ПРЕДИСЛОВИЕ

Новые языки и средства программирования появляются непрерывно, поэтому программист вынужден учиться всю жизнь. Очень важно это делать быстро и эффективно, а значит, к освоению каждого языка подходить системно: выделить составные части, понять их организацию и взаимосвязь, найти сходства и отличия от средств, изученных ранее.

Курс «Методы программирования и информатика» является базовым, в нем рассматривается общий подход к изучению языков программирования и современных объектно-ориентированных технологий.

В учебном пособии предлагается последовательное изложение основ программирования на примере языка Pascal, который, будучи достаточно простым (а следовательно, легко поддающимся изучению), содержит все типы данных и реализует все основные конструкции структурного, модульного и объектно-ориентированного программирования, присущие современным языкам высокого уровня. Строгий синтаксис обеспечивает хорошую диагностику ошибок, что очень важно не только для начинающих. Наиболее распространенные среды программирования обеспечивают удобные средства написания и отладки программ.

Авторы считают, что обучение должно вестись на основе:

- выделения элементарных операций при построении типовых алгоритмов обработки простых данных, структурированных статических и динамических данных;
- одинаковой формы записи алгоритма для решения задач с одинаковой структурой исходных данных;
- выделения вспомогательных алгоритмов, которые затем оформляются подпрограммами языка и могут объединяться в модули.

Учебное пособие разработано на основе учебных и методических материалов, используемых авторами в процессе обучения студентов алгоритмизации и программированию. Алгоритмизация и программирование – наиболее важные разделы учебной дисциплины «Методы программирования и информатика», которые позволяют формировать алгоритмическое мышление у обучающихся. Здесь рассматриваются структурная, модульная и объектно-ориентированная технологии программирования, методы проектирования и отладки программ, управляющие операторы, основные структуры данных и методы их обработки.

Авторы

ВВЕДЕНИЕ

Ключевые положения. В развитии компьютеров выделяют *четыре поколения*, каждое из которых характеризуется использованием своих базовых физических компонентов: электронно-лучевых трубок, транзисторов, интегральных схем и микропроцессоров.

В соответствии с размером, мощностью и стоимостью выделяют три типа компьютеров: универсальные машины, мини-компьютеры и персональные компьютеры.

Дж. Нейманом предложены основные архитектурные решения и принципы, реализованные в цифровых электронных машинах (ЦЭМ):

- *память*, в которой сохраняется программа, данные (числа) и результаты промежуточных вычислений (в двоичной системе счисления);
- *программа*, которая вводится в машину так же, как данные;
- *адресный принцип*: в команде указываются не сами числа, над которыми нужно выполнять действия, а адреса (т.е. номера ячеек памяти), где эти числа хранятся;
- *автоматизм*: после ввода программы и данных машина работает автоматически, выполняя указания программы без вмешательства человека; для этого машина всегда помнит адрес выполняемой команды, а каждая команда содержит (явное или нет) указание об адресе той команды, которую нужно выполнить следующей;
- *переадресация*: адреса ячеек памяти, указанные в команде, можно подсчитывать и преобразовывать как числа, значит, ЦЭМ может сама строить команды, которые выполняет.

Этих принципов придерживались при создании первых компьютеров, сейчас существуют и другие подходы.

Оборудование – это физические (аппаратные) компоненты компьютера. Схема типового персонального компьютера содержит компоненты, которые принадлежат *центральному процессору*, *монитору* (*дисплею*), *устройству ввода* (клавиатура, мышь, микрофоны, сканеры, цифровые видеокамеры и т.д.), *устройству вывода* (принтер, акустическая система и т.д.), *памяти* постоянной (ROM) и оперативной (RAM), *устройству долговременного хранения* или *носителям* (разнообразные диски).

Совокупность программ компьютера, позволяющих использовать его по назначению, называется ***программным обеспечением***. Наиболее важная часть программного обеспечения – *операционная система*, которая управляет работой других программ и взаимодействием всех компонентов компьютера.

Центральный процессор персонального компьютера работает только с двоичными числами. В двоичной системе счисления есть только две цифры – 0 и 1. В электронных устройствах эти две цифры представляют

два устойчивых состояния: включенное и выключенное. Каждый символ двоичного числа называется ***битом***. Информация в компьютерах хранится группами по 8 бит – ***байтами***. Объем памяти измеряют в байтах. На практике в качестве единиц измерения количества информации выступают *килобайты*, *мегабайты*, *гигабайты* и др. Для хранения данных используется конечное количество байтов, поэтому существуют ограничения на величину данных как целочисленных, так и вещественных. Два последовательных байта называют ***словом***, четыре – ***двойным словом***.

Физически центральный процессор представляет собой микропроцессор, размещенный на материнской плате компьютера. ***Микропроцессор*** – это интегральная микросхема, которая выполняет основные функции компьютера. В состав микропроцессора входят небольшая внутренняя оперативная память, элементы которой обычно называют *регистрами* – устройство управления, которое координирует взаимодействие различных частей компьютера, и *арифметико-логическое устройство*, которое выполняет арифметические и логические операции.

Развитие языков компьютерного программирования. Так как компьютер может обрабатывать информацию в системах счисления, отличных от десятичной (двоичная, троичная системы счисления), то для первых компьютеров программистам приходилось писать программы в кодах. Их называли ***программами на машинном языке***. Позднее, чтобы облегчить создание программ, был разработан язык ***ассемблер***. В ассемблере машинные команды представлялись *мнемонически-символьными инструкциями*. Когда программа на ассемблере написана, ее нужно преобразовать в программу на машинном языке. Для автоматического выполнения такого перевода создали специальную программу – ***транслятор ассемблера***. Каждой команде ассемблера приблизительно соответствует одна команда машинного языка, поэтому ассемблер называют языком низкого уровня.

В зависимости от порядка выполнения программ языки высокого уровня делятся на компилируемые и интерпретируемые.

Компилятор – это программа, которая автоматически преобразовывает (транслирует, компилирует) исходный код языка высокого уровня в машинный код и создает, таким образом, выполняемый файл (объектный код).

Интерпретатор – это программа, преобразующая (транслирующая, интерпретирующая) исходный код языка высокого уровня в машинный код шаг за шагом, т.е. каждая команда (оператор) исходной программы преобразовывается интерпретатором и тут же выполняется компьютером, и так до тех пор, пока не закончится исходная программа на языке высокого уровня.

В дальнейшем тенденция облегчения для человека процесса создания программ осталась доминирующей. В результате были разработаны

языки высокого уровня, программные конструкции которых подобны предложениям английского языка. Примерами таких языков в наше время служат Fortran, Basic, Pascal, C, Java. Большинство языков высокого уровня универсальные, они предназначены для решения самого широкого круга задач. Количество используемых языков программирования сейчас достаточно большое.

В последнее время разрабатываются объектно-ориентированные языки высокого уровня: в них поддерживается создание и применение объектов. Концепция объектно-ориентированного программирования приближает компьютерные программы к реальной жизни. Здесь происходит визуализация программирования. Примерами объектно-ориентированных языков являются C++, Java, Visual Basic, Object Pascal.

Эволюция Pascal. Язык программирования Pascal, созданный в 1970-е гг. швейцарским ученым и преподавателем Никлаусом Виртом в Цюрихе (Швейцария) как учебный язык компьютерного обучения программированию, назван в честь известного французского философа, математика, физика и писателя Блеза Паскаля. Хорошая структурированность языка помогает привить начинающим программистам правильные навыки программирования. В результате язык Pascal быстро приобрел широкую популярность и стал основным учебным языком во многих университетах мира. Благодаря богатым функциональным возможностям, легкости составления программ и высокой скорости компиляции язык Pascal стал интенсивно использоваться также для создания прикладных программ.

Компанией «Borland» была разработана популярная версия языка – Turbo Pascal. По мере развития версий операционных систем Windows и распространения концепции объектно-ориентированного программирования язык Pascal был расширен до Turbo Pascal for Windows и Object Pascal for Windows.

Следующим шагом стало создание Delphi – среды разработки программ на Object Pascal. В систему Delphi входят компилятор с Object Pascal, визуальная среда разработки, инструменты взаимодействия с базами данных и библиотека VCL (Visual Components Library – библиотека визуальных компонент). Система Delphi создана в полном соответствии с концепцией RAD (Rapid Application Development – быстрая разработка приложений), ее использование значительно повышает скорость разработки приложений под Windows.

Free Pascal. Free Pascal (полное название Free Pascal Compiler, часто используется сокращение FPC) – это свободно распространяемый компилятор языка Pascal с открытым исходным кодом.

Free Pascal – кроссплатформенный инструмент, поддерживающий большое количество платформ. Среди них – DOS, Linux, OS/2, MacOS(X), Win32.

Важной особенностью данного компилятора является ориентация на совместимость с распространенными коммерческими диалектами языка: Borland Pascal 7, Object Pascal и Delphi. Free Pascal поддерживает компиляцию в нескольких режимах, обеспечивающих совместимость с различными диалектами и реализациями языка:

- TP – режим совместимости с Turbo Pascal: совместимость практически полная, за исключением нескольких моментов, связанных с тем, что FPC компилирует программы для защищенного режима процессора, где невозможно прямое обращение к памяти, портам и т.д.;
- FPC – собственный диалект: соответствует предыдущему, расширенному дополнительными возможностями, такими как, например, перегрузка операций;
- Delphi – режим совместимости с Borland Delphi: включает поддержку классов и интерфейсов;
- OBJFPC – совмещает объектно-ориентированные возможности Delphi и собственные расширения языка;
- MacPas – режим совместимости с Mac Pascal.

Контрольные вопросы и задания

1. Расскажите о поколениях и типах компьютеров.
2. Что такое машинный язык? Что такое ассемблер? С какой целью были разработаны языки низкого и высокого уровней?
3. Чем языки высокого уровня отличаются от языков низкого уровня?
4. Дайте определение следующим понятиям: транслятор, компилятор, интерпретатор.
5. Перечислите основные этапы решения задач на ЭВМ.
6. Что такое псевдокод?

РАЗДЕЛ 1. СТРУКТУРНАЯ МЕТОДОЛОГИЯ РАЗРАБОТКИ ПРОГРАММ

1.1. АЛГОРИТМ

Слово «алгоритм» по сути является синонимом слов «способ, рецепт» и т.д. Возникло оно в Средние века, когда европейцы познакомились со способами выполнения арифметических действий (сложения и умножения) над числами, записанными в арабской системе счисления. Способы эти описаны в книге «Китаб аль-джебр ва-ль-мукабала» («Книга о сложении и вычитании») Абу Абдуллаха Мухаммеда ибн Муса аль-Хорезми (слово «алгоритм» возникло вследствие европеизированного произношения слов «аль Хорезм»).

В наше время интерес к алгоритмам связан с возможностью использования ПК в технике, науке, экономике, в повседневной жизни, ведь компьютер во время работы выполняет задаваемую программу, а программа является некоторым алгоритмом, записанным на языке, который переводится на язык ПК.

Язык программирования – совокупность средств и правил представления алгоритма в виде, приемлемом для компьютера. Основу языков программирования составляют *алгоритмические языки*.

Существует много определений понятия «алгоритм» (есть даже специальная дисциплина «Теория алгоритмов»). Мы остановимся на следующем определении.

Алгоритм – предписание исполнителю выполнить точно определенную последовательность действий, направленных на достижение заданной цели или решение поставленной задачи.

Существуют разнообразные методики и технологии разработки алгоритмов.

Разработка компьютерной программы – длинный и трудоемкий процесс. Чтобы окончательный вариант программы работал правильно и содержал как можно меньше ошибок, программисты стараются придерживаться полного цикла разработки программы.

1.2. ОСНОВНЫЕ ЭТАПЫ РЕШЕНИЯ ЗАДАЧ НА ЭВМ

Какими бы разнообразными по своей сложности ни были задачи, например решение квадратного уравнения или управление космическим кораблем, их решение на ЭВМ имеет ряд общих этапов.

1. Постановка задачи.

2. Анализ, формализованное описание задачи, выбор математической модели. Разработка методов решения и определение ограничений на поставленную задачу.

3. Выбор или разработка алгоритма и запись его формальными средствами.

4. Программирование решения задачи на одном из языков программирования.

5. Тестирование и отладка программы.

6. Решение задачи на ЭВМ.

При решении конкретных задач некоторые этапы могут отсутствовать, а другие быть сложно разрешимыми.

Этап 1. Постановка задачи выполняется заказчиком, и на первых порах она может не быть явно алгоритмической.

Этап 2. Анализ задачи включает определение входных и выходных данных, выявление возможных ограничений на их значения и обычно завершается формализованным описанием задачи, которое зачастую предполагает ее математическую формулировку.

Этап 3. Выбор или разработка алгоритма и метода решения задачи имеют огромное значение для успешной работы над программой. Точно продуманный алгоритм решения задачи – необходимое условие эффективного программирования.

Для формализации алгоритма существуют разные средства:

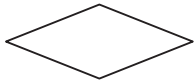
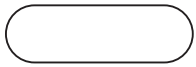
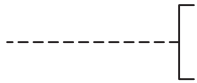
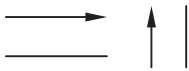
- запись на родном языке;
- запись на формализованном языке – псевдокоде;
- графические средства, используемые программистом (структурные схемы, структурограммы);
- графические средства, которые использует компьютер при создании *p*-схем.

Когда программирование происходит на языках высокого уровня, структурная схема используется как язык сверхвысокого уровня. Слишком подробная схема затрудняет работу, а вот принципиальная, которая задает последовательность действий, в том числе сложных, – наиболее приемлема.

В дальнейшем мы будем применять в основном структурные схемы и структурограммы.

В ***структурных схемах (блок-схемах)*** используются графические символы схем алгоритмов и программ. В табл. 1.1 приведены некоторые блоки и свойственные им функции.

Таблица 1.1

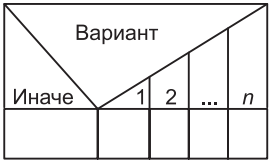
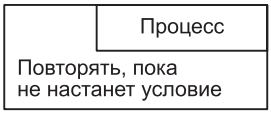
Обозначение блока	Функция
	Выполнение операций, в результате которых изменяется значение данных
	Выбор направления выполнения алгоритма в зависимости от некоторых условий
	Выполнение операций, которые изменяют команды или группы команд
	Начало или конец схемы программы
	Использование ранее созданных и отдельно описанных алгоритмов
	Указание связи между прерванными линиями потока, связывающими символами
	Обмен данными между внешней и оперативной памятью
	Ввод-вывод данных, носителем которых служит бумага
	Связь между элементами схемы и толкованием
	Очевидное (сверху вниз или слева направо без стрелок) и не очевидное (со стрелками) обозначение линий связи блоков

В блоках позволяется писать пояснительные действия. Например, $\boxed{x := 1}$ означает следующий процесс: переменной x присвоить значение 1. Знак «:=» – присвоить значение (вместо «=» – равно).

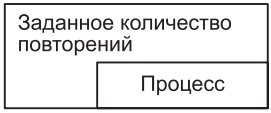
Недостаток структурных схем в том, что они занимают много места. Поэтому стали применяться другие способы, например диаграммы Насси – Шнейдермана, или структурограммы.

Структурограммы – это вспомогательные средства для графического отображения алгоритмов. Они представляют удобную систему описания и понимания программ и используются *для иллюстрации процесса передачи управления в программе*. Основными конструкциями структурограммы являются процессы, а ее наиболее простым элементом – прямоугольник (табл. 1.2). Вся структурограмма – это прямоугольник, который разделяется на процессы-прямоугольники. Структурограмма позволяет компактно сконструировать общую последовательность действий, которая приводит к решению задачи. Управление осуществляется сверху вниз.

Таблица 1.2

Обозначение символа	Функция	Операторы языка
	Любая группа действий, образующая блок	begin ... end
	Выбор направления выполнения алгоритма в зависимости от некоторого условия	if ... then ... else
	Выбор варианта дальнейшего действия	case ... of ... end
	Цикл с предусловием	while ... do
	Цикл с постусловием	repeat ... until

Окончание табл. 1.2

Обозначение символа	Функция	Операторы языка
	Цикл на заданное количество повторений	for ... to ... do for ... downto ... do

Блок-схемы требуют меньшей квалификации при их разработке. При использовании же структурограмм нужно знать о линейных и нелинейных процессах, циклах с пред-, постусловиями или цикле на известное количество повторений.

В каждой из форм представления алгоритма есть свои преимущества и недостатки. Структурограммы держат разработчика в более жестких рамках, но программировать на их основе проще, потому что современные алгоритмические языки – это, как правило, языки структурного программирования, снабженные специальными операторами, которые приспособлены к соответствующим конструкциям структурограммы: `begin … end; if … then … else; case … of … end; while … do; repeat … until; for … to(downto) … do` и др.

Этап 4. Программа на алгоритмическом языке состоит из инструкций – операторов. Каждый язык имеет свои свойства и ориентацию на определенные классы задач. Если имеется возможность выбора языка, нужно осмыслить, какой язык больше всего подходит для решения поставленной задачи.

Этап 5. Тестирование и отладка программ представляют собой очень важные составляющие процесса разработки программы. ***Тестирование*** – это процесс выполнения алгоритма с целью определения в нем наличия ошибок, ***отладка*** – процесс локализации и исправления ошибок. Когда результат, полученный программой, совпадает (с учетом погрешности машинного подсчета) с ожидаемым результатом, есть основания полагать, что программа работает корректно. Однако этого недостаточно. Среди начинающих программистов распространено мнение, что если программа успешно откомпилирована и после запуска на выполнение выдает на экран шеренги цифр, то задача решена. На самом деле программу можно считать готовой, когда разработчик смог доказать, что результат работы программы является решением поставленной задачи.

Этап 6. Когда все ошибки исправлены, начинается этап эксплуатации программы. При этом могут пригодиться описания как постановки задачи, так и алгоритма и текста программы.

Более подробно рассмотрим тестирование и отладку программ.

1.2.1. Тестирование программ

Полученный алгоритм, записанный на языке блок-схем или на алгоритмическом языке, нужно проверить на наличие ошибок, используя разные совокупности исходных данных.

Чем раньше будут обнаружены ошибки, тем меньший вред они принесут. Известно, например, что запуск первой американской станции к Венере окончился неудачно в результате единственной ошибки (описки) в программе. Пока не существует теории тестирования, применение которой гарантировало бы выявление всех ошибок.

Рассмотрим несколько практически полезных правил тестирования, применение которых позволит уменьшить количество невыявленных ошибок по сравнению с бессистемной проверкой программ.

1. Нужно готовить не только исходные данные для тестов, но и заранее находить результаты, которые должны получиться, ведь можно легко принять ошибочные результаты за правильные.

2. Очень важно, чтобы в программе осмысливались некорректные данные, поэтому среди тестовых данных подготавливают и некорректные исходные данные.

3. Составление тестов нужно начинать до составления программы. Когда выявлены случаи, которые нужно проверить, тогда и программу проще писать, а учет возможных ошибок в исходных данных сделает программу более устойчивой к этим данным.

4. Составление тестов продолжается параллельно с разработкой программы. Как только в программе пишется условная инструкция, тестовые данные пополняются, чтобы обеспечить проверку работы этой инструкции и для истинного, и для ложного условия. Как только программируется инструкция цикла, нужно обеспечить проверку работы программы в случаях, когда цикл не исполняется ни разу, один или несколько раз.

5. В тестах должны быть проверены крайние случаи. Среди тестовых данных обязательно должны быть значения, предельные между допустимыми и недопустимыми, а также значения, которые по условию задачи должны обрабатываться особенным образом.

6. Нужно тщательно анализировать итоги выполнения тестов, иначе тестирование теряет смысл.

7. Набор тестов нужно сохранять, чтобы при необходимости повторить тестирование (в частности, при исправлении программы).

Например, для задачи решения квадратного уравнения $ax^2 + bx + c = 0$ можно предложить такой тест (он же отображает схему разработки программы):

a	b	c	Корни	
0	0	0	Любое число	Это некорректные исходные данные, так как уравнение не квадратное
0	0	5	Корней нет	
0	2	–4	2	
1	–2	2	Корни комплексные ($D < 0$)	
1	–2	1	1 1 – корни совпали ($D = 0$)	
1	–2	0	0 2 ($D > 0$)	
6	–13	6	2 / 3 3 / 2 ($D > 0$)	
1	$2 \cdot 3{,}4$	$3{,}4^2$	$D = 0$	

Задача. Составить программу, которая вводит три целых числа; если эти числа задают длины сторон треугольника, то определить его вид (правильный, равнобедренный, равносторонний и т.п.).

Решение. Начать следует с составления теста к этой программе, и прежде всего – проверить длины сторон, из которых треугольник составить нельзя, а затем добавить другие случаи:

a	b	c	Треугольник	Соотношение сторон
0	0	0	Не существует	$a = b = c = 0$
2	2	4	Не существует (будет отрезок)	$a + b = c$
1	3	–1	Не существует	$c < 0$
1	1	5	Не существует	$a + b < c$
5	3	3	Равнобедренный	$b = c \neq a$
4	6	4	Равнобедренный	$a = c \neq b$
5	5	7	Равнобедренный	$a = b \neq c$
4	4	4	Правильный	$a = b = c$
9	8	7	Разносторонний	

По этому тесту можно написать фрагмент программы:

```
IF (a >= b + c) OR (b >= a + c) OR (c >= a + b)
   THEN Write ('не существует')
   ELSE IF (a = b) AND (b = c) THEN Write ('правильный')
        ELSE IF (a = b) OR (b = c) OR (a = c)
             THEN Write ('равнобедренный')
             ELSE Write ('разносторонний')
```

1.2.2. Отладка программ

Если тестирование показало, что в программе есть ошибки, следует приступить к ***отладке***: во-первых, проанализировать текст программы на предмет наличия описок; во-вторых, определить локализацию участка, который и приводит к ошибочному результату (некоторые операторы можно выделить как комментарии, сделать их невыполнимыми и продолжить тестирование).

Ошибки могут проявляться по-разному. Например, когда происходит аварийное завершение программы (`Деление на ноль, Overflow, Underflow, Выход за границы массива`), обычно указывается на оператор, выполнение которого привело к аварии. В режиме отладки можно просмотреть значения величин, которые входят в выражение, и проанализировать их. Нужно искать, что же привело к ошибочному результату.

Трудно найти ошибку, когда программа работает «почти правильно» либо, по вашим меркам, «длительно». В этом случае следует тщательно проанализировать алгоритм (эффективный способ — объяснить его кому-нибудь).

Исправление одной части алгоритма не должно приводить к ошибкам в другой. Лучше руководствоваться правилом: когда правок очень много, проще написать программу заново.

Задача. Решить приведенное квадратное уравнение $x^2 + px + q = 0$. (Этапы решения этой задачи сокращены, так как методика решения таких уравнений известна.)

Алгоритм.

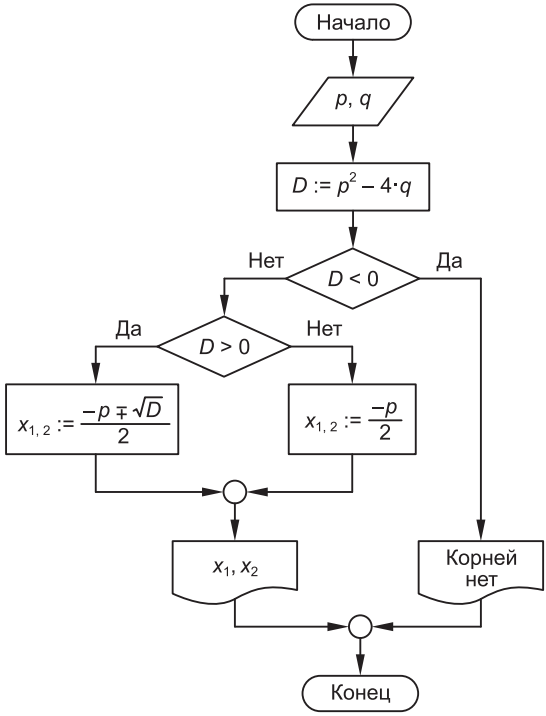
1. Задать коэффициенты p, q.

2. Подсчитать $D = p^2 - 4 \cdot q$.

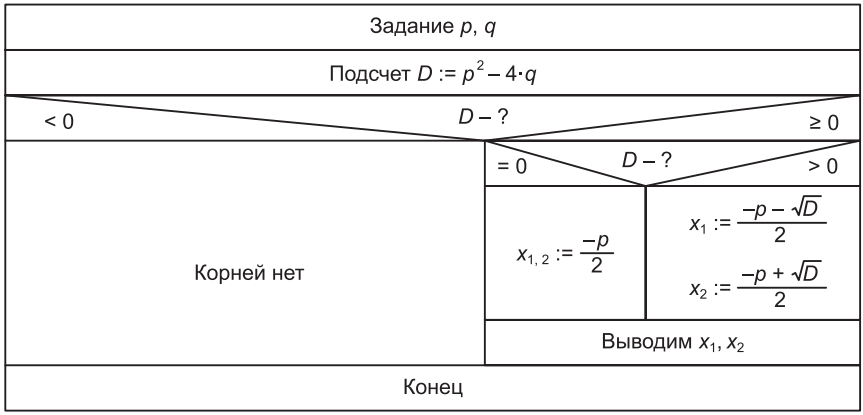
3. Если $D < 0$, то корней нет; если $D > 0$, то $x_{1,2} = \dfrac{-p \mp \sqrt{D}}{2}$, если $D = 0$, то $x_{1,2} = -p/2$.

4. Напечатать результат.

Блок-схема алгоритма:



Структурограмма алгоритма:



Программа, реализующая алгоритм на языке Pascal:

```
PROGRAM Pr1;
CONST p=2; q=1;       {здесь задаем значения коэффициентов}
VAR   x1, x2, d: Real;
BEGIN
d := p * p - 4 * q;
IF d < 0 THEN Writeln ('Нет корней')
   ELSE
   BEGIN
   IF d > 0 THEN
      BEGIN
       d:= sqrt(d);
       x1 := (-p - d) / 2;
       x2 := (-p + d) / 2;
      END
   ELSE
       BEGIN
        x1 := -p / 2;
        x2 := x1;
       END;
   Writeln('x1=', x1, '  x2=', x2)
   END;
END.
```

Задание. Для тестирования программы решите следующие уравнения:

$$x^2 - 5x + 6{,}25 = 0 \qquad ((x - 2{,}5)^2 = 0);$$
$$x^2 - 5{,}2x + 6{,}76 = 0 \qquad ((x - 2{,}6)^2 = 0);$$
$$x^2 - 5{,}4x + 7{,}29 = 0 \qquad ((x - 2{,}7)^2 = 0).$$

С помощью полученной программы подсчитайте корни этих уравнений. Этап тестирования покажет, что в двух примерах уравнений (а это полные квадраты) ответы отличаются от ожидаемых.

Не владея знаниями о представлении вещественных данных в ЭВМ, не всегда можно понять, отчего компьютер неправильно подсчитывает дискриминанты. Непредсказуемое поведение хорошо известных алгоритмов на ЭВМ связано с таким свойством, как патология чисел. Например, 1/3=0,(3). При a:=1/3, b:=3, выполнив действие `a*b`, не всегда можно получить в точности число 1.

Если тестовые уравнения привести к виду уравнений с целыми коэффициентами, то ошибка при подсчете дискриминанта исключена.

СОДЕРЖАНИЕ