергаются все без исключения элементы, то блок-схема будет такой, как на Рисунок 8.6 а). Если на элемент накладываются некоторые условия, то в тело цикла добавляется дополнительная развилка (Рисунок 8.6 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:

	0	1	2	3	4	5	6	7	8
F =	-2	3	8	1	0	9	-10	-3	

выход:

S = –2+3+8+1+0+9+(–10)+(–3) = 6.

Решение задачи целиком показано на Рисунок 8.7.

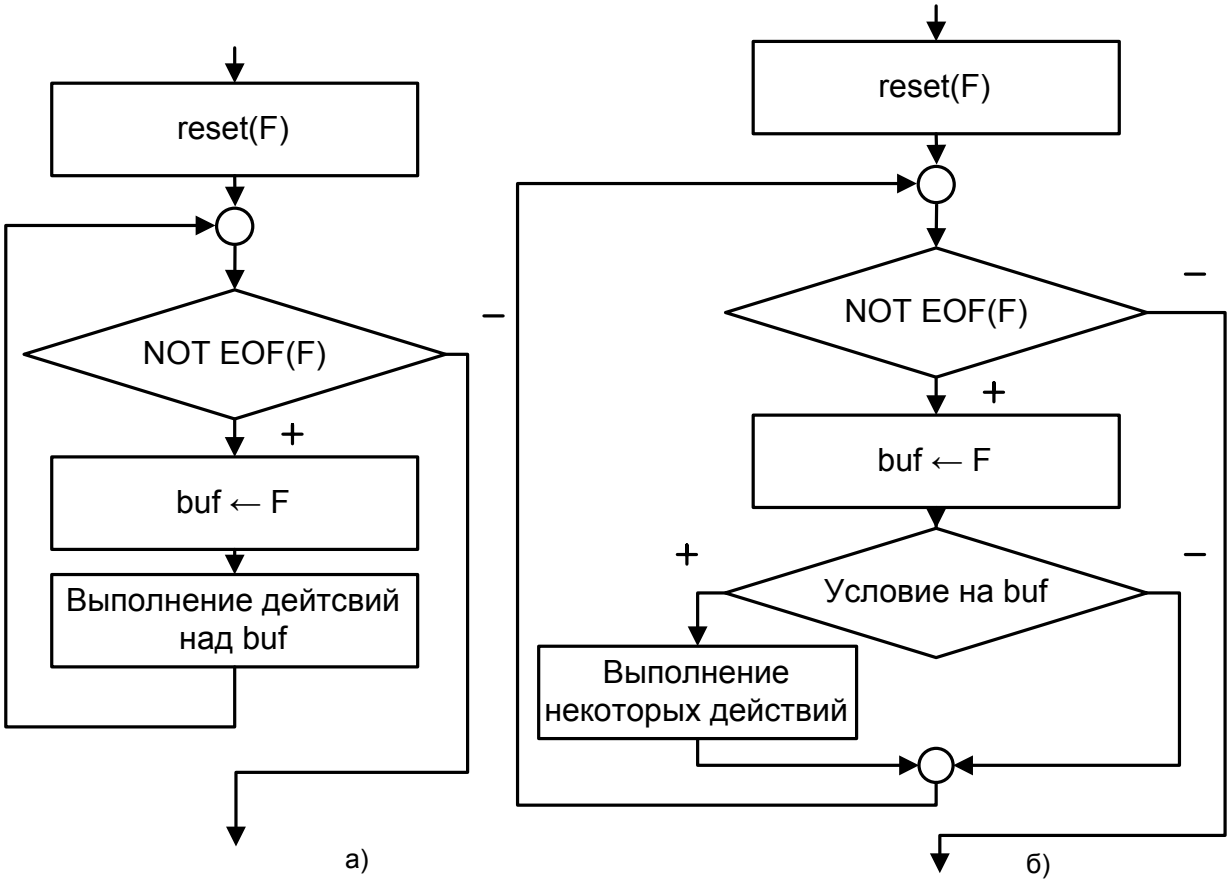


Рисунок 8.6 Последовательная безусловная обработка файла а) и последовательная условная обработка файла б)

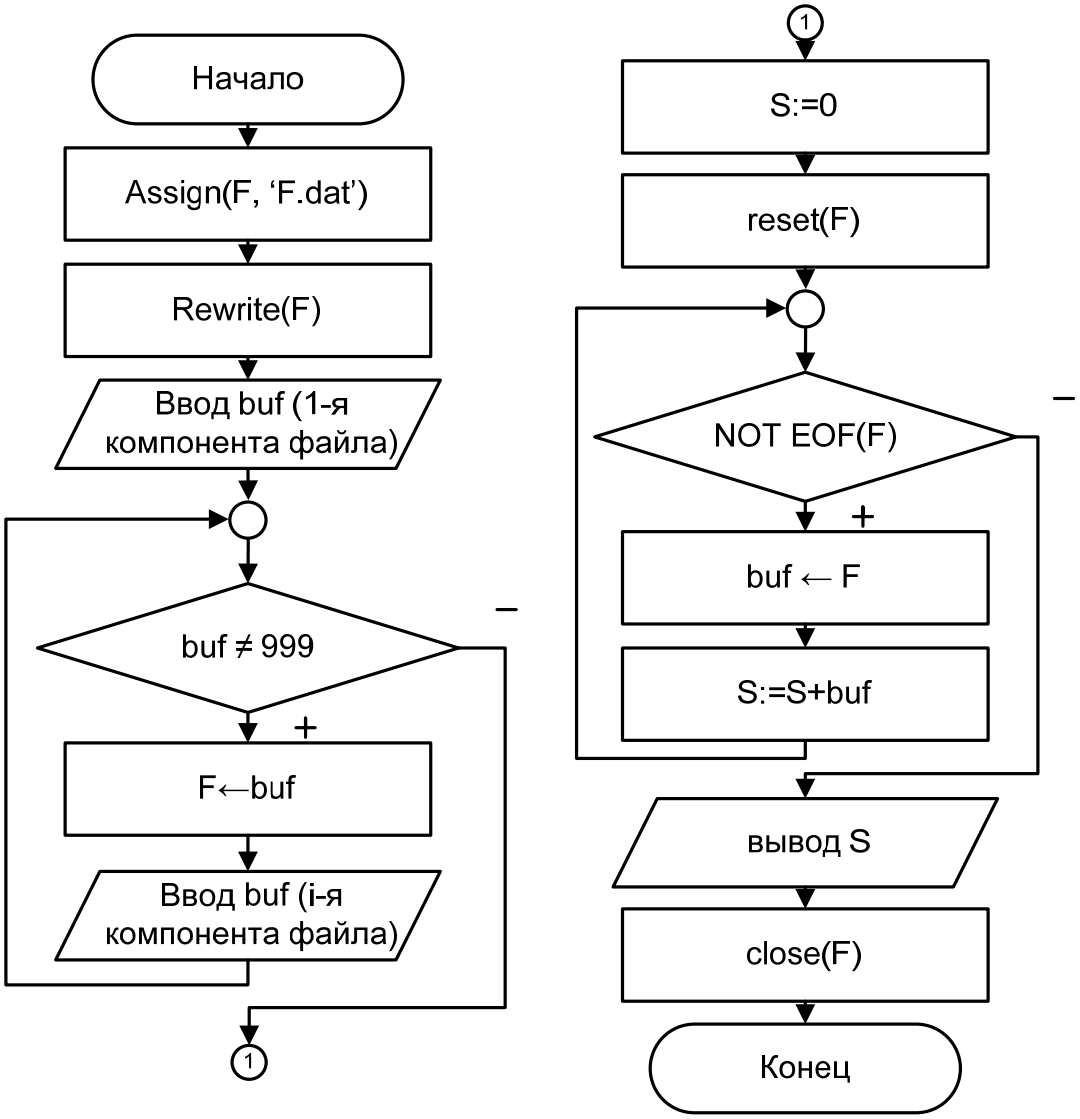


Рисунок 8.7 Поиск суммы элементов файла

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.
```

Следующий пример на последовательную обработку с анализом элементов.

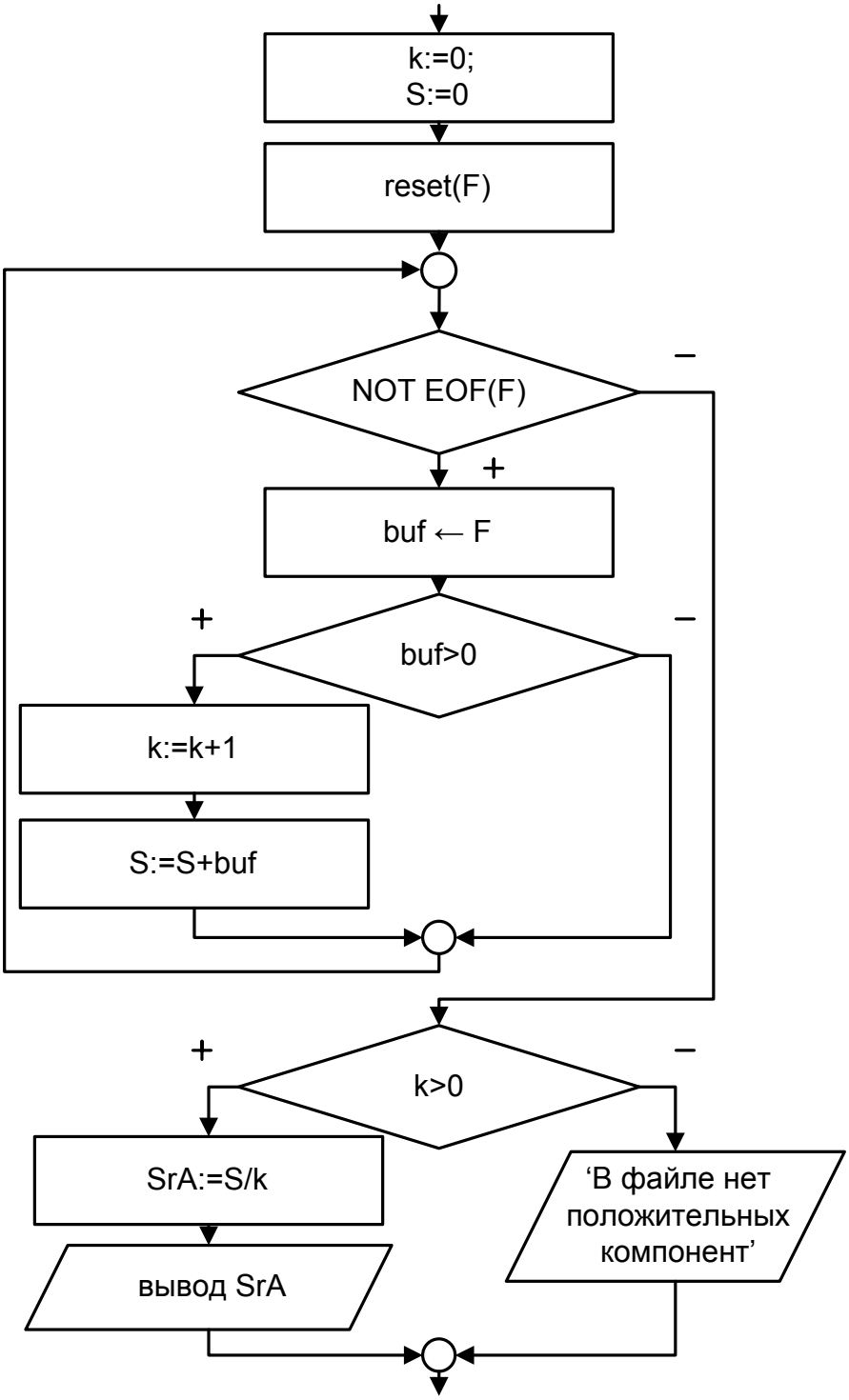
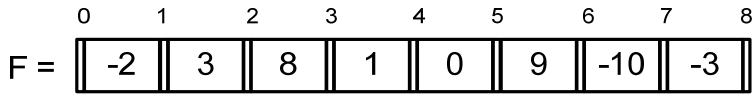


Рисунок 8.8 Среднее арифметическое положительных компонент

ПРИМЕР

Найти среднее арифметическое положительных компонент файла.

вход:



выход: SrA = (3+8+1+9)/4= 21/4 = 5,25.

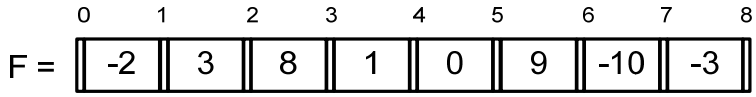
Приведем здесь блок-схему только основного алгоритма без ввода файла, который был расписан ранее. Мы читаем файл от начала до конца и анализируем прочитанные компоненты. Если компоненты оказываются большими нуля, то их добавляем в сумму ***S***, не забывая при этом увеличивать счетчик ***k***. Среднее арифметическое ***SrA*** существует лишь тогда, когда в файле есть положительные элементы. Блок-схема описанного процесса показана на Рисунок 8.8. Листинг приводить не будем.

Далее еще один пример на последовательную обработку.

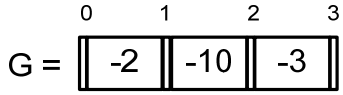
ПРИМЕР

Переписать отрицательные компоненты файла F в файл G.

вход:



выход:



Сначала руками вводим файл ***F***, далее проходим по нему в прямом направлении и встретившиеся отрицательные компоненты копируем 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 (Рисунок 8.9).

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;
```

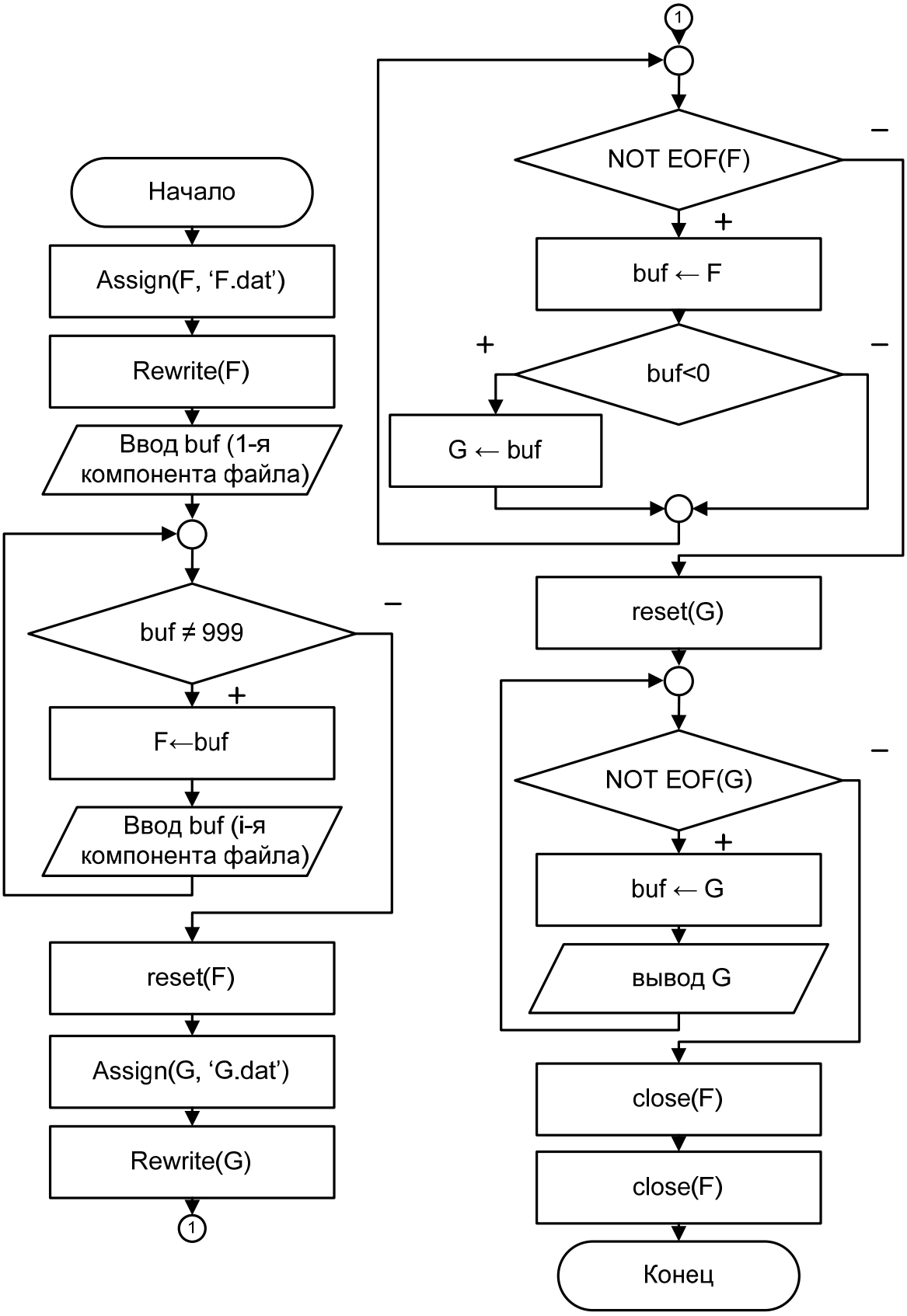


Рисунок 8.9 Переписывание отрицательных компонент одного файла в другой файл

```
  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.
```

8.4 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 Рисунок 8.10.

Далее рассмотрим задачу использующую среди прочего описанный алгоритм.

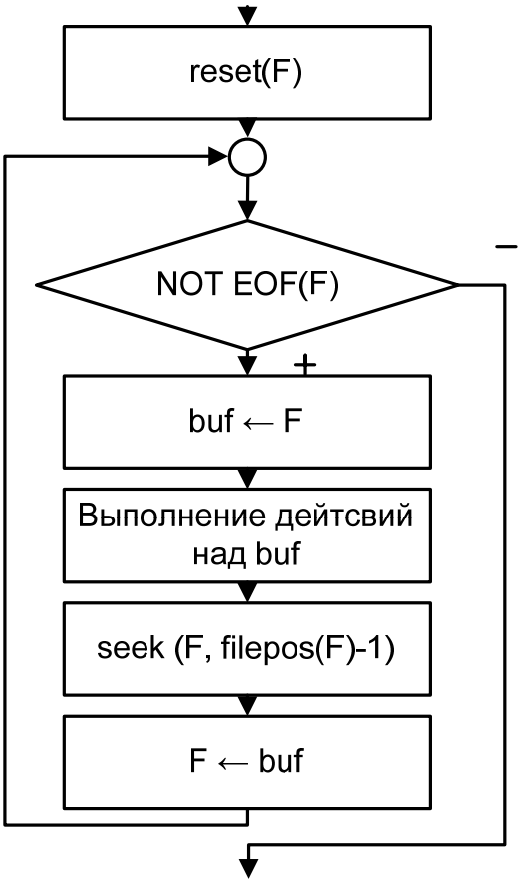


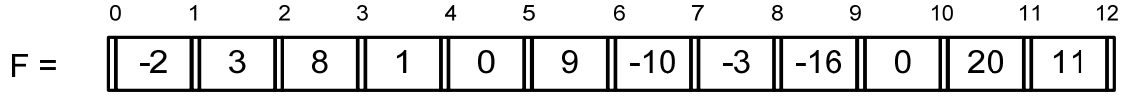
Рисунок 8.10 формирование новых компонент на основе прочитанных

ПРИМЕР

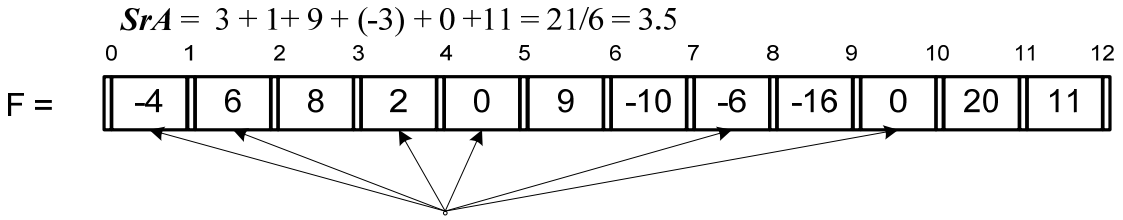
Ввести файл действительных чисел. Найти в нем среднее арифметическое каждой второй компоненты. Далее все компоненты, чей модуль меньше модуля найденного среднего арифметического, удвоить. Файл до и после преобразования вывести на экран.

Начинаем, как обычно, с тестового примера:

Входные данные:



Выходные данные:



Следующий этап – это составление блок-схемы алгоритма. Можно сначала записать общий алгоритм, а далее каждый шаг детализировать. Общая последовательность действий решения представлена на Рисунок 8.11 а). Здесь шаги 1-2, 2-3 и 5-6 уже были описаны ранее (см. Рисунок 8.4, Рисунок 8.5). Потому есть смысл детализировать шаги 3-4 и 4-5. Их детализация изображена на Рисунок 8.11 б) и . 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, т.к. 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*:

```
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;
  writeLn('999 – окончание ввода');
```

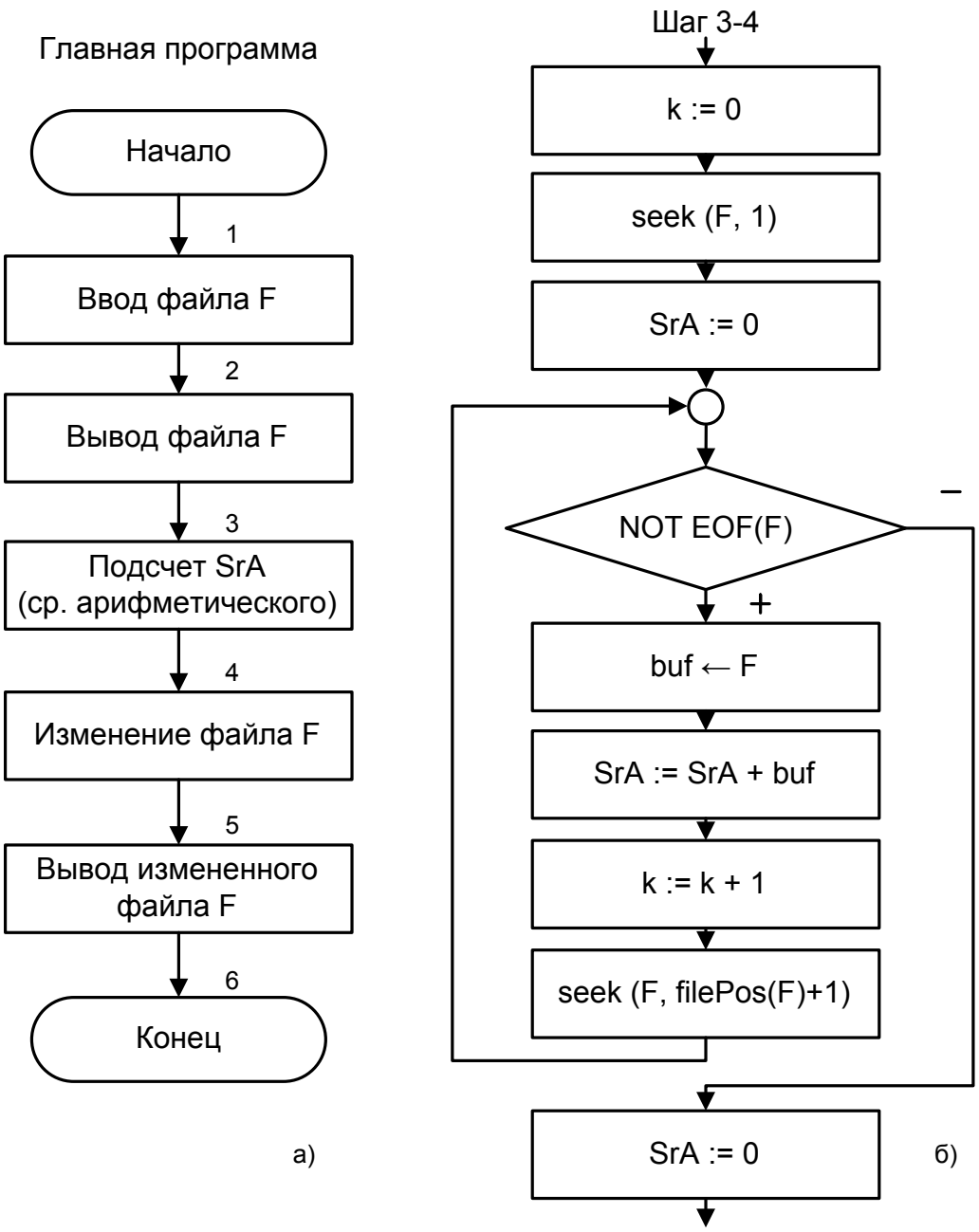


Рисунок 8.11 Главная программа – а)
и поиск среднего арифметического каждой второй компоненты – б)

```
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.
```

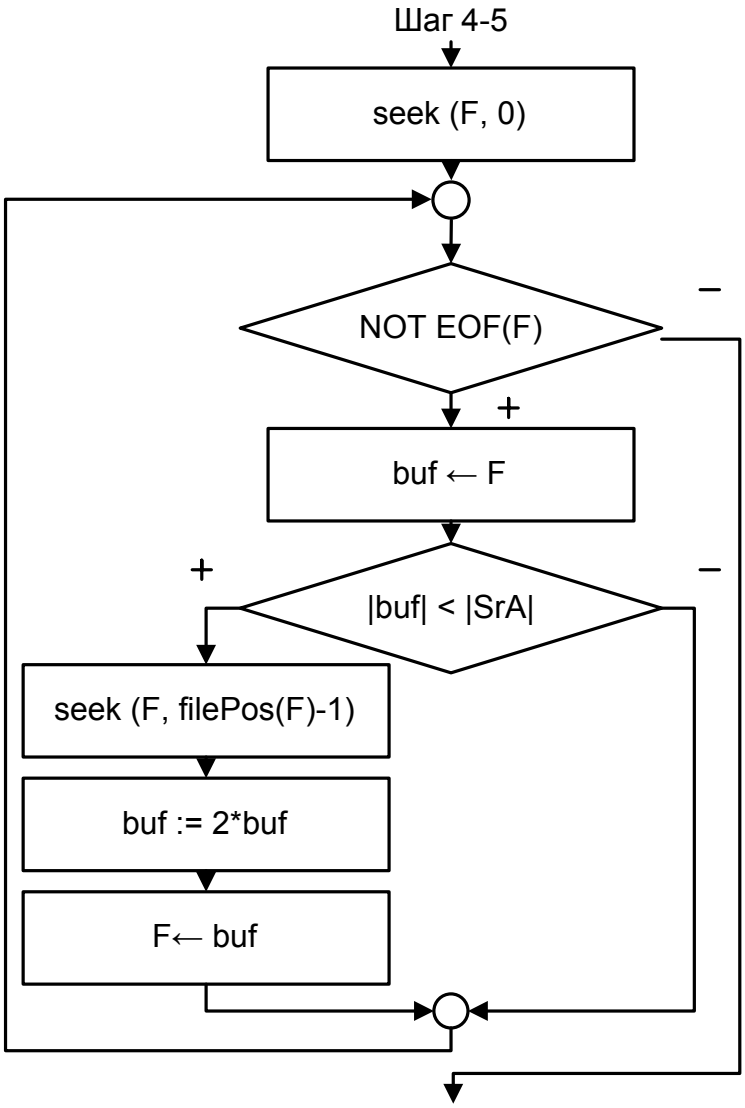


Рисунок 8.12 Удвоение компонент удовлетворяющих условию

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 идейную схожесть, все-таки имеются различия в реализации. Самое главное отличие, как отмечалось ранее, в том, что для доступа к компонентам файла используется указатель, в отличии 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 (Рисунок 8.13), а также алгоритм сортировки элементов по возрастанию (Рисунок 8.14). Прочие алгоритмы здесь приводить не будем.

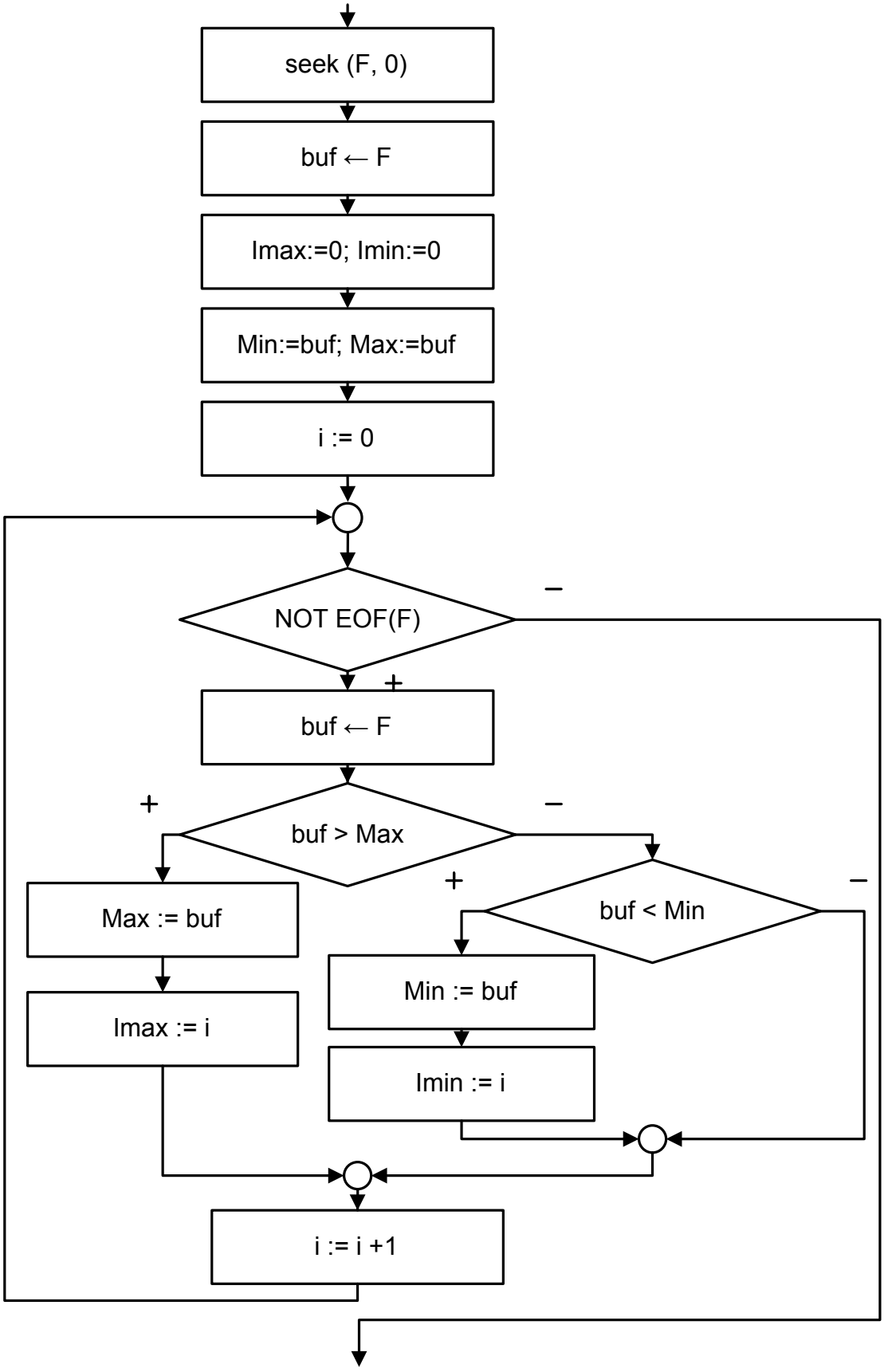


Рисунок 8.13 Поиск экстремальных компонент в файле и их позиций

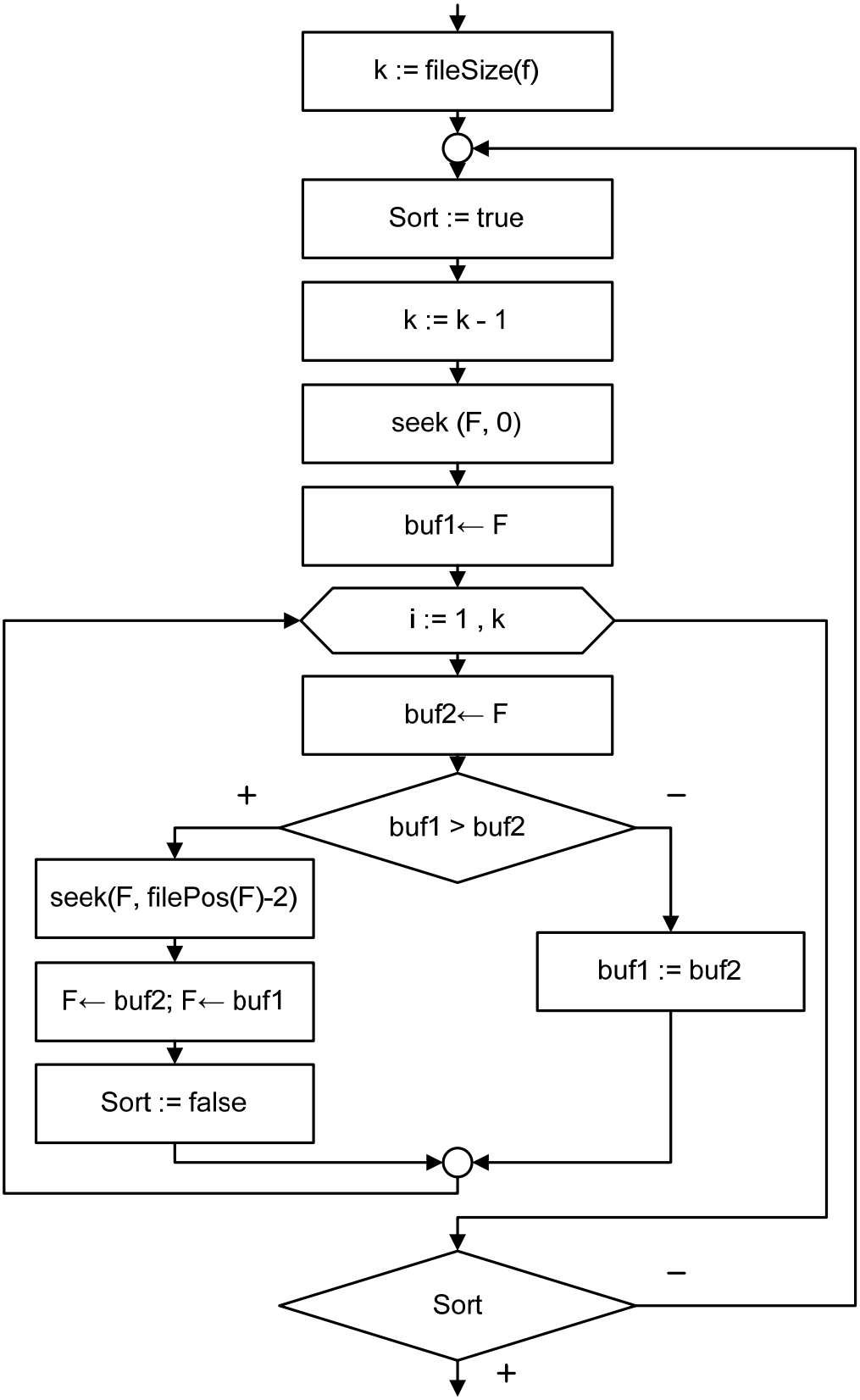


Рисунок 8.14 Сортировка файла по возрастанию методом «пузырька»

8.5 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 Рисунок 8.15.

Теперь опишем все процедуры входящие в данный алгоритм. Это, прежде всего, ввод и вывод (***InputFile*** и ***OutputFile***) – **Ошибка! Источник ссылки не найден.**, а также процедура ***FormNewFile*** (**Ошибка! Источник ссылки не найден.**). 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.

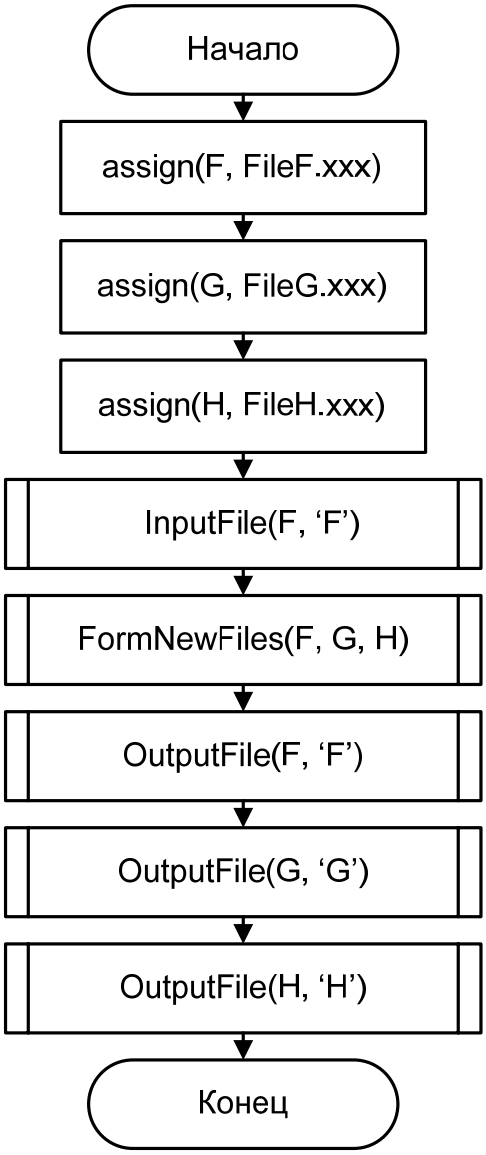


Рисунок 8.15 Основная программа к задаче на чтение файла в обратном порядке

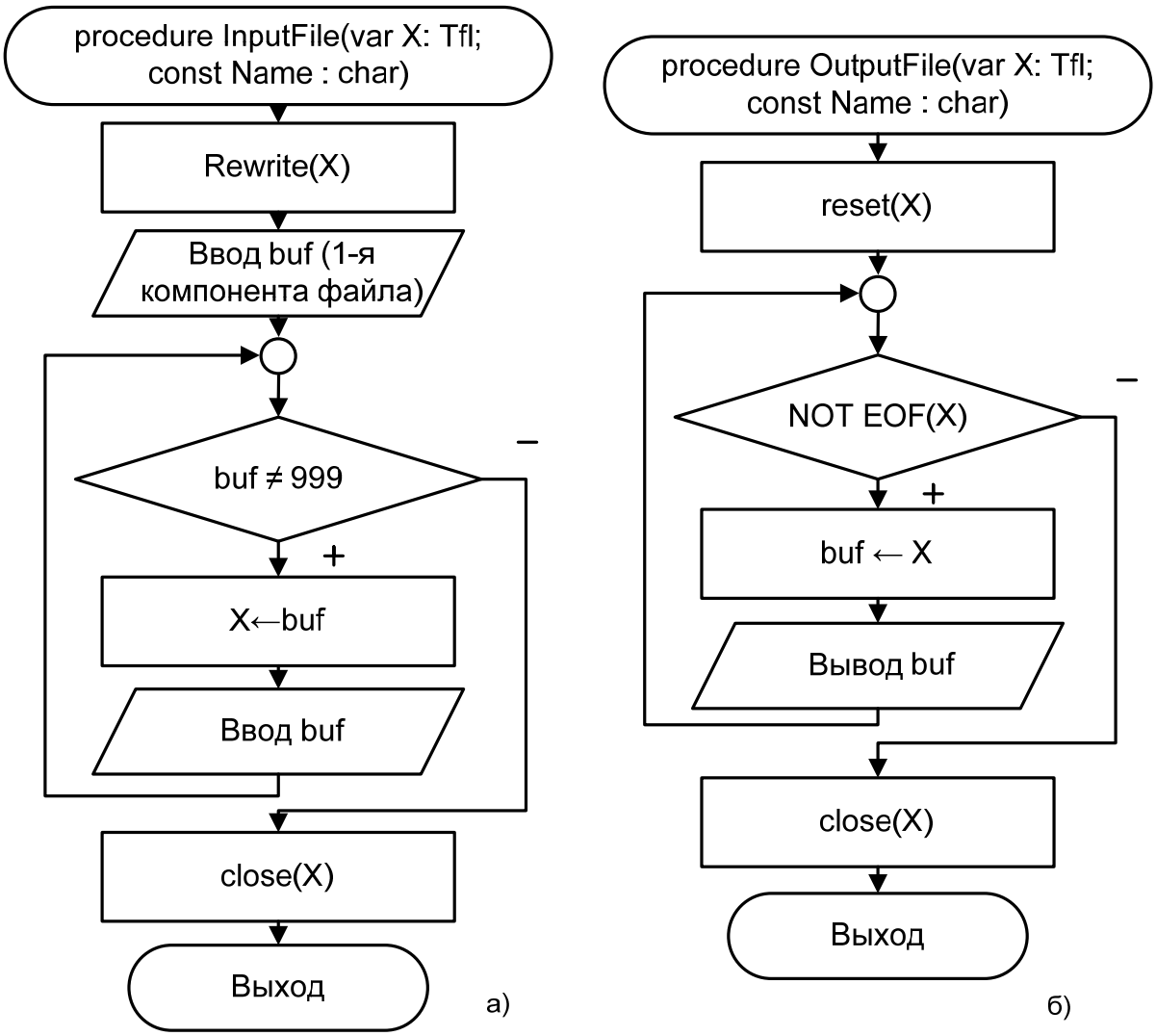


Рисунок 8.16 Процедуры ввода и вывода файла

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);
      write(i,'-я комп. файла = ');
      readLn(buf); inc(i);
    end;
  close(X);
end;
```

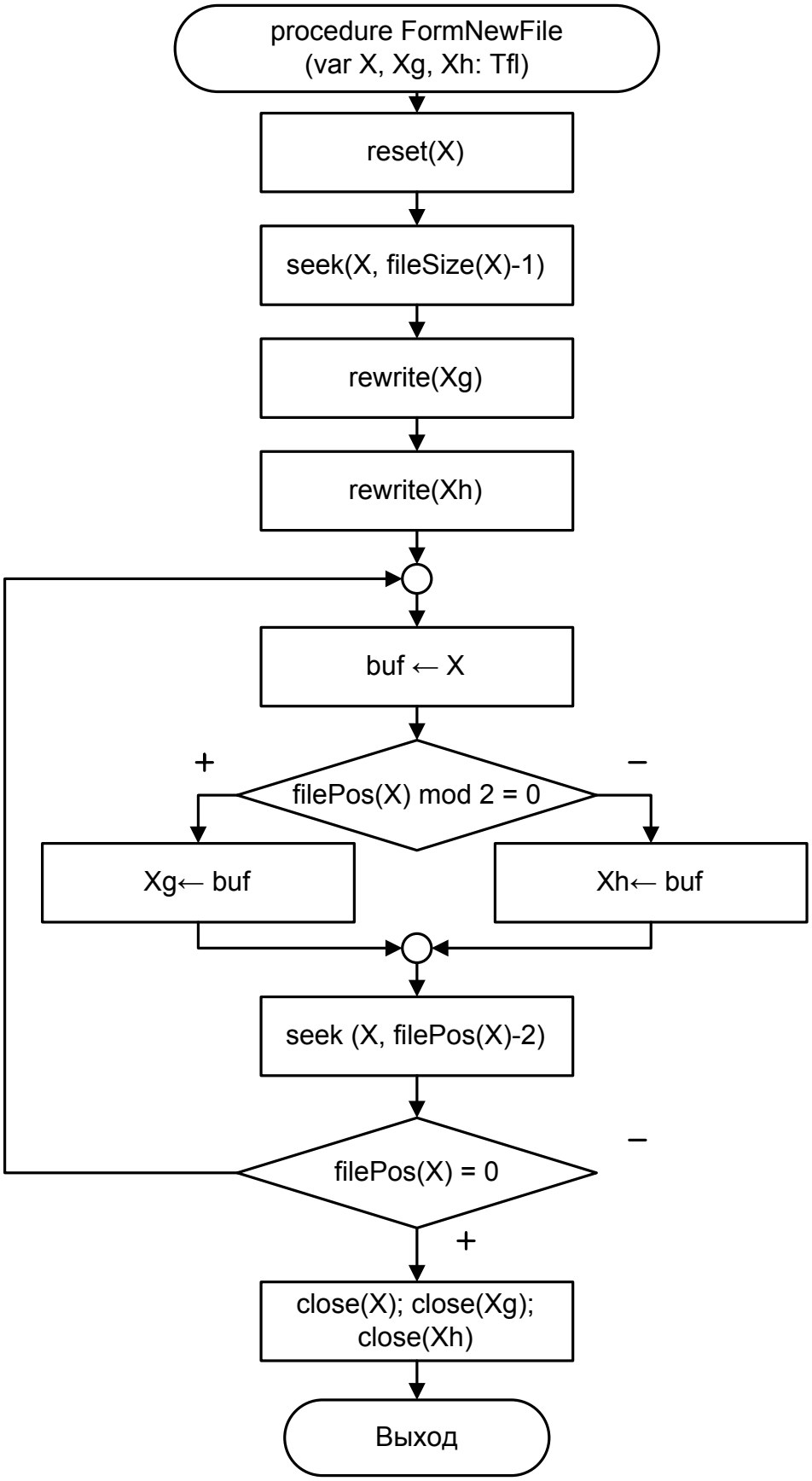


Рисунок 8.17 Процедура чтения файла в обратном порядке и записи результатов в пару новых файлов

```
{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) 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.
```

8.6 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:

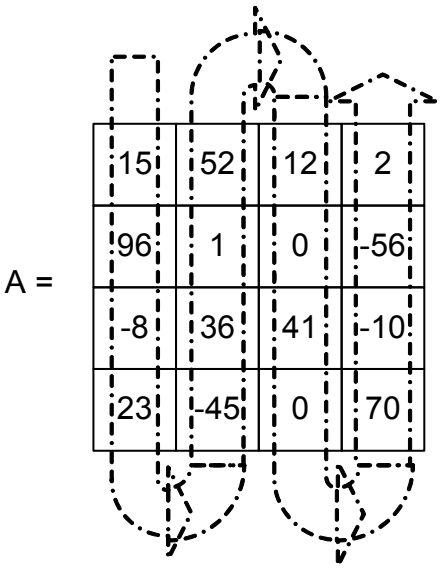
```
reset(F);
for i:=1 to N do
  for j:=1 to M do
    begin
      buf:= A[i,j]
      write(F,buf);
    end;
```

Обратная операция представляется более сложной ввиду гипотетической невозможности формирования двумерного массива из элементов файла. Если длина файла не равна произведению строк и столбцов

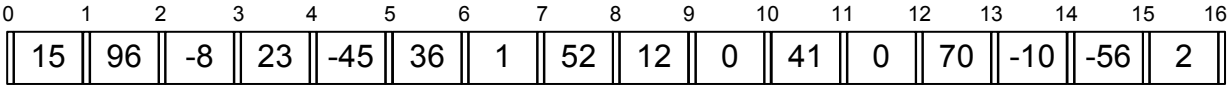
матрицы, то при ее формировании возникает множество вопросов: «А действительно ли в файле содержится матрица?», «В какой именно части файла она содержится?», «Может можно сформировать матрицу только из начальных элементов матрицы?». Рассмотрим случай формирования матрицы ***A*** размером ***N*×*M*** 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».



в результате получается файл F:



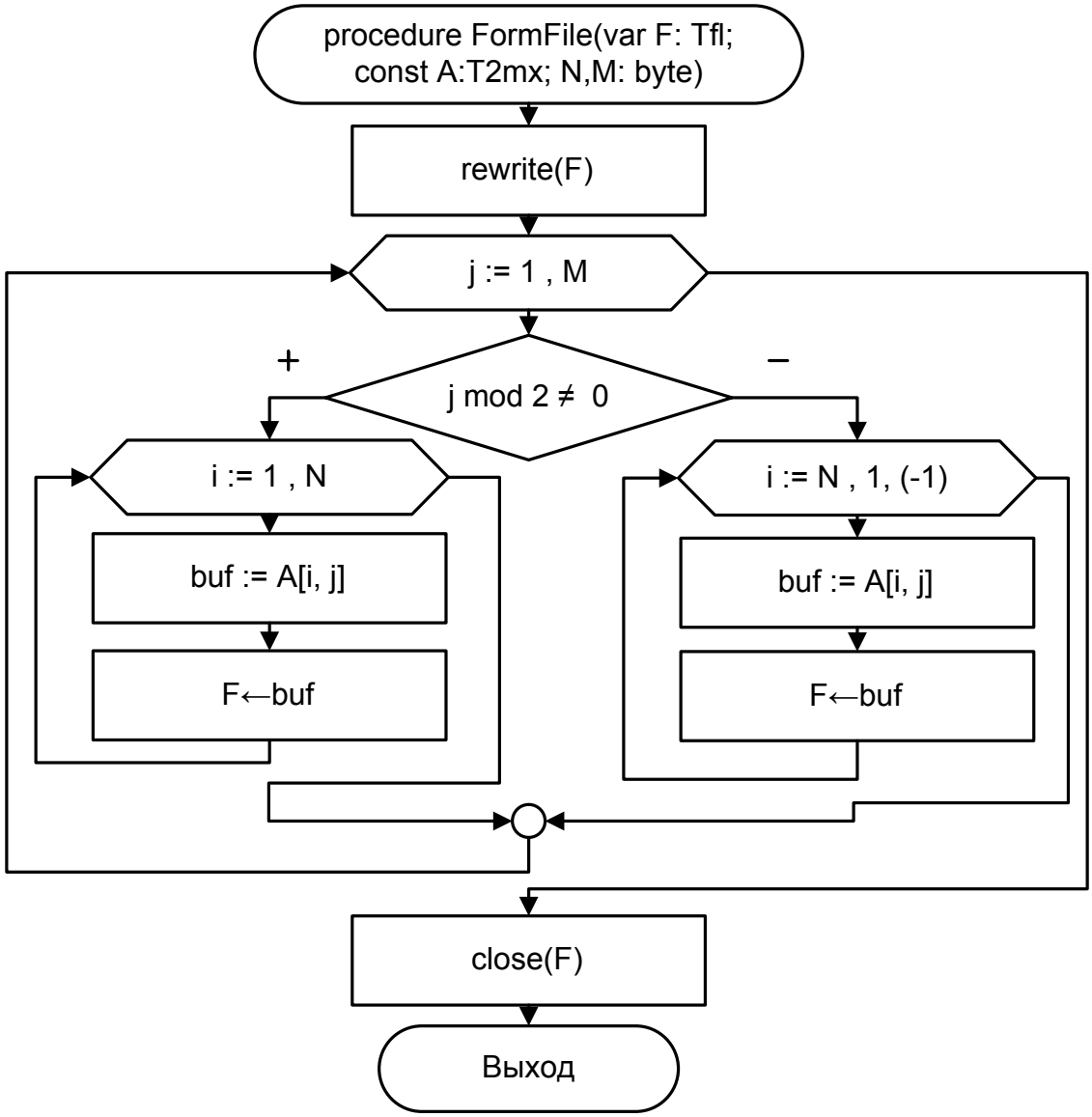


Рисунок 8.18 Формирование файла из матрицы "по змейке"

Не приводя стандартных процедур ввода/вывода матрицы и вывода файла, продемонстрируем процедуру собственно формирования файла, изображенную на Рисунок 8.18.

Рассмотрим полное решение достаточно сложной задачи на совместное использование файлов и массивов.

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 Рисунок 8.19.

а) – процедура нормировки матрицы; б) – процедура формирования файла из матрицы

Исходный файл F:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	4	15	-34	12	2	50	1	0	-9	-3	36	41	-10	

Матрица до нормировки:

15	-34	12	2
50	1	0	-9
-8	36	41	-10

***Max** = 50*

Матрица после нормировки:

0.30	-0.68	0.24	0.04
1.00	0.02	0.00	-0.18
-0.06	-0.72	0.82	-0.20

Нормированная матрица, занесенная обратно в файл F:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3.00	4.00	0.30	-0.68	0.24	0.04	1.00	0.02	0.00	-0.18	-0.16	-0.72	0.82	-0.20	

Вектор B:

3.00	4.00	0.30	0.24	0.04	1.00	0.02	0.82

Рисунок 8.19 Тестовый пример к задаче

Блок-схема 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;
     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;
```

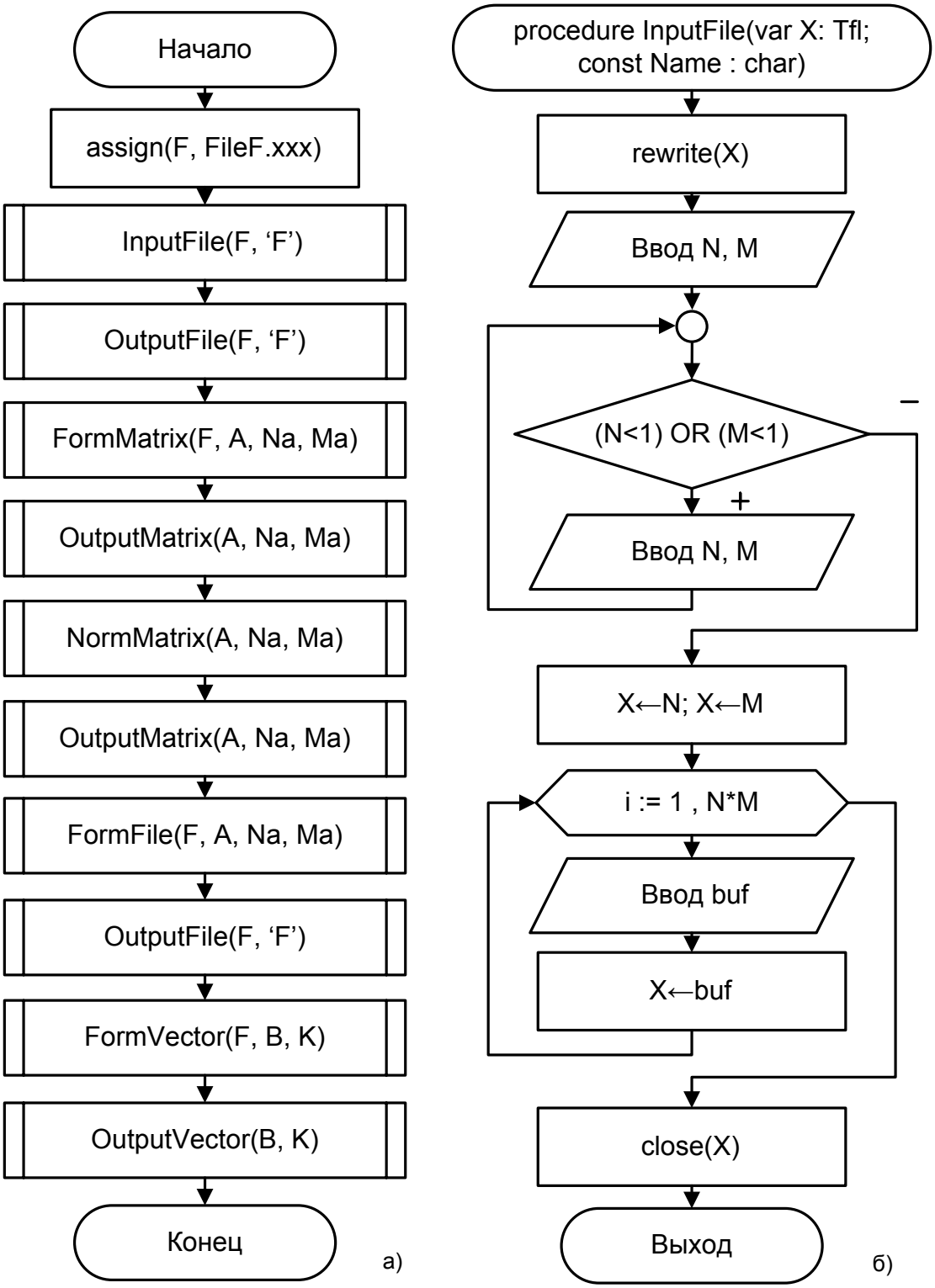


Рисунок 8.20 а) – главная программа; б) – формирование файла

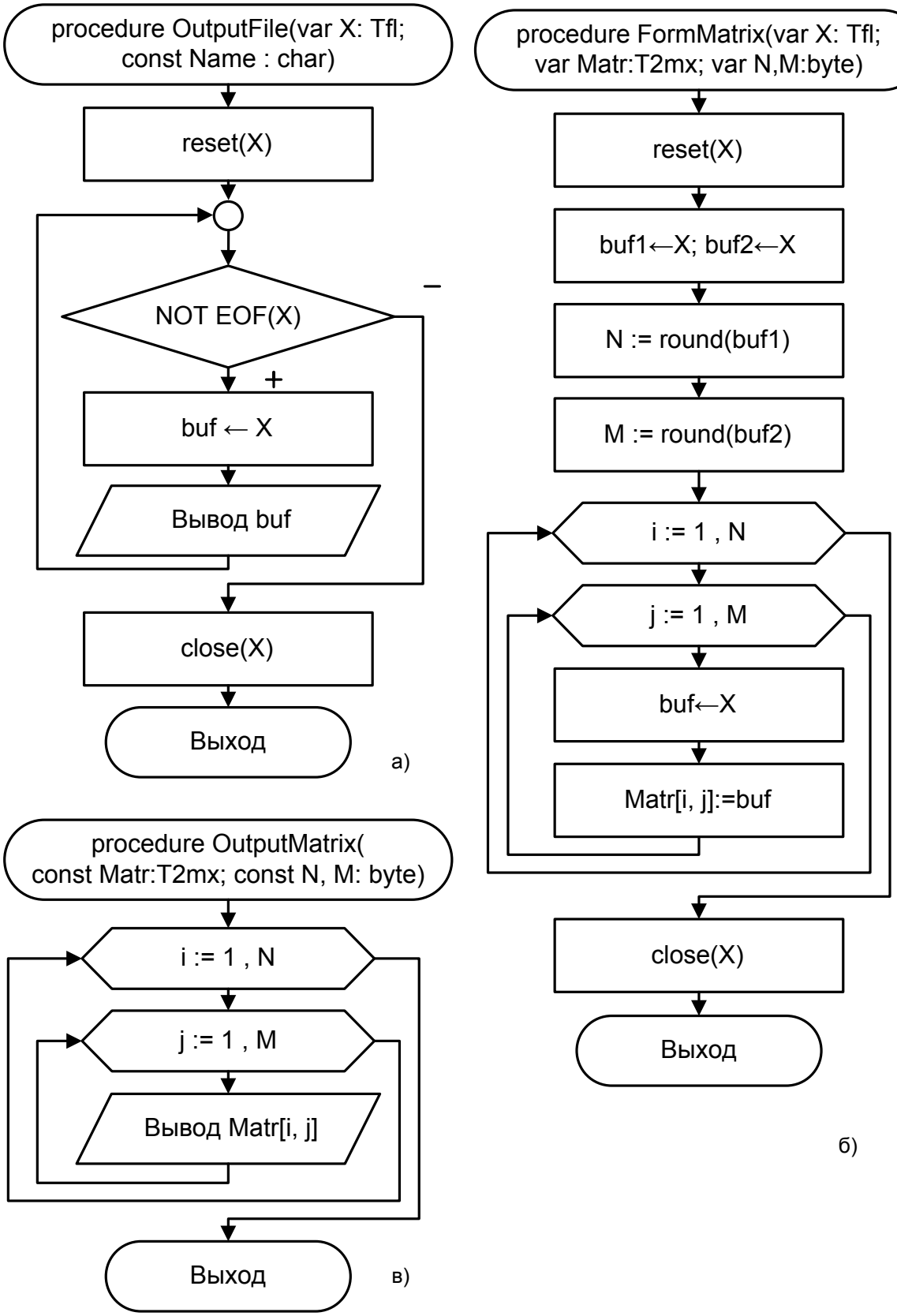


Рисунок 8.21 а) – процедура вывода файла; б) – процедура формирования матрицы из файла; в) – процедура вывода матрицы

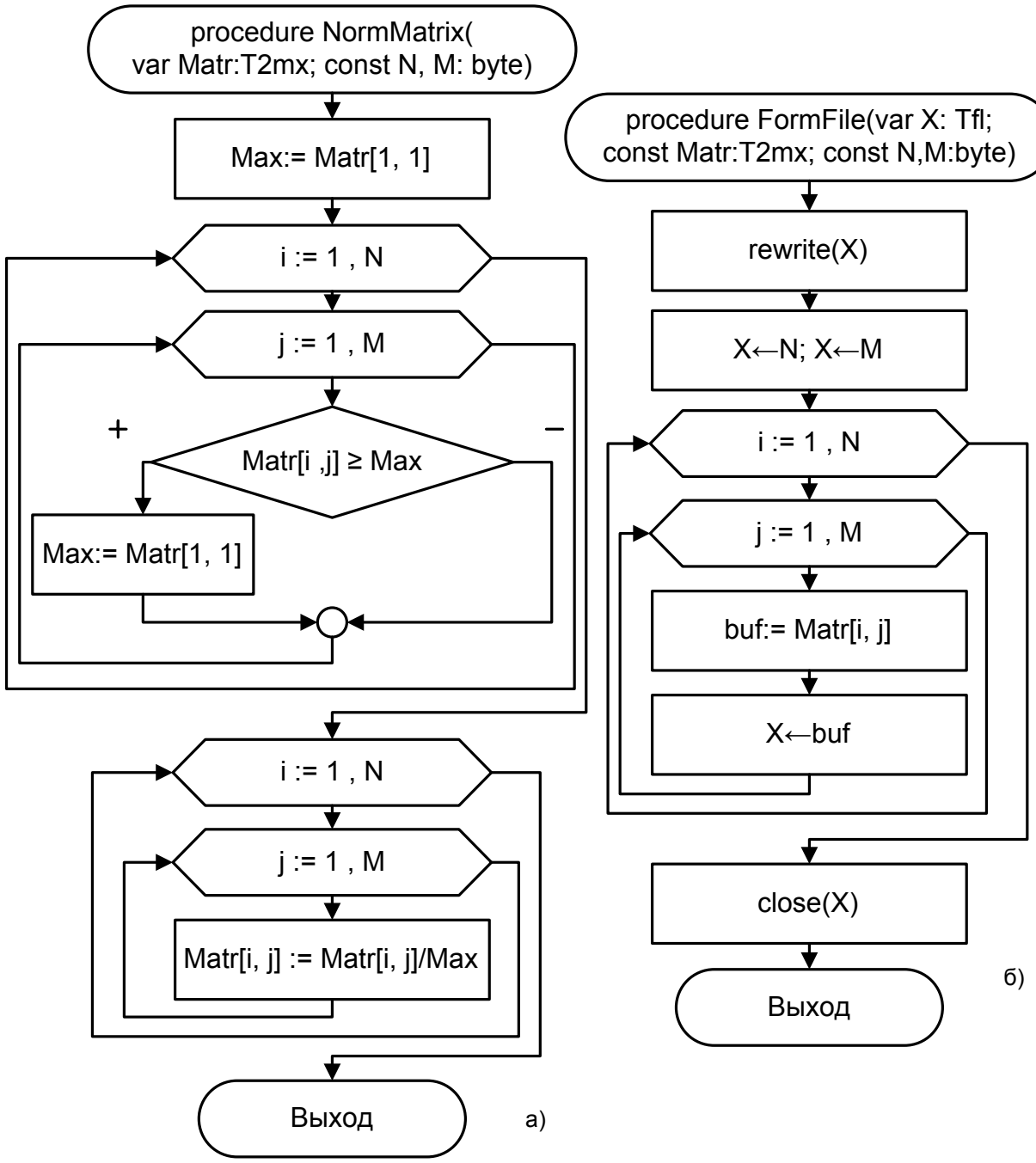


Рисунок 8.22 а) – нормировка матрицы; б) – формирование файла из матрицы

```
{****** 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;
```

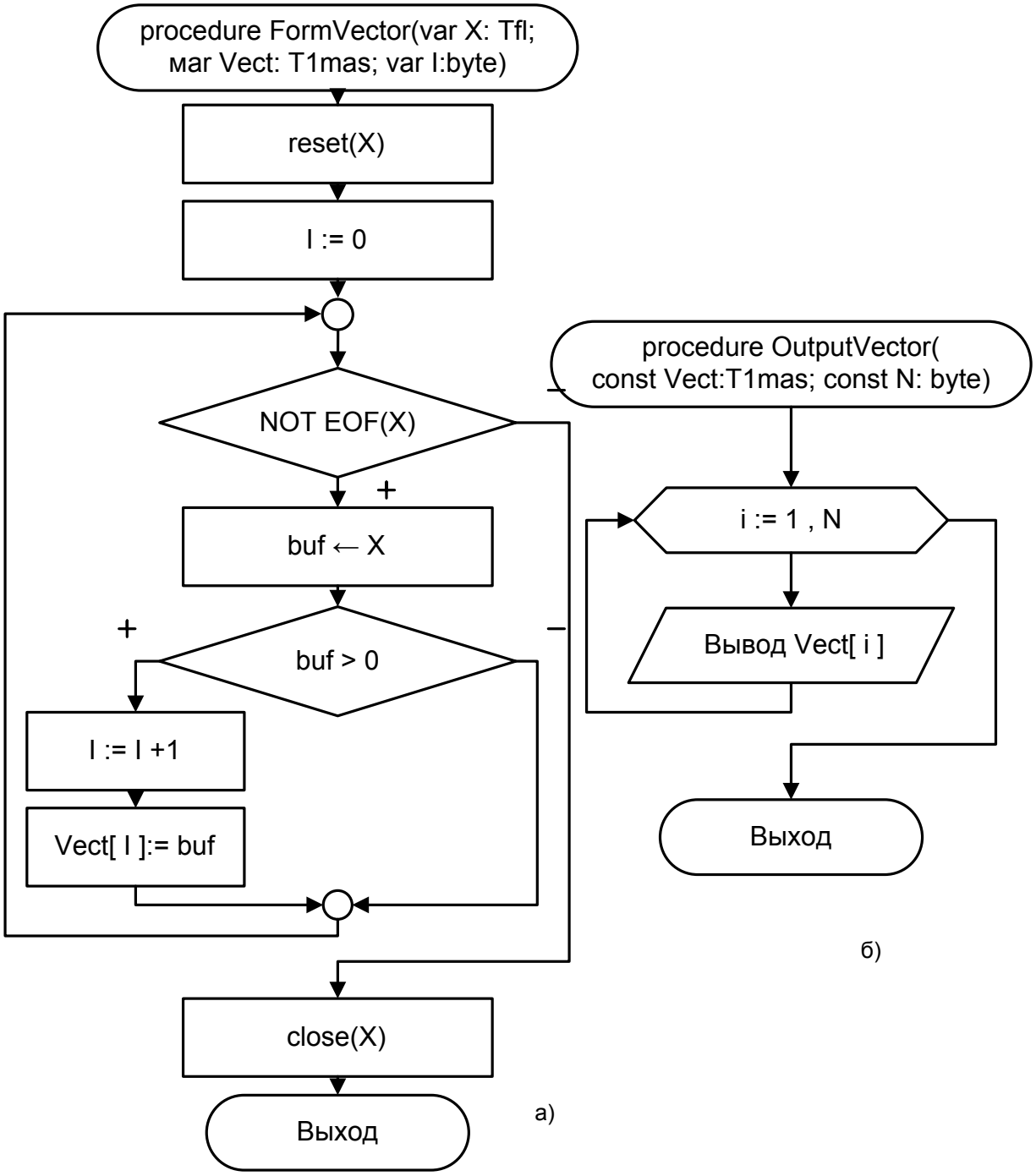


Рисунок 8.23 а) – формирование вектора; б) – вывод вектора

```
{****** 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:= round(buf1);
  M:= round(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.
```

9. ЗАПИСНОЙ ТИП ДАННЫХ. СУБД

9.1 Понятие записи

Часто возникает потребность в обработке информации, которая имеет сложную структуру. Если эта структура устойчива по отношению к различным объектам, которые подвергаются информационной обработке, то принято говорить о формализации свойств объектов. Если удается для нескольких описываемых предметов выделить постоянный набор свойств, то принято говорить о так называемых классах.

Рассмотрим, к примеру, класс «деревья». Очевидно, что в классе деревья экземпляры его представляющие будут обладать определенной общей структурой, несмотря на тот факт, что конкретные свойства у каждого дерева будут индивидуальные. Выделим, скажем, такие свойства, как «продолжительность жизни» и «строение листа».

Возьмем пару экземпляров. Пусть это будут дуб и сосна. У дуба строение листа опишем как «обыкновенное», а у сосны это будет «иголки». Продолжительность жизни дуба 400 лет, продолжительность жизни сосны 300 лет. Можно воспользоваться формальным языком описания сказанного таким образом:

Сосна.ПродолжительностьЖизни = 150;
Дуб.ПродолжительностьЖизни = 300;
Сосна.СтроениеЛиста = «иголки»;
Дуб.СтроениеЛиста = «обыкновенное».

Проблема обработки данных, подобных описанным выше, возникает достаточно часто. Для решения таких задач в современных языках программирования существуют специальные средства. В языке *Pascal* наиболее простым примером этих средств являются записи.

Записи – это сложный (структурированный) тип данных языка *Pascal*, особенностью которого является наличие в собственной структуре так называемых *полей*. Поле у записи можно представить в виде переменной, которая содержит сведения о собственном типе.

Концепция записи наряду с подпрограммами в *Pascal* являются основаниями для, так называемого *объектно-ориентированного программирования* (*ООП*). Следует сказать, что ООП сегодня является наиболее часто встречаемой технологией серьезного программирования.

При обработке доступна как вся запись целиком, так и ее отдельные поля. Поле у записи может быть не только простого типа, но и сложного, в том числе и другая запись (подобные конструкции именуют *вложенными записями*).

Объявление переменной-записи может происходить как напрямую, так и при помощи вспомогательного типа (через раздел ***type***). Ключевым словом при объявлении записи является слово ***record***, после которого идет список

названий полей и их типов. В конце ставится закрывающая операторная скобка ***end***, причем, следует отметить, отсутствие открывающей скобки ***begin***.

Пример объявления записи:

```
var SSS: record;
           aaa, bbb: byte;
           xxx: string[20];
           ddd: real;
           iii, jjj: integer;
           ttt: boolean;
    end;
```

Нетрудно подсчитать объем занимаемый переменной ***SSS***: 1 (***aaa:byte***) + 1 (***bbb:byte***) + 21 (***xxx:string[20]***) + 6 (***ddd:real***) + 2 (***iii:byte***)+ 2 (***jjj:byte***) + 1 (***ttt: Boolean***) = 34 байта.

Если бы мы использовали простой массив с элементами длиной равной максимальной из возможных полей данной записи (***xxx : string[20]***), то мы бы получили что один компонент массива занимает 7*21 = 147 байт, что в несколько раз больше чем мы насчитали ранее. Таким образом, использование записей дает преимущество перед массивами в экономии памяти. Следующим преимуществом является структурированность данных. Это, к примеру, означает, что при попытке ввода в поле числового типа строковых данных возникнет исключительная ситуация, гласящая об ошибке работы с программой. Давая отражающие суть имена полям записи, мы тем самым упрощаем работу с ними при программировании.

Собственно, обращение к полю записи происходит при помощи упоминания имени переменной и через точку название поля.

Вот примеры:

```
X.Y;
SSS.aaa;
Human.FIO.Name;
Human.FIO.Surname;
School[2].Klass.Uchenik[i];
```

Данные примеры показывают как обращение к простым полям, так и к сложным (вложенным). Последняя строчка представляет вариант использования в записях массивов.

Кроме прямого обращение к объекту записного типа, часто используется обращение через оператор присоединения пространства имен ***with*** *Запись* ***do***.

Пусть нам нужно обработать запись объявленную ранее с помощью переменной ***SSS***, тогда мы сможем запрограммировать следующее:

```
With SSS do
  begin
    bbb:= bbb+ aaa;
    xxx:= 'hello world';
    WriteLn(xxx);
    ttt:= 2<>2;
  end;
```

Видно, что использование оператора присоединения избавило нас от необходимости каждый раз писать имя переменной ***SSS***, к полям которой мы обращаемся. Это делает программный код более выразительным и наглядным.

9.2 Концепция БД

Если взять несколько переменных одинакового записного типа и начать с ними работать, то получится элементарная база данных.

База данных (***БД***) – это упорядоченная совокупность данных (и их описание), предназначенных для хранения, накопления и обработки с помощью ЭВМ.

Для создания и ведения базы данных (обновления, обеспечения доступа к ним по запросам и выдачи их пользователю) используется набор языковых и программных средств, называемых ***системой управления базы данных*** (***СУБД***).

Наиболее естественное для понимания представление БД – это табличное. Самая простая БД – это БД состоящая из одной таблицы. Если таблиц больше чем одна и между ними установлены определенные связи, то такую БД называют реляционной (*relation – англ. зависимость*).

Рассмотрим пример реализации однотабличной БД. Пусть, например, таблица содержит описание магазинов (Таблица 9.1).

Таблица 9.1 Описание магазинов

массив Base	Реестр (Reestr-Number)	Название (ShopName)	Адрес (Adress)		Профиль магазина (SaleProfile)	Площадь, кв. м (Area)
			Улица (Street)	Дом (Number-OfBld)		
Base[1]	1	Лимпопо	Зеленая	45	Детский	320
Base[2]	2	Хозяин	Продоль-ная	12	Хозяйственный	140
Base[3]	3	Свежий Хлеб	Продоль-ная	18	Продуктовый	25
Base[4]	4	Продукты	Ленина	123	Продуктовый	60
Base[5]	5	Все для ремонта	Пушкина	2	Хозяйственный	75

С точки зрения структуры записи удобно пользоваться иерархической диаграммой, изображенной на (Рисунок 9.1). Из диаграммы (как впрочем и из таблицы) видно что поле «Адрес» имеет сложную структуру (вложенная запись). Оно содержит поля «Улица» и «Дом». Здесь мы имеем дело с записным типом данных с пятью полями, одно из которых тоже записное, состоящее из двух подполей.

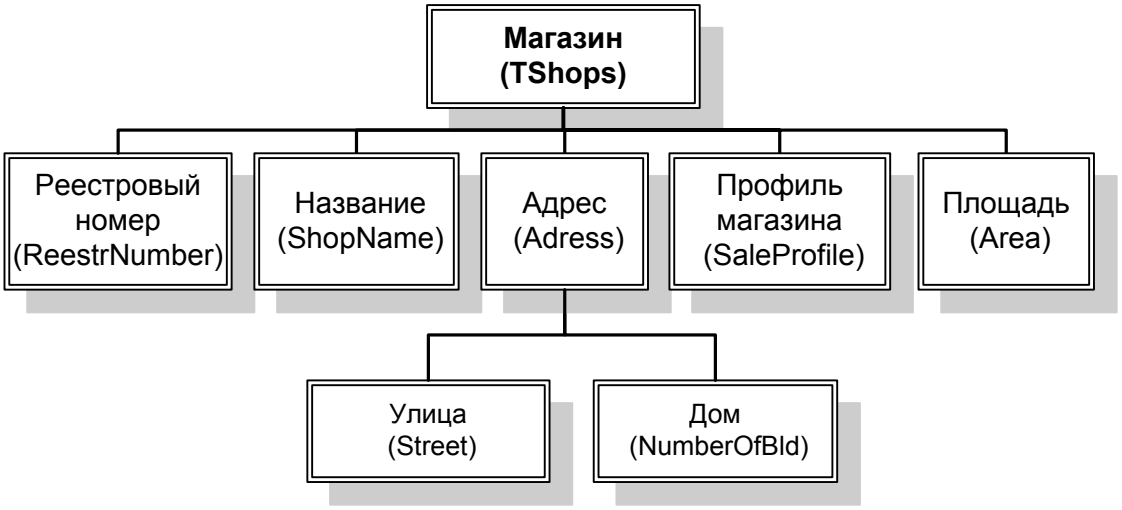


Рисунок 9.1 Структура записи TShops, предназначенной для хранения информации о магазине

Важно понимать отличия Таблица 9.1 и диаграммы Рисунок 9.1. Суть этих отличий в том, что диаграмма содержит лишь структуру записи, а таблица уже конкретные примеры этой структуры. Выражаясь другим языком, можно сказать, что диаграмма – это тип переменной, а таблица – это множество значений переменной данного типа.

ПРИМЕР

На основе предложенной структуры сведений о магазине можно написать простую систему управления этими сведениями. *К основным операциям управления отнесем операции создания описания магазинов, вывод таблицы со всеми сведениями на экран и выдачу информации о магазинах, удовлетворяющих определенным требованиям. Таковыми требованиями пусть будут определенный профиль магазина и площадь торгового зала, превышающая среднюю арифметическую площадей всех магазинов находящихся на данный момент в таблице*.

Для начала в качестве инструмента хранения всех наших данных возьмем массив, т.е. напишем программу используя массив записей. Это означает, что каждый раз при запуске нам придется вводить все сведения о магазинах заново, ввиду того, что массивы хранятся в энергозависимой памяти и уничтожаются вместе с прекращением работы программы.

```
program subd;
type
    TShops = record
      ReestrNumber: integer;
      ShopName: string[10];
      Adress : record
          Street : string[15];
          NumberOfBld : byte;
        end; {Adress}
      SaleProfile : string[10];
      Area : integer;
    end; {TShops}
    TBaseMass = array[1..255] of TShops;
```

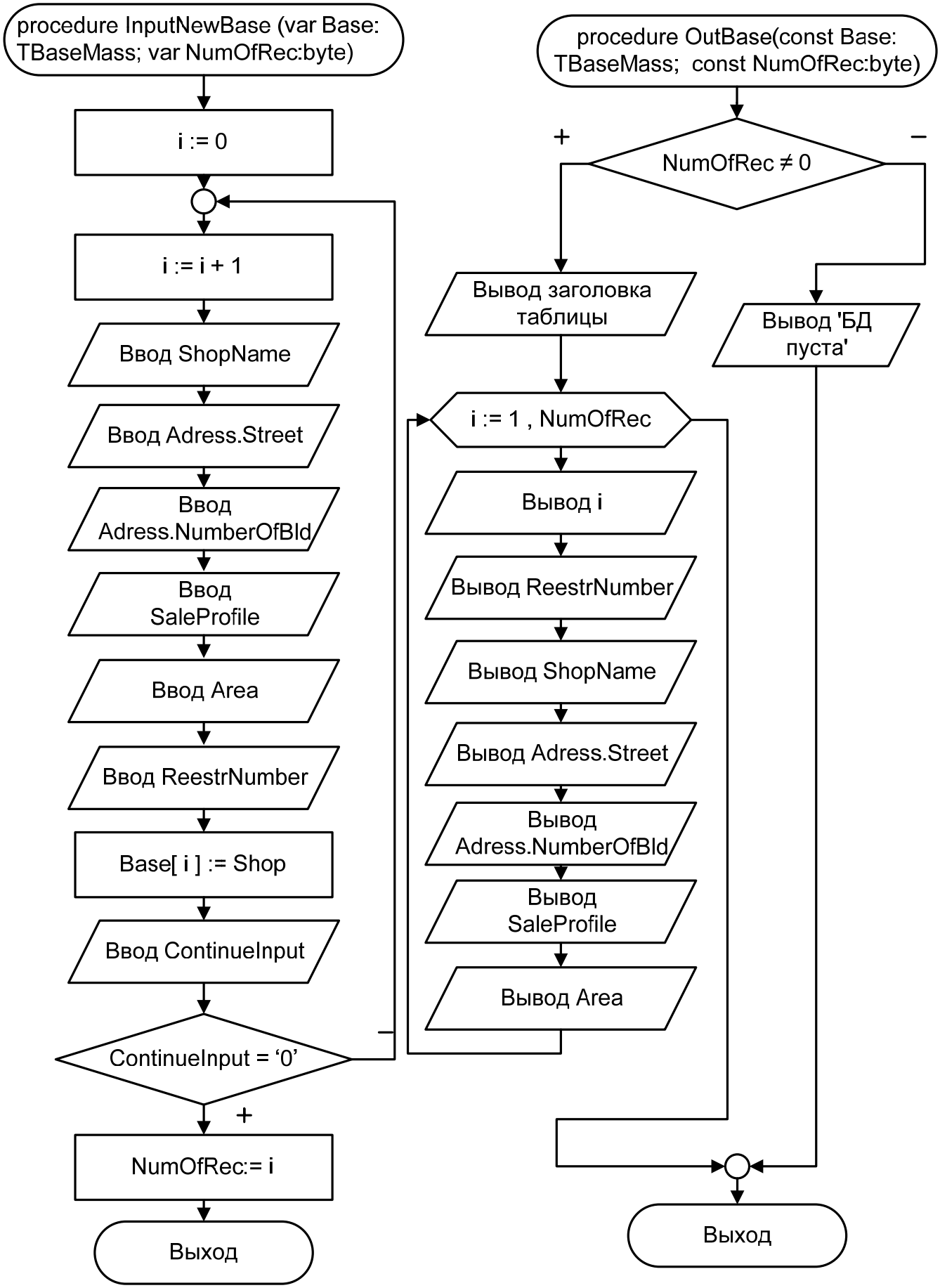


Рисунок 9.2 Блок-схемы процедур ввода (а)) и вывода (б)) массива записного типа

Здесь приведен пример программного объявления типа записи изображенной на Рисунок 9.1. Далее идет объявление типа массива записей (***TBaseMass***) состоящего из ***TShops***. В Таблица 9.1 показана возможная реализация массива описанного типа с именем ***Base***. Соответственно, ***Base[i]*** – отдельная компонента, включающая описание ***i***-го магазина.

Далее идет объявление и описание процедуры для формирования нового массива записей ***InputNewBase***. В качестве формальных параметров, как обычно у процедур для обработки массивов, выступает собственно сам массив и число компонент в нем. Блок-схема процедуры изображена на (Рисунок 9.2) а).

```
{процедура формирования новой таблицы}
procedure InputNewBase(var Base:TBaseMass;
          var NumOfRec:byte);
var i: byte;
    Shop: TShops;
    ContinueInput: char;
begin
  writeLn;
  i:=0;
  repeat
    i:=i + 1;
    with Shop do
      begin
        write('Введите название ',i,'-го магазина: ');
        readLn(ShopName);
        writeLn('Введите адрес магазина ',ShopName);
        write('      Улица: ');
        readLn(Adress.Street);
        write('        Дом: ');
        readLn(Adress.NumberOfBld);
        write('Профиль магазина ',ShopName,': ');
        readLn(SaleProfile);
        write('Площадь торговых площадей магазина
              ',ShopName,': ');
        readLn(Area);
        write('Реестровый номер магазина ',
               ShopName,': ');
        readLn(ReestrNumber);
      end;
    Base[i] := Shop;
    writeLn('Вводим данные о следующем магазине');
    writeLn('Если ввод окончен, то введите ''0''');
    readLn(ContinueInput);
  until ContinueInput='0';
  NumOfRec:= i;
end;
```

Аналогично можно написать процедуру вывода элементов массива записей. Здесь стоит обратить внимание на использование оператора присоединения ***with*** *объект* ***do*** к пространству имен массива ***Base[i]***. Это дает возможность обращаться к полям записи напрямую, минуя префикс "***Base[i]***.". Блок-схема этой процедуры представлена на (Рисунок 9.2) б).

```
{процедура вывода базы на экран}
procedure OutBase(const Base:TBaseMass;
          const NumOfRec:byte);
var i,j: byte;
begin
  writeLn;
  if NumOfRec <> 0 then
    begin
      writeLn('База данных содержит следующие
               сведения:');
      for j:=1 to 79 do
      write('-');
      writeLn;
      writeLn('N п/п':6,'| ','реестр. N |':11,
              'Название |':12,'Адрес.Улица |':17,
              'Дом |':7,'Торг. Профиль |':15,
              'Площадь |':9);
      for j:=1 to 79 do
      write('=');
      writeLn;
      for i:=1 to NumOfRec do
        with Base[i] do
          begin
            write(i:5,' |');
            write(ReestrNumber:10,' |');
            write(ShopName:10,' |');
            write(Adress.Street:15,' |');
            write(Adress.NumberOfBld:5,' |');
            write(SaleProfile:13,' |');
            writeLn(Area:7,' |');
            for j:=1 to 79 do
              write('-');
            writeLn;
        end;
      end
   else
    writeLn('БД пуста');
end;
```

Ниже представлена процедура, выдающая по запросу все магазины определенного профиля, площадь которых больше чем среднее арифметическое площадей всех магазинов данного профиля. Вначале идет подсчет среднего арифметического площадей магазинов удовлетворяющих

введенному профилю, а далее, собственно, селективный вывод на экран. Блок-схема процедуры изображена на (Рисунок 9.3).

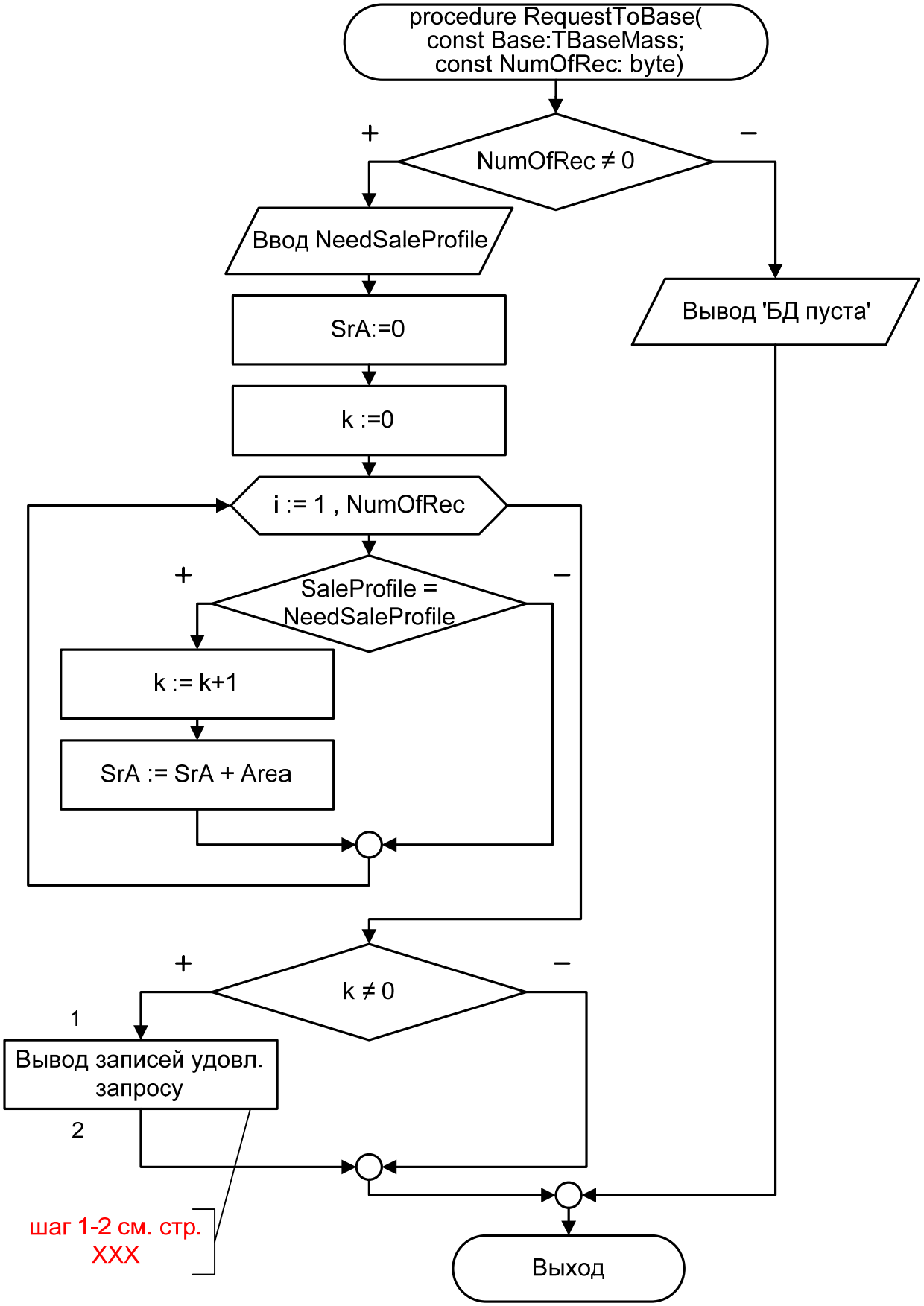


Рисунок 9.3 Процедура запроса

Шаг 1-2 (стр. XXX) Вывод записей удовл. запросу

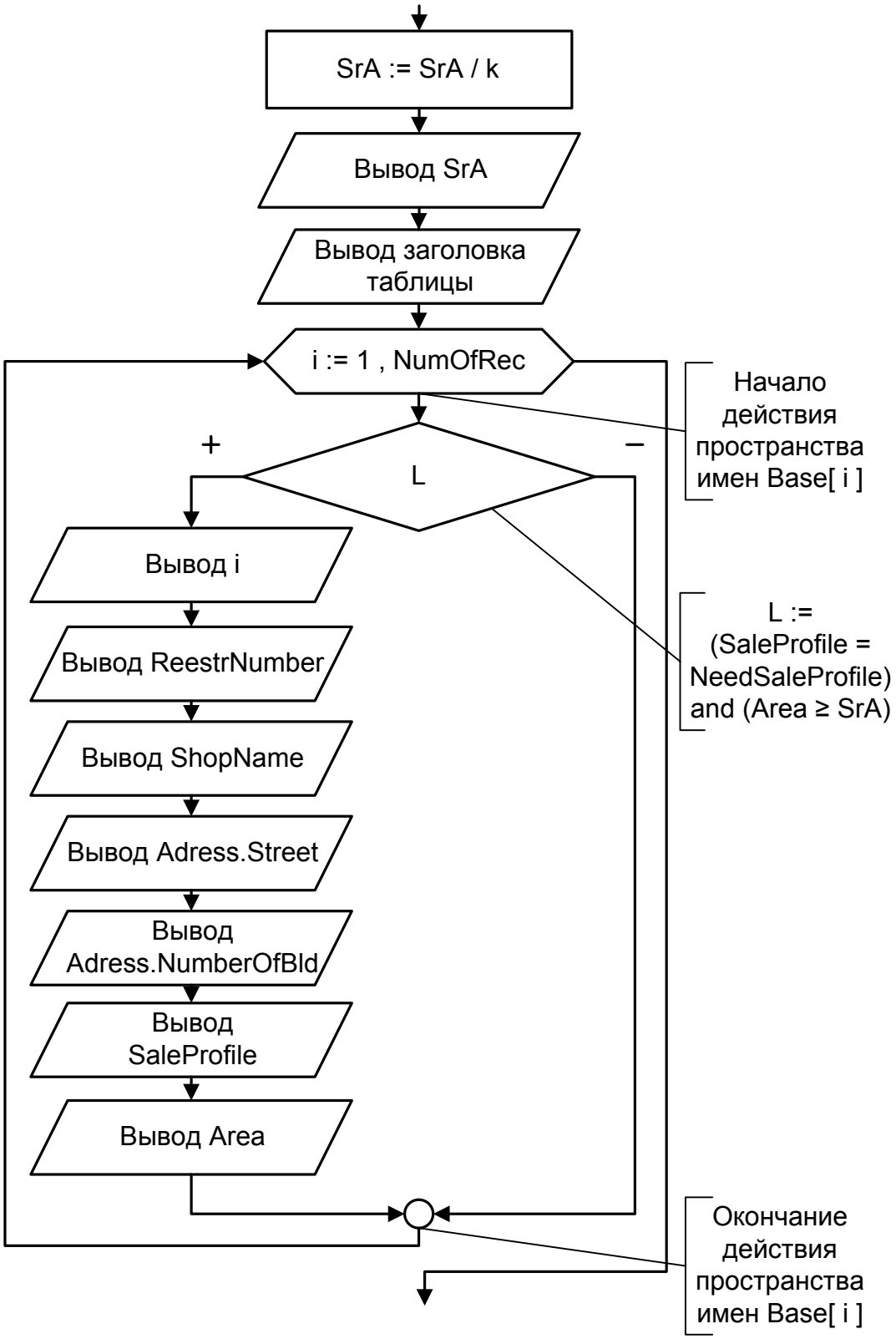


Рисунок 9.4 Детализация вывода к процедуре запроса

```
{Процедура обработки запроса к БД
 (поиск всех магазинов опр. профиля
 с площадью выше среднего)}
procedure RequestToBase(const Base:TBaseMass;
```

```
          const NumOfRec: byte);
var i,j,k: byte;
    NeedSaleProfile: string[10];
    SrA: real;
begin
  if NumOfRec <> 0 then
    begin
      writeLn;
      writeLn('Обработка запроса: Все магазины
               с площадью >= ср. арифметического');
      writeLn;
      write('Введите профиль магазина:');
      readln (NeedSaleProfile);
      SrA:= 0;
      k:=0;
      for i:= 1 to NumOfRec do
        with Base[i] do
          begin
            If SaleProfile = NeedSaleProfile then
              begin
                k:= k + 1;
                SrA:= SrA + Area;
              end;
          end;

      if k<>0 then
        begin
          SrA := SrA / k;
          writeLn('SrA = ', SrA:6:2);
          writeLn('Вот результат запроса:');
          for j:=1 to 79 do
          write('-');
          writeLn;
          writeLn('N п/п':6,'| ','реестр. N |':11,
                  'Название |':12,'Адрес.Улица |':17,
                  'Дом |':7,'Торг. Профиль |':15,
             'Площадь |':9);
          for j:=1 to 79 do
          write('=');
          writeLn;
          for i:=1 to NumOfRec do
            with Base[i] do
            if (SaleProfile = NeedSaleProfile)
                and (Area >= SrA) then
              begin
                write(i:5,' |');
                write(ReestrNumber:10,' |');
                write(ShopName:10,' |');
                write(Adress.Street:15,' |');
```

```
                    write(Adress.NumberOfBld:5,' |');
                    write(SaleProfile:13,' |');
                    writeLn(Area:7,' |');
                    for j:=1 to 79 do
                      write('-');
                    writeLn;
                end;
            end
          else
             WriteLn('В базе нет магазинов профиля ',
                      NeedSaleProfile);
      end
    else
      WriteLn('База данных пуста');
end;
```

Процедура вывода главного меню программы представлена реализацией оператора ***case***, при помощи которого происходит выбор нужного действия. При вводе той или иной цифры происходит вызов соответствующей процедуры. Практически весь код процедуры помещен в цикл. Выход из этого цикла происходит после ввода цифры 4.

```
{процедура выводящая главное меню программы}
procedure MainMenu;
var PunktOfMenu: byte;
    Base: TBaseMass;
    NumOfRec : byte;
begin
  NumOfRec := 0;
  repeat
    writeLn;
    writeLn('Выберите нужное действие:');
    writeLn('1 - Ввод новой БД');
    writeLn('2 - Вывод БД');
    writeLn('3 - Запрос к БД');
    writeLn('4 - Выход');
    write('Ваш выбор:');
    readLn(PunktOfMenu);
    case PunktOfMenu of
      1: InputNewBase(Base, NumOfRec);
      2: OutBase(Base, NumOfRec);
      3: RequestToBase(Base, NumOfRec);
      else
        if PunktOfMenu<>4 then
          writeLn('Введите корректный пункт меню');
    end; {конец case}
  until PunktOfMenu = 4 ;
end;
```

```
{очень короткая главная программа}
begin
   MainMenu;
end.
```

Блок схема процедуры ***MainMenu*** и основной программы изображены на Рисунок 9.5 а) и Рисунок 9.5 б) соответственно. Видно как проста основная программа, которая содержит лишь единственный вызов процедуры и все. Это является следствием проявления процедурно-модульного метода программирования.

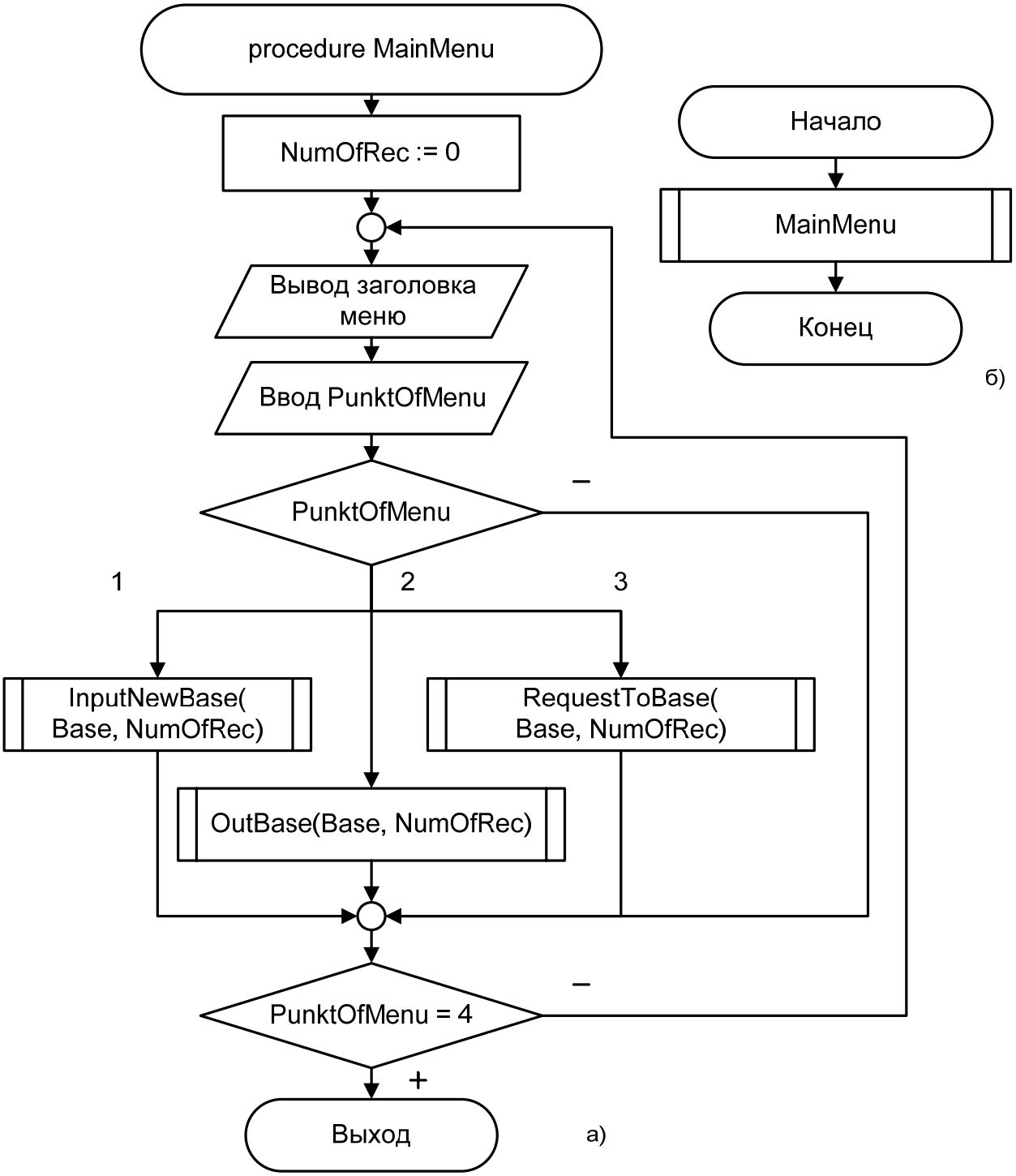


Рисунок 9.5 Блок-схемы MainMenu а) и главной программы б)

9.3 Пример программы реализующей файлы записей

Если ранее описывался пример работы с массивом записей, то теперь есть смысл рассмотреть аналогичную задачу с использованием файлов записного типа. Сам текст несколько усложним. Итак,

ЗАДАЧА

Создать систему управления базой данных (СУБД) в которой предусмотреть следующие режимы: создание БД, вывод БД, добавление записей в БД, удаление записей из БД, корректировка сведений в БД, печать сведений из базы по запросу, выход из БД. Все режимы должны представлены в виде функционального меню. Режим печати по запросу должен предусматривать подменю выбора запросов. Тестирование СУБД осуществить на БД регистрации новорожденных.

В бюро ведется учет всех регистраций новорожденных. О каждом новорожденном имеются следующие сведения: фамилия, имя, отчество, дата рождения, место рождения, дата регистрации, номер записи в книге регистрации, сведения о родителях: отце и матери (фамилия, имя, отчество, национальность), дата выдачи и номер свидетельства о рождении.

В СУБД предусмотреть следующие запросы:

- список новорожденных, отсортированный по датам рождения;

- список новорожденных, зарегистрированных в определенный день;

- данные о ребенке по данным о родителях.

Структура записи «Новорожденный» (***Trec***) изображена на Рисунок 9.6.

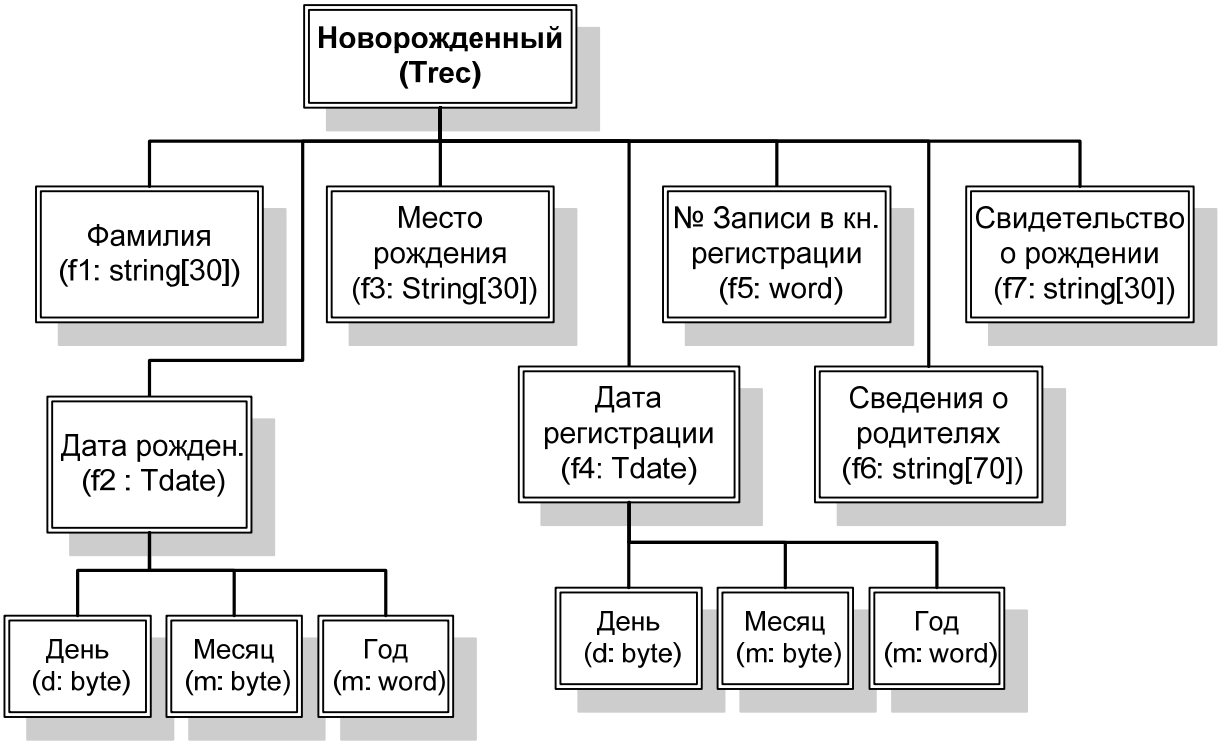


Рисунок 9.6 Структура записи "Новорожденный"

Без подробных комментариев, напишем программу, которая является одним из решений описанной задачи. Решение, конечно, не самое оптимальное и не самое красивое. Просто цель у приведенного алгоритма и кода на языке *Pascal* состоит в демонстрации принципиальной возможности осуществить выполнение задания средствами компонентных файлов.

Здесь мы используем функциональное меню, которое активно взаимодействует с человеком работающим с нашей базой. Реализация интерфейса «*Пользователь-СУБД*» зависит от конкретной ситуации, от возможностей операционной системы и от типа среды разработки. Поскольку реализация велась для *Turbo Pascal 7.0*, то и взаимодействие происходит в режиме доступном для *TP 7.0*, т.е. либо в *MS DOS*, либо в режиме эмуляции *MS DOS* в ОС *Windows*. Основной особенностью написанной ниже программы является активное использование средств стандартной библиотеки *CRT* для работы с экраном. Потому гарантии того, что приведенный код будет работать без доработок в иной, отличной от *TP 7.0* среде, нет. Все комментарии к программе выполнены в тексте в фигурных скобках («*{…}*») согласно синтаксису языка *Turbo Pascal 7.0*. Блок-схемы для получившейся программы показаны на рисунках 8.7–8.17.

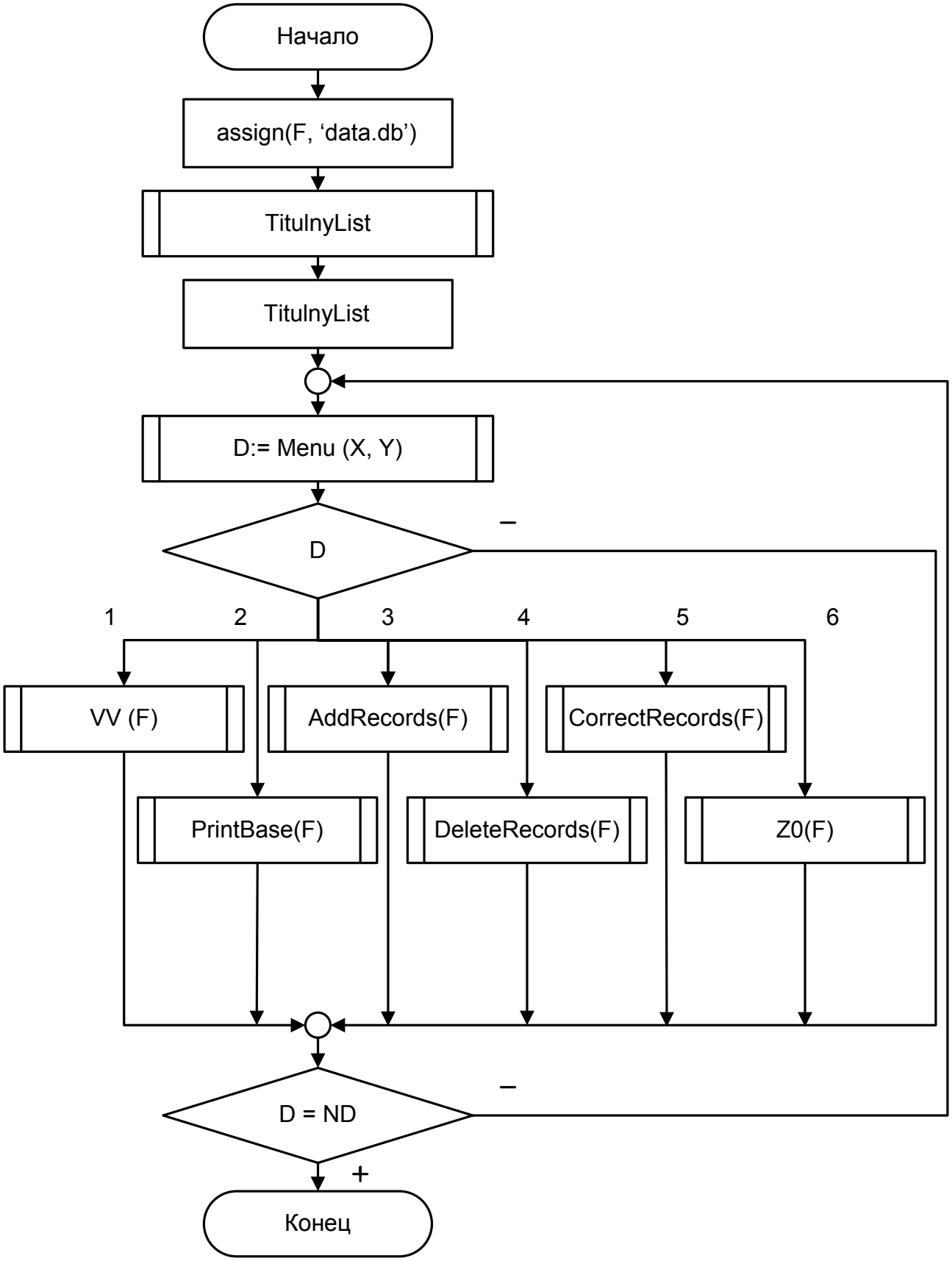


Рисунок 9.7 Основная программа

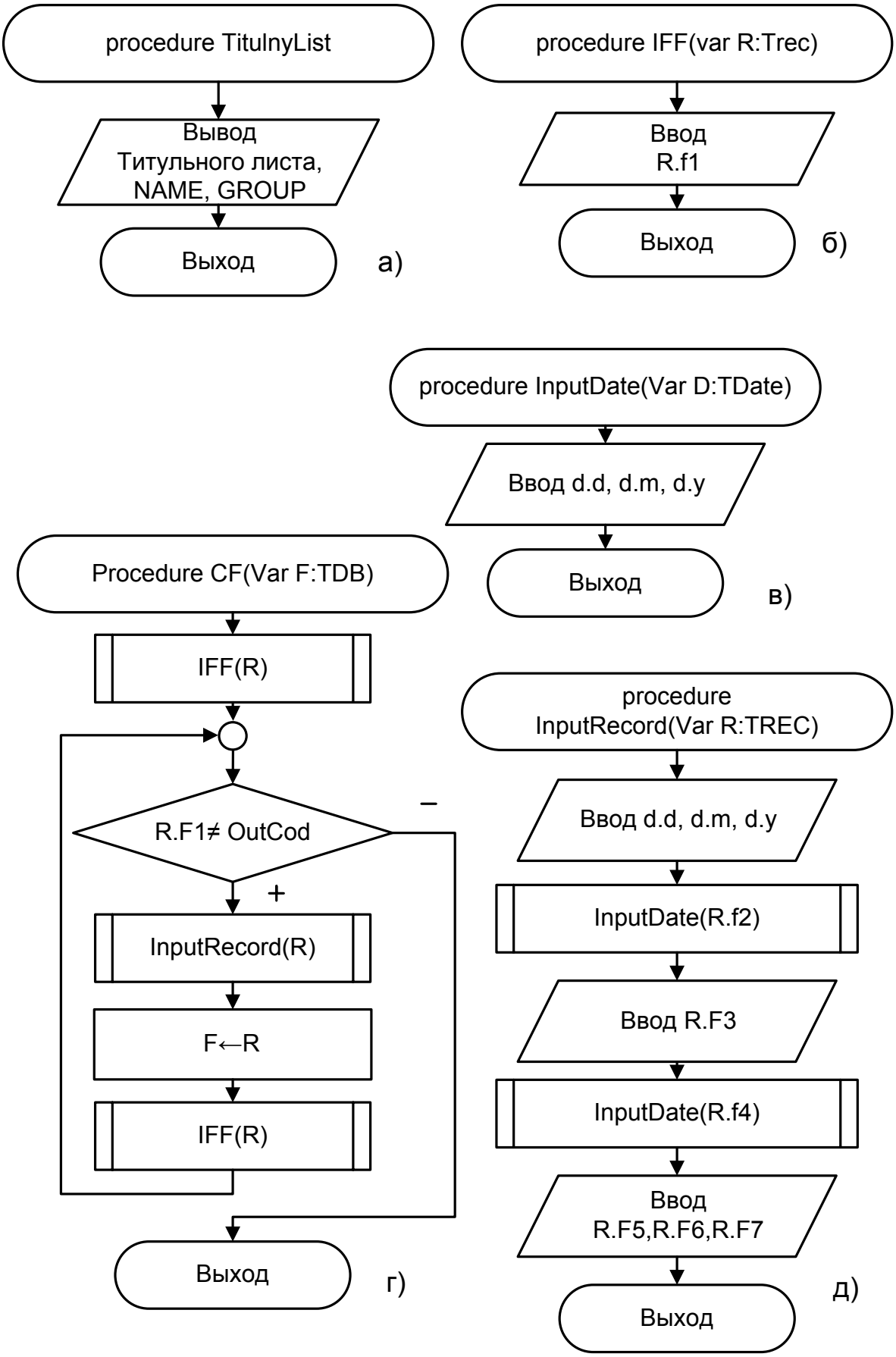


Рисунок 9.8 а) – титульный лист; б) – вывод заголовка при вводе записи; в) – ввод даты; г) – вызов ввода записи и сохранение ее в файл; д) – ввод записи

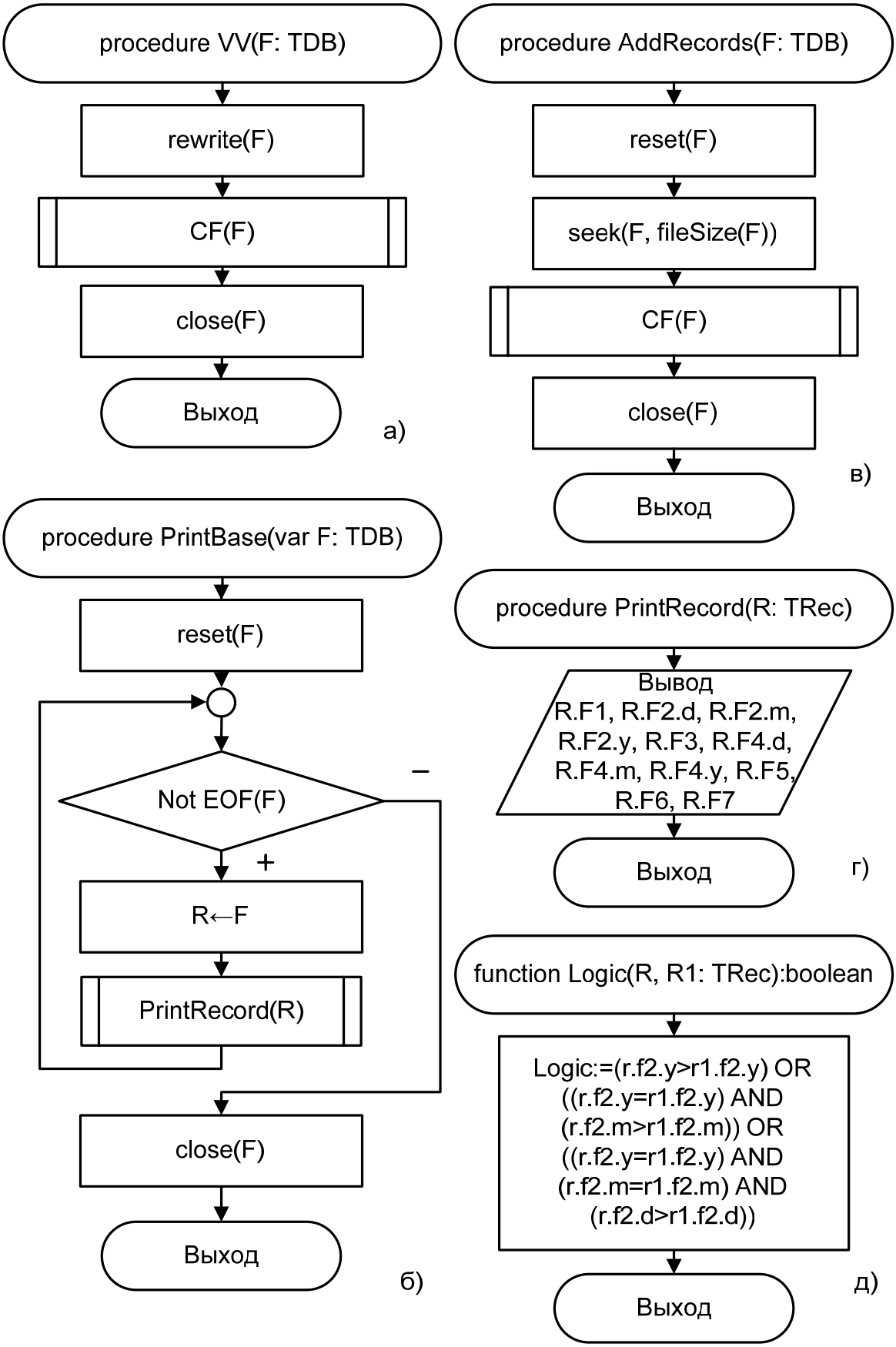


Рисунок 9.9 а) – создание файла и вызов его заполнения б) – чтение файла и вывод базы на экран; в) – добавление записи в файл; г) – вывод экземпляра записи; д) – логическая функция сравнения полей пары переменных

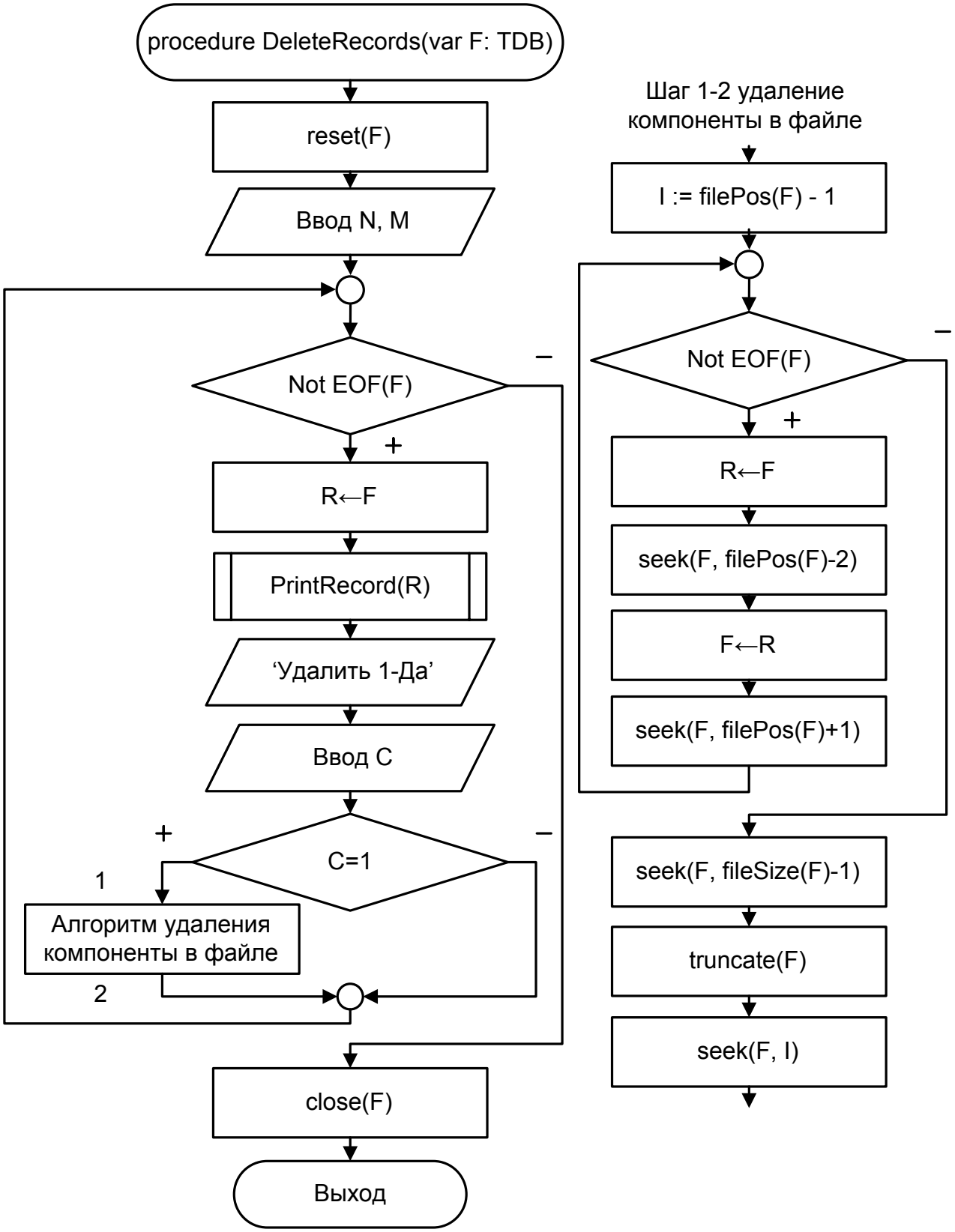


Рисунок 9.10 Удаление записи

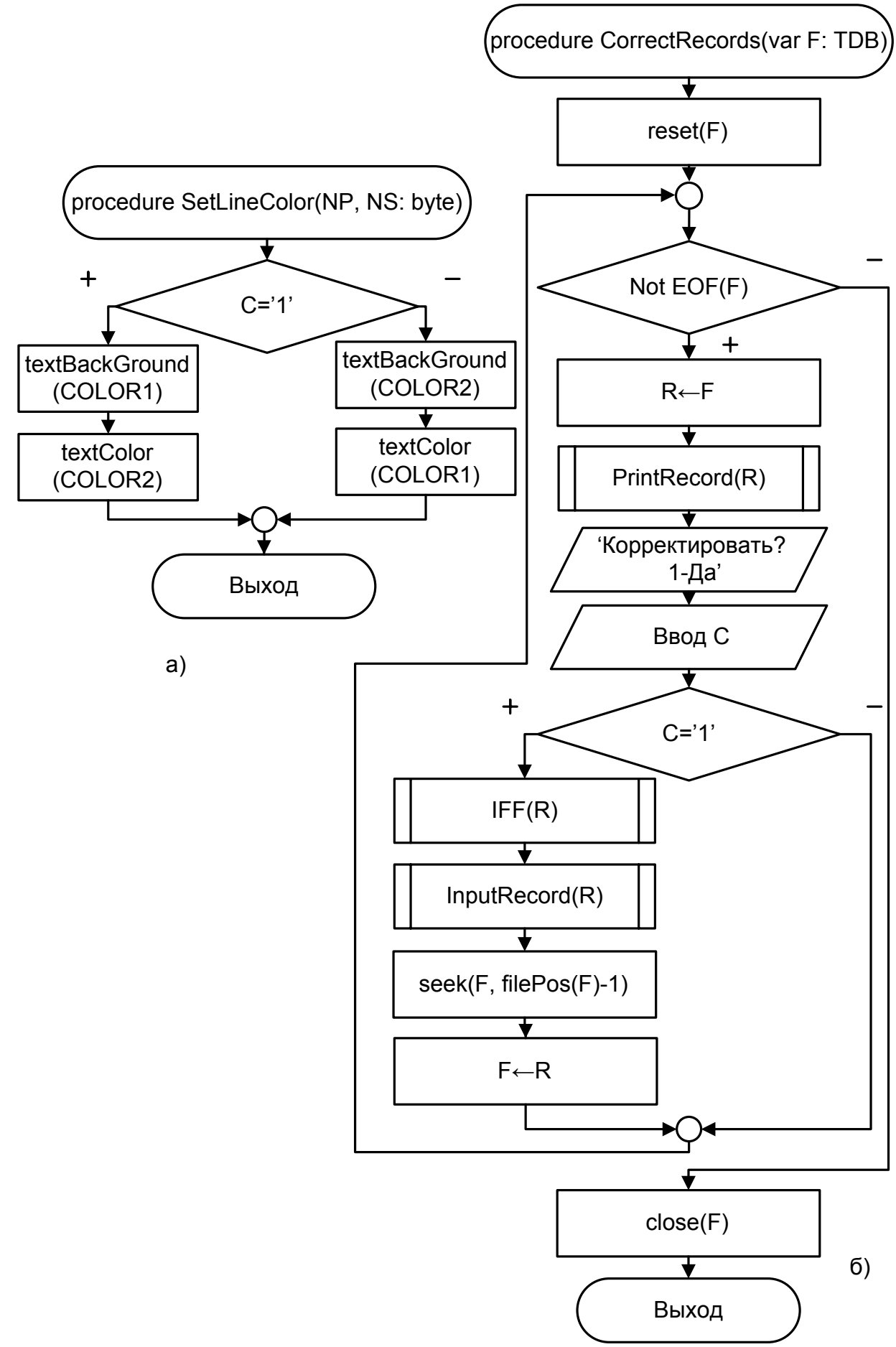


Рисунок 9.11 а) – установка цвета пункта меню; б) – корректировка записи

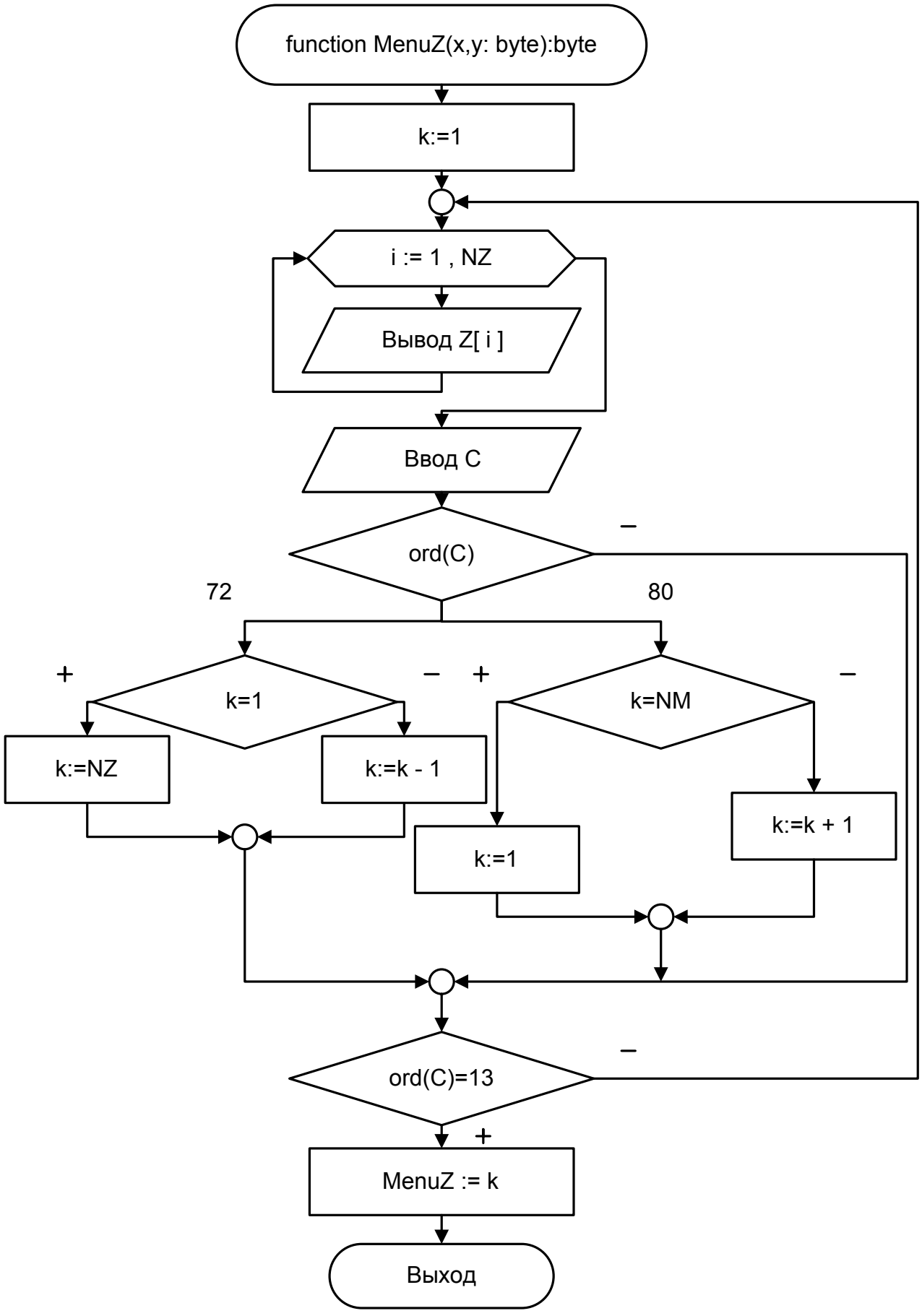


Рисунок 9.12 Функция меню запроса

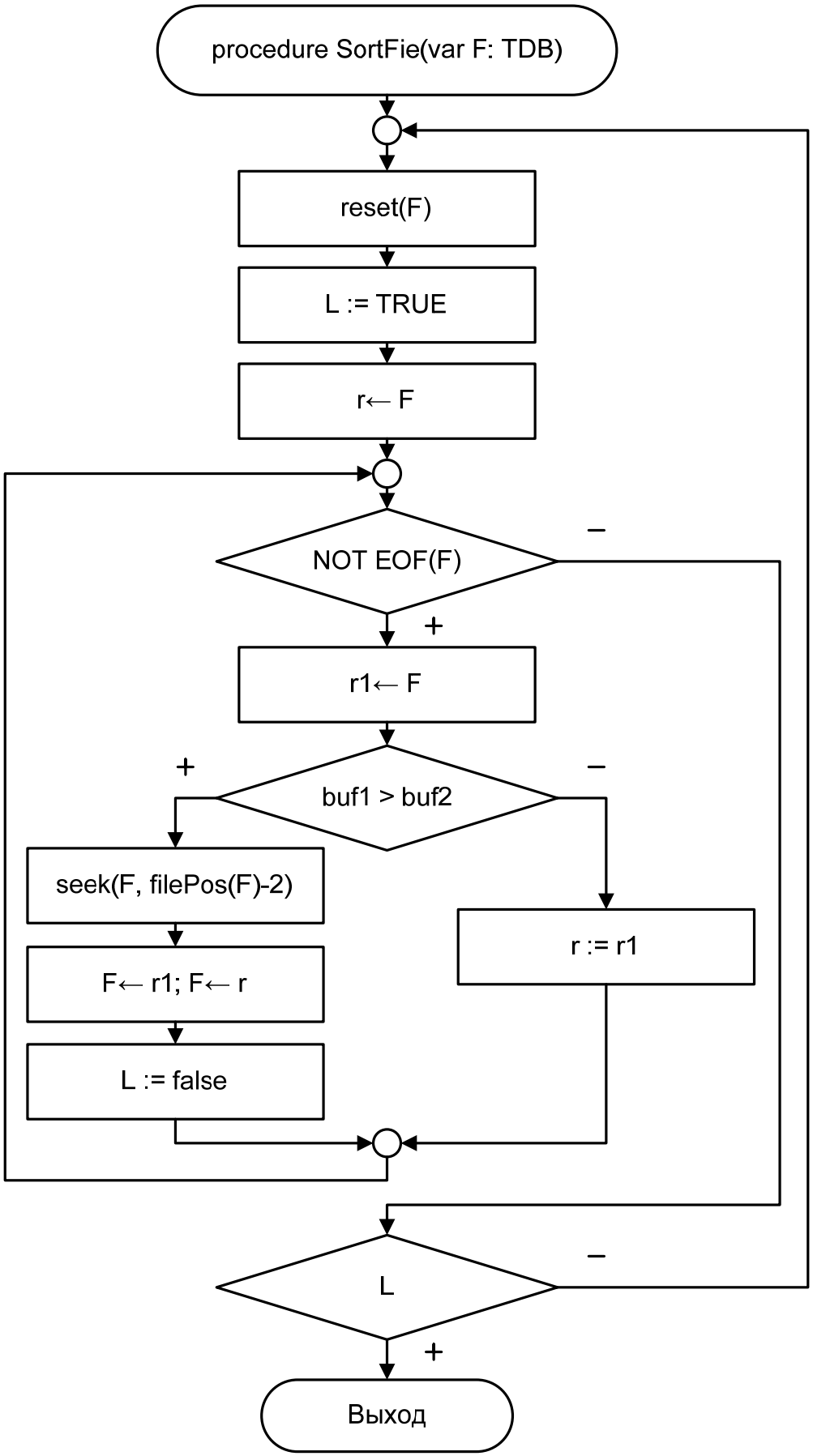


Рисунок 9.13 Сортировка файла

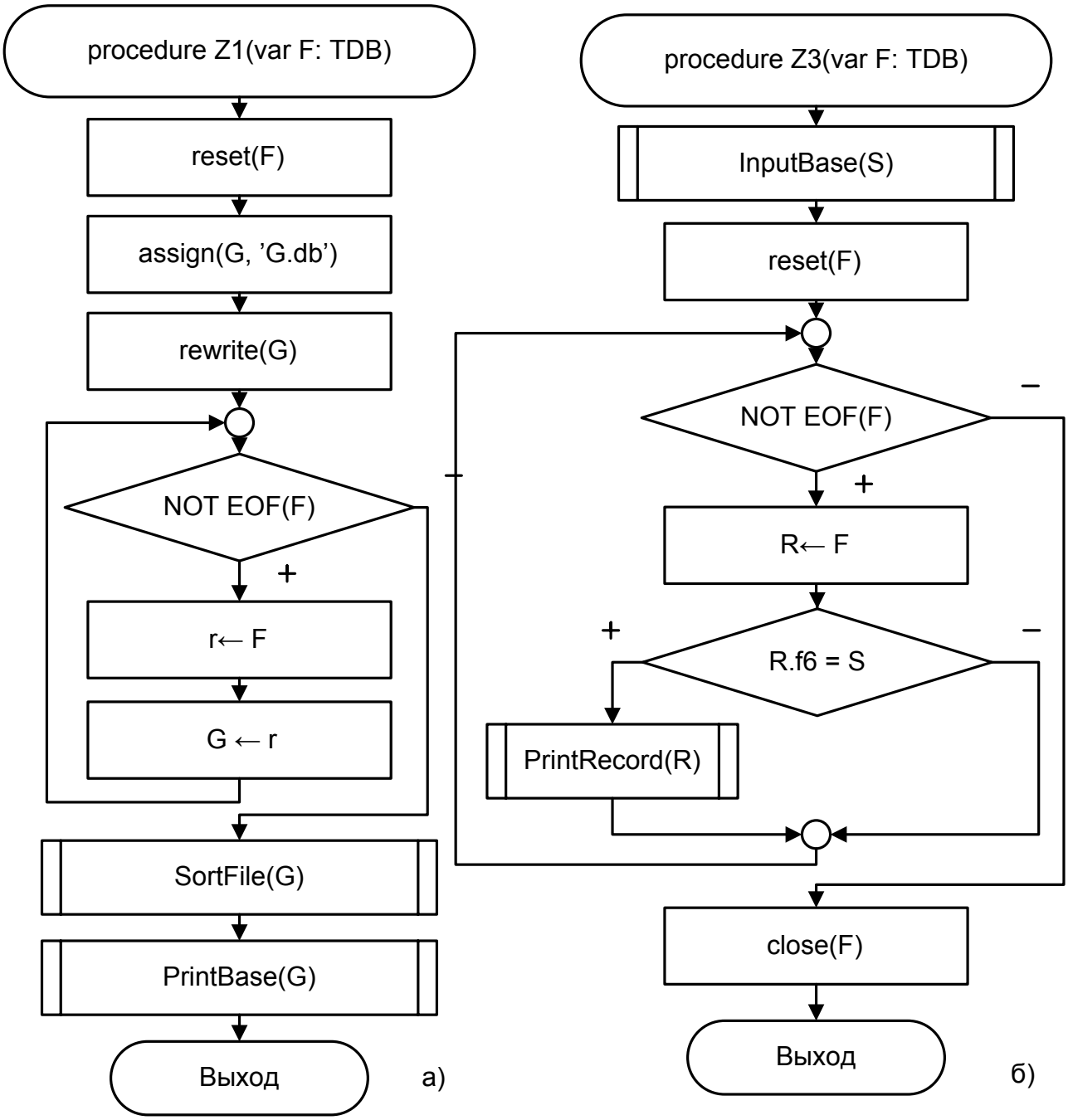


Рисунок 9.14 а) – формирование нового файла; б) – вывод записей из файла по запросу

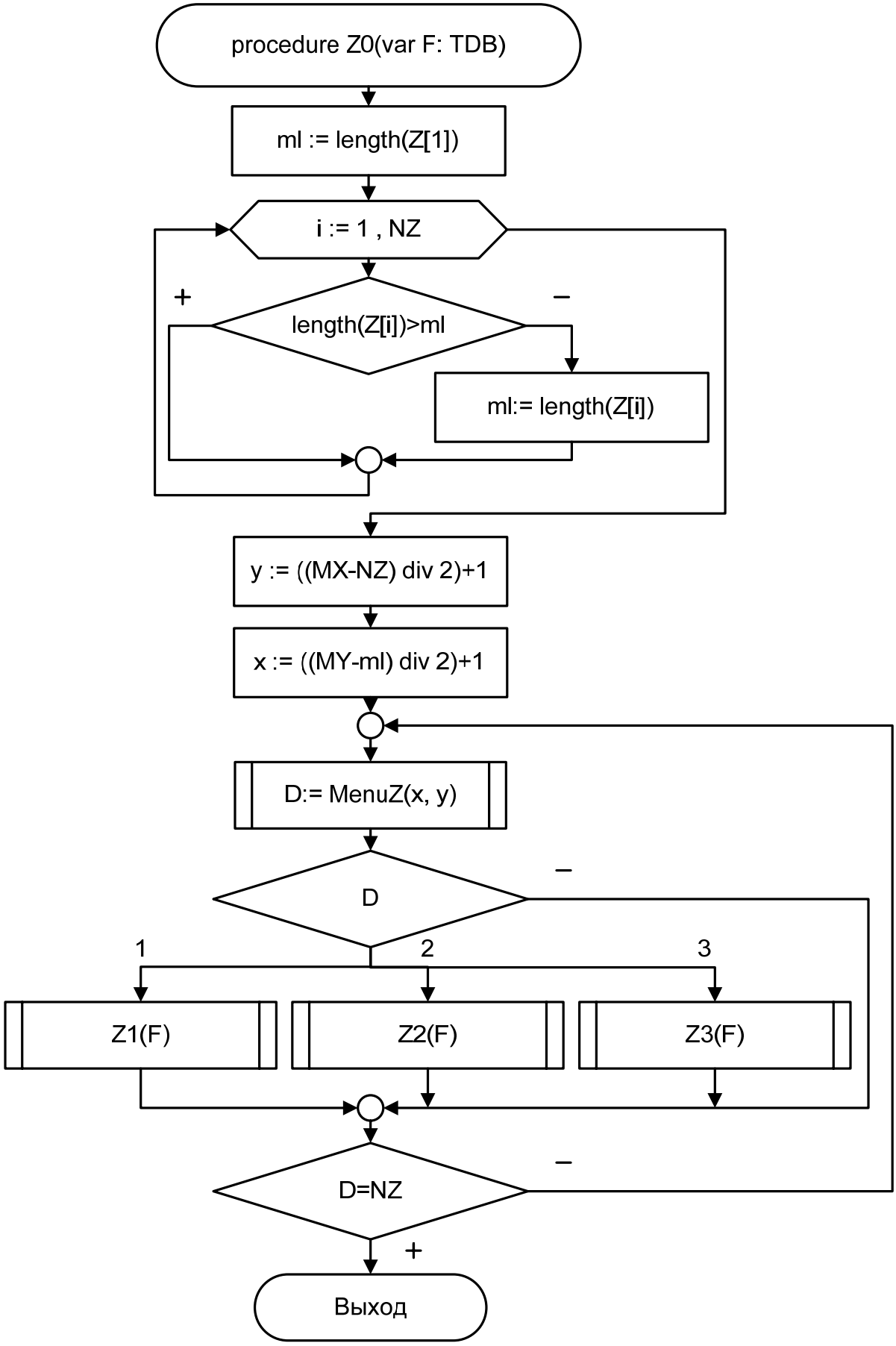


Рисунок 9.15 Меню запроса

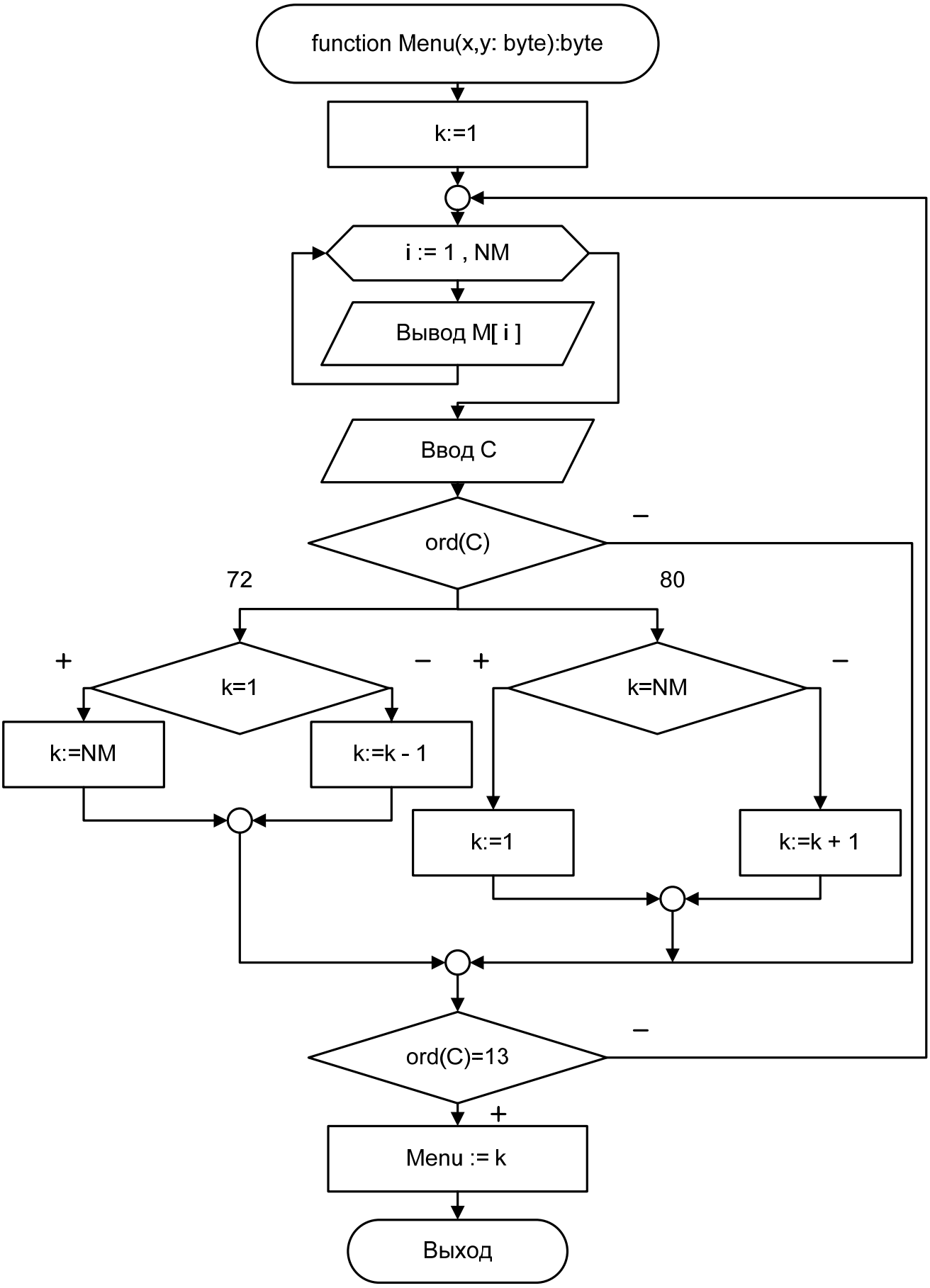


Рисунок 9.16 Основное меню

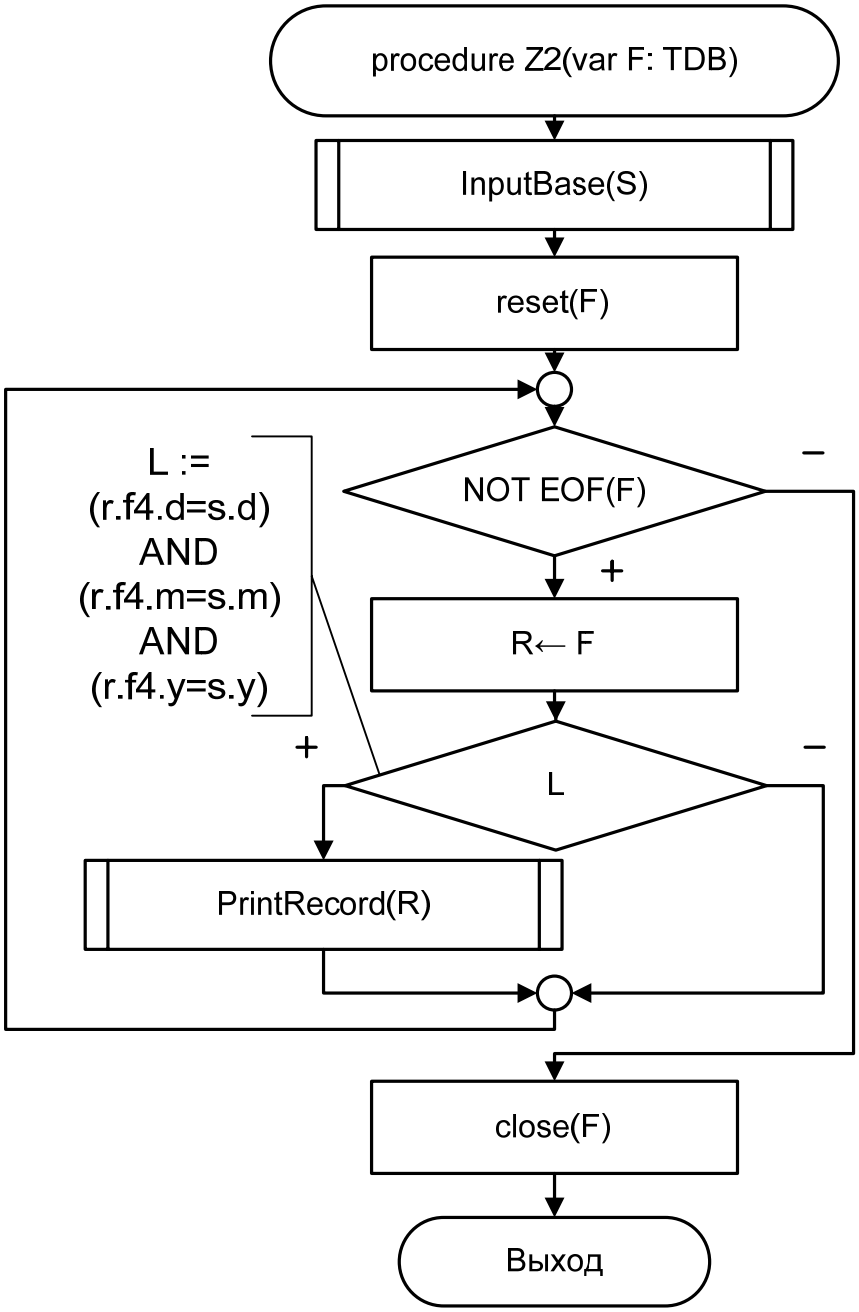


Рисунок 9.17 Запрос по дате

```
program SemesterWork;
uses CRT;
const NZ=4;
      NM=7;
      NF=7;
      COLOR1=blue;
      COLOR2=yellow;
      Group='АУ-120';
      NAME='ИВАНОВ ИВАН';
      MX=25;
```

```
      MY=80;
      X=27;
      Y=10;
      OUTCOD = '*';
type
  TDate = Record
    d,m:byte;
    y:word;
    end;
  Trec = Record
    f1:string[30];
    f2:TDate;
    f3:string[30];
    f4:TDate;
    f5:word;
    f6:string[70];
    f7:string[30];
    end;
  TDB = file of Trec;
  TZA = array[1..NZ]of string;
  TMA = array[1..NM]of string;
  TNF = array[1..NF]of string;
const
  Z:TZA = ('Список новорожденных, отсортированный
              по датам рождения',
           'Список новорожденных, зарегистрированных в
              определенный день',
           'Данные о ребенке по данным о родителях',
           'Выход');
  M:TMA = ('Создание БД',
           'Вывод БД',
           'Добавление записей в БД',
           'Удаление записей из БД',
           'Корректировка сведений в БД',
           'Запрос',
           'Выход');
  FLD:TNF = ('ФИО',
             'Дата рождения',
             'Место рождения',
             'Дата регистрации',
             'Номер записи в книге регистрации',
             'Сведения о родителях',
             'Свидетельство о рождении');
var
  F:TDB;
  R:Trec;
  C:Char;
  D,i:Byte;
```

```
{процедура вывода титульного листа}
procedure TitulnyList;
begin
  clrScr;
  textbackground(COLOR1);
  textcolor(COLOR2);
  clrScr;
  GotoXY(14,2);
  write('ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ
         ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ');
  GotoXY(24,4);
  write('КАФЕДРА "ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА"');
  GotoXY(32,8);
  write('СЕМЕСТРОВАЯ РАБОТА');
  GotoXY(45,16);
  write('ВЫПОЛНИЛ СТУДЕНТ ГРУППЫ: ',group);
  GotoXY(45,17);
  write(NAME);
  GotoXY(45,19);
  write('ПРОВЕРИЛ:');
  GotoXY(45,20);
  write('ПЕТРОВ А.Р.');
  GotoXY(33,24);
  write('ВОЛГОГРАД, 2007');
  readKey;
end;

{процедура вывода заголовка при вводе записи}
procedure IFF(var R:Trec);
var k:integer;
    c:char;
begin
  clrScr;
  write(FLD[1],' (',OUTCOD,' - конец ввода): ');
  readln(r.f1);
end;

{процедура ввода даты}
procedure InputDate(var D:TDate);
begin
  writeLn;
  write('День: ');
  readln(D.d);
  write('Месяц: ');
  readln(D.m);
  write('Год: ');
  readln(D.y);
end;
```

```
{процедура ввода записи}
procedure InputRecord(var R:Trec);
begin
  write(FLD[2],': ');
  InputDate(R.f2);
  write(FLD[3],': ');
  readln(R.f3);
  write(FLD[4],': ');
  InputDate(R.f4);
  write(FLD[5],': ');
  readln(R.f5);
  write(FLD[6],': ');
  readln(R.f6);
  write(FLD[7],': ');
  readln(R.f7);
end;

{процедура, которая вызывает ввод записи и проверяет
 признак конца ввода}
procedure CF(var F:TDB);
begin
  IFF(R);
  while R.f1<>OUTCOD do
    begin
      InputRecord(R);
      write(F,R);
      IFF(R);
    end;
end;

{процедура, которая создает файл и производит вызов
 процедуры его заполнения}
procedure VV(var F:TDB);
begin
  clrScr;
  rewrite(F);
  CF(F);
  close(F);
end;

{процедура, вывода экземпляра записи}
procedure PrintRecord(R:Trec);
begin
  clrScr;
  writeLn(FLD[1],': ',r.f1);
  writeLn(FLD[2],': ',r.f2.d,'.',r.f2.m,'.',r.f2.y);
  writeLn(FLD[3],': ',r.f3);
  writeLn(FLD[4],': ',r.f4.d,'.',r.f4.m,'.',r.f4.y);
  writeLn(FLD[5],': ',r.f5);
```

```
  writeLn(FLD[6],': ',r.f6);
  writeLn(FLD[7],': ',r.f7);
end;

{процедура печати базы}
procedure PrintBase(var F:TDB);
begin
  reset(F);
  while not eof(F) do
    begin
      read(F,R);
      PrintRecord(R);
      readKey;
    end;
  close(F);
end;

{процедура добавления записи}
procedure AddRecords(var F:TDB);
begin
  clrScr;
  reset(F);
  seek(F,Filesize(F));
  CF(F);
  close(F);
end;

{процедура удаления записи}
procedure DeleteRecords(var F:TDB);
var
  I:Longint;
begin
  reset(F);
  while not eof(F) do
    begin
      read(F,R);
      PrintRecord(R);
      writeLn('Удалить ? (1-Да)');
      C:=readKey;
      if C='1' then
        begin
          I:=filepos(F)-1;
          While not eof(f) do
            begin
              read(F,R);
              seek(F,filepos(F)-2);
              write(F,R);
              seek(F,filepos(F)+1);
            end;
```

```
          seek(F,filesize(F)-1);
          truncate(F);
          seek(F,I);
        end;
    end;
  close(F);
end;

{процедура корректировки записи}
procedure CorrectRecords(var F:TDB);
begin
  reset(F);
  while not eof(F) do
    begin
      read(F,R);
      PrintRecord(R);
      writeLn('Корректировать ? (1-Да)');
      C:=readKey;
      If C='1' then
        begin
          IFF(R);
          InputRecord(R);
          seek(F,Filepos(F)-1);
          write(F,R);
        end;
    end;
  close(F);
end;

{процедура выделения цветом пункта меню}
procedure SetLineColor(NP,NS: Byte);
begin
  if NP=NS then
    begin
      textbackground(COLOR2);
      textcolor(COLOR1);
    end
  else
    begin
      textbackground(COLOR1);
      textcolor(COLOR2);
    end
end;
```

```
{функция меню запроса}
function MenuZ(X,Y:Byte):byte;
var k:Byte;
begin
  k:=1;
  repeat
    textbackground(COLOR1);
    textcolor(COLOR2);
    clrScr;
    for i:=1 to NZ do
      begin
        GotoXY(X,Y+i-1);
        SetLineColor(k,i);
        write(Z[i]);
      end;
    C:=readKey;
    case Ord(C) of
      72:begin
           if k=1 then k:=NZ
                  else k:=k-1
         end;
      80:begin
           if k=NZ then k:=1
                   else k:=k+1
         end;
      end;
  until Ord(C)=13;
  MenuZ:=k;
end;

{логическая функция сравнения полей записи}
function Logic(r,r1:TRec):boolean;
begin
  Logic:=(r.f2.y>r1.f2.y) OR
  (
    (r.f2.y=r1.f2.y) AND (r.f2.m>r1.f2.m)
   ) OR
  (
   (r.f2.y=r1.f2.y) AND
   (r.f2.m=r1.f2.m) AND(r.f2.d>r1.f2.d)
  );
end;
```

```
{процедура сортировки файла с записями}
procedure SortFile(var f:TDB);
var r1:Trec;
    l:boolean;
begin
 repeat
   reset(f);
   L:=true;
   read(f,r);
   while NOT EOF(f) do
    begin
      read(f,r1);
      if Logic(r,r1) then
         begin
           seek(f,filepos(f)-2);
           write(f,r1);
           write(f,r);
           l:=false;
         end
      else
         r:=r1;
    end;
 until L;
end;

{процедура формирования нового файла}
procedure Z1(var F:TDB);
var g:tdb;
begin
  clrScr;
  reset(f);
  assign(g,'g.db');
  rewrite(g);
  while not EOF(f) do
    begin
      read(f,r);
      write(g,r);
    end;
  SortFile(g);
  PrintBase(g);
end;
```

```
{процедура запроса по родителям}
procedure Z3(var F:TDB);
var s:string[70];
begin
  clrScr;
  write('Введите данные о родителях: ');
  readln(s);
  writeLn;
  reset(f);
  While not eof(f) do
    begin
      read(f,r);
      if r.f6=s then
        begin
         PrintRecord(R);
         readKey;
        end;
    end;
  close(F)
end;

{процедура запроса по дате регистрации}
procedure Z2(var F:TDB);
var s:TDate;
begin
  clrScr;
  write('Введите дату регистрации: ');
  InputDate(s);
  writeLn;
  reset(f);
  while not eof(f) do
    begin
      read(f,r);
      if (r.f4.d=s.d) and
         (r.f4.m=s.m) and
         (r.f4.y=s.y) then
          begin
            PrintRecord(R);
            readKey;
         end;
    end;
  close(F);
end;
```

```
{процедура для работы с вложенным меню}
procedure Z0(var F:TDB);
var x,y,ml:integer;
begin
  ml:=length(z[1]);
  for i:=1 to NZ do
    if length(z[i])>ml then
      ml:=length(z[i]);
  y:=((MX-NZ) div 2) + 1;
  x:=((MY-ml) div 2) + 1;
  repeat
    D:=MenuZ(x,y);
    case D of
      1:z1(F);
      2:z2(F);
      3:z3(f);
    end;
    until D=NZ;
end;

{функция вывода главного меню}
function Menu(X,Y:Byte):byte;
var k:Byte;
begin
  k:=1;
  repeat
    textbackground(COLOR1);
    textcolor(COLOR2);
    clrScr;
    for I:=1 to NM do
      begin
        GotoXY(X,Y+I-1);
        SetLineColor(k,I);
        write(M[I]);
      end;
    C:=readKey;
    case Ord(C) of
      72:begin
         if k=1 then k:=NM
                else k:=k-1;
         end;
      80:begin
         if k=NM then k:=1
                else k:=k+1;
         end
      end;
  until Ord(C)=13;
  Menu:=k;
end;
```

```
{основная программа}
begin
  assign(F,'Data.db');
  TitulnyList;
    repeat
      D:=Menu(x,y);
      case D of
        1:VV(F);
        2:PrintBase(F);
        3:AddRecords(F);
        4:DeleteRecords(F);
        5:CorrectRecords(F);
        6:Z0(F);
      end;
    until D=NM;
end.
```