Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

В. Н. ЛОБКО

МАТЕМАТИЧЕСКИЕ МЕТОДЫ В ХИМИИ И ХИМИЧЕСКОЙ ТЕХНОЛОГИИ

Основы программирования вычислительных задач

Учебное пособие

Владимир 2018

УДК 004.432.2
ББК 32.973
Л68

Рецензенты:
Кандидат физико-математических наук, доцент
доцент кафедры физики и прикладной математики
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
А. Ю. Лексин

Кандидат химических наук
старший научный сотрудник Федерального государственного
бюджетного учреждения Всероссийского научно-исследовательского
института защиты животных
(Федерального центра охраны здоровья животных)
Д. С. Большаков

Издаётся по решению редакционно-издательского совета ВлГУ

Лобко, В. Н.

Л68 Математические методы в химии и химической технологии. Основы программирования вычислительных задач : учеб. пособие / В. Н. Лобко ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2018. – 107 с.

ISBN 978-5-9984-0863-2

Изложены основы теории программирования в приложении к решению вычислительных задач в химии и химической технологии. Рассмотрены основы программирования на языках высокого уровня Delphi, Lasarus (базовый язык – Pascal) в рамках их использования для научных и технологических расчётов. Курс является основой для изучения математических методов в химии.

Предназначено для студентов направления подготовки 04.03.01 – Химия, 18.03.01 – Химическая технология.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Табл. 2. Ил. 53. Библиогр.: 10 назв.

УДК 004.432.2
ББК 32.973

ISBN 978-5-9984-0863-2

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Химия, физическая химия и химическая технология относятся к дисциплинам, широко пользующимся математикой для проведения теоретических и технологических расчётов. В этом они приближаются к физическим наукам, которые немыслимы без математики. За последние годы ряд Нобелевских премий по химии присуждались за проведённые расчёты, причём иногда речь шла даже о составлении целых комплексов компьютерных программ.

Так как средства высшей математики весьма ограничены (например, взятие интеграла, дифференциальные уравнения), современное развитие вычислительных методик предполагает использование прикладной математики с её широким спектром специфических численных методов. Подавляющее число последних являются приближёнными и могут быть реализованы только на компьютерах. Это, в свою очередь, требует применения средств программирования вычислительных задач, т. е. языков программирования высокого уровня.

Именно этим вопросам использования средств программирования и численных методов для решения вычислительных задач в химии и химических науках посвящено настоящее учебное пособие, основанное на университетской программе. В нём рассмотрены вопросы теории программирования и изложены основы языка программирования высокого уровня – Pascal – базового для таких современных языков, как Delphi, Lasarus и др. Язык изложен лишь в той его части, которая совершенно необходима для программирования вычислительных задач.

Решение математических задач в химической области не может быть полностью передано профессиональным программистам, так как они всегда значительно хуже химиков понимают ту или иную конкретную задачу, и это может привести к ошибкам. В то же время высокопрофессиональный химик в рамках университетского курса вполне может освоить программирование в нужной ему части. Это позволит ему самостоятельно решать подавляющее количество спе-

цифических задач химии или в особо трудных для программирования случаях – проконтролировать решения профессиональных программистов.

В настоящее время в области вычислений чаще всего используются такие профессиональные языки высокого уровня, как C++ и Pascal. Выбор последнего для данного курса обусловлен, с одной стороны, его достаточной простотой для студентов-химиков по сравнению с C++, а с другой стороны – достаточно высоким профессиональным уровнем. В последнее время для математических расчётов стали широко применяться специализированные математические пакеты, такие как MATLAB, Mathcad, Maple, Mathematica и др. Во многих случаях эти средства содержат уже готовые программы и реализации численных методов, и их использование не требует высокой квалификации. Однако, как и все сложные программы эти пакеты содержат множество скрытых ошибок, проявляющихся лишь в некоторых случаях. Реализованные в них численные методы тестировались лишь на ограниченном количестве классов задач, и использование их для решения специфических задач химии может привести к ошибкам.

В практике решения вычислительных задач, возникающих в химической области, может быть несколько схем, подразумевающих профессиональный уровень. Во-первых, можно пользоваться только одним из языков высокого уровня. Во-вторых, за основу можно взять один или несколько математических пакетов, а для решения некоторых задач использовать языки программирования. В этом случае знание специалистом-химиком теоретических основ программирования и одного из профессиональных языков высокого уровня поможет легко самостоятельно освоить нужные математические пакеты.

Глава 1. АЛГОРИТМЫ ВЫЧИСЛИТЕЛЬНЫХ ЗАДАЧ. ЯЗЫК БЛОК-СХЕМ

§ 1. Понятие алгоритма. Основные блоки

Алгоритм – определённая последовательность действий для решения целого класса однотипных задач.

При решении математических и физических задач мы, как правило, не задумываемся, в какой последовательности осуществляется это решение. Однако, если пристальнее на это взглянуть, окажется, что мы всегда следуем какому-то определённому алгоритму, пусть и подсознательно. Если последовательность действий при этом вполне очевидна, нет необходимости выявлять этот алгоритм. При решении же более сложных задач выявление алгоритма с целью его обработки и дальнейшего использования помогает существенно упростить решение задачи и проведение расчётов по ней, так как при этом появляется возможность автоматизации с использованием компьютеров. Кроме того, разработанный компьютерный алгоритм позволит решить целую серию однотипных задач с другими исходными данными. Многие задачи могут быть решены только при предварительном составлении алгоритма.

Простейшие алгоритмы могут быть изображены графически в виде так называемых **блок-схем**. Блок-схемы представляют собой блоки нескольких определённых типов, соединённых между собой линиями со стрелками, показывающими путь алгоритма. Язык блок-схем полностью соответствует одному из первых языков программирования высокого уровня – Фортрану. Программа могла быть составлена непосредственно по блок-схеме и, наоборот, по программе можно было составить соответствующую блок-схему. В настоящее время в профессиональном программировании блок-схемы давно не используются, так как сложность программ неизмеримо возросла и современную программу невозможно изобразить блок-схемами. Однако они весьма полезны при введении в курс программирования.

Рассмотрим основные блоки. (Здесь используется упрощённая система обозначения блоков и схемы в общем, не полностью совпадающие с существующими правилами и стандартами).

1.1. Любая блок-схема начинается с блока «**Начало**» (рис. 1) и заканчивается блоком «**Конец**» (рис. 2).

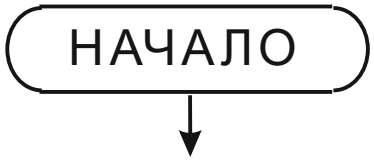

Рис. 1. Блок «Начало». Общее обозначение и соединительная линия

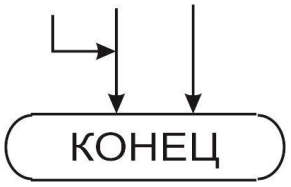

Рис. 2. Блок «Конец». Общее обозначение и соединительные линии

Необходимо всегда понимать, сколько может быть входов в тот или иной блок и сколько выходов. В блок «Начало» входов нет, а выход может быть только один. В этом и в других случаях возможность нескольких выходов означала бы неопределённость пути исполнения алгоритма, что недопустимо. В блоке «Конец» выходов нет, а входов может быть несколько, так как алгоритмы могут быть разветвлёнными, и все пути должны сходиться в одном конечном блоке.

1.2. Блок «**Ввода информации**». Изображается в виде параллелограмма (рис. 3):

Рис. 3. Блок «Ввода информации». Общее обозначение

Любой алгоритм обрабатывает какую-то исходную информацию. Поэтому практически всегда перед выполнением программы требуется ввод исходных данных. Данные хранятся в компьютере в «дисковой» памяти (жёсткий диск, оптический диск, флэш-память и т. д. – общим является то, что после выключения компьютера информация сохраняется) как правило в виде файлов и в оперативной памяти (после выключения компьютера информация не сохраняется). При выполнении программы данные и код самой программы находятся в оперативной памяти. Данные, т. е. такие объекты программы, как, например, переменные и постоянные, находятся в так называемых **ячейках памяти**, которые в начале исполнения алгоритма резервируются под каждый объект программы в оперативной памяти. Оперативная память разбита на **элементарные ячейки** памяти, которые расположены последовательно одна за другой и имеют

свой уникальный **адрес**. По адресу можно всегда безошибочно обратиться к той или иной элементарной ячейке. Объём этих элементарных ячеек зависит от разрядности системы (32-битные, 64-битные). С другой стороны, **ячейки памяти** тех или иных данных могут состоять из одной или нескольких элементарных. Например, ячейка памяти для хранения целого числа требует меньшего объёма по сравнению с действительным числом и содержит меньшее количество элементарных ячеек. Переменные, постоянные и другие объекты имеют имена; такие же имена имеют соответствующие ячейки памяти. К переменной (или другим объектам) можно обратиться по имени ячейки или по адресу первой элементарной ячейки. В самих ячейках памяти содержатся **значения** того или иного объекта. В случае переменных эти значения могут меняться в ходе исполнения программы, константы же не меняют раз полученное значение. Условимся изображать ячейки памяти прямоугольниками, например, для ячейки памяти, **зарезервированной** под переменную x (рис. 4):

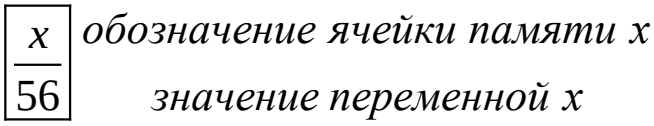

Рис. 4. Графическое обозначение ячейки памяти

Операция ввода информации состоит в записи соответствующего значения в ячейку памяти объекта. В блоке ввода указываются имена ячеек памяти – т. е. имена переменных. При вводе нескольких переменных в одном блоке они перечисляются через запятую (разделитель списка) и образуют **список ввода**, который исполняется последовательно. (При реализации ввода на компьютере на экране монитора (если ввод осуществляется с клавиатуры) происходит запрос значения в виде мигающего курсора. Значение переменной набирается, и нажимается клавиша «ввод». Неудобство здесь состоит в том, что запрос производится вслепую и нужно помнить, какую именно переменную нужно вводить). Пример (рис. 5). В этом блоке будут последовательно вводиться переменные $x1$, a и $z33$.

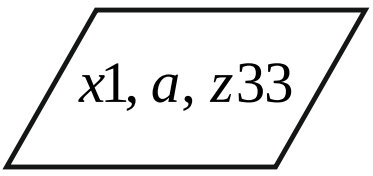

Рис. 5. Блок ввода. Пример использования

1.3. Блок «**Вывода информации**». Изображается в виде сложной фигуры (рис. 6):

Рис. 6. Блок «Вывода информации». Общее обозначение

Служит он для вывода рассчитанных значений, дополнительной и вспомогательной информации, чаще всего – на экран монитора или в файл. Поскольку данный курс ориентирован на научные и технологические расчёты, будем пользоваться несколько устаревшими терминами **арифметическое выражение** и **текстовая информация**; именно они и могут фигурировать в этом блоке. Под **арифметическим выражением** (будем обозначать «**а. в.**») понимается: а) переменная или постоянная, значением которой является число (целое или действительное), б) сами эти числа и в) сколь угодно сложное выражение (формула). Во всех трёх случаях значением а. в. служит число; оно и будет выводиться. **Текстовая информация** заключается в кавычки и состоит из одного символа или любой их последовательности (все 256 символов ПК, включая кириллицу). Текстовая информация не анализируется компьютером и выводится буквально. Чаще всего этот текст содержит справочную информацию о выводимых значениях переменных или выражений. В с очетании с блоком «ввода» текстовая информация блока «вывода» может содержать подсказку о вводимом значении. (В профессиональном программировании эти два термина не используются, там есть общий термин **выражение** такого-то **типа**, в частности: «выражение числового типа» (целого или действительного), «выражение символьного типа», «выражение строкового типа» (строка – последовательность символов) и т. д.). Несколько выводимых в одном блоке объектов, перечисляемых через запятую (**разделитель списка вывода**), образуют **список вывода**, который исполняется последовательно. Примеры (рис. 7, 8).

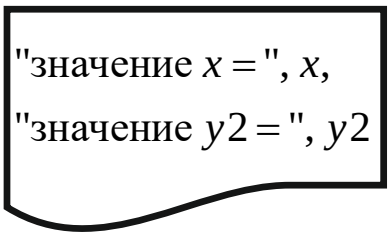

Рис. 7. Блок вывода. Пример использования 1

В этом блоке (рис. 7), состоящем из 4 элементов списка, будут последовательно, в сопровождении текстовой информации, выводиться значения двух объектов (например, переменных или постоянных), x (значение, например, 67.2) и $y2$ (значение 0.44); результат вывода будет такой:

значение $x = 67.2$

значение $y2 = 0.44$.

Пример рационально оформленного ввода. Сначала в блоке вывода выводится текстовая подсказка, а затем в блоке ввода вводится значение переменной $s54$. При вводе с клавиатуры на мониторе высветится подсказка «ввести переменную $s54 =$» с мигающим курсором в конце; необходимо набрать конкретное значение переменной $s54$ и нажать клавишу ввода.

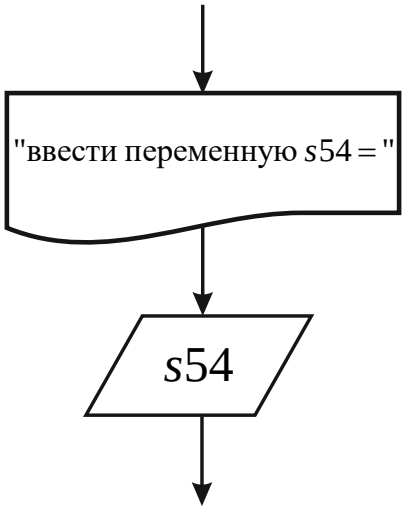

Рис. 8. Блок вывода. Пример использования 2

1.4. «**Операционный блок**». Изображается в виде прямоугольника (рис. 9).

Рис. 9. Операционный блок. Общее обозначение

В нём помещается **операция присваивания**. Присваивание заключается в записи в соответствующую ячейку памяти нового значения переменной. При этом старое значение бесследно пропадает. **Формат** операции (общая условная запись; все форматы далее будут обведены по контуру):

<обозначение ячейки памяти> **:=** <а. в.> ,

здесь обозначение ячейки памяти – имя переменной, **:=** – знак операции присваивания, а. в. – арифметическое выражение (в более общем случае – выражение того же типа, что и переменная). При непосредственном использовании формата содержимое треугольных скобок (< и >) заменяется конкретной информацией, а сами скобки опускаются. (В дальнейшем при использовании форматов нам встретятся также фигурные скобки { и }, в них помещаются необязательные элементы, т. е. те элементы, которые в одних случаях могут присутствовать, а в других – отсутствовать). Разумеется, что обводной контур также опускается. Например, при записи в ячейку памяти переменной xyz значения 77.8 будем иметь

$xyz := 77.8$

Работа операции присваивания: сначала вычисляется выражение, стоящее справа, а затем вычисленное значение помещается в ячейку памяти, стоящую слева (старое значение пропадает).

Операция присваивания кардинально отличается от равенства.

Правильными ли будут выражения:

$x = 2$ $x := 2$?

Да, всё правильно. Можно ли записать:

$x = x+1$?

Нет, ошибка.

$x := x+1$?

Всё правильно, сначала вычисляется выражение справа, т. е. из ячейки памяти x в оперативную память копируется его значение (2), складывается с единицей и результат (3) записывается в ячейку памяти, обозначенную слева (x), при этом значение 2 бесследно пропадает.

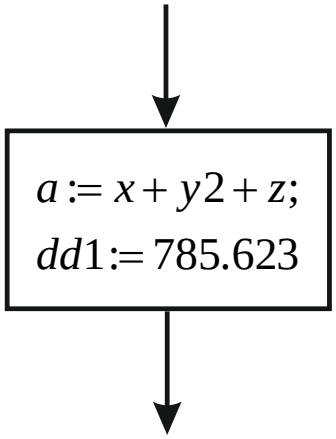

Рис. 10. Операционный блок. Пример использования

Можно ли записать:

$x+1 = x+1$?

Да, всё правильно.

$x+1 := x+1$?

Нет, ошибка! Результат, полученный справа (4), должен быть записан в ячейку памяти, но слева стоит а. в., а не имя ячейки памяти.

В блок-схемах операция присваивания просто помещается в операционный блок. Операции присваивания, следующие последовательно одна за другой, могут помещаться в одном блоке (рис. 10).

Общее замечание. В блоках 1.2; 1.3; 1.4 может быть несколько входов (так как алгоритмы могут быть разветвлёнными), но только один выход.

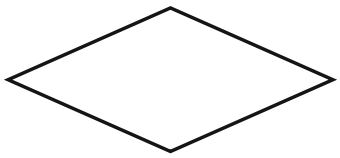

Рис. 11. Логический блок. Общее обозначение

1.5. «Логический блок», или **«Блок сравнения»**. Изображается в виде ромба, вытянутого по горизонтали (рис. 11):

Служит для организации сложных разветвлений алгоритма. Может иметь несколько входов, но всегда **два** выхода. В него помещается **логическое выражение** (в терминологии профессионального программирования –

выражение логического типа). Будем обозначать его «**л. в**.» Л. в. имеют всего два возможных значения – «**true**» («правда», или «да») и «**false**» («ложь», или «нет»). Для сравнения отметим, что а. в. имеют бесконечное количество значений (теоретически; на практике же из-за ограничения точности представления чисел в компьютере это число конечно, хотя и очень большое). Л. в. бывают простые и сложные.

Формат простых л. в.:

<a. в.> <знак сравнения> <a. в.>,

здесь знак сравнения: 1) > больше, 2) < меньше, 3) ≥ больше или равно, 4) ≤ меньше или равно, 5) = равно, 6) ≠ не равно. Л. в. может быть подсчитано и получено одно из двух возможных значений л. в. Если – true, то происходит выход из блока сравнения по стрелке «да», при этом выполняется (т. е. верно) **прямое условие**, стоящее в блоке. В случае false автоматически выполняется **обратное условие** и выход происходит по стрелке «нет». (Если, например, $x < 2$ – прямое условие, то обратным будет $x \geq 2$; для прямого $x = 5$ обратное – $x \neq 5$). Пример – рис. 12; обозначение стрелок «да» и «нет» обязательно. Стрелки «да» и «нет» могут выходить из любых двух из четырёх вершин ромба; в остальные две (или одну) вершины может осуществляться вход.

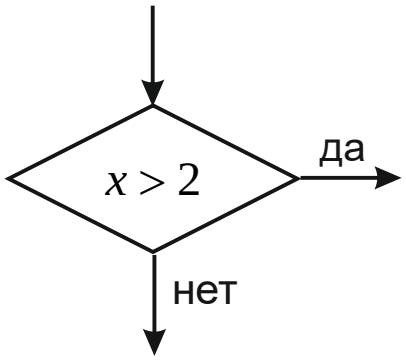

Рис. 12. Пример использования логического блока

При анализе логических выражений (особенно сложных) всегда полезно изображать числовые оси, на которых представлены соответствующие точки и дугами обозначены интервалы. Интервал может быть незамкнутым, если он уходит в бесконечность или минус бесконечность. Если точка принадлежит данному интервалу, она изображается закрашенной, если нет, то она изображается кружочком. Для рассмотренного логического блока (см. рис. 12) при выходе по стрелке «да» будем иметь ситуацию – рис. 13.

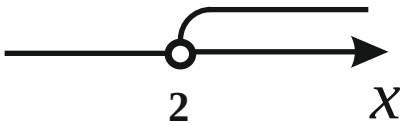

Рис. 13. Выход по стрелке «да» (см. рис. 12)

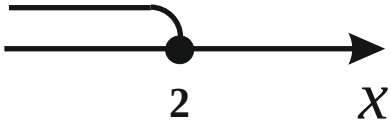

Рис. 14. Выход по стрелке «нет» (см. рис. 12)

После выхода по стрелке «да» алгоритм далее будет исполняться с учётом $x > 2$. При выходе по стрелке «нет» автоматически будет выполняться (т. е. true) обратное условие $x \leq 2$ (рис. 14).

Формат сложных л. в.:

<л. в.> <знак логической операции> <л. в.>,

здесь знак логической операции: **and** – логическое «и», **or** – логическое «или»; **and** выполняется, когда выполняются оба л. в., **or** выполняется, когда выполняется хотя бы одно из л. в. Также ещё существует логическая операция **not** – логическое «не», которая имеет специальный формат:

<знак логической операции> <л. в.>,

т. е. это унарная операция, **and** и **or** – бинарные операции, **not** меняет знак результирующего л. в. на противоположный.

В обоих случаях л. в. в этих форматах может быть как простым, так и сложным (вложение сложных л. в.).

Примеры:

$(5x < 67)$ or $(y \geq 122)$,

$(f < -88)$ and $(f > 0)$,

not $(k > 5)$,

$((r > 5x)$ and $(t \neq 4))$ or $(m < 8u - 3)$.

Именно в таком виде эти сложные выражения могут быть помещены в логический блок.

При записи сложных логических выражений нужно проверять их правильность. В одних случаях выражение должно выполняться, в других – нет (в особых случаях для некоторых алгоритмов это может быть и не так). Для анализа следует применять числовую ось. Рассмотрим ось с двумя точками -2 и 3, которые разбивают ось на три интервала.. Будем ставить различные логические условия и подсчитаем значения выражений для x из разных интервалов, например, для $x = -3$, $x = 0$ и $x = 5$.

Условие $(x > -2)$ and $(x < 3)$ – рис. 15.

Полученные полуинтервалы перекрываются и образуют замкнутый интервал.

Значения логического выражения:

при $x = -3$ false,

при $x = 0$ true,

при $x = 5$ false.

Мы видим, что в одних случаях логическое выражение выполняется, а в других – нет, что свидетельствует о его правильности.

или

Рис. 15. Числовая ось для $x > -2$ и $x < 3$.
Полученные полуинтервалы перекрываются

Условие $(x > -2)$ or $(x < 3)$ – см. рис. 15.

Значения логического выражения:
при $x = -3$ true,
при $x = 0$ true,
при $x = 5$ true.

Мы видим, что для любых x логическое выражение выполняется, значит, скорее всего, оно составлено неправильно.

Условие $(x < -2)$ and $(x > 3)$ – рис. 16.

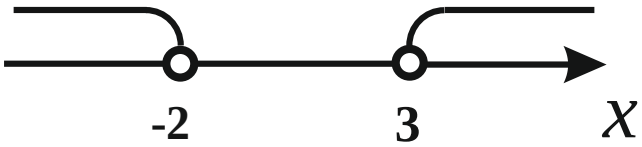

Рис. 16. Числовая ось для $x < -2$ и $x > 3$

Полученные полуинтервалы не перекрываются.

Значения логического выражения:
при $x = -3$ false,
при $x = 0$ false,
при $x = 5$ false.

Ни при каких x выражение не выполняется, значит, скорее всего, оно также составлено неправильно.

Условие $(x < -2)$ or $(x > 3)$ – см. рис. 16.

Значения логического выражения:
при $x = -3$ true,
при $x = 0$ false,
при $x = 5$ true.

В одних случаях логическое выражение выполняется, а в других – нет, значит, всё правильно.

(Следует отметить, что, конечно же, могут существовать алгоритмы, где использование описанных «неправильных» логических выражений оправдано).

Итак, сложные логические выражения могут быть помещены в логические блоки, которые будут работать точно так же. Однако далее, в учебных целях, мы не будем использовать эту возможность, а будем применять только простые логические выражения.

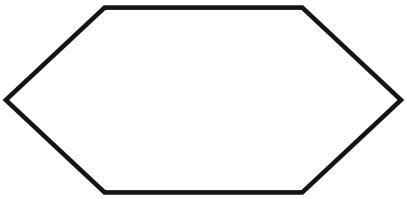

Рис. 17. Блок цикла. Общее обозначение

1.6. «Блок цикла». Изображается в виде вытянутого шестиугольника (рис. 17):

Этот блок будет рассмотрен ниже.

Рис. 18. Блок переноса. Слева – вход, справа – выход

1.7. «Блок переноса». Используется в сложных схемах для связи между блоками, когда непосредственная связь затруднена. Состоит из двух кружочков, обозначенных буквой или цифрой, располагающихся в разных частях схемы, которые нужно, но затруднительно соединить (рис. 18). К входному кружочку подводится стрелка от какого-то блока, а от выходного кружочка стрелка отходит в направлении дальнейшего хода алгоритма.

§ 2. Разветвляющиеся алгоритмы

Линейные алгоритмы (простейшие) – все действия (и соответствующие блоки) идут последовательно одно за другим. Всё очевидно, и нет смысла их рассматривать.

Разветвляющиеся алгоритмы предполагают несколько возможных путей развития алгоритма в зависимости от конкретных значений тех или иных переменных, постоянных и других объектов. Разветвления образуются чаще всего при использовании логических блоков. И хотя такой алгоритм предусматривает несколько возможных путей, определённые исходные данные ведут только к одному конкретному пути развития алгоритма.

Пример 1

Составить блок-схему для вычисления значения кусочной функции:

$$y = \begin{cases} \sin 2x, \text{ если } x \le 5, \\ \ln x, \text{ если } x > 5. \end{cases}$$

Значение y будет вычисляться по разным формулам в зависимости от вводимого x. При решении подобных задач (особенно для последующих, более сложных) полезно строить числовую ось с обозначением всех точек из условия задачи, без масштаба (важен лишь порядок следования). На числовой оси будем отмечать интервалами ситуации, возникающие по ходу развития алгоритма. Если поставить первое условие $x \le 5$, то числовая ось будет аналогична рис. 14 (только вместо 2 будет 5). В блок-схеме нужно ввести значение x и проанализировать его, поставив, например, первое условие. Выход по стрелке «да» означает, что функцию нужно вычислять по формуле $y = \sin 2x$; используем для этого операцию присваивания (рис. 19).

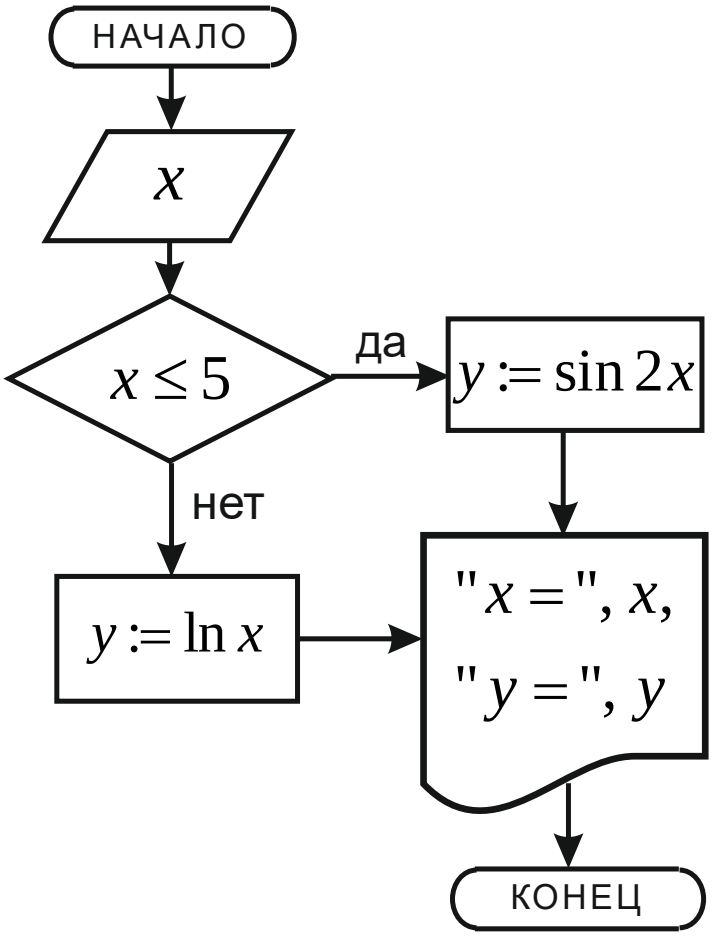

Рис. 19. Блок-схема простейшего разветвляющегося алгоритма. Пример 1

При выходе по стрелке «нет» (аналог рис. 13) **автоматически** выполняется обратное условие $x > 5$, т. е. дополнительно ставить это условие не нужно. Выход по стрелке «нет» приводит к формуле $y = \ln x$. После того как функция будет так или иначе вычислена, необходимо вывести полученное значение и заданный x (для справки) – см. рис. 19. (При составлении блок-схем стрелки ставить обязательно, чтобы не было неопределённости в направлении движения).

Пример 2

Составить блок-схему для вычисления значения кусочной функции:

$$y = \begin{cases} \sin 2x, \text{ если } x < -4, \\ 3x^2 + 6, \text{ если } -2 \le x < 0, \\ \ln x, \text{ если } x > 0. \end{cases}$$

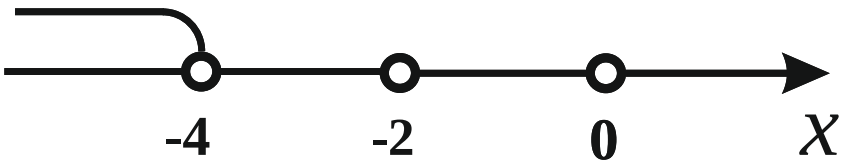

Рис. 20. Числовая ось

Эта схема будет отличаться от предыдущей только более сложным анализом значения x. В условии фигурируют три точки, которые на числовой оси дадут четыре интервала. Задачу будем решать без использования сложных логических выражений (в учебных целях). Анализ можно начинать с постановки любого условия и продолжать их в любой последовательности, но во избежание ошибок и простоты ради лучше двигаться в анализе последовательно, например, слева направо. Левая точка из условия – -4, значит, нужно получить интервал от $-\infty$ до этой точки. Для этого следует поставить условие $x < -4$, которое соответствует первой функции задачи, т. е. будет получен интервал, представленный на рис. 20.

Эта ситуация определяет вычисление функции по формуле $y = \sin 2x$ согласно условию задачи. Можно начать составление блок-схемы – ввести x, поставить условие и вычислить y (рис. 21).

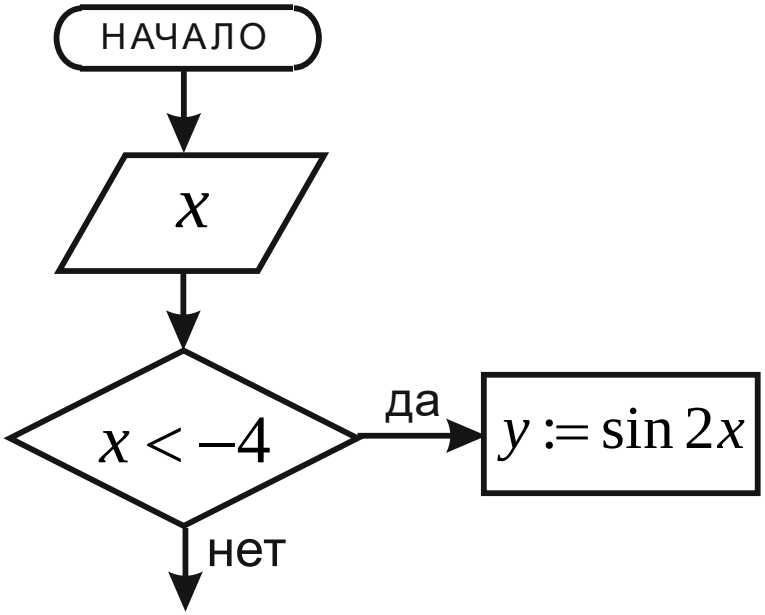

Рис. 21. Первый этап составления блок-схемы

Операционный блок пока оставляем; выход по стрелке «нет» означает автоматическое выполнение обратного условия $x \geq -4$ – рис. 22 (точка -4 входит в интервал).

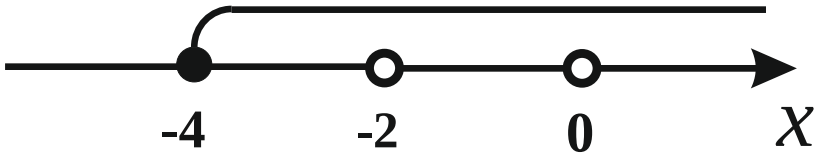

Рис. 22. Числовая ось

Имея этот интервал, мы никаких выводов и действий сделать не можем. Нужно получить следующий по порядку замкнутый интервал от -4 до -2. В условии задачи этот интервал никак не описан, т. е. x в нём не определён. Это означает, что если вводимый x будет принадлежать этому интервалу, то задача не будет иметь решения. Для получения замкнутого интервала нужно поставить условие $x < -2$ (точка -2 не должна входить в него, так как она принадлежит следу-

ющему интервалу). Два полуинтервала навстречу друг другу дадут замкнутый интервал (подобно рис. 15) – рис. 23.

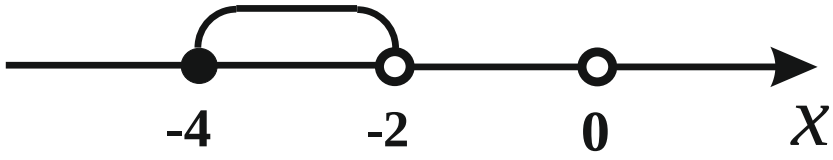

Рис. 23. Числовая ось

Выход по стрелке «да» ведёт к ситуации «нет решения», что надо отразить в блоке вывода – рис. 24.

Выход по стрелке «нет» приводит к ситуации рис. 25.

Для определённости нужно получить следующий замкнутый интервал – от -2 до 0, он будет соответствовать второму условию задачи. Для этого нужно поставить условие $x < 0$ (рис. 26), т. е. точка 0 не входит в этот интервал, а функция вычисляется по формуле $y = 3x^2 + 6$.

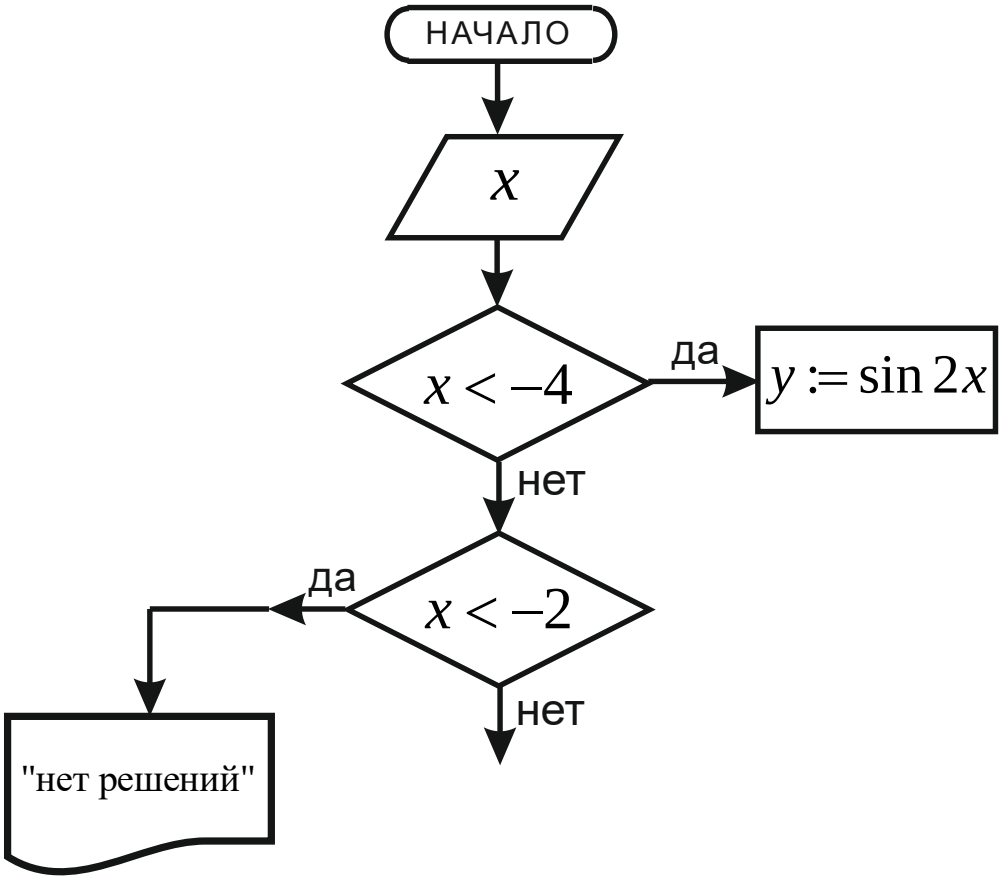

Рис. 24. Второй этап составления блок-схемы

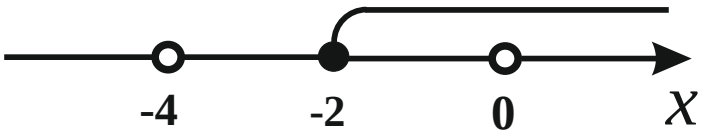

Рис. 25. Числовая ось

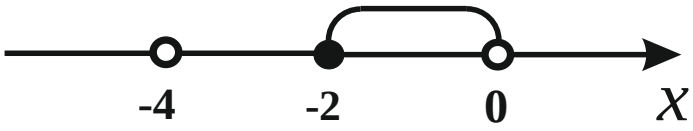

Рис. 26. Числовая ось

При выполнении этого условия (стрелка «да») будем иметь – рис. 27.

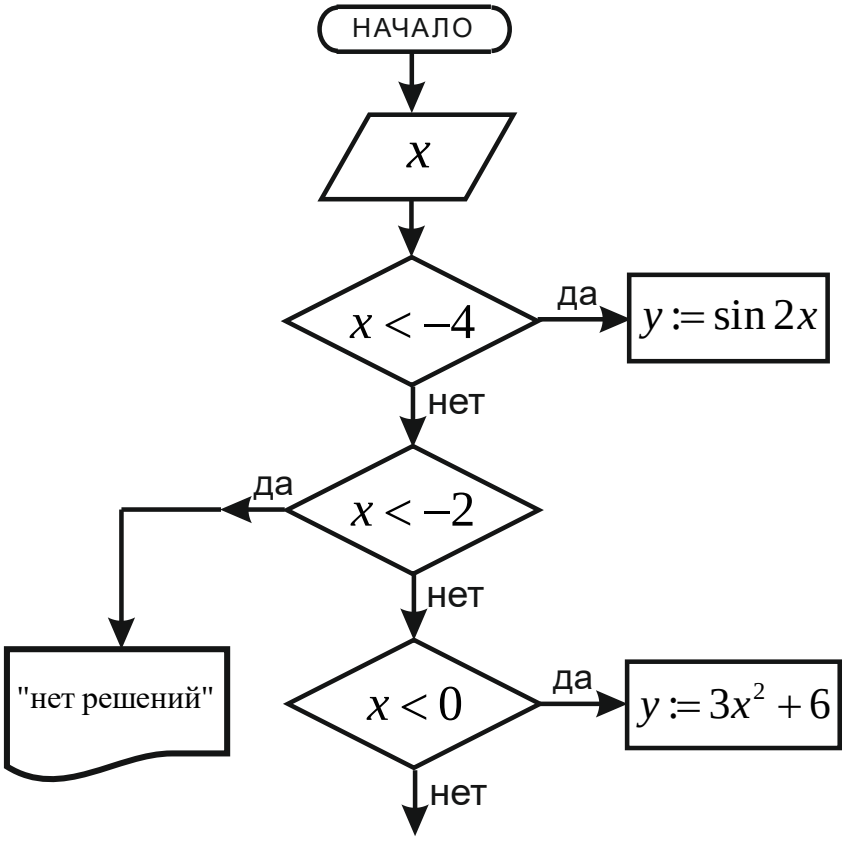

Рис. 27. Третий этап составления блок-схемы

Выход по стрелке «нет» ведёт к ситуации $x \geq -2$ (обратное условие) – рис. 28.

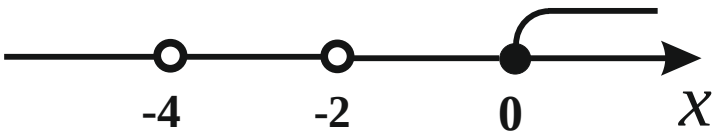

Рис. 28. Числовая ось

Это не соответствует последнему условию задачи, так как точка 0 будет входить в полученный интервал. Исключить эту точку можно, поставив условие $x = 0$. При этом решений задачи не будет, и выход по стрелке «да» приведёт к соответствующему блоку вывода (рис. 29).

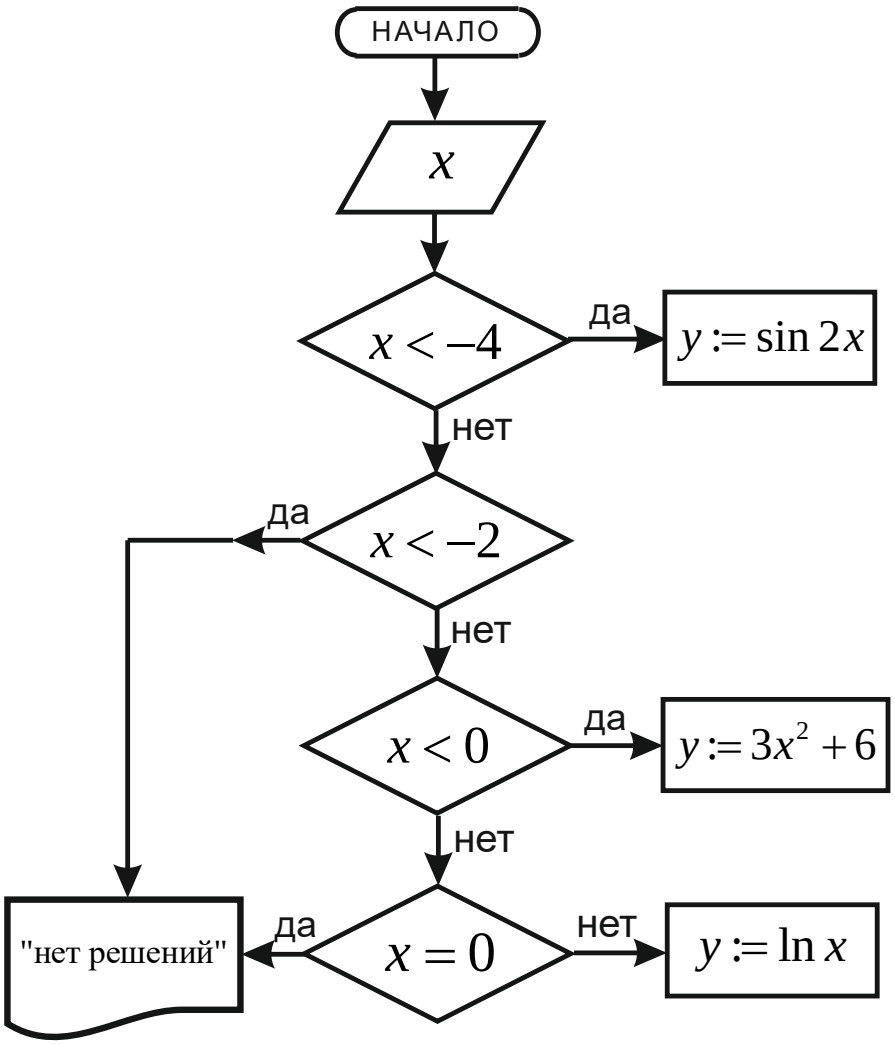

Рис. 29. Четвёртый этап составления блок-схемы

Отметим, что дублировать одинаковые блоки не надо, а нужно вести соединительные линии к имеющемуся блоку (если необходимо – использовать блок переноса).

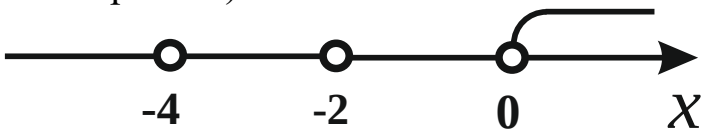

Рис. 30. Числовая ось

При выходе по стрелке «нет» получаем интервал – рис. 30.

Этот интервал соответствует последнему неравенству задачи – см. рис. 29. Анализ закончен. Мы получили четыре блока, «висящие в воздухе». Три операционных блока представляют три варианта вычисления функции согласно условию задачи. После них должен идти блок вывода полученного значения y (один). Тогда у нас останется два незавершённых блока вывода, мы их сводим в «конец». В итоге получится схема – рис. 31.

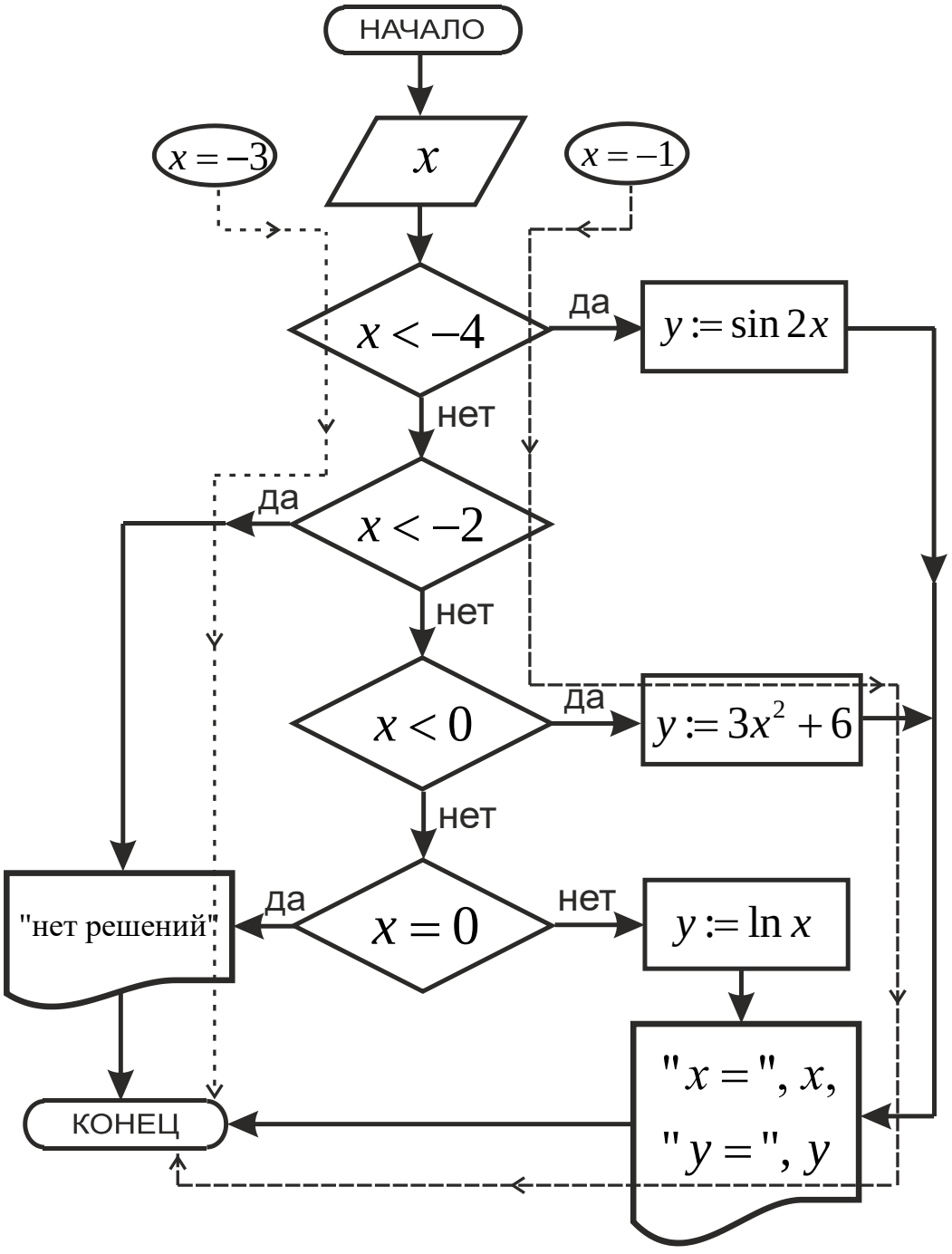

Рис. 31. Итоговая блок-схема разветвляющегося алгоритма

При составлении блок-схем трудно заранее предугадать рациональное расположение блоков. Сначала нужно составить «правильную» схему, а затем можно её сделать «красивой». При постановке логических условий можно ставить как прямые, так и обратные условия, совершенно всё равно. Рассмотрим второй логический блок схемы рис. 31. Заменим условие $x < -2$ на обратное $x \geq -2$. Вопрос: какие минимальные изменения нужно внести в блок-схему, чтобы опять всё стало правильно? Ответ: выходы по стрелкам «да» и «нет» нужно поменять местами – рис. 32.

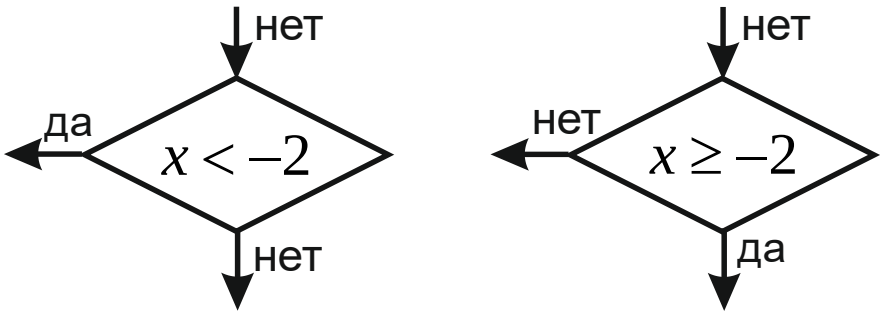

Рис. 32. Фрагмент схемы рис. 31. Изменение логического условия

Алгоритмы могут быть сильно разветвлёнными, однако путь конкретного решения при определённых данных представляет собой ломаную прямую без разветвлений, т. е. всё однозначно предопределено. На рис. 31 пунктирными линиями показаны пути алгоритма для конкретных исходных значений $x = -3$ и $x = -1$. Для определения пути после логических блоков нужно вместо x в условии подставить его значение и ответить на вопрос false или true.

Схема рис. 31 значительно упрощается, если использовать сложные логические условия. В данной задаче это одно условие – второе. Соответствующая схема представлена на рис. 33. Именно в такой манере составляются программы на конкретных языках программирования.

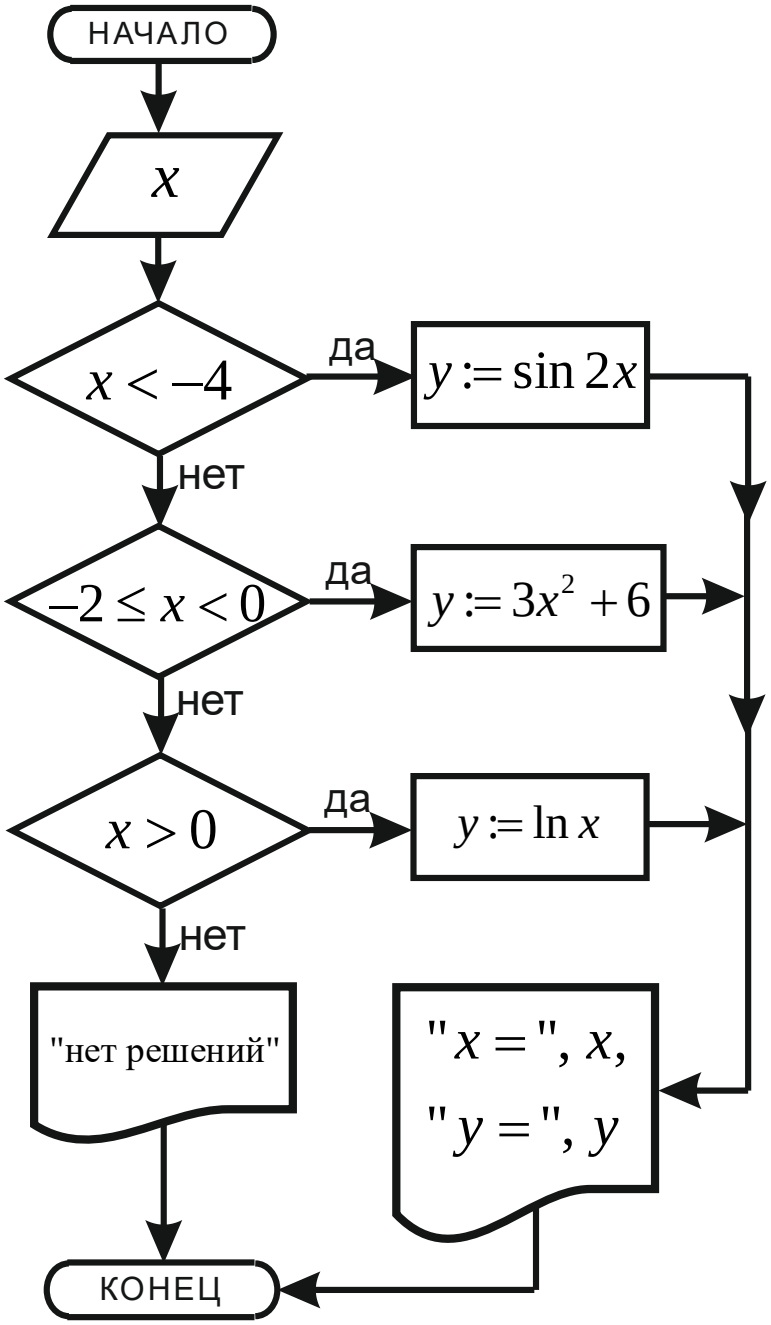

Рис. 33. Более рациональная блок-схема
разветвляющегося алгоритма

§ 3. Циклические алгоритмы

Очень часто в алгоритмах бывает необходимо провести несколько вычислений (или других действий) по одной и той же схеме с различными значениями переменных. Для организации таких **повторяющихся** вычислений служат **циклы**. Путь решения при этом остаётся линейным, но происходит как бы «кручение» на одном месте не-

сколько раз. Для того чтобы не произошло «зацикливания», должен быть предусмотрен механизм выхода из цикла (например, при выполнении какого-то условия).

Допустим, необходимо рассчитать несколько значений y при различных x:

$y = \sin^2 x^2$ для $x = 5, 7, 9, 11, 13$.

Нетрудно видеть, что алгоритм решения должен сводиться к следующему. Переменной x нужно сначала присвоить значение 5, произвести расчёт y и вывести результат, а затем взять следующее значение x (7) и повторить вычисления. Такое циклическое повторение алгоритма нужно проводить до значения $x = 13$ и затем выйти из цикла. Можно видеть, что переменная x в каждом цикле меняется на постоянный шаг 2. Выход из цикла можно организовать в блоке сравнения с условием $x \le xk$, где $xk = 13$. Таким образом, для организации цикла необходимо знать три величины: начальное значение xn, конечное значение xk и шаг h (в нашем случае $xn = 5$, $h = 2$; обозначения – любые). Именно их и нужно вводить как исходные данные. Нужно отметить, что эти величины – константы, они не меняются в ходе решения. Наоборот, x и y – переменные. При составлении алгоритмов необходимо постоянно следить, является ли данная величина постоянной и переменной. Чтобы сделать вычисления y при начальном x, нужно $x := xn$ (переменная получает начальное значение). После вычисления первого значения y нужно его вывести, так как в дальнейшем оно изменится. Далее нужно получить следующее значение x. Для этого его **текущее значение** нужно увеличить на шаг (ошибочным является действие $x := xn + h$, эта формула сработает для второго цикла, но дальше будут ошибки, так как справа стоят **постоянные** и x меняться не будет). Далее можно замкнуть цикл на операционный блок $y := \sin^2 x^2$, но тогда получится бесконечный цикл, поэтому в этом месте нужно вставить логический блок, который бы либо продолжал цикл, либо осуществлял выход из него. Можно поставить условие $x \le xk$ (или обратное). Разумеется, после выхода из цикла вывод информации не требуется, так как всё уже выведено. Описанный алгоритм можно изобразить следующей блок-схемой с использованием уже известных блоков – рис. 34.

В любых циклах всегда есть переменная, которая изменяется в цикле заранее известным образом, такая переменная называется **параметром цикла**. Можно сказать, что **цикл организуется по параметру цикла**. В нашем примере параметром является x. В большинстве случаев цикл может быть организован, если для параметра цикла известны **начальное значение**, **конечное значение** и **шаг** изменения. Шаг, как правило, постоянен и прибавляется к текущему значению параметра. Если рассмотреть другую переменную нашего примера – y, то до цикла (т. е. априорно) её значения и закон изменения неизвестны.

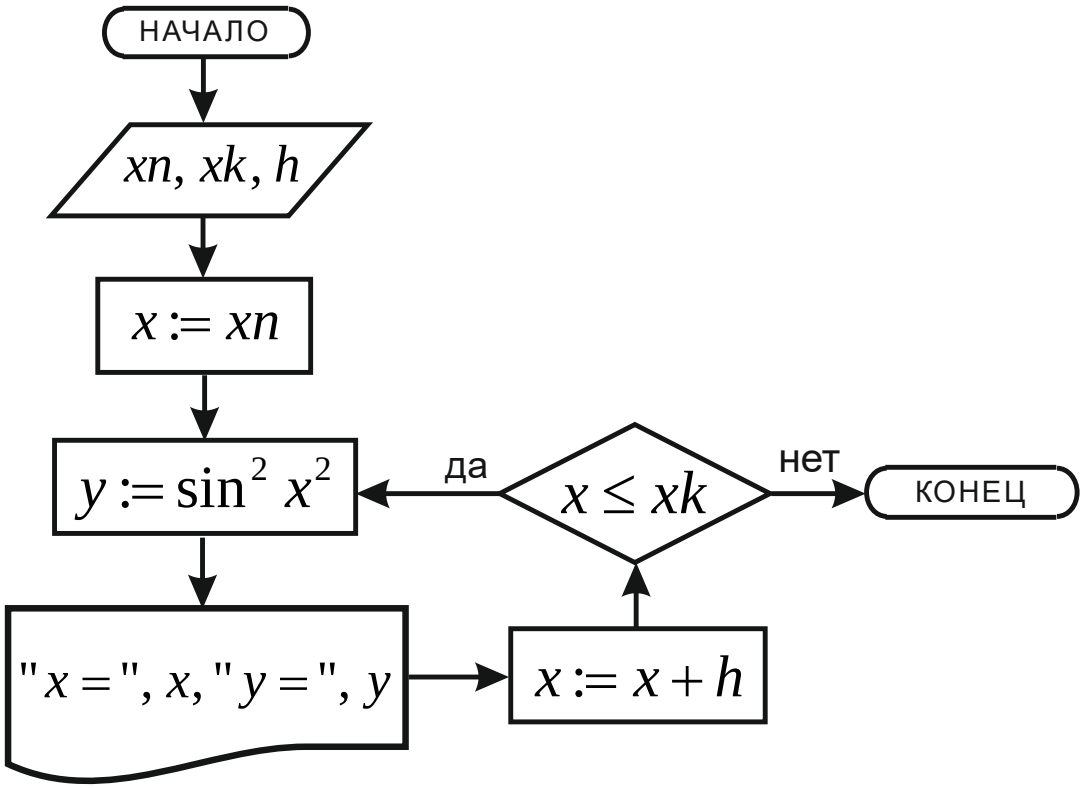

Рис. 34. Блок-схема циклического алгоритма (нерациональная)

При **организации** цикла необходимо прежде всего выбрать параметр цикла. Для этого нужно проанализировать все переменные, которые будут фигурировать в цикле, и подобрать такую, про которую известна вся необходимая информация. Если таких переменных окажется несколько, можно выбрать любую из них. Бывают случаи, когда таких переменных нет, но цикл необходим. Тогда вводят дополнительно целочисленную вспомогательную переменную и организуют цикл по ней (целый параметр цикла называется **счётчиком цикла**; как правило, он начинается с 1 и имеет шаг 1).

Иногда при организации циклов оказывается, что задано не конечное значение параметра цикла, а число циклов n. В этом случае конечное значение легко вычисляется по формуле

$$xk = xn + (n-1)h$$

(выражение $xk = xn + nh$ ошибочно, так как первый цикл выполняется с $x := xn$, а шаг прибавляется к параметру цикла $n-1$ раз; это легко проверить на простом примере).

Блоки цикла (рис. 35) служат для автоматизации циклов с известными для параметра цикла начальным значением, конечным значением и шагом. В предыдущей блок-схеме мы должны были сами продумывать, где и как будет меняться параметр цикла и где и как будет происходить выход из цикла. Блок цикла строится по определённым правилам, что минимизирует ошибки и экономит время. Это особенно актуально для больших программ.

Внутри самого блока помещается **заголовок цикла**, который строится по формату

$$\boxed{< x >:=< xn >,< xk >,< h >},$$

где x – параметр цикла, xn – начальное значение параметра цикла, xk – конечное значение параметра цикла, h – шаг (эти обозначения могут быть любыми).

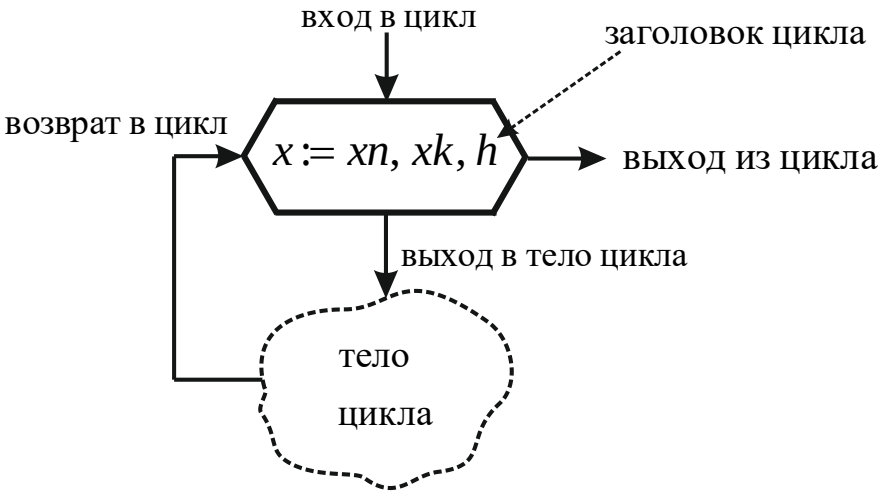

Рис. 35. Блок цикла (общее строение)

На месте xn, xk, h могут стоять а. в. В частности, если не известно xk, на его месте может стоять выражение $xn + (n-1)h$ (но не формула $xk = xn + (n-1)h$!). Шаг 1 опускается.

Вход в цикл осуществляется сверху (или снизу). Снизу (или сверху) производится **выход в тело цикла**. **Тело цикла** – это группа блоков (или – операторов в языках программирования), в которых осуществляется основной алгоритм цикла. После прохождения тела цикла происходит **возврат в цикл** – слева или справа. Справа или слева находится **выход из цикла.**

Работа блока цикла. После вхождения в цикл параметру цикла **автоматически** присваивается начальное значение параметра и происходит выход в тело цикла. После прохождения тела цикла осуществляется возврат в цикл, где к параметру цикла **автоматически** прибавляется шаг и **автоматически** происходит сравнение его с конечным значением. Если **превышает** – осуществляется выход из цикла, если **не превышает** – происходит выход в тело и цикл повторяется.

Изобразим на рис. 36 предыдущую блок-схему (см. рис. 34) с использованием блока цикла. (Отметим, что в той схеме заголовку цикла принадлежат блоки $x := xn$, $x := x + h$ и $x \le xk$, а телу цикла – $y := \sin^2 x^2$ и блок вывода).

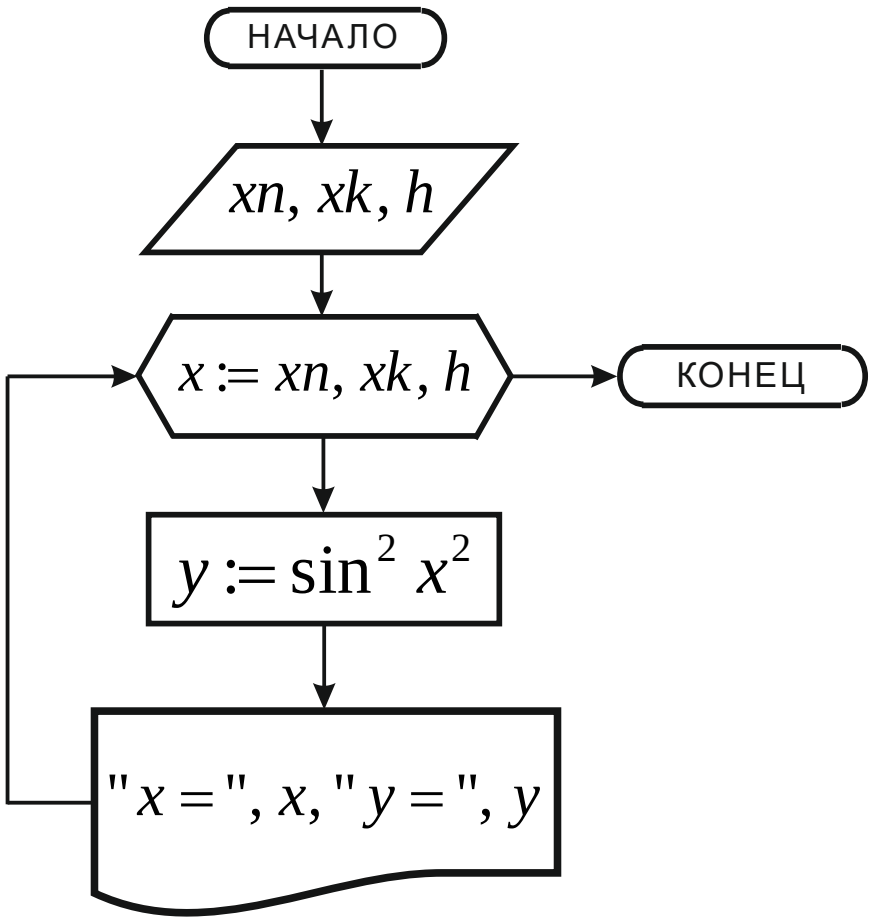

Рис. 36. Блок-схема циклического алгоритма (более рациональная)

После ввода данных сразу организуется цикл с помощью блока, так как в заголовке задаётся первоначальное значение переменной x (параметр цикла) и можно в первом же цикле рассчитать первое значение y. Количество блоков сократилось с 8 до 6, а усилия по составлению схемы уменьшились, так как не нужно продумывать, где и как строить блоки первоначального присваивания, прибавления шага и выхода из цикла. То есть, следуя определённым правилам, можно быстро и безошибочно организовать циклический алгоритм.

Пример 1. **Блок-схема для вычисления факториала числа *n*.** (Напомним, что факториал числа n – это произведение натурального ряда чисел от 1 до n. Факториал нуля принят равным 1). При вычислении факториала мы сначала 1 умножаем на 2 и получаем произведение 2. Затем это произведение умножаем на следующее число в натуральном ряду (3) и получаем новое значение произведения (6). Эти вычисления мы повторяем, пока не достигнем n. Таким образом, мы имеем дело с циклом. Конечное значение произведения и будет равно факториалу числа n. В задаче фигурируют две константы – 1 и n и две переменные – число натурального ряда (обозначим, например, i) и само произведение (обозначим, например, p). В качестве параметра цикла выбираем i, так как про p априорная информация неизвестна. Начальное значение i – 1, конечное – n, шаг изменения – 1. Переменная p начинается с 1, в каждом цикле меняется и принимает конечное значение – факториал числа n. Вводить нужно только n. До цикла первоначальное присваивание p – 1. Алгоритм приведён на рис. 37.

При вычислении текущего значения p старое значение p умножается на очередное i (справа от знака присваивания) и записывается в ячейку памяти p, образуя его новое значение. Выводить нужно только конечный результат после цикла.

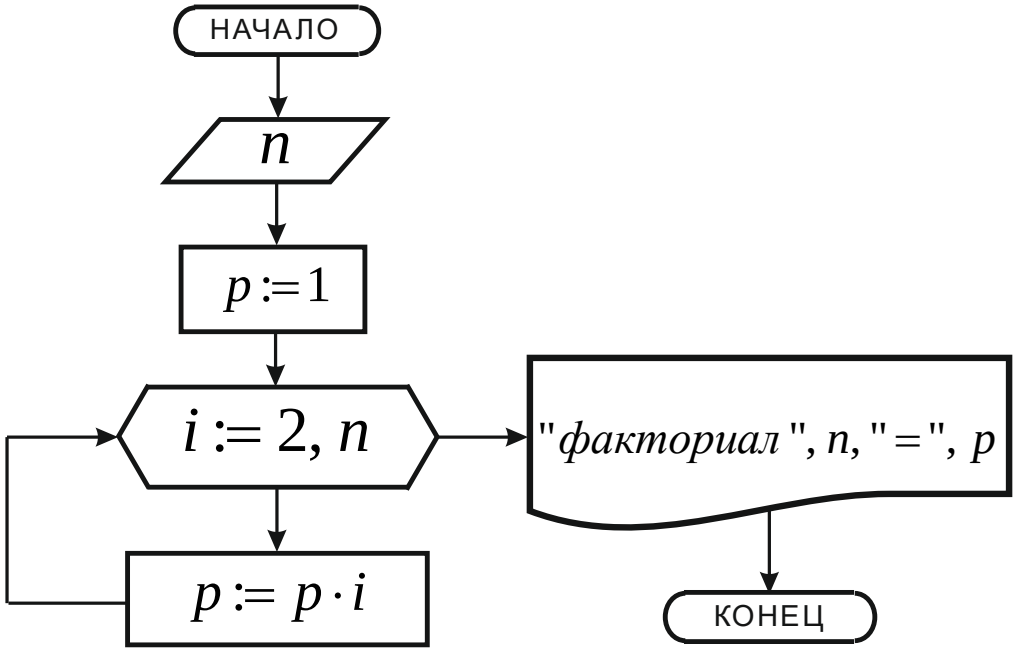

Рис. 37. Блок-схема вычисления факториала числа *n*

Пример 2. **Использование циклов для работы с массивами чисел.** Как известно, массивы чисел (чаще всего одномерные – векторы, и двумерные – матрицы) представляют собой совокупности чисел с совершенно произвольными значениями, но упорядоченные по местоположению. Обратиться к любому элементу массива (взять его для каких-то действий) можно по уникальному индексу, который является просто номером местоположения. Индексы – это натуральный ряд целых чисел, начинающийся чаще всего с 0 или 1 и заканчивающийся максимальным номером. Матрицы имеют два индекса (строки и столбцы). Понятно, что для перебора всех (или какой-то части) элементов массива требуются циклы. Удобными параметрами цикла при этом являются индексы (известно начальное значение индекса, шаг изменения (1) и количество элементов массива).

Для работы с массивами их нужно сначала ввести. Делается это в цикле. То же касается вывода массивов. Рассмотрим задачу ввода вектора x от 1 до n (в алгебре обозначается $\{x_i\}_n$, i – индекс), умножения каждого элемента на 2 и вывода полученного вектора. Если решать задачу поэтапно, получится блок-схема – рис. 38.

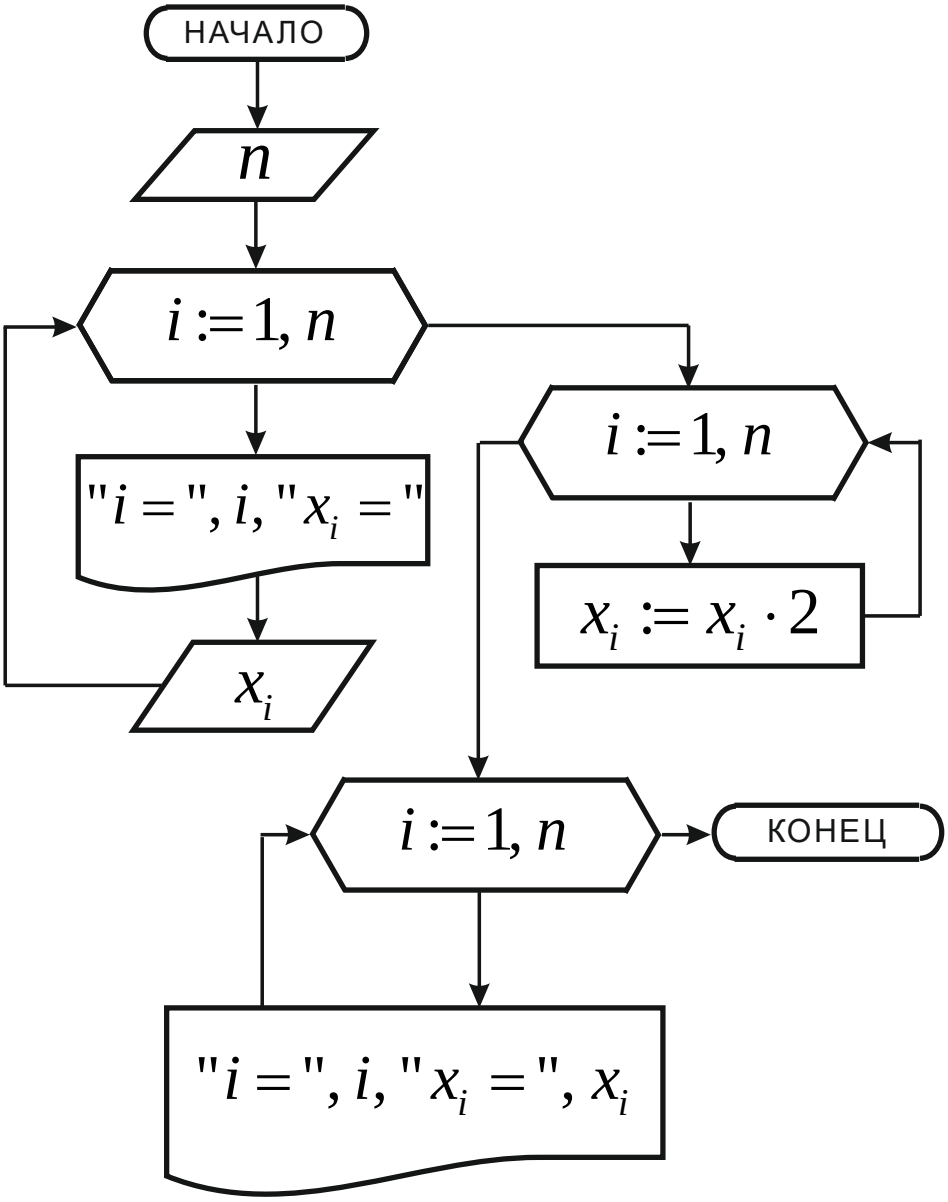

Рис. 38. Блок-схема для ввода и вывода вектора $\{x_i\}_n$ (нерациональная)

В этой схеме имеется три последовательных цикла с одинаковыми заголовками. Их можно объединить в один – рис. 39; объединяются тела циклов. (В данном случае ввод лучше осуществлять с клавиатуры, а вывод – в файл, чтобы не перемежать ввод и вывод на одном устройстве).

Именно таким образом осуществляется ввод и вывод массивов в профессиональных языках программирования. В блок-схемах применяют упрощённую схематичную запись – рис. 40.

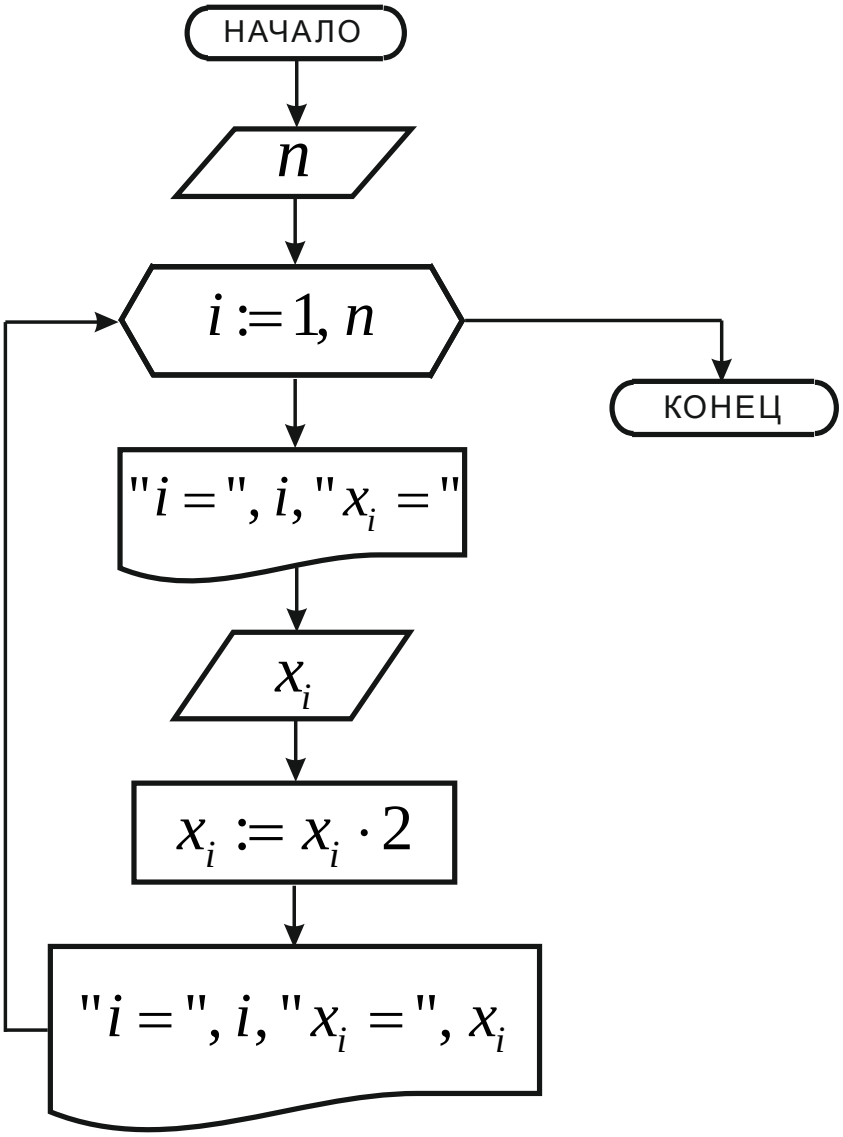

Рис. 39. Блок-схема для ввода и вывода вектора $\{x_i\}_n$
(более рациональная)

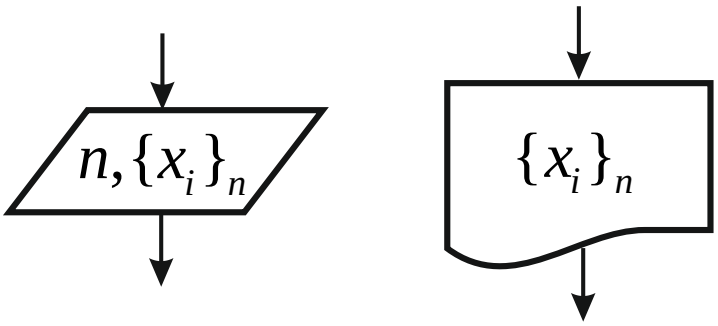

Рис. 40. Схематичный ввод и вывод массивов

Пример 3. **Ввод и вывод двумерных массивов. Вложенные циклы.** При вводе и выводе матриц необходимо задавать два индекса. В одном цикле можно задать изменение только одного индекса, поэтому после этого внутри цикла нужно открыть ещё один цикл, в котором задать изменение второго индекса. То есть один цикл можно целиком вставить внутрь другого цикла (он будет телом или частью тела второго цикла). Это **цикл в цикле** или **вложенный цикл**. Глубина вложений не лимитируется. При вводе матриц во **внешнем цикле** можно задавать номер строки (ввод по строкам) или номер столбца (ввод по столбцам). Во **внутреннем цикле** перебираются все его индексы при фиксированном индексе внешнего цикла. Затем изменяется индекс внешнего цикла и опять исполняется весь внутренний цикл.

Решим задачу примера 2 для двумерного массива $\{x_{i,j}\}_{n,m}$. Используем упрощённый ввод-вывод и вложенный цикл для преобразования элементов матрицы – рис. 41. На схеме представлена работа с матрицей по строкам, для работы по столбцам нужно поменять местами заголовки двух циклов.

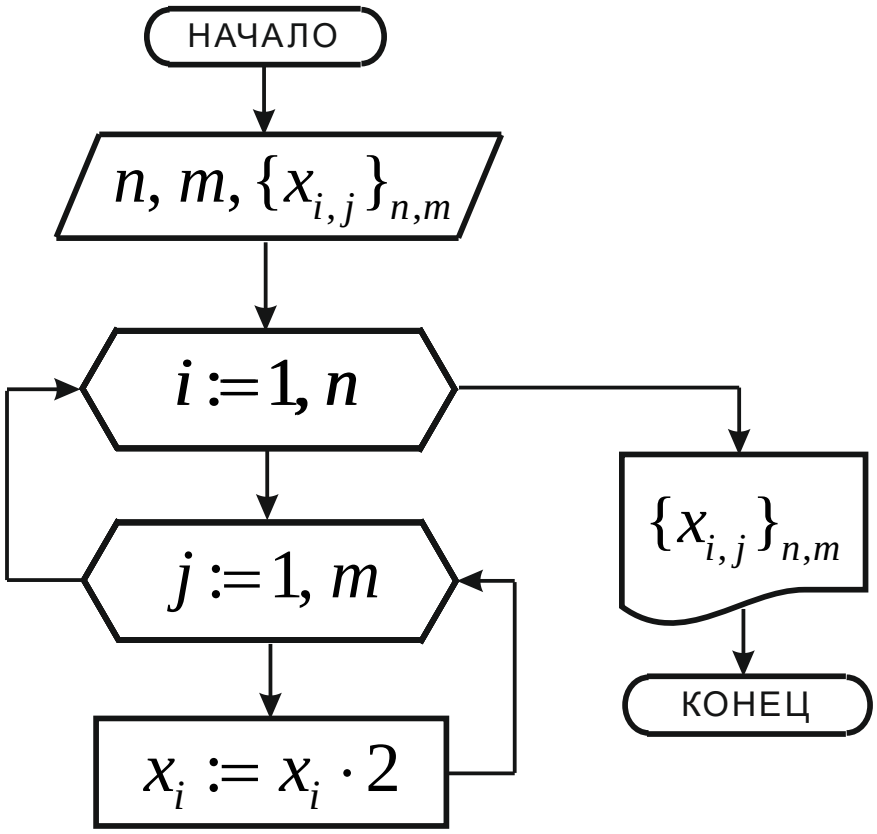

Рис. 41. Работа с двумерными массивами. Вложенные циклы

Пример 4. **Накопление суммы и произведения в программировании.** Накопление суммы. Когда мы, например, собираем грибы, у нас имеется корзинка (аналог ячейки памяти), вначале **пустая**. При сборе грибов мы к имеющимся в корзине грибам добавляем вновь найденный. Новое количество грибов получается как старое количество плюс новый гриб (операция присваивания!). Сбор грибов – циклический алгоритм.

При накоплении суммы в программировании она сначала обнуляется, а затем последовательно накапливается в цикле путём прибавления очередного слагаемого. Если имеется свободное слагаемое, то можно начинать с него, а не с нуля.

При накоплении произведения, оно сначала объединичивается, а затем последовательно накапливается в цикле путём умножения на очередной множитель.

Понятно, что произведение вначале обнулять нельзя. Эти правила часто используются в прикладном программировании вычислительных задач. Примером накопления произведения является вычисление факториала (см. рис. 37).

Задача: дан двумерный массив $\{x_{i,j}\}_{n,m}$. Найти сумму неотрицательных элементов. Решение: нужно перебрать все элементы массива (вложенный цикл), проанализировать их и вычислить соответствующую сумму (обозначим, например, S) – рис. 42.

Пример 5. **Простейшие алгоритмы поиска.** Задача: в массиве $\{a_i\}_n$ найти максимальный и минимальный элементы и переставить их местами. Пример: 5, 2, 4, 8, 3 (в общем случае это могут быть любые действительные числа, положительные или отрицательные; для простоты примем, что они не повторяются). Здесь важно разделять числовое значение элемента и его местоположение, которое определяется индексом. Максимальный – 8, стоящий на 4-м месте; минимальный – 2, стоящий на 2-м месте. Перестановка означает, что максимальный должен стоять на 2-м месте, а минимальный – на 4-м.

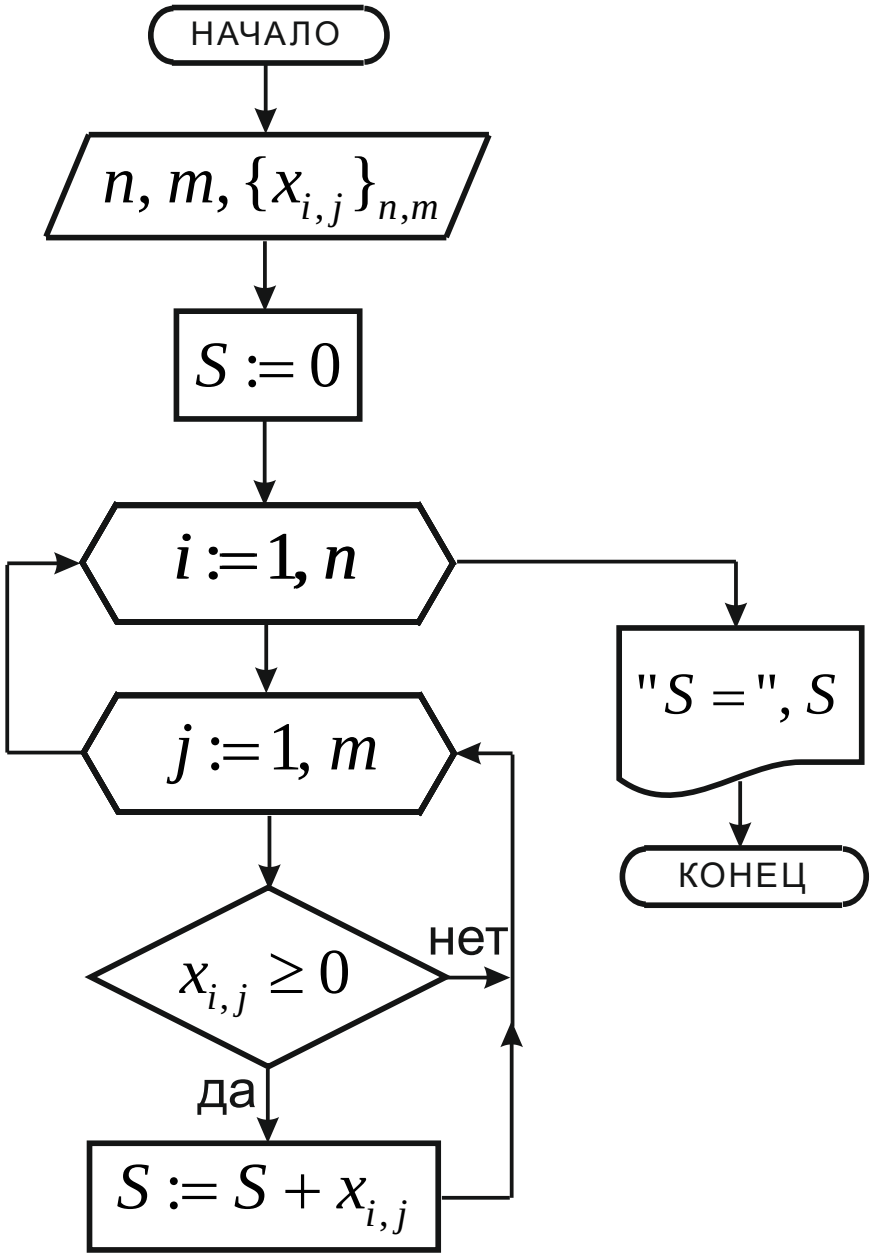

Рис. 42. Алгоритм вычисления суммы неотрицательных элементов матрицы

Чтобы найти максимальный элемент, нужно последовательно (т. е. в цикле!) сравнивать каждый элемент с каким-то выбранным объектом (например, заведомо минимальным) и, если встречается больший элемент, присвоить его значение этому объекту, т. е. объект должен быть переменной, отдельной от массива (обозначим его, например, *Max* – имя из трёх символов). С одной стороны, для действительных чисел не существует заведомо минимального числа; с другой стороны, первоначально присвоить *Max* любое число нельзя,

так как это может привести к ошибке, если выбранное значение окажется большим, чем любой из элементов. Поэтому *Max* можно присвоить значение любого из элементов массива, лучше всего – первого (a_1), так как их количество может быть разным, а первый есть всегда. Кроме этого, требуется отслеживать местоположение максимального (на данный момент) элемента (обозначим его переменной *iMax*, это **целочисленная** переменная). Первоначальное присваивание *iMax* – индекс выбранного для *Max* элемента массива (т. е. 1). Поскольку вспомогательные переменные вначале имеют значение первого элемента массива, цикл надо начинать с 2, а не с 1. После перебора всех элементов массива и выхода из цикла *Max* получит значение максимального элемента (8), а *iMax* – его местоположения (4) – рис. 43.

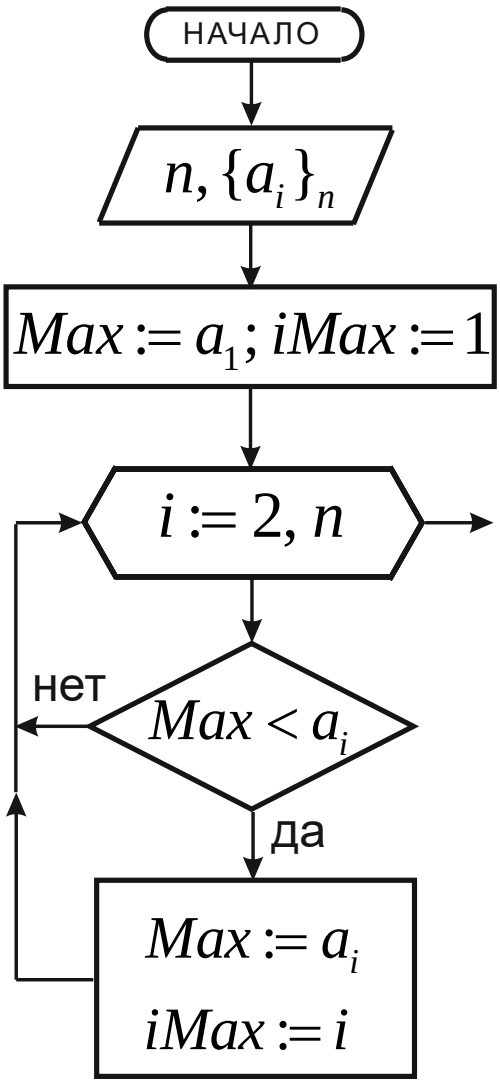

Рис. 43. Первый этап составления
блок-схемы алгоритма поиска

Добавим в эту схему поиск минимального значения по такому же алгоритму (соответственно – *Min* и *iMin*) – рис. 44. Поиск минимума в теле цикла будет последовательно вслед за поиском максимума. Первоначальные присваивания *Min* и *iMin* можно сделать те же самые.

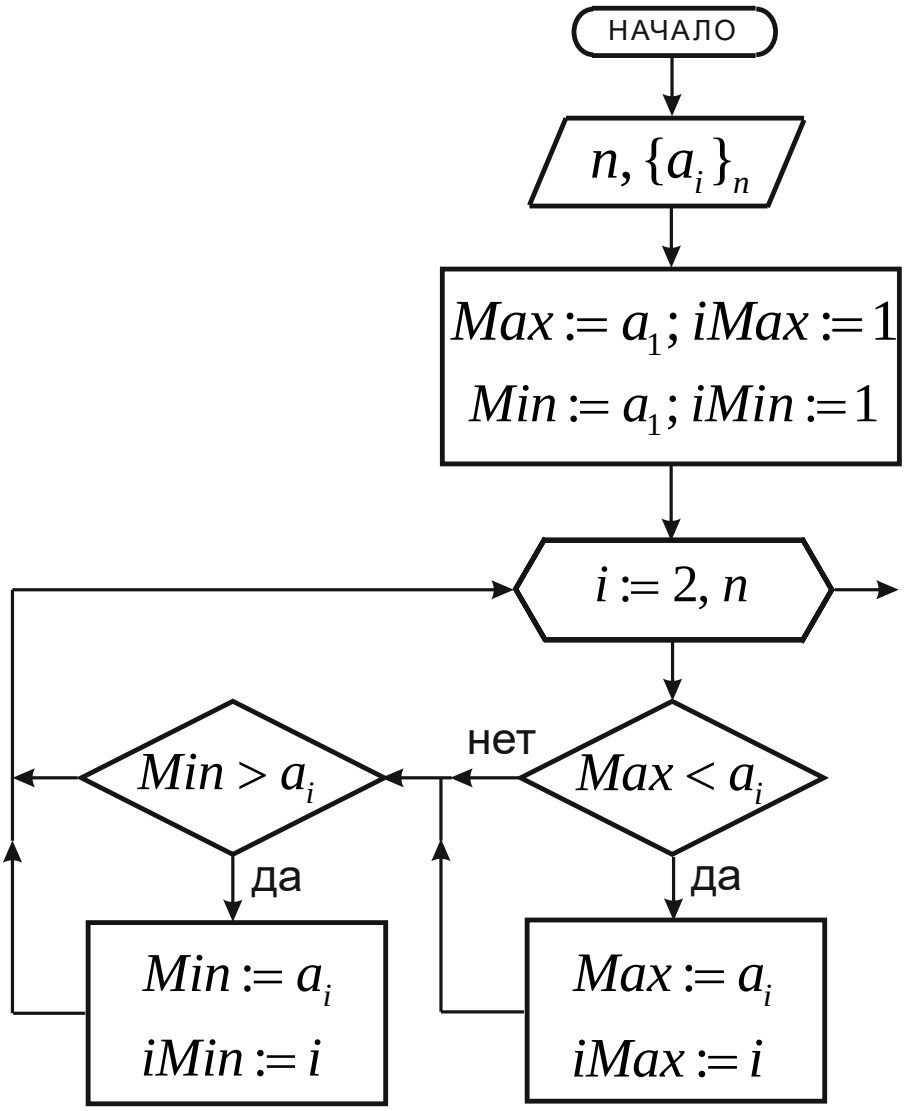

Рис. 44. Второй этап составления блок-схемы алгоритма поиска

После выхода из цикла вспомогательные переменные получат значения: $Max = 8$, $iMax = 4$, $Min = 2$ и $iMin = 2$. Кроме этого, максимальное и минимальное значения останутся в элементах: $a_4 = 8$ и $a_2 = 2$. Перестановка не означает, что у этих двух элементов надо поменять индексы, нужно в эти ячейки памяти записать соответствующие значения. Сделать это можно несколькими способами. Например,

можно в один элемент записать значение другого: $a_{iMax} := a_{iMin}$. После этого a_{iMax} получит своё окончательное значение, но потеряет старое. Тогда обратное присваивание $a_{iMin} := a_{iMax}$ будет неверным. Но старое значение a_{iMax} сохраняется ещё в переменной Max, поэтому $a_{iMin} := Max$. Переставлять местами эти два присваивания нельзя. После первого присваивания исходный массив примет вид: 5, 2, 4, 2, 3, а после второго – 5, 8, 4, 2, 3, что и требовалось. Второй способ (аналогичный): $a_{iMin} := a_{iMax}$, $a_{iMax} := Min$. Пожалуй самым рациональным будет способ: $a_{iMax} := Min$, $a_{iMin} := Max$, причём в этом случае присваивания можно менять местами. Результирующий алгоритм представлен на рис. 45.

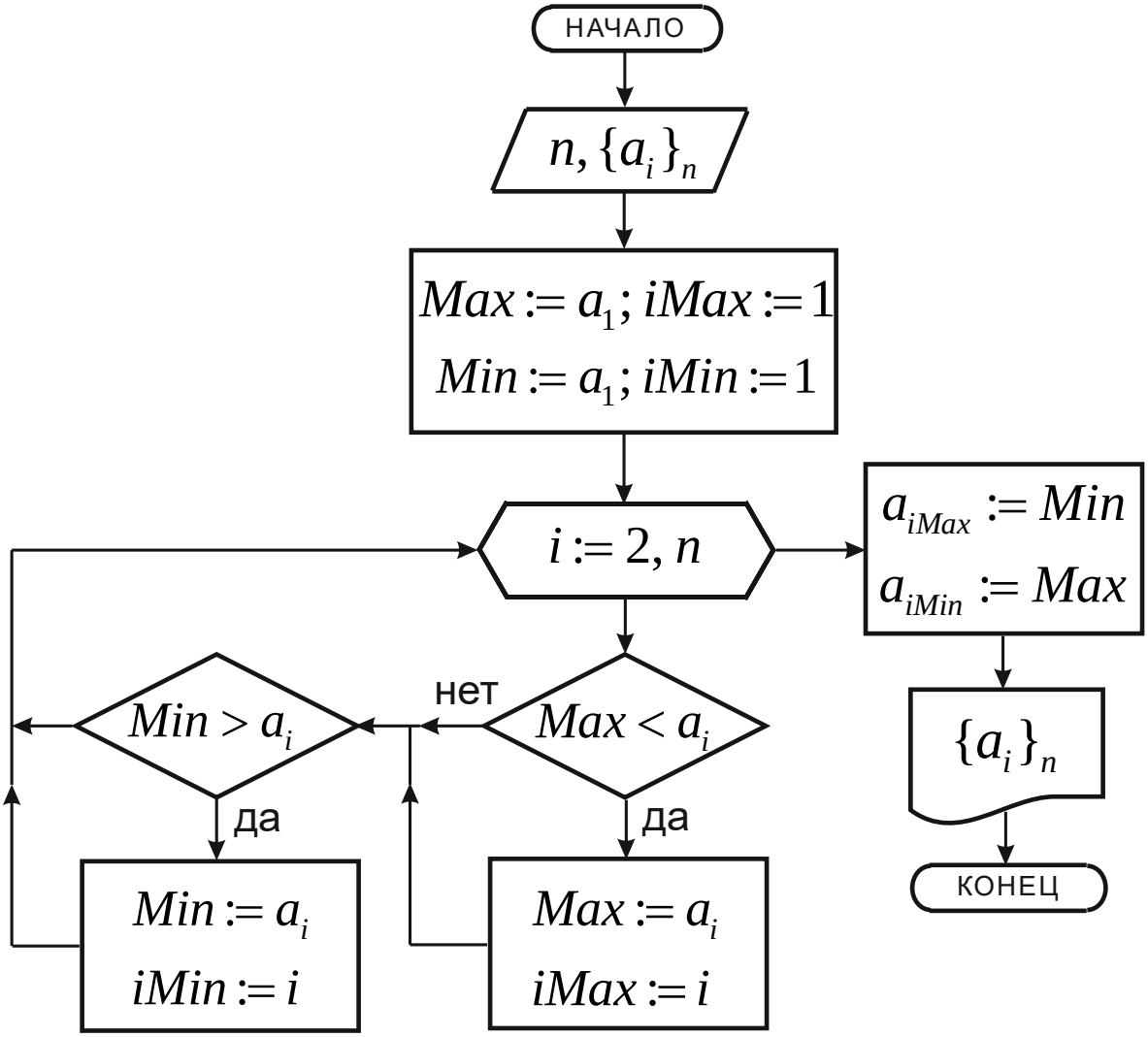

Рис. 45. Блок-схема поиска максимального и минимального элементов и их перестановки

Нужно заметить, что такой простой алгоритм поиска неприменим в случае очень больших массивов. В современном программировании разработаны гораздо более мощные алгоритмы, эта задача давно превратилась в отдельную отрасль и высоко актуальна, в том числе в Интернете.

Пример 6. **Задачи сортировки.** Рассмотрим алгоритм задачи ранжирования – расстановки элементов массива $\{a_i\}_n$ в порядке возрастания. Возьмём тот же пример: 5, 2, 4, 8, 3. Самый простой и очевидный алгоритм сводится к следующему. В исходном массиве находим минимальный элемент и ставим его на первое место. Поиск и обмен местами (в данном случае – с первым элементом) нам уже известен (см. пример 5). Полученный таким образом первый элемент закрепляем на своём месте и в дальнейшем не трогаем. Среди оставшихся четырёх отыскиваем минимальный, ставим его на второе место и закрепляем. И так далее. Вырисовываются два цикла: поиск минимума (внутренний) и сужение поля поиска после установки очередного элемента на своём месте (внешний). В нашем примере таких полей поиска будет 4: пятое поле будет состоять из одного элемента и не будет требовать анализа.

Сначала составим алгоритм для первого поля поиска (5, 2, 4, 8, 3), обозначим счетчик цикла j ($j=1$). Используя опыт примера 5, имеем – рис. 46. После выполнения этого алгоритма получим массив 2, 5, 4, 8, 3, где первый элемент уже стоит на своём месте.

Следующее поле поиска – 5, 4, 8, 3 ($j=2$). Расширим алгоритм для общего случая. Для этого полученный цикл вставим внутрь внешнего цикла по параметру j. Внесём изменения во внутренний цикл: 1 заменяем на j, а 2 в заголовке цикла – на $j+1$. Внешний цикл, как уже отмечалось, будет выполняться до $n-1$. Окончательно получим – рис. 47.

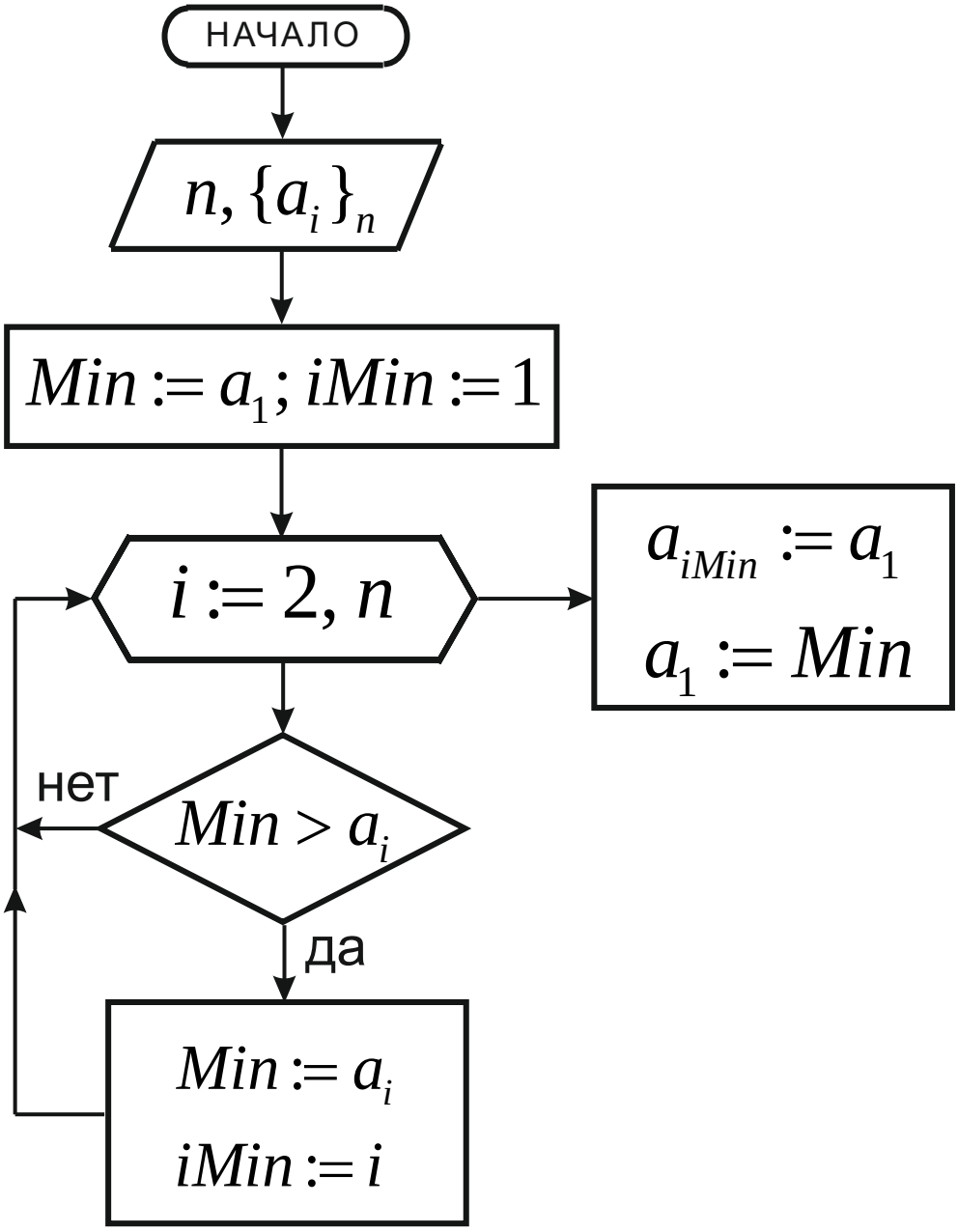

Рис. 46. Первый этап составления блок-схемы
алгоритма ранжирования

Рассмотрим подробно работу этого алгоритма. Схематично это можно представить так (см. рис. 47):

j=1; *Min*=5; *iMin*=1; *i*=2; $Min > a_i$? т. е. 5>2 ?; да; *Min*=2; *iMin*=2; *i*=3; $Min > a_i$? т. е. 2>4 ?; нет; *i*=4; $Min > a_i$? т. е. 2>8 ?; нет; *i*=5; $Min > a_i$? т. е. 2>3 ?; нет; *i*=6; выход из внутреннего цикла; преобразование массива: 5, 5, 4, 8, 3 затем 2, 5, 4, 8, 3; *j*=2 и так далее.

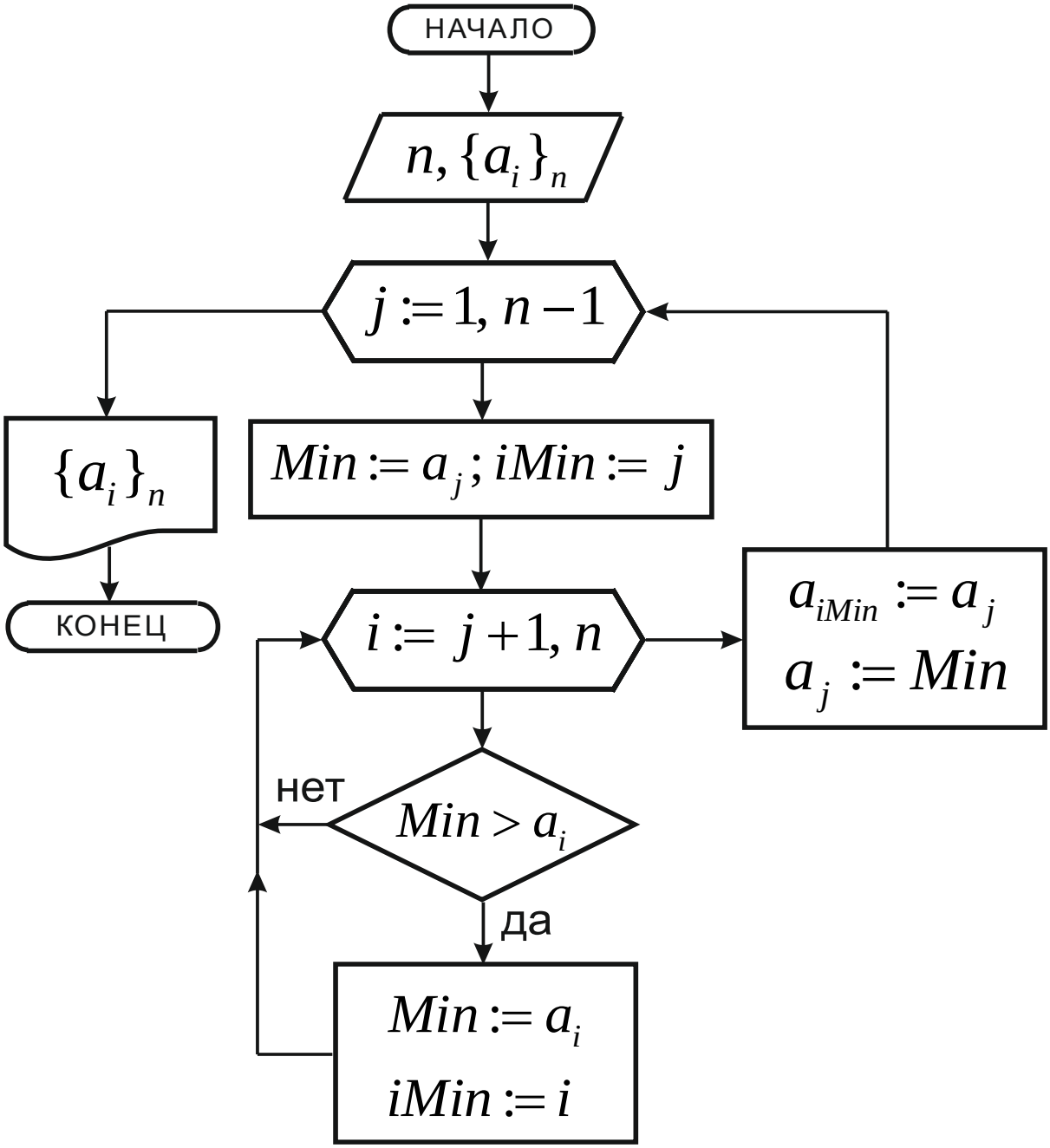

Рис. 47. Блок-схема алгоритма ранжирования; окончательный вариант

Очень полезно для усвоения материала пройти эту схему до конца. В табл. 1 представлено, как последовательно меняется содержимое ячеек памяти соответствующих переменных в ходе исполнения алгоритма.

Таблица 1. Изменение переменных *j*, *Min*, *iMin*, *i* в ходе исполнения алгоритма (см. рис. 47) и соответствующие преобразования массива

n	5				
j	1	2	3	4	5
Min	5, 2	5, 4, 3	4	8, 5	
iMin	1, 2	2, 3, 5	3	4, 5	
i	2, 3, 4, 5, 6	3, 4, 5, 6	4, 5, 6	5, 6	
преобразование массива	5, 5, 4, 8, 3 2, 5, 4, 8, 3	2, 5, 4, 8, 5 2, 3, 4, 8, 5	2, 3, 4, 8, 5 2, 3, 4, 8, 5	2, 3, 4, 8, 8 2, 3, 4, 5, 8	

Исходный массив 5, 2, 4, 8, 3.

Преобразованный после ранжирования массив: 2, 3, 4, 5, 8.

§ 4. Неявные и итерационные циклы

В циклах часто встречаются случаи, когда закон изменения параметра цикла не сводится к прибавлению шага, а более сложен. Использовать блок цикла при этом нельзя, и блок-схемы составляются из обычных блоков, т. е. неявно (подобно схеме рис. 34).

Пример 1

Рассчитаем несколько значений y при различных x:

$y = \sin^2 x^2$ для $x = 2, 6, 18, 54, 162$.

Можно заметить, что каждое следующее число получается из предыдущего путём умножения на 3 (обозначим этот множитель f). Параметром цикла выберем x. Допустим, что задано начальное значение (обозначим $xn = 2$), число циклов ($n = 5$) и множитель. Использовать блок цикла в этом случае нельзя. Конечное значение параметра цикла можно найти по формуле $xk = xn \cdot f^{n-1}$ (начальное значение xn умножается на f $(n-1)$ раз). Результат представлен на рис. 48.

(Заметим, что эта задача может быть решена с использованием счётчика цикла от 1 до n без необходимости определения конечного значения x по формуле).

Пример 2

Решим похожую задачу для

$y = \sin^2 x^2$ для $x = 3, 9, 81, 6561$.

Здесь каждое следующее значение x получается возведением в степень 2 (обозначим f) предыдущего. Допустим, что задано начальное значение (обозначим $xn = 3$), конечное значение ($xk = 6561$) и степень. Алгоритм представлен на рис. 49.

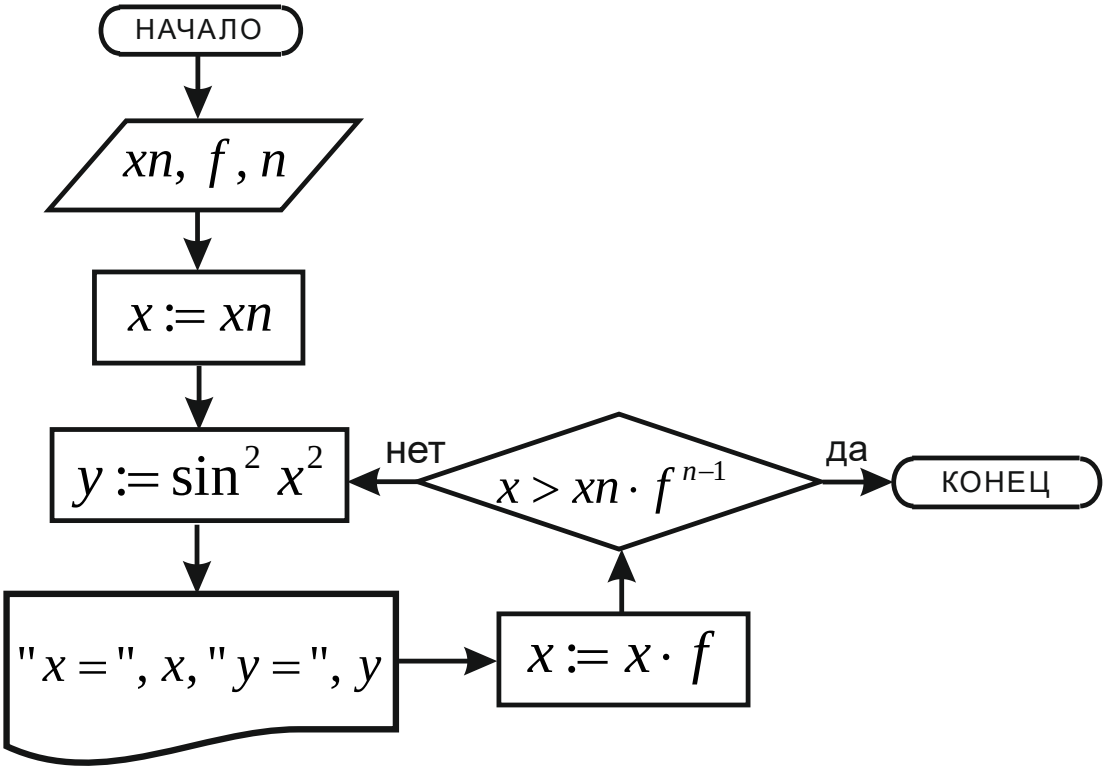

Рис. 48. Блок-схема вычисления функции в неявном цикле (пример 1)

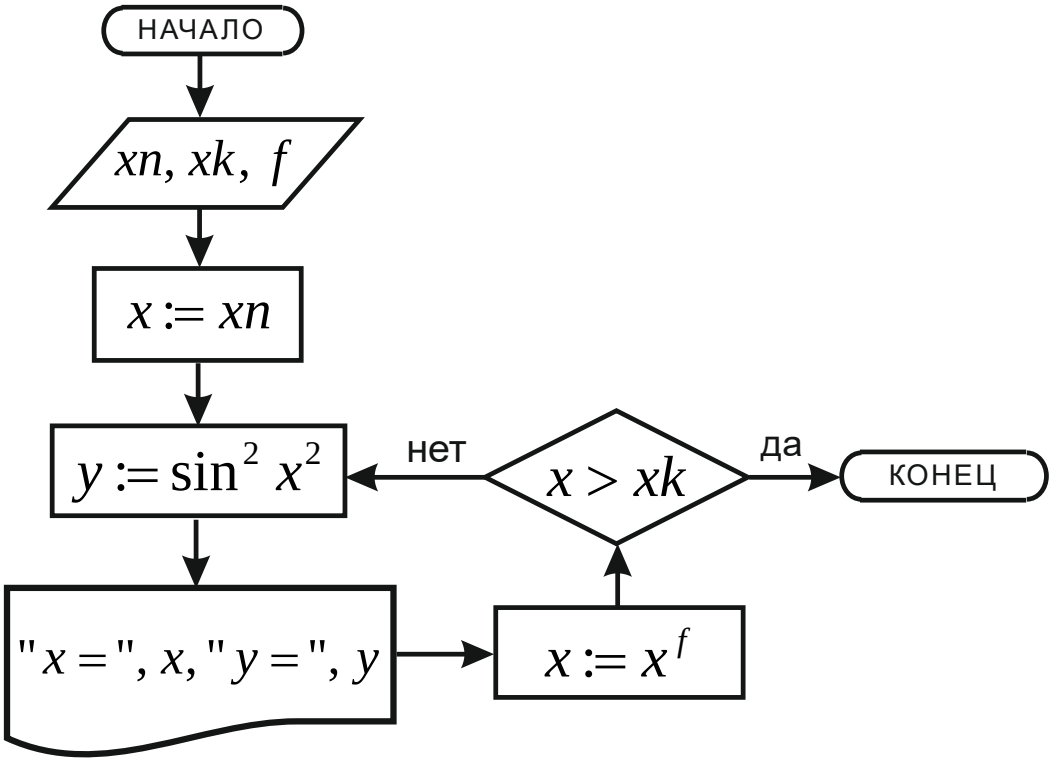

Рис. 49. Блок-схема вычисления функции в неявном цикле (пример 2)

Итерационным циклом называется цикл, выход из которого осуществляется не по параметру цикла, а по какому-то другому, независимому условию. Число циклов (или конечное значение параметра цикла) при этом заранее неизвестно. **Итерация** – это последовательное приближение к чему-либо. Итерационные процессы часто встречаются в вычислительной математике и осуществляются с помощью итерационных циклов. Например, нужно найти с какой-то точностью ε сумму бесконечного сходящегося ряда

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...+\frac{1}{n}+...$$

Каждый следующий член ряда будет меньше предыдущего и будет вносить всё меньшую долю в накапливаемую сумму. Наконец, наступит момент, когда сумма перестанет расти в пределах заданной точности. Заранее неизвестно, сколько циклов накопления суммы придётся сделать, поэтому цикл будет итерационным. Параметром цикла выступает номер члена ряда n. Каждый следующий член ряда (обозначим его, например, p) вычисляется по формуле $p=\frac{1}{n}$ и суммируется. Выход из цикла производится по условию $p<\varepsilon$, т. е. количество членов ряда, которое потребуется для суммирования, будет зависеть от точности. При решении задач такого рода нужно обязательно проверить, с какого номера n начинается ряд, с 0 или с 1. Иногда сумму (обозначим, например, S) нужно начинать не с 0, а с первого члена ряда. В нашем случае $n=1$ и $S=0$. Алгоритм вполне очевиден – рис. 50 (в схеме ε обозначено как *eps*).

Бывают случаи, когда очередной член ряда рациональнее вычислять не по общей формуле, а исходя из предыдущего члена. Часто это бывает, когда в формулу входит факториал, который не рационально вычислять каждый раз заново. Рассмотрим ряд

$$1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\frac{x^4}{4!}+...+\frac{x^n}{n!}+...$$

Можно заметить, что каждый следующий член ряда получается из предыдущего путём умножения его на множитель $\frac{x}{n}$. В данном случае ряд начинается с номера 0, поэтому суммирование лучше начать со второго члена $n=1$, тогда первоначальное присваивание $S:=1$ и $p:=1$. Алгоритм представлен на рис. 51.

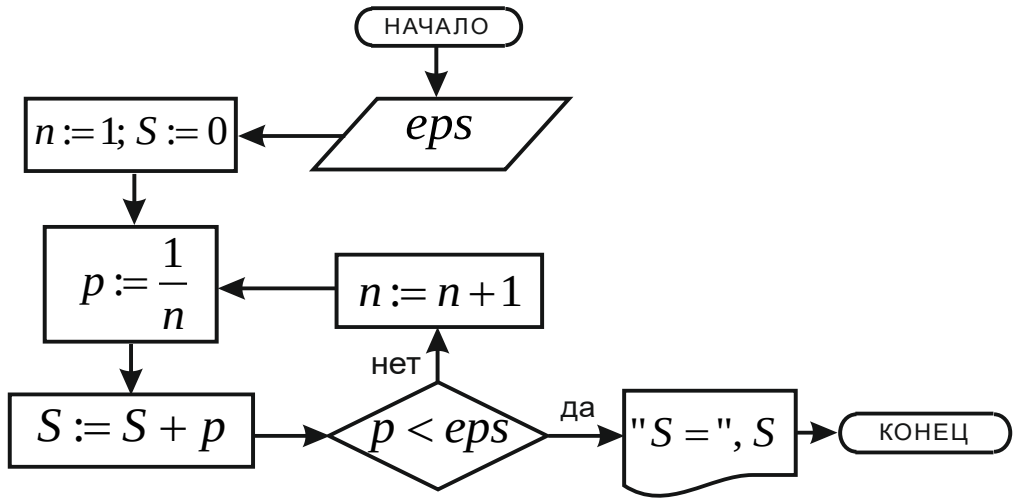

Рис. 50. Итерационный цикл вычисления суммы сходящегося ряда (пример 1)

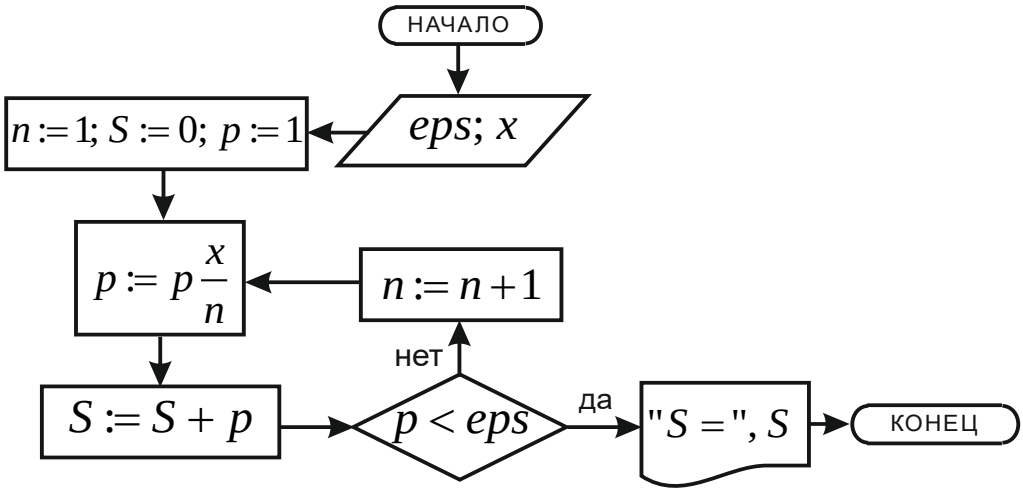

Рис. 51. Итерационный цикл вычисления суммы сходящегося ряда (пример 2)

Очень часто сходящиеся ряды имеют члены с одной общей формулой, но с чередующимся знаком впереди, который зависит от номера члена ряда. Обычно знак формируется множителем $(-1)^n$ или $(-1)^{n+1}$, который включается в формулу (его работу всегда надо проверять). Однако в силу особенностей вычисления степени в конкретных языках программирования возводить отрицательное число в степень нежелательно, поэтому такой множитель лучше заменить на $\cos(n\pi)$ $\cos((n+1)\pi)$. Следует отметить, что в обоих примерах параметром цикла является номер члена ряда n.

Глава 2. ЯЗЫК ПРОГРАММИРОВАНИЯ ПАСКАЛЬ

Паскаль – базовый язык таких профессиональных языков программирования, как Delphi, Lasarus и др. При работе в консольных приложениях программы на Паскале могут быть практически без переделок использованы в этих языках.

§ 1. Алфавит языка

Алфавит устанавливает строгий перечень символов, которые могут использоваться в Паскале:

1. Буквы латинского алфавита A – Z или a – z; различия между строчными и прописными буквами в классическом Паскале нет.

2. Арабские цифры 0 – 9.

3. Шестнадцатиричные цифры (в пособии рассматриваться не будут).

4. Специальные знаки:

а) *символы:*

+ - * / = , ‘ . : ; < > [ ] ( ) { } ^ @ $ # _

б) *пары символов:*

<> <= >= := (* *) (. .)

(. аналогично [, .) аналогично].

в) *невидимые символы:*

пробел, перевод строки (возврат каретки, клавиша Enter).

г) *служебные слова* – специально зарезервированные слова (выделяются подчёркиванием или жирным шрифтом), которые используются строго по своему назначению и не могут использоваться для других целей. В частности, они не могут служить именами переменных. Например, служебное слово **begin**; нельзя называть переменные таким именем, но можно назвать, скажем, begin1, begin33 и т. д. Служебные слова могут, например, входить в операторы и участвовать в построении алгоритма. В операторы циклов входят служебные слова **for, to, downto, do, repeat, until, while**. Для целей, ориентированных на вычислительные задачи, потребуются далеко не все служебные слова, содержащиеся в языке; мы будем знакомиться с ними по мере изложения материала.

Перечень **используемых** служебных слов:
and – логическое «и»,
array – «массив»,
begin – «начало»,
case – «случай»,
const – «константа»,
div – целочисленное деление,
do – «выполнить»,
downto – «вниз к»,
else – «иначе»,
end – «конец»,
file – «файл»,
for – «для»,
function – «функция»,
goto – «идти к»,
if – «если»,
label – «метка»,
mod – остаток от деления,
not – логическое «не»,
of – «из»,
or – логическое «или»
procedure – «процедура»,
program – «программа»,
repeat – «повторять»,
string – «строка»,
then – «то»,
to – «до», «к»,
type – «тип»,
until – «до тех пор пока»,
uses – «использовать»,
var – «переменная»,
while – «пока»

Кроме объектов алфавита языка, в программе могут встречаться **стандартные директивы** и **директивы препроцессора**. Последние служат, например, для подключения модулей, расширяющих возможности языка: математического модуля, графического и др. Они в сам язык не входят и зависят от его конкретной реализации какой-то фирмой (например, язык Delphi).

§ 2. Идентификаторы

Идентификаторы – это имена констант, переменных, меток, типов, процедур, функций, программ и других объектов языка, не совпадающие со служебными словами и составленные по определённым правилам. В классическом Паскале идентификатор состоит не более чем из 8 символов, может включать латинские буквы, арабские цифры и знак подчёркивания, не может начинаться с цифры. Пробелы и другие специальные символы не допускаются. Знак подчёркивания можно использовать для имитации пробелов между словами. Например: a, bb, vasia_g3, petia777, _ss20. В Delphi количество символов может достигать 64.

В качестве идентификаторов нельзя использовать служебные слова и не рекомендуется использовать названия типов (см. ниже), а также стандартные директивы, функции (например – математические) и процедуры, иначе те перестанут работать.

§ 3. Константы

Это данные, которые не меняют своего первоначального значения в ходе исполнения программы. Константы бывают **безымянные** (это просто сами числа или, например, **false**, **true** и т. д.) и **именованные**, т. е. имеющие имя, образованное как идентификатор. Именованные константы имеют ячейку памяти определённого класса памяти; обратиться к ней можно по имени постоянной. Константы, которые нами будут использоваться:

1. Целые от –2147483648 до +2147483647 со знаком и без.

2. Вещественные со знаком и без, с экспоненциальной частью и без; **обязательно** – с **десятичной точкой,** по ней определяется **тип** константы (целая она или действительная).

3. Логические **false** и **true**.

4. Символы – любой символ ПК в апострофах: ‘Ф’, ‘в’, ‘?’, ‘щ’, ‘!’, ‘G’, ‘@’, ‘k’ и т. д. Прописные и строчные буквенные символы в Паскале различаются.

5. Строки символов – любая последовательность символов (кроме CR), заключённая в апострофы. В том числе допускаются слова с пробелами и на кириллице. Смысл строк для компьютера закрыт, для него это просто набор символов. Строки могут использоваться в программах для вывода пояснительной информации при расчётах.

Примеры:

‘Слово word’

‘ergy567$%*&@+)4592ллрот кырнгоЕН#**&?SERttty65’

‘Pascal’

§ 4. Операции

Операции – конструкции языка, в которых **знак операции** действует на **операнды**; операция имеет **результат**. По количеству операндов операции подразделяются на унарные и бинарные (в языке C++ имеется тернарная операция). Формат бинарных операций:

<операнд 1><знак операции><операнд 2>.

Формат унарных операций:

<знак операции><операнд>.

Например, существует 6 операций сравнения: $>, <, \geq, \leq, =, \neq$, им в языке Паскаль соответствует 6 знаков операций: >, <, >=, <=, =, <>. Операндами простых логических операций являются арифметические выражения, например:

<а. в. 1><знак сравнения><а. в. 2> ;

x<=89

(5-d)/6<3

sin(3*g)<>5–k

Для сложных логических выражений (знаки операций: **and** – логическое «и», **or** – логическое «или», **not** – логическое «не») операндами являются другие **логические выражения**. Операция **not –** унарная.

<л. в. 1><знак логической операции><л. в. 2> ,

not <л. в.>.

Примеры:

(x<–5) **or** (x>8)

(x>–5) **and** (x<8)

(x2>–4) **and** ((x<–5) **or** (x>8))

not x1>=3

Операция присваивания:

<имя ячейки памяти переменной>:=<выражение>.

Переменная и выражение должны соответствовать друг другу по типу: например – целочисленная переменная или действительная; если же переменная – логического типа или символьная – выражение не может быть числовым. Таким образом, для числовых переменных:

<имя числовой переменной>:=<а. в.>.

Примеры:

f4:=563.4

g3:=1.e-63

s4:=sin(6*x)+f1*h7

Операции выполняются строго в порядке их **приоритета**. В алгебре тоже существует приоритет: сначала выполняется умножение и деление, затем сложение и вычитание. Приоритет операций в программировании значительно сложнее. Для Паскаля:

1. Первый приоритет (высший):
   - ✓ Логическое «не» **not.**
   - ✓ Взятие адреса **@**.

   (Все эти операции унарные).
2. Второй приоритет. Мультипликативные операции:
   - ✓ Умножение *****.
   - ✓ Деление **/.**
   - ✓ Целочисленное деление **div**.
   - ✓ Остаток от целочисленного деления **mod**.
   - ✓ Логическое «и» **and**.
   - ✓ Левый сдвиг **shl** (нами не используется).
   - ✓ Правый сдвиг **shr** (нами не используется).
3. Третий приоритет. Аддитивные операции:
   - ✓ Сложение **+**.
   - ✓ Вычитание **–**.
   - ✓ Логическое «или» **or**.
   - ✓ Исключающее «или» **xor** (нами не используется).
4. Четвёртый приоритет: Знаки сравнения >, <, >=, <=, =, <>.

Из перечня следует, что в сложных л. в. с операндами из простых л. в. последние необходимо заключать в круглые скобки, например:

(x7>–5.56) **and** (x7<8.5),

иначе первым будет выполняться сложное л. в.:

–5.56 **and** x7,

при этом будет нарушен его формат (ошибка).

Операция взятия адреса **@**. Ячейки памяти переменных располагаются в общей оперативной памяти и могут занимать несколько элементарных. К каждой элементарной ячейке оперативной памяти можно обратиться (получить доступ к её содержимому) по уникальному адресу. Таким образом, к любой переменной можно обратиться по имени или по адресу первой элементарной ячейки. Взятие адреса приводит к выявлению этого адреса для дальнейшего использования в каких-то целях. Пример: @x3.

Целочисленное деление. Деление целых чисел с отбрасыванием дробной части, если она образуется. Результат операции – наименьшее целое число. Операция неприменима, например, к логическим переменным.

Пример:

8 **div** 3

Результат – 6.

Логическое «и» **and** выполняется (т. е. результат – **true**), если выполняются оба операнда (оба л. в. – **true).** Логическое «или» **or** выполняется (**true**), если выполняется хотя бы одно л. в.

§ 5. Выражения

Это конструкции языка, состоящие из имён переменных и постоянных, числовых, логических, символьных констант, знаков операций, индексированных переменных (элементов массивов), вызовов функций, круглых скобок и других элементов. Выражения представляют собой строгую логическую систему и могут быть подсчитаны. При этом получается результат какого-то определённого типа, например, числового (целого или действительного), логического (**true** или **false**), строкового и т. д. Выражения подсчитываются в соответствии с приоритетом операций, порядок вычисления может быть изменён с помощью круглых скобок. В первую очередь вычисляются функции, так как их аргументы стоят в скобках. На место вызова функции ставится её результат, и вычисления проводятся далее. К моменту вычисления выражений значения всех именованных констант и переменных должны быть определены. Поскольку данный курс ориентирован на научные и технологические вычисления, подробнее рассмотрим так называемые арифметические выражения а. в. (Правильнее их называть «выражения числовых типов» – целых и действительных).

Функции в Паскале имеются стандартные и библиотечные (с точки зрения пользователя между ними различия нет), а также могут создаваться самим программистом. Отличительной особенностью функций является то, что после имени функции, которое образуется по правилам построения идентификаторов, всегда идёт открывающая круглая скобка (за исключением одного случая – функция pi – число π). В круглые скобки заключаются аргументы, в стандартных и библиотечных функциях имеется только один аргумент. Перечислим основные математические функции (по сравнению с языком C++ их очень мало; большее количество содержится в подключаемом математическом модуле). Через «x» обозначим аргумент, который в общем случае может быть а. в.

abs(x) – модуль $|x|$

arctan(x) – arctg x

cos(x) – cos x

exp(x) – экспонента e^x (эта функция – не степень!)

frac(x) – дробная часть аргумента

int(x) – целая часть аргумента

ln(x) – ln x

pi – число π

sin(x) – sin x

sqr(x) – x^2

sqrt(x) – $\sqrt{x}$

Многие практически значимые функции и выражения могут дополнить этот список.

tg x – sin(x)/cos(x)

ctg x – cos(x)/sin(x)

$\arcsin x = \operatorname{arctg}\dfrac{x}{\sqrt{1-x^2}}$ – arctan(x/sqrt(1–sqr(x)))

$\arccos x = 2\operatorname{arctg}\sqrt{\dfrac{1-x}{1+x}}$ – 2*arctan(sqrt((1–x)/(1+x)))

$\lg x$ – ln(x)/ln(10)

степень a числа x – x^a – exp(a*ln(x))

$\left(x^a = b;\ \ln b = a\ln x;\ e^{\ln b} = e^{a\ln x} = b;\ x^a = e^{a\ln x}\right)$

(Эта формула непригодна для $x \le 0$).

Корень n степени представляется как дробная степень:

$\sqrt[5]{x} = x^{1/5}$.

Отрицательную экспоненту (и степень в общем), в Паскале лучше представлять как обратное число:

$$e^{-x} = \frac{1}{e^x}.$$

(из-за особенностей работы функции exp(x)).

§ 6. Правила построения арифметических выражений

1. Основной принцип – последовательная запись отдельных символов в одну строку. Никаких многоэтажных дробей, надстрочных и подстрочных символов не бывает.

2. Программа на Паскале, и арифметическое выражение в частности, представляет собой одну большую строку; разбиение на строки в тексте программы производится для удобства и не влияет на смысл программы. Длинные арифметические выражения могут быть в любом месте разбиты на строки, никакие знаки переноса при этом не ставятся и символы не дублируются (как в математике). Знак умножения «*» ставится обязательно. Следует помнить, что в программировании используется **десятичная точка**, а не десятичная запятая (в том числе при наборе на клавиатуре исходных данных).

3. При использовании умножения и деления нужно помнить, что ввод операций ведётся «от строки», например выражение $\frac{a}{b \cdot c \cdot d}$ можно набирать не a/(b*c*d), а более рационально: a/b/c/d.

4. Операции, входящие в выражение, выполняются строго по приоритету, порядок этот можно изменить с помощью круглых скобок. Таким образом, сначала выполняются функции, затем действия в скобках, а потом сами операции. Внутри скобок порядок действия такой же.

5. Количество открывающих и закрывающих скобок должно быть одинаковым, и их структура должна соответствовать смыслу арифметического выражения и алгоритма в целом.

6. Для обозначения переменных и других объектов нельзя применять греческие и кириллические буквы, так как они не входят в алфавит Паскаля. Их обычно заменяют латинскими эквивалентами из одной или нескольких букв.

Например:

ε – eps,

α – al,

β – bt,

γ – gm,

τ – tau.

7. Индексированные переменные (элементы массива) состоят из имени массива и индексов (одного или нескольких через запятую), которые заключаются в квадратные скобки – a[5], x[4, 3] и т. д.

8. Функции составляются строго по понятиям «имя функции» и «аргумент».

Пример:

$\sin^2 x$ – sqr(sin(x)),

$\sin x^2$ – sin(sqr(x)).

Пример составления арифметического выражения:

$$y = e^{-\ln|a|} - \frac{q^2 + c - \sqrt[3]{z}}{\sqrt{a^2 + b^2}} + \mu \cdot \sin^2 \frac{x^2 + 1}{x^2 - 1} + \varepsilon \cdot \mathrm{tg} \frac{A_{33}}{A_{15}};$$

```
y := 1/exp(ln(abs(a)))–(sqr(q)+c–exp(1/3*ln(z)))/
sqrt(sqr(a)+sqr(b))+mu*sqr(sin((sqr(x)+1)/(sqr(x)+1)))
+eps*sin(a[33]/a[15])/cos(a[33]/a[15]);
```

Фрагмент exp(1/3*ln(z)) может быть набран более рационально: exp(ln(z)/3).

§ 7. Типы данных

7.1. Общие положения

Тип – важнейшее понятие теории программирования, он является атрибутом таких объектов программы, как переменные, постоянные, значения функций, значения выражений и т. д. То есть **данные**, в самом широком смысле этого слова, имеют свой тип. В простейшем представлении типы можно сравнить с разделением чисел на целые и действительные в алгебре. Разумеется, в программировании понятие типа гораздо более широкое и разветвлённое.

Типы характеризуются:

1) **классом памяти**,

2) **множеством допустимых значений**,

3) **множеством допустимых операций**.

В настоящее время **класс памяти** данных подразумевает способы организации хранения объекта в оперативной памяти и сводится к понятиям **статических** и **динамических** данных. **Статические объекты**, как правило, образуются с началом работы программы (точнее – программной единицы) и уничтожаются с её концом. **Динамические объекты** могут образовываться и уничтожаться в ходе исполнения программы средствами самой программы. В понятие **класс памяти** также входит объём оперативной памяти – строго определённое количество **машинных слов** (или **элементарных ячеек**), которое необходимо для хранения данных определённого типа. Например, для хранения действительных чисел памяти требуется больше, чем для хранения целых.

Множество **допустимых значений** данных того или иного типа определяет весь возможный набор этих данных. Например, для целых чисел допустимые значения – числа натурального ряда со знаком, ограниченные определённым максимумом; дробные числа или числа больше этого максимума не входят в область допустимых значений. Для символьных переменных областью допустимых значений являются все символы ПК или сопоставленные с ними целые числа (от 0 до 255). Другие целые или действительные числа не являются данными этого типа.

Допустимые операции строго определены для данных того или иного типа. Например, для числовых типов допустимы арифметические операции сложения, вычитания, умножения; для действительных – деления. Деление для целых чисел может приводить к потере точности, если целые числа не кратны, так как результат также должен быть целым. Логические операции неприменимы для числовых типов, умножение неприменимо для символьных и логических типов и т. д.

Типы подразделяются на **стандартные** и **абстрактные**. Стандартные типы делятся на **базовые** и **производные** и строго определены правилами самого языка. Языком регламентируются области допустимых значений и допустимых операций, а также способы организации памяти для хранения данных этого типа. Производные типы строятся на основе базовых и могут быть или простым их расширением или структурой, параметры которой могут задаваться пользователем.

Абстрактные типы данных практически полностью создаются пользователем, который должен описать и область допустимых значений данных нового типа, и область допустимых операций, часто создавая какие-то новые операции, и, кроме этого, набор нестандартных функций, которые могут действовать на данные этого нового типа. Полное описание абстрактного типа данных называется **классом** (в Delphi и Lasarus – **объектом**) и является основой **объектно-ориентированного программирования**. Последнее выходит за рамки настоящего пособия, так как требует длительного времени для освоения. В то же время подавляющее количество вычислительных задач могут быть решены средствами **процедурно-ориентированного программирования**, на котором и построен материал данного курса.

Поскольку в настоящем пособии имеет место ориентация на вычислительные задачи, будут рассмотрены не все типы, имеющиеся в Паскале, а лишь те, которые необходимы.

Базовые **простые** типы подразделяются на **порядковые** и **вещественные**. Порядковые типы имеют конечное число возможных значений, причём с каждым этим значением сопоставлено **целое число** – **порядковый номер** значения. Вещественные типы порядкового номера не имеют и представлены очень большим количеством значений (теоретически – бесконечным).

7.2. Порядковые типы: целые типы, логический, символьный и тип-диапазон

7.2.1. Целые типы

Целые числа охвачены пятью типами, имеющими различный объём памяти в байтах. Конкретные границы диапазонов представляют собой степени двойки.

byte	$0 \div 255$
shortint	$-128 \div 127$
word	$0 \div 65535$
integer	$-32768 \div 32767$
longint	$-2147483648 \div 2147483647$

(Названия типов служебными словами не являются).

При использовании целых типов всегда нужно представлять, какое максимальное по модулю значение может иметь та или иная переменная и может ли у неё быть знак «минус»; в зависимости от этого

подбирается тип. Наиболее употребимые типы в расчётных алгоритмах – **integer** и **longint**, причём последний используется реже, лишь в необходимых случаях, когда значения целых переменных могут выходить за диапазон **integer**. Целые числа, превышающие пределы **longint**, не могут быть представлены в Паскале, их приходится описывать с потерей точности как действительные числа.

7.2.2. Логический тип

boolean $0 \div 1$

(Название типа служебным словом не является).

Переменные этого типа имеют только два значения: **false** (с этой константой сопоставлено целое число 0) и **true** (с ней сопоставлена 1). Это порядковый тип с диапазоном $0 \div 1$. Этот тип могут иметь как отдельные переменные (как правило в их создании большого смысла нет), так и выражения.

7.2.3. Символьный тип

char $0 \div 255$

(Название типа служебным словом не является).

Это все символы ПК, с каждым из которых сопоставлено целое число. Символы ПК делятся на стандартные символы $0 \div 127$, которые одинаковы для всех кодировок и регионов Земли, и нестандартные $128 \div 255$, которые варьируются в широких пределах. Следует помнить, что значениями переменных этого типа являются именно сами символы, а целые числа лишь сопоставлены с ними. Символьной переменной можно присвоить значение целого числа, но при этом ей будет присвоено значение соответствующего символа. Переменной можно присвоить символ напрямую, при этом он заключается в апострофы.

7.2.4. Тип-диапазон

Общий формат:

<минимальное значение>..<максимальное значение>,

где «..» считается одним символом; «минимальное значение» должно быть меньше «максимального значения»,

«минимальное значение» и «максимальное значение» имеют любой порядковый, кроме типа-диапазона, тип.

Например:

aa:=-2..58; (всего у переменной aa 61 значение),

bb:=’d’..’p’; (всего у переменной bb 13 значений).

В вычислительных алгоритмах тип-диапазон чаще всего используется при описании массивов для задания диапазона изменения их индексов. При этом он заключается в квадратные скобки.

7.2.5. Перечисляемый тип

За пределами нашего рассмотрения.

Операции, функции и процедуры, допустимые для работы с порядковыми типами, можно найти в справочниках.

7.3. Вещественные типы

Отличительная особенность этих числовых типов – наличие в числе **десятичной точки** (в целых типах, например, десятичная точка недопустима). В отличие от порядковых типов здесь число представляется лишь с некоторой точностью. Числа могут быть представлены в виде целой и дробной части, со знаком и без, либо только целой (если есть десятичная точка, такое число всё равно будет вещественным), либо только дробной части. В любых вариантах это так называемая **мантисса** (строго говоря, мантиссой в математике называется число с десятичной точкой перед левым разрядом); к ней может быть добавлен **порядок** – символ «e» или «E» с целым десятичным порядком, со знаком или без.

Примеры:

2563.1562

8.52642 («классическая» мантисса)

0.00052

45600. (действительное число!)

1. (действительное число!)

.1

.000045

2.2463e-05 (число $2.2463 \cdot 10^{-5}$)

45.12E86 (число $45.12 \cdot 10^{86}$)

.00045e5 (число 45.0)

В Паскале имеется пять вещественных типов (табл. 2). (Названия типов служебными словами не являются).

Таблица 2. Вещественные типы

Количество байт	Название	Значащие цифры мантиссы	Порядок
4	**single**	$7 \div 8$	$-45 \div +38$
6	**real**	$11 \div 12$	$-39 \div +38$
8	**double**	$15 \div 16$	$-324 \div +308$
10	**extended**	$19 \div 20$	$-4951 \div +4932$
8	**comp**	$19 \div 20$	$-2^{63}+1 \div +2^{63}-1$

В вычислительных задачах в основном используются типы **real** и, когда требуется повышенная точность, – **extended**. Тип **comp** служит для представления очень больших целых чисел и применяется в основном в бухгалтерии.

Ограничение разрядности мантисс и степеней обусловливает конечность количества значений того или иного типа. Но это количество очень большое, и условно можно считать, что представление действительных чисел континуально (т. е. непрерывно). Приближённость представления вещественных чисел приводит к накоплению погрешностей при операциях с этими числами – так называемые погрешности машинной арифметики. При составлении алгоритмов вычислительных задач нужно стремиться к минимизации действий над числами, когда это возможно. Арифметические операции над целыми числами выполняются точно (кроме деления).

Операции, функции и процедуры, допустимые для работы с вещественными типами, можно найти в справочниках.

7.4. Структурированные типы

Это стандартные типы, производные от базовых, таких как числовые, логические, символьные и др. Данные этих типов объединяются в некую определённую структуру, позволяющую правильно организовать работу с ними. Среди структурированных типов очень широко в вычислительных задачах используются **массивы**, и **строки**; **записи** и **множества** в данном пособии не рассматриваются.

7.4.1. Массивы

Это структурированные типы данных **одного типа**. Массивы в языках программирования схожи с массивами в алгебре. Понятие «**размерность**» массива двояко. С одной стороны, это количество

«размеров»; например, трёхмерный массив можно сопоставить с длиной, шириной и высотой геометрических фигур, причём протяжённость этих размеров может быть разной. Двумерные массивы называются **матрицами**, одномерные – **векторами**. С другой стороны, под **размерностью** понимают количество элементов в каждом «размере». Например, если в трёхмерной матрице размеры содержат 8, 3 и 5 элементов, то можно сказать, что, с одной стороны, размерность этой матрицы равна трём, а с другой стороны, – что это трёхмерная матрица размерностью 8 × 3 × 5. Всего в этой матрице 120 элементов, происходит перемножение, как при нахождении объёма геометрических фигур. Двойственность понятия «размерность» к путанице не приводит, так как по контексту всегда понятно, о чём идёт речь.

Каждый элемент «размера» имеет порядковый номер – **индекс**, по которому можно **обратиться** к любому элементу, т. е. взять его для работы. Каждый «размер» должен иметь свой индекс; традиционно их чаще всего обозначают буквами i, j, k. Следует понимать, что индекс – величина переменная, а пределы изменения индекса – постоянные (в программировании в некоторых случаях они могут быть переменными).

Чтобы получить возможность использовать массивы в программе, их нужно сначала **описать**, т. е. полностью задать строение массива и его размерность. Это выполняет сам программист, руководствуясь требованиями алгоритма, но по строгим правилам, в отличие от создания абстрактных типов данных. Делается это в специальном месте программы, называемом разделом описаний. Затем созданные таким образом массивы используются в основной части программы в соответствии с алгоритмом.

Описание массивов производится по двум вариантам: напрямую и с образованием нового типа данных и последующим созданием на его основе конкретных массивов.

Прямое описание массивов. Задаются имена массивов и их размерность в обоих смыслах, указываются типы элементов, составляющих массив. Формат:

<имя массива>: **array**[<тип-диапазон>] **of** <тип элементов массива>;

имя массива – по правилам образования идентификаторов;

array – служебное слово «массив»;

тип элементов массива – любой тип Паскаля;

тип-диапазон – задаёт пределы изменения индексов; в Паскале индексы могут начинаться с любого целого, в том числе отрицательного, но эта возможность в вычислительных алгоритмах практически не используется. Чаще всего индексы начинаются или с 1 или с 0. С точки зрения вычислительной математики лучше использовать 0. Если «размеров» несколько, для каждого из них указывается свой тип-диапазон через запятую. Имена однотипных массивов перечисляются через запятую.

Примеры:

a1,a3,bc:**array** [-2..56, 4..45] **of** integer;

tyk3:**array** [0..526, 0..145, 0..365] **of** real;

d1f:**array** [1..345] **of** extended.

Описание массивов с помощью образования нового типа данных. Сначала в специальном подразделе **type** описывается новый тип данных (стандартный производный), который устанавливает структуру массива. Затем в месте, где описываются различные переменные (подраздел **var**), описываются конкретные массивы с указанием этого нового типа данных. Формат:

type

<имя нового типа данных> = **array**[<тип-диапазон>] **of** <тип элементов массива>;

…

var

…

<имя массива>:<имя нового типа данных>;,

где имя нового типа данных – по правилам образования идентификаторов;

array – служебное слово «массив»;

тип-диапазон – задаёт пределы изменения индексов;

имя массива – имя конкретного массива.

Примеры:

type

dim2=array [0..50, 0..100];

…

var

mm1,mm2,mm3:dim2.

Одним из важнейших применений такого описания массивов является возможность передачи массивов в подпрограммы Паскаля – процедуры и функции (см. ниже).

При описании массивов в памяти выделяется место под их элементы, но не происходит заполнения элементов массива конкретными значениями (инициализации), т. е. ввод массива или другое присваивание значений его элементам должен производиться специально соответствующими средствами программы.

В общем случае строение массивов в Паскале может быть более сложным, вместо одного или нескольких типов-диапазонов могут быть порядковые типы, типом элементов массива может быть тип, описанный как тип массива (вложение массивов), и так далее, но это обычно редко используется в вычислительных задачах.

После описания массива с ним можно работать, т. е. производить какие-то действия, в том числе – инициализацию. Следует помнить, что эти действия можно проводить только с конкретными элементами массива, к которым можно обратиться («взять» для работы) по формату:

<имя массива>[<список индексов>],

где список индексов – перечисление конкретных индексов (через запятую) или один индекс.

Чаще всего работа с массивами происходит в основной части программы, где осуществляется основной алгоритм, поэтому имя массива само по себе, без индексов, здесь встречается крайне редко; например, в Паскале допустима операция присваивания между однотипными массивами:

```
aa1:=bb1;
```

где aa1 и bb1 – массивы, описанные одним типом.

В подавляющем большинстве случаев в основной части программы при работе с массивами нужно обязательно указывать индексы (элементы массива – **индексированные переменные**):

```
a[7]:=3.4587;
a[3]:=b[1,3];
a[3]:=c22;
b[1,3]:=d22;      (!)
g22:=a[3];        (!)
```

где a и b – массивы, c22, d22 и g22 – неиндексированные переменные того же типа, что и элементы массивов. (Разумеется, работа с массивами не сводится только к присваиванию).

7.4.2. Строки

Это цепочки символов, заключённые в апострофы, которые чаще всего служат для представления текстовой информации. Описание строк проводится по формату:

<имя строки>:**string**[<n>];

<имя строки>:**string**;,

где имя строки – идентификатор Паскаля,

string – служебное слово – «строка», имя типа,

n – число типа byte, т. е. от 0 до 255; если n не указано, то n = 255.

При описании строк регламентируется максимальное количество символов в ней, индекс 0 зарезервирован под значение текущей длины строки.

Примеры:

str1:**string**[20]; (в строке не более 20 символов),

str2:**string**; (в строке не более 255 символов).

Строка может быть инициализирована следующим образом:

str1:=’Stroka в Паскале’;

str2:=’Маша + Петя = любовь’.

Элементами строки могут быть любые символы ПК, в том числе – кириллические буквы (последние могут встречаться в программах на Паскале только внутри строк или как символы), при этом надо контролировать кодировку нестандартной части символов.

Инициализировать строки можно и другими способами. К элементу строки можно обратиться по индексу: str1[5] (в нашем примере это – символ k).

str2[1]:=’Д’; («Маша» будет заменено на «Даша»).

В Паскале существует множество процедур и функций для работы со строками и операций для действий над ними. Это достаточно мощное средство для работы с текстами. Но в вычислительных задачах строки чаще всего служат для ввода и вывода текстовой информации, поэтому их можно даже не описывать, а использовать, как строковые константы, т. е. просто заключать нужный текст в апострофы.

7.4.3. Записи и множества

Оставлены за пределами рассмотрения данного курса. В записях, в отличие от массивов, могут содержаться элементы разных типов.

7.5. Совместимость и преобразование типов

При выполнении некоторых операций операндами могут служить выражения разных типов, например числовых. Часто встречается ситуация, когда арифметическая операция выполняется над действительными и целыми числами.

5.246e1/7

14523–2.34 и т. д.

При этом возникает вопрос, какого типа будет результат операции и не будет ли происходить потери точности чисел. Подобные проблемы возникают и внутри целых или действительных типов; также это касается и многих других, нечисловых типов.

Если ограничиться числовыми типами, то совместимыми типами можно считать: а) целые между собой, б) вещественные между собой, в) если один тип является типом-диапазоном другого, г) если оба типа являются типами-диапазонами одного и того же базового типа.

Операция присваивания:

<имя переменной>:=< выражение>

для числовых и логического типов допустима, если:

1) переменная и выражение одного типа;

2) их типы – совместимые порядковые и значение выражения лежит внутри диапазона значений переменной;

3) их типы – вещественные и значение выражения лежит внутри диапазона значений переменной;

4) переменная – вещественного типа, выражение – целого.

В этих операциях присваивания происходит **неявное преобразование типов**, и результат всегда имеет более «широкий» тип. Такое же преобразование имеет место и в арифметических операциях с участием данных целого и вещественного типа, результатом всегда является вещественный тип.

Явное преобразование типов – это преднамеренное изменение типа переменной или выражения, причём программист должен отдавать себе отчёт в возможных последствиях такого преобразования, включая потерю точности. Довольно часто эти последствия отвечают требованиям алгоритма.

Наиболее универсальным средством явного преобразования типов является функциональная форма с общим форматом:

<тип>(<выражение>),

где тип – тип, к которому приводит преобразование, выражение – переменная или выражение, тип которого преобразуется.

Примеры:

integer(‘Ф’)

real(5.24856321567153281e5)

char(127 **mod** 5)

boolean(1)

Также для явного преобразования типов можно использовать специальные функции, которые можно найти в справочниках.

§ 8. Операторы языка Паскаль

Операторы – конструкции языка, реализованные по строгим правилам, представляющие собой более или менее законченные части алгоритма и обладающие определённой самостоятельностью. Языки программирования недаром называются языками, они имеют известную аналогию с лингвистическими языками. Например, общими (по названию) терминами являются алфавит, слова (служебные), синтаксис. В этом плане операторы можно сравнить с предложениями, из которых строятся отдельные абзацы и текст в целом. Синтаксис в языках программирования помимо всего прочего определяет правила построения операторов. Нарушение этих правил довольно легко может быть обнаружено средствами самого языка, выдаётся список **синтаксических ошибок**, и программист получает возможность их исправить.

Многие операторы соответствуют блокам графического представления алгоритмов.

Операторы бывают простые и сложные; в состав сложных операторов могут входить другие операторы, как простые, так и сложные. Мы рассмотрим **простые операторы**: **присваивания**, **пустой**, **оператор безусловного перехода**, **операторы ввода и вывода**, **вызов процедуры** и **комментарии**; а также **сложные операторы**: **составной**, **условного перехода**, **операторы циклов**, **выбора**.

Необходимо помнить, что **в конце каждого оператора обязательно ставится точка с запятой** «;».

8.1. Оператор присваивания

Соответствует операционному блоку. Общий формат:

<имя переменной>:=<выражение>;

Типы переменной и выражения должны совпадать.

Частные случаи:

<числовая переменная>:=<а. в.>;

<логическая переменная>:=<л. в.>;

Примеры:

x:=y;

x:=2*sin(x/2)+sqr(y);

z:=325;

pr1:=(s>-56) **and** (s<48).

В Паскале операция присваивания (и соответствующий оператор) определена над однотипными массивами. Если, например, массивы a1 и b1 описаны как:

a1,b1:**array**[5..78,0..100] **of** real;

и все элементы массива a1 имеют свои значения, то после выполнения оператора:

b1:=a1;

соответствующие элементы массива b1 получат те же самые значения. Следует отметить, что другие распространённые операции, как например операции сравнения и арифметические операции, над массивами не определены.

8.2. Составной оператор

Правила синтаксиса часто требуют, чтобы в том или ином месте стоял только один оператор, а алгоритм подразумевает несколько операторов. Выходом служит возможность объединения нескольких операторов в один с помощью **операторных скобок begin** (открывающая) и **end** (закрывающая). Полученный таким образом оператор называется **составным**. Общий формат:

begin <цепочка операторов> **end**;

Этот оператор широко используется в программах; раздел программы, в котором осуществляется её основной алгоритм, представ-

ляет собой один большой составной оператор. Допускается произвольная глубина вложений составных операторов. Перед **end** точку с запятой «;» можно не ставить.

Пример:

begin x1:=ln(abs(y*z));x2:=45.58;y:=d3+d1 **end**;

8.3. Пустой оператор

…;;…

Это два разделителя операторов, следующих друг за другом. Пустой оператор может быть помечен меткой для организации перехода исполнения алгоритма в эту точку (безусловный переход), т. е. – между операторами. Метка – это специальный объект языка, имеющий имя (идентификатор Паскаля) и описанный как метка – **label** (см. ниже). При использовании позади метки ставится двоеточие:

…;<метка>:;…

Пример:

…;s22:;… (s22 – имя метки)

8.4. Оператор безусловного перехода

goto <метка>; (**goto** пишется без пробела!)

Осуществляет переход исполнения алгоритма с данного места в любую точку программы, помеченную меткой, как после оператора **goto**, так и после.

Пример:

goto s33; …

…

…;s33:;…

При написании программ неопытными программистами очень часто возникает соблазн использования этого оператора. Для простых программ это допустимо. Но при усложнении алгоритма в геометрической прогрессии возрастает возможность ошибок. Поэтому программы нужно стремиться делать **хорошо структурированными**, т. е. использовать операторы, строго следующие один за другим, а все переходы и сложные разветвления алгоритма должны осуществляться работой самих операторов. Использование оператора **goto** не рекомендуется.

8.5. Оператор условного перехода (условный оператор)

Служит для организации сложных разветвлений алгоритма; соответствует логическому блоку. Имеет короткую и длинную формы. Общий формат:

if<л. в.>**then**<оператор 1>{**else**<оператор 2>};

if – служебное слово «если»,

then – служебное слово «то»,

else – служебное слово «иначе»,

л. в. – логическое выражение, простое или сложное,

оператор 1 и оператор 2 – любой оператор Паскаля, в том числе составной, когда на это место нужно поместить несколько операторов.

Перед **else** точка с запятой «;» **не ставится**.

Работа оператора. Длинная форма: если л. в. выполняется (**true**), то выполняется оператор 1, если л. в. не выполняется (**false**), то выполняется оператор 2. Короткая форма: если л. в. выполняется (**true**), то выполняется оператор 1, если л. в. не выполняется (**false**), то оператор 1 игнорируется и управление передаётся следующему по программе оператору.

Примеры:

if x4<8.5 **then** y2:=y2–4 **else** y2:=y2+88;

if zz2>zz1 **then** dsd:=sin(n1)+sqr(kl66);

if (x>–4) **and** (x<99) **then if** (y<45) **or** (x>99) **then** kkk66:=125.36;

В последнем случае на месте оператора 1 стоит ещё один условный оператор (оба оператора – короткой формы). В подобных случаях, если используются операторы длинной формы, может возникнуть неопределённость:

if<л. в. 1>**then if**<л. в. 2>**then**<оператор 1>**else**<оператор 2>;

Вопрос: к какому **if** относится **else** с оператором 2? На этот счёт существует правило: **else** относится к ближайшему слева или сверху **if** (запись оператора возможна как слева направо, так и сверху вниз, в зависимости от стиля), т. е. в данном примере **else** относится ко второму **if**. Чтобы **else** относился к первому **if**, нужно использовать операторные скобки:

if <л. в. 1> **then begin if** <л. в. 2> **then** <оператор 1> **end else** <оператор 2>;

8.6. Операторы циклов

В Паскале имеются три оператора цикла; условно называемые по ключевым словам: **for**, **repeat** и **while**. Циклы могут быть вложенными в любых комбинациях; глубина вложений не ограничивается. Эти операторы пригодны для реализации практически любых циклов с целыми и действительными параметрами, неявных и итерационных циклов. Частично соответствуют блоку цикла.

8.6.1. Оператор цикла for

Служит для организации явных циклов с целым параметром цикла и шагом 1 или –1. Чаще всего применяется для работы с массивами как механизм перебора индексов. Имеет две формы. Первая форма, **с шагом 1**:

for<параметр цикла>:=<н. з. п. ц.>**to**<к. з. п. ц.>**do**<оператор>;

for – служебное слово «для»,

параметр цикла – **переменная** порядкового типа,

н. з. п. ц. – начальное значение параметра цикла, **выражение** того же типа,

to – служебное слово «к»,

к. з. п. ц. – конечное значение параметра цикла, **выражение** того же типа,

do – служебное слово «выполнить»,

оператор – любой оператор Паскаля (в том числе – составной).

Вторая форма, **с шагом –1**:

for <параметр цикла> := <н. з. п. ц.> **downto** <к. з. п. ц.> **do** <оператор>;

downto – служебное слово «вниз к».

Следует подчеркнуть, что «н. з. п. ц.» и «к. з. п. ц.» – выражения, т. е. могут быть числами, переменными или а. в. (т. е. вычисляться).

Телом цикла является **do**<оператор>, всё остальное – заголовок цикла.

Работает этот оператор аналогично блоку цикла: вычисляется «н. з. п. ц.», выполняется цикл, параметр сравнивается с «к. з. п. ц.», если не превышает, выполнение тела цикла повторяется. Это оператор с **постпроверкой условия**, цикл выполнится хотя бы один раз.

Примеры:

for i:=0 **to** 100 **do for** j:=0 **to** 200 **do begin** a[i,j] := i*j; b[i,j]:=sqr(i)*j;**end**;

Осуществлён перебор элементов двумерных массивов во вложенном цикле и присваивание этим элементам значений, зависящих от индексов.

for i:=0 **to** n **do for** j:=0 **to** n **do** aa3[i,j]:=0;

for i:=0 **to** n **do for** j:=i **to** n **do** aa3[i,j]:=1;

Сначала всем элементам квадратной матрицы aa3 присвоены значения 0, а затем всем элементам главной диагонали и выше присвоены значения 1.

Недостатком оператора **for** Паскаля является то, что параметром цикла может быть только целое число (точнее – переменная порядкового типа), а шагом 1 и –1. В этом он уступает даже блоку цикла. Для сравнения можно сказать, что аналогичный оператор **for** языка C++ гораздо более универсален, в нём можно работать с действительными числами и любыми шагами, осуществлять неявные циклы с любым законом изменения параметра цикла, а также итерационные циклы; кроме того – циклы с неполными заголовками и бесконечные циклы.

8.6.2. Оператор цикла repeat

Это оператор с **постпроверкой условия**, служит для организации неявных и итерационных циклов. Использование его для обычных циклов, как правило, нерационально, хотя и допустимо. Недостаток его – то, что необходимо продумывать организацию самого цикла, т. е. структуру его заголовка. Общий формат:

repeat <операторы> **until** <л. в.>;

repeat – служебное слово «повторять»,

операторы – последовательность операторов (или один оператор),

until – служебное слово «пока»,

л. в. – логическое выражение, простое или сложное.

В последовательности операторов не нужно использовать составной оператор; роль **begin** и **end** выполняют **repeat** и **until**.

Работа оператора: выполнять последовательность операторов, пока не выполнится условие (л. в. – **true**).

8.6.3. Оператор цикла while

Это оператор с **предпроверкой условия**, цикл может ни разу не выполниться. В остальном – он полный функциональный аналог **repeat**. Общий формат:

while <л. в.> **do** <оператор>;

while – служебное слово «пока»,

л. в. – логическое выражение, простое или сложное,

do – служебное слово «выполнять»,

оператор – любой оператор Паскаля (в том числе – составной).

Работа оператора: пока выполняется условие (л. в. – **true**), выполнять оператор.

Примеры (для **repeat** и **while**).

Реализуем с помощью этих операторов первый пример цикла **for** (см. стр. 71):

```
i:=0; while i<=100 do begin j:=0;
repeat a[i,j]:=i*j;b[i,j]:=sqr(i)*j; j:=j+1;until j>200;i:=i+1;end;
```

или

```
i:=0; repeat j:=0; while j<=200 do
begin a[i,j]:=i*j;b[i,j]:=sqr(i)*j; j:=j+1;end; i:=i+1;until i>100;
```

В первом и втором варианте к заголовку внешнего цикла относятся элементы: i:=0;, i<=100 и i:=i+1;, а к заголовку внутреннего цикла – j:=0;, j<=200 и j:=j+1;.

Но, как уже отмечалось, в данном случае использовать эти циклы нецелесообразно. Для неявных и итерационных циклов:

```
a1:=6.35; while a1 <= 100.85 do begin dd4 := sin(sqr(a1))*45; a1:=a1+.05; end;
a1:=6.35; repeat dd4:=sin(sqr(a1))*45;a1:=a1*2; until a1<=635;
s:=0;n:=0; repeat n:=n+1;a:=1/n;s:=s+a: until a<.000001;
```

Во втором примере шаг непостоянный (неявный цикл). Третий пример – вычисление суммы бесконечного сходящегося ряда s с заданной точностью .000001.

Первый пример можно реализовать с помощью **for**, но это, в свою очередь, тоже нерационально:

```
a1:=6.35; for i:=1 to int((100.85 – 6.35)/.05+1) do dd4 := sin(sqr(a1))*45;
```

Здесь цикл строится по счётчику цикла; выражение для количества циклов получено из выражения для конечного значения параметра цикла (см. описание блока цикла):

xk=xn+h(n-1),

где xk – к. з. п. ц., в нашем случае равно 100.85,

xn – н. з. п. ц., в нашем случае равно 6.35,

h – шаг, в нашем случае равен 0.05,

n – искомое число циклов.

Таким образом n=(xk-xn)/h+1.

8.7. Оператор выбора

Служит для организации сложных разветвлений алгоритма, когда из одной точки возможно движение по нескольким альтернативным путям. В этих случаях он рациональнее условного оператора. Общий формат:

case<ключ выбора>**of**<список выбора>{**else**<оператор>}**end**;

case – служебное слово «случай»,

of – служебное слово «из»,

else – служебное слово «иначе»,

end – служебное слово «конец»,

ключ выбора – выражение любого порядкового типа; чаще всего это выражение целого типа, которое подсчитывается и значение которого определяет дальнейший путь исполнения алгоритма,

список выбора – одна или несколько конструкций вида:

<константа выбора>:<оператор>;

константа выбора – константа того же типа, что и ключ выбора; чаще всего это целое число,

оператор – любой оператор Паскаля, в том числе составной.

Работа оператора. Вычисляется значение ключа выбора (чаще всего получается небольшое целое число), в списке выбора разыскивается равная ему константа выбора и выполняется соответствующий оператор, после чего оператор **case** завершается. Если же константа выбора не найдена, то выполняется оператор после **else**, если же конструкция с **else** отсутствует, то управление передаётся следующему по алгоритму оператору.

Пример:

```
case (3*m+n)*2 of
8: fg2:=4526.32;
10: fg2:=152.328;
14: fg2:=-45.218;
16: fg2:=0.145;
else fg2:=100;end;
```

8.8. Оператор ввода (стандартная процедура ввода)

Служит для ввода информации в компьютер, соответствует блоку ввода. Существует в двух вариантах, без перевода строки и с переводом строки после ввода. Строго говоря, это не операторы, но их так называют для простоты. Форматы:

read(<список ввода>); – без перевода строки, (читать);

readln(<список ввода>); – с переводом строки (line – строка);

read и readln – имена стандартных процедур ввода, служебными словами не являются (не следует называть какие-то переменные этими именами, иначе стандартный ввод и вывод перестанут работать);

список ввода – перечисление через запятую «,» (разделитель списка) имён переменных, обозначающих ячейки памяти, куда осуществляется ввод. Допускается ввод переменных типа integer, real, char, string. В подавляющем большинстве случаев при правильно организованном вводе в списке находится только одна переменная.

Примеры:

```
read(x,y,z);
readln(x3);
```

Ввод чаще всего осуществляется с клавиатуры или из файла. При вводе с клавиатуры на мониторе появляется мигающий курсор, на месте которого нужно набрать число и нажать клавишу «Enter», пока она не нажата, число можно редактировать.

8.9. Оператор вывода (стандартная процедура вывода)

Служит для вывода информации на дисплей, в файл и так далее, соответствует блоку вывода. Имеет такой же статус, как оператор ввода и также существует в двух вариантах. Форматы:

write(<список вывода>); – без перевода строки (писать),

writeln(<список вывода>); – с переводом строки (line – строка).

write и writeln – имена стандартных процедур вывода (служебными словами не являются,),

список вывода – перечисление через запятую «,» (разделитель списка) **выражений** различных типов – integer, real, char, string, boolean. Выводится всегда **значение** выражений, в частности, а. в. подсчитываются и выводятся числа. Для вывода текстовой информации в список вывода помещается строковая константа – т. е. любой текст в апострофах «’».

Пример:

writeln(‘Число пи равно=’,pi);

На экран монитора будет выведено: «Число пи равно=3.14159».

При вводе информации в компьютер оператор вывода целесообразно использовать для вывода подсказок, так как работа оператора ввода начинается с «приглашения» мигающего курсора и пользователю не всегда ясно, какую переменную нужно вводить:

write(‘Введите x=’); readln(x);

Сначала на экране появятся надпись «Введите x=» и мигающий курсор после неё (в той же самой строке, поэтому используется write, а не writeln), затем нужно набрать числовое значение x и нажать клавишу «Enter».

Особенно актуальна такая схема для **ввода и вывода массивов**, так как это делается **в цикле** и нужно вводить и выводить текущие значения индексов.

Ввод массива $\{a_i\}_n$ поэлементно:

for i:=0 **to** n **do begin** write(‘a[‘,i,’]=’);readln(a[i]);**end**;

Вывод массива $\{a_i\}_n$ поэлементно:

for i:=0 **to** n **do** writeln(‘a[‘,i,’]=’,a[i]);

Здесь в списке вывода 4 элемента – 2 текстовые информации и 2 переменные.

Для двумерных массивов задача выглядит сложнее, но принцип остаётся тот же.

Ввод массива $\{a_{i,j}\}_{n,m}$ поэлементно:

for i:=0 **to** n **do for** j:=0 **to** m **do begin** write(‘a[‘,i,’,’,j,’]=’); readln(a[i,j]); **end**;

Здесь запятая между i и j выводится как текстовая информация «’,’».

Вывод массива $\{a_{i,j}\}_{n,m}$ поэлементно:

for i:=0 **to** n **do for** j:=0 **to** m **do** writeln(‘a[‘,i,’,’,j,’]=’,a[i,j]);

Здесь в списке вывода 6 элементов – 3 текстовые информации и 3 переменные.

Результат вывода одного элемента массива может быть, например, такой:

a[5,8]=6.153e-5

Особенности ввода и вывода в файл будут рассмотрены ниже.

8.10. Оператор вызова процедуры

Будет рассмотрен в гл. 3 § 2.

8.11. Комментарии

Любая часть текста программы, заключённая в скобки «{«, «}» или «(*» «*)», является комментарием и игнорируется средой программирования (т. е. конкретной реализацией языка). Комментарии служат для пояснения тех или иных фрагментов программы или могут использоваться при отладке программы для временного исключения некоторых частей алгоритма. Допускается вложение комментариев.

Кроме этого, в таких средах программирования, как Delphi и Lasarus, закомментировать можно отдельные строки текста путём помещения в начало строки двойного слэша «//».

Примеры:

{это самая хитрая программа на свете}

// x:=sin(2*y) – 5;

§ 9. Строение программы на Паскале

Программа на Паскале состоит из одной или нескольких **программных единиц**, имеющих схожее строение, а именно – из четырёх разделов. Имеется три вида программных единиц: программа (основная) **program**, функция **function** и процедура **procedure**. Строение первого и четвёртого разделов этих программных единиц – индивидуально, а строение второго и третьего – аналогично. Вспомогательные программные единицы – **function** и **procedure** – представляют собой **описания** функций и процедур и помещаются внутри основной

программы как отдельные блоки. Использоваться же они могут в самих алгоритмах основной программы а также в алгоритмах процедур и функций.

Строение программной единицы

Первый раздел – обозначение программной единицы

Для основной программы:

program <имя программы>;

program – служебное слово «программа», имя программы – идентификатор Паскаля. Может опускаться.

Строение функций и процедур будет рассмотрено в гл. 3 § 1, 2.

Второй раздел – раздел описаний

В современных профессиональных языках программирования все объекты программы, такие как переменные, постоянные, новые типы, метки и так далее, должны быть полностью определены перед использованием, т. е. описаны. Если в программе встречаются неописанные объекты, это сразу вызывает ошибку на этапе отладки программы. В одном из первых языков программирования, Фортране, объекты не описывались, что очень часто приводило к трудновыявляемым ошибкам. Например, если в программе использовалась переменная r28i4, а при наборе ошибочно было набрано r28l4, то такую подмену очень трудно было обнаружить визуально, а среда программирования эту ошибку вообще не могла заметить, и в программе действовала совершенно посторонняя переменная.

В Паскале все объекты программы описываются в одном месте – разделе описаний. Раздел описаний состоит из **подразделов**: **const**, **var**, **type**, **label**; также здесь помещаются **функции** и **процедуры** (как самостоятельные программные единицы) и описания объектов (классов C++), если используется объектно-ориентированное программирование. Подразделы могут дублироваться и следовать в любом порядке; описание процедур и функций помещается в конце раздела описаний.

Подраздел const

Служит для описания именованных констант различных типов. Общий формат:

<имя константы>=<значение>;

Тип константы определяется самим значением; например, тип действительного числа определяется по наличию десятичной точки, а

конкретный вещественный тип – как минимальный по ёмкости без потери точности:

const – имя подраздела,
a1=45; – целый тип byte,
a2=.56; – single,
a3=2.256321578; – real,
a4=2.3e-200; – double.
Другие примеры:
f4=**false**; – boolean,
ok=’хорошо’; – **string**.

Следует помнить, что в константах полученные таким образом значения не меняются до конца работы программы.

Подраздел var

Служит для описания переменных различных типов, массивов, строк. При описании обязательно указывается **тип**. При этом под переменную выделяется соответствующая область памяти (ячейка памяти), к которой можно обратиться по имени переменной (или адресу). **Значение переменная не получает**, т. е. требуется присваивание переменной какого-то значения, что можно сделать только в другом разделе программы. (Строго говоря, при описании происходит первоначальное присваивание в виде нуля или «мусора», т. е. совершенно случайного числа; это никак не связано с алгоритмом программы). Общий формат:

<имя переменной>:<тип>;
имя переменной – идентификатор Паскаля;
тип – любой тип Паскаля.
Однотипные переменные можно перечислять через запятую «,».
Примеры:

var – имя подраздела,
n,m,k:**integer**;
nmk:**extended**;
i1,j1,i2,j2:**real**;
st:**string**; – строка из 255 символов,
st:**string**[10]; – строка из 10 символов,
mass3:**array** [1..52, 0.. 65] **of real**; – двумерный массив,
a1,a3:**array** [0..n, 0..m] **of integer**; – два двумерных массива.

В последнем случае, к моменту описания массивов a1 и a3 константы n и m должны быть определены в подразделе **const**, так как при объявлении массивов (как и других переменных) происходит выделение соответствующей памяти. При этом возникает вопрос реализации в программе работы с переменной размерностью массивов.

Имеется в виду то, что одна и та же программа должна обрабатывать по одному алгоритму массивы с разным количеством элементов в каждом размере. Кардинальное решение этой задачи – работа с динамической памятью, когда массив объявляется не как статический, а как динамический, и память под него выделяется уже после начала работы программы и ввода значений, например n и m (при этом они описываются как переменные, хотя по алгоритму являются постоянными). Но рассмотрение динамической памяти выходит за пределы данного курса, поэтому мы рассмотрим другие, не лучшие, хотя и простые, методы.

Самый простой способ – описать размерность массива «с запасом». Например, если известно, что размерность каждого размера массива не будет превышать нескольких сотен элементов, можно задать максимальную размерность 1000:

```
a1,a3:array [0..1000, 0..1000] of integer;
```

Если в конкретном случае применения программы размерности будут, например 200 и 300, то в массив вводятся соответствующие значения, а остальные элементы будут нулевыми и программой обрабатываться не будут. Конечно, при этом происходит нерациональное расходование памяти.

Другой способ состоит в описании, например n и m, как констант (им присваиваются любые значения), и в дальнейшем описании массивов через эти константы:

```
const
n=200;m=300;
…
var
a1,a3:array [0..n, 0..m] of integer;
```

Для работы с другими значениями n и m в тексте программы делаются соответствующие изменения (но только в одном месте программы, что исключает ошибки), и программа перекомпилируется (т. е. заново подготавливается в среде программирования к использованию). Это не лучший вариант, но сравнительно безопасный.

Подраздел type

Служит для описания новых типов данных, стандартных производных. В вычислительных задачах используется практически только для описания массивов как новых типов данных. Как уже отмечалось, сначала описывается новый тип массив, а затем, в подразделе **var**, – конкретные массивы. Предыдущий пример может быть преобразован так:

const – имя подраздела,
n=200;m=300; – n и m – константы,
…
type – имя подраздела,
ms=**array** [0..n, 0..m] **of integer**; ms – имя нового типа данных,
…
var – имя подраздела,
a1,a3:ms; – a1 и a3 – конкретные массивы типа ms.

Такое описание массивов делает возможным их передачу в процедуры и функции, так как при этом нужно указывать тип, а указание целой конструкции (со словом **array**) неприемлемо.

Подраздел label

При описании меток просто перечисляются их имена:

label – имя подраздела,
s1,s2,f1; – конкретные метки, имена – идентификаторы Паскаля.

Метками могут быть помечены пустые операторы для осуществления безусловных переходов (гл. 2, § 8, п. 8.4). Их использование не рекомендуется.

При составлении программы не нужно стремиться заранее описывать исключительно все объекты, описания можно добавлять по ходу составления алгоритма.

Третий раздел – раздел исполняемых операторов

В нём осуществляется основной алгоритм программы. Начинается со служебного слова **begin** и заканчивается служебным словом **end**, т. е. представляет собой один большой составной оператор:

begin … **end**

Формально в этом разделе после **end** ничего не стоит. Алгоритм представляет собой цепочку операторов, следующих друг за другом и разделённых точкой с запятой «;». Многие операторы – сложные и включают в себя другие операторы, простые или сложные. В хорошо структурированных программах оператор безусловного перехода, нарушающий этот строгий порядок, не используется, что резко сни-

жает риск ошибок в больших программах. При этом все сложные разветвления и переходы алгоритма можно осуществить безопасно средствами самих операторов.

Как правило, в начале алгоритма должен быть осуществлён ввод исходных данных. В ходе алгоритма возможно и часто происходит, обращение к функциям – стандартным, библиотечным и пользовательским, и вызов процедур. В конце алгоритма, как правило, происходит вывод результатов работы программы.

Парадигма **процедурно-ориентированного программирования**, позволяющая успешно осуществлять подавляющее большинство вычислительных алгоритмов, заключается в распределении частей алгоритма между процедурами и функциями и «сшивание» их основной программой. Очень часто основной объём программы составляют процедуры и функции, а в короткой основной программе осуществляется ввод данных, вызов процедур и вывод результатов.

Четвёртый раздел – символ конца программной единицы

Для основной программы – точка «.», для процедур и функций – точка с запятой «;».

Также в программе может встречаться **подключение стандартных модулей** (модули также могут быть созданы пользователем). В большинстве случаев оно осуществляется до раздела описаний. Формат:

uses <имя модуля>;

uses – служебное слово «использовать».

Модуль SYSTEM подключается к любой программе и не указывается (содержит основные стандартные средства языка). Примерами подключения могут служить модули CRT (управление текстовым режимом работы экрана), GRAPH (графический модуль), математические модули и т. д.

Следует также упомянуть о стилях программирования. Программа на Паскале представляет собой одну большую строку (с «точки зрения» компилятора языка). Разбиение на строки и табуляция производятся пользователем произвольно, и средой программирования (компилятором языка) игнорируется. Пожалуй, самый распространённый стиль – написание программы «сверху вниз», с выделением отдельных блоков, операторов и составных операторов в отдельных строках. При этом в случае больших программ на мониторе видна лишь небольшая часть алгоритма, что часто затрудняет написание и понимание программы. Альтернативным стилем является разбивка программы на длинные строки по ширине окна редактирова-

ния, что позволяет охватить значительно больший фрагмент текста, но несколько затрудняет его восприятие.

Примеры:

Программа перемножения двух чисел и вывод результата:

```
program abc;
var
a,b,c:real;
begin
write('a=');readln(a); write('b=');readln(b);
c:=a*b;
writeln('c=',c);
end.
```

Программа ввода двумерного массива, умножение каждого элемента на известное заданное число f и вывод полученного массива. Все три операции нужно проводить во вложенных циклах, которые будут следовать один за другим, причём заголовки этих циклов будут совершенно одинаковыми, поэтому их можно объединить в один цикл.

```
program mmm;
const
n=20;m=30;
var
a:array [0..n, 0..m] of real; f:real; i,j:integer;
begin
write('f=');readln(f);                          // ввод числа f
for i:=0 to n do for j:=0 to m do begin
write('a[',i,',',j,']=');readln(a[i,j]);       // ввод массива поэлементно
a[i,j]:=a[i,j]*f;                       // умножение элемента на число f
writeln('a[',i,',',j,']=',a[i,j]);     // вывод преобразованного массива
end;
end.
```

Так как преобразованный массив сам по себе нигде не используется, операторы:

```
a[i,j]:=a[i,j]*f;
writeln('a[',i,',',j,']=',a[i,j]);
```

можно заменить на один:

```
writeln('a[',i,',',j,']=',a[i,j]*f);
```

Глава 3. ОСНОВНЫЕ СРЕДСТВА ПРОЦЕДУРНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

§ 1. Функции

В Паскале имеется широкий набор библиотечных и стандартных функций, в частности математических. При их использовании (вызове) записывается имя функции и в круглых скобках указывается аргумент.

Например:

sin(x*4) sin – имя функции,

sqr(s2) sqr – имя функции.

После вычисления полученное значение подставляется на место вызова функции. Это означает, что алгоритм вычисления функции уже написан и содержится в библиотеках языка программирования, а основная программа лишь использует этот алгоритм. В Паскале имеется возможность написания и использования функций самим программистом (пользовательские функции). Необходимость этого возникает очень часто, например, если в тексте программы несколько раз используется длинное арифметическое выражение (а. в.) одного вида, но с разными (однотипными) переменными:

(sin(y)+sqr(cos(x)))/(a*b-c)

(sin(x)+sqr(cos(y)))/(c*d-a)

(sin((8*g-5)+r)+sqr(cos(k-u)))/(ff*sin(p)-b)

Главным отличием пользовательских функций от библиотечных является возможность создания функции не с одним аргументом, а с несколькими. В приведённых примерах вид а. в. совершенно одинаков, а аргументы – разные:

y, x, a, b и c

x, y, c, d и a

(8*g-5)+r, k-u, ff, sin(p) и b

Можно написать функцию, вычисляющую а. в. данного вида и использовать её, передавая в функцию соответствующие аргументы. **В отличие от циклов, в которых один и тот же алгоритм осуществляется для расчётов с разными значениями одних и тех же переменных, в функции расчёты производятся с разными переменными** (они, конечно могут быть и одинаковыми).

Следует помнить, что любая функция **возвращает только одно значение**.

Пользовательские функции должны быть описаны и только после этого могут использоваться.

1.1. Описание функций

Представляет собой отдельную программную единицу и помещается в конце раздела описаний основной программы. Определяет основной алгоритм функции и характеризует переменные, которые нужно в функцию передать (они называются **параметрами**). Общий формат:

Первый раздел

function <имя функции>{(<список входных параметров>)}:<тип в. з.>;

function – служебное слово «функция»,

имя функции – идентификатор Паскаля,

список входных параметров – одна или несколько переменных, передаваемых в функцию; разделитель списка – точка с запятой «;». Однотипные параметры могут перечисляться через запятую «,». Список может отсутствовать, но, как правило, присутствует. Общий формат элемента списка:

<имя переменной>:<тип>;

тип в. з. – тип возвращаемого значения (носителем этого значения является имя функции).

Второй раздел

В разделе описаний, как правило, описываются только те объекты, которые используются внутри самой функции и не используются в основной программе. Раздел может отсутствовать, что бывает очень часто. Здесь может находиться описание других процедур и функций (вложение).

Третий раздел

Здесь осуществляется основной алгоритм функции. В третьем разделе **обязательно должен быть оператор, в котором имени функции присваивается какое-то значение** (формирование выходного параметра).

Четвёртый раздел

Символ конца программной единицы – точка с запятой «;».

Параметры подразделяются на **входные и выходные**, **формальные и фактические** (ниже будут рассмотрена еще одна градация – **параметры-значения и параметры-переменные**). В **описании** функций (и процедур) все параметры – **формальные**, потому что это описание. При **вызове** функций (и процедур) параметры **фактические**. **Входные** параметры **функции** находятся **в списке** параметров, **выходной** параметр связан с **именем функции**.

Пример. Полное описание для функции предыдущего примера может выглядеть так:

function fff(c:integer;e,x,a.b:real):real;

begin fff:=(sin(y)+sqr(cos(x)))/(a*b-c);**end**;

1.2. Вызов функции (обращение к функции)

Вызов осуществляется в третьих разделах основной программы и других программных единиц и **представляет собой арифметическое выражение а. в.** (или просто выражение). Функции могут стоять везде, где может стоять а. в. (или просто выражение), в том числе – входить в другие выражения. Общий формат:

<имя функции>(<список фактических входных параметров>)

имя функции – соответствует имени в описании,

список фактических входных параметров – перечисление **выражений** через запятую «,».

В общем случае точка с запятой после обращения к функции не ставится. В списке параметров находятся именно выражения, например, в случае а. в. это могут быть имена постоянных или переменных, числа и сложные выражения. **При вызове функций (и процедур) формальные и фактические параметры должны соответствовать друг другу по общему количеству, порядку следования в списке и типу.**

После вычисления функции полученное значение подставляется на место вызова функции.

Примеры вызова функции (по предыдущему примеру):

fff(y, x, a, b, c)

fff(x, y, c, d, a)

fff((8*g-5)+r, k-u, ff, sin(p), b)

Разумеется, в функциях могут осуществляться гораздо более сложные алгоритмы, с циклами, вызовами функций и процедур и т. д.

§ 2. Процедуры

В отличие от функций **процедуры могут возвращать несколько значений** (т. е. могут иметь несколько выходных параметров). Они также представляют собой самостоятельные программные единицы и перед использованием должны быть описаны в разделе описаний основной программы.

2.1. Описание процедур

Общий формат:

Первый раздел

procedure <имя процедуры>{(<список параметров>)};

procedure – служебное слово «процедура»,

имя процедуры – идентификатор Паскаля,

список параметров – входные и выходные параметры, передаваемые в процедуру (в любом порядке); разделитель списка – точка с запятой «;». Однотипные параметры могут перечисляться через запятую «,». Список может отсутствовать, но, как правило, присутствует. Общий формат элемента списка:

{**var**} <имя переменной>:<тип>;

где **var** – модификатор (см. гл. 3, §2, п. 2.2).

Следует отметить, что имя процедуры, в отличие от имени функции никакой смысловой нагрузки не несёт, т. е. не связано с какими-то параметрами.

Второй раздел

В разделе описаний, как правило, описываются только те объекты, которые используются внутри процедуры и не используются в основной программе. Раздел может отсутствовать, что бывает довольно часто. Здесь может находиться описание других процедур и функций (вложение).

Третий раздел

Здесь осуществляется основной алгоритм процедуры. Выходные параметры должны получать значения в соответствии с алгоритмом.

Четвёртый раздел

Символом конца программной единицы является точка с запятой «;».

2.2. Вызов процедуры (обращение к процедуре)

Вызов осуществляется в третьих разделах основной программы и других программных единиц и **представляет собой оператор** Паскаля. Он может стоять или в общей цепочке операторов, или входить в другие, сложные, операторы. Общий формат:

<имя процедуры>(<список фактических параметров>);

имя процедуры – соответствует имени в описании,

список фактических параметров – перечисление **выражений** через запятую «,» (если формальные параметры описаны с модификатором **var**, то фактические могут быть только **именами переменных**; (см. гл. 3, § 2, п. 2.2).

В конце, как и после каждого оператора, ставится точка с запятой «;». В списке параметров находятся именно выражения, например в случае а. в. это могут быть имена постоянных или переменных, числа и сложные выражения. **При вызове процедуры формальные и фактические параметры должны соответствовать друг другу: по общему количеству, порядку следования в списке и типу.** Следует отметить, что, если формальные и соответствующие фактические параметры – имена переменных, то эти имена **могут** быть разными или **совпадать**, при этом никой путаницы не возникает, так как они принадлежат разным программным единицам и ячейки памяти всё равно будут разными.

После выполнения процедуры выходные параметры получают значения в соответствии с алгоритмом и программа исполняется дальше.

Для того чтобы прояснить механизм работы с выходными параметрами, рассмотрим пример, в котором все параметры описаны обычным способом. При таком описании при вызове процедуры в неё передаются **значения** соответствующих фактических параметров, поэтому эти формальные параметры называются **параметрами-значениями**. Схема работы процедуры при её вызове:

procedure GFa(n:**integer**;a,b,c:**real**); …

Допустим, по смыслу алгоритма, параметры n и a – входные, b и c – выходные. Перед вызовом процедуры, например:

GFa(m,d,f,g);

все четыре фактических параметра должны иметь свои значения: m и d – по алгоритму, а f и g – либо значения, полученные при

инициализации, либо – после каких-то действий, но, так или иначе, это значения, которые будут изменены процедурой. Допустим, $m = 5, d = 8, f = 0, g = 0$. При вызове процедуры всегда образуются временные переменные (в нашем случае – n, a, b, c), т. е. в оперативной памяти выделяются соответствующие ячейки памяти (обозначим их прямоугольниками с именем и значением), которые будут уничтожаться после окончания работы процедуры. В эти ячейки и передаются значения всех фактических параметров (рис. 52).

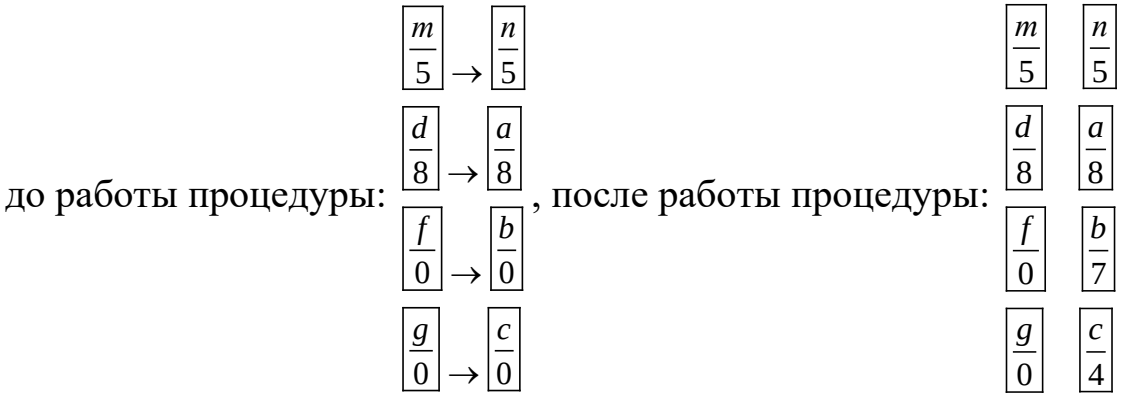

Рис. 52. Работа процедуры (неправильная организация). Ячейки памяти переменных основной программы и процедуры

После работы процедуры выходные параметры получают новые значения согласно алгоритму программы. Например, $b = 7, c = 4$; входные параметры, как правило, свои значения не меняют. После этого работа процедуры завершается и временные переменные n, a, b, c уничтожаются, при этом вычисленные значения 7 и 4 пропадают, так как они не были переданы в основную программу (см. рис. 52).

Чтобы этого избежать, выходные параметры в процедуре нужно описать не как **параметры-значения**, а как **параметры-переменные**, с использованием модификатора **var**. При этом будет меняться механизм взаимодействия процедуры и основной программы:

procedure GFa(n:**integer**;a:**real**; **var** b,c:**real**); …

Как отмечалось ранее, к переменной, содержащейся в ячейке памяти, можно обратиться по имени или по адресу. Если параметр описан как параметр-переменная, при вызове процедуры формальный параметр получает не значение фактической переменной, а её **адрес**. Таким образом, процедура, работая с формальными выходными па-

раметрами (в нашем примере – b и c), будет менять не их значение, а значение соответствующих фактических выходных параметров основной программы по их адресу (в нашем примере – f и g).

Таким образом, после уничтожения временных переменных n, a, b и c результат работы процедуры остаётся в переменных f и g (фактические выходные параметры) – рис. 53.

Рис. 53. Работа процедуры (правильная организация).
Ячейки памяти переменных основной программы и процедуры

Из сказанного следует, что **фактические выходные параметры, соответствующие формальным параметрам-переменным, не могут быть выражениями, а только именами переменных**.

В Паскале нет жёсткого требования, чтобы выходные параметры описывались как параметры-переменные, но чаще всего это так и бывает.

§ 3. Передача массивов в процедуры и функции

Поскольку описание массивов является сложным (это производный тип на основе стандартных базовых), невозможно передать в процедуру или функцию массив, который описан напрямую. Чтобы это сделать, нужно описать массив через создание нового типа данных (в подразделе **type**) и в описании формальных параметров процедуры (или функции) указывать этот новый тип данных примерно по такой схеме:

```
program WWW;          // основная программа
const n=100; m=150;   // описание констант n и m
…
type mass= array [0..n, 0..m] of real; // описание нового типа данных
                                       // mass (массив)
var
…
f,g,h: mass;          // описание конкретных массивов f,g и h
…
procedure VVV(ff,gg: mass; var hh: mass);
// заголовок процедуры VVV с описанием формальных массивов
// ff,gg и hh через тип mass (ff и gg – входные параметры, hh –
// выходной); это даёт возможность передачи массивов в процедуру
…                     // тело процедуры
begin                 // начало третьего раздела основной программы
…
VVV(f,g,h);           // вызов процедуры с передачей в неё массивов
                      // f,g и h
…
end.                  // конец основной программы
```

Пример. Составим программу, перемножающую квадратные матрицы с использованием процедуры для этой операции и выводом результирующей матрицы.

```
program MULTIPL;                        // основная программа
const n=100;       // начало раздела описаний основной программы
type matr= array [0..n, 0..n] of real;  // описание нового типа
                                        // данных
var
a1, b1, c1, a2, b2, c2, c3: matr; i, j: integer;   // описание матриц и
                                                   // целых i, j
procedure mpm (aa, bb: matr; var cc: matr);        // заголовок
                                                   // процедуры
var i, j: integer;                      // раздел описаний процедуры
```

```
begin
for i:=0 to n do for j:=0 to n do cc[i, j]:=aa[i, j]*bb[j, i];
                                                    // тело процедуры
end;
begin                  // начало третьего раздела основной программы
for i:=0 to n do for j:=0 to n do begin
write('a1[',i,',',j,']=');readln(a1[i,j]);          // ввод массива a1
                                                    // поэлементно
end;
for i:=0 to n do for j:=0 to n do begin
write('b1[',i,',',j,']=');readln(b1[i,j]);          // ввод массива b1
                                                    // поэлементно
end;
for i:=0 to n do for j:=0 to n do begin
write('a2[',i,',',j,']=');readln(a2[i,j]);          // ввод массива a2
                                                    // поэлементно
end;
for i:=0 to n do for j:=0 to n do begin
write('b2[',i,',',j,']=');readln(b2[i,j]);          // ввод массива b2
                                                    // поэлементно
end;
mpm(a1, b1, c1);   // перемножение матриц a1 и b1, результат – c1
mpm(a2, b2, c2);   // перемножение матриц a2 и b2, результат – c2
mpm(c1, c2, c3);   // перемножение матриц c1 и c2, результат – c3
for i:=0 to n do for j:=0 to n do
writeln('c3[',i,',',j,']=',c3[i,j]);       // вывод результирующей
                                           // матрицы c3 в цикле
end.                        // конец основной программы
```

Анализ этой программы позволяет рассмотреть ещё один интересный вопрос, касающийся **пространства имён** в Паскале. И в основной программе, и в процедуре описаны целые переменные с одним и тем же именем – i и j. Возникает вопрос об их взаимозависимости, так как возможно возникновение ошибок, если и в основной программе и в процедуре будут действовать одни и те же переменные i и j. Например, если процедура DDD вызывается в основной программе

внутри цикла по i и j, а в самой процедуре действует свой цикл по тем же переменным i и j:

```
program qqq;              // основная программа
…
var i, j:integer;         // описание целых i, j
…
procedure DDD(…);  // заголовок процедуры
…
begin                     // начало тела процедуры
…
for i:=0 to n do for j:=0 to n do begin
…
end;
…
end;                      // конец описания процедуры
begin               // начало третьего раздела основной программы
…
for i:=0 to n do for j:=0 to n do begin
…
DDD(…);                   // вызов процедуры в цикле
…
end;
…
end.                      // конец основной программы
```

Относительно **пространства действия имён** существует следующее правило.

Подпрограммы (процедуры и функции) «видят» все объекты, описанные во внешней программной единице (т. е. в основной программе или в процедуре или функции, если они в них, в свою очередь, входят). Внешние программные единицы не «видят» объекты, описанные во внутренних программных единицах. «Видимость» означает возможность их использования. Если во внешней и внутренней программной единице описаны однотипные объекты с одинаковыми именами, то во внутренней программной единице «внешние» объекты будут маски́роваться «внутренними», т. е. станут «невидимыми».

Таким образом, в рассмотренном примере будут возникать ошибки, если в процедуре не описаны i и j:

```
procedure DDD(…);   // заголовок процедуры
…
begin               // начало тела процедуры
```

После вхождения в цикл основной программы (i=0 и j=0) и выполнения процедуры (переменные в конце получат значения i=n+1 и j=n+1) произойдёт возврат в заголовок цикла по j основной программы, где i и j будут уже иметь совсем не те значения, которые обусловлены алгоритмом. Если же такое описание присутствует:

```
procedure DDD(…);   // заголовок процедуры
…
var i, j:integer;   // описание целых i, j внутри процедуры;
                    // внешние i, j маскируются
…
begin               // начало тела процедуры
…
```

то никаких ошибок не возникнет.

Из сказанного следует рекомендация, что в процедуры (и функции) не нужно передавать через параметры объекты основной программы, которые будут одними и теми же при любых обращениях к процедуре. Именно этим объясняется возможность существования процедур и функций без параметров.

При работе с процедурами и функциями в Паскале допускается **рекурсия** – обращение подпрограммы к самой себе и опережающее описание (**forward**). Эти средства иногда позволяют оптимизировать алгоритмы; их рассмотрение выходит за рамки данного курса.

§ 4. Некоторые средства работы с файлами

При необходимости ввода в программу больших массивов данных крайне неудобно использовать клавиатуру; однако если программа выдаёт в результате большие объёмы информации, снимать их с экрана монитора часто просто нереально. Выход из этого – использование файлов как хранителей информации; можно считывать данные с файла или выводить информацию в файл.

В Паскале имеется множество средств работы с файлами, в том числе форматированный ввод-вывод. В большинстве эти средства осуществлены в соответствующих библиотечных функциях.

Мы рассмотрим лишь минимальное число этих средств, позволяющих считывать информацию из файла и записывать данные в файл.

Сначала в программной единице должны быть описаны так называемые **файловые переменные**, через которые происходит вся работа с файлами. Затем должна быть установлена связь программы и внешнего файла (**открытие файла**). Затем происходит собственно **работа с файлом**, после чего файл должен быть **закрыт**.

Файловые переменные можно подразделить по типу данных, хранимых в файле. Среди них можно выделить два основных типа – **текстовые** файлы и **бинарные**. Данные хранятся и используются в компьютере в бинарном виде; при вводе-выводе в текстовый файл числа преобразуются в строки с некоторой потерей точности (в случае действительных чисел). Пользователь может воспринимать текстовые файлы и не может – бинарные.

Файловые переменные описываются в разделе описаний соответствующей программной единицы (подраздел **var**). Формат описания текстовых переменных:

<имя файловой переменной>: **text**;

имя файловой переменной – идентификатор Паскаля,

text – тип текстовой файловой переменной.

Формат описания бинарных переменных:

<имя файловой переменной>: **file of** <тип>;

имя файловой переменной – идентификатор Паскаля,

file – служебное слово «файл»,

of – служебное слово «из»,

тип – в нашем случае – числовой тип данных.

Примеры:

```
…
var
…
ff1: text; fb1,fb2: file of real;
…
```

В бинарных файлах числа хранятся в бинарном виде (комбинация нулей и единиц), каждое число в соответствии с **типом** имеет определённую длину в байтах, причём числа следуют друг за другом в одной строке подряд без пробелов (или без других разделителей). Отдельные числа идентифицируются по длине типа. В текстовом представлении такой файл выглядит как бессмысленная цепочка символов (любых из 256) – т. е. последовательность нулей и единиц нарезается по два байта и выводится в виде соответствующих символов. Текстовый же файл вполне понятен для человека, но перед использованием в программе его содержимое должно быть преобразовано в бинарный вид.

После описания файловых переменных в основной части программной единицы (третий раздел) перед работой с файлами эти переменные должны быть связаны с соответствующим файлом – с использованием функции assign:

assign(<имя файловой переменной>,'<имя файла>');

имя файла – имя файла с возможным путём; если путь не указан, файл должен находиться в той же директории (папке), что и сама программа. Если файла с указанным именем нет, то он **создаётся**.

Затем файл должен быть **открыт** для работы; открытие возможно в трёх режимах: режим чтения из файла, режим записи в файл и смешанный режим. Из многих библиотечных функций можно рекомендовать две:

rewrite(<имя файловой переменной>);

– для вывода в файл, если в файле содержится какая-то информация, она уничтожается;

reset(<имя файловой переменной>);

– для чтения из файла, если в файле содержится какая-то информация, при открытии файла она сохраняется.

Нужно следить, чтобы связка с файлом и его открытие не происходило в цикле (если, конечно, это не предусмотрено программой).

После открытия файла с ним можно **работать** в соответствии с алгоритмом. Для этого используются те же операторы ввода-вывода: read, readln, write, writeln, только в **начале списка ввода или вывода указывается имя файловой переменной**. При работе с бинарными файлами существуют ограничения: если, например, файл – числового

типа, то не допускаются текстовые атрибуты, нельзя использовать readln и writeln, нельзя выводить и вводить текстовую информацию.

По окончании работы файл должен быть **закрыт** с помощью функции close:

close(<имя файловой переменной>);

Если файл не закрыть, то возможна полная или частичная потеря информации в нём.

В качестве примера приведём обмен информацией между двумя последовательными программами. Первая программа выводит в бинарный и текстовый файлы двумерный массив, как результат своей работы. Текстовый файл служит для визуального контроля информации, а с бинарного файла вторая программа будет считывать информацию как исходные данные. Вторая программа будет выводить в другой текстовый файл результирующий массив.

```
program pr1;                                  // первая программа
const
n=20;m=30;
var
a:array [0..n, 0..m] of real; f:real;        // описание массива
i,j:integer; ff1: text; fb1: file of real;    // описание файловых
                                              // переменных
…
begin                    // начало третьего раздела программы
…
for i:=0 to n do for j:=0 to m do begin
…
a[i,j]:=…                // формирование массива a согласно
                         // какому-то алгоритму
… end;
…
assign(ff1, ‘filet.txt’); rewrite(ff1);       // открытие текстового
                                              // файла filet.txt
assign(fb1, ‘fileb.bin’); rewrite(fb1);       // открытие бинарного
                                              // файла fileb.bin
// расширение .bin – нестандартное, этот файл можно открыть в
// «Блокноте» как текстовый
…
```

```
for i:=0 to n do for j:=0 to m do begin
writeln('a[',i,',',j,']=',a[i,j]);          // вывод массива a на экран
writeln(ff1,'a[',i,',',j,']=',a[i,j]);      // вывод массива a в текстовый
                                            // файл filet.txt
write(fb1,'a[',i,',',j,']=',a[i,j]);        // вывод массива a в бинарный
                                            // файл fileb.bin
end;
…
close(ff1); close(fb1);          // закрытие файлов filet.txt и fileb.bin
…
end.                             // конец первой программы
```

Итак, первая программа рассчитала массив a и вывела его в два файла, текстовый filet.txt (для контроля) и бинарный; последний будет исходным для второй программы. Вторая программа считывает исходные данные с файла fileb.bin и заносит их в массив b. Затем осуществляется алгоритм второй программы, в результате чего формируется выходной массив c, который выводится в файл filet2.txt.

```
program pr2;                                  // вторая программа
const
n=20;m=30;
var
b,c:array [0..n, 0..m] of real; f:real;       // описание массивов
i,j:integer; ff2: text; fb2: file of real;    // описание файловых
                                              // переменных
…
begin                        // начало третьего раздела программы
…
assign(fb2, 'fileb.bin'); reset(fb2);  // открытие бинарного файла
                                       // fileb.bin для чтения
…
for i:=0 to n do for j:=0 to m do begin
…
read(fb2, b[i,j]);  // ввод массива b поэлементно из файла fileb.bin
… end;
close(fb2);                                   // закрытие файла fileb.bin
…
```

```
for i:=0 to n do for j:=0 to m do begin
…
c[i,j]:=…                    // формирование массива с согласно
                             // какому-то алгоритму
… end;
…
assign(ff2, ‘filet2.txt’); rewrite(ff2);      // открытие текстового
                                              // файла filet2.txt
…
for i:=0 to n do for j:=0 to m do begin
writeln(‘c[‘,i,’,’,j,’]=’,c[i,j]);            // вывод массива a на экран
writeln(ff2,‘c[‘,i,’,’,j,’]=’,c[i,j]);        // вывод массива a в текстовый
                                              // файл filet2.txt
end;
…
close(ff2);                                   // закрытие файла filet2.txt
…
end.                                          // конец второй программы
```

Результаты, выведенные в текстовый файл, очень удобны, их можно проанализировать, построить график функции, численно продифференцировать и проинтегрировать и так далее.

КОНТРОЛЬНЫЕ ЗАДАНИЯ

Задание № 1. Разветвляющиеся алгоритмы. Построить блок-схемы и написать программы.

1. $y=\begin{cases} x & \text{при } -10<x<0 \\ \sin x & \text{при } 0<x<20 \end{cases};$

2. $y=\begin{cases} \ln|x| & \text{при } -2<x<-1 \\ \lg x & \text{при } -1<x<0 \end{cases};$

3. $y=\begin{cases} x^2-x & \text{при } -5<x<0 \\ x^3+\sin x & \text{при } 0<x<5 \end{cases};$

4. $y=\begin{cases} \arcsin x & \text{при } -1<x<0 \\ 1-\cos x & \text{при } 0<x<1 \end{cases};$

5. $y=\begin{cases} \ln\left|\dfrac{1+x}{1-x}\right| & \text{при } 2<x<3 \\ x^3 & \text{при } 3<x<6 \end{cases};$

6. $y=\begin{cases} \dfrac{1}{x^2+1} & \text{при } -5<x<0 \\ x^2+1 & \text{при } 0<x<5 \end{cases};$

7. $y=\begin{cases} e^x & \text{при } -5<x<0 \\ 1-e^x & \text{при } 0<x<5 \end{cases};$

8. $y=\begin{cases} \dfrac{x^2-1}{x^2+1} & \text{при } -7<x<0 \\ x^2-1 & \text{при } 0<x<7 \end{cases};$

9. $y=\begin{cases} \ln|1+x| & \text{при } -2<x<0 \\ \ln|1-x| & \text{при } 0<x<2 \end{cases};$

10. $y=\begin{cases} \sin e^{-x} & \text{при } -20<x<0 \\ \cos e^{-x} & \text{при } 0<x<20 \end{cases}.$

Задание № 2. Записать арифметические выражения по правилам языка Паскаль.

1. $y=\cos^2(\beta)+\ln\left|\dfrac{1+x}{1-x}\right|+\dfrac{a^2+b^2+d^3}{\sqrt[3]{z}-c}+E_1.$

2. $y=\sqrt{\sin(\alpha)}+\dfrac{\varepsilon^2-\sqrt[3]{\delta}}{c^2-z^2+d}-G_2+\ln\left|x-\sqrt[5]{x^2-1}\right|.$

3. $y=\operatorname{tg}^3(\varepsilon)-E_3-\ln\left|\dfrac{1-x}{\sqrt{x^2-1}}\right|+\dfrac{\varepsilon^2+\delta^3-d^2}{\sqrt[3]{a}-c}.$

4. $y=\ln|\sin(\beta)|-\sqrt{\dfrac{\varepsilon^2+\sqrt[3]{\delta}}{a^2-b^2+d}}+F_3-e^z.$

5. $y=\ln|\cos(\beta)|-\exp(-\varepsilon+d/b+a)-\sqrt{\dfrac{\sqrt[3]{G}-c}{a+\varepsilon}}.$

6. $y = \exp(-\cos^2(\beta)) + (1+x)/(1-x) + \ln\left|a^2 + b^2 - \frac{d^3}{\sqrt[3]{z} - c}\right|$.

7. $y = \exp\left(-\sqrt{|\sin(\alpha)|} + x\right) - \sqrt{x^2 + 1} + \ln\left|\varepsilon^2 - \frac{\sqrt[3]{\delta}}{a^{1/3} - c}\right|$.

8. $y = \operatorname{arctg}\left(\exp\left(\sqrt{|\sin(\alpha)|}\right)\right) + \ln\left|\frac{a^2 + b^2 - d^3}{c^2 - z^2 + d}\right|$.

9. $y = \exp\left(-\operatorname{tg}^2(\varepsilon)\right) - G_6 + \frac{1+x}{\sqrt{1+x^2}} - \ln\left|\frac{\varepsilon^2 + \delta^2 - \xi^2}{a^{1/3} - c}\right|$.

10. $y = \sin\left(e^{-\operatorname{tg}^2(\varepsilon)}\right) + \ln\left|x - \sqrt{x^2 - 1}\right| + \frac{a^2 - d^3}{\sqrt[3]{z} - c}$.

Задание № 3. Простые циклы. Составить блок-схемы и написать программы с использованием оператора **for**.

№ п/п	Функция	Начальное значение аргумента	Шаг изменения аргумента	Число циклов
1	$y = 2\frac{\sin x}{x}$	0.1	0.05	10
2	$y = 0.51\sqrt{\|x\|} - x$	-17.5	0.5	5
3	$y = 1 + 2x - x^2$	2.54	0.02	8
4	$y = 2x - \frac{9.81x^2}{2}$	5	10	7
5	$y = \frac{1.24}{\sqrt{x}} - x^2 \cos x$	13.5	0.2	8
6	$y = 1265x^2 - 2.3\sqrt{\|x\|}$	0.05	0.01	10
7	$y = 2 - 5\sin^2 x$	0.28	0.02	12
8	$y = 0.05x^2 - 0.075x - 26$	12	2	6
9	$y = \cos^2 x - 2x^2$	0.5	0.5	8
10	$y = \sin^2(x^2 - 2.5)$	1.2	0.2	10

Задание № 4. Вложенные циклы. Составить блок-схемы и написать программы с использованием операторов **repeat** и **while** в любом порядке.

№ п/п	Функция	Начальное значение аргумента	Шаг изменения аргумента	Число циклов
1	$y=\frac{a^4-\sin a}{a^3-2ax}$	$x_0=1.6$ $a_0=-30$	$\Delta x=0.2$ $\Delta a=5$	$n=6$ $k=4$
2	$y=\frac{4\sin x-1}{\ln a}$	$x_0=-10.5$ $a_0=3.4$	$\Delta x=0.5$ $\Delta a=0.4$	$n=6$ $k=4$
3	$y=(x^2-b)(3x+5)$	$x_0=2.4$ $b_0=-23.5$	$\Delta x=0.2$ $\Delta b=1.5$	$n=5$ $k=6$
4	$z=\frac{x^2-y^2}{2xy}$	$x_0=0.25$ $y_0=9.1$	$\Delta x=0.5$ $\Delta y=0.1$	$n=4$ $k=4$
5	$z=\frac{\sqrt{\lvert x+y\rvert}}{x^2+y^2}$	$x_0=-12$ $y_0=8.5$	$\Delta x=2$ $\Delta y=2.5$	$n=5$ $k=4$
6	$z=x^2-xy+\sin y$	$x_0=0.93$ $y_0=1.4$	$\Delta x=0.02$ $\Delta y=0.1$	$n=4$ $k=5$
7	$z=\cos^2 x+2xy$	$x_0=-3.5$ $y_0=12.5$	$\Delta x=0.5$ $\Delta y=0.5$	$n=5$ $k=3$
8	$z=x^2+y^2-\sin y$	$x_0=-2.5$ $y_0=5.0$	$\Delta x=0.25$ $\Delta y=1$	$n=2$ $k=10$
9	$z=\ln(x^2-xy)$	$x_0=0.85$ $y_0=3.8$	$\Delta x=0.05$ $\Delta y=0.4$	$n=3$ $k=6$
10	$z=2\sin^2(x-y)-x$	$x_0=7.75$ $y_0=2.1$	$\Delta x=0.25$ $\Delta y=0.1$	$n=4$ $k=5$

Задание № 5. Работа с массивами чисел. Построить блок-схемы и написать программы.

1. $Z = 0.5 - \sum_{i=1}^{n} \left(x_i^{\,2} - 0.5 y_i \right);$

$x_i = 0.53;\ 0.28;\ 0.64;\ 0.34;\ 0.60;\ 0.97;\ 0.09;\ 0.34;\ 0.33;\ 0.42;$

$y_i = 1.93;\ 1.78;\ -1.56;\ 0.13;\ 0.58;\ 0.95;\ 0.86;\ 0.82;\ 0.16;\ -0.37.$

2. $V = \prod_{i=1}^{n} \left(v_i - \frac{1}{v_i} \right)^2;$

$v_i = 6.60;\ 6.57;\ 4.71;\ 1.73;\ 6.01;\ 1.08;\ 8.05;\ 2.69;\ 9.77;\ 1.35;\ 7.33;$

$9.95;\ 5.94;\ 7.31;\ 3.48;\ 7.51;\ 6.49;\ 4.01;\ 0.75;\ 6.50;\ 6.28.$

3. $Y = 1 - \prod_{i=1}^{n} \left(x_i^{\,2} - \sqrt{x_i} \right);$

$x_i = x_i = 4.49;\ 9.35;\ 2.49;\ 9.40;\ 9.97;\ 7.65;\ 3.59;\ 3.15;\ 8.05;\ 8.42;$

$2.36;\ 8.09;\ 1.39;\ 9.22;\ 2.68;\ 3.91;\ 6.47;\ 0.24.$

4. $W = 10,72 + \prod_{i=1}^{n} \left(u_i - \frac{1}{u_i} \right)^2;$

$u_i = u_i = 4.49;\ 3.51;\ 2.49;\ 7.65;\ 3.15;\ 8.39;\ 8.08;\ 5,98;\ 6.40;\ 6.33;$

$3.82;\ 4.58.$

5. $Y = \frac{1}{n-1} \sum_{i=1}^{n} \left(x_i - \frac{1}{n} \sum_{i=1}^{n} x_i \right);$

$x_i = 0.75;\ 0.24;\ 0.80;\ 0.93;\ 0.07;\ 0.63;\ 0.60;\ 0.64;\ 0.42;\ 0.19;\ 0.77;$

$0.85;\ 0.57;\ 0.87;\ 0.89.$

6. $Y = 0.68 \sum_{i=1}^{n} \left(x_i - \sqrt{|x_i|} \right);$

$x_i = 0.98;\ 0.52;\ 0.01;\ 0.77;\ -0.67;\ -0.14;\ 0.90.$

7. $Z = \frac{1}{n}\sum_{i=1}^{n}\left(1+\cos^2 x_i\right);$

$x_i = 0.48;\ 0.35;\ 0.02;\ 0.68;\ 0.70;\ 0.97;\ 0.52;\ 0.99;\ 0.77;\ 0.53;\ 0.31;$

$0.43;\ 0.24;\ 0.07;\ 0.05;\ 0.70;\ 0.74;\ 0.57.$

8. $S = 36.8 + \prod_{i=1}^{n}\left(1-\cos x_i\right);$

$x_i = 0.48;\ 0.35;\ 0.02;\ 0.70;\ 0.97;\ 1.81;\ 1.51;\ 1.82;\ 1.06;\ 1.09;\ 0.52;$

$0.14;\ 0.27;\ 0.45.$

9. $C = 56.4 - \sum_{i=1}^{n}\left(\sin x_i - 0.2\right)^2;$

$x_i = 0.65;\ 0.97;\ 0.09;\ 0.34;\ 0.33;\ 0.85;\ 0.47;\ 0.33;\ 0.60;\ 0.52;\ 0.03;$

$0.68;\ 0.66;\ 0.80;\ 0.34.$

10. $V = \prod_{i=1}^{n} x_i - \prod_{i=1}^{n}\left(2-x_i\right)^2;$

$x_i = 0.31;\ 0.74;\ 0.73;\ 0.06;\ 0.01;\ 0.33;\ 0.09;\ 0.47;\ 0.93;\ 0.79;\ 0.34;$

$0.69;\ 0.36;\ 0.68;\ 0.51;\ 0.24.$

Задание № 6. Итерационные циклы. Построить блок-схемы и написать программы для вычисления суммы сходящегося ряда с заданной точностью ε.

1. $1 + \frac{x\ln a}{1!} + \frac{\left(x\ln a\right)^2}{2!} + \frac{\left(x\ln a\right)^3}{3!} + \ldots + \frac{\left(x\ln a\right)^n}{n!} + \ldots$ при $x = 2;\quad a = 10;$

2. $x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + \left(-1\right)^n \frac{x^{n+1}}{n+1} \pm \ldots$ при $x = 0.5;$

3. $2\left(x+\frac{x^3}{3}+\frac{x^5}{5}+\frac{x^7}{7}+\ldots+\frac{x^{2n+1}}{2n+1}+\ldots\right)$ при $x=0.5$;

4. $x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\ldots+(-1)^n\frac{x^{2n+1}}{(2n+1)!}\pm\ldots$ при $x=1$;

5. $\frac{\pi}{2}-\frac{1}{x}+\frac{1}{3x^3}-\frac{1}{5x^5}+\frac{1}{7x^7}-\ldots+(-1)^{n-1}\frac{x^{2n+1}}{(2n+1)x^{2n+1}}\pm\ldots$ при $x=3$;

6. $x+\frac{x^3}{3!}+\frac{x^5}{5!}+\frac{x^7}{7!}+\ldots+\frac{x^{2n+1}}{(2n+1)!}+\ldots$ при $x=1$;

7. $2\left(\frac{x-1}{x+1}+\frac{(x-1)^3}{3(x+1)^3}+\frac{(x-1)^5}{5(x+1)^5}+\ldots+\frac{(x-1)^{2n+1}}{(2n+1)(x+1)^{2n+1}}+\ldots\right)$ при $x=3$;

8. $\frac{(x-1)}{x}+\frac{(x-1)^2}{2x^2}+\frac{(x-1)^3}{3x^3}+\ldots+\frac{(x-1)^n}{nx^n}+\ldots$ при $x=2.5$;

9. $\cos x-\frac{1}{2}\cos 2x+\frac{1}{3}\cos 3x-\ldots+(-1)^n\frac{\cos(n+1)x}{n+1}+\ldots$ при $x=3$;

10. $-\frac{1}{2}\sin x+\left(\frac{4\sin 2x}{1\cdot 3}-\frac{6\sin 3x}{3\cdot 5}+\ldots+\frac{(-1)^n\sin(n+1)x\cdot 2(n+1)}{(2n-1)(2n+1)}\pm\ldots\right)$ при $x=-2$.

ЗАКЛЮЧЕНИЕ

Представленный в учебном пособии курс – вводный для всех математически ориентированных химических дисциплин, таких как численные методы в химии, математические методы в химии и химической технологии, математическое моделирование, системное моделирование химико-технологических процессов, компьютерная химия. Он может использоваться при изучении аналитической химии, физической химии, строения вещества, коллоидной химии, химической технологии.

Изучение материала начато с графического представления алгоритмов на языке блок-схем. Рассмотрены основные понятия, методы и приёмы программирования для решения вычислительных задач, даны основные сведения по профессиональному языку высокого уровня – Паскалю, который является базовым для Delphi и Lasarus. После освоения студентами материала и получения необходимых навыков программирования на Паскале дальнейшее использование блок-схем не нужно, и они не будут применяться в математических курсах химии.

Полученные студентами базовые знания основ программирования и практическое освоение языков Delphi или Lasarus в необходимом минимуме позволят им решать вычислительные задачи и проводить соответствующие расчёты. Кроме этого данный курс поможет будущим специалистам быстро самостоятельно освоить прикладные математические пакеты, такие как Matlab и Mathcad.

В следующих учебных пособиях, в которых будут освещены численные методы, использующиеся для решения вычислительных задач в химии, физической химии и химической технологии, а также основы моделирования состояний и процессов в этих областях, практическая реализация этих методов будет представлена на Паскале как базовом языке для Delphi и Lasarus.

РЕКОМЕНДАТЕЛЬНЫЙ БИБЛИОГРАФИЧЕСКИЙ СПИСОК*

Основной

1. *Фаронов, В. В.* Turbo Pascal : учеб. пособие для вузов по направлению «Информатика и вычислительная техника» / В. В. Фаронов. – СПб. : Питер, 2007.

2. *Фаронов, В. В.* Delphi 2005: язык, среда, разработка приложений / В. В. Фаронов. – СПб. : Питер, 2007.

3. Marco Cantu. Delphi 2010 Handbook (CodeGear Delphi 2010). 2010.

4. *Фаронов, В. В.* Delphi. Программирование на языке высокого уровня : учеб. для вузов / В. В. Фаронов. – СПб. : Питер, 2006.

5. *Бобровский, С. И.* Delphi 7 : учеб. курс / С. И. Бобровский. – СПб. : Питер, 2006.

6. *Фаронов, В. В.* Турбо Паскаль 7.0: Начальный курс : учеб. пособие для вузов / В. В. Фаронов. – М. : КНОРУС, 2005.

7. *Культин, Н. Б.* Программирование в Turbo Pascal 7.0 и Delphi. – СПб. : BHV-Санкт-Петербург, 1997.

Дополнительный

1. *Санников, Е. В.* Курс практического программирования в Delphi. Объектно-ориентированное программирование [Электронный ресурс] / Е. В. Санников. – М. : СОЛОН-ПРЕСС, 2013. – URL: http://www.studentlibrary.ru/book/ISBN9785913591227.html (дата обращения: 12.03.2018).

2. *Федотова, С. В.* Создание Windows-приложений в среде Delphi. [Электронный ресурс] / С. В. Федотова. – М. : СОЛОН-ПРЕСС, 2010. – URL http://www.medcollegelib.ru/book/ISBN5980031766.html (дата обращения: 15.03.2018).

3. *Осипов, В. П.* Практикум по программированию на языке Delphi. В 2 ч. Ч. 1. Структурное программирование [Электронный ресурс] : учеб. пособие / В. П. Осипов. – М. : Изд-во МГТУ им. Н. Э. Баумана, 2010. – URL http://www.studentlibrary.ru/book/bauman_0359.html. (дата обращения: 12.04.2018).

* Публикуется в авторской редакции.

Учебное издание

ЛОБКО Владимир Николаевич

МАТЕМАТИЧЕСКИЕ МЕТОДЫ В ХИМИИ И ХИМИЧЕСКОЙ ТЕХНОЛОГИИ

Основы программирования вычислительных задач

Учебное пособие

Редактор Е. В. Невская

Технический редактор А. В. Родина

Корректор О. В. Балашова

Компьютерная верстка Л. В. Макаровой

Выпускающий редактор А. А. Амирсейидова

Подписано в печать 26.12.18.
Формат 60×84/16. Усл. печ. л. 6,28. Тираж 50 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.