12 лекций о том, для чего нужен школьный курс информатики и как его преподавать

А.Г. Кушниренко, Г.В. Лебедев

Книга содержит описание основных понятий, идей и целей школьного курса информатики по учебнику А.Г. Кушниренко, Г.В. Лебедева, Р.А. Свореня «Основы информатики и вычислительной техники» (М.: Просвещение, 1990, 1991, 1993, 1996), а также ряд практических советов и методических приемов по организации и проведению курса.

Особое внимание уделено формированию представлений о структуре и взаимосвязях отдельных частей курса, об их относительной значимости, о месте курса в школьном образовании, о том, что такое информатика вообще и школьная информатика в частности. Обсуждается, какие понятия «настоящей информатики» как науки должны войти в общеобразовательный школьный курс и почему.

Материал пособия будет интересен учителям и методистам, использующим учебник авторов, а также всем тем, кто захочет сравнить разные подходы к преподаванию школьного курса информатики или разработать свой собственный курс, независимо от того, на какой возраст этот курс будет рассчитан.

Оглавление

Предисловие

Основная задача этой книги — систематизация материала учебника, демонстрация основных целей курса и методики его преподавания, демонстрация относительной значимости и сложности тех или иных понятий, формирование у учителя представления об общей структуре и взаимосвязях отдельных частей курса, о месте курса в школьном образовании, а также о том, какую часть «настоящей информатики» как науки представляет курс.

В отличие от методического пособия для учителей [Авербух], материал настоящего пособия весьма неоднороден — одним темам уделено много внимания, другие почти не затронуты, кое-где обсуждаются вопросы «вокруг и около» курса, некоторые темы и тезисы повторяются или рассматриваются несколько раз, на разных уровнях и с разных точек зрения.

История этой книги такова. Много лет мы писали учебники информатики для школы и создавали программное обеспечение для этих учебников на механико-математическом факультете МГУ, в Академии Наук и в объединении «ИнфоМир». Параллельно мы вели курсы переподготовки учителей, регулярно встречались с учителями и методистами, отвечали на их вопросы.

В результате этих встреч мы поняли: учителю и методисту негде найти ответы на важные общие вопросы о том, что такое информатика вообще и школьная информатика в частности, что мы считаем важным в школьном курсе, почему мы отобрали для учебника именно этот материал, почему он излагается в такой последовательности и т. д. Эти вопросы повторялись вновь и вновь на каждой встрече.

В 1991 году в Архангельском институте усовершенствования учителей Г.В. Лебедев прочел цикл лекций, посвященный этим вопросам. Лекции были засняты любительской видеокамерой, и комплект из 6 видеокассет разошелся по России в небольшом числе экземпляров, а Л.Е. Самовольнова изготовила компьютерную стенограмму видеозаписи.

Мы подправили полученный текст, исключили повторы, добавили новый материал, но сохранили стиль стенограммы лекций и изложение от первого лица. Мы добавили также фрагменты лекций А.Г. Кушниренко для учителей информатики, которые он читал в 1993–1995 годах.

Авторы глубоко благодарны Л.Е. Самовольновой, сотрудникам Архангельского ИУУ А.С. Личутину, Ю.Н. Коноплеву, И.А. Захаровой, Н.Н. Нечай и Л.Н. Нестеровой, сотрудникам объединения «ИнфоМир» А.Г. Леонову и Р.Д. Перышковой, а также многим другим не названным здесь людям за помощь и поддержку при создании этой книги.

Введение

Напомним, что изложение идет от первого лица, как в стенограмме лекций

Построение лекций будет следующим:

— сначала я изложу основные идеи, исходя из которых учебник был написан, его главные цели и задачи;

— потом расскажу методику построения курса и постараюсь дать общее представление об учебнике в целом, об относительной важности, значимости его частей и месте и роли понятия «исполнитель»;

— далее я просто изложу учебник — параграф за параграфом, делая основной упор на методические приемы: как подавать материал на уроках, на чем акцентировать внимание, как устроить небольшое представление, трехминутную контрольную и т.д. (нумерация разделов в этой части пособия будет соответствовать номерам параграфов в учебнике);

— затем, думаю, это будет интересно, поговорю «вокруг и около» учебника: организация курса, методпособие, разные полезные книги, программное обеспечение, сравнение с другими учебниками, ответы на вопросы;

— далее еще раз затрону методику, но уже считая, что вы получили достаточно полное представление о курсе. А именно, расскажу про «машинную» и «безмашинную» методики и т. д.;

— потом немного поговорю о месте курса в «большой информатике»: какие области «большой информатики» курс затрагивает, какие не затрагивает, как можно развивать его дальше на факультативе или при углубленном изучении;

— наконец, изложу нашу точку зрения на место курса информатики в школьном цикле (в частности, поделюсь нашими планами переноса курса в 7–8 классы), а также (в послесловии) расскажу о том, как возникли Робот и алгоритмический язык.

Лекция 1.

А. Основные цели, или три «кита» курса

Я, естественно, рассказываю о нашем учебнике, о том понимании информатики, которое заложено в этот учебник. А связь с другими учебниками и его место в «большой информатике» мы обсудим позже.

Курс преследует три основные цели, или, образно говоря, базируется на трех «китах»:

1) первый и основной «кит» называется «Алгоритмический стиль мышления»: ***главная цель курса — развитие алгоритмического стиля мышления***, как самостоятельной культурной ценности, независимо, в каком-то смысле, от компьютеров и всего прочего;

2) второй «кит»: ***курс должен быть «настоящим»***. Слово «настоящий» означает, что в процессе упрощения понятий информатики мы должны не «выплеснуть вместе с водой и ребенка», т.е. упрощать можно лишь до тех пор, пока не теряется содержание, суть дела;

3) третий «кит»: ***курс должен формировать «адекватное представление о современной информационной реальности»***. Это означает некоторую замкнутость, законченность, достаточность набора понятий курса. Другими словами, если второй «кит» запрещает в процессе упрощения переходить к чему-то удобному для изложения, но не имеющему отношения к «настоящей информатике», то третий «кит» требует, чтобы адекватное представление об информатике было тем не менее сформировано, чтобы материала было достаточно и курс содержал необходимый для этого набор понятий, покрывающий сегодняшние реальности.

Забегая вперед, скажу: эти три «кита», как водится, базируются на «черепахе». Наш подход предполагает, что для усвоения курса и учителя и ученики должны много и напряженно работать. Это — не «беллетристический» курс: слушать учителя, читать учебник и отвечать у доски недостаточно. Ученик должен много работать, решать задачи. Мы уверены: только так он сможет что-то воспринять. Можно сказать, что «черепахой» курса является русская пословица: «Без труда не вытащишь и рыбку из пруда».

Для школы это не новость — в математике и в физике все тоже рассчитано на регулярную работу ученика. Посмотрите

на коллег-математиков, вечно они с тетрадками, с контрольными — и иначе никак нельзя.

Сейчас я раскрою смысл слов и понятий «развитие алгоритмического стиля мышления», информатика должна быть «настоящей» и др., а потом, когда буду излагать учебник параграф за параграфом, покажу, как это все в нем воплотилось.

А1. Главная цель курса — развитие алгоритмического стиля мышления

Существует особый алгоритмический стиль мышления, и его развитие есть важная общеобразовательная задача

Итак, первое и основное: алгоритмический стиль мышления.

Мы исходим из того постулата, что человек может мыслить по-разному, что существуют разные стили мышления. Математика и логика развивают математический (или логический) стиль мышления, т.е. умение рационально рассуждать, пользоваться математическими формулами в рассуждениях, из одних утверждений логически выводить другие (теоремы из аксиом и уже известных теорем). Литература и история, наоборот, связаны с какими-то менее рациональными аспектами сопереживания, этики и морали (этот стиль мышления условно можно назвать «гуманитарным»). Ряд школьных предметов — таких, как астрономия или география, — вообще не нацелен на развитие какого-либо стиля мышления (цель этих предметов — изложение знаний в конкретной области, расширение кругозора учащихся).

Первое мое утверждение состоит в том, что есть особый «алгоритмический стиль мышления». Как всегда, когда мы затрагиваем такие фундаментальные понятия, я не могу вам дать точного определения, что такое «алгоритмический», или что такое «логический» стиль мышления. Я просто чуть позже на наглядно-интуитивном уровне покажу, что я под этим понимаю.

А сейчас — следствие из этого утверждения. Если существует (а мы считаем, что существует) такой особый «алгоритмический» стиль мышления человека, то его развитие представляет самостоятельную ценность, так же как и развитие мышления человека вообще. Мы должны развивать все стороны мышления, какие только сможем выделить. И

если в некоторой области удается выделить характерный для нее стиль мышления (умение думать) человека, то его развитие, по моему мнению, должно объявляться самоцелью и внедряться в общее образование, как необходимый элемент общей культуры.

И поэтому учебник весь нацелен на развитие алгоритмического стиля мышления, умения думать алгоритмически, а тем самым и умения думать вообще.

Математика, как курс, в рамках которого преимущественно формируется логический стиль мышления, и информатика, как курс, специально нацеленный на развитие алгоритмического стиля мышления, должны в обязательном порядке входить в общеобразовательные базовые курсы средней школы. Более того, основы алгоритмизации, как и основы логики, в идеале должны закладываться на ранних стадиях обучения, до 12 лет.

Цель курса — развить алгоритмический стиль мышления. Умение обращаться с компьютером или знание конкретных программных систем не входит в непосредственные цели курса

Обратите внимание, я ***не говорю***, что цель курса — научить школьников работать за компьютером, обучить их языкам программирования, современным базам данных или электронным таблицам. Все это — полезные и важные вещи, но наш курс построен совсем по-другому и нацелен совсем на другое — на развитие, в каком-то смысле, головы ученика, его умения думать. Вот это первое и основное.

У нас сейчас много разных учебников и, я думаю, будут появляться все новые и новые, и большинство из них — хорошие. Но когда вы выбираете тот или иной учебник, вы должны, на мой взгляд, исходить, в первую очередь из целей учебника, сопоставляя их с теми целями, которые перед курсом информатики ставите вы.

Если, например, вы считаете, что вам надо, чтобы ученики знали БЕЙСИК, умели работать за компьютерами, пользоваться базами данных или электронными таблицами, то наш учебник вам почти бесполезен.

Наш курс в этом смысле больше похож на предметы физико-математического цикла: на первом плане большое количество задач, умение решать эти задачи, умение думать,

владение соответствующими методами. А компьютер стоит как-то немного в стороне и играет вспомогательную роль.

Я еще и еще раз повторяю: главная и основная цель учебника — развитие алгоритмического стиля мышления как общей культуры ученика. Учебник не нацелен ни на знакомство с языками программирования, ни на знакомство с системами программирования, ни на знакомство с компьютерами как таковыми. Изучение работы на компьютерах, языков и систем программирования не входит в основные цели нашего учебника.

Что такое «алгоритмический стиль мышления»

Я тут повторяю слова «алгоритмический стиль мышления», «алгоритмический стиль мышления», а что это значит — каждый понимает по-своему. И хотя обычно имеется некоторое общее представление, о том, что это такое, я хочу привести пример ситуации, демонстрирующей, что «алгоритмический» стиль мышления вообще существует. Не давая точного определения, я попытаюсь показать, что стоит за словами «алгоритмический стиль мышления», сформулировать проблемы, задачи, для решения которых человек должен мыслить алгоритмически, рассмотреть какие-то рассуждения, характерные для данного стиля мышления, привести методы решений «чисто алгоритмических» задач. В учебнике для этого используется специально сконструированная среда — Робот на клетчатом поле. И я тоже буду использовать ее.

Пусть мы имеем вертикальную металлическую клетчатую стену (в учебнике — «клетчатое поле») с выступающим прямоугольным «препятствием», а на стене — несколько выше препятствия — в одной из клеток находится Робот (R).

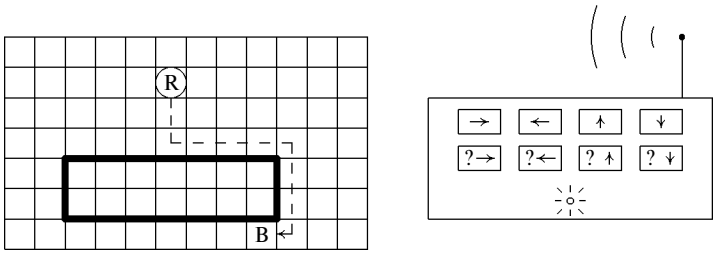


Рис. 1. Робот (R) и пульт управления им

Робот — это машинка с антенной, батарейками, моторами, магнитными присосками и т. п. И пусть у нас в руках имеется пульт радиоуправления Роботом с кнопками: «→», «←», «↑», «↓» (см. рис. 1).

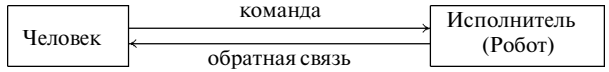
Нажимаем на кнопку «→» — Робот смещается на клетку вправо, нажимаем кнопку «←» — смещается влево , на «↑» — вверх, на «↓» — вниз.

Простейшую задачу управления без каких-либо затруднений решает каждый школьник

Если такую машинку — радиоуправляемого Робота — принести в класс, прикрепить к клетчатой доске и дать ученику пульт управления, то любой ученик в состоянии, глядя на Робота, понажимать на кнопки так, чтобы Робот спустился вниз под препятствие, объехав его. Тут, я повторяю, нет вообще никакой задачи. Даже ребенок, начиная с 5–7 лет, в состоянии это проделать.

Такой стиль взаимодействия с электронными устройствами называется «непосредственным управлением»: я нажимаю на кнопку, смотрю на результат. Нажимаю на другую кнопку, смотрю, что получилось, принимаю решение, какую нажать следующую кнопку. и т. д. — т. е. ***принимаю решения по ходу управления***.

СХЕМА НЕПОСРЕДСТВЕННОГО УПРАВЛЕНИЯ



В этой схеме человек нажимает на кнопки, а «исполнитель» выполняет действия. Глобальный план действий (как и куда двигаться) человек может придумывать по ходу, по мере выполнения команд и прояснения ситуации.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Поскольку вы все здесь учителя и методисты, я сразу скажу, как все это может быть организовано на уроке. Пока у нас нет Робота в «металле», учитель может играть его роль — рисуя Робота и клетчатое поле на доске. Удобно также использовать магнитные доски — на них легко перемещать Робота и не надо постоянно стирать и рисовать его положение.

Итак, на уроке — даже в младших классах — можно поднять самого слабого ученика и сказать: «Иванов,

командуй!». И он без труда, глядя как вы выполняете его команды (перемещаете Робота по доске), продиктует вам последовательность команд «вниз», «вправо» и т. д., по которым Робот обойдет препятствие.

Более сложная задача управления с использованием команд «обратной связи» также не вызывает затруднений

Теперь немного усложним задачу. Будем считать, что Робот — в соседней комнате или вообще далеко от нас (т. е. мы его не видим), а у нас на пульте есть специальные кнопки: «? ←», «? →», «? ↑», «? ↓» и лампочка. Нажимаем на кнопку «? ↓» — Робот анализирует, можно ли сделать шаг вниз, и, если вниз шаг сделать можно, то лампочка на пульте загорается зеленым цветом (если нельзя — то красным). Итак, нажали на кнопку «? ↓», если зажегся зеленый свет, значит снизу свободно, а если красный, значит снизу — препятствие.

Наша задача: не видя ничего, кроме пульта управления, заставить Робота спуститься под препятствие (расстояние от начального положения Робота до препятствия неизвестно). Происходит незначительное усложнение: мы не видим обстановку, мы должны себе ее воображать и принимать решение по миганиям лампочки. Но, хотя и подумав, уже не так мгновенно, как в первом случае, и, возможно, не с первой попытки, но практически все ученики такую задачу решат.

Как? Известно, что Робот стоит где-то выше препятствия, обстановку не видно, размеры препятствия неизвестны. Что надо делать? Надо шагать вниз, пока не дойдем до препятствия, т. е. при каждом шаге проверять (нажимая кнопку «? ↓»), свободно ли еще снизу (зеленый свет) или уже есть препятствие (красный). Как только загорится красный свет (препятствие), надо начать шагать вправо, при каждом шаге проверяя (нажимая кнопку «? ↓»), не кончилось ли препятствие. Потом спускаться вниз, проверяя наличие препятствия слева (кнопка «? ←»). И, наконец, сделать один шаг влево, чтобы оказаться под препятствием. Такие последовательные нажатия на кнопки — даже с анализом невидимой и неизвестной обстановки — доступны любому школьнику.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. На уроке учитель может нарисовать обстановку у себя на листке, никому не показывать и предложить ученикам командовать.

В ответ на «снизу свободно» («? ↓») лучше сразу говорить «да» или «нет» вместо слов «зеленый» и «красный».

Другой вариант — вызвать одного ученика, поставить его спиной к доске, на доске нарисовать обстановку и Робота (чтоб видел весь класс, кроме вызванного ученика), и предложить вызванному ученику командовать, а самому с помощью мела и тряпки исполнять поступающие команды.

Если обход препятствия «втемную» кажется слишком сложным или длинным, то можно рассмотреть только первую часть задачи — «спуститься вниз до препятствия».

На этом же примере можно показать типичные ошибки. Если ученик начинает с команды «вниз», а не с вопроса «снизу свободно?», то в ответ на первую же команду учитель вместо «сделано» может сказать: «отказ — Робот разбился» (и тем самым дать ученикам понять, что проверять обстановку надо ***перед*** перемещением, а не после).

Я повторю, что кто-то, быть может, сначала ошибется (например, уведет Робота вбок от препятствия), но тем не менее, после 2–3 попыток, подавляющее большинство учеников (во всяком случае, начиная с 7-го класса) с такой задачей справится. Это по-прежнему «непосредственное управление»: я нажимаю на кнопки, смотрю на ответ (лампочку), нажимаю на другие кнопки и т. д.

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Конечно, дети не родились с умением решать эти задачи. Просто раньше им приходилось решать много подобных задач: когда они гремели погремушкой, а потом возились в песочнице, а потом запихивали в портфель книжки, учились звонить по телефону и управлять лифтом или радиоприемником. А вот в ситуации, к рассмотрению которой мы сейчас перейдем, — в задаче «программного управления» — жизненный опыт у школьников не накоплен, потому она и оказывается трудной.

Записать или объяснить кому-нибудь алгоритм труднее, чем выполнить работу самому

А вот если теперь школьника попросить: «ЗАПИШИ КАК-НИБУДЬ всю последовательность нажатий на кнопки для обхода препятствия неизвестных размеров, находящегося где-то ниже Робота», то тут-то и выяснится, что значительная часть учеников:

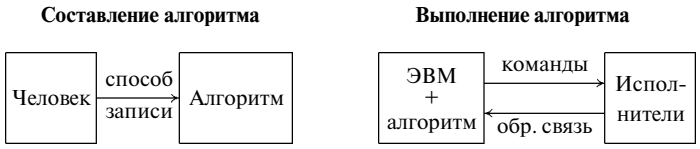
а) прекрасно представляет, на какие кнопки и как надо нажимать, чтобы заставить Робота препятствие обойти, но

б) не в состоянии четко описать (записать) эту последовательность действий.

Нет ничего удивительного в том, что алгоритм легче выполнить, чем записать

Вы можете называть эту запись алгоритмом или программой, либо никак не называть. Вы можете предложить записать это хоть на русском языке, хоть на школьном алгоритмическом, хоть на Бейсике, хоть на языке, придуманном самим школьником, — суть дела это не изменит. А суть в том, что, если раньше человек прямо нажимал на кнопки (схема «непосредственного управления»), то теперь он пишет программу (алгоритм), которая далее будет выполняться без его участия — обычно на ЭВМ, которая «нажимает на кнопки» и командует исполнителем:

СХЕМА ПРОГРАММНОГО УПРАВЛЕНИЯ



Итак, сначала человек выбирает какой-то способ записи, какой-то язык и записывает алгоритм. Затем (это на нашей схеме вообще не отражено) алгоритм как-то попадает к ЭВМ. И, наконец, ЭВМ начинает командовать Исполнителем в соответствии с алгоритмом, полученным от человека.

Почему же алгоритм трудно записать?

Это трудно по трем причинам.

Во-первых, алгоритм придется сразу («наперед») продумать во всех деталях, ничего нельзя отложить на потом — ведь выполнять алгоритм будем уже не мы, а ЭВМ.

Во-вторых, мы должны не только все продумать «наперед» во всех мыслимых вариантах, но и записать это без двусмысленностей и выражений типа «и т. д.».

В-третьих, выполнять алгоритм будет ЭВМ — техническое устройство. Оно не может догадаться, что мы что-то «имели в виду», — все должно быть описано явно, точно, формально и на понятном ЭВМ языке.

Алгоритм — план будущей деятельности, записанный в заранее выбранной формальной системе обозначений (нотации). Составляет алгоритм человек, а выполняет — ЭВМ

Введение ЗАПИСИ будущих действий с помощью какой-то нотации (в нашем учебнике используется школьный алгоритмический язык, в учебнике [Каймин] — Пролог и Бейсик, в учебнике [Гейн] — Бейсик и некоторая разновидность алгоритмического языка) — это и есть переход от непосредственного управления к программированию. Здесь и возникают сложности. Человек может прекрасно представлять себе, на какие кнопки и как надо нажать для получения результата, но иногда не в состоянии ***записать*** эту последовательность команд в виде программы. Те, кто уже преподавал информатику, я думаю, неоднократно сталкивались с подобной ситуацией.

Повторю, что трудности не в форме записи, не в языке. Какую бы нотацию мы не ввели, коль скоро мы от непосредственного управления перешли к программированию, т. е. к записи плана ***будущих*** действий, нам придется предвидеть эти будущие действия во всем многообразии возникающих вариантов. Именно здесь и возникают сложности.

И вот тот стиль мышления, методы и способы, которые необходимы для перехода от непосредственного управления к программному, от умения ***сделать*** к умению ***записать*** алгоритм, я и называю ***алгоритмическим стилем мышления***. Повторю, что это не строгое определение. Я лишь привел пример ситуации, которая показывает, что алгоритмический стиль мышления существует. Его можно и нужно тренировать, развивать, и именно этому посвящен наш учебник.

Исходя из цели развития алгоритмического стиля мышления ясно, что необходимо включать в общеобразовательный курс информатики, а без чего можно обойтись

Теперь еще одно лирическое отступление, связанное, попрежнему, с первым «китом» — алгоритмическим стилем

мышления. Чтобы было понятнее, я специально буду несколько категоричен, буду, скажем так, «перегибать палку».

Во-первых, из цели развития алгоритмического стиля мышления вытекает, что для нас не так важно, есть компьютер или нет. Я хочу это особо подчеркнуть, хотя вовсе не призываю вас преподавать без компьютера. Логика тут простая. Если нет компьютера, то научиться пользоваться им, конечно, нельзя. Но мы ведь такой цели и не ставим. Вы все знаете известный пример про обучение плаванию без воды в бассейне — действительно нельзя научить учеников пользоваться компьютерами, научить их работать с системами, не имея оных.

Но это не относится к развитию алгоритмического стиля мышления. ***Для развития мышления нужна только голова***. Развивать алгоритмический стиль мышления можно без компьютера и даже независимо от того, есть какие-либо компьютеры в мире или нет. Подобно тому, как электрическое сопротивление и закон Ома можно изучать независимо от того, придуманы ли уже электромоторы и телевизоры, курс основ информатики можно изучать независимо от того, придуманы ли уже ЭВМ или еще нет. И зачатки информатики — задачки типа «Волка, козы и капусты» или «Ханойской башни» существовали испокон века. Такие задачки развивали именно алгоритмическую сторону мышления, развивали в меру тех потребностей, которые существовали в обществе. Сейчас с развитием компьютеров эти потребности резко возросли, информатика из «Волка, козы и капусты» превратилась в отдельную научную область с массой своих методов, направлений, подразделений и т. д. И, как видите, дело дошло уже до отдельного курса в школе.

Нельзя научиться решать алгоритмические задачи, изучая устройство компьютеров и языки программирования

Во-вторых, поскольку речь идет о развитии алгоритмического стиля мышления, то компьютеры из предмета, из цели обучения превращаются в средство. Компьютеры нам нужны ***для того, чтобы*** развивать алгоритмический стиль мышления. Обращаю ваше внимание на эту очень важную смену ориентиров, ***компьютеры из цели обучения превращаются в средство обучения***. С ними учить интереснее, с ними учить эффективнее. Школьники больше успевают усвоить, если у них есть компьютеры с соответствующим программным обеспечением, но, тем не менее, это только средство.

Проведем аналогию с математикой. Когда мы решаем математические задачи, их необходимо записывать, то ли с помощью мела и доски, то ли с помощью ручки и бумаги. Но мел и доска, ручка и бумага — это средства записи. Они не имеют отношения к математике как таковой. Если у нас не будет ручки, бумаги, мела, доски и понадобится, к примеру, каменным топором вырубать формулы на деревянных досках, то, ясное дело, задачи придется выбирать попроще и мы сможем решить меньше задач. И вообще нам начнут нравиться задачи, в которых поменьше надо «вырубать формул». И в школьном курсе мы оставим не самое важное из элементарной математики, а самое удобное для «вырубания». Значит, инструмент должен, в каком-то смысле, быть более или менее адекватным специфике предмета, иначе он начинает предмет деформировать.

Но тем не менее и ручка, и мел, и топор, да и компьютеры в нашем курсе — только средства. Это означает, что как нельзя научиться решать математические задачи, изучая устройство ручки и бумаги, так же точно нельзя научиться решать алгоритмические задачи, изучая устройство компьютеров и языки программирования.

Это чрезвычайно важная вещь, и я хочу, чтобы вы ее поняли. Это ключ ко всему построению курса и учебника.

ЭВМ и языки — средства для проведения курса информатики, подобно ручке и бумаге в математике. Если, скажем, ручка плохо пишет и мы с ней много возимся или прежде чем написать слово, нам надо ее десять минут заряжать, то мы будем больше времени уделять ручке, и меньше решать с ее помощью задач. Точно так же, если наши компьютеры, программные средства и т. д. требуют к себе особого внимания, изучения, если у нас нет возможности просто пользоваться ими для решения задач, а надо заниматься самими ЭВМ, самими языками, то те задачи, которые мы будем решать, естественно упростятся, — мы не сможем уделить им достаточно времени. Я на этом потом еще остановлюсь, потому что именно этим объясняется появление Робота и всей этой кажущейся игрушечности в начале курса.

И ещё раз скажу вам: ЭВМ и языки — только средства

Итак, повторю, ЭВМ и языки — только средства, их изучение не дает почти ничего для развития алгоритмического стиля мышления. Может быть расширяется кругозор, так же как при изучении других достижений техники, но в плане развития структур мышления в человеческой голове не добавляется практически ничего.

Но и средства тоже важны

С другой стороны, совсем без средств обойтись нельзя. Как в математике мы можем писать ручкой, карандашом, фломастером, но чем-то мы писать должны, так и в информатике выбрать какие-то средства (ЭВМ, язык) мы должны.

И, что очень важно, ***выбор средств оказывает влияние на сложность решаемых нами задач И НА ТО, ЧТО МЫ ИЗУЧАЕМ***. Если мы выберем неадекватные средства — плохую ручку, ненадежную ЭВМ, сложный язык программирования, неудобную программную систему — то волей-неволей мы будем вынуждены потратить часть времени на ремонт и изучение собственно ЭВМ, собственно языков, собственно ручки, собственно бумаги. И тем самым эта часть времени будет изъята из курса, т. е. пропадет время, которое можно было бы потратить на решение задач, на другие содержательные вещи.

Первый «кит»— выводы

Вот, пожалуй, и все о первом «ките». Подведем итоги:

1) существует особый «алгоритмический» стиль мышления;

2) главная цель нашего курса — развитие этого стиля мышления, как самостоятельной культурной ценности;

3) ЭВМ и системы программирования в нашем курсе не цели, а всего лишь средства обучения, используемые для развития алгоритмического стиля мышления;

4) выбор этих средств очень важен — удобные средства дают нам возможность продуктивно развивать алгоритмический стиль мышления, тратя минимум времени и сил на технические детали.

А2. Курс должен быть «настоящим»

Второй «кит» — курс должен быть «настоящим». Это означает, что нельзя ради простоты обучения подменять понятия.

Теоремы, доказательства и многие понятия школьной математики возможно и сложны для преподавания, но из этого не следует, что их можно заменить какими-нибудь легкими придуманными понятиями, отсутствующими в «настоящей математике» (в науке). Если в иностранном языке какие-то конструкции слишком сложны для преподавания в школе, то их лучше не преподавать вообще, а не заменять придуманными, простыми, но не существующими в языке суррогатами.

Точно так же в школьной информатике, мы, конечно, должны упростить, адаптировать набор понятий «настоящей информатики» для школьников, но при этом ни в коем случае нельзя производить подмену понятий. Учить надо настоящему, либо — если что-то слишком сложно для школьников — не учить вовсе. Но нельзя учить придуманному, тому, чего в жизни нет.

Я могу привести несколько примеров. В одном из пособий по информатике всерьез изучались такие надуманные «свойства» информации, как ее «своевременность» или «ясность». Эти придуманные (точнее, «бытовые») понятия легко объяснять, о них массу всего можно сочинить, о них легко говорить. Но ничего подобного в «настоящем» курсе информатики быть не должно. «Настоящий» означает, что следует взять науку под названием «Информатика» (англ. «Computer science») и выбрать из нее то, что можно и нужно адаптировать для школьников. В школьном курсе можно ограничиться небольшим набором понятий, но нельзя их подменять легкими для изучения, но «ненастоящими» понятиями, не существующими ни в теоретической, ни в практической информатике.

Еще один пример «ненастоящести» — неформальные алгоритмы из первых школьных учебников [Ершов] типа алгоритма перехода улицы или алгоритма заваривания кофе. Беда в том, что при этом заранее четко не задавался набор базовых команд, которыми можно пользоваться и которые дальше уже детализировать не надо, что сразу переводит задачу из информатики в художественно-психологическую область — никаких формальных критериев для проверки и оценки ответа не существует, ответ просто должен совпадать с мне-

нием учителя. Вообще, изложение понятия исполнителя на неформально-художественном уровне — одна из основных ошибок первых учебников. О системе команд исполнителей в этих учебниках упоминалось, но ни одна система команд не приводилась. Алгоритмы были либо изложением чего-то из математики, либо чисто художественными, и потому легко заменимыми на любые другие. (Вспомните мучения учителей, пытавшихся объяснить «неправильность», например, алгоритма заварки кофе «пойти в бар и заказать чашечку» или алгоритма выхода из кинотеатра «идти за всеми»).

Возвращаясь к «настоящести», хочу еще раз подчеркнуть, что «настоящесть» состоит в том, что мы не должны преподавать (как бы приятно или легко это ни было) понятия, не имеющие отношения к реальности, придуманные ради легкости их изложения; мы не должны решать придуманные задачи, которые не имеют отношения к информатике. Мы не должны «вместе с водой выплескивать и ребенка». Можно упрощать под школьный возраст только до тех пор, пока не теряется суть. То, что нельзя преподать адекватно, не нужно преподавать вообще. Лучше не преподавать ничего, чем преподавать ерунду, не имеющую отношения к действительности.

А3. Курс должен формировать адекватное представление о современной информационной реальности

Третий кит — это другая сторона медали «курс должен быть настоящим». Но если второй кит запрещает придумывать новые понятия ради легкости изложения, то третий требует, чтобы некоторое минимальное множество понятий и методов, достаточное для формирования представления о современной информационной реальности и современной информатике, было изложено и доведено до учеников, как бы сложно это ни оказалось.

Развивать тот или иной стиль мышления не обязательно только научными методами

Математический (логический) стиль мышления можно развивать не с помощью современных курсов математики, а на кубиках, головоломках, занимательных задачках и т. д. Точно так же алгоритмический стиль мышления можно начинать развивать с задачек типа «Волк, коза и капуста», «Ханойская башня», или с игры в «Черный ящик» (см. стр. 61), где, наблюдая, как информация поступает на «вход» Черного ящика, и видя, что получается на «выходе», нужно угадать правило (алгоритм) преобразования информации.

Но отдельными занимательными задачками, развивающими алгоритмический стиль мышления, ограничиться нельзя. Курс информатики должен охватить некоторый замкнутый, самосогласованный набор понятий, взятый из «настоящей информатики». Чтобы лучше объяснить, что я имею в виду, снова сошлюсь на устоявшийся школьный курс — курс математики.

Школьный курс математики — часть настоящей математики. Школьный курс информатики должен быть частью настоящей информатики

Современный школьный курс математики, формируя представление о современной математической культуре, не может ограничиться только головоломками и развитием математической сообразительности, а вынужден охватывать формулы алгебры, аксиомы геометрии, теоремы, леммы, доказательства и др., составляющие вместе замкнутый курс

«элементарной математики». Этот курс, конечно, намного проще, чем все здание «большой» математики или курс «высшей математики» в ВУЗе, но, тем не менее, он «настоящий» — и академик-математик, и инженер-конструктор, и восьмиклассник решают квадратное уравнение по одной и той же формуле и используют одну и ту же математическую символику.

Точно так же и курс школьной информатики должен быть частью информатики «настоящей». А это значит, что мы не можем в школьном курсе остановиться на уровне задач типа «Волк, коза и капуста». Такие задачи к «большой информатике» имеют примерно такое же отношение, как головоломки к «настоящей математике». Да, есть интересные занимательные задачки, решение которых развивает алгоритмический стиль мышления, их, конечно же, надо использовать как для повышения интереса учащихся, так и просто для разнообразия. Но тем не менее, если мы говорим про курс информатики в школе, то он должен относиться к «настоящей информатике» вне школы, примерно так же, как математика к «математике», физика к «физике» и т. д.

«Настоящий» школьный курс должен знакомить с необходимым — пусть минимальным — набором фундаментальных понятий информатики сегодняшнего дня. Как это ни удивительно, но такой набор определен вполне однозначно и невелик — всего четыре понятия. Сейчас я их сформулирую.

Четыре основных понятия школьного курса информатики

В плане развития алгоритмического стиля мышления фундаментальных базовых понятий информатики всего четыре:

(1) команда (оператор, управляющая конструкция), и в первую очередь — команда повторения (цикл);
(2) величина (переменная, объект), и в первую очередь — табличная величина (массив);
(3) вспомогательный алгоритм (подпрограмма, процедура) с параметрами;
(4) «информационная модель исполнителя» (как это изложено в 3 главе нашего учебника), или просто «исполнитель» (пакет, модуль, объект, экземпляр класса).

Конечно, когда очередной лектор (сейчас — я), выписывает такой набор и говорит, что вот это — 4 основных понятия информатики, у вас немедленно должна появиться мысль, что другой лектор написал бы 7 других «основных понятий», третий добавил бы еще пару и т. д. Вообще возникает вопрос о роли личности лектора в выборе этих понятий. Так вот, понятия, которые я выписал, не случайны и выбраны отнюдь не принципу «нравятся они мне или не нравятся». Они являются не исходной точкой, а результатом некоторого рассуждения, доказательства. И я сейчас приведу набросок этого рассуждения — обоснования, что именно эти четыре понятия являются фундаментом современного алгоритмического стиля мышления.

От действий над объектами к их записи — командам и величинам

Мы можем начать с устройства ЭВМ. Как вы знаете, машина состоит, в основном, из процессора и памяти. Процессор выполняет действия, память хранит информацию об объектах и величинах, с которыми работает процессор. Все остальное (принтер, экран, дисковод и пр.) отличается от Робота только системой команд. Устройству дается команда, оно ее выполняет. А вот процессор с памятью — это особые, фундаментальные компоненты ЭВМ. Им соответствуют два фундаментальных свойства ЭВМ — умение выполнять действия и умение хранить информацию в памяти машины.

Можно, впрочем, не апеллируя к устройству ЭВМ, начать сразу с содержательного уровня действий и объектов. Я думаю, любой специалист по информатике согласится с тем, что объекты и совершаемые над ними действия — две самые фундаментальные сущности управления.

Мы, однако, занимаемся не просто непосредственным управлением, а информатикой. Поэтому объекты и совершаемые над ними действия нам надо как-то ***записывать*** в виде программ (алгоритмов). В школьном языке для записи действий используются команды (в других языках они называются операторами, или управляющими конструкциями), а для записи объектов — величины (в других языках называемые переменными, или объектами):

Действия → Команды
Объекты → Величины

Запись большого количества действий и работа с большими объемами информации

Возникновение информатики как науки тесно связано с появлением ЭВМ по той причине, что ЭВМ умеет быстро выполнять действия и хранить много информации. Соответствующие характеристики ЭВМ называются «быстродействием» и «объемом памяти». Чтобы задействовать эти возможности ЭВМ, необходимо научиться ***записывать*** выполнение большого количества действий над большими объемами информации. Соответствующие методы компактной (короткой) записи длинных последовательностей действий и компактной записи больших объемов информации и стали зачатками информатики. Запись простого действия в виде команды или запись объекта в виде простой величины обычно не вызывает труда. Информатика как наука начинает работать, когда нужно записать длинную последовательностей действий или обработать большой объем информации. Для этого в языках программирования используются конструкции циклов (короткая запись большого количества действий) и величины, состоящие из многих элементарных объектов, прежде всего массивы (в школьном языке названные «табличными величинами»):

Действия → Команды → Циклы
Объекты → Величины → Табличные величины

Цикл — это команда, позволяющая коротко ***записать*** длинную последовательность действий. Табличная величина — это кратко записываемая величина, которая дает возможность хранить много информации.

«Циклы» и табличные величины соответствуют устройству современных ЭВМ.

Отвлекаясь в сторону, давайте рассмотрим еще раз взаимосвязь этих понятий с устройством современных ЭВМ.

Чтобы задействовать «быстродействие», надо уметь приказать ЭВМ выполнить громадное количество действий, скажем миллион, или миллиард. Если мы начнем эти действия выписывать: первое, второе, третье, ... — жизни не хватит. Какой прок в быстродействии миллион операций в секунду, если приказ на каждую операцию мы будем записывать минуту! Другое дело, если мы напишем небольшую последовательность действий, скажем десять или сто команд, но прикажем компьютеру выполнить ее много раз: миллион, или миллиард. Тут мы сможем задействовать быстродействие в полной мере. Такая запись многократного выполнения заданной последовательности действий и называется «циклом».

Теперь давайте поговорим о памяти ЭВМ. Память состоит из отдельных элементов, имеющих свою внутреннюю нумерацию. Но записываем мы алгоритм на каком-то языке, а в языках имеем дело с отдельными порциями информации. Эти порции называются переменными, объектами, или, с легкой руки А.П. Ершова, величинами (по аналогии с математикой и физикой). Чтобы работать с величинами и отличать одни от других, их нужно снабдить именами. В памяти современной ЭВМ могут храниться значения миллионов числовых величин. Снова жизни не хватит, чтобы придумать для каждой из них имя. Таким путем задействовать большой объем памяти ЭВМ не удастся. На выручку приходят табличные величины (массивы). Такая величина имеет только одно имя и много элементов, занумерованных целыми числами. Поскольку элементы памяти ЭВМ также имеют свою внутреннюю нумерацию, понятие табличной величины (массива) хорошо соответствует структуре памяти современных ЭВМ и есть во всех современных языках программирования.

Замечу, что сами методы и понятия компактной (короткой) записи большого количества действий или компактного обозначения больших объемов информации носят фундаментальный характер и могут изучаться безотносительно ЭВМ, хотя развитие этих методов было вызвано появлением ЭВМ и их указанными выше особенностями.

Итак, мы разобрали, почему в наш список вошли первые два понятия: (1) команды и особенно циклы, и (2) величины и особенно табличные величины. В принципе, этими двумя понятиями можно было бы ограничиться, и получился бы вполне замкнутый курс, я бы сказал нулевого уровня. Кстати, Бейсик идейно близок именно этому уровню знаний.

Можно ли ограничиться понятиями «цикл» и «табличная величина»

А нужны ли остальные понятия? Ответ на этот вопрос не такой простой. Набор из первых двух понятий, простой и замкнутый, хорошо отвечает структуре ЭВМ. Есть язык программирования — Бейсик, прекрасно поддерживающий эти два понятия и отвечающий требованиям 60-х годов. Но с точки зрения теории и практики информатики сегодняшнего дня этот набор из двух понятий мал, алгоритмический стиль мышления отражает весьма бедно и не охватывает фундаментальные для этого стиля понятия «структуризации».

Способность структуризации — фундаментальное свойство человеческого мышления

Лучше всего мне, наверное, снова прибегнуть к аналогиям. Представьте себе математику, где каждое утверждение доказывается, исходя из аксиом, от начала до конца. Любое доказательство превращается в огромный текст, и понятно, что так работать нельзя. Точнее, существует естественный порог сложности задач, которые можно решить таким образом.

То же произойдет, если не вводить в русский язык новые слова. Попробуйте представить себе человека, ничего никогда не слышавшего о метро, и попробуйте описать все то знание (поезда, эскалаторы, под землей, но не всегда и т. д.), которое в вашей голове связано с этим словом. А теперь представьте себе, что вы в обычном разговоре ***всякий раз*** вместо слова «метро» используете это описание из сотни или тысячи слов!

Для всех случаев, когда встречается проблема такого разрастания описаний, человечество придумало способы сбора, накопления и компактификации знаний. Если в русском языке появляется интенсивно используемое понятие, то его называют новым словом. Если в математических формулах начинает повторяться какой-то громоздкий фрагмент, то для него вводится новое короткое обозначение. Если в математике начинает повторяться одно и то же рассуждение, то оно формулируется в виде теоремы. Теорема получает имя, на которое отныне достаточно просто сослаться, не повторяя доказательство.

Все это мы будем называть структуризацией. Структуризация в языке позволяет коротко назвать, обозначить то, что раньше требовало длинного описания. Структуризация в математике дает возможность доказывать утверждения, исходя не только из аксиом, но и из ранее доказанных утверждений, т. е. задействовать знания, полученные ранее.

Способность задействовать сделанное раньше (может быть даже предшествующими поколениями) — одно из фундаментальных свойств человеческой культуры, и оно проявляется почти во всех областях человеческой деятельности.

Поэтому для решения современных задач, для развития алгоритмического стиля мышления в его современном понимании необходимо овладеть понятиями и методами структуризации в информатике.

Способы структуризации в информатике опираются на понятие вспомогательного алгоритма

Если в информатике мы ограничимся циклами и величинами, то будем вынуждены любую задачу доводить до базовых конструкций языка программирования и писать любой алгоритм, любую программу одним куском. Как и в математике без теорем, в информатике без каких-то способов структуризации серьезно работать не удастся. И в информатике придуманы такие способы, позволяющие использовать ранее сделанные вещи как целые, как готовые.

Для действий таким структурирующим понятием является понятие «вспомогательного алгоритма», или «подпрограммы» с параметрами. Мы можем некоторую последовательность действий записать отдельно, сказать, что это — вспомогательный алгоритм (подпрограмма), и потом им пользоваться. Здесь полная аналогия с леммами и теоремами в математике.

Поскольку вспомогательный алгоритм, или подпрограмма, — понятие к настоящему моменту широко известное, я думаю, здесь у вас никаких особенных вопросов возникать не должно.

Действия $\rightarrow$ Команды (Циклы) $\rightarrow$ Всп. алгоритмы
Объекты $\rightarrow$ Величины (Таблицы) $\rightarrow$?

Кстати, язык Паскаль, который в свое время придумал Н. Вирт, содержит именно эту тройку понятий, т. е. действия можно разбить на вспомогательные алгоритмы (подпрограммы, процедуры). С объектами в Паскале хуже — их трудно разделять по назначению. Мы не в состоянии записать, что вот эти объекты будут предназначены для того-то и того-то и связаны с такими-то действиями; а другие объекты предназначены для другого и связаны с другими действиями.

Уровень Паскаля — это уровень информатики 70-х годов: вспомогательными алгоритмами (подпрограммами) уже пользовались, а аналогичные средства для работы с объектами еще не появились.

Структурировать в информатике нужно не только действия, алгоритмы, но и саму информацию, объекты

Я попробую пояснить структуризацию объектов на житейском примере. Допустим, вы решили сделать ремонт, до-

ма или в школе, неважно. Нужны материалы, инструменты, рабочие. Первый подход: нанять рабочих, самому закупить материалы и инструменты и отвечать за них. Отдавая любой приказ любому рабочему, выдавать ему инструменты и материалы, а по окончании работы забирать инструменты и остатки материалов. К чему приводит такой подход? Все материалы и инструменты — ваша забота, вы за них отвечаете, вы ими постоянно манипулируете, у вас полон рот хлопот. А рабочий выполнил работу, все сдал и без хлопот пошел домой. В каком-то смысле это похоже на использование вспомогательных алгоритмов: при вызове алгоритма ему передаются аргументы, по завершении работы от него забираются результаты, и после окончания выполнения все, что было связано с алгоритмом, стирается из памяти ЭВМ («рабочий пошел домой»). Второй подход — нанять сразу целую бригаду со всеми инструментами и материалами, а самому с ними дела не иметь. Рабочие внутри бригады сами по мере надобности будут передавать друг другу инструменты и материалы, сами за них отвечать и разбираться с тем, что им необходимо. Бригада оказывается коллективным Исполнителем ремонта. И если вы наймете две бригады для разных целей (например, маляров и сантехников), то инструменты и материалы каждой из них будет хранить она сама и между собой они не перепутаются.

В информатике второй подход обеспечивает понятие исполнителя — хранителя какого-то набора объектов. Исполнитель и есть недостающее в классическом Паскале средство структуризации.

Все вновь создаваемые языки программирования имеют средства структуризации объектов, а все классические языки такими средствами дополняются

Насколько мне известно, первый язык, в котором появились способы такой «структуризации» объектов, это язык Simula-67. Как следует из названия языка, предназначен он был для информационного моделирования («симулирования») различных процессов. Примерами процессов могут служить ремонт школы, воздушный бой или работа компании по продаже авиабилетов. Позже подобные конструкции для структуризации объектов появились практически во всех других языках программирования.

Сейчас средства структуризации объектов есть во всех современных языках программирования, но называются они везде по-разному. Здесь пока нет единой терминологии. Наиболее распространенный термин, используемый в научных журналах и рекламных объявлениях, — «объектно-ориентированное программирование». Термин подчеркивает, что первичны, вообще говоря, не действия, не алгоритмы, а объекты, с которыми алгоритмы работают. А на практике важны и алгоритмы, и объекты: надо уметь структуризовать вместе и информацию, и обрабатывающие ее алгоритмы. Для этого языку нужны средства выделения некоторого набора объектов (величин в нашей терминологии) и действий, которые над ними производятся. И эти средства должны позволить все это назвать одним именем и впоследствии использовать целиком, подобно бригаде маляров при ремонте.

В нашем учебнике соответствующее понятие называется «информационной моделью исполнителя», или просто «исполнителем», и вводится в третьей главе. В других языках это понятие называется по-разному, скажем, в языке Ада — «пакетом» (package), в языке Модула — «модулем», в C++ и Simula-67 — «объектом» и «экземпляром класса»; классический Паскаль расширен конструкцией Unit, в классическом C для этих целей используется свойство локализации объектов в файлах. Достаточно широко распространено слово «модуль», хотя это слово применяется и для других целей. Например, в книге [Майерс] дается классификация около 15 типов разных модулей и обсуждаемое нами понятие называется «информационно-прочным» модулем.

Здесь пока нет единой общепринятой терминологии. Важно то, что во всех современных языках теперь такая конструкция есть, ибо она фундаментальная и без нее не обойтись. Поскольку, однако, единой терминологии до сих пор нет, мы с вами, следуя учебнику, будем говорить «информационная модель исполнителя», или просто «исполнитель».

Фундаментальное понятие информационной модели в школьных курсах информатики, в том числе и в нашем курсе, отражено недостаточно

Я тут столько языков программирования назвал, что у вас могло создаться ощущение, что понятие информационной модели быть может важное и нужное, но вовсе не в общеобразовательном курсе информатики, а в профессиональ-

ном программировании. Это не так. Умение строить информационные модели — одна из базисных компонент алгоритмического стиля мышления. Задачи по придумыванию информационных моделей увлекательны и поучительны не меньше, а может даже и больше, чем задачи на составление алгоритмов. Такие задачи должны занять подобающее место в школьном курсе.

Подробнее ко всему этому мы подойдем в конце, когда я буду рассказывать про третью главу. Сейчас я хочу вас вернуть к четырем фундаментальным понятиям информатики:

1) Команды (Циклы) → 3) Вспомогательные алгоритмы
2) Величины (Таблицы) → 4) Исполнители

Давайте взглянем на этот список еще раз. Первые два понятия отражают методы записи действий и объектов, и, в особенности, записи большого количества действий и большого количества объектов. Вторые два понятия отражают фундаментальные приемы структуризации, которые человечество выработало за последние три десятка, а отчасти и за последние три тысячи лет. Все четыре понятия просты, конкретны, доступны школьникам, могут быть поняты и освоены в процессе решения задач, и все вместе образуют замкнутое целое, фундамент. На нем можно развивать и внутренние способности человека к алгоритмическому мышлению, и понимание реальностей окружающего нас мира.

Итак, эти четыре понятия выписаны не случайно. Не просто потому, что я решил, что они важнее других. За этим выбором стоят фундаментальные свойства действий и величин и фундаментальные способы их ***записи*** — их организации и структуризации, которые к настоящему моменту выработало человечество.

Нужно ли выходить за пределы этих понятий

Если эти четыре понятия освоены, то, образно говоря, мы забрались на некое «плато» современной информационной культуры. Карабкались, карабкались вверх по горным склонам, наконец добрались до плато, а дальше можно идти во многих разных направлениях. (Если есть потребность, желание, время.) На этом «плато информатики» протоптано много тропинок, проложено много дорог. Одна дорога ведет к развитию способов конструирования структур данных:

стеки, деки, графы, деревья и т. д. Другая — к конструированию новых языков программирования. Третья — к решению классов прикладных задач, например задач трехмерной графики. Этого всего в школьном курсе может и не быть, а вот базовые понятия в нем быть должны. Именно поэтому я говорю, что эти четыре понятия обязаны быть в школьном ОБЩЕОБРАЗОВАТЕЛЬНОМ курсе.

Тут я должен сделать одну оговорку — я говорил про плато информатики в так называемой «процедурной традиции» информатики, «процедурной традиции программирования», где важны действия и объекты и наше умение их организовывать и записывать, которая и является выражением понятия «алгоритмический стиль мышления». Есть и другие традиции — «функциональная» и «логическая», которые связаны скорее с математическим (логическим) стилем мышления и про которые я расскажу позже (см. с. 328).

Но в процедурной традиции, там, где алгоритмический стиль мышления является ключевым, именно четыре указанных выше понятия образуют то культурное «плато», начиная с которого можно решить любую задачу, изучить любой язык программирования и т. д. Общее плато, которое обязано быть в базовой общеобразовательной подготовке любого школьника.

Одним развитием алгоритмического стиля мышления не обойтись — нужно сформировать представление о том, как ЭВМ, потоки информации, алгоритмы вписаны в жизнь современного общества

«Адекватное представление» конечно включает в себя еще и представление о месте информации и ЭВМ в нашем обществе, а также о том, что можно и что нельзя, как можно и как нельзя делать с помощью компьютеров, где их разумнее применять, а где — обойтись без них и т. д.

Другими словами, мы должны сформировать у школьника некоторое представление обо всей «компьютерной» стороне жизни. Пусть упрощенное, неполное, но отвечающее реальности, адекватное представление об информационной реальности. Здесь уже другая задача, отличная от развития алгоритмического стиля мышления.

Практически это означает, что, объясняя какие-то применения ЭВМ, нужно описать их не только внешне, но и изнутри. Всякий раз, когда мы объясняем, что компьюте-

ры используются в информационных системах, для продажи билетов, или еще где-то, мы должны не только сказать, что, нажав на клавиши, можно увидеть на экране наличие свободных мест (это школьники смогут узнать и при покупке билетов), но и объяснить, ***как*** компьютер используется, какие методы обработки, представления информации при этом применяются, почему компьютер что-то может делать, а что-то не может, какова здесь информационная модель, какие стороны жизни она отражает, а какие нет, какие алгоритмы используются для переработки информации в этой модели и т. д.

После окончания курса школьники, как мне кажется, должны себе представлять, что сделать систему для продажи билетов — хотя и сложная, но в общем понятная задача, а отличить по фотографии кошку от собаки с помощью компьютера — задача не только крайне сложная, но и не очень понятная. Они должны представлять себе, что и как происходит внутри компьютера, и каковы сложности этих процессов.

Наконец и совсем простыми вещами пренебрегать нельзя. Всякий сегодня знает, что 20–30 килограмм груза легко перевезти на легковом автомобиле, для 2–3 тонн понадобится уже грузовик, а для 200–300 тонн придется придумывать что-то особое. Точно так же и в информатике нужно воспитать интуитивное ощущение «тяжести» тех или иных задач и «грузоподъемности» тех или иных моделей компьютеров.

Примеры неадекватности и адекватности

Я здесь люблю приводить два примера. Один я заимствовал у Сергея Попова — это игротехник, организатор деловых игр, в том числе по Байкалу. Он однажды объяснял, как в России начали вводить рынок. Мы, сказал он, очень похожи на папуасов, которые увидели, что из прилетевшего самолета выгружают продукты. «Ага»,— сказали папуасы,— «продукты выгружают из самолета», сделали самолет из коры и листьев, сели вокруг него и стали ждать появления продуктов. Это пример неадекватного представления об окружающей реальности.

Другая любимая мною аналогия состоит в следующем. После школьного курса физики все мы имеем некоторую физическую картину мира в голове. Мы, может быть, уже не помним ни формул, ни чему нас учили, но каждый знает, что между выключателем и лампочкой есть провода, даже если никаких проводов не видно. Всем известно, что когда щелкаешь выключателем и зажигаются лампочки — это не отдельно выключатель и отдельно лампочки; что есть провода, электростанции, есть провода к электростанциям и т. д. И каждый знает, что солнце таким образом включить/выключить нельзя. Также более или менее всем понятно, что если из выключателя торчат провода, то лучше их не трогать, потому что мы знаем что-то о токе и напряжении. Мы знаем не только о том, что видно, но кое-что о содержании, о физических величинах, процессах и пр. У нас есть — обычно достаточно адекватная — физическая картина мира.

Адекватную информационную картину мира подобного рода и должна сформировать информатика. Школьники должны представлять себе, какие примерно процессы протекают в ЭВМ, когда она что-то делает. Должны знать, что существует программное обеспечение, в котором могут быть ошибки, и т. д. Как и в примере с выключателем и лампочкой школьник должен — пусть примерно — представлять себе информационную картину миру. Вот мы пришли в класс и включили компьютер — что при этом происходит, почему появляется Бейсик или алгоритмический язык, почему мы можем набрать алгоритм и как он выполняется, что такое компьютерный «вирус» и как он работает, что вообще в этой машине происходит? Только после формирования такого рода представления, на мой взгляд, возможна демистификация ЭВМ, сознательное использование ЭВМ, как обычного средства производства, отношение к ЭВМ, как к нормальному орудию труда, а не как к таинственному и загадочному устройству.

Завершая эту часть, хочу еще раз повторить, что ЭВМ — самое обычное орудие труда, как ручка, утюг, молоток и пр. И в задачу третьего «кита» входит формирование общей картины мира с учетом наличия в этом мире ЭВМ.

В школьном курсе информатике должны быть представлены основные понятия и методы информатизации и алгоритмизации

Ну и поскольку я затронул провода, лампочки и электричество, то, чтобы потом уже не повторяться, скажу еще одну вещь. Еще совсем недавно бушевали страсти о том, чему же мы должны учить школьников в курсе информатики. Одна

из точек зрения звучала так: «Зачем учить программированию, давайте учить пользоваться готовыми системами, они же не будут профессиональными программистами» и т. д.

Здесь, как мне кажется, очень полезными являются аналогии. Давайте взглянем с этой точки зрения на такой традиционный школьный предмет, как физика. В школьном курсе физики изучают электрическое сопротивление, ток, законы Ома и другие вещи, в общем-то не имеющие отношения к включению и выключению лампочек. И именно они формируют общую физическую картину мира. Конечно, многое потом забывается — формулы, законы, даже понятия. Конечно, лишь единицы становятся электриками, а большинство только включает и выключает свет. Тем не менее, никому и в голову не приходит предложить исключить это содержание из курса физики. Даже в страшном сне не приснится курс физики, в котором на первом занятии школьники учатся включать свет, на втором — выключать, на третьем — выворачивать лампочки и т. д. (хотя преподавать такой курс сможет кто угодно и без всякой подготовки).

Также и школьная информатика должна давать школьникам нечто большее, чем просто навыки, которые к тому же устареют еще до начала их реального применения. Использовать готовые программные системы, базы данных, электронные таблицы и системы коммуникаций каждый, если понадобиться, научится столь же быстро, как включать и выключать свет.

На наш взгляд, школьные курсы физики, информатики и других дисциплин призваны закладывать основы общей культуры. В этом смысле информатика должна учить основным методам информатизации и алгоритмизации, тому, что принято в современной «большой» информатике, в науке. Именно это должно составлять базу школьного курса, а вовсе не умение нажимать на кнопки.

Лекция 2. Методика и структура курса

B. Методика построения курса

Наш курс и, соответственно, учебник построены на трех глобальных методических принципах:

1) все познается через работу («черепаха» курса);
2) проблемный подход;
3) выделение алгоритмической сложности в «чистом виде».

Конечно, кроме этих трех общих принципов существует много частных методических приемов, полезных при изучении какой-то конкретной темы или отдельного урока, но сейчас мы их затрагивать не будем (эти приемы я покажу потом, когда буду идти по учебнику параграф за параграфом).

А вот указанные три общих принципа пронизывают весь учебник и во многом определяют структуру курса. Вместе с основными целями («китами») курса они определяют, почему учебник и курс построены именно так, а не иначе, какие понятия и как именно в этот курс включены.

В1. «Черепаха» курса — все познается через работу

Весь наш учебник базируется на идее, что ничего нельзя выучить и понять, если не работать. Можно как угодно красиво и занимательно писать учебники, излагать их, показывать кинофильмы и пр., но если не создать рабочую обстановку, в которой школьники будут много и напряженно работать, решать задачи, то ничего усвоить не удастся, все усилия пропадут даром.

Необходимость напряженной работы для усвоения материала является, на наш взгляд, и основным достоинством, и основным недостатком курса, поскольку — как следствие — текст учебника достаточно сжат, выдержан в сухом физико-математическом стиле и очень многого не содержит. Например того, что обычно любят и учителя, и ученики, и что было бы очень полезным: разных занимательных примеров, историй, которые вводят в соответствующую область, создают мотивацию. Их надо искать по другим книжкам.

У этого принципа есть психологические и педагогические обоснования, он сейчас очень модный и даже обозначается специальным термином — «активное усвоение материала», или, если буквально переводить с английского, «понимание через делание». Практически это означает, что доля рассказов и объяснений учителя на уроках существенно (скажем вдвое) ниже, чем доля активной работы школьников. Под активной работой мы понимаем решение задач на компьютерах или в тетрадках, работу с учебными программными системами и т. д. То есть предполагается, что понимание будет приходить не столько при чтении или слушании объяснений, сколько в процессе практической работы, решения задач, практического применения изучаемых понятий и форм записи.

Позже я еще вернусь к этому принципу — и когда буду вам приводить «статистику» учебника (см. с. 47), и потом, при изложении материала учебника. А сейчас я перейду ко второму методическому принципу, который тесно связан с первым, — к проблемному подходу.

В2. Проблемный подход

Второй методический принцип называется «проблемный подход». Я поясню его на примере. Предположим, мы хотим ввести какую-то новую конструкцию школьного алгоритмического языка. Тут возможны два подхода. Первый — мы объясняем новую конструкцию, рассказываем все подробно школьникам, приводим примеры, показываем, как ее использовать, потом даем задачки, чтобы проверить усвоение материала. Это — подход, начинающийся с изложения понятий, и мы от такого подхода в нашем учебнике сознательно отказались.

Материал в нашем учебнике построен по-другому. Сначала ставится задача, хотя у учеников нет достаточных знаний для ее решения. Все начинается с задачи. Например, для введения цикла **пока** ставится задача: «Робот на клетчатом поле, где-то под ним на неизвестном расстоянии есть стена, надо подвести Робота вплотную к стене». Цикла **пока** ученики в этот момент еще не знают, у них нет средств для решения этой задачи — однако задача поставлена и они как-то должны ее решать.

Опыт показывает, что большинство конструкций языка школьниками изобретается. В тех или иных формах, более или менее правильные, но обычно пять-шесть разных разумных вариантов в классе появляется. Кто-то знает какой-то язык программирования и пишет на этом языке, кто-то придумывает сам. То есть школьники как-то выкручиваются. Если им поставить задачу и не сказать, как ее решать, они исхитрятся и как-то эту задачу решат.

Учитель в это время должен выполнять свою работу: кого-то похвалить, кого-то поправить, кому-то помочь сформулировать свое предложение. После некоторого обсуждения осознание необходимости конструкции приходит в голову всем. Кто-то придумывает, кто-то чувствует, что требуется нечто, чего он не знает, и т. д. В этот момент учитель может весь класс похвалить (хвалить — это главное, это залог успеха), выбрать самое правильное решение (а соответствующих учеников похвалить особенно, отметив, что человечество столько столетий к этому шло, а Петров раз — и придумал). Сказать, что примерно такая конструкция введена во всех современных языках программирования, после чего добавить, что в школьном алгоритмическом языке мы будем ***записывать*** ее так-то.

При этом происходит следующее: вместо ***изучения*** неизвестно откуда взявшейся новой конструкции, фактически дается всего лишь ***форма записи*** конструкции, только что придуманной на уроке. Происходит некоторая подмена — мы не излагаем новую конструкцию языка, мы поставили задачу, школьники ее решали, обсуждали, думали, а потом мы дали им ***форму записи того, что они сами придумали***. Почти все необходимые конструкции алгоритмического языка школьники могут изобрести сами. И в этом смысле изучения алгоритмического языка как такового не происходит. Он не является предметом. Учитель лишь показывает, как в школьном языке записывается то, что школьники так замечательно придумали сами.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. В системе «КуМир» конструкции языка порождаются целиком нажатием всего двух клавиш — ESC и первой буквы конструкции (цикл **пока** — ESC «П»). Поэтому при работе в этой системе запоминать детали записи конструкции нет необходимости, и изложение ***формы записи*** учитель может заменить объяснением, какие надо нажать клавиши. А к виду конструкции школьники привыкнут сами по ходу работы.

Проблемный подход для учителя труднее, так как требует больших знаний и мгновенного обсуждения возникающих на уроке предложений учеников

Конечно, проблемный подход для учителя труднее — ему нужно уметь на месте реагировать на предложения учеников, а для этого надо знать кое-что и за пределами школьного курса.

Пусть, например, Петров использовал цикл с проверкой условия в конце цикла (в Паскале этот цикл называется «repeat—until», а в системе КуМир — «**нц**—**кц** **при**»). Можно сказать ему, что он замечательно все придумал, вот и Вирт в свое время такую же конструкцию ввел в языке Паскаль, но у Иванова получилось еще более замечательно, у него условие в начале. А потом заметить, что мы пользуемся школьным алгоритмическим языком, и в нем это записывается в виде «**нц** **пока**—**кц**» т. е. условие проверяется в начале.

Но если у учителя есть понимание, то эти два цикла можно обсудить, рассмотреть их отличия, показать чем цикл с **пока** в начале лучше: например, показать, что если Робот

стоит вплотную к стене, то цикл с проверкой условия в конце (repeat-until) Робота разобьет, а цикл с проверкой условия в начале (**пока**) — нет. И это будет гораздо глубже, чем просто формальная ссылка на школьный алгоритмический язык.

И последнее о «проблемном подходе». При таком подходе конструкции языка не выступают, как какие-то сложные, содержательные вещи. На первом плане остается задача, на решение которой и уходят все усилия школьников. А форма записи, язык и прочее остаются на подобающем им втором плане.

Таким образом, все время учеников расходуется на решение задач. Проблемный подход — это когда конструкции вводятся на проблемах, а не наоборот. Мы не задачи подбираем для усвоения каких-то готовых конструкций, а решаем задачи и по ходу дела вводим те конструкции, которые для этого нужны.

В3. Выделение алгоритмической сложности «в чистом виде»

Третий методический прием — «разделение сложностей» и «очистка от всего лишнего»

Коль скоро мы провозгласили основной целью развитие алгоритмического стиля мышления и сказали, что в этой области есть самостоятельные сложности, которые надо преодолевать, с которыми надо учиться работать, то нужно эти сложности попытаться выявить и предъявить — по возможности — в «чистом виде». Чтобы школьники преодолевали именно алгоритмические сложности информатики, а не комбинацию сложностей информатики и, скажем, математики. То есть задачи, которые решает ученик, нужно очистить от математики, от лишних технических деталей, от английского алфавита и английских слов и т. д., словом, очистить от всего, от чего можно. И именно поэтому появился Робот.

Для понимания и управления Роботом нужен всего лишь здравый смысл

Робот сам по себе чрезвычайно прост, его и изучать-то нечего. Его можно изложить за три минуты, и даже второклассники все поймут. Клетчатое поле, Робот ходит по этим клеткам «вправо», «влево», «вверх», «вниз». Еще бывают «стены», и можно спрашивать, есть ли «стена» в каком-то направлении или там «свободно». Чтобы такое понять, вообще ничего знать не нужно, кроме «право» и «лево», разумеется.

Вместо трех минут у доски можно усадить детей за компьютеры и потратить десять минут в гипертексте «Знакомство с Роботом» в системе «КуМир». Там на экране они и поле Робота увидят, и стены, и пульт управления Роботом будет нарисован, и мышкой кнопки на нем можно будет нажимать, и контрольные вопросы им компьютер задаст, и т. д.

Таким образом, в самом Роботе нет вообще никаких сложностей. Работа с Роботом на клетчатом листе в тетрадке, да и на компьютере, в отличие, скажем, от каких-то математических действий, не требует никаких предварительных знаний вообще, т. е. ничего, кроме здравого смысла, которого на это хватает у всех учеников. И, следовательно, сам по себе Робот никакой сложности не представляет.

Поэтому можно быть уверенным, что если ученик думает над задачей по управлению Роботом, то все сложности у него алгоритмические, а не из других областей. В Роботе сложности нет, как и в постановке задачи: если я рисую на поле Робота прямоугольник 3 на 4 и говорю: «Вот здесь находится Робот, надо, чтобы он обошел вокруг этого прямоугольника неизвестных размеров», то такая задача тоже понятна любому, хотя на доске или на экране нарисован вполне конкретный прямоугольник.

Вопрос в том, смогут ли ученики написать соответствующие циклы, алгоритмы и т. д., но это и есть алгоритмическая сложность. В этом смысле алгоритмическая сложность задачи, алгоритмическая составляющая при работе с Роботом выделена «в чистом виде» и предъявлена как таковая.

> **ПРОГРАММНОЕ ОТСТУПЛЕНИЕ.** Тот же методический принцип лежит и в основе всего программного обеспечения курса. Система «КуМир» специально разрабатывалась таким образом, чтобы ученик мог решать задачи и концентрироваться на преодолении соответствующей алгоритмической сложности, а не разбираться с особенностями программного обеспечения. Я уже затрагивал этот вопрос, когда говорил о выборе средств обучения.
>
> Это касается и средств «порождения» конструкций языка, и немедленной диагностики ошибок при вводе алгоритма, и средств исполнения и показа результатов работы написанного алгоритма. В процессе исполнения на экране изображаются и алгоритм, и результаты его работы — перемещения Робота по полю. Выполнение можно прервать, алгоритм подправить и запустить вновь.
>
> Работа в «КуМире» «очищена» от понятий файлов и файловой системы, компиляции или интерпретации, не говоря уже о «командах операционной системы» или специальных отладчиках.

Но у принципа «очистки от всего лишнего» есть и обратная сторона. Конечно, при работе с Роботом на первых порах возникает ощущение игрушечности, кажется, что все это — «ненастоящее». Это может продолжаться до тех пор, пока мы не достигнем уровня, когда задачи станут сложными для лидера класса (чуть позже я остановлюсь на проблеме игрушечности более подробно).

С другой стороны, если мы не «очищаем» курс от технических и иных сложностей, если не вводим какой-то спе-

циальной учебной среды, то неизбежно падает алгоритмическая сложность решаемых задач, потому что мы должны отвлекаться на изучение английского языка, особенностей системы программирования, команд операционной системы и т. п. Мы также вынуждены подбирать задачи, в которых все происходит с числами (как правило, это математические задачи), либо рисовать графические картинки. В эти две области обычно все и скатывается.

Игрушечности Робота боятся не надо по тем же причинам, по которым не нужно бояться тренажеров при подготовке пилотов. Во-первых, мы же не весь курс будем Робота гонять, постепенно начнем от него отходить. Во-вторых, Робот-то хоть и игрушечный, но понятия и конструкции, которыми мы овладеваем в процессе возни с Роботом, вполне настоящие.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Но для учителя этот подход создает и некоторые дополнительные сложности. При работе с Роботом на алгоритмическом языке, ответ кажется тем очевиднее, чем лучше решение. Поэтому у некоторых учащихся может возникать ощущение «игрушечности» задач («что мы все Робота гоняем!»), а у других — при сравнении с чужими результатами, — разочарование («это ж так просто!»). Чтобы не потерять интерес, учитель должен идивидуально варьировать сложность задач для каждого ученика.

Забегая вперед, я проиллюстрирую это на примере следующей задачи: Робот стоит слева от вертикальной стены, надо составить алгоритм, который заставит Робота обойти стену и закрасить все примыкающие к стене клетки. Значительная часть класса сделает такую задачу быстро. Пока остальные раздумывают или (в машинном варианте) вводят свои алгоритмы в ЭВМ, тех, кто задачу уже сделал, можно попросить изменить алгоритм применительно к более сложной ситуации. Например, от исходной вертикальной стены кое-где влево и вправо могут отходить выступы, сначала размером в одну клетку, а потом и неизвестной длины. При каждом таком усложнении будет происходить очередное расслоение класса. Учитель может заготовить десяток разных усложнений и предлагать их в зависимости от уровня и особенностей ученика, наличия или отсутствия ЭВМ и пр.

Таким образом можно и создать напряженную рабочую обстановку для сильных учеников, и предоставить самым слабым возможность ***самостоятельно*** справиться с исходной задачей.

Но подобного рода работа ***в разном темпе*** с разными учениками требует больших усилий от учителя.

Возвращаясь к принципу «очистки от всего лишнего», повторю, что если алгоритмическую составляющую не выделить «в чистом виде», то ее сложность падает, и ***изучение основ алгоритмизации начинает подменяться изучением либо разного рода технических деталей, либо изучением другой предметной области, обычно математики.*** Мы начинаем решать квадратные, или разные другие уравнения, искать корни функции и, фактически, значительную часть сил тратим в математической области, а не в алгоритмической.

А поскольку, как вы прекрасно знаете, школьники разные, у одних лучше идет математика, у других информатика (во всяком случае при их разделении), у третьих еще что-то, при таком преподавании информатики на материале математики просто выпадает целый пласт учеников, которые математику знают плохо, боятся ее, неуверенно себя чувствуют или теряют интерес, когда начинается что-то математическое. Поэтому, «отделяя» информатику от математики, мы, кроме всего прочего, создаем условия для появления в классе новых лидеров — лидеров по информатике, даем ученикам проявиться в этой новой области, более или менее независимо от их успехов по другим предметам.

Специальные учебные среды для обучения информатике широко распространены во всем мире

Завершая обзор этого глобального методического подхода, замечу, что учебные среды для обучения информатике не есть чисто российское явление. Самая известная учебная среда — это «черепашья графика» и язык «Лого», придуманные С. Пейпертом (США) как методический прием для обучения младших школьников.

Достоинства этой среды — простота и наглядность. Недостаток — отсутствие «обратной связи», и, как следствие, ненужность в этой среде конструкций ветвления и повторения (за исключением цикла «N **раз**»). Мы эту идею немного развили. Мы дискретизировали и упростили «черепашку» — т. е. сделали «вправо», «влево», «вверх», «вниз» на шаг без аргументов. И, что самое главное, мы ввели обратную связь. «Черепашка» у Пейперта может только выполнять коман-

ды — «повернуться на заданный угол», «вперед на . . . » и др. Мы ввели обратную связь — вопросы типа «справа стена?», «слева свободно?», «клетка закрашена?», которые можно задавать Роботу. Соответственно, в алгоритмах стало возможным писать не только последовательности команд и вспомогательные алгоритмы, но и конструкции типа «если клетка не закрашена, то закрасить ее». А это позволило ввести циклы и и развилки ***до появления величин.***

Разные авторы в разных странах независимо развивали это направление. Поскольку стремились они к одной цели — создать среду для развития алгоритмического стиля мышления, то получилось примерно одно и то же: что-то, движущееся на клетчатом поле. Все это началось в районе 1980 года. У нас — в курсе программирования на механико-математическом факультете МГУ — появился Путник (на основе которого и был потом создан Робот для школы). В США — Карел-робот, который ходит по клетчатому полю со стенами, может собирать и переносить какие-то предметы. В Новосибирске — Муравей, двигающий по клетчатому полю кубики с буквами, и т.д.

Специальные учебные языки и системы типа «КуМира» также широко распространены в мире. Например, в США пользуется популярностью система Карел-джин, позволяющая составлять программы управления Карел-роботом на учебном языке программирования без переменных. А в Европе для управления тем же Карел-роботом широко распространен учебный язык программирования Elan. Для управления Путником мы в 1980 году использовали учебный язык Мини — также без переменных. Даже Бейсик и Паскаль были созданы первоначально как учебные языки, предназначенные для обучения студентов основам программирования (хотя про учебные системы для управления роботами в те годы еще и речи не шло).

Пейперт в конце восьмидесятых тоже занялся роботами и управлением ими с помощью компьютера, но роботами реальными, например, собранными из конструкторов Lego или Fischer. Работают такие машинки не на клетчатой плоскости, а на обычном столе или на полу. Это делает задачи и интереснее, и намного труднее. Я бы их рекомендовал для школ с углубленным изучением математики или для факультативов. В системе «КуМир» есть похожий исполнитель Вездеход, и вы можете в гипертексте «Знакомство с Вездеходом» посмотреть типовые задачки. Но, конечно, это всего лишь компьютерная модель Вездехода, а не реальный вездеход.

С. Общий обзор учебника

Теперь, когда вы знакомы с основными целями и основными методическими принципами построения курса, я хочу описать наш учебник «в целом», создать у вас целостное представление о нем.

С1. Распределение материала в учебнике

Начну с формальных вещей — с так называемой «статистики» распределения материала в учебнике. Я бы хотел, чтобы при восприятии этой информации Вы помнили о «черепахе» учебника — об ориентации всего курса на создание обстановки для активной и напряженной работы учеников.

Всего в нашем учебнике 240 страниц, из них:

153 с. — так называемый основной текст, т. е. изложение основного материала со всем, что ему сопутствует, всего 28 параграфов;

62 с. — формулировки примерно 700 задач и упражнений (если считать с подпунктами). Конечно, все 700 решить невозможно даже за 102-часовой курс. Но учитель может выбирать те или иные задачи, варьировать их сложность и пр.;

16 с. — цветная вклейка. Здесь 40 фотографий, примерно 30% из них используется содержательно (т. е. на них есть ссылки в тексте учебника), а остальные — просто иллюстративный материал;

2 с. — предметный указатель, включающий как слова «алгоритм», «алгоритмизация», «информация», так и «драйвер», «шина» и т. п., — всего 182 термина;

5 с. — подробное рабочее оглавление. В конце — список литературы из двух книг. Почему их всего две, я скажу позже.

Основной текст (153 страницы) содержит:

— 120 полных алгоритмов («полных» означает, что алгоритмы написаны от начала и до конца);

— 20 информационных моделей, начиная с «информационной модели кинозала» и кончая информационными моделями учебных информационных систем: система

продажи билетов в «вагон», едущий без остановок; «телефонный справочник»; учебный экранный редактор текстов и пр.;
— **146** рисунков, схем, таблиц и т. д.;
— и еще около **30** фрагментов алгоритмов для пояснения чего-либо.

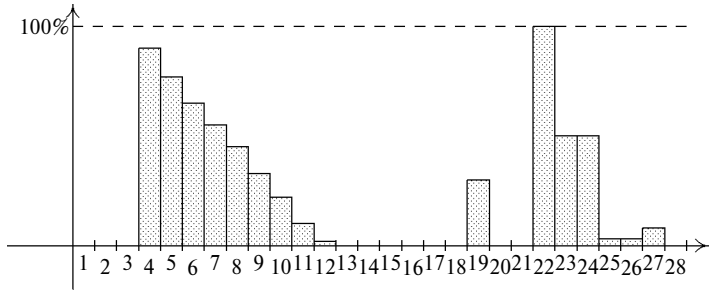
Вы видите, что на каждую страницу приходится примерно по одному алгоритму или фрагменту алгоритма, по одному рисунку или схеме и т. д. Как следствие, учебник является очень сжатым. Он удобен для повторения, для использования в качестве справочника при составлении алгоритмов, а также в качестве задачника — для выбора задач. Но, в том, что касается собственно изложения, материал крайне сжат, и, за редкими исключениями, практически не содержит занимательных примеров, мотивировок, зажигательных введений, исторических экскурсов и т. п. Все это — важная задача учителя, но в учебнике этого нет.

НЕБОЛЬШОЕ ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Правда мы считаем, что нам удалось эту сжатость частично компенсировать за счет подачи материала в стиле учебников математики А.П. Киселева. Основное отличие учебников Киселева от многих других состоит в том, что если открыть такой учебник в любом месте, то увидишь заголовок, за ним несколько строк объяснений. Потом опять заголовок — формулировка теоремы, за ней доказательство. То есть все разбито на небольшие фрагменты и в начале каждого объявлено, о чем идет речь. Мы заимствовали киселевский метод — разбили все на небольшие фрагменты. Таких фрагментов обычно два, три, четыре на каждом развороте учебника. И мы считаем, что за счет этого удалось написать сжато, но в то же время еще достаточно понятно.

С2. Понятие исполнителя в курсе и учебнике

Теперь от формально-статистического обзора учебника перейдем к более содержательному. Второй мой обзор будет посвящен месту и роли понятия «исполнитель».

Относительную значимость понятия «исполнитель» на разных стадиях прохождения курса можно пояснить следующей картинкой:



Первые три параграфа идет «Введение», я их пока пропущу. В § 4 появляется «школьный алгоритмический язык». Начинается он с исполнителей «Робот», «Чертежник» и, соответственно, алгоритмов управления этими исполнителями. Если взять общий объем материала параграфа за 100%, то в начале почти все 100% будут заняты понятием исполнителя (а именно, Роботом). То есть понятие «исполнитель» на старте доминирует, вокруг него группируется все остальное. Понятие «алгоритм» вводится как алгоритм управления исполнителем, а понятие «алгоритмический язык» — как средство записи будущих действий ЭВМ по управлению исполнителем.

В следующих параграфах в процессе решения различных задач по управлению Роботом и Чертежником постепенно излагаются конструкции школьного алгоритмического языка: **цикл**, **если**, **выбор**, далее вводятся память ЭВМ, «величины», «табличные величины». Акцент с исполнителей переносится на язык и ЭВМ. И к § 13 исполнители совершенно исчезают. В § 13–18 все делается без привлечения «исполнителей» вообще.

Робот и Чертежник становятся как бы историей. В этот момент можно просто решать задачи на алгоритмическом языке, не используя «исполнителей». Здесь главным для учеников являются знание конструкций языка, умение соста-

влять алгоритмы, а вовсе не система команд Робота или понимание его «исполнительской» сущности.

Дальше, в § 19 вводятся исполнители «клавиатура», «монитор», «дисковод» и др., которые являются настоящими «аппаратными» исполнителями, подключенными к ЭВМ.

А вот начиная с § 22, исполнители появляются снова, но уже не как физические устройства, подключенные к ЭВМ, а как основная конструкция нашего алгоритмического мышления и, соответственно, алгоритмического языка («**исп**—**кон**»). В § 23, например, рассматривается исполнитель «вагон», содержащий информацию о проданных и непроданных билетах в один вагон, следующий без остановок, а также алгоритмы продаж этих билетов и т. д. Здесь начинается, если угодно, диалектическая спираль, новый виток понимания исполнителей на совершенно ином качественном уровне: ***исполнители — как конструкция языка программирования, как способ структуризации нашего мышления.***

Конечно, такое новое понимание исполнителей не противоречит, а лишь развивает и обогащает старое. Материал, связанный с этим понятием, присутствует в §§ 22–24, 27, занимая всю содержательную половину §§ 23–24, что и показано на рисунке.

Это — общая схема построения курса с точки зрения понятия исполнителя, и она проясняет место и роль Робота и других «игровых» исполнителей. На старте они занимают все и служат базой для постепенного, плавного введения основных конструкций языка, а также для опережающего введения понятия вспомогательного алгоритма с параметрами. Со временем Робот заменяется более содержательным материалом. К § 12 Робот вытесняется почти полностью, — в этот момент все конструкции языка уже освоены и можно составлять алгоритмы без всяких исполнителей.

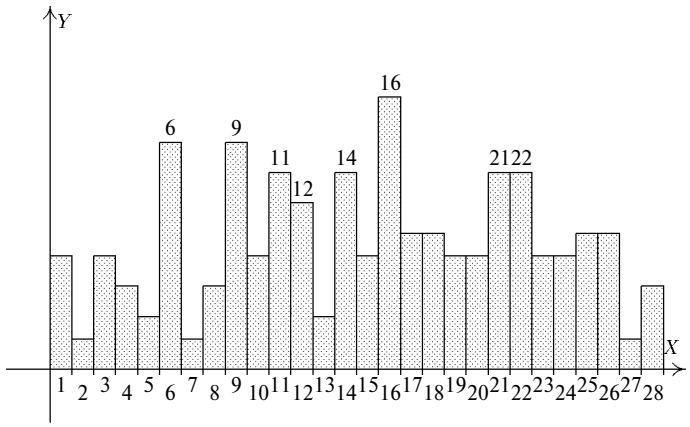
Есть курсы информатики, построенные без такого плавного нарастания, в которых изложение начинается с объяснения понятия величины, команды присваивания, команд ветвления и повторения. Все это вываливается на ученика одной кучей, из которой ученик должен постепенно выбираться, решая задачи (в основном математического содержания). Если в таком курсе в качестве языка выбран классический Бейсик, то вспомогательные алгоритмы вообще не возникают. Если же выбран более современный язык, то вспомогательные алгоритмы появляются, как правило, в конце курса.

Есть и курсы, которые построены с использованием исполнителей для плавного ввода конструкций языка. В частности, примерно так преподавал Г.А. Звенигородский. Но у Звенигородского исполнители играли чисто методическую роль и нужны были только для последовательного введения конструкций. А потом начиналось просто составление алгоритмов без каких-либо исполнителей. То есть это как бы для старта, для маленьких, для лучшего понимания и т. д.

Я повторю, что часть нашего курса, связанная с исполнителями как моделями (§§22–24), вызвана более поздними этапами развития информатики. У нас были довольно большие споры по поводу того, включать или не включать этот материал в школьный учебник. Но — по причинам, которые я вам при обсуждении третьего «кита» обозначил, — мы решили, что совсем без этого понятия обойтись нельзя. А кроме того, без «исполнителей» невозможно содержательно рассказать о применениях ЭВМ. Нельзя объяснить все эти информационные системы и пр., не показав, как они, собственно говоря, устроены внутри.

С3. Относительная важность и сложность материала в учебнике

Ну а теперь я еще раз пройду по учебнику от начала до конца и расскажу, где в нем наиболее трудные места, где наиболее важные, что можно опустить при нехватке времени, и на чем следует остановиться поподробнее. Я приведу еще одну картинку, только на этот раз я не могу сказать, что именно будет отложено по оси Y. Дело в том, что параграфы в учебнике разные. Одни из них вводят понятия фундаментальные, но не очень сложные; другие — может быть вводят не столь фундаментальные, но более сложные понятия и т. д. Можно пытаться изобразить как-то условно, относительно некоторую интегральную оценку параграфов: то ли важность, то ли сложность. Иначе говоря место параграфа в курсе в целом. Вот эта картинка — во всяком случае, она показывает основные акценты, основные места нашего учебника:



Итак, по оси X указан номер параграфа: 1-й, 2-й и т. д., а по оси Y изображается условно некоторая интегральная оценка каждого параграфа, отражающая его место в курсе в целом (например, время, которое надо выделить на изучение параграфа).

А теперь я бегло пройду по параграфам и отмечу их основные особенности.

1) § 2 — первый кандидат на пропускание, если вам не хватает времени. Этот параграф лишь утверждает, что машина состоит из процессора и памяти, клавиатуры и монитора. Да и упражнения после него тоже не особенно важны для нашего курса: в них надо что-то сложить, умножить, поделить и получить, скажем, сколько учебников влезает в память ЭВМ. Довольно часто изложение содержания этого параграфа учителя совмещают с первой работой на компьютере: «Вот это — монитор, это — клавиатура, а теперь сделайте то-то и то-то».

2) §§ 6, 9, 11, 12, 14 (про 16 скажу чуть позже), 21, 22 — это семь «ключевых» параграфов учебника, в которых вводятся те самые 4 фундаментальные понятия информатики, которые я вам выписывал при изложении третьего «кита»: 6 — понятие «вспомогательного алгоритма» и 12 — «вспомогательного алгоритма с результатами»; 9 — цикл **пока**; 11 — понятие «величины» и 14 — «табличной величины»; наконец, 21, 22 — понятие «информационной модели исполнителя», про которое я вам уже несколько раз говорил.

3) § 16 так выделяется за счет того, что он является, по-видимому, самым сложным параграфом курса. Это единственный параграф, который посвящен методам алгоритмизации как таковым. Если в других параграфах решаются задачи и лишь мимоходом говорится, что их можно решать так-то и так-то, то § 16 излагает именно методы решения задач. Если опять проводить аналогию с математикой, то представьте себе, что мы решаем квадратные уравнения, скажем, подбором, или ссылаясь на теорему Виета, или еще как-то, а потом изучаем общую формулу решения корней квадратного уравнения. Так вот, изучение методов информатики в § 16 — это аналог изучения формул и теорем математики, которые потом позволяют решать любые задачи из соответствующего класса задач.

Соответственно § 16 более сложен для изучения и требует большей работы и более высокого уровня знаний. Он так «высок» на картинке за счет своей фундаментальности — это действительно методы, те самые методы информатики как науки.

Замечу, впрочем, что мы этот материал в достаточной степени локализовали. За исключением моделирования падения парашютиста (§ 26, п. 26.3), методы из § 16 нигде в учебнике прямо не используются. Поэтому в слабых классах этот параграф можно пропустить, или же задать его сильным ученикам, чтоб они не скучали, пока класс изучает простые конструкции алгоритмического языка.

4) § 17 и 18 достаточно «высоки» по совсем другой причине — это та демистификация ЭВМ, о которой я вам говорил при обсуждении третьего «кита» (помните, — провода, лампочки, включаем, выключаем и прочее). Эти параграфы дают представление (на чисто физическом уровне и на уровне машинных команд), как в принципе ЭВМ может какую-то информацию хранить и перерабатывать, как она вообще может работать.

5) среди «низких» параграфов имеются § 7 и § 13, которые носят характер, я бы сказал, «перерывов».

§ 7 — это линейная запись арифметических выражений. Он никак не связан с предыдущим материалом и не является абсолютно необходимым для изложения последующего. Там, где встречается линейная запись выражений, она может быть пояснена просто по ходу изложения. Поэтому сам § 7 можно переместить в любое другое удобное место, и он, при всей его «настоящести», носит характер «отдыха», переключения на совсем другую — простую и приятную — деятельность после § 6.

§ 13 — это команды **ввод**, **вывод** и цикл **для**. Цикл **для** используется далее в § 14 при составлении алгоритмов работы с таблицами, а сам § 13 опирается на § 11 (величины). Но в целом § 13 тоже носит характер «отдыха»: после сложного материала про «величины», алгоритмы с результатами и алгоритмы функции — это пауза, в которой изучается уж совсем простой и легкий к этому моменту материал. Соответственно, § 13 также можно частично (например, ввод/вывод) переместить в другое место.

6) Параграфы с 17 по 20 — это раздел «Устройство ЭВМ». В принципе, если устройство ЭВМ и формирование картины «ЭВМ в мире» не затрагивать, т. е. заботиться только о развитии алгоритмического мышления, то этот раздел можно целиком пропустить.

7) Начальные §§ 21–22 главы 3 «Применения ЭВМ» очень важны. В § 21 вводится понятие информационного моделирования. В § 22 — четвертое фундаментальное понятие информатики — понятие «информационной модели исполнителя», а также конструкция алгоритмического языка «**исп**—**кон**».

8) В § 23 введенные в § 21–22 понятия применяются к нескольким информационным системам. Точнее, в первой части параграфа идут обычные беглые описания нескольких систем (подобные описания можно найти в сотнях разных популярных книжек). А вот во второй части параграфа на

учебных примерах показывается, как информационные системы устроены ***внутри***, как они работают.

Рассматривается, например, система продажи билетов в один вагон, следующий между пунктами А и Б без остановок. Как представлять информацию о проданных, непроданных билетах? Как написать алгоритм «продать два места в одно купе» или «продать место на нижнюю полку»? Или учебная модель «телефонная книжка», в которой хранятся фамилии, имена, отчества и номера телефонов абонентов, а школьникам надо решать задачи типа «придумайте способ кодирования адресов и напишите алгоритм, который по заданному адресу определяет, сколько в этом доме установлено телефонов».

> **МЕТОДИЧЕСКОЕ И ПРОГРАММНОЕ ОТСТУПЛЕНИЕ**. Заметьте, что здесь речь идет отнюдь не о «неформальных алгоритмах» в стиле первых учебников информатики А.П. Ершова, а об абсолютно нормальных, формальных алгоритмах, которые немедленно можно выполнить на компьютере. В гипертексте «учебник» системы «КуМир» есть все алгоритмы, модели и формулировки упражнений учебника. И указанные выше задачи можно прямо решать — т. е. набирать и проверять (выполнять) на компьютере.

9) §§ 24–27 повторяют структуру § 23: в первой — описательной — части рассказывается о применениях ЭВМ в какой-то области, а во второй — содержательной — части на учебных примерах показывается, как представлять и обрабатывать информацию в этих областях. (Исключение составляет только § 27, на содержательную часть которого просто не хватило места. Мы еще сумели втиснуть туда простейшую конечноэлементную информационную модель для представления поверхности автомобиля, но сделать с ней ничего уже не успели — в учебнике кончилось место.)

§§ 23–27 чрезвычайно важны для построения адекватного представления о применениях ЭВМ, о том, какую информацию из внешнего мира хранит система; как эта информация представлена, на какие вопросы и почему система может или не может ответить.

В облегченных курсах (при нехватке времени или при иных целях курса) вы можете:

— затронуть только описательные части этих параграфов и опустить содержательные — тогда, соответственно, можно пропустить и понятие исполнителя в §§ 21, 22. Это бу-

дет означать, что из перечисленных мною в начале четырех фундаментальных понятий информатики вы ограничились изучением трех;

— наоборот, разобрать только информационные системы (§ 23), т. е. есть показать методы информатики лишь в одной области, сказав, что в остальных происходит примерно то же самое.

— идти по учебнику, но пропустить §§ 25 и 26, которые, как и § 16, достаточно сильно математизированы.

10) Наконец, § 28 — чисто «беллетристический», но он, на мой взгляд, очень важен, поскольку это единственный параграф в курсе, где отмечается, что ЭВМ — не только хорошо, но и плохо. В начале учебника мы старались создать возвышенное представление об ЭВМ, сказать, что ЭВМ — это будущее; что три фундаментальные составляющие мира — вещество, энергия и информация; что ЭВМ — универсальная машина для обработки информации; что отсутствие компьютерных сетей — как отсутствие дорог и электричества, и т. д. В заключительном § 28, наоборот, утверждается, что компьютеры — не панацея, и показывается, что они могут быть просто вредными, если мы стремимся к ним, как к таковым, а не к решению каких-то задач с их помощью.

С4. Несколько слов о месте курса в школьном образовании

Исключив более сложные и математизированные § 16 и § 25, которые максимально локализованы, не связаны с другими параграфами, материал учебника можно положить в основу курса для 7–8 классов (где, по моему мнению, ему и место).

Но такой курс, как нам кажется, не может иметь общеобразовательного продолжения для старших классов. Курс и учебник, конечно, можно улучшить, сделать более занимательным, подать материал совсем по другому, усилить разделы по информационным моделям и т. д. Но по набору понятий и по целям, которые мы хотели достичь, наш курс законченный, замкнутый. Вы можете вспомнить, что я говорил про трех «китов» и четыре фундаментальных понятия информатики, чтобы попытаться меня понять. Курс не имеет общешкольного продолжения.

Поэтому, если наш курс сместить в 7–8 классы, то в старших классах ничего нового изучать не придется, — надо будет просто применять компьютеры и, возможно, использовать понятия курса в математике, физике, иностранном языке, биологии и других предметах. Позже я вам покажу, как это может быть устроено, но это уже будет выход за рамки нашего предмета — курса информатики.

Хотя общешкольного продолжения у курса нет, это, конечно, не значит, что он охватывает все понятия всей «настоящей» информатики. Курс можно развивать в сторону предпрофессиональной или профессиональной подготовки по программированию и смежным областям. Можно проводить специализированные, факультативные занятия для тех, кто хочет более углубленно изучить, например, структуры данных (тут я могу рекомендовать наш вузовский учебник [ПдМ]), параллельное программирование, локальные и глобальные сети ЭВМ, методы защиты данных от несанкционированного доступа и пр. Простор тут, можно сказать, безграничный.

Лекция 3.

Ну а теперь будем двигаться по учебнику. Я буду акцентировать внимание на наиболее значимых, на мой взгляд, моментах, понятиях. Буду расставлять акценты, которые в самом учебнике отсутствуют.

Введение

Все «Введение» посвящено раскрытию понятия информатики, как науки изучающей обработку информации с помощью ЭВМ. Три параграфа «Введения» объясняют 1) что такое информация, 2) что такое ЭВМ, 3) что такое обработка информации на ЭВМ.

§ 1. Информация и обработка информации

Начинается параграф с воодушевляющего введения суть которого сводится к лозунгу: «Информатика — это очень важно».

Если от этого неформального введения отвлечься, то параграф посвящен понятиям «информация» и «обработка информации» (пока без ЭВМ).

Таким образом, основные цели параграфа таковы:

0) сформировать общее ощущение места и значимости информатики (триада «вещество, энергия, информация»);

1) показать отличия понятия «информации» в информатике от бытовой трактовки информации как смысла или значимости сообщения;

2) дать представление об «обработке информации» как о техническом преобразовании, производимом по строгим правилам и не связанном с восприятием информации или ее осмыслением.

Здесь сразу надо отметить самое главное — в этом параграфе понятия «информация» и «обработка информации» даются как технические понятия информатики. Основной акцент всего параграфа — отличие формальных понятий информатики от бытового смысла этих слов. Например, разница «бытового» представления об информации, как о смысле

или значимости, и «технического», как о последовательности битов и байтов.

Для усвоения этого различия в учебнике ***вначале*** речь идет о технических вещах — о двоичном кодировании информации. Причем, что такое ***информация***, в этот момент не говорится вообще — вопрос тщательно обходится, «замазывается». Вместо этого обсуждается проблема измерения ***объема информации***, в случае, когда информация ***уже закодирована двоично***.

А потом сообщается, что никакого строго определения понятия «информация» нам и не нужно. Представляйте, мол, себе порцию информации как последовательность нулей и единиц, а количество информации — как длину этой последовательности. И все.

Я хочу заметить, что при всей своей простоте такой подход — совершенно честный. Нам действительно ничего больше и не нужно. Житейские представления о том, что информация может быть актуальная или устаревшая, верная или неверная и т. д., если даже и можно формализовать, все равно бессмысленны в школьном курсе информатики. С понятиями своевременности или нужности информации нам делать абсолютно нечего.

Наконец, и об этом в учебнике говорится прямо, мы не можем дать строгого определения понятия «информация». И не потому, что оно длинное или сложное, а потому, что в «большой» информатике понятие информации считается первичным, неопределяемым.

Отличия понятия «информация» в информатике от бытового смысла этого слова

Итак, первый концентр параграфа, первое, на чем вы должны заострить внимание — это отличие «технического» понятия «информации», как последовательности битов, байтов, символов, от бытового значения «информации», как какого-то содержания, смысла. Если мы, скажем, получаем телеграмму: «ПРИЕЗЖАЮ ПОНЕДЕЛЬНИК ДЕСЯТЬ УТРА = МАМА», а мама час назад позвонила и все уже сказала, то в бытовом смысле эта телеграмма для нас никакой информации не несет: мы и так знаем, когда мама приедет. То есть с бытовой точки зрения мы не получаем никакой информации, получаем «ноль» информации.

А с технической точки зрения — с точки зрения информатики — мы можем подсчитать, сколько символов в этой телеграмме, сколько затрачено времени на ее передачу по телеграфным линиям и т. д., и это совершенно не зависит ни от текста телеграммы, ни от состояния головы получателя телеграммы. С точки зрения информатики, информация есть в любой «белиберде». Можно подсчитать объем этой «белиберды» в байтах, можно измерить время передачи ее по телеграфной линии, поделить полученные числа и получить скорость передачи информации. И все это можно делать, не вникая в смысл.

Это «техническое» понятие информации, как последовательности битов или байтов, никак не связанное ни со смыслом, ни с значимостью сообщения — вот главное, что здесь вводится.

Как всегда («черепаха» курса), чтобы это различие усвоить, школьники должны поработать, что-нибудь поделать. И именно для этой цели используется двоичное кодирование. Двоичное кодирование здесь играет роль того средства, повозившись с которым (покодировав, пораскодировав) школьники должны освоить, как-то воспринять понятие информации, как последовательности байтов, битов. Тут полезно повозиться с нулями и единицами на любом материале, поиграть в разные шифры, в шифровку и дешифровку. И чем бессмысленнее будут последовательности символов при этой возне, тем лучше. Смысл и значимость должны отойти на второй план, чтобы остались только последовательности нулей и единиц или последовательности символов.

Вы можете также смело задействовать любые иные (взятые из жизни, из литературы или просто придуманные) способы кодирования/декодирования и передачи информации. Здесь можно использовать разные шифры, азбуку Морзе, морскую азбуку (поднятие/опускание флажков) и т. п. От последовательности поднятых/опущенных флажков, например, очень легко перейти к двоичному кодированию, к последовательности нулей и единиц. Можно и наоборот, вместо нулей и единиц, использовать флажки. Скажем, в упражнении 6 можно предложить передать оценку с помощью флажка. Рекомендуется при этом постоянно подчеркивать формальность понятия «информация», предлагать кодировать и передавать не только осмысленные, но и произвольные последовательности символов. Как правило, физические действия (поднятие/опускание флажков) способствуют восприятию единиц измерения информации, как не зависящих от ее смысла.

Это первый и основной концентр параграфа. К нему примыкает введение единиц измерения информации, в которых никакого особенного содержания и никакой сложности нет. Нужно просто привыкнуть к новым словам (бит, байт, килобайт, мегабайт) и к их смыслу, для чего в конце параграфа есть несколько упражнений.

Понятие «обработки информации» в информатике

Второй концентр параграфа — это понятие ***обработки*** информации. Здесь также надо показать, что под «обработкой информации» понимается строгое формальное преобразование.

Если я в тексте все буквы «е» меняю на буквы «и» — это формальное преобразование, т. е. обработка информации с точки зрения информатики.

А вот если я посмотрел кино и рассказал другу, какое оно замечательное, то это — с точки зрения информатики — назвать обработкой информации нельзя. Это моя оценка качества кино — «человеческое» действие, оценка информации, а не ее формальная обработка по четким и однозначным правилам.

Второй концентр нацелен на то, чтобы сформировать представление об «обработке информации» как о техническом преобразовании, проводимом по строгим формальным правилам. В самом параграфе — это небольшой раздел (1.7). Усвоение этого понятия должно происходить в ходе выполнения упражнений, приведенных в конце параграфа.

Игра в «черные ящики»

Я хочу особо обратить ваше внимание на игру в упражнениях 12–13. Мы узнали об этой игре в г. Переславле-Залесском от авторов «Роботландии» — замечательной программной системы для обучения информатике младшеклассников. (Я не знаю, существовала ли эта игра раньше или была придумана переславцами). Имеются и компьютерные реализации этой игры, но, по-моему, без компьютера играть интереснее. Переславцы называют эту игру «Черный ящик». Термин «черный ящик» возник в кибернетике и означает устройство, получающее информацию на входе и выдающее

переработанную информацию на выходе, причем внутреннее устройство «ящика» нам не известно (отсюда и термин «черный», т. е. непрозрачный).

Содержание игры состоит в том, что учащимся предлагается разгадать, как работает тот или иной «черный ящик» (в упражнениях учебника роль «черного ящика» играет Петя): на «вход» «ящика» поступает информация, последовательность символов, на «выходе» получается другая, учащиеся должны догадаться, как этот «черный ящик» работает.

Например, на «вход» поступает информация «1», на «выходе» получается «2». На «входе» — «17», на «выходе» — «18», и т. д. Сделав несколько проб, учащиеся начинают догадываться — «черный ящик» прибавляет единицу. Другой «черный ящик» может, скажем, считать число символов в строке (упражнение 12б).

Хотя внешне это облечено в игровую форму, тем не менее основное и фундаментальное содержание, которое здесь вводится, — формальность правил обработки информации.

В «черные ящики» можно играть на уроках: первоначально учитель придумывает «черные ящики» и играет их роль. Например, я говорю, что я — «черный ящик», вы, ученики, можете мне задавать входные последовательности. Когда ученик диктует, я выписываю на доске входную последовательность, рисую стрелочку, как в учебнике, по своему внутреннему правилу преобразую, выписываю ответ. На доске получается строка: «вход $\longrightarrow$ выход», под ней вторая и следующие. Ученики всем классом смотрят на доску и пытаются догадаться, как же работает «черный ящик».

Пример игры в «черный ящик»

Хотите сыграть? Давайте. Я задумал некоторое правило преобразования. Задавайте последовательности. (Из зала: «3», «9», «-7», «0»...). На доске появляются 4 строки:

$$\begin{aligned} 3 &\longrightarrow 0 \\ 9 &\longrightarrow 0 \\ -7 &\longrightarrow 0 \\ 0 &\longrightarrow 0 \end{aligned}$$

Вы можете меня остановить, если догадались, и сказать: «Все — я знаю». Когда какой-то ученик произнес «Все», возможны два пути продолжения игры. Один путь — ученик вслух формулирует правило. Например:

Лектор (обращается к слушательнице): Скажите Ваше правило.

Ответ: Вы умножаете на ноль.

Я, поскольку знаю, как «черный ящик» устроен, говорю «правильно» или «неправильно». Если «правильно», то игра заканчивается. Другой путь позволяет игру продолжить — я прошу «не называйте свое правило, а скажите мне, что оно даст на какой-то последовательности, которой еще нет на доске». Например:

Лектор (обращается к слушательнице): Вы можете сказать, что дает Ваше правило для слова «мама»?

Ответ: Нет.

В такой ситуации я говорю ученику: «Неверно, мое правило для слова «мама» работает и дает выход «2»:

$$\text{мама} \longrightarrow 2$$

Значит, ученик еще не догадался.

Из зала: «бабушка»

Лектор: бабушка$\longrightarrow$3

Из зала: Если число, то умножаем на ноль, если текст, то считаем число слогов.

Лектор: Я задумал число гласных букв. Видимо, это одно и то же — давайте мы пока закончим.

Если бы Вы правило не назвали, то мы могли бы поиграть еще. Я дал бы Вам пару слов, убедился, что Ваше правило дает тот же результат, что и мое, объявил бы Вас отгадавшим (не зная Вашего правила), и продолжал бы играть с остальными учениками. То есть, если в классе один ученик догадался, то можно продолжать игру. Не обязательно, как только первый додумался, заканчивать. В классе Вы сами сообразите, когда продолжать играть, а когда пора закончить или устроить совместное обсуждение результатов.

Использование игры при изложении понятия «обработка информации»

Для раскрытия понятия формальной обработки информации я хочу привести один пример, который нигде в учебнике не фигурирует и который, по-моему, вы обязательно должны использовать на уроках. Представьте себе, что я играю роль «черного ящика» в классе. Ученики задают мне

входную информацию, а я в ответ пишу «1» или «0» по следующему «правилу»: если «вход» мне нравится, то я пишу «1», если нет, то «0».

Через некоторое время, когда ученики совершенно уже запутаются и «сдадутся», я раскрываю им свое «правило»: «1» — если нравится, и «0» — если не нравится. Обычно класс хором взрывается: «Это не честно».

Выражение «это не честно» является очень содержательным, поскольку в данном случае нет никакой формальной обработки информации, а есть ее «человеческая оценка». И приведенный пример позволяет четко противопоставить эти два понятия. Это контрпример — пример не обработки, а оценки информации, и именно поэтому он так важен.

Другой очень важный пример «ящика», демонстрирующий в каком-то смысле противоположную грань термина «обработка информации» это «черный ящик», который выдает на выходе «да» и «нет» по очереди, через раз, независимо ни от какой входной информации. Когда ученики этот «ящик» разгадают, попросите их сначала самих подумать, является ли такое преобразование «обработкой» информации в смысле информатики. Как правило, они начинают путаться, поскольку привычный бытовой смысл термина «обработка» сопротивляется его применению в данной ситуации. А потом объясните им, что (а) они это правило смогли угадать, (б) изготовить техническое устройство — «ящик», который будет осуществлять это преобразование, можно, т. е. «преобразование информации» происходит по совершенно строгому, формальному правилу. Таким образом, речь идет безусловно об «обработке информации» в смысле информатики, хотя это преобразование от входной информации и не зависит.

Забегая вперед, замечу, что в игре в «черный ящик» задача учеников состоит в том, чтобы, глядя на результаты преобразования информации, угадать ***алгоритм*** преобразования. Именно поэтому игра в «черные ящики» является основным методическим приемом при прохождении темы «формальная обработка информации».

Как и когда следует играть в «черный ящик» в курсе информатики

Резюмируем, основные рекомендации по организации игры в «черный ящик».

1. В классе рекомендуется играть одному против всех. Это значит, что учитель или один из учеников, придумавший «черный ящик», стоит у доски и выписывает все задаваемые ему вопросы («входную информацию») и свои ответы на них («выходную информацию»). Все остальные, задавая вопросы и анализируя ответы, пытаются угадать алгоритм.

2. Когда кто-либо сообщает, что он уже угадал алгоритм, можно, не оглашая правило, провести проверку, которая состоит в следующем: учитель (играющий роль «черного ящика») задает контрольные вопросы угадавшему ученику, а тот, преобразуя информацию ***по своему алгоритму***, выдает ответы. Если ответ неверен — значит, алгоритм еще не угадан, учитель («черный ящик») должен привести правильный ответ, и игра продолжается. Если ответ верен (это еще не означает, что алгоритм учителя действительно угадан: возможно и просто случайное совпадение), то, тем не менее, учитель может еще какое-то время поиграть с остальными учениками. Такой подход позволяет ученикам, угадавшим алгоритм, проверить себя, не раскрывая алгоритм другим играющим, что и дает возможность продолжить игру, даже когда несколько учеников уже знают алгоритм.

3. Игра в «черный ящик» наибольший эффект приносит в начале курса. Однако ее можно использовать и в дальнейшем:

а) для введения новых алгоритмических конструкций (с помощью «ящиков», алгоритмы которых содержат циклы и развилки);

б) для разрядки и повышения мотивации, играя в соответствующий «ящик» перед составлением того или иного алгоритма преобразования информации (особенно это относится к литерным величинам в § 15).

4. Наконец, игру можно использовать, если в конце урока осталось 2–3 свободных минуты, а также на переменах, школьных вечерах. Полезно устроить конкурс на придумывание лучшего «черного ящика» с награждением победителя и т. п.

Вообще, эта игра нравится в любом возрасте, даже в 11 классах в нее играют с удовольствием. И если сначала «черные ящики» придумывает учитель, то потом ученики начинают предлагать «свои». Приходит перед уроком ученик и говорит: «Я придумал замечательный новый «черный ящик». А вы ему в ответ: «Ну хорошо, я в конце 3 минутки оставлю, и мы поиграем». И в конце урока встает этот ученик и исполняет роль «черного ящика», а весь класс отгадыва-

ет. Возникает даже некоторая соревновательная атмосфера, когда они придумывают все новые и новые «черные ящики».

Итак, первый параграф учебника концентрируется вокруг двух пунктов. Первое: информация есть последовательность байтов, битов, символов и не зависит от смысла. Второе: обработка информации есть преобразование таких последовательностей по формальным правилам, никакие «человеческие» категории типа «нравится», «не нравится», «считаю важным», «считаю не важным» и прочие не допускаются.

Ответы к упражнениям про «черные ящики».

Поскольку я столько раз повторил вам про важность упр. 11–13 («черные ящики»), приведу ответы к ним.

Упр. 11. а) алфавит; б) натуральный ряд: каждое следующее число на 1 больше предыдущего; в) степени двойки: каждое следующее число вдвое больше предыдущего; г) последовательность Фибоначчи: третий элемент есть сумма первого и второго, четвертый — сумма второго и третьего, пятый — сумма третьего и четвертого и т. д., каждый следующий — сумма двух предыдущих; д) берется натуральный ряд и выписываются цифры, т. е. до девяти идут сами числа, а дальше — цифры чисел 10, 11, 12, . . . , (1, 0, 1, 1, 1, 2, . . .); е) первый символ просто вычеркивается; ж) первые буквы слов: «один», «два», «три», «четыре», «пять», «шесть», «семь», «восемь», «девять», «десять» и т. д.;

з*) эта последовательность устроена так: каждое следующее число описывает распределение повторяющихся цифр в предыдущем. Например, первый член последовательности — это одна (1) единица (1), поэтому второй член будет 11. В числе 11 две (2) единицы (1), поэтому третий член равен 21. В числе 21 одна (1) двойка (2) и одна (1) единица (1), поэтому четвертый член равен 1211. В числе 1211 одна (1) единица (1), одна (1) двойка (2), а потом две (2) единицы (1). Поэтому, выписав подряд все цифры в скобках, мы получим число 111221. Соответственно, следующим будет 312211 и т. д. Обычно четвероклассники разгадывают этот «ящик» очень быстро, шестиклассники — с трудом, а десятиклассники — не разгадывают вообще.

Упр. 12. а) каждая буква заменяется на следующую по алфавиту: «а» на «б», «б» на «в», . . . , «м» на «н», . . . , «я» на «а»; б) число символов; в) считается число согласных плюс число

гласных; г) осуществляется перестановка (которая хорошо видна на «ЭВМ»): первый символ переставляется в конец.

Упр.13. а) число уменьшается на единицу; б) берется ближайшее (с недостатком) число, кратное трем (то есть из чисел 0, 1 и 2 получается 0; из чисел 3, 4 и 5 — 3; из чисел 6, 7 и 8 — 6; из чисел 9, 10 и 11 — 9 и т. д.); в) количество цифр в числе;

г*) эту трудную задачу придумали третьеклассники в Переславле-Залесском. Ответ здесь — количество «дырок» в цифрах числа. Например, в числе 687 три «дырки» — одна в «6» и две в «8». Количество «дырок» зависит от цифр и их написания, в «1» «дырок» нет, в «2» и «3» — тоже нет, в «4» (если она сверху «замкнута») — одна, в «5» — нет, в «6» — одна, в «7» — нет, в «8» — две, в «9» — одна.

§ 2. Электронные вычислительные машины

Этот параграф совсем маленький и не содержит ничего, кроме слов. Если ученик слышит их впервые, то потом ему легче будет услышать их во второй раз. Никакого иного содержания в параграфе нет. Тут имеется небольшая, краткая история вычислительной техники, как обычно, про Лейбница, Беббиджа и т. д. А потом говорится, что в состав любой ЭВМ входит память, процессор, клавиатура, монитор, они обмениваются между собой информацией и т. д. Этот урок обычно совмещают с выходом на ЭВМ. Там просто все показывают «в натуре» (это — клавиатура, это — монитор и пр.), а потом начинают работать с клавиатурным тренажером или еще с чем-то.

Вы можете задать ученикам прочесть этот параграф самостоятельно, а потом ответить на их вопросы, если они будут. Упражнения в этом параграфе, в основном, арифметические. Например, говорится, что оперативная память ЭВМ «Корвет» — 64 Кбайта. Спрашивается: сколько примерно страниц нашего учебника можно уместить в эту память. Понятно, надо сначала прикинуть, сколько примерно килобайт занимает одна страница, а потом разделить и получить ответ.

Разумно все эти задачи переформулировать для той ЭВМ, на которой класс будет работать. Смысл этих арифметических задач в том, чтобы ученики примерно запомнили некоторые характеристики ЭВМ. Если уж мы поставили цель сформировать у школьников какое-то представление о мире, в котором они будут жить, то они должны уметь прикинуть, помещается учебник на дискету или нет. То есть в процессе решения этих не слишком важных задач у учеников формируются какие-то связи между характеристиками ЭВМ и вещами, встречающимися в повседневной жизни.

В этом смысле, я повторяю, здесь нет ничего, никакой особенной методики, никакого особенного содержания, кроме новых слов. Во всех сжатых курсах этот параграф обычно опускается или задается для самостоятельного чтения, чтобы не занимать время.

§ 3. Обработка информации на ЭВМ

Этот параграф завершает «раскрытие» определения информатики, которое было дано в преамбуле. Уже изложены понятия «информация», «обработка информации», «ЭВМ». Теперь — и это тема третьего параграфа — «обработка информации на ЭВМ».

Отмечу всего два важных момента.

Первый — ЭВМ всегда работает по программе: сначала мы должны ***записать*** то, что надо сделать, и только потом ЭВМ написанную нами ***программу*** выполнит. Здесь в неявном виде, еще без исполнителей, вводится схема программного управления (с. 16 пособия). Но сама схема не рисуется и противопоставление с непосредственным управлением (за исключением самого начала параграфа) не делается.

Второй важный момент, — программа должна быть записана специальным образом — так, чтобы ЭВМ эту программу поняла, т. е. на так называемом «языке программирования». Приводится три примера: одна и та же программа, записанная на Бейсике, Паскале и на школьном алгоритмическом языке (с. 20 учебника). Поясняется, что сама программа также является информацией и, как всякая информация, может храниться в памяти ЭВМ и обрабатываться с помощью ЭВМ. Более глубоко это будет раскрыто в главе 2 и в п. 21.6 главы 3.

Итак, в § 3 вводятся понятия программы (алгоритма) и программной обработки информации на ЭВМ. Кроме этих ключевых для раскрытия определения информатики понятий, параграф содержит также важные с мировоззренческой точки зрения разделы 3.4, 3.5. Они посвящены часто обсуждаемой роли информации, информатики, ЭВМ в нашей жизни сейчас и в будущем. Это все изложено в разделах 3.4, 3.5. Поскольку учебник предельно краток, я этот материал повторю.

Информатика и ЭВМ в жизни общества

Говорится следующее. На ранних стадиях автоматизации производства, когда хотели, скажем, новое производство автоматизировать, делали новые станки-автоматы, и всякий раз при налаживании нового производства нужно было переналаживать или даже заново делать станки-автоматы. Это трудоемкая, дорогостоящая работа. С появлением ЭВМ понятие станка-автомата и его переналадки коренным обра-

зом изменилось. Станок стал разделяться на механическую часть, способную выполнять элементарные операции, например, резание, сверление, шлифовку, штамповку, и на управляющую часть (ЭВМ), которая командует, в каком порядке эти операции следует выполнять для изготовления необходимой детали. Тем самым единый станок-автомат был заменен на две составляющие: 1) станок, который делает элементарные операции, но не знает, какие детали он будет производить, и 2) ЭВМ, которая управляет этим станком в нужной последовательности, делает те или иные действия и выпускает ту или иную деталь.

Что это дает? Теперь, если мы хотим начать выпуск другой детали по той же технологии, скажем, штамповкой или резанием, то мы должны менять не станок, а программу, т. е. задать какие операции, в какой последовательности надо делать. Другими словами мы меняем только информационную часть производства, пишем новую программу, а материальная часть остается такой, какой и была. А завод начинает производить новые детали.

Таким образом, если раньше материальная и информационная составляющие были слиты вместе неразрывно и для изготовления новой детали нужно было что-то механически менять, переналаживать, изготавливать «в металле» новые станки, то сейчас составляющие разделились. Отдельно есть материальная составляющая, отдельно информационная.

Доля информационной части все возрастает, поскольку материальная не меняется — от нее ведь требуется только умение выполнять элементарные операции. Смена материальной составляющей нужна только при переходе на какую-то новую технологию. Мы можем от резания, штамповки перейти на порошковые технологии. Для такого перехода нужны новые станки, материальное что-то. Но если мы деталь вырезаем из листа металла, и станок у нас универсальный, может просто резать, то с помощью новой программы мы в состоянии вырезать любые детали. Происходит разделение информационной и материальной составляющей в мире. И, следовательно, появляется работа чисто в информационной области — написание программ.

Информационная индустрия

Растет и число работников, обслуживающих информационную компоненту производства. Если программа для производства какой-то детали раньше уже использовалась на данном предприятии, то ее просто нужно помесить в память управляющей станком ЭВМ (взяв из архива, например, с магнитного диска). Если готовой программы нет, то ее можно купить на другом предприятии или поручить изготовление программы какой-нибудь программистской фирме. Таким образом, программы становятся и ***средствами производства***, и ***товаром***. А значит, готовые программы надо где-то хранить, организовывать «компьютерные склады» — базы данных. Для этих складов требуются «кладовщики» — программисты, обслуживающие эти базы данных. Нужны «магазины программ» и квалифицированные продавцы; рабочие, которые будут создавать новые средства производства — программы; начальники, которые разбираются как в самом новом товаре — программах, так и в процессе его производства — программировании. Необходимы учебные заведения и преподаватели, обучающие новым информационным технологиям в производстве и других сферах жизни общества. Именно этот рост информационной индустрии и вызвал к жизни лозунг А.П. Ершова, вынесенный в заголок п. 3.5: «Программирование — вторая грамотность».

Запись ответа в виде алгоритма

Упражнения в конце параграфа во многом обращены к следующей главе учебника, к более глубокому пониманию алгоритма. Начинаются упражнения с задачи о «Волке, козе и капусте», с «поездов, которые должны разминуться на тупике» и т. д. Все они посвящены развитию алгоритмического стиля мышления. И, что самое главное, — в них во всех ответом является ***алгоритм***. Но, в отличие от игры в «черный ящик», здесь школьники уже должны этот алгоритм ***как-то записать***. Никакой фиксированной формы записи им при этом не предлагается. Форму записи в каждой задаче школьникам приходится придумывать самим. Скажем, решение задачи о «Волке, козе и капусте» они обычно записывают с помощью стрелок, используя какие-то короткие обозначения для волка, козы и капусты.

Главное, что учитель здесь должен требовать, — чтобы решение хоть как-то, было ***полностью записано*** на бумаге, чтобы не было двусмысленностей, неясностей и пр.

Это и будет пропедевтикой для введения алгоритмического языка. Именно тут должна возникнуть потребность в какой-то стандартизованной форме записи алгоритмов. И уже можно сказать, что бывают разные формы записи алгоритмов. Чем сложнее задача, тем сложнее форма записи, а если мы вдобавок собираемся поручить исполнение алгоритма ЭВМ, то нужен такой язык, который поймет ЭВМ. И в нашем курсе мы для этих целей будем использовать школьный алгоритмический язык.

Задача, которую нельзя решить

Я также хочу обратить ваше внимание на упр. 12. Это чрезвычайно сложная задача, которую, я думаю, решить нельзя. Она недаром помечена тремя звездочками. Ее решение существует, и его можно прочесть в трехтомнике Д. Кнута «Искусство программирования», том II [Кнут]. Задача, в каком-то смысле, провокационная. Придумана и вставлена по просьбе учителей, чтобы была задача, которую вообще нельзя решить (это бывает полезно в некоторых классах с особо одаренными, скажем так, «компьютерными» ребятами).

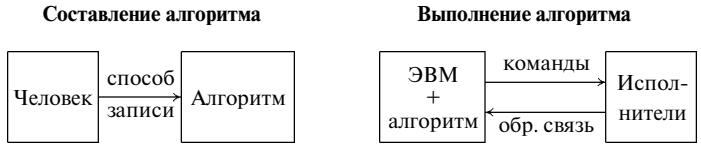
Мне не известен ни один человек, включая профессоров механико-математического факультета МГУ, который бы сумел додуматься до решения сам, а не прочел его в книге. Умом я, конечно, понимаю, что кто-то когда-то это решение все-таки придумал, но поверить в это не могу. Методическая значимость этой задачи в том, что задачи с очень простой формулировкой иногда имеют очень сложное содержательное решение. Вы можете рассматривать это, как пропедевтику к Роботу, поскольку там тоже будут достаточно сложные задачи с очень простыми формулировками.

Итак, если говорить о Введении в целом, то в нем есть два концентра: формальность понятий «информация» и «обработка информации» (параграф 1) и понятие программы (алгоритма) для ЭВМ (параграф 3). Кроме того, затрагиваются важные с мировоззренческой точки зрения темы о роли информатики (программирования) и ЭВМ в жизни общества.

§ 4. Исполнитель «Робот». Понятие алгоритма

Смысл параграфа сводится к введению схемы программного управления «Человек—ЭВМ—исполнитель», а также к введению алгоритмического языка, как способа записи плана будущих действий ЭВМ. Схему мы подробно обсуждали с вами, когда говорили про цели курса, про первого «кита», про развитие алгоритмического стиля мышления. Поскольку схема важная, давайте я ее снова нарисую, чтобы вы вспомнили:

Схема Человек—ЭВМ—Исполнитель



Чтобы ввести и объяснить эту схему, нужны какие-то конкретные исполнители. Таких исполнителей в учебнике два. В § 4 вводится исполнитель «Робот», а в следующем § 5 — исполнитель «Чертежник».

Я не буду повторять материал, который есть в учебнике, тем более, что я уже говорил про Робота, когда описывал главную цель нашего учебника — развитие алгоритмического стиля мышления учащихся.

Но я хочу подчеркнуть ряд важных моментов и объяснить, почему в учебнике сделано так, а не иначе.

Система команд Робота фиксирована, мы не можем «допридумывать» ему команды

Первое такое место на с.26 учебника. «Робот умеет выполнять всего 17 команд. Мы пока изучим 5. . .».

Здесь не случайно точно названо общее число команд Робота — 17. Опыт показывает, что это улучшает понимание «правил игры». Конечно, все понимают, что Робот — придуманное устройство. Но цифрой 17 мы с самого начала говорим: Робот уже ***есть***, у него есть 17 команд, их можно посмотреть на первом форзаце учебника. Нам пока из 17

нужно только 5 команд. Но произвольно вводить команды Робота нельзя, ведь они заложены в конструкцию Робота, фиксированы заранее.

Этот подход полностью противоположен подходу, при котором Робот «наделяется» новыми командами по мере надобности. Опыт показывает, что фразы типа «давайте введем еще несколько команд Робота» полностью запутывают ситуацию — так объяснять новые команды нельзя. Нужно с самого начала подчеркнуть, что Робот уже есть, все его 17 команд фиксированы (можно показать их на форзаце учебника), и сказать, что ***никаких других*** команд Робот выполнять не умеет.

Другими словами, мы можем выбирать только порядок изучения команд. Но система команд Робота задана и неизменна.

Это же, кстати, относится и к изучению школьного алгоритмического языка. С самого начала, с раздела 4.1, школьный алгоритмический язык вводится как нечто, уже существующее, уже заданное. Эта мысль постоянно подкрепляется замечанием, что программы на алгоритмическом языке можно выполнить на любой серийной школьной ЭВМ. (Вы можете добавить — и на любой ЭВМ, совместимой с IBM PC.) При изучении языка недопустимы фразы типа «давайте добавим в язык еще одну конструкцию», так как язык уже существует и мы можем менять только порядок изучения конструкций языка, но не сам язык.

Схема программного управления «Человек—ЭВМ—исполнитель» в § 4 только начинает вводиться. Далее формирование этой схемы продолжается в § 6 (пункт 6.6 — разделение труда между ЭВМ и исполнителями), в § 9 (9.1 — команды обратной связи; 9.2 — использование команд обратной связи при управлении Роботом вручную; 9.4 — диалог ЭВМ-Робот при выполнении цикла **пока**) и далее.

В § 4 рассматривается совсем простая схема управления без обратной связи и вводится она без особых пояснений. На первом этапе используется всего пять команд. Соответственно, можно представлять себе пульт ручного управления Роботом с пятью кнопками и 5 разных сигналов, которые ЭВМ посылает Роботу.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. Все это можно наглядно продемонстрировать на экране компьютера в гипертексте «Знакомство с Роботом» в системе «КуМир». На экране будут изображены и поле Робота, и пульт управления, и можно будет мышкой нажимать кнопки на пульте (см. рис. 2) и наблюдать результаты выполнения команд.

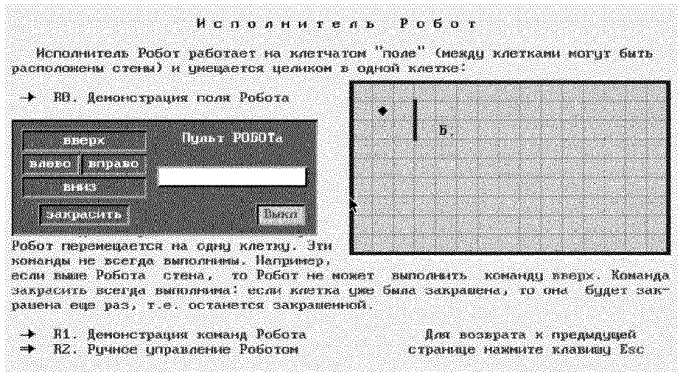


Рис. 2.

Поскольку учебник не содержит никаких обоснований, никакой мотивации для введения схемы программного управления Роботом, я приведу пару таких обоснований. Естественно, вы можете придумать, найти и использовать другие. Мои примеры будут задействовать команды обратной связи, но в этом месте их можно подробно не объяснять. А можно ввести и объяснить по ходу изложения.

Первый пример — космический

Предположим, мы посылаем Робота на какую-нибудь далекую планету. Пусть, мы — на Земле, а Робот — на клетчатом поле на Нептуне. Тогда, если мы сами командуем Роботом с Земли, то наша команда дойдет до Нептуна примерно за 4 часа, после чего еще столько же времени ответ будет идти назад. Скажем, мы нажали на кнопку «справа стена?», через 4 часа эта команда дойдет до Робота, Робот проверит, есть ли справа стена, и еще 4 часа ответ будет идти обратно на Землю. И только после этого мы сможем его проанализировать, принять решение, куда двигаться дальше и послать, соответствующую команду Роботу (которая еще 4 часа будет идти до Робота).

Получается такое неторопливое управление — меньше 6 команд в сутки. Могут быть ситуации, в которых такой темп недопустим. Кроме того, на Нептуне за эти 8 часов со стенками может что-нибудь случиться: одни стенки могут исчезнуть, а другие — возникнуть.

Чтобы управлять Роботом с нормальной скоростью, можно сделать следующее: вывести на орбиту вокруг Нептуна спутник и командовать Роботом со спутника. От спутника до Робота сигнал будет идти доли секунды — задержек не будет, и процесс ускорится.

Поскольку, однако, мы не собираемся держать на борту спутника экипаж для управления Роботом, на спутнике надо установить ЭВМ. Тогда мы с Земли сможем послать на спутник программу, алгоритм управления Роботом. Эта программа будет идти до спутника те же 4 часа, но зато потом ЭВМ сможет командовать Роботом без нашего участия, выдавая команды и получая ответы за доли секунды.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Можно, конечно, ЭВМ смонтировать и на Роботе. Но поскольку одна из наших целей — четкое разграничение того, что делает исполнитель, а что — ЭВМ (см. пункт 6.6 — разделение труда между ЭВМ и исполнителями), то предпочтительнее (если это возможно) держать ЭВМ «на некотором расстоянии» от Робота.

Конечно, это достаточно искусственное построение, хотя оно вполне адекватно отражает суть дела. Мы использовали это введение при преподавании программирования на мехмате МГУ в 1980 году (с луноходом, позже переименованным в «Путника»).

Второй пример — Чернобыльский

Можно привести много других примеров, демонстрирующих необходимость программирования, т. е. программного управления исполнителями. Например, Робот должен выполнить некоторую работу недалеко, но в сильно радиоактивной зоне, в которой человек находиться не может. А радиосвязь из-за больших помех тоже не работает. В таком случае нужно смонтировать ЭВМ прямо внутри Робота, за защитным экраном. И опять, еще до входа в опасную зону нужно будет поместить в ЭВМ программу, алгоритм управления Роботом.

Это объяснение хорошо согласуется с командами Робота по измерению температуры и радиации, задачами по радиационной разведке, закрашиванию радиоактивных клеток коридора и другими примерами учебника (см., например, алгоритмы «вход в радиоактивную зону» (А34), «разметка опасных клеток коридора» (А40), «радиационная разведка коридора» (А66) и др.).

Схему программного управления следует постоянно противопоставлять управлению Роботом «вручную»

Каковы бы ни были обоснования, главная задача учителя здесь — ввести схему программного управления и противопоставить ее непосредственному управлению исполнителями «вручную».

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Противопоставление схемы программного управления непосредственному управлению «вручную» — это и один из основных методических приемов при преподавании курса. При введении новых алгоритмических конструкций, при прохождении нового материала, да и просто при решении конкретной задачи один из базовых приемов — спросить у школьников: «Что бы вы делали, если бы работали вручную, если бы управляли Роботом сами?». Они отвечают: «Нужно нажать на такие-то кнопки, сделать то-то и так-то», т. е. разбираются с новой ситуацией. Дальше мы им говорим: «Хорошо, теперь у нас та же ситуация. Давайте составим ***алгоритм. Опишите***, что должна делать ***ЭВМ***, чтобы повторить то, что делали вы».

Такое противопоставление, такой переход является основным в курсе, и § 4 в этом смысле является важным. Он вводит обе схемы. И Робота, самого по себе, с понятными командами, и понятие алгоритма, записи программы для ЭВМ — в данном случае на школьном алгоритмическом языке.

Основная трудность параграфа

Основная трудность параграфа состоит в том, что пока у нас только пять команд, алгоритмы получаются уж очень простыми. Поэтому на старте, особенно если занятие проводится не за компьютером, может быть вообще непонятно, зачем эти алгоритмы как-то оформлять. Но, если мы хотим

поручить исполнение алгоритма ЭВМ, то следует (как бы это ни было просто) писать по правилам.

Первый алгоритм учебника, алгоритм (А1), тривиален. И постановка задачи, и решение очевидны — надо два раза скомандовать Роботу «вправо» и один раз — «вниз». Содержательно задача совершенно неинтересная. Поэтому тут придется на учеников нажать, сказать: «Привыкайте — это форма записи. Мы ведь не собираемся сами всю жизнь алгоритмы выполнять, а хотим это дело перепоручить ЭВМ. А ЭВМ подавай стандартную форму записи алгоритма, так что придется освоить язык, понятный ЭВМ».

Конечно, лучше просто все делать на ЭВМ. Все эти комментарии (4.5), ошибки (4.7) и запись нескольких команд в одной строке (4.8) на практике осваиваются за несколько минут. Вы можете также запустить гипертекст «Знакомство со школьным алгоритмическим языком» в системе «КуМир», и демонстрационная программа сама все покажет и даже потребует понажимать на кнопки, чтобы ученик освоил поскорее все технические детали.

Основная тема этого параграфа — переход от непосредственного управления Роботом к записи алгоритма. В данном случае — к записи на школьном алгоритмическом языке.

Ну а дальше идет самое важное — упражнения. Начнем с задач на анализ алгоритмов.

Умение рассуждать про алгоритм в общем виде, предсказывать результат его работы, глядя на текст алгоритма, — важная компонента алгоритмической культуры

В задачах 1а) и 1б) ученик должен выступить в роли ЭВМ и выполнить алгоритм. Попутно ему придется сделать кое-какую чисто человеческую работу: придумать названия алгоритмов и комментарии. Эти задачи лучше решать без ЭВМ, иначе она возьмет на себя ту работу, которую должен выполнить ученик.

Задачу 2 тоже лучше делать на бумаге или обсудить у доски, чтобы не отвлекаться на нажатия кнопок.

Задачи 3а) и 3б) требуют мысленного выполнения алгоритма, они довольно простые.

Задача 4 очень хорошая, я ее настоятельно рекомендую. Это упражнение на формальность исполнения и на анализ программы независимо от того, куда и как ходит Робот. Эта

задача и на восприятие (анализ) того алгоритма, который приведен в учебнике, и на составление нового алгоритма. В ее условие и решение вовлечено большинство понятий, введенных в параграфе. Прежде чем, задавать эту задачу на дом, полезно обсудить ее условие у доски.

Задачи 6, 7 на стр. 29 и 7–12 на стр. 31–32 (из-за ошибки в нумерации в учебнике встречается два упражнения с номером 7 и два упражнения с номером 8) имеют формально-логический характер. Их цель — воспитать представление об алгоритме, как о некоторой информации, которую можно исследовать и перерабатывать. Попутно, работа над этими задачами заставляет продумать и команды Робота, и понятие алгоритма, и схему программного управления. Скажем, задача 11 заставляет задуматься о всевозможных алгоритмах управления Роботом, о том, что из каждого из них получится при перестановке местами двух команд, и о многом другом.

Задачи, начиная с 7 на с. 31, и дальше, не вполне алгоритмические, я бы сказал, что они относятся к «комбинаторике на клетчатом поле Робота».

Задачи на составление алгоритма

Два семейства задач — 5а)–и) и 8а)–з) (с. 31) — посвящены составлению алгоритмов. Их лучше делать на ЭВМ. Эти задачи построены на одной идее: алгоритмы их решения состоят из нескольких повторяющихся одинаковых кусков. Используя правило записи нескольких команд в одной строке, каждый такой кусок можно уместить в одну строку. Тогда алгоритм будет содержать несколько одинаковых строк, подобно алгоритму (А2) на с. 28 учебника. Например, для задачи 8ж) алгоритм будет содержать пять одинаковых строк.

```
алг 8ж
  дано | Робот в клетке А (см. рис. 8<<ж>>, стр. 31 уч-ка)
  надо | Робот в клетке Б (см. рис. 8<<ж>>, стр. 31 уч-ка)
нач
| закрасить
| вниз;вниз;вправо;вверх;закрасить;вверх;закрасить
| вниз;вниз;вправо;вверх;закрасить;вверх;закрасить
| вниз;вниз;вправо;вверх;закрасить;вверх;закрасить
| вниз;вниз;вправо;вверх;закрасить;вверх;закрасить
| вниз;вниз;вправо;вверх;закрасить;вверх;закрасить
кон
```

При записи на бумаге (в тетрадях) или на доске ученики обычно ленятся пять раз повторять одну и ту же строку. Чаще всего они записывают одну строку, справа рисуют фигурную скобку и пишут «5 раз». Либо придумывают что-то аналогичное. За любую такую форму записи их нужно хвалить, заметив, что позже мы изучим строгую форму записи повторяющихся кусков алгоритма — цикл «N раз».

Если же ученики будут работать на ЭВМ в системе «КуМир», то, набрав повторяющуюся строку в первый раз, они смогут ее «запомнить» (нажав клавишу F3), а затем «вспомнить» нужное число раз (нажимая клавишу F4). Так что набирать одну и ту же строку несколько раз им тоже не придется.

На самом деле здесь, в виду легкости материала параграфа, все основные упражнения сделаны на выделение повторяющихся элементов. Вся их сложность состоит в том, что в них во всех надо выделить повторяющийся фрагмент и его записать. Где-то получается это попроще, где-то посложнее. Где-то надо отдельно записать начальные или конечные нерегулярности (например, в приведенном выше алгоритме «8ж» требуется отдельно закрасить первую клетку).

В целом же эти упражнения можно рассматривать как пропедевтику понятий вспомогательного алгоритма и цикла.

§ 5. Исполнитель «Чертежник» и работа с ним

Сам по себе Чертежник — это обычный графопостроитель, типичное широко распространенное устройство ЭВМ, используемое в инженерии и в науке. Он умеет поднимать и опускать перо, ставить его в заданную точку и прямолинейно перемещаться из заданной точки в другую. Путем такого опускания и перемещения пера можно рисовать картинки, составленные из отрезков. Запустите гипертекст «Знакомство с исполнителем «Чертежник» в системе «КуМир» — и вы увидите примеры работы этого исполнителя.

В принципе, можно пройти весь курс и без исполнителя «Чертежник», пропустив все, что с ним связано. Введение исполнителя объясняется несколькими соображениями, которые я сейчас изложу.

Команды с аргументами

Первое — возможно более раннее появление команд с аргументами. Команды Робота («вправо», «влево» и пр.) — это простые команды без всяких аргументов. У команд Чертежника есть аргументы. Ему можно скомандовать поставить перо в точку с координатами, скажем, (1,3) или сместиться на вектор с координатами (1,1). Чтобы показать команды с аргументами, а точнее говоря, чтобы к ним привыкнуть прежде, чем они понадобятся по существу, и введен Чертежник.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Подчеркну еще раз — в нашем курсе до § 11 нет «величин». Нет описания «величин», нет оператора «присваивания». Весь курс строится на управлении Роботом и Чертежником. Более того, вспомогательные алгоритмы вводятся и объясняются раньше всех управляющих конструкций, даже раньше простейших «циклов», скажем, цикла «5 раз». И управляющие конструкции, и величины появляются позже. Это существенное отличие нашего курса от большинства других. И именно Чертежник позволяет ввести и использовать вспомогательные алгоритмы ***с аргументами*** до начала работы с величинами и даже до команд ветвления и повторения.

Забегая вперед, хочу заметить, что вспомогательные алгоритмы (§ 6) были введены до управляющих конструкций с вполне определенной целью. Опыт показыва-

ет, что первые понятия в новой области запоминаются гораздо глубже, чем последующие (в психологии это, кажется, называется «imprinting», что в переводе означает «впечатывание», «запечатление»). А одна из распространенных проблем школьных (да и многих других) курсов состоит в том, что школьники не усваивают понятие «вспомогательный алгоритм» и часто пишут алгоритмы «одним куском». Это и понятно, — в иерархии основных понятий алгоритмизации (см. с. 32 пособия) «вспомогательные алгоритмы» стоят во второй колонке, а циклы и величины — в первой.

Вот почему мы постарались построить курс так, чтобы понятие вспомогательного алгоритма «впечаталось» хотя бы за счет отсутствия других конструкций на старте. При нашем построении курса понятие вспомогательного алгоритма с параметрами появляется сразу после понятия алгоритм. То есть с самого начала — до всего остального — вводятся средства структурирования алгоритмов. Поэтому и появился Чертежник, для которого потом можно будет составлять вспомогательные алгоритмы с аргументами. Скажем, алгоритм рисования квадрата произвольных размеров.

Второе соображение для введения Чертежника состоит в том, что, рассказывая про команды с аргументами и позже про вспомогательные алгоритмы с аргументами, можно устроить пропедевтику понятия «величина», т. е. объяснить, как ЭВМ запоминает аргументы алгоритма, привести модель памяти ЭВМ и т. д. В этом параграфе это все в зачаточном состоянии — сколь нибудь связные объяснения и модель памяти ЭВМ появятся в следующем параграфе. Но желательно, чтобы к этому времени какие-то слова и понятия в головах учеников — пусть в зачаточном состоянии — уже были.

Исполнителей должно быть несколько

Есть еще один методический аспект, о котором я должен упомянуть. Мы считаем, что исполнителей в школьном курсе должно быть не меньше двух. Просто, чтобы материал не был связан только с одним исполнителем. Чтобы в схеме программного управления можно было нарисовать, что ЭВМ управляет и Роботом и Чертежником. Чтобы можно было написать алгоритм, управляющий двумя исполнителями одновременно. Например, алгоритм, при выполнении которого Робот идет по коридору, измеряя уровень радиоактивности, а Чертежник строит соответствующий график радиоактивности. То есть, чтобы у нас появились алгоритмы, управляющие более, чем одним, исполнителем. Иначе очень трудно понять, зачем ЭВМ отделена от исполнителей и кто же, собственно говоря, понимает программу: Робот или ЭВМ.

Если вы читали книги Г.А. Звенигородского или самые первые школьные учебники информатики А.П. Ершова, то, наверное, помните, что там исполнители «обучались» командам языка (**`если`**, **`пока`** и т. д.). Каждый исполнитель «знал» и понимал весь язык программирования, а никаких ЭВМ не было вообще. Если вам нужно было сделать что-то двумя исполнителями, то вы должны были конструировать нового, который включал бы оба предыдущих и знал весь язык.

У нас этой путаницы нет. Исполнитель имеет только те кнопки, которые имеет. Робот и Чертежник исполняют ***команды***, а не ***программы***. А со всеми ***алгоритмами***, со всеми ***программами*** разбирается ЭВМ. Именно ЭВМ понимает алгоритм, и командует — управляет исполнителями. И для прояснения этого нужны, как минимум, два исполнителя.

Наконец, некоторые педагоги-практики рекомендовали нам ввести не двух исполнителей, а гораздо больше. В программном обеспечении мы это сделали. В базовом комплекте системы «КуМир» кроме Робота и Чертежника есть еще упомянутый во введении Вездеход, который перемещается по непрерывной плоскости, Двуног, Строитель, Редактор и ряд других исполнителей (см. цветную вкладку учебника). Эти исполнители успешно использовались на кружках и факультативах, в летних школах программирования и т. д. Но в основном тексте учебника, опираясь на принцип «ничего лишнего», мы ввели только двух исполнителей, поскольку их оказалось достаточно. Учитель при желании может использовать в некоторых темах и других исполнителей, но никаких конкретных рекомендаций я вам не дам. Главное — помнить, что наша цель — развитие алгоритмического стиля мышления, а не изучение исполнителей самих по себе. Если интересные исполнители могут помочь в достижении главного — их надо использовать, если лишь отвлекают и занимают время — от них следует отказываться.

Алгоритмы рисования букв — это подготовка к введению вспомогательных алгоритмов

Для подготовки к введению в следующем параграфе вспомогательных алгоритмов здесь разбираются алгоритмы рисования букв (алгоритм А7, рис. 12–13 учебника). Основное, на что надо обратить внимание: алгоритмы составлены таким образом, чтобы при их последовательном выполнении получалось слово, т. е. буквы шли бы в ряд с некоторым промежутком между ними. Вопрос о том, кто и как организует это последовательное выполнение, в § 5 намеренно не поднимается. Фактически он откладывается до § 6, чтобы дать школьникам возможность сперва освоиться с Чертежником.

Рисование букв можно заменить чем-то другим, но с буквами легко объяснять, почему к алгоритмам предъявляются такие требования. Ясно, зачем нужно рисовать букву за буквой. Ясно, почему важно уметь рисовать букву с любого места, — чтобы можно было писать потом из этих букв слова, и т. д.

Графические пакеты

В принципе, если не строго следовать учебнику, то Чертежника можно заменить на какой-нибудь графический пакет, который умеет закрашивать кружки, прямоугольники и прочее. И рисовать разные красивые картинки (именно так обычно и преподают на Бейсике). Впрочем, красивые картинки можно рисовать и в системе «КуМир», используя школьный алгоритмический язык и графический пакет (исполнитель) «ГраТекс», умеющий рисовать и закрашивать все эти прямоугольники, круги, эллипсы, дуги и пр. По содержанию это будет то же, что и Чертежник, но картинки на компьютере получаются более красивыми и привлекательными.

Упражнения

Поскольку ничего нового, кроме самого Чертежника, в § 5 не появилось, упражнения оказываются похожими на те, что были для Робота в § 4. Часть — на анализ алгоритмов, часть — на составление алгоритмов. Правда, при составлении алгоритмов можно на экране получить какие-то картинки, что обычно нравится.

Недостаток сложности алгоритмов (как и в предыдущем параграфе) компенсируется дополнительными условиями из других областей. Например, в упражнении 9 предлагается рисовать картинки, не отрывая пера от бумаги и не проводя линий дважды, т. е. сначала решить топологическую задачу, а потом составить алгоритм ее решения.

Из конкретных упражнений я, прежде всего, обращаю ваше внимание на начинающиеся со слов «нарисовать какой-нибудь». Например: «Нарисовать какой-нибудь квадрат со сторонами, не параллельными осям координат». Необходимость в них вызывается тем, что при решении большинства задач в информатике правильных ответов бывает очень много. А школьники, да и учителя привыкли к тому, что правильный ответ обычно только один. Чтобы от этого отучить, мы дали несколько задач, в которых правильных ответов заведомо много. Сначала это оказывается трудным — идея, что в задаче может быть много разных правильных решений и нужно придумать только одно из них, для школьника достаточно нова. Но как только эта идея понята на одном примере, все остальные подобные задачи уже не вызывают затруднений.

Кстати, «квадрат со сторонами, не параллельными осям координат», как оказалось, — не самая простая задача. И вы можете первоначальную растерянность и последующее «облегчение» учеников превратить в шутку, разрядить атмосферу в классе.

Упражнение 7 — рекламное. Дело в том, что построить графики приведенных в ней функций не так-то просто. А не зная графика мы пока ничего нарисовать не можем. Продемонстрируйте сложность этой задачи ученикам и порадуйте их тем, что в § 12 они познакомятся с алгоритмом рисования произвольной функции. Так что эта задача просто реклама алгоритма, который в курсе появится позже.

Упражнения параграфа (да и вообще главы 1) можно разбить на несколько типов:

- задачи на выполнение готовых алгоритмов: написан готовый алгоритм, нужно его выполнить, описать результат его выполнения (таковы упр. 2, 11б);
- задачи на понимание алгоритмов: дан алгоритм, требуется понять, что он делает, написать **дано**, **надо** и т. п.
- задачи на модификацию алгоритмов, например, упр. 3, 10, 11в, 11г, когда готовый алгоритм надо переделать так, чтобы он делал что-то новое (например, чтобы буквы стали в два раза больше);

— и лишь после этого по сложности идут задачи на составление алгоритмов, когда ничего нет и надо составить алгоритм (упр. 4, 5, 6, 8).

Подбирая упражнения к уроку, желательно использовать задачи из разных типов. Скажем, «выполнить такой-то алгоритм», «изменить такой-то алгоритм», «составить такой-то» и т. д. с тем, чтобы различные типы задач были охвачены.

Кроме того, я обращаю ваше внимание на задачи типа упр. 11, которые я бы назвал «на анализ алгоритмов». В них надо что-то сказать про алгоритм, не выполняя его, а глядя на текст. Как это сделать? Например, в упр. 11 последняя строчка «поднять перо», значит перо будет поднято после выполнения алгоритма, независимо от всего предыдущего. Предпоследняя строчка «сместиться в точку (2,1)», следовательно, перо окажется в точке (2,1) независимо от всего предыдущего. Такие навыки анализа алгоритмов крайне нужны для их понимания, анализа, отладки, поиска ошибок и т. д. Этот класс задач, когда, глядя на текст, надо что-то сказать про алгоритм, очень полезен.

Анализ может быть и более сложным. Например, в упражнении 12, чтобы определить x-координату конечного положения пера после выполнения алгоритма, надо просуммировать первые аргументы всех команд алгоритма.

Наконец, задачи 13, 14 (вы сами можете составить аналогичные задачи) посвящены придумыванию новых исполнителей. Эти задачи должны снять ореол исключительности с Робота и Чертежника и показать, что исполнителей можно придумывать самим. Если вы дали школьникам такую задачу на дом, то соберите решения в письменном виде заранее и запланируйте как минимум полурока на обсуждение. Либо, если считаете нужным, сами можете рассказать про других исполнителей, например, про «Вездеход», «Двуног», «Строитель», «Гратекс» из комплекта «КуМир».

Лучше всего идет «Двуног». У него есть две ноги, длинная-длинная шея и голова без туловища. Двуног может двигать ногами и наклонять голову. Если Двуног оторвет от земли одну ногу и наклонит голову, то он начнет медленно падать, пока не упрется в землю ногой. Задача состоит в том, чтобы научить его ходить, сначала по ровному месту, потом по лесенкам, потом в пещере, где нельзя выпрямиться, и т. д.

Можно заниматься всем этим, разнообразить уроки, но если остаться в рамках учебника, то задачи 13 и 14 или обсуждение каких-то других исполнителей становятся необходимыми.

Я также рекомендую дополнить набор упражнений учебника упражнением следующего содержания: «Придумайте какого-нибудь исполнителя, все равно какого, выпишите систему его команд и составьте два разных алгоритма для него». Эта задача почему-то очень привлекательна для школьников: что-то среднее между сочинением сказок и научно-фантастических рассказов. Все позволено, никаких ограничений нет, и школьники начинают с энтузиазмом придумывать самых разнообразных исполнителей. Среди таких придуманных исполнителей, запомнившихся мне, был исполнитель «Бог», с командами «создать твердь», «создать хлябь», «отделить твердь от хляби» и т. д. Все это носит характер художественного творчества. Тем не менее, на этом художественном фоне и в такой художественной обстановке можно и нужно обсудить ряд полезных формальных понятий, из которых основное — система команд исполнителя, т. е. набор команд, которые исполнитель умеет выполнять. Прежде всего необходимо, чтобы это был четкий набор, в котором должно быть понятно, как выполнение команды начнется, как команда будет выполнятся, что получится в результате и, наконец, когда выполнение команды закончится.

Наконец, даже в четкой системе команд, у школьников часто встречается такая ошибка: они придумывают исполнителя, командами которого можно воспользоваться только в одной последовательности, т. е. исполнителя, который, фактически, умеет делать только одно дело. Другими словами, школьники разбивают это одно дело, скажем, на 5–7 этапов и пишут систему команд типа «выполни этап 1», «выполни этап 2», ... «выполни этап 7». Для такого исполнителя есть только одна разумная программа, и он совершенно не интересен. Его с успехом можно заменить исполнителем с одной единственной командой, который по этой команде выполняет все 7 этапов сразу. Именно поэтому нужно требовать от школьников написать ***два*** существенно разных алгоритма, а не один. Я обычно начинал обсуждение этого упражнения в классе, потом задавал на дом, и зачитывал лучшие работы на уроке. Получается довольно интересно и доставляет большое удовольствие и ученикам, и учителю.

В целом же § 5 мало чем отличается от § 4, просто вводится другой исполнитель. Основная разница — команды с аргументами. Это введено в качестве пропедевтики и из некоторых педагогических соображений.

Лекция 4.

§ 6. Вспомогательные алгоритмы и алгоритмы с аргументами

Это первый содержательный параграф учебника. В нем излагается понятие вспомогательного алгоритма — третьего фундаментального понятия информатики в классификации, которую я вам приводил (см. с. 32).

В отличие от большинства других содержательных параграфов, здесь применяется не проблемный подход, а стандартный: понятие сначала вводится, а потом используется для решения задач. Все начинается, как обычно, с задачи (составить алгоритм для написания слова «МИР»), но мы не просим учеников самих придумывать конструкции для ее решения, — потому что, если попросить учеников составить такой алгоритм, то они просто напишут достаточно длинный алгоритм «одним куском». Поэтому здесь мы прибегаем к обычному стилю — сначала учитель рассказывает и объясняет, а потом школьники решают задачи. Такой стиль, такой подход я обычно называю «догматическим», поскольку ученики должны воспринимать то, что рассказывает учитель, как некоторую догму, как нечто заданное и неизменное, без особенных обсуждений, почему это именно так, а не иначе.

Логически параграф делится на пять частей:

(1) вспомогательные алгоритмы без аргументов (пп. 6.1–6.4);
(2) метод последовательного уточнения (п. 6.5);
(3) разделение труда между ЭВМ и исполнителями (п. 6.6);
(4) вспомогательные алгоритмы с аргументами (пп. 6.7–6.8);
(5) модель памяти ЭВМ (п. 6.9).

В гипертексте «Вспомогательные алгоритмы» в системе «КуМир» есть демонстрационные примеры и задачи ко всему материалу этого параграфа.

Вспомогательные алгоритмы без аргументов

Ключевые понятия, которые здесь вводятся, — это понятия «вспомогательного алгоритма», «вызова» вспомогательного алгоритма, «основного алгоритма», а также понятие ***относительности*** отношения вспомогательного и основного алгоритмов.

Последнее чуть более сложно, но вы легко можете эту сложность снять, приведя бытовой аналог. Например, сказав, что основной и вспомогательный алгоритм — как отец и сын (т. е. речь идет об отношении ***между*** двумя алгоритмами). Это ***роли*** алгоритмов. Один и тот же алгоритм может быть вспомогательным для одного алгоритма и основным для другого (см. в п. 6.3 учебника).

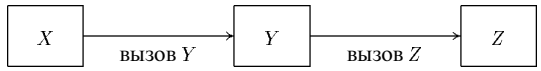
Чтобы подчеркнуть важность понятия вспомогательного алгоритма, в учебнике дается высоко научное определение в самом общем виде:

«В общем случае, если в записи алгоритма X встречается вызов алгоритма Y, то алгоритм Y называется вспомогательным для X, а алгоритм X называется основным для Y.»

Догматический подход нуждается в подобного рода опорных точках: определение можно продиктовать, заставить выучить наизусть, повесить на большом плакате в кабинете информатики. Но вы должны понимать, что это будет всего лишь «реклама» понятия «вспомогательный алгоритм», демонстрация его важности. Освоение понятия может произойти только через «делание» — в ходе практической работы, в ходе самостоятельного решения задач.

В учебнике демонстрация понятий вспомогательного и основного алгоритмов базируется на примере рисования слов: вспомогательные алгоритмы «буква М», «буква И» и «буква Р» вызываются из основного алгоритма «МИР». В предыдущем параграфе был еще и алгоритм «буква У», что позволяет составлять разные основные алгоритмы: «МИРУ МИР», «РИМ», «МИМ», «УМРИ» и т. д.

Очень важным, повторю, является то, что на самом алгоритме не написано, основной он или вспомогательный, т. е. понятие вспомогательного алгоритма относительно. Если смотреть на один отдельно взятый алгоритм «сам по себе», то нельзя сказать, основной он или вспомогательный. Он — алгоритм. И только при его использовании он может становиться основным или вспомогательным ***по отношению к какому-то другому алгоритму.*** Например, Y может быть вспомогательным по отношению к X. Но если внутри Y мы вызываем Z, то Z будет вспомогательным для Y, а Y — основным для Z. Если рисовать в виде стрелочки, что X вызывает Y и Y вызывает Z, то получится такая картинка:



В вызове $X \longrightarrow Y$ X является основным, а Y — вспомогательным, но в вызове $Y \longrightarrow Z$ Y является основным, а Z — вспомогательным. Таким образом, Y является вспомогательным ***по отношению*** к X, но основным ***по отношению*** к Z. Другими словами, Y является и вспомогательным, и основным по отношению к разным алгоритмам. Эта ситуация разобрана в учебнике в п. 6.3.

Изложение вспомогательных алгоритмов без аргументов завершается практическим примером в п. 6.4. Это уже достаточно хороший пример, потому что без вспомогательных алгоритмов его решение получится очень длинным. Здесь Роботу нужно довольно много ходить, много всего закрашивать — писать «одним куском» все команды скучно, длинно и нудно. И это сделано намеренно. Конечно, задачу можно решить «одним куском», но использование вспомогательных алгоритмов сокращает объем писанины раза в три. Так что в этом примере вспомогательные алгоритмы впервые приносят хоть какую-то пользу.

При объяснении материала желательно попросить учеников учебники закрыть, нарисовать на доске копию рис. 18 учебника и, глядя на рисунок, всем вместе (под руководством учителя!) написать алгоритм А13 учебника. В крайнем случае (если «не пойдет») учитель может написать его сам.

А после этого, можно очень содержательно обсудить с учениками, как писать вспомогательные алгоритмы «закрашивание блока» и «обход стены». Напомню, что к этому моменту на доске есть общая картинка, что нужно закрасить, и есть основной алгоритм «из А в Б с закрашиванием». А вспомогательные алгоритмы еще только предстоит написать.

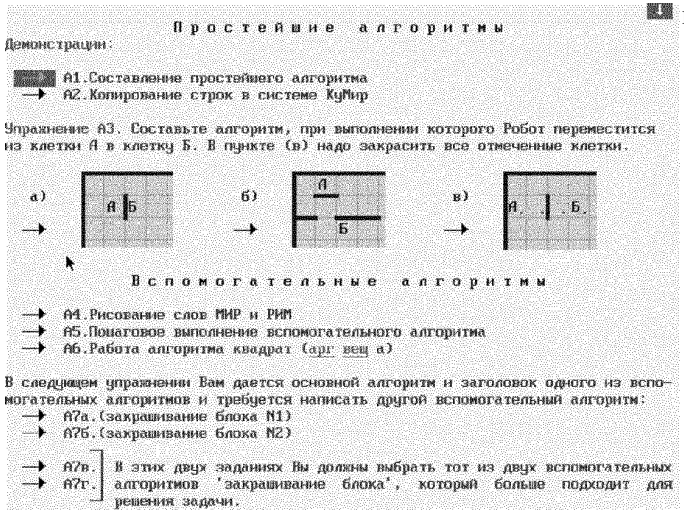
Тут нужно убедить учеников, что ***сначала*** необходимо описать в **дано** и **надо**, что должен делать каждый из вспомогательных алгоритмов, и проверить, что написанные алгоритмы подходят к основному алгоритму А13. И только ***после этого*** следует приступать к написанию тел вспомогательных алгоритмов — последовательностей команд для Робота. Чрезвычайно важно научить учеников различать и разделять эти два этапа: при ***использовании*** алгоритма читать, что написано в его **дано** и **надо** (не читая тело алгоритма), а при ***написании*** алгоритма — записывать в его **дано** и **надо**, что делает алгоритм.

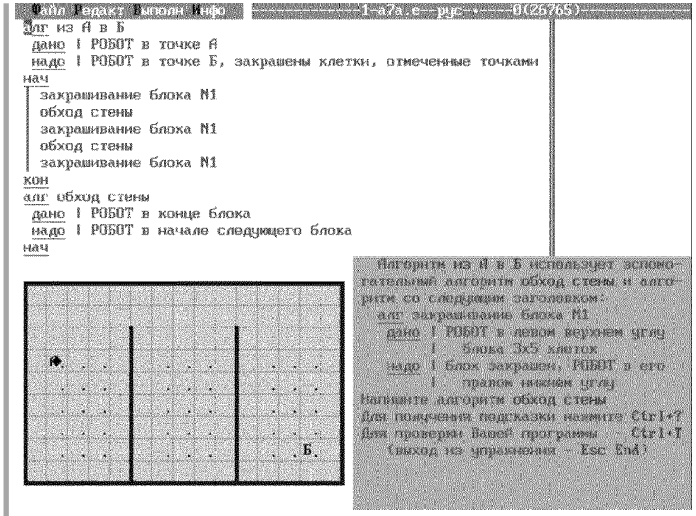
Умение различать и разделять эти два этапа — (а) использование алгоритма и формулировка его **дано** и **надо**; (б) составление алгоритма с заданными **дано** и **надо** является основным навыком, необходимым для метода последовательного уточнения (п. 6.5 учебника — см. ниже), да и вообще для составления алгоритмов с использованием вспомогательных алгоритмов.

Кроме того, я настоятельно рекомендую вам прочесть с. 44–50 нашего вузовского учебника [ПдМ], где подробно (и гораздо глубже, чем в школьном учебнике) изложена роль **дано** и **надо** в процессе составления программ.

И последнее замечание. Помните, мы с вами обсуждали, что в информатике задача может иметь несколько правильных решений. Алгоритмы А14 и А15 учебника дают только одно из них. Могут быть и другие. Например, мы можем изменить алгоритм «закрашивание блока» так, чтобы Робот после закрашивания оказывался в правом верхнем углу блока. При этом, правда, нам придется изменить и алгоритм «обход». Эти два алгоритма работают в паре, и их работа должна быть согласована. На такую согласованность можно придумать специальные задачи, например, задать один из алгоритмов пары и попросить составить второй. Заметьте, что для задания алгоритма достаточно привести его **дано** и **надо**, а тело алгоритма выписывать не нужно.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. Идея задания одного алгоритма из пары «закрашивания блока» и «обход» и задача на составление второго — реализована в гипертексте «Вспомогательные алгоритмы» в системе «КуМир». Ниже приведены копии экрана, соответствующие го-





МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я считаю, что пока использование вспомогательных алгоритмов не будет практически усвоено, дальше двигаться нельзя. В этом месте можно порешать уже довольно много разных задач — и их следует решать до тех пор, пока материал не будет усвоен и освоен. Контролировать освоение материала, естественно, следует также на задачах. Можно провести контрольную на бумаге с проверкой учителем вручную, можно — на компьютере с проверкой учителем или с автоматизированной проверкой решений компьютером в системе«КуМир» (такая контрольная входит в комплект гипертекстов системы «КуМир»).

Метод последовательного уточнения

Это — один из важнейших методов алгоритмизации, да и человеческого мышления вообще, очень распространенный и широко используемый. Здесь он изложен не слишком глубоко, — скорее декларативно. Но ввиду важности метода ему посвящен отдельный раздел.

Метод состоит в следующем. Если нам надо нечто сделать, то мы можем разбить требуемое действие на крупные шаги и ***сначала*** записать общий план действий в терминах

этих крупных шагов (как-то назвав их). В момент записи общего плана мы можем не знать, как конкретно проводить эти крупные шаги (как их реализовать, т. е. записать). Нам достаточно предполагать, что они выполнимы, что ***как-то*** нам их ***потом*** записать удастся.

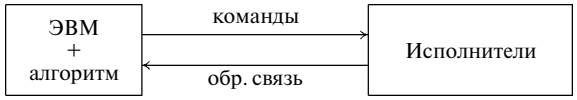
Основной выигрыш тут в том, что после записи общего плана над каждым крупным шагом можно думать независимо. Если проводить аналогию с математикой, то это в точности, как запись доказательства теоремы с использованием еще не доказанных лемм. Если леммы потом удастся доказать, то мы получим полное доказательство. Но над каждой леммой можно думать отдельно и независимо.

Ничего более глубокого про метод последовательного уточнения в пункте 6.5 не сказано. Но я вам хочу дать одну простую и легко выполнимую методическую рекомендацию — во всех случаях, когда вы вводите новый алгоритм, сначала запишите только его название и **дано**/**надо**, а уж потом, после паузы, приступайте к телу алгоритма. Это и будет пропедевтика «метода последовательного уточнения».

Разделение труда между ЭВМ и исполнителями

Тут нам нужно снова вернуться с схеме программного управления, к этапу выполнения алгоритма:

Выполнение алгоритма



Теперь, с учетом освоенных нами вспомогательных алгоритмов, мы должны понимать, что текст, который мы передаем для исполнения ЭВМ, может состоять из нескольких алгоритмов: основного, вспомогательных для этого основного, вспомогательных для вспомогательных и т. д. Именно такой «сложный» текст, состоящий из многих алгоритмов, мы отдаем ЭВМ в схеме программного управления. А уж потом ЭВМ выполняет этот текст и командует одним или несколькими исполнителями (например, Роботом и Чертежником).

Самое важное здесь — сказать, что это новое понятие вспомогательного алгоритма, относится к левому квадрати-

ку на рисунке («ЭВМ + алгоритм»). Робот каким был, таким и остался. В его конструкции ничего не изменилось. Он — всего лишь ***исполнитель команд***. А вспомогательные алгоритмы — это способ закодировать указания для ЭВМ, конструкция языка программирования.

Язык программирования должен знать Человек, составляя алгоритм. Язык программирования должна знать ЭВМ, чтобы выполнить алгоритм так, как написал Человек. С точки зрения ЭВМ выполнение алгоритма состоит в том, что она его анализирует и посылает команды Роботу, получая от него в ответ некоторую информацию, обратную связь. Робот же ни про какие алгоритмы и вспомогательные алгоритмы ничего не знает и лишь выполняет последовательно поступающие от ЭВМ команды.

Я так долго говорю одно и то же, потому что это разделение труда между ЭВМ и исполнителями — ключевой момент в понимании общей картины мира. Если это разделение труда будет осознано школьниками, то им дальше будет легко вплоть до величин (до § 11). Конечно, при прохождении цикла **пока** возможны некоторые сложности, но незначительные. Если у ученика в голове будет правильная модель, кому адресованы управляющие конструкции, что делает ЭВМ, а что — исполнители, то все остальное приложится.

Например, как только усвоено, что Робот не понимает ничего, кроме своих команд, легко разобраться в цикле «*N* **раз**», в размещении нескольких команд в одной строке и во многом другом. Кто понимает и выполняет цикл «*N* **раз**»? Конечно, ЭВМ. Именно ЭВМ, встретив

```
нц 5 раз
| вправо; закрасить; вниз
кц
```

понимает, что строку «вправо; закрасить; вниз» нужно выполнить 5 раз, что в этой строке записаны 3 команды Робота, разделенные точками с запятой, и т. д.

После пп. 6.5 и 6.6, естественно примыкающих к первой части параграфа, идет своего рода «водораздел». Вы можете, если хотите, в этом месте сделать остановку, порешать задачи с применением вспомогательных алгоритмов (на бумаге или на компьютере) и только потом двигаться дальше.

Вспомогательные алгоритмы с аргументами

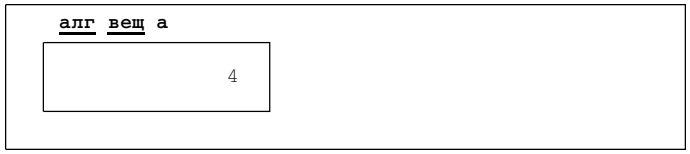
При введении алгоритмов с аргументами нужно опереться на здоровое чувство лени, которое есть и у каждого ученика, и у человечества в целом. Заставьте учеников написать на бумаге решение задачи 4,б) с. 37 учебника и сразу же решение задачи 4,в). Разумеется, школьники вам скажут: «Это то же самое, только нужно всюду вместо 4 написать 6». Тут можно спросить: «А как насчет длины 7 или 8?» и сказать, что во всех случаях, когда возникает подобного рода рутинная, никому не нужная деятельность, люди придумывают какие-то способы, как ее обойти. Для данного случая в школьном алгоритмическом языке предусмотрена конструкция вспомогательного алгоритма с аргументами.

И далее вы можете рассмотреть алгоритм А16 учебника, а затем пояснить в духе п. 6.8, как записывается и как выполняется вызов этого алгоритма.

А вот дальше возникает выбор. Можно ограничиться объяснением в пункте 6.8 и перейти к следующему параграфу. А можно ввести модель памяти ЭВМ и дать более глубокое объяснение. Против такого углубления то, что в данный момент (и до § 11) без модели памяти ЭВМ можно обойтись. За — то, что модель памяти ЭВМ позволяет объяснить работу с аргументами более глубоко, а кроме того, введение модели памяти здесь облегчает ее использование при изучении величин (в § 11).

Модель памяти ЭВМ и работа ЭВМ с аргументами алгоритмов

Я напомню, что в учебнике (п. 6.9) в качестве модели памяти ЭВМ мы используем школьную классную доску. Для алгоритма «квадрат» ЭВМ выделяет в памяти какое-то место (мы его на доске изображаем прямоугольником), а внутри него — место для запоминания аргумента (мы — на доске — внутри большого прямоугольника рисуем прямоугольник поменьше и записываем туда значение аргумента, например, 4).



Теперь, встретив при выполнении алгоритма имя аргумента «*а*», ЭВМ ***заменяет*** его на содержимое маленького прямоугольника (на 4) и дает Чертежнику конкретные числовые команды.

Хотя никакой классной доски внутри ЭВМ нет, эта модель замечательно ***точно*** отражает то, что происходит при выполнении алгоритма. Все житейские слова «стирание», «вписывание нового значения», «отведение места» и прочие ***в точности*** соответствуют всему реально происходящему. Очень важно, что с самого начала прямоугольники аргументов рисуются не сами по себе, а ***внутри*** прямоугольников алгоритмов (это будет полезно при введении величин в § 11).

Ну а в рамках темы нашего параграфа введение модели памяти ЭВМ позволяет еще раз и более глубоко объяснить, как ЭВМ выполняет алгоритмы с аргументами. Это улучшает понимание и аргументов, и модели.

> **ПРОГРАММНОЕ ОТСТУПЛЕНИЕ**. В системе «КуМир» есть поддержка описанной модели памяти ЭВМ. Все эти прямоугольники алгоритмов и аргументов (а в дальнейшем и величин) автоматически рисуются на экране. Поэтому вы можете не только рассказать про модель памяти в классе, но выделить несколько минут для работы на компьютере, чтобы школьники просто посмотрели, что происходит при выполнении алгоритма «квадрат» и других алгоритмов.

Упражнения

В упр. 1–4 никакой особой изюминки нет. В задачах 5а) и 5б), если делать их без вспомогательных алгоритмов, приходится писать слишком много команд. За счет введения вспомогательных алгоритмов объем текста можно значительно сократить. В задаче 5в) можно использовать, как вспомогательные, алгоритмы, полученные при решении задач 5а) и 5б).

Задача 6а) (закрашивание слова «СССР») устарела и уже не интересна.

В остальных вариантах задачи 6 можно пользоваться без изменений основным алгоритмом А13 и менять только вспомогательные алгоритмы. Причем необязательно для каждой новой задачи писать два новых алгоритма. Например в задаче 6б) можно алгоритм «закрашивание блока» оставить без изменений, а изменить только алгоритм «обход».

Упражнение 7 носит подготовительный характер к упражнению 8. В упражнении 8 (рис. 23, 24 на с. 48, 49 учебника) все задачи устроены одинаково. Каждая картинка получается из 5 одинаковых фрагментов, накладывающихся друг на друга. Чтобы решить задачу, нужно прежде всего увидеть на картинке такой фрагмент. Его выбор может быть не однозначен. Например, в задаче на рис. 24а) можно в качестве фрагмента выбрать как полный квадрат 5×5, в котором закрашены все 25 клеток, так и фрагмент, где закрашены только 16 клеток по краю квадрата. Второй вариант легче для программирования. В задаче на рис. 24б) имеется повторяющийся фрагмент «квадрат без одной стороны» (т. е. буква «С»).

Особо хочу отметить очень важную, на мой взгляд, задачу 9 на с. 49. Это — единственная задача, в которой нужно составить алгоритм, когда не известно, ни что будет нарисовано, ни какой вспомогательный алгоритм используется. Известно лишь, что некоторый алгоритм «фрагмент» закрашивает каким-то неизвестным нам образом клетки в квадрате 4×4. Нужно закрасить 25 одинаковых фрагментов в квадрате 20×20. Здесь не сказано, ни какие клетки надо закрашивать, ни что это за алгоритм. Да и сам алгоритм не приведен. А требуется составить основной алгоритм, который решит поставленную выше задачу.

Это чрезвычайно важная задача, потому что для ее решения необходимо использовать понятия основного и вспомогательного алгоритмов, но нельзя задействовать общее представление о задаче (которого нет). Кроме того, задача очень наглядно показывает, что мы можем заставить ЭВМ 25 раз сделать нечто, даже не зная конкретно, что именно. И ЭВМ все сумеет выполнить.

Технически написать такой алгоритм несложно. Но если ученики напишут основной алгоритм, в котором 25 раз будет вызываться «фрагмент», то это будет означать, что они недостаточно освоили вспомогательные алгоритмы. При «правильном» решении в основном алгоритме должен 5 раз вызываться какой-нибудь вспомогательный алгоритм «ряд» (или «колонка»), а уже внутри этого алгоритма — 5 раз вызываться «фрагмент». Другими словами, ученикам следует ввести ***еще один уровень*** вспомогательных алгоритмов в ходе решения задачи.

Упражнения 11 и 23 — основные в данном параграфе. Они посвящены рисованию орнаментов с помощью Робота и Чертежника. Любой орнамент — это несколько рядов,

в каждом из которых несколько фрагментов. Ученик должен суметь выделить ряды и фрагменты, а потом записать соответствующие алгоритмы. Все решения имеются в методическом пособии [Авербух]. Я лишь скажу, что в задачах 23 е),ж),з) есть хитрости. В е) и з) картинка получается наложением ***пересекающихся*** фрагментов, как в задаче 6. Кроме того, в задачах ж) и з) при задании фрагментов лучше использовать вспомогательные алгоритмы с параметрами. Например, для задачи ж) это может выглядеть так:

```
алг фрагмент
нач
| кольцо(1)
| кольцо(2)
| кольцо(3)
| кольцо(4)
кон
```

```
алг кольцо(арг вещ a) | a -- высота кольца минус 2
дано | перо в нижней точке левой вертикальной
     | стороны кольца
надо | нарисовано кольцо, перо смещено на
     | (-1,-1) от исходной точки
нач
| опустить перо
| сместиться на вектор (0,a+2)
| сместиться на вектор (1,1)
| сместиться на вектор (a+2,0)
| сместиться на вектор (1,-1)
| сместиться на вектор (0,-a-2)
| сместиться на вектор (-1,-1)
| сместиться на вектор (-a-2,0)
| сместиться на вектор (-1,1)
| поднять перо
кон
```

Из остальных отмечу упражнения 21 и 22 про рисование спиралей (рис. 29 учебника). Это отличные задачи. На рис. 29 нужно увидеть 5 полувитков. Из них одни идут вверх и вправо из исходной точки, а другие — вниз и влево. Чтобы не писать два разных вспомогательных алгоритма, можно в основном алгоритме в части вызовов написать положительный аргумент, а в части — отрицательный. Вот как может выглядеть основной алгоритм:


```
алг спираль
нач
| полувиток(1)
| полувиток(-2)
| полувиток(3)
| полувиток(-4)
| полувиток(5)
кон
```

Вспомогательный алгоритм «полувиток» вы теперь уже можете написать сами или посмотреть в методическом пособии [Авербух]. Впрочем, для слабых учеников можно пририсовать к спирали на рис. 29 еще два звена, и они смогут использовать вспомогательный алгоритм «виток», аргумент которого будет задавать размер витка. Никаких знаков, задающих направление рисования, не понадобиться.

В заключение хочу особо обратить ваше внимание, что хотя все эти задачи на рисование орнаментов и других картинок похожи друг на друга, на них надо потратить достаточное количество времени. Здесь у учеников нет никаких других инструментов решения задач, кроме вспомогательных алгоритмов (ведь это — первое понятие алгоритмического языка, после понятия алгоритма в самом начале!). Такие задачи развивают один из важнейших, базовых навыков алгоритмизации — умение разбивать задачу на подзадачи, применять вспомогательные алгоритмы. И материал должен быть усвоен, каким бы простым он вам ни казался.

§ 7. Арифметические выражения и правила их записи

Этот параграф посвящен линейной записи арифметических выражений и является абсолютно техническим.

В обоснование необходимости линейной записи можно сказать, что если мы собираемся работать с компьютером, то нам надо вводить формулы с клавиатуры компьютера, т. е. путем последовательного нажатия на кнопки. Поэтому, те формулы, которые мы привыкли записывать с корнями, дробями, степенями и индексами (как, например, формула для корней квадратного уравнения), лучше бы сначала расположить в одну строчку, без всяких дробей и прочего, чтобы потом было удобно вводить их в компьютер (нажимать на клавиши).

Впрочем, такие обоснования (хотя все, что сказано выше, — чистая правда) дети обычно пропускают мимо ушей. И в данном случае это совершенно ничему не мешает.

Материал параграфа чрезвычайно прост, и поэтому я вам про содержание этого параграфа ничего больше рассказывать не буду. Вместо этого я еще раз скажу вам о его месте.

Параграф является «остановочным». Смена рода деятельности. Поработали с Роботом, с Чертежником. Поднадоело, а тут что-то новенькое: преобразование формул из привычной школьной записи в компьютерную линейную. Материал этот обычно проходится без труда и никаких сложностей не вызывает. Да и ошибок школьники, как правило не делают. За исключением широко распространенной ошибки, когда дробь

$$\frac{a}{b \times c}$$

записывается, как $a/b \times c$ без скобок, вместо правильной записи $a/(b \times c)$.

Удивительно, но упражнения к этому параграфу пользуются популярностью, вызывают интерес. Видимо они похожи на какие-то головоломки, мальчишеские игры с шифровкой и дешифровкой, когда написано одно, а надо преобразовать его в другое. Этому параграфу, по вашему желанию, можно либо посвятить отдельный урок, либо потратить скажем, 5–10 минут на объяснение форм записи, а потом, по ходу курса, в числе прочих упражнений задавать еще и упражнения из него, пока ученики не освоят материал.

§ 8. Команды алгоритмического языка. Цикл n раз

Сам по себе этот параграф очень прост, но я на нем немного остановлюсь и расскажу про методические аспекты введения цикла n раз, а также — отчасти — управляющих конструкций вообще.

Методика введения цикла n раз

Как я уже говорил при изучении вспомогательных алгоритмов, лень — двигатель прогресса. Вместо того чтобы «честно» написать

```
вверх; вверх; вправо; вниз; вниз; вправо
вверх; вверх; вправо; вниз; вниз; вправо
вверх; вверх; вправо; вниз; вниз; вправо
вверх; вверх; вправо; вниз; вниз; вправо
вверх; вверх; вправо; вниз; вниз; вправо
```

как мы это делали при решении задачи 6а) в § 4, хочется написать что-то вроде

```
вверх; вверх; вправо; вниз; вниз; вправо } 5 раз
```

Именно так обычно и пишут школьники. Если повторить что-то надо 5 или 10 раз, то лень еще можно превозмочь и записать один и тот же кусок текста нужное число раз. Но если повторить что-то требуется тысячу или миллион раз, то записать это «честно» не удастся.

Попросите, например, школьников записать ***как-нибудь*** алгоритм смещения Робота вправо на тысячу клеток. Пусть не для ЭВМ, а для человека. Разные школьники придумают разные формы записи: кто-то напишет с многоточием, кто-то — без; кто-то нарисует фигурную скобку справа, кто-то — слева и т. п. Но как-то запишут все.

В этот момент и следует ввести цикл n раз. Сказать школьникам, что они все замечательно и совершенно правильно записали. Только мы используем школьный алгоритмический язык, а в алгоритмическом языке это записывается так:

```
нц 1000 раз
| вправо
кц
```

И такая конструкция заставляет ***ЭВМ*** выполнить команду «вправо» 1000 раз (т.е. 1000 раз скомандовать Роботу «вправо»).

Вы можете также в соответствующем гипертексте системы «КуМир» посмотреть, как работает команда «**нц** 6 **раз** елочка **кц**» в алгоритме «аллея».

Это и есть проблемный подход. Заметьте, мы не столько вводим новую конструкцию, новое понятие, сколько показываем ***форму записи***. Смысл этой формы понятен и очевиден, исходя из решаемой задачи.

Учитель, впрочем, может привести и некоторые обоснования, почему выбрана именно такая форма записи, а не иная. Например, заметив что смысл конструкции n **раз** в точности такой же, как и при рисовании фигурной скобки, но фигурную скобку на компьютере рисовать неудобно, особенно, если надо повторить несколько команд. Поэтому и введены специальные слова **нц**, **раз**, **кц**, с помощью которых указывается, сколько раз нужно повторять (записывается между **нц** и **раз**) и что именно (записывается между **раз** и **кц**): «**нц** число повторений **раз** повторяемые команды **кц**».

Можно также (и это будет совершенно точно) провести аналогию с линейной записью арифметических выражений (формул), — сказать, что это у нас своего рода «линейная запись» для повторения группы команд. А строки **нц** . . . **раз** и **кц**, выделяющие повторяемую группу команд, как скобки в формулах, выделят некоторую часть формулы. Здесь весьма глубокая параллель, и в «большой» информатике эти строки так и называют — «***структурные скобки***».

Простые и составные команды алгоритмического языка

На примере цикла n **раз** удобно ввести и объяснить еще два понятия: (1) понятие команды ***алгоритмического языка*** и (2) понятие ***составной команды*** алгоритмического языка. Начинать, впрочем, удобнее с объяснения понятия составной команды, а понятие простой команды ввести в качестве противопоставления.

Цикл n **раз** называется ***составной командой алгоритмического языка***. (В других языках программирования, в «большой» информатике и в литературе вместо «составной команды» чаще используется термин «управляющая конструкция»).

Почему выбрано слово команда, кто кем тут командует? Командует Человек. Написав такой цикл, он приказывает ЭВМ выполнить команды, записанные между **раз** и **кц**, нужное число раз. Поскольку мы говорим о команде ***для ЭВМ***, это — ***команда алгоритмического языка***. В отличие от команд Робота, команды алгоритмического языка адресованы ЭВМ.

Что значит «составная»? Это значит, что «в состав» команды могут входить другие команды, в том числе и составные.

Кроме составных команд алгоритмического языка, бывают и простые. Они других команд не содержат, но, как и составные команды, — это тоже команды ***для ЭВМ***. И здесь — одно тонкое и чрезвычайно важное место. Когда Человек ***в алгоритме*** пишет команду «вправо», то это ***НЕ*** команда Роботу. Это команда ***ЭВМ***, чтобы она скомандовала Роботу «вправо», т. е. ***простая команда алгоритмического языка***. А поскольку мы вообще всегда весь алгоритм отдаем на выполнение ЭВМ, он состоит исключительно из команд ***алгоритмического языка***.

При практической работе, при составлении алгоритмов мы, допуская некоторую вольность, для удобства говорим о командах Робота, Чертежника и пр. Но абсолютно необходимо, чтобы школьники понимали, что и как выполняется на самом деле, кому адресованы команды в алгоритме. Им должно быть ясно, что именно ЭВМ разбирается в записи алгоритма и выполняет его, при необходимости командуя исполнителями.

Итак, составная команда — это команда, внутри которой встречаются другие команды. Или, как я уже говорил, ***составная команда*** — это команда, ***в состав*** которой входят другие команды. Используемый обычно во всей остальной литературе термин «управляющая конструкция» подчеркивает, что эти команды, эти конструкции важны не сами по себе — они лишь ***управляют порядком выполнения*** того, что написано внутри их: указывают ***ЭВМ*** как, сколько раз и в каком порядке выполнять ***другие*** команды.

Составные команды и структурные скобки

В школьном алгоритмическом языке принято, за небольшими исключениями, графическое представление конструкций. Структурные скобки **нц**—**кц** пишутся, как прави-

ло, одна под другой и связываются вертикальной линией, правее которой записывается остальная часть конструкции (содержимое «скобок» — последовательность команд). Сам термин «структурные скобки» и аналогия со скобками совсем не случайны. Эту аналогию можно развивать. Подобно тому как в формулах внутри одних скобок могут быть другие скобки, в алгоритмическом языке внутри одних «структурных скобок» могут быть другие «структурные скобки», т. е. внутри одной составной команды могут быть другие составные команды. ***Скобки внутри скобок*** в выражении

$$\left(x + (x + y)^5\right)^{10}$$

аналогичны ***структурным скобкам внутри структурных скобок*** в команде

```
нц 10 раз
| вправо
| нц 5 раз
| | вправо; вверх
| кц
кц
```

Цикл n раз — основа для изучения составных команд

Цикл n **раз** — первый пример ***составной команды*** в нашем курсе. И подобная последовательность изложения не случайна. Дело в том, что цикл n **раз** — самая простая из всех составных команд, понимание которой не требует от учеников практически никаких усилий. Единственное неочевидное место — п. 8.5, где говорится, что при отрицательном числе повторений тело цикла n **раз** не выполнится ни разу, но отказа тоже не произойдет. Цикл **пока** и команда **если** более содержательны. Именно поэтому изучение управляющих конструкций (составных команд) начинается с цикла n **раз**.

Все эти обсуждения — что такое составная команда; почему она так называется; что это за **нц**—**кц** и вертикальная черта; можно ли внутри цикла написать второй цикл, а внутри третий — фактически подготовка к трудному материалу в параграфе 9. Когда мы в § 9 перейдем к циклу **пока**, на **нц**—**кц** уже никакого внимания можно будет не обращать. В цикле **пока** есть содержательный смысл, содержательная

трудность, и лучше эту трудность выделить в «чистом виде», не комбинируя ни с какими новыми обозначениями и понятиями. А в цикле n **раз** никакого содержательного смысла, с которым следовало бы разбираться, нет. Конструкция цикла n **раз** кристально ясна, и на фоне этой конструкции нетрудно ввести новые термины и обозначения, чему, собственно, и посвящен данный параграф.

Новые возможности, возникающие после введения цикла n раз

Таким образом § 8 в значительной степени работает на будущий материал, на цикл **пока**. Но параллельно он работает и на закрепление и углубление предыдущего материала. Основной пример тут — вспомогательный алгоритм с аргументами, внутри которого есть цикл n **раз**.

Посмотрите на алгоритм А28 «вверх на (n)» на с. 59 учебника, который заставляет Робота сместиться вверх на n шагов, где n неизвестно. Заготовив единожды такой алгоритм, мы сможем далее писать «вверх на (5)», «вверх на (10)» и т. д.

Обратите внимание, что от соединения двух простых вещей — возможности написать вспомогательный алгоритм с аргументами и возможности написать цикл n **раз** — возникло новое качество. Теперь мы можем записать то, что раньше записать вообще не могли. Раньше ввести команду «вверх на n шагов» у нас вообще не было возможности, не было таких конструкций. Все, что мы могли записать, — это повторение одной или нескольких команд фиксированное число раз: скажем 5, 7 или 1000. А теперь у нас появилась возможность написать «в алгебраическом виде» алгоритм, делающий что-то, зависящее от неизвестного заранее числа n.

> **МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ**. В п. 8.4 вводятся четыре алгоритма
>
> ```
> вверх на (n)
> вниз на (n)
> вправо на (n)
> влево на (n)
> ```
>
> Эти алгоритмы в дальнейшем всюду в учебнике используются как базовые, т. е. при решении любых задач считается, что эти алгоритмы уже написаны и можно использовать их наряду с обычными командами Робота.

На этом содержание параграфа (что касается изложения управляющих конструкций) заканчивается. Но с точки зрения мировоззрения остается обсудить еще один важный момент. А именно, пункт 8.7, где говорится, что новая команда — цикл n **раз** уже позволяет продемонстрировать одно из важнейших свойств информатики.

Короткие алгоритмы могут описывать длинные последовательности действий

Как всегда, все объяснения и пояснения проводятся на конкретном примере. В учебнике используется алгоритм А29 для закрашивания ряда. Обратите внимание на выписанную в п. 8.7 полную последовательность команд, которую ЭВМ выдаст Роботу при выполнении вызова «закрасить ряд (4)». Еще и еще раз я повторяю: необычайно важно, чтобы школьники понимали, что при взаимодействии ЭВМ с исполнителем от алгоритмов, циклов и прочего не остается никакого следа. ЭВМ все это преобразует в последовательность обычных команд Роботу.

После того как это понято, можно задать и следующий — ключевой для этого пункта вопрос: сколько команд Роботу даст ЭВМ при выполнении команды «закрасить ряд (1000)»? Давайте подсчитаем. При выполнении команды

```
нц 1000 раз
| закрасить; вправо
кц
```

ЭВМ выдаст Роботу 2000 команд. Далее ЭВМ выполнит команду вызова вспомогательного алгоритма «влево на (1000)», а при выполнении алгоритма будет выполнена команда

```
нц 1000 раз
| влево
кц
```

Таким образом, ЭВМ выдаст Роботу еще 1000 команд. Всего получится 3000 команд.

Я настоятельно рекомендую вам подробно разобрать все это на уроке. Здесь очень простая арифметика, но чрезвычайно важная. Алгоритмы сами коротенькие, но за счет цикла n **раз** последовательности действий, которые они описывают, могут оказаться очень длинными. С одной стороны, это хорошо: человек написал немного, а компьютер делает громадную работу. А с другой стороны, тут и скрыта сложность, возникающая при алгоритмической работе. ***Для понимания программы человеку приходится представлять себе всю ту огромную работу, которую проделывает компьютер, выполняя программу.***

Алгоритмический стиль мышления как раз и позволяет сворачивать длинные последовательности действий в короткие программы и, наоборот, разворачивать мысленно короткие программы в длинные последовательности действий. В таком «сворачивании—разворачивании» задействованы не только циклы и вспомогательные алгоритмы, но и весь остальной инструментарий программирования. Однако, уже после освоения вспомогательных алгоритмов и простейшего цикла n **раз**, можно объяснить и понять следующий важный тезис:

> если необходимо заставить компьютер выполнить огромную последовательность действий, то обычно, применяя подходящие алгоритмические конструкции, ее удается описать достаточно коротким алгоритмом.

Если бы этого не было, то алгоритмизация не имела бы никакого смысла. Весь смысл алгоритмизации и состоит в том, что мы способны с помощью алгоритмических конструкций компактно описывать и задавать как огромные последовательности действий (что мы уже чуть-чуть показали), так и огромные массивы информации (что мы покажем, когда будем проходить табличные величины).

И хотя, я повторяю, пункт 8.7, сам по себе элементарен и прост, с философской, общеметодологической точки зрения он очень важен. И я еще раз повторю, что замечательная возможность коротко описывать длинные последовательности действий скрывает в себе одну из основных трудностей в понимании даже коротких алгоритмов и программ, одну из основных сложностей алгоритмизации.

«Внешние» исполнители должны уметь выполнять только простейшие, базовые команды

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Толчком к этому отступлению послужили разговоры в кулуарах на перерыве. Мне задали такой вопрос: почему у Робота есть команда «вправо», но нет команды «вправо на (**аргцел** m)»?

Точнее, вопрос был гораздо более глубокий: ***как объяснить школьникам, что для выполнения такого, казалось бы, естественного действия нам приходится писать алгоритм, а готовой команды для него нет.***

Прежде всего, я хочу вас вернуть к пп. 3.4–3.5 про отделение материального производства от информационного. В той модели мира, которую я излагал, от станков требовалось умение выполнять только ***элементарные, базовые*** операции (штамповка, резание и пр.). Робот у нас — модель таких станков. И поэтому умеет выполнять только элементарные операции.

Вы можете продолжить эту аналогию. Ведь мы считаем Робота «внешним», существующим реально, «в металле». Любое усложнение системы команд Робота — это дополнительные затраты труда, материалов, удорожание его конструкции. В ситуации, когда Роботом все равно управляет ЭВМ, которая без труда может выполнить цикл n **раз**, такое удорожание ничем не оправдано. ***Дешевле*** иметь ЭВМ и Робота, выполняющего элементарные команды, чем иметь ту же ЭВМ и какого-то более сложного Робота.

Наконец, понятие «естественного действия» зависит от типа решаемых задач. Быть может для какой-то группы задач Робот должен все время «ходить конем» (алгоритм А1 учебника). В таком случае «естественно» было бы встраивать в Робота «ход конем», а не команды прямолинейного смещения типа «вправо на (n)».

И еще, и еще раз. Замечательность информатики, алгоритмизации, связки «ЭВМ—Робот с простейшими командами» как раз в том и состоит, что настройка на класс задач производится без изменения Робота, исключительно в информационной области, за счет написания соответствующих алгоритмов.

Забегая вперед, впрочем, замечу: это мое замечание про то, что исполнители должны уметь выполнять лишь простейшие команды, в полной мере относится лишь к «внешним», «настоящим» исполнителям. В главе 3 мы доберемся до исполнителей, реализуемых программно, записываемых на алгоритмическом языке. К этим «внутренним» исполнителям все, сказанное выше, неприменимо. Но для них между «добавлением команды» и «написанием алгоритма» не будет и абсолютно никакой разницы.

Упражнения

Из упражнений к § 8 я хочу остановиться только на одном упражнении 8. Оно не столько про цикл n **раз**, сколько про вспомогательные алгоритмы. И я могу повторить все, что я говорил про упражнение 9 в § 6. Здесь также фигурирует какой-то неизвестный нам вспомогательный алгоритм «картинка», который управляет Чертежником; рисует какую-то картинку в квадрате 1×1 и возвращает перо в начальное положение — левый нижний угол квадрата.

Требуется составить алгоритм, который изобразит эту картинку в 100 экземплярах в квадрате 10×10, т. е. нарисует 10 рядов по 10 картинок в каждом.

Смысл заключается в том, чтобы подчеркнуть метод разбиения задачи рисования 100 картинок в квадрате 10×10 на две совершенно разные части: (а) рисование одной картинки в квадрате 1×1, и (б) расположение 100 ***каких-то*** одинаковых картинок «в пространстве».

Это упражнение является, повторяю, очень важным не столько для освоения цикла n **раз**, сколько для дальнейшего освоения вспомогательных алгоритмов и идеи разделения задачи на слабо связанные подзадачи.

Лекция 5.

Ну а теперь — «поставьте спинки кресел в вертикальное положение и пристегните ремни» — мы, наконец, переходим к § 9, который является одним из самых важных и самых сложных среди первых 15 параграфов учебника. Замечу заранее, что § 16 будет гораздо сложнее, но он уже из другой области — из методов алгоритмизации. А из параграфов, посвященных алгоритмическому языку, § 9, по-моему, самый сложный.

§ 9. Алгоритмы с «обратной связью». Команда пока

Этот параграф естественно разбивается на три части:
(1) понятие команд «обратной связи» (пп. 9.1–9.2);
(2) цикл **пока** (пп. 9.3–9.10);
(3) составление алгоритмов с циклом «**пока**» (пп. 9.11–9.14).

Команды «обратной связи»

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я вам уже говорил, что если мы работаем с Черепашьей графикой в ЛОГО, то мы только командуем. То, что мы делали до сих пор, было тоже чистым «командованием». И в этом смысле практически все, что мы уже прошли, может быть пройдено на базе Черепашки в ЛОГО. То есть мы можем заменить школьный алгоритмический язык на ЛОГО, Робота на Черепашку и примерно с тем же успехом все, что мы делали, повторить. Другими словами, до § 9 наш подход методически почти не отличается от подхода Пейперта с его Черепашьей графикой. Вспомогательные алгоритмы, с аргументами и без аргументов, рисование — все это можно делать и на языке Лого с использованием Черепашки.

А вот в § 9 возникает очень важное отличие в подходе, причем не техническое, а принципиальное. Разница заключается в следующем. До сих пор мы только командовали Роботом, т. е. заставляли его выполнять предопределенную последовательность действий, следовательно, уже на этапе написания алгоритма мы знали, сколько раз и что именно он должен сделать. В § 9 впервые выясняется что Робот устроен куда сложнее Черепашки: у него есть команды обратной связи. Выполняя эти команды, Робот сообщает нам информацию об обстановке, в которой он в данный момент находится. Например, Робота можно спросить, закрашена ли клетка, где он стоит, верно ли, что правее него — стена, и пр.

Я также сразу хочу обратить ваше внимание, что у Чертежника никаких команд обратной связи нет. Поэтому он является почти полным аналогом Черепашки, только в отличие от Черепашки чуть более «прямоугольным», лучше приспособленным для рисования графиков, а не картинок. Поэтому Робот в нашем курсе играет куда более фундаментальную роль, чем Чертежник. Без Чертежника в принципе можно обойтись (или заменить его Черепашкой либо еще чем-то). Без Робота или его аналога мы получим просто другой курс.

Команды обратной связи — это команды, о которых я говорил, описывая алгоритмический стиль мышления. Мы можем обратиться к Роботу с вопросом и получить от него ответ. Можно представлять себе пульт дистанционного управления Роботом, на котором есть кнопки и лампочки. Мы нажимаем на кнопку «снизу стена», и в ответ загорается лампочка. Только в отличие от того, что я вам говорил (с. 14), лампочку здесь надо представлять себе одноцветной. Скажем, нажимаем на кнопку «снизу стена?» — если лампочка зажглась («да»), то значит стена снизу есть, а если не зажглась, то нет.

Когда Роботом управляет ЭВМ, кнопки и лампочки не нужны — ЭВМ посылает и принимает электрические сигналы. Но суть дела остается прежней: информация поступает не только от ЭВМ к Роботу, но и ***обратно*** — от Робота к ЭВМ. Отсюда и термин «обратная связь».

В рамках описанной выше модели с одноцветной лампочкой легко объяснить, и почему у Робота имеется два диаметрально противоположных набора команд-вопросов — почему есть и команда «снизу стена?» и команда «снизу свободно?» (если ответ на первый вопрос — «да», то на второй — «нет»; и наоборот, если на первый вопрос ответ — «нет», то на второй — «да»).

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Теоретически можно обойтись только одной командой-вопросом из каждой пары. Но это потребует введения отрицаний, что

приведет к появлению уже на ранних стадиях обучения записи типа «**не** клетка закрашена **и не** сверху свободно» вместо «клетка не закрашена **и** сверху стена». Мы посчитали, что с методической точки зрения жертвовать понятностью в угоду навыкам построения логических отрицаний в данном случае не следует. И именно поэтому наделили Робота диаметрально противоположными парами команд-вопросов с естественными и простыми названиями.

Три группы команд обратной связи у Робота

Команды обратной связи можно разделить на три группы. Первая группа позволяет анализировать наличие или отсутствие стен сверху, снизу, справа и слева. Таких команд 8, потому что про каждое из четырех направлений можно задать два диаметрально противоположных вопроса, например, «справа стена?» и «справа свободно?».

Вторая группа состоит также из двух диаметрально противоположных команд-вопросов «клетка закрашена?» и «клетка не закрашена?», позволяющих узнать, закрашена ли клетка, в которой стоит Робот.

А вот третья группа — особая, ее составляют две команды: «температура» и «радиация». Если команды-вопросы выдают в качестве ответа «да» или «нет», то последние две команды выдают в качестве ответа вещественное число. С помощью этих команд можно узнать, какова температура и какова радиация в клетке, в которой находится Робот.

Например, мы можем спросить: «Эй, Робот, какая температура в клетке, где ты стоишь?». И он ответит, скажем, «–7.3». Или можем спросить: «Какова радиация в клетке?» и получить в ответ, допустим, «5.5». Единицы измерения температуры, и радиации, по-моему, нигде в учебнике не фигурируют. Хотя из формулировок некоторых упражнений можно неявно сделать вывод, что температура измеряется в градусах Цельсия. Температура в клетке может быть и положительной и отрицательной. Что же касается уровня радиации, то я, честно говоря, не знаю, в каких единицах он у нас измеряется, но этот уровень либо равен нулю (т. е. радиации нет), либо положителен (больше нуля) — тогда радиация есть, хотя возможно и безвредная. Но вот отрицательным уровень радиации быть не может.

За счет команд обратной связи, и особенно таких, как «температура» и «радиация», мир Робота становится заметно богаче, и наши взаимоотношения с Роботом существенно усложняются. Благодаря этому мы можем ставить и решать больше разнообразных задач по управлению Роботом.

Вы можете запустить гипертекст «Команды обратной связи» в системе «Кумир» и быстро познакомить учеников со всеми этими командами.

Еще раз о непосредственном и программном управлении

Как вы помните, управление бывает непосредственное и программное. Если у нас в руках пульт дистанционного управления, а игрушечный Робот ездит по лабиринту, который установлен перед нами на столе, то никаких команд типа «справа свободно» нам не нужно: мы и так видим, свободно справа или нет. То же происходит, когда поле Робота изображается на экране вместе со всеми стенами и закрашенными клетками. Но если Робот находится в соседней комнате или на Марсе (т. е. мы не видим поля) или если на экране изображается только пульт управления Роботом, а поле не показывается (такой режим есть в системе «КуМир»), то команды обратной связи становятся единственным средством как-то согласовать управление Роботом с окружающей его обстановкой. Как вы убедились, даже простейшие задачи непосредственного управления становятся интересными, если поле Робота не видно и работать нужно, ориентируясь по миганиям лампочки («да» или «нет») на пульте управления. Попробуйте, например, обвести Робота вокруг прямоугольного препятствия неизвестных размеров, глядя не на поле Робота, а только на пульт.

Замечу, что даже когда поле Робота видно, мы неявно подразумеваем, что видны стены и закрашенные клетки. Что же касается температуры или радиации, то все мы привыкли к тому, что температура невооруженным глазом не видна и уж тем более не «чувствуется» радиация. Так что даже если поле Робота мы видим (на экране или непосредственно), то нахождение, скажем, самой прохладной клетки в заданном ряду клеток требует использования команд обратной связи.

Задачи о температуре и радиации на поле Робота могут заменить традиционные задачи по обработке массивов

Забегая вперед, сразу скажу, что «оснащение» Робота «термометром» и «счетчиком Гейгера» — хороший методический подход при решении большого класса стандартных задач информатики, обычно формулируемых для массивов (линейных таблиц в терминологии учебника), таких как «минимальный элемент», «индекс максимального элемента», «сумма элементов», «среднее арифметическое» и пр.

Горизонтальный ряд из n клеток с заданной в каждой клетке температурой (при изложении удобно значения температуры просто вписать внутрь клеток на классной доске) — это точный аналог ***линейной таблицы вещественных чисел***. В учебнике мы обычно такой горизонтальный ряд клеток ограничиваем (выделяем) стеной сверху и снизу и называем «коридором». Подобным же образом для работы с ***прямоугольной таблицей вещественных чисел*** можно использовать прямоугольник из клеток на поле Робота.

Тем самым почти каждую задачу по работе с таблицами можно переформулировать, как задачу про Робота. В такой формулировке будет более понятно, и почему такая задача возникла, и зачем ее нужно решать.

Возьмем, например, простейшую задачу: «найти сумму элементов линейной таблицы». Давайте переведем ее на язык Робота. Представим себе, что на поле Робота есть коридор, по которому должны будут пройти люди — ремонтники или, может быть, спасатели. До этого необходимо разведать, насколько прохождение по коридору опасно для здоровья, каков уровень радиации в коридоре. Если люди будут продвигаться по коридору с примерно постоянной скоростью, то полученная ими доза радиации окажется пропорциональной сумме уровней радиации во всех клетках коридора. Поэтому анализ опасности коридора для жизни можно переформулировать, как задачу подсчета суммарной радиации в клетках коридора. Это и есть задача нахождения суммы элементов таблицы, сформулированная в терминах управления Роботом.

Подобным же образом можно переформулировать практически любую задачу по обработке таблиц (массивов). Замечательно то, что такая переформулировка переводит указанные задачи из математической в какую-то более простую «житейскую» область. Когда для понимания формулировки задачи нужен только здравый смысл, когда математические

обозначения не используются, а условие задачи одинаково понятно всем школьникам, независимо от уровня их математической культуры. И если при решении таких задач возникают трудности, то учитель может быть уверен, что они тут именно алгоритмические, а не не в математических обозначениях и математическом подходе.

Посмотрите на следующие две формулировки одной и той же задачи:

(а) найти индексы минимального элемента прямоугольной таблицы;

(б) в прямоугольнике на поле Робота найти клетку с минимальной радиацией и переместить Робота в эту клетку (например, «на ночь»).

Первая формулировка — чисто математическая. Кому и зачем нужны эти индексы остается далеко за рамками задачи. В отличие от второй — более понятной и осмысленной формулировки. Хотя Робот — «придуманный» (и это понятно), хотя, скорее всего, школьники раньше дела с радиацией не имели, все-таки им более или менее ясно, что высокий уровень радиации — это «плохо», что требуемая в задаче клетка — самая безопасная, и если уж оставлять Робота на долгое время («на ночь») в «поле», то лучше выбрать самую спокойную и безопасную клетку.

Понятие «обратной связи» — важное универсальное понятие

Понятие «обратной связи» не есть что-то, связанное специфически с Роботом и основанной на нем методике обучения информатике. Понятие «обратной связи» — это фундаментальное понятие, активно используемое в разных областях, а особенно — в кибернетике.

Про команды обратной связи можно, если вы захотите, поговорить и на философско-художественном уровне, привести примеры из экономической жизни, сравнить с командно-административным управлением экономикой, которое, как принято считать, преобладало в СССР до «перестройки». Можно даже затеять «перестройку» управления Роботом: если раньше мы просто командовали Роботом, ничего у него не спрашивая и ничем не интересуясь, то теперь мы иногда будем у Робота что-то спрашивать (как там у него обстановка), а уж потом — как всегда — командовать.

Если же говорить серьезно, то использование команд обратной связи позволяет сделать управление более гибким, более эффективным. Будь то управление экономикой или управление Роботом. В учебнике упоминаются начальник, повар, шофер — и вы сами без труда придумаете десятки ситуаций, в которых без обратной связи не обойтись.

Управление Роботом вручную и необходимость введения цикла пока для программного управления с использованием обратной связи

После короткого описания команд обратной связи (п. 9.1) применяется основной методический прием учебника — решение задачи управления «вручную» и последующее противопоставление схемы непосредственного управления схеме программного управления. Здесь, наверное, самое яркое место для демонстрации сути такого методического приема, и именно потому пример из этого параграфа был использован мною при изложении методических основ курса (с. 39).

Теперь, когда мы добрались до самого параграфа, я кратко повторю в чем суть приема.

Сначала ставится задача (проблемный подход). В учебнике (п. 9.2) выбрана следующая задача: «Где-то ниже Робота на поле есть стена. Надо переместить Робота вертикально вниз до стены, т. е. установить Робота в клетку непосредственно над стеной на одной вертикали с исходным положением Робота.»

Далее ученикам говорится: «Как бы вы решали такую задачу, если бы управляли Роботом вручную?». Напомню, что в этот момент учитель может нарисовать у себя на бумаге поле Робота, никому его не показывать и проверять решения учеников, играя роль Робота, — т. е. «выполняя» команды, которые будут произносить ученики.

После того как школьники поймут, что надо все время спрашивать «снизу свободно?» (или «снизу стена?») и командовать «вниз» до тех пор, пока ниже Робота будет свободно, рекомендуется еще раз проверить их в ситуации, когда Робот с самого начала уже стоит вплотную над стеной. Соответственно, у тех учеников, которые начинают с команды «вниз» (а не с вопроса), Робот разобьется.

ИСТОРИЧЕСКИЕ ЗАМЕЧАНИЕ. В некоторых языках программирования, в частности в Паскале, кроме цикла **пока** (специалисты называют его «while-do») есть и другой цикл — «repeat-until». Разница между ними состоит в том, что в первом цикле проверка условия производится ***до*** первого выполнения тела цикла, а во втором — ***после*** первого выполнения тела цикла. Таким образом, во втором случае тело цикла выполняется как минимум один раз, а уж потом производится проверка.

При управлении Роботом какая-то часть школьников может пойти по второму пути: сначала шаг, потом проверка. Это путь соответствует циклу «repeat-until». Наш пример, когда Робот с самого начала уже стоит у стены в нужной клетке, показывает, что цикл «repeat-until» здесь не годится.

Цикл **пока** работает правильно в любых ситуациях. Да и в целом он гораздо более общий и «правильный», чем «repeat-until». Поэтому в учебнике рассматривается и излагается только цикл **пока**.

Но в системе «КуМир» есть обе конструкции, и цикл с проверкой условия в конце записывается так:

```
нц
| тело цикла (серия)
кц при условие
```

После решения задачи «вручную» обычно каждый ученик в классе в деталях представляет себе как именно надо командовать Роботом, чтобы решить поставленную задачу. Именно это понимание и является целью всей «игры».

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. В идеале, конечно, хорошо было бы иметь недорогого Робота-игрушку, поле, на котором можно устанавливать стены, и пульт дистанционного управления (чтобы повозиться с «живым» Роботом), а также возможность подключить этот пульт к компьютеру (чтобы компьютер мог Роботом управлять). Вообще в мире подобные игрушки есть, скажем, варианты игры Лего, где такого Робота с датчиками, моторами и пр. можно собрать и даже подключить к компьютеру, чтобы дальше составлять алгоритмы (в программной системе Лого).

Но для целей нашего курса вполне достаточно иметь имитацию всего этого на ЭВМ и разыграть пару спектаклей описанного выше типа в классе.

Переход к схеме программного управления и проблемный подход

Когда ученики в деталях разберутся в том, как решать поставленную выше задачу при управлении Роботом «вручную», происходит ключевое для нашего методического приема противопоставление этой работы «вручную» и схемы программного управления.

«Замечательно!» — говорит учитель, — «Все разобрались, как надо управлять Роботом. Но ведь мы занимаемся не ручным управлением Роботом, а информатикой. Поэтому наша задача — ***написать алгоритм*** для ЭВМ, при выполнении которого ЭВМ прокомандует Роботом так, чтобы задача была решена. Заметьте, что расстояние от Робота до стены неизвестно, но ЭВМ, выполняя алгоритм (и не зная расстояния), должна сместить Робота к стене. Попробуйте записать такой алгоритм. Вы ведь понимаете, ***как*** нужно управлять Роботом, ***какие*** команды ЭВМ должна выдать Роботу. Так ***запишите*** это ***в виде алгоритма*** для ЭВМ».

Я еще раз обращаю ваше внимание на этот проблемный подход. Ведь практически в этом месте мы просим школьников ***придумать*** конструкцию цикла **пока**. Т. е. в отличие от стандартной последовательности (сначала форма записи, потом ее семантика (смысл), потом решение задач), мы сначала ставим задачу, потом разбираем как ее решать, т. е. разбираемся со смыслом (семантикой) будущей конструкции, а потом просим учеников придумать форму записи.

Если среди решений встретится запись нескольких первых вопросов и команд (возможно с использованием **если**), а потом многоточие и фраза типа «и так далее пока снизу свободно», то либо объясните, чем такое решение неудовлетворительно, либо просто явно запретите использовать многоточия и обороты «и так далее». Напомните про цикл n **раз**, в котором никаких многоточий нет. И через некоторое время какие-то варианты цикла **пока** появятся у всех.

С большой вероятностью в решениях школьников вы увидите скобки **нц**—**кц**, поскольку они уже встречались в цикле n **раз**. Вы, конечно, встретите и самый обычный цикл **пока**, иногда в англоязычной версии «while-do»: кто-то, возможно, знает другие языки, или уже прочитал учебник, а кто-то может и придумать прямо форму записи этого цикла в алгоритмическом языке. Можно ожидать, что в классе встретится три—пять вариантов записи цикла **пока**.

И лишь после этого и после того, как вы их похвалите, следует перейти к формальному изложению цикла **пока** — «в алгоритмическом языке эта конструкция ***записывается*** так»:

```
нц пока снизу свободно
| вниз
кц
```

Другими словами, в этот момент учитель говорит лишь, как ***записывается*** конструкция, смысл которой понятен каждому ученику. Благодаря этому изучение конструкций алгоритмического языка ***заменяется*** изложением форм записи. Все время тратится на решение задач, а формы записи вводятся по мере необходимости. Это очень важное смещение акцентов.

Необходимость детального обсуждения цикла **пока**

Тут я должен сказать, что цикл **пока** содержательно является очень сложной конструкцией. У школьников возникает масса гипотез про эту конструкцию, которые необходимо частью развеять, частью подкрепить, чтобы они ясно себе представляли, как цикл **пока** работает.

И мы использовали здесь методический прием, который состоит в том, что при изложении сложного материала нужно попытаться дать все возможные взгляды на этот материал с разных точек зрения. Поэтому после введения цикла **пока** в учебнике он некоторое время обсуждается с самых разных сторон. Это и работа ЭВМ в ходе выполнения цикла **пока** (так называемые ***диалоги*** «ЭВМ—Робот»). И графическое представление (стандартная «блок-схема») этого цикла. И разные конкретные примеры и особые случаи при выполнении цикла.

Как нас учат психологи, мы постарались воздействовать на разные полушария мозга ученика, представить материал и в абстрактной символьной форме (левое полушарие), и в наглядной графической (правое полушарие). Цикл **пока** достаточно труден. И все, что можно про него сказать с разных точек зрения, должно быть сказано в надежде, что ученик, на которого не подействуют диалоги «ЭВМ—Робот», быть может воспримет графическую схему. А другой ученик, наоборот, быть может не воспримет графическую схему, но поймет все через диалоги «ЭВМ—Робот». Третий, возможно, поймет что-то из диалогов, что-то из графической схемы,

что-то из обсуждения конкретного примера, как-то это все в голове у него свяжется, уляжется и сформирует цельную картину работы цикла **пока**. Чем больше разных представлений об этом цикле мы дадим, тем больше вероятность, что все ученики его освоят.

Объяснение цикла **пока** *через диалоги «ЭВМ—Робот»*

Первое и самое важное объяснение цикла **пока** — это ***диалог*** «ЭВМ—Робот» в ходе выполнения цикла. Здесь на нескольких примерах разбирается, что делает ЭВМ, какие команды ЭВМ отдает Роботу, что делает Робот и какую информацию он передает «обратно» ЭВМ. Фактически, это продолжается линия про разделение труда между ЭВМ и исполнителями, которая пронизывает несколько параграфов.

Материал также можно представить в виде небольшого спектакля (спектаклей) на уроке. Действующие лица и исполнители (не обращайте внимания — это просто игра слов) берутся из схемы программного управления: Человек, ЭВМ, Робот. Человек составляет алгоритм и отдает его ЭВМ. Обычно роль Человека играет учитель — ведь примеры должны быть тщательно подобраны, чтобы продемонстрировать особенности цикла **пока**. Впрочем, можно сначала задать задачу, а потом ее решение (алгоритм) «разыграть» и продемонстрировать в спектакле только двух актеров (роли обычно играют два ученика): ЭВМ и Робот.

При этом важно следить, чтобы ученик, играющий роль ЭВМ, был достаточно формален и, если можно так выразиться, «туп». Ведь он имитирует рутинную работу по формальному выполнению алгоритма. Здесь нет и не может быть никаких «догадок», никакого творчества — вот главное, за чем должен следить учитель. Можно (и нужно) дать и алгоритм, который разобьет Робота, и проследить за тем, чтобы ученики сыграли свои роли правильно.

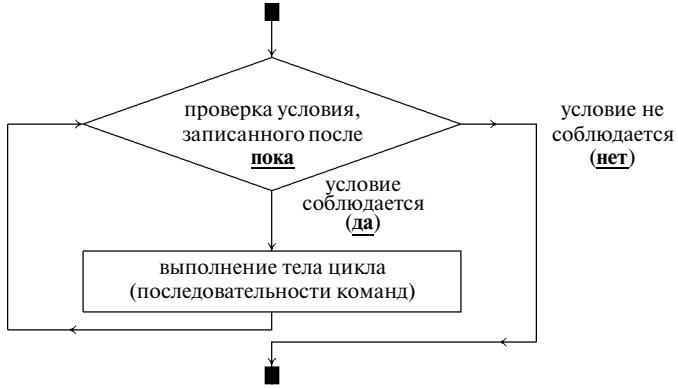
Заметьте, что раньше ученик был автором (впрочем, нигде не записанного) алгоритма, мог гордиться, если алгоритм отработал правильно, и нес ответственность, если по его вине Робот разбивался о стену. Теперь же — в роли ЭВМ — ученик выполняет механическую работу, гордиться ему нечем и отвечать он ни за что не должен. «Мое дело — компьютерное, я тут алгоритмы исполняю, а что Робот разбился — так это вы к ***автору алгоритма*** обращайтесь», может сказать «ЭВМ», если вы затеете обсуждение того, кто виноват в поломке техники.

Повторю, что подобный спектакль еще раз подчеркивает разделение труда между ЭВМ и Роботом, что очень важно: цикл **пока** понимает и выполняет ЭВМ, Робот как был Роботом, так и остался. Никаких циклов, никаких вспомогательных алгоритмов, никакого алгоритмического языка Робот не знает. Алгоритм (программу) на школьном алгоритмическом языке от Человека получает ученик, играющий роль ЭВМ, а вовсе не Робот.

> **ПРОГРАММНОЕ ОТСТУПЛЕНИЕ**. В системе «КуМир» на IBM PC есть «живые» демонстрации всех трех приведенных в учебнике диалогов «ЭВМ—Робот», а также демонстрации подобных диалогов для некоторых других ситуаций и фрагментов алгоритмов. При запуске такой демонстрации на экране с интервалом в полторы—две секунды появляются команды ЭВМ, ответы Робота и показывается изменение положения Робота на поле. Просмотр этой демонстрации занимает намного меньше времени, чем розыгрыш спектакля в классе. Но даже при использовании компьютерных демонстраций я вам рекомендую хотя бы один раз «спектакль» в классе поставить.

Объяснение цикла **пока** *с помощью описания и блок-схемы*

Следующее объяснение цикла **пока** в учебнике — это изложение его общего вида и описания работы ЭВМ при выполнении цикла (п. 9.5), а также иллюстрация этого описания в виде стандартной блок-схемы (п. 9.6).



После слова **пока** (внутри ромба на блок-схеме) пишется условие ***продолжения цикла***, внутри прямоугольника (между строками **нц** и **кц**) — так называемое ***тело цикла***. Блок-схема показывает наглядно, как работает ЭВМ при выполнении цикла.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Сейчас, по-моему, это уже несколько ушло в прошлое, но еще недавно было целое «методическое течение» в преподавании информатики, которое предлагало записывать алгоритмы (программы) на «языке блок-схем»: никакого алгоритмического языка не вводить, алгоритмы рисовать в виде блок-схем, а при «выходе на ЭВМ» перекодировать соответствующую блок-схему на используемый язык программирования (скажем, БЕЙСИК).

Поскольку мы первый раз встретились с блок-схемой в нашем учебнике, я хочу изложить нашу точку зрения на роль и место блок-схем.

Во-первых, я хочу напомнить цели нашего курса, четыре фундаментальных понятия информатики и сказать, что «язык блок-схем» еще менее подходит для достижения этих целей, чем Бейсик.

Во-вторых, блок-схемы хороши для ***иллюстрации отдельных*** очень простых конструкций, вроде цикла **пока**, у которого внутри других составных команд нет. И даже некоторые простые конструкции (например, цикл n **раз**) с помощью ромбиков и стрелок выразить нельзя.

В-третьих, блок-схемы провоцируют «беспорядочное» программирование, когда стрелку из любого места можно провести в любое другое, когда не применяются вспомогательные алгоритмы и все пишется одним куском, и пр. Уровень алгоритмической культуры, которого можно достичь, используя такой инструмент практически не превышает уровня, достигаемого при решении задачи про «Волка, козу и капусту».

Поэтому отношение к блок-схемам у нас совершенно стандартное для нашего курса: если их можно использовать для ***иллюстрации*** каких-то понятий, для достижения целей курса (как в случае с циклом **пока**) — это надо делать, т. е. задействовать их как одно из средств. Но «сами по себе» они ничего не представляют, а записывать на так называемом «языке блок-схем» алгоритмы просто вредно.

Особые ситуации и свойства цикла пока

Дальше в учебнике идут три важных пояснения, цель которых — завершить формирование полного и исчерпывающего представления о цикле **пока**:

(1) тело цикла может не выполниться ни разу;

(2) тело цикла может выполняться бесконечно («зацикливание»);

(3) условие окончания цикла в теле цикла ***не*** проверяется.

Первые два пояснения очень просты, а третье — чуть посложнее, поскольку должно развеять некоторые заблуждения, которые, увы, могут возникнуть у школьников.

Однако прежде чем комментировать эти три пояснения учебника, я замечу, что есть еще одно важное и очень простое замечание, которое мы в школьный курс не включили, поскольку оно нам не понадобилось. Это замечание о том, что по завершении цикла **пока**, — ***что бы и как бы ни происходило внутри цикла, какие бы команды мы ни написали*** будет выполнено ***отрицание*** условия, записанного в **пока**. Даже на блок-схеме вы можете увидеть, что это совершенно тривиальное и очевидное утверждение. Но оно очень важно при анализе алгоритмов и при доказательстве их правильности. В нашем курсе эти вопросы почти не затрагиваются и поэтому слишком акцентировать внимание на этом простом факте мы не стали.

Ну а теперь — разъяснения из учебника.

Тело цикла может не выполниться ни разу

Первое простое замечание состоит в том, что тело цикла может вообще не выполняться ни разу, и это совершенно нормальная ситуация: ЭВМ начинает выполнять цикл, проверяет условие, обнаруживает, что оно не соблюдается, и выполнение цикла заканчивается. Такая ситуация аналогична выполнению цикла 0 **раз**. Важно подчеркнуть, что никаких неприятностей в подобной ситуации не возникает, цикл нормально завершается.

Опыт показывает, что некоторые ученики почему-то считают эту ситуацию ненормальной и пытаются ее избежать. Например, после введения команды **если** вы можете столкнуться с записью типа:

```
если снизу свободно
| то
|  нц пока снизу свободно
|  | вниз
|  кц
все
```

вместо имеющей в точности тот же смысл более простой записи

```
нц пока снизу свободно
| вниз
кц
```

Оба приведенных выше фрагмента будут выполняться совершенно одинаково (точнее, в первом случае ЭВМ в начале может дважды спросить Робота «снизу стена?», прежде чем скомандует «вниз»). Важно показать, что если Робот уже стоит у стены, то во втором фрагменте при проверке условия выяснится, что оно не выполнено с самого начала, и тело цикла выполняться не будет. Не выполнено условие — и не выполнено, нормальная ситуация, ничего страшного.

Тело цикла может выполняться бесконечно («зацикливание»)

Второе пояснение тоже простое: возможны ситуации, в которых условие в цикле **пока** всегда выполнено и выполнение цикла будет продолжаться бесконечно. Эта ситуация называется «зацикливанием» и, как правило, связана с ошибками при составлении алгоритма.

Вот один пример. Пусть известно, что на поле, где-то ниже Робота есть стена, а мы хотим переместить Робота вниз до этой стены так, чтобы по дороге Робот (как Мальчик-с-Пальчик из известной сказки) закрасил все клетки, по которым он пройдет. Если при написании алгоритма мы «зазеваемся» и случайно забудем в теле цикла написать команду «вниз», то вместо правильного цикла


```
нц пока снизу свободно
| вниз
| закрасить
кц
```

может получиться (ошибочно):

```
нц пока снизу свободно
| закрасить
кц
```

Что произойдет при его выполнении? Если Робот стоит вплотную к стене, то цикл сразу закончится. Если же ниже Робота стены нет, то ЭВМ скомандует Роботу «закрасить» и снова спросит, свободно ли снизу. А поскольку Робот остался на том же месте, он снова ответит, что снизу свободно. ЭВМ снова скомандует «закрасить» и снова спросит, свободно ли снизу. И так все будет повторяться до бесконечности. А точнее, до тех пор пока не прекратится по каким-то внешним причинам (например, выполнение алгоритма будет прервано или компьютер вообще выключат из сети).

Условие цикла в теле цикла не проверяется

Условие продолжения цикла (условие после **пока**) ***перед*** каждым выполнением тела цикла проверяется, а ***в процессе*** выполнения тела цикла — нет.

Как это ни удивительно, но, несмотря на все предыдущие разъяснения, диалоги, блок-схему и пр., некоторые школьники (возможно, даже не отдавая себе в этом отчета) считают, что как только условие после **пока** перестанет выполняться (даже если в этот момент ЭВМ находится «в середине» выполнения тела цикла), цикл немедленно прекратится. Например, если в цикле написано «**пока** снизу свободно», то, как только под Роботом окажется стена, ЭВМ каким-то чудесным образом про это узнает и немедленно прекратит выполнение цикла.

Быть может, это издержки подхода, и школьники, представляя себя вместо и Робота и ЭВМ одновременно, считают, что ЭВМ постоянно следит, не пора ли остановиться.

Конечно, это не так. Условие проверяется ***перед*** выполнением тела цикла. В ходе выполнения тела цикла ЭВМ про

условие «забывает». Ведь чтобы узнать, что под Роботом стена, ЭВМ должна потратить «время и силы» — задать Роботу соответствующий вопрос, получить ответ и проанализировать его. И это одна из причин, по которой я столько раз говорил вам о важности понимания «разделения труда» между ЭВМ и исполнителями.

С этим заблуждением школьников, на наш взгляд, надо целенаправленно и специально побороться, чтобы эту ложную гипотезу с самого начала искоренить. Бороться надо а) диалогами, б) графическими схемами, в) бытовыми примерами, которых у нас в учебнике нет, но которые можно взять из первых учебников Ершова. Именно искоренению этого заблуждения посвящены два примера в п. 9.9. В первом из них приведена ситуация, когда при выполнении простого цикла

```
нц пока снизу свободно
| вниз
| вниз
кц
```

Робот врезается в стену (поскольку в теле цикла — между командами «вниз» — условие «снизу свободно» не проверяется).

Второй пример (выход из закрашенной полосы с тремя командами «вниз» в теле цикла) показывает, что Робот может и не разбиться, а цикл — нормально завершиться. Но если у школьников будет ошибочное представление о порядке выполнения цикла **пока**, то они будут неправильно представлять себе конечное (после выполнения цикла) положение Робота. Обсуждаемая нами распространенная ошибка в данном случае состоит в том, что ученики считают: раз написано «**пока** клетка закрашена», значит, как только Робот выйдет на незакрашенную клетку, цикл завершится. И опять — с помощью диалога «ЭВМ—Робот», с помощью блок-схемы или любым иным образом нужно показать, что ЭВМ после вопроса «клетка закрашена?» будет выдавать Роботу три команды «вниз» подряд — без всяких дополнительных вопросов. Поэтому при последнем выполнении тела цикла Робот проскочит две незакрашенные клетки и только после этого остановится.

Помимо приведенных в учебнике, вы можете использовать и другие примеры для прояснения и «впечатывания» в учеников цикла **пока**. Например, замечательные «бытовые»

примеры, иллюстрирующие цикл **пока**, есть в первых учебниках А.П. Ершова [Ершов], скажем, «**пока** в тарелке есть хоть ложка супа, съесть еще ложечку».

Возвращаясь к вопросу о проверке условия цикла только ***перед*** выполнением тела цикла, я хочу напомнить вам «бытовой» пример из этих учебников с 7-литровым ведром, в которое наливают поочередно литр холодной и литр горячей воды, чтобы сделать теплую воду:

```
нц пока ведро не полно
| долить литр холодной воды
| долить литр горячей воды
кц
```

Поскольку проверка условия производится только ***перед*** выполнением тела цикла, то выполнение такого цикла приведет к тому, что последний литр горячей воды прольется на пол.

> **МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Хотя мы такого рода бытовые примеры в учебник не включали, их использование может облегчить изложение и усвоение материала, а также разрядить обстановку в классе. Первоначально у нас был план нижнюю треть каждой страницы учебника отвести под неформальную развлекательную полосу с «ложечками супа», с веселыми историями и прочим, — но по ряду причин это не получилось. Поэтому все такие мотивационные, «бытовые», занимательные добавления, примеры и истории, без которых жизнь скучнеет и сереет, учителю надо добывать из других учебников, из книжек типа [ЗК] и иных источников.

Общие правила составления цикла **пока**

На этом изложение цикла **пока**, как конструкции алгоритмического языка, заканчивается. Но, в отличие от других параграфов, ввиду сложности цикла **пока**, остаток параграфа (пп. 9.10–9.14) содержат не только примеры алгоритмов, но и

(а) маленький, но важный п. 9.11, где изложена методика составления алгоритмов с циклом **пока**;
(б) содержательный п. 9.14, в котором на конкретном примере демонстрируется применение и метода последо-

вательного уточнения (п. 6.5), и методики составления цикла **пока**.

Это второе место в учебнике (после п. 6.5), где говорится не только о форме записи и смысле отдельной конструкции алгоритмического языка, но и о некотором ***методе*** составления алгоритмов, ***методе алгоритмизации***. В п. 6.5 был кратко изложен метод последовательного уточнения. Здесь (в п. 9.11) также в очень упрощенном и сжатом виде, без особых пояснений, в директивной форме дается схема ***составления, придумывания*** цикла **пока**: расписано, ***как надо думать*** при составлении цикла, изложены все 4 ключевых этапа этого процесса. Кроме того, в первом предложении п. 9.11 («Всякий раз, когда число повторений каких-то действий заранее неизвестно, используется цикл **пока**») фактически содержится еще один — нулевой — этап этой методики, определяющий класс задач, к которому она применима.

До сих пор мы обсуждали, как выполняются алгоритмы, но ***методов*** решения алгоритмических задач (за исключением п. 6.5) мы не рассматривали. Как должен действовать ученик, встретив новую задачу? В п. 9.11 впервые приводится достаточно последовательное изложение того, над чем надо думать, чтобы составить алгоритм с циклом **пока**.

Прежде всего нужно решить, понадобится ли нам цикл **пока**. Рецепт тут очень простой: если число повторений некоторых действий заранее не известно, то придется использовать цикл **пока**. (Если количество повторений известно, то мы будем пользоваться циклом n **раз** или, забегая вперед, циклом **для**.)

Если число повторений неизвестно, т. е. будет использоваться цикл **пока**, то для его составления можно воспользоваться следующей методикой.

1) Понять, когда цикл должен закончится. Обращаю ваше внимание на то, что каково бы ни было тело цикла, что бы мы ни написали внутри, когда цикл закончится будет нарушено условие, которое написано после слова **пока**. Так что первая рекомендация — начать с придумывания ***условия окончания цикла*** — цели нашего цикла. Если наша цель «спустить» Робота к ближайшей горизонтальной стене, то условие окончания будет «снизу стена». Дальше это условие окончания можно «вывернуть наизнанку», записать противоположное условие (как говорят в информатике и логике, записать ***отрицание*** этого условия), и получится условие ***продолжения цикла, которое следует записать после слова*** **пока**. В самом цикле пишется условие продолжения цикла: при

каком условии тело цикла нужно выполнить хотя бы еще один раз. Но начинать думать над будущим циклом лучше с условия окончания. Другими словами, вначале нужно придумать, когда же цикл все-таки должен закончиться, а уже потом после **пока** написать его отрицание.

Обращаю также ваше внимание, что поскольку мы составляем цикл **пока** для ЭВМ, составляем алгоритм, условие после **пока** должно быть выражено формально. Мы не можем писать «**пока** Робот еще не дошел до стены», мы должны писать «**пока** снизу свободно», поскольку «снизу свободно» — это команда Робота, формальное действие, которое может выполнить ЭВМ.

2) После того, как мы выяснили и записали условие после **пока**, нужно решить, что мы все-таки собираемся делать внутри цикла, точнее, что будет меняться в теле цикла (положение Робота, количество закрашенных клеток, значения величин алгоритмического языка и т. п.). Следует отдавать себе отчет в том, что цикл нам нужен для совершения какой-то работы. Эта работа будет делаться не сразу, а поэтапно. После некоторого числа повторений что-то уже будет сделано, какие-то промежуточные результаты будут достигнуты, а что-то еще останется недоделанным. И когда мы думаем над составлением цикла, надо как-то представить себе, описать, какие именно промежуточные результаты мы собираемся получить после нескольких выполнений тела цикла. Скажем, мы должны Робота провести по какому-то пути и закрасить клетки на этом пути. Тогда, после некоторого числа повторений цикла одна часть пути уже будет пройдена и закрашена, а другая — еще не пройдена и не закрашена.

3) Далее нужно понять, какие команды мы собираемся записать в теле цикла, что мы собираемся сделать за один раз, за одно повторение. То есть если в пункте 2 мы должны представить себе процесс повторений «в общем», какие будут промежуточные состояния, то здесь мы должны уже строго формально записать переход из одного промежуточного состояния в соседнее — записать тело цикла на алгоритмическом языке.

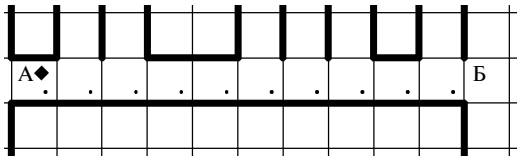
4) Ну и наконец нужно убедиться, что не произойдет «зацикливания», что после некоторого числа повторений тела цикла условие, записанное после **пока**, перестанет выполняться, т. е. цикл рано или поздно закончится.

Эта методика составления алгоритмов с циклом **пока** сначала ***иллюстрируется*** на очень простом примере с закрашиванием коридора (п. 9.12), а потом ***применяется*** для со-

ставления уже вполне содержательного алгоритма «в левый верхний угол лабиринта» (п. 9.14).

Иллюстрация методики составления алгоритмов с циклом **пока**

Иллюстрация методики проводится на примере следующей задачи. Робот стоит в левой клетке горизонтального коридора. Нижняя стена коридора сплошная, а в верхней могут быть какие-то выходы вверх. ***Длина коридора неизвестна***. Надо «провести» Робота по коридору и закрасить все клетки этого коридора (рис. 41 учебника).



Важно, что алгоритм должен быть универсальным, и решать задачу для любого коридора, подходящего под заданное описание. Если бы коридор был один — нарисованный в учебнике, то мы могли бы либо написать алгоритм вообще без циклов, (подобно тому, как делали это в самом начале в § 4), либо подсчитать длину коридора по рисунку и использовать цикл 10 **раз**.

Но наша задача — составить ***один*** универсальный алгоритм, который будет правильно работать для коридора ***любой длины***, лишь бы нижняя стена коридора была сплошная.

Я сейчас более подробно, чем в учебнике, повторю иллюстрацию применения методики составления алгоритмов с циклом **пока** для решения данной задачи, поскольку я считаю ***методы алгоритмизации*** чрезвычайно важным материалом.

Итак, давайте ***думать*** «по правилам». Прежде всего (этап «0») надо разобраться, будем ли мы вообще использовать цикл **пока**. Поскольку длина коридора неизвестна и может быть любой, мы вынуждены будем использовать цикл **пока** (правило «0»).

Раз цикл **пока** будет использоваться, то первое, что мы должны ***понять*** (этап «1»), — каково условие окончания цикла. Давайте посмотрим на рисунок, — цикл должен закончиться, когда Робот выйдет из коридора и окажется в клетке Б. Как эту клетку Б отличить от клеток внутри коридора? Что значит, что Робот в клетке Б? Из картинки видно, что в любой клетке внутри коридора снизу Робота есть стена, а в клетке Б снизу свободно. Значит мы можем использовать условие «снизу свободно» как условие окончания цикла, а значит условием продолжения цикла будет «снизу стена». В этот момент, мы не знаем еще, что будет внутри цикла, но уже можем написать:

```
нц пока снизу стена
| какие-то действия внутри цикла
кц
```

ЗАМЕЧАНИЕ. Заметьте, что мы не только придумали, чем клетка Б отличается от остальных клеток коридора, но и записали это условие (точнее его отрицание) на алгоритмическом языке. Если бы это не удалось, то нам пришлось бы придумывать другое условие окончания — такое, чтобы условие продолжения можно было записать после **пока**.

ЕЩЕ ОДНО ЗАМЕЧАНИЕ. Придумывать сначала условие ***окончания*** цикла обычно проще, чем условие ***продолжения***, по той причине, что условие ***окончания*** описывает ***одну*** — финальную — позицию Робота (***одно*** финальное состояние изменяющихся в цикле объектов), в то время как условие ***продолжения*** обычно описывает целое ***множество*** разных промежуточных состояний, которые достигаются в ходе исполнения цикла.

И ЕЩЕ ОДНО ЗАМЕЧАНИЕ. Вы могли заметить, что разделение на этапы слегка условно. Ведь при придумывании условия окончания мы уже считали, что Робот будет двигаться по коридору слева направо, а «принятие» такого решения по нашей методике относится к этапу «2».

Конечно, некоторое общее интуитивное представление о том, как вообще в целом задача будет решаться должно возникнуть в голове ученика с самого начала. Методика лишь помогает от общей идеи типа «идти направо и по дороге красить» перейти к формально записанному алгоритму. Но никакая методика не поможет придумать такую общую идею.

Теперь (этап «2») надо попытаться описать промежуточное состояние после некоторого числа повторений неизвестного пока тела цикла. А также (этап «3») записать формально, что происходит за одно повторение тела цикла (в программировании говорят «за один ***шаг*** цикла»). Эти два этапа довольно сильно пересекаются, поскольку при продумывании этапа «2» обычно — правда без формальностей — решается и что будет происходить за один шаг цикла. Поэтому в этапе «3» остается лишь ***формально*** выразить то, что было придумано на этапе «2».

Итак, этап «2». Что мы вообще собираемся делать в цикле? Нам надо двигать Робота вправо и красить клетки. Пусть мы уже несколько раз выполнили тело цикла. Значит Робот будет стоять в какой-то клетке внутри коридора, часть коридора левее Робота будет закрашена, а правее — еще нет. Возникает, однако, вопрос: должна ли быть закрашена клетка, в которой в этот момент стоит Робот?

Обратите внимание, что в соответствии с этапом «2» я пытаюсь описать промежуточное ***состояние*** коридора и Робота. Это очень важный момент. Я не описываю здесь, что Робот должен ***делать***, я описываю в каком ***состоянии*** будут находиться Робот и коридор, какие клетки закрашены, а какие — нет. Я описываю ***состояние***, а не ***действия*** Робота.

И при описании промежуточного ***состояния*** у меня возникли две возможности: считать закрашенной или незакрашенной клетку, в которой стоит Робот. Чтобы разобраться с этим вопросом, рекомендуется проанализировать конечное состояние Робота: ведь после выполнения тела цикла в последний раз наше «промежуточное» состояние становится «конечным». По условиям задачи конечную клетку — клетку Б вне коридора — красить не надо. Поэтому ***и в промежуточных состояниях*** клетка, в которой стоит Робот, должна быть ***не закрашена***.

В этот момент этап «2» завершен — мы выяснили, что и как будет меняться в теле цикла (положение Робота, «закрашенность» коридора), и описали промежуточные состояния.

> **ЗАМЕЧАНИЕ**. И опять, конечно, все вышесказанное мы делали в рамках некоторой «общей идеи», в которую, я замечу, неявно входило и то, что Робот за шаг цикла будет смещаться вправо только на одну клетку. Существуют задачи, когда количество клеток, на которые должен сместиться Робот за один шаг цикла, неочевидно.

> Желающих более глубоко разобраться в этом вопросе я отсылаю к нашему вузовскому учебнику [ПдМ], с. 60.

Теперь (этап «3») надо формально записать тело цикла, т. е. ту последовательность команд, которую следует выполнить за один шаг цикла, за одно повторение тела цикла. Для этого мы должны посмотреть на два ***соседних состояния*** Робота и коридора, сформулированных нами на этапе «2». Как перейти из одного состояния в следующее? Для этого надо закрасить клетку, где стоит Робот (напомню, она еще не закрашена), и сместить Робота на одну клетку вправо:

```
нц пока снизу стена
| закрасить
| вправо
кц
```

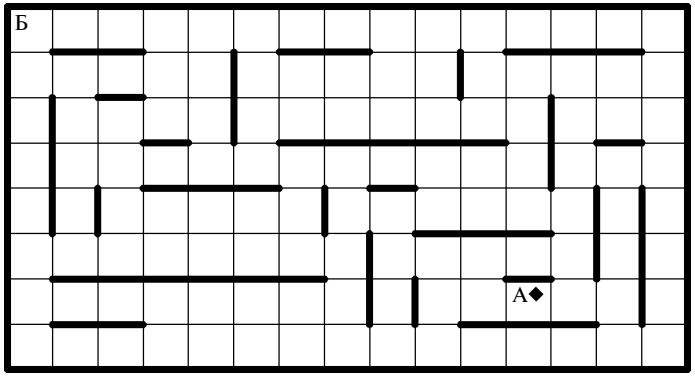
После первого выполнения тела цикла будет закрашена первая клетка коридора и Робот сместиться во вторую. После второго выполнения будут закрашены две клетки, а Робот окажется в третьей. После последнего — будет закрашен весь коридор, а Робот попадает в клетку «Б».

Таким образом, нам осталось только (этап «4») убедиться, что рано или поздно цикл закончится. В данном случае это очевидно, потому что при каждом выполнении тела цикла Робот смещается на клетку вправо и, значит, в конце концов выйдет из коридора. Несмотря на простоту, это рассуждение, показывающее, что в какой-то момент цикл закончится, все же нужно для контроля провести. В нашем случае это рассуждение оказалось тривиальным, но в более сложных задачах оно вполне может оказаться содержательным.

> **ЗАМЕЧАНИЕ**. Я должен еще раз заметить, что приведенная выше задача — одна из простейших. Мы не столько решаем задачу с помощью методики, сколько ***иллюстрируем*** методику с помощью задачи. Иллюстрируем, как должен ***думать*** человек, чтобы составить алгоритм.
>
> Приведенная выше задача очень простая, поэтому некоторые (достаточно сильные) ученики могут — именно из-за простоты задачи — не понять ни сути методики, ни зачем вообще нужна какая-то методика, когда решение «видно и так». Этим ученикам излагать методику лучше на более содержательном примере выхода из лабиринта в п. 9.14.

Пример содержательной задачи — выход в левый верхний угол лабиринта

Дано, что Робот находится внутри прямоугольного «лабиринта», огороженного с четырех сторон стенами. Внутри лабиринта есть прямолинейные вертикальные и горизонтальные стены, но они не касаются ни друг друга, ни наружных стен и не образуют никаких углов (рис. 42, стр. 70, п. 9.14 учебника).



Неизвестно, ни где в точности находится Робот, ни как именно расположены стены внутри «лабиринта». Надо переместить Робота в начало — в левый верхний угол лабиринта.

Вот это уже вполне содержательная задача. Почему? Дело в том, что в простых задачах (типа задачи о закрашивании коридора выше) из картинки сразу видно, куда Робот должен идти и что он должен делать. Траектория движения Робота просто сразу «видна».

Здесь, учитывая, что точное расположение Робота и стен неизвестно, это не так. Понятно, что надо двигаться куда-то влево и вверх, но это лишь «общее направление» — ведь в неизвестном месте «дорога» может оказаться загорожена стеной. Поэтому новая задача существенно сложнее закрашивания коридора.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Забегая вперед, скажу, что позже нам встретятся задачи, в которых мы будем составлять алгоритмы, не имея представления даже об «общем направлении» движения Робота — см., например, раздел «Рекурсия». А здесь мы лишь несколько усложнили задачу, оставив ее на таком уровне, чтобы было можно ее понять и решить без специальных методов алгоритмизации — за счет здравого смысла. Поэтому применение методики будет демонстрировать только, что за счет методики решение задачи можно сделать ***простым*** и почти рутинным.

ЕЩЕ ОДНО МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Как всегда, когда встречается хоть сколько-нибудь содержательная задача, я настоятельно вам рекомендую попытаться решить ее самостоятельно. Только в данном случае попробуйте кроме решения задачи, параллельно уследить за тем, ***как и над чем*** вы будете думать. А потом сравните ваше решение и ваш ход мысли с тем, что буду излагать я.

Тот же прием вы можете применять на уроках (а еще лучше, задавая соответствующую задачу на дом накануне того, как вы собираетесь излагать ее на уроке). Конечно, в этом случае лучше взять пример, решения которого нет в учебнике. Ну и, конечно, просьба проследить за ходом мысли, чтобы потом сравнить, — это задача для вас, а не для учеников.

Применение методики составления алгоритмов с циклом пока

Итак, решим эту задачу в соответствии с методикой, изложенной выше.

Этап «0». Поскольку размеры лабиринта и положение Робота неизвестны, нам придется использовать цикл **пока** (правило «0») .

Прежде чем перейти к собственно применению методики, напомню, что ее применение и составление алгоритма производятся в рамках некоторой «общей идеи». В данной задаче это идея «двигаться как-нибудь вверх и влево, пока не доберемся до клетки Б».

Этап «1». Сформулируем условие окончания цикла. В соответствии с постановкой задачи и «общей идеей» цикл должен кончиться, когда Робот окажется в клетке Б. Чем клетка Б отличается от других клеток, в которых Робот может оказаться «по дороге»? Из рисунка видно, что клетка Б — угловая: у нее есть и стена сверху, и стена слева. Во всех остальных клетках лабиринта хотя бы одной из этих двух

стен нет. Поэтому в качестве условия окончания цикла можно написать «слева стена **и** сверху стена». Противоположное («перевернутое») условие — условие ***продолжения*** цикла — тогда можно записать так: «слева свободно **или** сверху свободно».

```
нц пока слева свободно или сверху свободно
| какие-то действия внутри цикла
кц
```

> **ЗАМЕЧАНИЕ**. Заметьте, что мы опять не только придумали, чем клетка Б отличается от остальных, но и ***формально*** записали условие продолжения цикла на алгоритмическом языке. Правда, при этом мы «забежали вперед» — воспользовались правилами записи логических выражений (условий), которые будут изложены в § 10.
>
> Здесь, впрочем, у нас простейший случай — смысл этих условий абсолютно ясен и знания ***общих*** правил для их записи не требуется. Это такое же «забегание вперед», как и использование выражений $a + 1$ и $-a - 1$ (алгоритм А22, упражнения к § 6) до прохождения линейной записи арифметических выражений (§ 7).
>
> Тем не менее, очень важно, что и в алгоритме А22, и в разбираемом нами примере мы все записываем формально — на алгоритмическом языке. В учебнике мы ограничились замечанием, что условие «слева свободно **или** сверху свободно» в алгоритмическом языке записать можно, и что общие правила записи таких «составных» условий будут изложены в § 10.
>
> Если вы считаете такое «забегание вперед» недопустимым, то переставьте материал и пройдите пример из п. 9.14 после общего вида записи условий в алгоритмическом языке (п. 10.6).

Этапы «2» и «3». Теперь нам надо описать промежуточные состояния Робота после некоторого числа повторений тела цикла ***(этап «2»)***. Это, однако, достаточно трудно. Ясно, что за шаг цикла ***(этап «3»)*** Робот должен сместиться либо влево, либо вверх, либо и влево, и вверх. То есть он должен приблизиться к цели — к левому верхнему углу лабиринта. Однако как-то более точно описать состояния Робота трудно.

И именно поэтому — коль скоро подзадача смещения Робота куда-то влево-вверх оказалась содержательной, — совершенно естественно воспользоваться методом последовательного уточнения: назвать как-нибудь это действие, например, «сместиться к углу» (как в учебнике) и записать основной алгоритм, используя внутри цикла вызов вспомогательного алгоритма «сместиться к углу»:

```
нц пока слева свободно или сверху свободно
| сместиться к углу
кц
```


(Полностью, т. е с заголовком, **дано**, **надо**, алгоритм А35 приведен в учебнике.)

Обратите внимание, что за счет использования вспомогательного алгоритма (который пока еще не написан) этап «3» у нас практически исчез. Ведь неформальное решение о том, что должно происходить при однократном выполнении тела цикла, мы принимаем на этапе «2» — сместиться куда-нибудь и насколько-нибудь влево и/или вверх. А формально нам это теперь записывать ***в теле цикла*** не надо — мы написали вызов вспомогательного алгоритма. Если говорить более точно, то окончательное решение вопросов о том, каковы промежуточные состояния Робота (этап «2») и как должен смещаться Робот за один шаг цикла (этап «3»), мы ***отложили*** до момента составления вспомогательного алгоритма «сместиться к углу».

Этап «4». Теперь нам надо убедиться, что цикл рано или поздно закончится. Здесь можно использовать следующее рассуждение: если при каждом выполнении тела цикла расстояние от Робота до левого верхнего угла (или сумма расстояний до левого и верхнего края лабиринта) хоть на сколько-нибудь уменьшается, то, значит, Робот рано или поздно попадет в левый верхний угол и цикл закончится.

На этом, собственно, и применение методики, и составление цикла **пока** в алгоритме «в левый верхний угол лабиринта» закончены. У нас, правда, остались ***отложенные*** вопросы — ведь нам еще надо составить алгоритм «сместиться к углу». Но цикл **пока** ***мы уже составили***.

Метод последовательного уточнения замечателен тем, что теперь, решая подзадачу «сместиться к углу», мы про основную задачу, про цикл **пока** и все, что с ним связано, можем забыть. Нам надо составить алгоритм «сместиться к углу», который должен приблизить Робота к левому-верхнему углу — ***как-нибудь и насколько-нибудь***. Мы можем не вспоминать про циклы, про то, что таких смещений будет

много. Мы можем ***думать*** над этой подзадачей, как над отдельной и независимой: Робот в какой-то клетке лабиринта (отличной от Б), надо сместить его влево-вверх — ***все равно куда***, чтобы Робот стал ближе к Б.

Если мы такой алгоритм составим, то можно будет выполнить основной алгоритм и получить результат. В связи с использованием вспомогательных алгоритмов и метода последовательного уточнения этапы нашей методики в значительной степени оказались преобразованными в формулирование требований к вспомогательному алгоритму (формулирование его **дано** и **надо**):

```
дано | Робот внутри (не в "Б") в лабиринте заданного вида
надо | Робот стал ближе к клетке "Б" -- все равно как
```

Теперь, составляя разные алгоритмы «сместиться к углу», удовлетворяющие этим **дано** и **надо**, мы можем получать разные решения исходной задачи. В учебнике приводится алгоритм (А36), с телом, состоящим из двух строк:

```
вверх до стены
влево до стены
```

За этими двумя строками скрываются два вызова вспомогательных алгоритмов (типа алгоритма А31 «вниз до стены»), в каждом из которых выполняется свой цикл **пока**.

Важно убедиться, что ключевое требование «Робот стал ближе к клетке «Б» после выполнения этих команд будет выполнено. Действительно, команда вызова вспомогательного алгоритма «вверх до стены» перемещает Робота хотя бы на одну клетку вверх во всех случаях, за исключением ситуации, когда в исходном положении сверху над Роботом вплотную есть стена. Но если над Роботом сверху стена, то по условию в **дано** слева от Робота стены нет. Следовательно, в этом случае Робот сместиться хотя бы на одну клетку влево при выполнении команды «влево до стены». Другими словами, во всех случаях хотя бы на одну клетку то ли вверх, то ли влево Робот сместится.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Конечно, это решение чуть-чуть искусственно. Школьникам гораздо проще было бы здесь написать смещение Робота на одну клетку:


```
если сверху свободно
| то вверх
| иначе влево
все
```

Но, формально, конструкция **если** в этот момент школьникам еще не известна. Отсутствие конструкции **если** ***усложняет*** решение задачи, поэтому в § 9 эта задача по уровню оказывается несколько более сложной, чем после § 10. Тем не менее, она вполне «решаема» и — что очень важно — показывает, что за один шаг цикла, за одно выполнение тела цикла Робот может смещаться «на много». Здесь очень полезны задачи типа: «отметьте точками положения Робота после первого выполнения тела цикла; после второго; . . .». Обратите также внимание на упр. 14 после параграфа. С другой стороны, если в ходе решения задачи школьники придумают конструкцию **если** (помните проблемный подход?), то я рекомендую это всячески приветствовать, дать им форму записи конструкции **если** в алгоритмическом языке и двигаться дальше.

В конце параграфа «для порядка» приводится решение задачи про лабиринт без каких-либо «забеганий вперед» — алгоритм А37. Этот алгоритм верный, но первую строку в нем можно без ущерба для результата вычеркнуть. Тогда останется алгоритм:

```
влево до стены
нц пока сверху свободно
| вверх до стены
| влево до стены
кц
```

Вся его «хитрость» состоит в том, что в теле цикла Робот ходит «повернутой буквой Г» (вверх до стены, влево до стены) ***от одной «левой» стены до другой.*** Т. е. всякий раз до и после выполнения цикла Робот стоит вплотную к стене, расположенной слева от него. Вы можете сформулировать только что сказанное, в качестве «общей идеи» и попросить школьников самостоятельно составить алгоритм, используя методику п. 9.11. Единственная сложность будет с «убиранием начальной нерегулярности» — ведь ***до цикла*** надо переместить Робота в клетку, слева от которой есть стена (команда

вызова вспомогательного алгоритма «влево до стены»), а методика относится только к составлению самого цикла **пока**.

Упражнения

Решения всех упражнений этого параграфа есть в методическом пособии [Авербух]. Я, со своей стороны, лишь замечу, что для Робота естественны задачи с коридорами, которые я вам и рекомендую. В особенности — основные для данного параграфа упражнения 7а)–7г).

В задачах, где Роботу надо пройти по краю некоторого прямоугольника (упражнения 9, 10, 12, 13), приходится практически одно и то же записывать минимум четыре раза, меняя «вправо» и «снизу стена» на «вниз» и «слева стена» и т. п., поскольку Робот не имеет ориентации. Эти задачи можно решить красиво и записать компактно с циклом 4 **раз**, если использовать вместо Робота исполнителя «Путник» [ПдМ].

В то же время задачи на закраску прямоугольника (упр. 2, 7д) хороши для задействования и цикла **пока**, и вспомогательных алгоритмов. Вообще, при решении ряда упражнений можно в качестве вспомогательных использовать алгоритмы решения других упражнений.

§ 10. Условия в алгоритмическом языке. Команды **если** и **выбор**. Команды контроля

Этот параграф очень прост. Единственное сколько-нибудь неочевидное место — это запись составных условий в п. 10.6. Овладение командами **если** и **выбор** после цикла **пока** не вызывает никаких сложностей. Команда контроля **утв** тоже очень проста. Хотя о ней говорится достаточно бегло и ответ на вопрос «зачем она нужна?» остается в этом параграфе недораскрытым.

Самым сложным в параграфе является разбор примеров и решение упражнений. Поскольку команды ветвления завершают изложение управляющих конструкций (части «до величин»), после этого параграфа можно решать любые задачи, не требующие работы с величинами. Кроме того, основные примеры параграфа — это алгоритмы, содержащие цикл **пока**, внутри которого используется команда **если**. Алгоритмическая сложность у такой конструкции выше, чем у цикла **пока**. Хотя цикл **пока** к этому моменту уже усвоен, а сама команда **если** очень проста, тем не менее примеры получаются достаточно содержательные.

Технически параграф можно разбить на четыре части:
(1) команда **если** (пп. 10.1–10.5);
(2) условия (п. 10.6);
(3) команда **выбор** (пп. 10.7–10.9);
(4) команды контроля (пп. 10.10–10.11).

Команда **если**

Как обычно при проблемном подходе, изложение начинается с постановки задачи («разметка выходов из коридора»), в которой для записи решения необходимо задействовать команду **если**. Ввиду общей простоты команды все ученики, как правило, придумывают не только конструкцию, но и форму записи, практически совпадающую с требуемой алгоритмическим языком. Учителю нужно лишь при представлении формы записи этой конструкции в алгоритмическом языке обратить внимание учеников на наличие «закрывающей» структурной скобки **все**, объяснив, зачем она нужна.

Поскольку конструкция **если** очень проста, то ее пояснение в учебнике (если не считать картинки с демонстрантами

на с. 77) ограничено общим описанием (п. 10.2) и графической блок-схемой работы ЭВМ (п. 10.3). Но вы можете, если сочтете это необходимым, поразбирать и диалоги «ЭВМ—Робот» при выполнении этой конструкции.

Ничего более содержательного по существу этой темы мне сказать нечего. Но вас, как учителей, я думаю, должен интересовать вопрос, почему команда **если** появилась только теперь (ведь она намного проще цикла **пока**), почему в учебнике нет примеров на команду **если** саму по себе (не внутри цикла) и пр. Поэтому я сделаю два небольших замечания о нашем подходе и методике.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Как я уже не раз говорил (и еще не раз скажу), мы старались демонстрировать что бы то ни было на вполне содержательных задачах, которые без ЭВМ, без компьютера решить либо невозможно, либо очень сложно. Мы всегда и всюду стремились демонстрировать наличие у информатики собственного содержания, собственного предмета.

Конечно, легко ввести и пояснить команду **если** на каком-нибудь примере типа «осторожного» шага: «**если** снизу свободно **то** вниз **все**». Беда, на наш взгляд, состоит в том, что если с этого начать (а особенно, если ввести команду **если** до циклов), то у школьников в голове не возникает ***никакого*** (пусть не формализованного) внешнего контекста, в рамках которого написание такой команды было бы осмысленным ***для решения каких-то содержательных задач***.

После того, как разобран хотя бы один пример с ***циклом*** и с **если** внутри, такой внешний контекст появляется. Тогда уже можно команду **если** изучать и без циклов — ученики представят себе обстановку, в которой нужда в такой команде возникает. Но когда этот первый пример (с циклом и **если**) разобран, в команде **если** уже нечего изучать. Другие ***иллюстрационные*** примеры просто не требуются, — нужно решать задачи.

Поэтому мы считаем, что впервые команда **если** должна появиться внутри цикла. Правда, это может быть цикл n **раз**. Вполне допустимо объединить команды обратной связи с командой **если**, а цикл **пока** пройти позже, фактически поменяв местами **пока** и **если**.

Наконец, я еще раз хочу подчеркнуть, что блок-схемы у нас носят чисто иллюстративный характер и используются исключительно и только для пояснения смысла вновь вводимых конструкций. Никогда и ни при каких условиях их не следует использовать при ***составлении*** алгоритмов.

Условия в алгоритмическом языке

Общая форма записи условий в алгоритмическом языке излагается в учебнике на примерах и использует хорошо известную школьникам форму записи равенств и неравенств. Логическая связка **и** также знакома и понятна школьникам. Поэтому при практическом изучении этой темы внимание следует уделить связке **или**, которая чуть менее проста, а также — и в основном — построению сложных составных условий, содержащих **и**, **или**, **не** и скобки.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. Для тех, кто знаком с другими языками программирования, я хочу особо отметить, что в школьном алгоритмическом языке (и в системе «КуМир») допустимы условия вида $a = b < c = d$, $0 \le x \le 1$ и др. То есть неравенства в алгоритмическом языке записываются также, как в математике (с той лишь разницей, что используется линейная запись арифметических выражений). Формально условие $a < b < c$ эквивалентно условию $a < b$ **и** $b < c$.

Хороший класс задач на составление условий — это описание на алгоритмическом языке разных ситуаций на поле Робота. Вы можете, например, попросить учеников записать формально условие «Робот стоит в углу»:

```
(слева стена и сверху стена) или
(сверху стена и справа стена) или
(справа стена и снизу стена) или
(снизу стена и слева стена)
```

или условие «Робот стоит у стены, но не в углу»:

```
      (сверху стена или снизу стена или
       справа стена или слева стена)
и не  (слева стена и сверху стена) или
      (сверху стена и справа стена) или
      (справа стена и снизу стена) или
      (снизу стена и слева стена))
```

Замечу, что последнее условие можно записать, например, и так:

```
(справа свободно и слева свободно и
(сверху стена или снизу стена)) или
(сверху свободно и снизу свободно и
(справа стена или слева стена))
```

Вы также можете задавать задачи, в которых условие уже написано, а от ученика требуется ответить, в каких клетках оно выполнено, а в каких — нет.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Один из моих знакомых учителей, Сергей Александрович Бебчук, придумал давать ученикам трехминутные контрольные, на которых ученикам раздаются карточки. На каждой карточке изображена горизонтальная полоса из десяти клеток, некоторые из них закрашены, кое-где есть какие-то стены. Кроме того, на карточке написано условие на алгоритмическом языке, содержащее **и**, **или**, **не**, скобки и команды-вопросы Робота. Ученик должен быстро написать ряд из десяти плюсов и минусов, соответствующих этим десяти клеткам: «+» — если условие выполнено, «–» — если не выполнено. Всего карточек (вариантов контрольной) несколько десятков (больше, чем учеников в классе), так что каждый ученик получает индивидуальное задание. Ответы на все варианты (строчки из «+» и «–») сведены в одну таблицу, которая есть у учителя. Получив от учеников решение, учитель может (пока ученики думают над очередной задачей) мгновенно сверить его с правильным ответом и тут же поставить оценку. Поэтому к концу урока (в крайнем случае с «захватом» нескольких минут на перерыве) он имеет возможность всему классу выставить оценки за контрольную.

На следующем уроке карточки перемешиваются и раздаются вновь. В итоге, практически не тратя специального времени, параллельно с другими темами ученики усваивают запись логических выражений (условий).

Можно в таком же стиле организовать и прохождение линейной записи арифметических выражений, с той лишь разницей, что на проверку решений учеников придется тратить время.

Естественно, вы можете и не использовать Робота, а задавать задачи типа упражнения 2, где нарисована некоторая область на плоскости и надо написать условие, задающее это

множество точек (т. е. условие, которое в точках множества выполнено, а вне множества — нет).

Команда выбор

Команда **выбор** — это первая «не необходимая» команда алгоритмического языка в нашем учебнике в том смысле, что невозможно сформулировать задачу, которую без этой команды нельзя решить. Команда **выбор** лишь ***упрощает*** запись в ситуации, когда вариантов много. Но и в этой ситуации алгоритм вполне можно записать, используя команду **если**. (Впрочем, с другой стороны, верно и обратное, — любой алгоритм можно записать используя не команду **если**, а команду **выбор**.)

Соответственно, если расставлять акценты, то я должен сказать, что для развития алгоритмического стиля мышления вполне достаточно изучить одну из этих двух конструкций (причем в такой постановке лучше изучать **выбор**). Изучение после этого второй конструкции практически не дает никакого вклада в алгоритмическую культуру, но позволяет более естественно записывать алгоритмы в ситуации с одним-двумя (**если**) или тремя и больше (**выбор**) вариантами.

Поскольку, однако, изучение команды **выбор** после команды **если** почти не требует времени, а запись некоторых алгоритмов (например, А91 и А92) без команды **выбор** существенно усложняется, мы эту команду ввели. Все объяснение смысла команды **выбор** в учебнике сводится к блок-схеме работы ЭВМ при выполнении этой конструкции и простому примеру ее использования (алгоритм А41).

Команды контроля

Всего в алгоритмическом языке используется три конструкции контроля: **дано**, **надо** и **утв** (сокращение от «утверждение»). Во всех в них может быть либо записано так называемое ***контрольное условие***, либо не написано ничего, либо написан только комментарий. Последний случай — самый частый в нашем учебнике. Другими словами, в школьном курсе **дано**, **надо** и **утв** используются для пояснения алгоритмов человеку, а не для формальной проверки условий в ходе выполнения алгоритма ЭВМ.

Соответственно, при желании, вы можете объявить эти конструкции «предназначенными для комментариев» и — практически без ущерба для целей курса — объяснение их реального смысла пропустить.

В отличие от конструкций **дано** и **надо**, являющихся частью записи алгоритма, команда **утв** — это команда, т. е. она может быть помещена в произвольное место в теле алгоритма. Условие в **дано** проверяется перед началом выполнения алгоритма, в **надо** — после окончания выполнения алгоритма, а в **утв** — в том месте, где эта команда написана.

Необходимость **дано** и **надо** в учебнике вообще не обсуждается. А ответ на вопрос «зачем нужна команда **утв**?» в учебнике сведен к бытовой аналогии с объяснением дороги («заверните за угол, там будет видна булочная»), где «будет видна булочная» — контрольное условие, подлежащее проверке. И — так же, как и в жизни — если контрольное условие выполнено, то надо просто продолжать дальше выполнять алгоритм. Убедились, что все в порядке — и работаем дальше. Если же контрольное условие не выполнено (после поворота за угол булочная не видна), то значит, где-то есть ошибка: либо алгоритм неверен, либо он применяется в недопустимой для него ситуации, либо еще что-то.

С более содержательной точки зрения значимость уже столь привычных нам **дано** и **надо** состоит в том, что в них описываются ***состояния*** объектов (точнее, условия, которым должны удовлетворять состояние Робота, значения величин и пр.), в отличие от остальных команд, суть которых состоит в выполнении некоторых ***действий***. Даже если указанные условия записаны неформально — в виде комментариев, то они позволяют понять, ***что*** делает (должен делать) алгоритм, не разбираясь с тем, ***как*** он это делает. То есть можно использовать алгоритм, не зная, как он записан внутри, что дает возможность при составлении алгоритма не думать о том, где и как он используется.

Другими словами, это полный аналог формулировки теоремы в математике, средство, позволяющее нам структурировать и накапливать знания (алгоритмы).

Если же говорить о значимости команды **утв**, то в рамках учебника (за исключением единственного алгоритма А114, где записано формальное контрольное условие) эта команда используется исключительно для комментариев, для пояснения алгоритма. Заметьте однако, что комментарии в команде **утв** описывают ***состояния*** Робота и величин в момент ее выполнения. Благодаря этому алгоритм в целом или некоторые его части становится возможным понимать статически, понимать ***что*** делает, ***чего*** достигает данный фрагмент алгоритма. Это отличается от анализа и восприятия того, какие ***действия*** происходят при выполнении алгоритма.

С методической точки зрения введение команды **утв** здесь необходимо исключительно как введение формы записи. Чтобы в дальнейшем, когда в каком-нибудь алгоритме встретится запись **утв** <условие>, школьники смогли прочесть ее как «утверждение: в этот момент выполнено <условие>».

> **МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Хочу обратить ваше внимание на то, что в **дано** и **надо**, как и в **утв** записываются условия, описывающие ***состояния***, например положение, Робота. И я настоятельно вам рекомендую — ***даже когда ученики пишут комментарии*** — внимательно следить за ***стилем*** этих комментариев, чтобы они описывали ***состояния***, а не действия. Другими словами, чтобы ученики после **надо** писали «Робот в противоположном углу», а не «переместить Робота в противоположный угол». Тогда им будет гораздо легче перейти к записи формальных условий.

Упражнения

Упражнения после параграфа можно разбить на две группы: первая — очень простые упражнения 1–4; вторая — содержательные упражнения 5–8. Решения всех упражнения есть в методическом пособии [Авербух]. Я настоятельно рекомендую вам прочесть эти решения, а также комментарии и замечания к ним (с. 173–178 методического пособия). И обратите внимание на замечание про «подводные камни» на с. 176, которое достаточно важно и заслуживает обсуждения с учениками.

Задачи 5–8 уже достаточно интересны. Их лучше решать последовательно, используя при решении следующих задач как вспомогательные алгоритмы, которые были составлены при решении предыдущих. Это — задачи с циклами **пока**, с командами **если**, с использованием вспомогательных алгоритмов, т. е. вполне достойные и содержательные.

На этом раздел «до величин» заканчивается

Заканчивается важная часть курса, часть «до величин». Обращаю ваше внимание, что к этому моменту вы с учениками прорешаете массу вполне содержательных, сложных задач, не используя понятия величины, без единого оператора присваивания. Это — важная методическая особенность, отличающая наш курс от всех остальных, мне известных. В других курсах величины (переменные) и оператор присваивания вводятся в самом начале — и все начинается с задач на вычисление каких-то величин.

Если вы почему либо считаете, что величины, операторы «присваивания» и «ввода-вывода» надо ввести сразу — ну просто такая у вас точка зрения, то, мне кажется, наш учебник вам не очень подходит.

По нашему учебнику тоже не обязательно идти так, как я здесь излагал: параграф за параграфом, конструкция за конструкцией. Можно весь материал «до величин» сразу «выплеснуть» на учеников за один урок (если класс достаточно высокого уровня — в слабых классах это не очень проходит), а потом решать задачи. Но тем не менее часть «до величин» должна быть выделена. Если вы решаете вводить величины с самого начала, то это уже другой методический подход и по-другому построенный курс.

Лекция 6.

Ну а теперь мы приступаем ко второй части главы 1 «Алгоритмический язык». Вся первая часть была связана с действиями, с записью действий на алгоритмическом языке, с конструкциями алгоритмического языка, позволяющими структурировать действия (вспомогательные алгоритмы), либо задавать порядок выполнения действий (управляющие конструкции — циклы и ветвления). Теперь мы переходим ко второй части, ко второму фундаментальному понятию информатики — понятию объекта, называемому в школьном курсе «величиной».

Термин «величина», который используется в школьном учебнике и в алгоритмическом языке, ввел Андрей Петрович Ершов, заимствовав его из математики и физики и считая, что так школьникам будет проще для понимания. Обычно в программировании используется термин «переменная», который, впрочем, сейчас все чаще вытесняется термином «объект». Термин «переменная» в свое время тоже заимствованный из математики, уже начал уходить в прошлое. Сейчас чаще говорят про объект и состояние объекта, а также — более широко — про объектно-ориентированный подход в программировании. Но в школьном курсе вместо слов «объект» и «состояние» объекта, используются термины «величина» и «значение» величины. Дальше я буду пользоваться терминологией школьного учебника.

§ 11. Величины в алгоритмическом языке. Команда присваивания

Методика введения величин — проблемный подход

Как всегда при проблемном подходе, все начинается с постановки задачи, для решения которой нам понадобятся величины. Задача, рассматриваемая здесь такова: Робот расположен вплотную над горизонтальной стеной неизвестных размеров, расстояние от Робота до краев стены неизвестно, надо провести Робота «сквозь стену» на одну клетку вниз. Поскольку Робот сквозь стены ходить не умеет, т. е. просто скомандовать «вниз» нельзя, то надо заставить Робота эту стену как-то обойти.

И, как всегда, основным методическим приемом на старте выступает противопоставление схемы программного уп-

равления непосредственному управлению Роботом «вручную». Если поставить школьникам задачу «вручную» прокомандовать Роботом (при условии, что размеры стены им неизвестны, т. е. учитель им поле и стену не показывает), то довольно быстро они сообразят, что нужно идти к одному из краев стены, например, вправо, по дороге все время спрашивая «снизу стена?» и считая число сделанных шагов. А когда стена кончится, скомандовать «вниз» и сделать «влево» столько шагов, сколько перед этим сделано шагов «вправо». Если над стеной Роботу 7 раз было скомандовано «вправо», то после шага вниз, ему надо 7 раз скомандовать «влево», чтобы он попал в нужную клетку.

Когда школьники все это сообразят и будут в состоянии решить задачу при управлении Роботом «вручную», следует обратить их внимание, что при этом была задействована их — школьников — голова: они запоминали число сделанных «вправо» шагов. Если теперь от управления Роботом «вручную» перейти к написанию алгоритма, который потом будет выполнять ЭВМ (т. е. перейти к схеме программного управления), то нам необходимо как-то записать в алгоритме, что ***ЭВМ*** должна запоминать число выполненных Роботом команд «вправо» и использовать эту информацию при обратном движении влево. Это и есть тот новый материал, которому посвящен параграф.

После того, как проблема запоминания информации в памяти ЭВМ понята и осознана, вы можете в зависимости от вашего настроения либо просто повторить модель памяти ЭВМ и ввести понятие «величины», либо (прежде чем излагать материал параграфа) дать возможность ученикам придумать свои формы записи для задания величин.

Вообще говоря, «придумывание» таких форм записи можно начинать и раньше. Вернемся на минутку к вспомогательным алгоритмам «квадрат 4», «квадрат 6» и т. д. Уже тогда можно было просить школьников придумать что-нибудь для задания алгоритма «квадрат» с произвольной длиной стороны. Поскольку какая-то математическая культура у них есть, они обычно длину стороны как-то обозначают, после чего совершенно естественно возникает запись «**арг вещ** a».

Здесь все то же самое, только сначала лучше напомнить ученикам про модель памяти ЭВМ в виде классной доски: когда начинает выполняться какой-то алгоритм, в памяти под него отводится место. Тогда дальше, поскольку форма описания аргументов алгоритмов уже была, ученики вполне могут придумать не только смысл величин (отведение памяти внутри прямоугольника алгоритма), но и даже более или менее правильную форму их описания.

Соответственно, именно здесь появится понятие «промежуточной» величины — величины, которая нам нужна «на время» для подсчета числа сделанных шагов. Здесь же можно детально изложить, как ЭВМ отводит место в памяти для промежуточных величин.

Проблемный подход и команда присваивания

А вот дальше, для перехода к команде присваивания, очень полезно попросить учеников снова решить исходную задачу, но уже с использованием нарисованного на доске прямоугольника для величины «**цел** n», в которой должно подсчитываться число сделанных Роботом шагов. Это будет такой переходный вариант, когда ученикам придется выполнять работу ЭВМ, и, вместо подсчета числа шагов в уме они вынуждены будут вписывать, стирать и снова вписывать значение числа шагов внутрь прямоугольника «**цел** n». Первый шаг вправо сделали — записали в прямоугольник на доске «1», сделали второй шаг — «1» стерли и написали «2» и т. д.

После этого уже можно говорить, что, как мы видели, для работы с памятью ЭВМ надо уметь не только выделять кусок памяти, но и узнавать (читать из памяти) информацию, которая в ней хранится, а также стирать старое значение и записывать новое.

Получается всего три операции: отведение места в памяти, чтение из памяти, запись новой информации в память.

Первые две из этих трех операций почти ничем не отличаются от работы с аргументами алгоритмов и «де факто» уже знакомы школьникам. А для записи информации в память в алгоритмическом языке (как и во всех процедурных языках программирования) есть специальная команда — команда присваивания (пп. 11.6–11.7).

Поскольку собственно с величинами (переменными) и командой присваивания на примере того или иного языка программирования все вы прекрасно знакомы, я лишь расставлю некоторые акценты и поясню то, что специфично для нашего курса.

Использование модели памяти ЭВМ

Все, что касается величин, можно и нужно излагать и пояснять на модели памяти ЭВМ. Как я уже говорил «доска» — чрезвычайно точная модель, которая в деталях соответствует тому, как ЭВМ работает со своей памятью.

В частности, обязательность ***описания*** величины связывается с необходимостью в какой-то момент нарисовать прямоугольник этой величины. Говорится, что компьютер, начав выполнять алгоритм «сквозь стену», отведет место для этого алгоритма («нарисует большой прямоугольник»), после чего, встретив строку «**нач** **цел** n», внутри большого прямоугольника алгоритма «сквозь стену» отведет место («нарисует маленький прямоугольник») для целочисленной величины n. Именно поэтому величину необходимо описывать в строке **нач**.

Прямоугольник величины рисуется внутри прямоугольника алгоритма, откуда следует, что в разных алгоритмах можно использовать одно и то же имя, но промежуточные величины будут ***разными***. Это свойство промежуточных величин в «большой» информатике называется ***локализацией***: промежуточные величины ***локализованы*** в алгоритме по тексту (то же имя в другом алгоритме обозначает другую величину) и по времени существования (прямоугольник величины рисуется в начале выполнения алгоритма и стирается по завершении выполнения алгоритма).

Использование значения величины в терминах модели памяти объясняется так же, как и использование аргументов: встретив имя величины, ЭВМ вместо имени возьмет и подставит значение из прямоугольника величины.

Изложение команды присваивания также происходит на базе модели памяти: сначала ЭВМ вычисляет значение выражения в правой части команды присваивания, подставляя, при необходимости, вместо имен величин их значения, а затем старое значение величины стирается и внутрь прямоугольника записывается новое. (Это, пожалуй, единственный недостаток модели — в реальности внутри ЭВМ эта одна операция: при записи нового значения «внутрь прямоугольника» величины старое значение автоматически пропадает. В нашей же модели мы сначала «стираем», чего в реальности не делается, а лишь потом записываем.)

Понятия, связанные с понятием величины

Очень важным является п. 11.2, где вводятся такие характеристики величины, как ***имя, значение, тип*** (целочисленная, вещественная и пр.). При желании вы можете прямо здесь рассказать еще и про ***вид*** величины (аргумент, результат, промежуточная величина алгоритма, общая величина исполнителя). Все ***типы*** и ***виды*** величин алгоритмического языка перечислены на третьей странице форзаца. Четверка ***имя, значение, тип, вид*** — это ***полный*** набор понятий, связанных с величинами. И ученики должны ***мгновенно*** отвечать на вопросы типа: «каково ***имя*** этой величины?», «какого ***типа*** эта величина?», «какого ***вида*** эта величина?». Заметьте также, что ***имя, тип*** и ***вид*** величины фиксированы и определяются по тексту алгоритма, а ***значение*** величины может меняться в ходе выполнения алгоритма. Это все простые понятия, но важные для дальнейшего. И обратите внимание, что мы не случайно в нашей модели памяти ЭВМ пишем ***имя***, ***тип*** и ***вид*** величины над прямоугольником величины (внутри которого пишется ***значение*** величины).

Разделение труда между ЭВМ и исполнителями

По-прежнему мы считаем существенным и продолжаем подчеркивать разделение труда между ЭВМ и исполнителями. Важно понимать, что понятие промежуточной величины, выполнение команд присваивания, вообще все действия с величинами и их значениями осуществляет ЭВМ. Исполнители по-прежнему не умеют выполнять ничего, кроме своих команд. Чтобы прояснить это, на с. 90 учебника приведена полная последовательность конкретных команд, которые ЭВМ выдаст Чертежнику в процессе выполнения алгоритма рисования параболы А45.

С другой стороны, такое разделение, на наш взгляд, способствует усвоению собственно конструкций алгоритмического языка, их смысла, пониманию того, что и как делает ЭВМ в ходе их выполнения.

Фактически этим все и исчерпывается. Параграф, кроме того, содержит несколько примеров алгоритмов, работающих с величинами. Помимо очень простых алгоритмов «вниз сквозь стену» и «закрасить радиоактивные клетки», здесь рассматривается и вполне содержательная задача рисования графика функции на примере параболы.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Мы сделали все, чтобы этот переход от алгоритмов без величин к алгоритмам с величинами прошел незаметно, как простая вещь. Модель памяти была введена в § 6 при описании аргументов алгоритмов. Поэтому здесь она должна казаться знакомой: просто эта модель применяется для описания нового явления — работы ЭВМ с промежуточными величинами. Использование имени величины в выражениях ничем не отличается от использования имен аргументов, что тоже уже было, причем с тем же самым смыслом. Единственное, что появилось действительно нового — это команда присваивания, которая позволяет устанавливать и изменять значение величины. Вот, собственно, и все. Замечу еще, что термин «присваивание» — пожалуй, единственный, который при всей своей странности перекочевал в школьный курс — в первые учебники А.П. Ершова — в неизменном виде (что, видимо, связано с полным отсутствием какого-либо аналога для этого термина в математике и физике).

Упражнения

Естественно, что после введения величин класс задач, которые мы можем решать, существенно расширяется.

Упражнения к параграфу можно разделить на две части. Первая часть (упр. 1–4) — просто отработка команды присваивания, т. е. упражнения типа «до команды присваивания значения величины были такие-то, скажите, какими они будут после выполнения команды» или, наоборот, «после команды стали равны тому-то, скажите, какими они были до». Очень хорошее упражнение здесь — это упражнение 4, где из четырех вариантов обмена значений величин x и y надо указать, где в результате значения действительно будут обменены, а где — нет. Эти упражнения очень просты и нацелены просто на лучшее усвоения команды присваивания и понятия величины.

Упражнения 5 и 6 носят промежуточный характер — алгоритм уже задан, надо понять, как он будет работать в заданной ситуации (упр. 5), или что он вообще делает (упр. 6). В упражнении 5 основное, что должны сделать ученики, — сказать, чему будет равно значение величины n в момент окончания работы алгоритма (или отказа).

Вторая часть (упр. 7–9) — нормальные задачи на составление алгоритмов, в которых для решения задачи надо использовать величины и команду присваивания. Решения всех этих задач вы можете найти в методическом пособии [Авербух].

Повторю, что в целом этот параграф очень важный, — он вводит второе фундаментальное понятие информатики из той четверки, о которой я вам говорил. Это второе фундаментальное понятие, понятие объекта (величины) будет потом более глубоко раскрыто в § 14 (табличные величины). Но в силу проведенной подготовительной работы (модель памяти ЭВМ, работа с аргументами) сам материал параграфа оказывается достаточно простым, во всяком случае намного проще, чем циклы с **если** внутри, которыми закончился § 10.

§ 12. Алгоритмы с результатами и алгоритмы-функции

Этот параграф идет вслед за введением величин, но посвящен совсем другому — углублению третьего фундаментального понятия информатики, понятия вспомогательного алгоритма (подпрограммы с параметрами).

В § 6 были введены понятия вспомогательного алгоритма и вспомогательного алгоритма с аргументами. Здесь вводятся понятие ***результатов*** вспомогательного алгоритма и понятие ***алгоритмов-функций***. И то и другое позволяет осуществить аналог «обратной связи» — передать в основной алгоритм информацию, полученную в результате выполнения вспомогательного алгоритма.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Давайте я еще раз бегло пройду по этой четверке фундаментальных понятий информатики.

Первое понятие — команды и, прежде всего, циклы. Это § 4 (команда) и 8–9 (команды циклов). Все остальное (команда вызова, команда присваивания, команды ветвления и пр.) лишь разнообразит наше представление о том, какие бывают команды, но практически уже ничего не добавляет к фундаментальной сущности первого понятия информатики. Итак, это понятие можно считать полностью пройденным.

Второе понятие — объекты (величины) и, прежде всего, таблицы. Это § 11 (понятие величины) и 14 (табличные величины).

Третье понятие — вспомогательный алгоритм (подпрограмма) с параметрами. Это § 6 и 12.

Наконец, четвертое понятие — понятие исполнителя появится только в § 22.

ЕЩЕ ОДНО МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. То, что материал этого параграфа идет сразу вслед за величинами, отнюдь не случайно. С одной стороны, как я уже говорил, мы постарались все, связанное с вспомогательными алгоритмами, максимально приблизить к началу, чтобы это понятие по возможности глубже «впечаталось» в память учеников. С другой стороны, изложить понятие результатов алгоритма до величин невозможно, поскольку невозможно ни записать, чему должен быть равен результат, ни объяснить, куда он попадет. Эти два условия (максимально рано, но после величин) и определяют место темы «результаты алгоритмов» в нашем курсе.

Параграф в целом можно разбить на две части:

(1) ***виды*** величин и алгоритмы с результатами (пп. 12.1–12.7), включая общие правила выполнения команды вызова вспомогательного алгоритма (п. 12.4);
(2) алгоритмы-функции (пп. 12.8–12.11).

Между этими частями имеется достаточно сильное неравенство. Если первая часть носит фундаментальный, базовый характер, то вторая — про алгоритмы-функции — никаких фундаментальных сущностей не содержит и вообще может быть опущена без особого ущерба для алгоритмической культуры учащихся. Но — давайте по порядку.

Алгоритмы с результатами

Ввести и объяснить алгоритмы с результатами более или менее просто, и ситуация здесь полностью повторяет ситуацию с введением команд обратной связи: если раньше с помощью аргументов мы передавали информацию из основного алгоритма в вспомогательный, то теперь, используя результаты, можем передать в основной алгоритм информацию из вспомогательного.

Дальше, как всегда, вводится форма записи результатов (которая, ввиду ее полной аналогичности записи аргументов никаких вопросов вызывать не должна) и объясняется, как будет работать ЭВМ (на модели памяти ЭВМ с рисованием соответствующих прямоугольников для результатов и пр). Поскольку объяснять все это надо на каком-то примере, то используется алгоритм А47, который по значениям катетов прямоугольного треугольника вычисляет длину гипотенузы. Соответственно, объясняется, что результаты вспомогательного алгоритма копируются внутрь величин, указанных в вызове, перед завершением работы вспомогательного алгоритма и стиранием его прямоугольника из памяти ЭВМ.

С учетом этого уточнения мы уже получаем полную, завершенную схему выполнения вспомогательного алгоритма. Если нам нужно передать значения от основного алгоритма к вспомогательному, то используются аргументы, если от вспомогательного к основному, то результаты. Пользуясь аргументами и результатами мы можем устроить любой обмен информацией между основным и вспомогательным алгоритмами.

Это полная схема и просуммирована в п. 12.4, где изложено все, что мы прошли про вспомогательные алгоритмы. Кроме того, здесь — единственное место в учебнике, где написано, что на время выполнения вспомогательного алгоритма выполнение основного приостанавливается. Мы это неявно подразумевали, но нигде раньше даже не обсуждали. Здесь же, в п. 12.4, собраны воедино все правила, описывающие выполнение вспомогательного алгоритма и его связь с основным.

В этот момент, после п. 12.4, все уже пройдено. Теперь мы можем решить любую задачу, при необходимости передав информацию и от основного алгоритма вспомогательному и обратно.

> **МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Я обращаю ваше внимание, что в алгоритме «гипотенуза» уже не используются ни Робот, ни Чертежник. И таких алгоритмов, начиная с § 12, будет понемногу становится все больше и больше. Помните график роли понятия «исполнитель» в нашем курсе (с. 49), где к § 12 исполнители «сходят на нет»? Мы действительно прошли уже все необходимое для работы без исполнителей, для записи чисто информационных алгоритмов, которые выполняет только ЭВМ.
>
> Методическая роль исполнителей, позволивших плавно вводить конструкции алгоритмического языка и решать задачи разного уровня, полностью исчерпана.
>
> Позже мы вернемся к исполнителям, но это будут уже «совсем другие» исполнители, точнее, они будут играть совсем другую роль.

С задачами по этой теме ситуация следующая. Все задачи после параграфа формулируются в духе «написать алгоритм с такими-то аргументами и такими-то результатами». К сожалению, объем школьного курса не позволяет дойти до задач, в ходе решения которых эти алгоритмы нужно будет самим придумывать и вводить. В идеале школьники должны были бы овладеть не только понятием вспомогательного алгоритма с результатами, но и научиться самостоятельно выделять эти вспомогательные алгоритмы в задачах, формулировать их аргументы, результаты, **дано** и **надо**. Но для этого необходимо значительно увеличить сложность решаемых задач так, чтобы каждую пришлось разбивать на подзадачи, для которых и будут составляться вспомогательные алгоритмы. Поэтому в упражнениях после параграфа мы ограничились задачами на составление алгоритмов с уже заданными аргументами и результатами.

Алгоритмы-функции

С алгоритмами-функциями ситуация осложняется тем, что невозможно сформулировать задачи, которые было бы нельзя решить, не применяя алгоритмы-функции. Любой алгоритм, использующий вспомогательный алгоритм-функцию, всегда можно записать и без алгоритмов-функций, заменив их вспомогательными алгоритмами с результатами.

Вместо команды присваивания $a := f(x)$, где f — алгоритм-функция, мы всегда можем написать вызов $F(x, a)$, где x — аргумент, а a — результат. Всюду, где применяются алгоритмы-функции, можно заменить их на обычные алгоритмы с результатами.

Конечно, функции в ряде случаев удобнее. Сравните, например, эти две записи:

С использованием алгоритмов-функций	С использованием алгоритмов с результатами
$a := f(x1) + f(x2)$	$F(x1, a1)$ $F(x2, a2)$ $a := a1 + a2$

Вы можете смотреть на этот пример как на обоснование необходимости алгоритмов-функций. Алгоритмы-функции удобны тем, что мы можем сразу записать их в выражение и за счет этого упростить запись.

Но поскольку речь идет всего лишь об упрощении записи, то никакой фундаментальной сущности в алгоритмах-функциях нет. Это исключительно и только особая форма записи вспомогательного алгоритма с результатом. По существу же при использовании и алгоритмов-функций, и алгоритмов с результатами происходит одно и то же: из основного алгоритма вспомогательному передаются аргументы, обратно передается результат. Фундаментальная сущность у алгоритмов с результатами и алгоритмов-функций совпадает. Все отличия в форме записи и, в особенности, в форме использования (в форме вызова).

Поэтому подход у нас к алгоритмам-функциям такой же, как и к команде выбора, т. е. мы это понятие излагаем, считаем, что оно полезно и школьники должны уметь им пользо-

ваться. В ряде случаев это намного удобнее, чем применение алгоритмов с результатами, но не более того.

Технически изложение формы записи и порядка работы ЭВМ при вызове и выполнении алгоритма-функции, как обычно, опирается на модель памяти ЭВМ. Для значения функции, которое после выполнения алгоритма-функции будет передано в основной алгоритм, используется величина со специальным служебным именем **знач**. Заметьте, что **знач** — это и специальное имя, и отдельный ***вид*** величин алгоритмического языка. Внутри алгоритма-функции с величиной **знач** можно работать, как с любой другой, обычной величиной. По окончании выполнения алгоритма-функции значение величины **знач** подставляется в выражение вместо вызова алгоритма-функции.

Слово **знач** является сокращением от слов «значение функции» и, как и вся остальная терминология школьного курса информатики и школьного алгоритмического языка (алгоритм, аргумент, результат, величина, значение и пр.), было введено Андреем Петровичем Ершовым путем заимствования из математики.

Итогом всего этого параграфа, итогом овладения понятиями вспомогательного алгоритма, аргументов, величин, результатов, алгоритмов-функций и т. д. является составление алгоритма построения графика произвольной функции f, изложенного в п. 12.11. Точнее, здесь отдельно имеется алгоритм, который рисует график произвольной функции, а отдельно алгоритм-функция (в смысле информатики), который эту функцию (в смысле математики — увы, иначе и не скажешь) задает.

На этом изложение третьего фундаментального понятия информатики — понятия вспомогательного алгоритма — заканчивается. Тем самым, из четырех понятий, про которые я неустанно твержу с самого начала, два (циклы и вспомогательные алгоритмы) пройдены полностью. Величины затронуты, но пока не пройдены таблицы. А четвертое понятие — информационные модели исполнителей — даже еще и не упомянуто.

§ 13. Команды ввода/вывода информации. Цикл **для**

Этот параграф, как и § 7, носит характер своего рода «перерыва»: после содержательного § 12 про алгоритмы с результатами и алгоритмы-функции, это очень простой параграф с очень простым материалом, который можно рассматривать, как смену деятельности, как некоторый отдых. Параллельно ученики могут «подтянуть» уже пройденный материал, если кто-то что-то не успел.

В соответствии со своим названием параграф содержит две части:

(1) команды ввода/вывода информации;

(2) цикл **для**.

Эти две части практически друг с другом не связаны, но мы их пересекли за счет подбора задач, в которых используется и то, и другое.

Сейчас, когда курс информатики в школах преподается уже несколько лет, я думаю, нет необходимости рассказывать вам, что это за команды и как они работают.

Команды ввода/вывода

При объяснении этих команд вы можете задействовать ранее пройденный материал. Если, например, выполняя команду **ввод** n, человек наберет на клавиатуре число 25, то ЭВМ произведет ***в точности*** те же действия, что и при выполнении команды присваивания $n := 25$. Таким образом, число 25 станет значением величины n (будет записано внутрь «прямоугольника» величины n).

Обратите внимание, что, в отличие от Бейсика и Паскаля команда **вывод** в школьном алгоритмическом языке сама по себе не приводит к переходу на новую строку. Если мы хотим указать ЭВМ, что следующая порция информации должна выводится с новой строки, то надо явно написать служебное слово **нс** (сокращение от «новая строка») в команде вывода. Если же мы слово **нс** не пишем, то никакого неявного перехода на новую строку не осуществляется и разные команды вывода будут выводить информацию друг за дружкой в одну строку. Это, пожалуй, единственный нюанс школьного алгоритмического языка.

С другой стороны, ввод информации человек обычно заканчивает нажатием на клавишу «Enter» (т. е. переходом на

новую строку). Поэтому при чередовании команды вывода с командами ввода, как правило, никаких **нс** писать не надо. Другими словами в обычных простых диалогах можно обойтись и без **нс**.

В качестве примера я вам рекомендую алгоритм А57 со с. 104 учебника, вычисляющий размер вклада. Если вы этот алгоритм будете демонстрировать на компьютере, то я советую вам «поиграть» с процентами и понаблюдать, как растет вклад при разных процентных ставках.

ПРОГРАММНОЕ И МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Мы смогли отодвинуть команды ввода/вывода так далеко от начала курса по следующим причинам.

(1) Никакой фундаментальной сущности, никаких фундаментальных для развития алгоритмической культуры, алгоритмического мышления учащихся понятий за этими командами не скрывается (поэтому они и столь просты для изучения). Соответственно, в безмашинном курсе они и вовсе не нужны.

(2) В случае машинного курса использование системы «КуМир», в которой значения аргументов основного алгоритма ***автоматически*** запрашиваются у человека, а значения результатов автоматически выводятся на экран, вместо команд ввода/вывода можно оформлять соответствующие величины как аргументы и результаты алгоритмов (что соответствует сути дела). А до того, как введены понятия аргументов и результатов, можно просто менять числа внутри алгоритма и тут же запускать его на выполнение, наблюдая на «полях» алгоритма результаты его работы.

Соответственно, если у вас курс машинный, но адекватного программного обеспечения (системы «КуМир») нет, то, скорее всего, вам придется перенести изучение команд ввода/вывода гораздо ближе к началу курса.

Диалоговые системы и схема программного управления

Я также хочу обратить ваше внимание на небольшое замечание про диалоговые системы в п. 13.5. Это замечание вносит существенное изменение в схему программного управления, в которой раньше у нас человек не участвовал. Точнее, я должен признаться, что в схеме программного

управления слово «Человек» означало ***программиста***, того, кто составляет алгоритм.

В ходе выполнения алгоритма ЭВМ может выводить какую-то информацию для человека и вводить какую-то информацию от него. Здесь под словом «человек» понимается ***пользователь***, т. е. тот, кто использует ЭВМ вместе с алгоритмом для решения каких-то своих задач, например, указывая ЭВМ, какую именно деталь надо изготовить (вспомните про информационную индустрию — с. 71).

Таким образом, наша схема «программного управления» существенно усложнилась. Во-первых, мы уже знаем, что исполнителей может и не быть — алгоритм бывает «чисто информационным», вычислительным. Во-вторых, как мы только что выяснили, с помощью команд ввода/вывода ЭВМ может взаимодействовать с человеком в ходе выполнения алгоритма.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Впрочем, схема программного управления как картинка нужна на старте, чтоб было понятно, о чем идет речь. И ее лучше не усложнять, потому что, если с самого начала оговаривать все возможные варианты и случаи, то вместо схемы в голове у учеников будет «каша». А теперь, когда схема уже устоялась и свою роль сыграла, мы можем посмотреть на нее и критически. Наше знание шире и глубже, чем эта схема.

Я также, забегая вперед, скажу, что мы пока не изображаем на схеме таких исполнителей, как «клавиатура» и «экран» (они будут введены в § 19). В этом месте экран и клавиатура не отделяются от ЭВМ.

Цикл **для**

Как и команда **выбор**, цикл **для** является избыточной конструкцией в том смысле, что мы абсолютно любой алгоритм можем записать и без этого цикла, заменив его циклом **пока**:

```
i:=i1
нц пока i≤i2
| тело цикла
| i:=i+1
кц
```

Но это не очень удобно. Кроме того, можно забыть написать присваивание начального значения перед циклом или увеличение величины i внутри цикла. А при работе с табличными величинами циклы такого вида встречаются очень часто. Поэтому цикл **для** вводится здесь просто, как удобная и компактная форма записи.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Цикл **для** помещен именно в это место учебника по двум причинам:

(1) он очень прост и попадает в разряд «отдых», а § 13, повторю, имеет именно такой статус;

(2) содержательно цикл **для** будет использован нами при работе с табличными величинами, а это следующий § 14.

И еще одно замечание. Первоначально (в издании 1988 года) цикл **для** излагался в самом начале, еще до цикла **пока**. Но по отзывам учителей это было неудачно, так как объяснять цикл **для** без использования понятия «величины» (для i) и оператора присваивания оказалось сложным. Именно поэтому цикл **для** переехал в § 13.

Упражнения после параграфа, естественно, задействуют введенные в нем команды ввода/вывода и цикл **для**, но в остальном являются обычными задачами на составление (упр. 1, 4–6), изменение (упр. 2) и анализ (упр. 3) алгоритмов.

§ 14. Табличные величины

Этот параграф посвящен второму фундаментальному понятию информатики — «величины и, прежде всего, табличные величины». Соотношение между «табличными величинами» и просто «величинами» примерно такое же, как между «циклами» и просто «командами». Если циклы позволяют нам коротко описывать огромные последовательности действий для ЭВМ, то табличные величины дают возможность коротко описывать огромные массивы информации, которые должны быть обработаны.

Я повторю, что во всех языках программирования (честно говоря, я не знаю ни одного исключения) вместо термина «таблица» используется термин «массив» (массив данных, массив информации). Соответственно «линейные таблицы» называются «одномерными массивами», «прямоугольные таблицы» — «двумерными массивами», а также существуют не затронутые в учебнике «трех-», «четырех-», «пяти-» и пр. «мерные массивы».

Линейные таблицы

Изложение этого материала базируется на модели памяти ЭВМ. С учетом всего ранее пройденного для объяснения понятия линейной таблицы достаточно нарисовать картинку типа той, что приведена в п. 14.2 учебника, сказать, что табличная величина состоит из «элементов», ввести понятие «индекса» и форму записи элемента таблицы $k[i]$. Этого будет вполне достаточно для практического решения задач. Другими словами, материал здесь достаточно прост.

Но я хочу обратить ваше внимание на следующее. Табличная величина имеет одно общее для всех ее элементов имя, а элементы таблицы отдельных имен не имеют. Именно этим табличная величина отличается от просто набора из нескольких величин. За счет этого мы можем:

(1) компактно описать большое количество «элементарных» величин (например, запись «**цел таб** $k[1:1000]$» заставляет ЭВМ отвести память для тысячи целых чисел);

(2) используя имя таблицы k и ***вычисленное в алгоритме*** значение величины i, получить или изменить i-ый элемент таблицы, написав $k[i]$ вместо отсутствующего имени элемента.

И здесь происходит внешне не слишком заметный, но очень важный, качественный скачок. Поскольку значение

величины i может меняться при выполнении алгоритма, мы получаем возможность компактно записать обработку большого количества информации. Например, написав

```
нц для i от 1 до 1000
| k[i]:=0
кц
```

мы можем заставить ЭВМ всем этим 1000 элементам присвоить значение 0. Такой цикл заставляет ЭВМ не только выполнить массу действий (это мы уже проходили), но и изменить при этом массу информации — 1000 чисел. А ведь в приведенном фрагменте всего три строчки!

Таким образом, использование табличных величин позволяет составлять компактные алгоритмы, обрабатывающие огромное количество информации, задействовать не только быстродействие, но и объем памяти ЭВМ.

Использование Робота при изложении таблиц

Для лучшего усвоения понятия таблицы можно использовать аналогии с соответствующими структурами на поле Робота. Скажем, горизонтальный коридор на поле Робота с заданной в каждой клетке коридора радиацией — полный аналог линейной таблицы с вещественными элементами. Чтобы это подчеркнуть, а также обеспечить плавный переход от Робота к таблицам и слегка окрасить скучные математические формулировки типа «подсчитать сумму элементов таблицы», в учебнике приведен алгоритм А62, копирующий информацию о радиации в коридоре (n вещественных чисел) в таблицу «**вещ таб** $a[1:n]$». Хотя в этот момент, как правило, никакой «мотивации» уже не требуется, продолжая «игры» с Роботом, можно объяснить постановку задачи: у нас «одноразовый» Робот — уровень радиации в коридоре таков, что Робот может пройти коридор только один раз. Второго раза даже его железный организм не выдержит.

После алгоритма А62 все наши стандартные задачи для Робота в коридоре (найти максимальный уровень радиации, подсчитать число клеток с максимальным уровнем и пр.) переформулируются в задачи обработки линейных таблиц.

За счет указанной постановки получается такой плавный переход от Робота и алгоритмов его управления к таблицам. Поскольку после запоминания информации об уровнях радиации в коридоре мы начинаем работать с таблицами, появляется некоторое обоснование, осмысление, какая информация и откуда может в этих таблицах появиться. Соответственно, последующие задачи про таблицы приобретают какую-то «жизненную» окраску.

Кроме линейных таблиц, в конце параграфа вводятся прямоугольные таблицы, т. е. массивы с двумя индексами, которые изображаются на «доске» (в модели памяти ЭВМ) в виде прямоугольных таблиц (откуда и название). Насколько мне известно, этот материал никогда ни у кого никаких сложностей не вызывал.

Упражнения

Конечно, ключевой материал этого параграфа — введение самих таблиц и работа с ними. Как всегда в нашем курсе («черепаха» курса — «понимание через делание»), имеется достаточное количество задач, которые должны быть разобраны в классе, и почти бесконечное количество упражнений, которые можно задать для самостоятельного выполнения.

Именно в упражнения после этого параграфа попадает, я бы сказал, «джентльменский набор» простейших задач и упражнений по программированию: индекс максимального или минимального элемента таблицы, среднее арифметическое элементов таблицы, упорядочивание элементов таблицы по возрастанию и пр.

§ 15. Логические, символьные и литерные величины

Этот параграф завершает изложение второго фундаментального понятия информатики — понятия величины. Здесь суммируется все, что было пройдено про величины ранее, а также вводятся величины трех новых ***типов***: логические, символьные и литерные. Таким образом, понятию величины всего в нашем курсе посвящены три параграфа: 11 (введение понятия величины), 14 (табличные величины) и 15 (тип величины, логические и прочие величины).

Тип величины

Всякая величина — это четверка (имя, значение, тип, вид). С понятиями имени и значения, как правило, никаких проблем у школьников не возникает. (Для тех, кто знаком с другими языками программирования, отмечу только, что имя в алгоритмическом языке может состоять из нескольких слов, например, «число горячих клеток», и при этом их можно обычным образом писать через пробел, не используя никаких специальных символов типа подчеркивания).

Говоря, что с понятием «значение» проблем не возникает, я имею ввиду само это понятие и обычные числовые (целые и вещественные) значения. В этом параграфе будут введены значения новых типов — логические (где значением является **да** или **нет**), а также символьные (где значением является символ, например, «a», «+», «;», «1», «!» и пр.). Но связанные с этим вопросы мы отнесем к понятию ***типа*** величины.

Понятие вида величины содержательно понадобится нам только при изложении общих величин исполнителей, а до той поры может никак специальным образом не вводиться вообще (что и сделано в учебнике).

А вот ***тип*** величины — понятие и достаточно сложное, и очень важное. Поэтому я на нем остановлюсь гораздо более подробно, чем в учебнике.

Прежде всего, формальное определение. «Тип» — это характеристика величины, задающая

(а) множество значений, которые может принимать величина, и

(б) множество действий, операций, которые можно совершить с величиной (т. е. со значением) данного типа.

Например, величина целочисленного типа может принимать только целые значения. Другими словами, значением такой величины может быть только целое число. Величины такого типа (а точнее, значения такого типа, т. е. целые числа) можно складывать, вычитать, умножать, и делить — при этом будут получаться целые числа (при делении получится частное, а остаток (дробная часть) будет потерян). Целые числа также можно сравнивать: «<», «≤», «=», «≠», «>» и пр.

Впрочем, в алгоритмическом языке для величин и значений ***любого*** простого типа допустимы операции сравнения «=» и «≠», а также команда присваивания «величина := значение».

Поэтому при описании нового типа величин необходимо указать,

(а) какие значения может принимать величина этого типа;

(б) какие действия и операции (кроме «=», «≠», «:=») над ней допустимы.

Например, когда мы вводим понятие величины логического типа, мы должны сказать:

(а) что величина этого типа может принимать только одно из двух значений: либо **да**, либо **нет** (полезно напомнить ученикам о битах и отметить, что для хранения значения логической величины достаточно одного бита);

(б) что над величинами и значениями данного типа допустимы все логические (откуда и название типа) операции, как-то: **и**, **или**, **не**.

Кстати, для таблицы значением является набор чисел, а количество элементов таблицы и их индексы ***входят*** в ***тип*** таблицы. Другими словами «**цел таб** a[1 : 10]» и «**цел таб** b[0 : 9]» — это величины двух ***разных***, хотя и «родственных» типов. Для первой величины допустима операция получения элемента a[10], которая недопустима для второй величины — для b.

Итак, тип указывает какие значения может принимать величина и что с этими значениями можно делать. Те типы, которые мы уже изучали (**цел**, **вещ**, **цел таб** и пр.), как и те, что изложены в этом параграфе (**лог**, **сим**, **лит**), являются так называемыми встроенными, предопределенными типами алгоритмического языка. Других (не предопределенных) типов в алгоритмическом языке нет. Доопределить, построить «свой» тип нельзя.

Я уже говорил и еще скажу в заключение, что в этом месте алгоритмический язык отличается от всех современных языков программирования и от языка Паскаль, в которых есть средства конструирования новых типов данных. Здесь я лишь хочу бегло обратить на этот факт ваше внимание, а подробно мы остановимся на нем в заключении, когда будем обсуждать место курса в «большой» информатике.

Логические величины

Для объяснения, что такое величина логического типа, необходимо указать, какие значения может принимать такая величина, а также где и как ее можно использовать. Я это практически уже проделал.

Добавлю только для полноты картины, что значением логической величины может быть либо **да**, либо **нет**, либо ***значение величины может быть неопределено***. Последнее, впрочем, относится к величинам любых типов.

Кроме того, величины логического типа (логические значения) можно использовать в условиях в алгоритмическом языке. Простые примеры имеются в учебнике.

Я же хочу обратить ваше внимание на задачу и алгоритм в п. 15.5. Задача такая. От горизонтального коридора на поле Робота кое-где вверх отходят тупики разной высоты. Надо закрасить клетки коридора напротив тех тупиков, высота которых больше трех клеток (мы их назвали «длинными тупиками»).

Если мы начнем решать эту задачу, т. е. составлять алгоритм, у нас, конечно, будет цикл, потому что неизвестно, сколько клеток в коридоре. Условие окончания цикла (вспомните п. 9.11) — условие выхода из коридора, т. е. «снизу свободно». Внутри цикла надо идти по коридору вправо и, если сверху обнаружится «длинный» тупик, то соответствующую клетку коридора закрасить.

И вот тут очень важное место. Я бы сказал — интерференция, взаимное усиление всех пройденных нами понятий. Теперь, после введения величин логического типа, мы, в соответствии с методом последовательного уточнения, эту изложенную выше ***идею*** решения можем записать прямо на алгоритмическом языке:


```
если сверху длинный тупик
| то закрасить
все
```

Здесь условие «сверху длинный тупик» — вызов вспомогательного алгоритма-функции, значением которого является ***логическая*** величина.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Собственно говоря, многие конструкции в алгоритмическом языке вводятся именно для того, чтобы, произнеся нормальную человеческую фразу типа «если длинный тупик, то закрасить клетку коридора», мы могли бы ее примерно в таком же нормальном виде записать в алгоритм. Но уже строго и недвусмысленно — в виде фрагмента алгоритма на алгоритмическом языке.

Конечно, после того как мы написали этот фрагмент, мы, в соответствии с методом последовательного уточнения, должны еще написать вспомогательный алгоритм-функцию, который будет анализировать, есть ли сверху тупик и является ли он «длинным». Это — алгоритм А73 на с. 119 учебника.

Теперь я готов сказать, что уровень «письма и счета» достигнут. Если школьники в состоянии так записать алгоритм, понимая, что это — не неформальная запись, а абсолютно строгая, представляя себе, как ЭВМ будет выполнять такой алгоритм, что будет происходить в памяти ЭВМ и т. д., то это и значит, что базовые понятия и навыки алгоритмизации, а с ними и алгоритмический язык, как инструмент записи алгоритмов, освоены. С этого момента можно переходить к изучению применений ЭВМ, а также к использованию полученных знаний для понимания устройства окружающего нас мира.

Символьные и литерные величины

Раздел про символьные величины необычайно прост. Достаточно сказать, что значением символьной величины может быть любой символ, а в качестве операций можно использовать операции сравнения.

А вот литерная величина — это совершенно новое понятие. Литерную величину удобно представлять себе в виде линейной таблицы, элементами которой являются символь-

ные величины. Но, в отличие от числовых линейных таблиц, ***длина*** литерной строки (т. е. количество «элементов таблицы») может меняться в ходе выполнения алгоритма. Кроме того, для литерных величин определены новые операции: «вырезка» куска строки, добавление одной строки в конец другой и пр.

Этот материал уже очень «технический» или «программистский» (в обычном, приниженном смысле этого слова). Вы можете весь материал про литерные строки опустить без особого ущерба для алгоритмической культуры учащихся (с соответствующими изъятиями из дальнейшего материала учебника). Если же вы его проходите, то я рекомендую поработать в гипертексте «литерные величины» в системе «КуМир», ответить на контрольные вопросы и т. д.

Мы поместили этот материал по двум причинам.

Во-первых, его можно рассматривать как новый материал, на котором еще раз проверяются, отрабатываются и углубляются все ранее полученные школьниками знания и навыки. К тому же опыт показывает, что формулировки задач со строками и составление соответствующих алгоритмов вызывают интерес. Я чуть ниже на этом остановлюсь.

Во-вторых, мы немного задействуем этот материал в дальнейшем, при изложении понятий компиляции и интерпретации, построении информационных моделей алгоритмов и при задании названий городов в информационной модели транспортной сети.

Если величины литерного типа изучаются, то здесь можно сформулировать достаточно много разных задач, в которых аргументами и/или результатами являются строки. Эта область восходит к детским играм с шифровкой и дешифровкой, она не имеет отношения ни к Роботу, ни к Чертежнику, ни даже — в каком-то смысле — к вычислениям. Это совсем другая область, и здесь вполне могут проявиться те ученики, кому работа со словами нравится больше, чем, скажем, геометрия.

Хочу напомнить вам следующий методический прием: здесь очень хорошо использовать для усиления мотивации и лучшего понимания задач игру в «черные ящики» (см. с. 61 пособия). Этот методический прием состоит в том, что прежде чем, как сформулировать ученикам задачу на составление алгоритма, учитель «задумывает» соответствующий алгоритм как алгоритм работы «черного ящика» и начинает играть с учениками в этот «черный ящик», выписывая на доске аргументы и результаты работы «ящика». После того,

как школьники отгадают правило работы «ящика» (напомню, что такое отгадывание весьма увлекательно), учитель говорит: «Молодцы, алгоритм работы «черного ящика» вы угадали. А теперь ***запишите*** этот алгоритм на алгоритмическом языке».

Еще одно методическое замечание, которое я хочу тут сделать, состоит в том, что если вы будете преподавать наш курс в более младших классах, или если у вас просто будет достаточно времени, то еще на ранних стадиях курса можно устроить «пропедевтику» работы с символьными и литерными величинами с помощью соответствующих исполнителей. В учебнике таких исполнителей нет, но можно либо использовать таких исполнителей, как «Редактор слова» из нашего вузовского учебника [ПдМ], «Муравей» Г.А. Звенигородского [Звенигородский], либо просто расширенного Робота, который дополнительно умеет читать и записывать символы (буквы, цифры и пр.) в клетках поля. Тогда значительную часть задач и упражнений этого параграфа можно будет переформулировать в терминах соответствующего исполнителя (подобно тому, как задачи обработки табличной информации мы формулировали в терминах обработки радиации в коридоре на поле Робота). Опыт показывает, что задачи такого словесного характера пользуются большой популярностью. Бывают просто жемчужины, например, когда в ходе решения задачи «получить путем перестановки букв из данного слова максимальное количество других осмысленных слов» школьники придумывают, как из слова «вертикаль» сделать «кильватер».

Резюме главы 1 «Алгоритмический язык»

С содержательной, логической точки зрения мы на этом главу 1 «Алгоритмический язык» закончили. Конечно, остался еще большой и самый сложный в курсе § 16, но он в каком-то смысле лежит за пределами первой главы. В предварительных вариантах учебника это была отдельная глава, которая называлась «Методы алгоритмизации». Затем мы эту главу упростили, сжали до одного параграфа и в итоге поместили в конец главы 1. Но § 16 — методам алгоритмизации — я посвящу две следующих лекции.

А тему «Алгоритмический язык» мы почти прошли. Повторю, что цель этой главы — научить школьников владеть алгоритмическим языком не хуже, чем они владеют пись-

мом и счетом. Владение алгоритмическим языком должно стать базовым навыком, инструментом для решения задач. У нас еще будет некоторое дополнение к языку, когда мы будем проходить четвертое фундаментальное понятие информатики в § 21–22 — понятие информационной модели исполнителя. Но в целом алгоритмический язык на этом можно считать изученным.

Лекции 7–8

Итак, нами пройдены понятия «величины» и «табличной величины». И если говорить о современной «большой информатике», то единственное, чего мы не рассмотрели (да и вообще в учебнике никак не затронули) при изучении величин — это область, именуемая конструированием типов и структур данных (т. е. конструирование типов величин). Я подробнее остановлюсь на этом вопросе, когда буду говорить о месте нашего курса в большой информатике (с. 324).

Ну и, конечно, нами пока не изучено четвертое фундаментальное понятие информатики — информационные модели исполнителей. Информационными моделями исполнителей мы займемся позже (§ 21–22), а сейчас несколько отвлечемся от изучения основных понятий информатики. Следующий параграф учебника (§ 16) приоткрывает дверь в совсем иную область информатики — в область методов алгоритмизации.

Честно говоря, первые 15 параграфов нашего учебника содержат материал, который любой человек, знакомый с каким-нибудь языком программирования, любой человек, знакомый с информатикой, может просто пролистать. Весь материал до § 16 будет простым и знакомым — с точностью до обозначений алгоритмического языка. Даже если вы сейчас слушаете мои лекции и видите учебник в первый раз, то после лекций вы первые 15 параграфов спокойно и бегло пролистаете и убедитесь, что там все очень просто и понятно. Конечно, какие-то задачи могут быть достаточно сложны, но сам теоретический материал чрезвычайно прост, общепризнан и более или менее знаком всем.

§ 16 — первый параграф, который не пролистывается. Более того, многие специалисты в области информатики с методами алгоритмизации вообще не знакомы (что, конечно, объясняется относительной «юностью» всей этой области).

§ 16. Методы алгоритмизации

Этот параграф учебника является, по-видимому, самым сложным и, соответственно, одним из самых интересных. Если по учебнику преподавать в 7–8 классах, то его, на мой взгляд, следует пропустить. Также его можно пропустить, если у вас слабый класс.

Параграф этот — повышенной сложности, а кроме того в нем существенно задействован математический (логический) стиль мышления, т. е. от учащихся требуется умение рассуждать и мыслить логически, необходимо наличие у них минимальной математической культуры.

Но в то же время, всякая наука становится наукой, когда в ней появляются ее собственные методы решения задач. Если опять использовать аналогию с математикой, то существует огромная разница между

— умением угадывать корни квадратных уравнений или находить их подбором и
— знанием общей формулы для нахождения корней квадратного уравнения и умением ее применять.

И хотя угадывание или подбор корней квадратных уравнений, несомненно, — деятельность, как-то связанная с математикой, назвать такое «угадывание» наукой (математикой), конечно, язык не повернется.

Материал § 16 в нашем учебнике как раз и демонстрирует, что информатика достигла определенной зрелости. Обращаю ваше внимание, что до сих пор мы именно «угадывали» алгоритмы. Конечно, мы изучили много новых понятий — алгоритмы, управляющие конструкции, величины и пр. (подобно тому, как для угадывания корней квадратного уравнения надо предварительно изучить, что такое «квадратное уравнение» и что такое «корень»). Но алгоритмы мы придумывали «по наитию». Мы их «подбирали» или «угадывали». Конечно, мы при этом о чем-то думали, как-то рассуждали, и хотя бы интуитивно использовали какие-то общие соображения типа «без цикла тут не обойтись». Но ***мы не использовали явно никаких методов создания алгоритмов*** — мы их «придумывали», а не «выводили» по общим правилам и формулам.

Лишь мельком я говорил про какие-то методы последовательного уточнения (с. 92). Но в целом все задачи решались, образно говоря, «озарением» — в каждой задаче решение просто придумывалось, а объяснение решения часто

сводилось к слову «Смотри!» (легенда гласит, что именно так древние греки объясняли решения геометрических задач).

Конечно, и задачи у нас были не самые сложные, поэтому, зная формулировку задачи и конструкции языка (циклы, **если** и пр.), мы эти простые задачи как-то решали. Решение часто было «видно», и — честно говоря — мы не столько учились решать задачи, сколько записывать решения, которые «видно» и так. (В науке предпочитают говорить «решение очевидно»). Повторю, что при этом никакого аналога «формулы для корней квадратного уравнения» у нас не было. Мы придумывали алгоритмы, а не «выводили» их по каким-то формулам.

В § 16 впервые излагаются простейшие методы решения алгоритмических задач.

Освоение любого метода состоит овладении двумя разными «навыками и умениями». Ученик должен

1) узнавать задачи из соответствующего класса задач, т. е. увидев задачу, он должен понять, что эта задача попадает в соответствующий класс, подходит для решения данным методом. Аналогично тому, как в математике, увидев уравнение, он должен понять квадратное это уравнение или какое-то другое, т. е. уметь узнавать квадратные уравнения среди множества других. Итак, первое, — уметь узнавать задачи из соответствующего класса.

2) уметь быстро и без ошибок применить метод к решению данной конкретной задачи (для формулы корней квадратного уравнения — уметь определить чему равны в конкретном уравнении a, b и c, подставить их значения в формулу, провести вычисления и проверить правильность решения подстановкой в уравнение или по формуле Виета).

В каком-то смысле п. «1)» — узнавание задач, решаемых данным методом — это самая сложная часть в овладении методом. И лично я уверен, что нет никакого иного пути научиться узнавать задачи, определять, какой метод можно применить, кроме как «решать, решать и еще раз решать» задачи на соответствующий метод, смотреть, как решает такие задачи учитель как их решают другие, решать самому (на первых порах с помощью учителя) и т. д.

В § 16 последовательно изложено несколько методов алгоритмизации. Сами методы более или менее независимы, и мы их будем проходить также последовательно — метод за методом.

16.1. Метод 1 — «рекуррентные соотношения»

В учебнике этот метод излагается в несколько этапов. Я остановлюсь на всех этапах и притом достаточно подробно, поскольку, как я уже говорил, этот параграф — самый сложный в курсе.

Что такое метод рекуррентных соотношений

Сначала вступление. Очень часто встречаются задачи, в которых значения каких-то величин надо вычислять шаг за шагом по каким-то формулам, используя ранее вычисленные значения. Вспомните, например, последовательность Фибоначчи из самого первого параграфа (1, 1, 2, 3, 5, 8, 13, ... — каждый следующий элемент получается как сумма двух предыдущих). Для этой последовательности n-ый элемент получается как сумма $n-1$ и $n-2$-го:

$$a_n = a_{n-1} + a_{n-2}$$

а первые два элемента (чтобы было с чего стартовать) задаются явно:

$$a_1 = 1, \quad a_2 = 1.$$

Приведенные выше формулы, задающие последовательность, называются рекуррентным соотношением. Слово «рекуррентный» означает, что очередной элемент последовательности выражается через предыдущий или несколько предыдущих элементов.

Другой пример рекуррентной последовательности, из рассматривавшихся в § 1, — последовательность степеней двойки:

$$a_n = 2 \times a_{n-1}, \quad a_1 = 2.$$

Обычно задача состоит в нахождении какого-то конкретного элемента такой последовательности, например, «найти 15-е число Фибоначчи» или «найти 20-ю степень двойки».

Итак, общая постановка задачи выглядит следующим образом: задана последовательность, в которой очередной элемент выражается через один или несколько предыдущих, и надо вычислить какой-то конкретный элемент этой последовательности. В таком случае для составления алгоритма

можно воспользоваться одним из самых простейших методов алгоритмизации, который называется так же, как и соотношения, задающие последовательность, — метод рекуррентных соотношений.

Практически мы несколько раз пройдем по этому методу, постепенно углубляя наши знания.

Простейший пример рекуррентного соотношения

Рассмотрим какую-нибудь рекуррентную последовательность, например, из учебника

$$a_i = 2 \times a_{i-1} - 1, \quad a_1 = 2.$$

Если начать ее вычислять, то мы получим последовательность 2, 3, 5, 9, ... Если нам надо написать алгоритм для вычисления n-го члена этой последовательности, то мы легко можем написать алгоритм А78, приведенный в учебнике на с. 126:

```
алг цел элемент (арг цел n)
  дано n>0
  надо | знач=n-й элемент последовательности
нач цел i, цел таб a[1:n]
| a[1]:=2
| нц для i от 2 до n
| | a[i]:=2*a[i-1]-1
| кц
| знач:=a[n]
кон
```

Здесь мы еще до самого метода не дошли. Пока у нас просто имеется какая-то последовательность, заданная рекуррентным соотношением, и мы — один к одному — переписали ее в виде алгоритма. Для запоминания элементов последовательности ввели табличную величину, в первом элементе которой запомнили значение первого элемента последовательности, во втором — второго и т. д.

Это такой нулевой уровень — если есть рекуррентное соотношение, то можно записать алгоритм, который, буква в букву, соответствует формуле. Поэтому здесь пока никаких сложностей (а значит и никаких методов и даже потребности в них) не возникает.

Основная идея метода рекуррентных соотношений — «исчезновение» индексов

А вот дальше начинается собственно содержание. Прежде всего заметим, что если мы начнем вычислять n-ый элемент такой последовательности сами — «вручную», то не станем запоминать ***все*** ранее вычисленные ее элементы. В этом нет никакой необходимости — ведь каждый следующий элемент вычисляется через предыдущий. При вычислении $a_5 = 2 \times a_4 - 1$, мы подставим $a_4 = 9$ и по формуле получим, что следующий элемент равен $2 * 9 - 1 = 17$. Чтобы получить следующий элемент (a_6), в формулу надо подставлять полученное значение — 17 (a_5). Про предыдущие значения (a_1, a_2, a_3, a_4) уже можно забыть — при дальнейших вычислениях они нам больше не понадобятся.

При вычислении нового элемента используется только один предыдущий элемент и, следовательно, хранить все предыдущие элементы последовательности в памяти ЭВМ совершенно незачем — для вычислений они не нужны. Чтобы подсчитать следующий элемент нужно знать только одно число — значение предыдущего элемента последовательности.

На этой основе алгоритм можно попытаться переписать без использования таблиц — ведь хранить надо только одно число. Этот алгоритм приведен в учебнике следующим (А79, стр. 126).

```
алг цел элемент (арг цел n)
  дано n>0
  надо | знач=n-й элемент последовательности
нач цел i, a
| a:=2          | запоминание 1-го члена посл-ти
| нц для i от 2 до n
| | a:=2*a-1 | вычисление i-го члена посл-ти
| кц
| знач:=a
кон
```

Фактически, в этом новом алгоритме всюду, где ранее использовалась таблица, написана одна и та же обычная величина a. Если сравнить эти два алгоритма — А78 и А79, то видно, что $a[i]$ и $a[i-1]$ заменены на одну и ту же букву a без каких-либо индексов.

Это уже некоторое содержание изучаемого метода. Математические индексы, использовавшиеся при математическом описании (задании) последовательности и ***служащие в математике для обозначения разных элементов последовательности***, совершенно необязательно отображать в индексы таблиц. В алгоритме А78 математические обозначения a_i и a_{i-1} (т. е. разные элементы последовательности) соответствовали $a[i]$ и $a[i-1]$ — т. е. разным элементам таблицы. А в алгоритме А79 они соответствуют ***одной и той же величине*** a, при этом математические обозначения a_i и a_{i-1} (разные элементы последовательности) соответствуют не самой этой величине, ***а ее значениям в разные моменты времени***, в разные моменты вычислений.

Ведь в отличие от статического (неизменного) математического задания последовательности рекуррентным соотношением, в ходе выполнения алгоритма значения величин ***меняются***. В начале исполнения алгоритма А79, после выполнения команды $a := 2$ значением величины a станет первый элемент последовательности — 2. Но уже после первого выполнения в теле цикла команды $a := 2 * a - 1$ старое значение a послужит для вычисления следующего элемента ($2*2-1 = 3$), и величине a будет присвоено новое значение 3. С течением времени значение величины a меняется.

Итак, в алгоритме А79 используется ***одна*** величина a, меняется только ее ***значение*** по мере выполнения алгоритма. Математические обозначения a_i и a_{i-1} относятся при этом к разным ***значениям*** одной и той же «алгоритмической» величины a в разные моменты времени, т. е. отражают, как ***меняется*** величина a.

Сравнение двух подходов к вычислению рекуррентной последовательности

Это чрезвычайно важное место, можно сказать, сердце метода рекуррентных соотношений (хотя до самого метода мы еще не дошли). Давайте еще раз сравним алгоритмы А78 и А79. В алгоритме А78 запоминаются все элементы последовательности — каждый в отдельном элементе таблицы. Это просто, алгоритм записывается очевидным образом, но требует много памяти, расточительно и не всегда возможно. В алгоритме А79 элементы последовательности a_i и a_{i-1} обозначают разные значения величины a. Величина a меняется в ходе вычислений — это и есть переход от одного i к

другому. Я называю этот переход от математических индексов в описании последовательности к обычным величинам в алгоритмах и их значениям в разные моменты времени ***«исчезновением индексов»***.

Самым важным является тот факт, что, когда мы вычисляем по рекуррентным соотношениям, таблицы и индексы, вообще говоря, не нужны. Для больших n это может оказаться очень существенным. Чтобы вычислить $a_{1000000}$ (миллионный элемент последовательности), в алгоритме А78 придется завести таблицу для хранения миллиона чисел. В алгоритме же А79 будет храниться только одно число, а вычислять можно и миллионный, и миллиардный, и любой другой элемент последовательности. Таким образом, с чисто прагматической точки зрения — это еще и экономия памяти ЭВМ.

> **ПРОГРАММНОЕ ОТСТУПЛЕНИЕ**. В большинстве языков программирования, будь то Паскаль, Бейсик или Школьный алгоритмический язык, величина целого типа может принимать не слишком большие значения, не больше нескольких десятков цифр. Для вычисления по алгоритму А78 или А79 миллионного члена последовательности необходимо, чтобы целое число могло содержать до миллиона двоичных цифр. Если на запоминание каждой двоичной цифры тратить один бит, то для выполнения алгоритма А79 хватит памяти размером около 1Мбит. А вот для выполнения алгоритма А78 понадобится (по минимуму, если усложнить алгоритм и на каждый элемент последовательности отводить ровно столько бит, сколько надо)
>
> 1 бит на первый элемент последовательности;
> 2 бита на второй элемент, . . .
> . . . . . .
> 999999 бит на предпоследний элемент и
> 1000000 бит на последний элемент.
>
> Итого 1 + 2 + 3 + . . . + 999999 + 1000000 бит, или около 500 Гбит, т. е. в 500000 раз больше памяти.

С содержательной же точки зрения этот эффект «исчезновения индексов» означает, что мы начинаем лучше учитывать особенности алгоритмической (динамической) составляющей. Говоря очень грубо, разница между А78 и А79 — это разница между математическим и алгоритмическим стилем мышления. Последовательность ***значений*** величины a в алгоритме А79 — и есть последовательность a_i в математическом описании последовательности. Вот самое первое и самое важное, в чем вы должны разобраться.

Больше, собственно, мне на эту тему сказать нечего. Вы должны просто потренироваться, пососставлять такие алгоритмы, «набить руку» и привыкнуть. Дело в том, что наша математическая культура и предыдущий школьный опыт «сбивают» человека — как правило, если специально не поработать, человек интуитивно пишет элемент последовательности как $a[i]$, как элемент таблицы — и получается алгоритм А78. Нужно некоторое время пососставлять алгоритмы «без индексов» (как А79), чтобы от статическо-математического подхода сместиться в сторону алгоритмического, чтобы привыкнуть к тому, что математические a_i «скрываются» здесь за одной и той же величиной a — это ее начальное значение, следующее, следующее и т.д.

Алгоритм А78 можно понимать статически — это как бы то же самое математическое задание последовательности, только в алгоритмических обозначениях. Для понимания алгоритма А79 уже нужно знать ***и представлять*** себе — как он будет выполняться и что при этом будет происходить. Алгоритмическая составляющая мышления в алгоритме А79 задействована существенно глубже. Именно в этом и состоит некоторая сложность в составлении и понимании таких алгоритмов. С другой стороны, именно поэтому такие алгоритмы обычно намного эффективнее. Овладение методом рекуррентных соотношений позволяет преодолеть эти сложности и легко писать алгоритмы без «лишних» индексов.

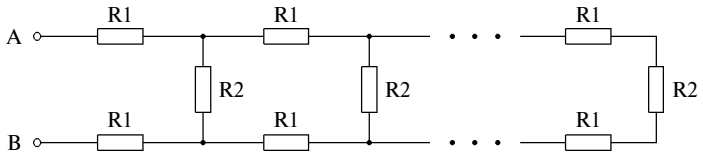
Итак, первый важный вывод состоит в том, что если надо вычислить значение элемента последовательности, заданной рекуррентными соотношениями, то соответствующий алгоритм можно записать без использования таблиц, «без индексов».

Как применять метод на практике

Ну а теперь, после этого продолжительного вступления, мы перейдем уже к самому методу рекуррентных соотношений.

Метод рекуррентных соотношений начинается отнюдь не с рекуррентных соотношений. Вообще в жизни очень мало задач, которые прямо так и формулируются: «вычислить n-ый элемент последовательности, заданной та-

ким-то рекуррентным соотношением». Подобные формулировки характерны для тренировочных задач в школе. Жизненные задачи формулируются совсем по иному. В качестве такой «жизненной» задачи в учебнике рассмотрена задача по физике (требующая, соответственно, элементарного знания физики) — см. с. 127 учебника.



Имеется гирлянда из проводов с лампочками (рис. 65 учебника) из достаточно большого числа (n) параллельно включенных лампочек, сопротивлением $R2$ каждая, и при этом сопротивлением соединительных проводов ($R1$) нельзя пренебречь. Требуется посчитать общее сопротивление гирлянды между клеммами A и B. Такая вот задача.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я уже говорил, что мы старались все задачи в учебнике подобрать таким образом, чтобы продемонстрировать полезность информатики, показать, что можно решать задачи, которые в рамках обычной математики или физики не решаются.

На наш взгляд попытка, скажем, представить алгоритм вычисления корней квадратного уравнения как нечто содержательное с точки зрения информатики — это просто дискредитация информатики. Трудозатраты на написание, ввод и выполнение такого алгоритма гораздо выше, чем на непосредственное нахождение корней по формуле. Смысл этой деятельности остается непонятным школьнику независимо от того, какие аргументы приводит учитель. Конечно, переписывание формулы в виде алгоритма требует отчетливого понимания того, что корней может и не быть, а также знания алгоритмических обозначений (соответствующего языка программирования), но практически не дает никакого вклада в развитие алгоритмического мышления (а для нас, повторю, развитие алгоритмической составляющей мышления — главная цель курса). Поэтому, на наш взгляд, алгоритмы типа «нахождения корней квадратного уравнения» могут быть составлены только «мимоходом», обсуждаться, как фрагменты решения какой-то более интересной и понятной задачи.

Базовые задачи информатики должны быть задачами с явно выраженной алгоритмической составляющей, задачами, которые школьники без компьютера, без информатики решить не могут.

Задача с гирляндой лампочек — одна из них (во всяком случае, ***формула*** сопротивления в такой цепи школьникам неизвестна, и, думаю, они не в состоянии ее вывести). Написать для этой задачи ответ так, как это принято в физике или в математике (в виде какой-то формулы), школьники не могут, т. е. задача не решается обычными физическими и математическими методами. И именно поэтому она хороша для нас, поскольку демонстрирует собственную область применения информатики, ее несводимость и невторичность по отношению к математике и физике.

И ЕЩЕ ОДНО ОТСТУПЛЕНИЕ. Единственная разумная формула, которую могут придумать школьники, это формула для сопротивления ***бесконечной*** гирлянды. Сопротивление x такой ***бесконечной*** гирлянды может быть найдено, как корень квадратного уравнения, которое получится, если в рекуррентном соотношении вместо a_i и a_{i-1} подставить x.

Итак, наша задача — найти сопротивление между клеммами A и B. Помните, я говорил, что первое, что необходимо уметь — это видеть. В данном случае увидеть, что это — задача на рекуррентные соотношения, хотя в ее постановке нет ни слова ни о каких рекурентных соотношениях. Такое «видение» дает только практика, только опыт. Но обычно, если в постановке задачи есть многоточие, n каких-то элементов и т. п., то это вполне может оказаться задача на рекуррентные соотношения.

Что значит «задача на рекуррентные соотношения»? Это значит, что мы можем выразить сопротивление гирлянды из n лампочек через сопротивление такой же гирлянды из $n-1$ лампочки. Т. е., получив задачу, школьник должен это просто «увидеть» — увидеть, что можно рассмотреть последовательность таких гирлянд, начиная с $n=1$ и постепенно увеличивая n. И что сопротивление для следующей гирлянды можно выразить через сопротивление для предыдущей. Если он это заметил, то цель достигнута — он распознал

эту задачу как задачу на рекуррентные соотношения, дальше можно начать писать формулы.

Действительно, для $n = 1$ получится гирлянда, изображенная на рис. 66а) учебника. Сопротивление для такой простой гирлянды (электрической цепи) школьники легко могут найти, используя формулы из курса физики. При $n = 1$ общее сопротивление гирлянды будет равно $a_1 = 2 \times R1 + R2$.

Теперь давайте попробуем выразить сопротивление для гирлянды из n лампочек (обозначим его a_n) через сопротивление a_{n-1} гирлянды из $n-1$ лампочки. Для этого гирлянду из n лампочек надо нарисовать, как гирлянду из $n-1$ лампочки, к которой добавлена еще одна лампочка (рис. 66б) учебника). Получается опять довольно простая цепь, сопротивление которой может быть выражено по формулам физики:

$$a_n = 2 \cdot R1 + \frac{1}{\frac{1}{R2} + \frac{1}{a_{n-1}}} = 2 \cdot R1 + \frac{a_{n-1} \cdot R2}{a_{n-1} + R2}.$$

Таким образом мы получили формулу, выражающую a_n через a_{n-1}. Вместе с формулой $a_1 = 2 \times R1 + R2$ это уже полное рекуррентное соотношение, задающее последовательность.

Я еще раз обращаю ваше внимание, что в постановке задачи никакого рекуррентного соотношения не было. Надо было просто вычислить сопротивление некоторой цепи. Когда я говорю, что овладение методом возможно только через работу («понимание через делание»), то это означает, что в процессе решения такого рода задач школьники должны научиться распознавать рекуррентные соотношения там, где их в явном виде «нет», и научиться переформулировать задачу в терминах нахождения конкретного элемента какой-то последовательности, видеть эту последовательность и уметь выразить следующий его элемент через предыдущий или предыдущие. Все точно так же, как с квадратным уравнением: надо его «видеть» даже если оно внешне выглядит подругому, например, $c \times (x-d)^6 + a \times (x-d)^3 + b = 0$.

Самая большая сложность — это научить школьников в подобного рода задачах, где никаких рекуррентных соотношений нет, эти соотношения «видеть», распознавать. Т. е. использовать метод рекуррентных соотношений как средство решения задач, когда в постановке задачи не сказано, что это задача именно на этот метод.

После того как рекуррентное соотношение составлено (т. е. математические формулы получены), мы можем — уже

не задумываясь — записать алгоритм (см. алгоритм А80 учебника).

Посмотрите на этот алгоритм — он чрезвычайно прост! Один цикл, внутри которого всего один оператор присваивания. И это для подсчета сопротивления такой схемы! А ведь на старте задача казалась весьма сложной.

Дело в том, что все сложности этой задачи в результате оказались «скрытыми» в методе ее решения. Это — общее свойство: применяя какой-то метод, мы всегда используем ранее накопленное знание. В этот момент мы не разбираемся в деталях метода, не анализируем «почему это так» — в освоенном методе мы «знаем, что это так» и используем все вместе, как единый и простой элемент знания. Вывод формулы для корней квадратного уравнения — весьма содержательная задача. Но если мы формулу уже вывели и запомнили, то при решении конкретных задач мы просто ее используем, не задумываясь, откуда она взялась и почему верна. И решение конкретных квадратных уравнений становится делом достаточно простым.

Так же и у нас: получив рекуррентные соотношения для гирлянды из лампочек, мы, используя старое знание (например, про «исчезновение индексов»), можем, «не задумываясь», просто записать алгоритм. И алгоритмы, как правило, получаются очень простые, как А80. Они намного проще, чем те, с которыми мы уже возились (цикл, внутри него еще цикл, внутри **если** или еще что-нибудь). Но, поскольку эти алгоритмы сразу «не видны», не являются очевидными и сами по себе в голову не приходят, их надо выводить, используя какие-то методы алгоритмизации. Поэтому научиться составлять такие алгоритмы сложнее, хотя, повторю, сами алгоритмы могут оказаться много проще уже нами изученных. Ведь нам теперь надо овладеть не просто алгоритмическими обозначениями, но и ***методами алгоритмизации***, а это, конечно, сложнее.

Таким образом, метод рекуррентных соотношений состоит в том, что в задаче надо (1) увидеть какую-то последовательность величин (в задаче выше — увидеть последовательность гирлянд для растущего n), (2) выразить искомые величины рекуррентными соотношениями и (3) записать соответствующий алгоритм.

Рекуррентные соотношения с несколькими «предыдущими» элементами

Заканчивается раздел про метод рекуррентных соотношений еще одним «углублением». Дело в том, что до сих пор рассматривались только соотношения, в которых следующий элемент выражался через ***один*** предыдущий элемент. Существуют ситуации, когда очередной элемент выражается не через один предыдущий, а через два, три или более. Например, в последовательности Фибоначчи очередной элемент выражается через два предыдущих. Тогда описанное выше простое «выбрасывание индексов» не срабатывает — ведь нам надо хранить не одно, а несколько предыдущих значений (предыдущих элементов последовательности), поэтому и величин в алгоритме понадобится несколько. Если в рекуррентной формуле справа фигурируют разные a_{i-1}, a_{i-2}, а мы просто выкинем индексы, то получится ерунда — одна буква будет ***в одно и то же время*** обозначать разные элементы последовательности. Поэтому следующая часть параграфа посвящена вопросу, что делать, если очередной элемент последовательности выражается через ***несколько*** предыдущих.

Рассмотрим это на примере последовательности Фибоначчи, когда для вычисления очередного элемента последовательности нужно знать два предыдущих. Естественно, мы легко можем записать алгоритм с использованием таблицы для запоминания элементов последовательности (как мы это делали в алгоритме A78). Но для записи алгоритма без индексов нам понадобятся уже две величины — для хранения двух предыдущих значений. Назовем их, например, a и b. Пусть в какой-то момент времени значением величины a является a_{i-1}, а значением b — a_{i-2}. Тогда, если я от этой пары хочу перейти к паре a_i, a_{i-1}, т. е. в математической схеме увеличить i на единицу, то мне нужно сделать следующее. Поскольку величина b должна стать равной a_{i-1}, (это старое значение a), следует выполнить команду присваивания $b := a$. Новое значение a должно стать равным $a_i = a_{i-1} + a_{i-2}$, т. е. новое значение a можно получить, как сумму старых значений a и b, т. е. командой присваивания $a := a + b$.

Необходимость запоминания некоторых «старых» значений

Беда в том, что эти два действия в моих рассуждениях — $b := a$ и $a := a+b$ — должны выполниться как бы параллельно. Если мы сначала выполним первое действие, то «потеряем» старое значение b и не сможем его использовать во второй формуле. Поэтому нам придется совершить некоторые дополнительные действия, например, запомнить старое значение b в какой-нибудь промежуточной величине (c) и после изменения b в формуле для нового значения a использовать это запомненное значение (c):

$$c := b; \quad b := a; \quad a := a + c.$$

Это один из возможных вариантов, при котором значением c является a_{i-2}, а сама величина c нужна только на время перехода от $a = a_{i-1}$, $b = a_{i-2}$ к $a = a_i$, $b = a_{i-1}$.

Вы видите, что если очередной элемент последовательности выражается через несколько предыдущих, алгоритм становится сложнее: надо запоминать эти предыдущие значения и следить за порядком действий, чтобы эти «предыдущие значения» случайно не испортить. Сложность составления алгоритма довольно резко вырастает, хотя содержание нашего «исчезновения индексов» почти не меняется. Содержание остается тем же — увидев, что очередной элемент выражается через предыдущие, мы должны понять, сколько предыдущих элементов нам необходимо, и ввести соответствующее количество величин для хранения их значений. Этих величин будет ровно столько, сколько чисел нам надо было бы помнить, если бы мы считали вручную.

Дополнительная сложность возникает позже, при записи в величинах формул перехода к следующему элементу последовательности, поскольку в этот момент следует позаботиться, чтобы последовательность присваиваний (которые будут выполняться одно за другим) не испортила нужных нам значений. Возможно, при этом придется ввести какие-то промежуточные величины, запомнить в них старые значения, а лишь потом присваивать новые.

Запоминание всех «старых» значений

Впрочем, есть простой путь, который позволяет и в этом случае писать алгоритм, «не задумываясь» о порядке при-

сваиваний и прочем. Надо ввести «второй комплект», продублировать величины. Если мы использовали a и b, то надо ввести «a старое» (as) и «b старое» (bs). После чего переход к следующему элементу последовательности записывается в виде:

$$\begin{array}{lll} as & := & a; \quad bs := b \\ a & := & \ldots \text{ (формула от } as \text{ и } bs) \ldots \\ b & := & \ldots \text{ (формула от } as \text{ и } bs) \ldots \end{array}$$

Для последовательности Фибоначчи можно написать:

$$\begin{array}{lll} as & := & a; \quad bs := b \\ a & := & as + bs \\ b & := & as \end{array}$$

Если бы величин было пять, то надо было бы ввести еще пять величин для «старых» значений и в формулах перехода использовать величины с этими «старыми» значениями. Такой простой подход срабатывает абсолютно всегда и позволяет не думать о порядке вычислений (но требует удвоить число величин).

Вы можете видеть, что простая и ясная связь между рекуррентными соотношениями и текстом алгоритма здесь несколько теряется. Эффективность вычислений достигается за счет частичной потери простоты и ясности. Но зато мы не храним все элементы последовательности. Это также «исчезновение индексов», но в чуть более сложной ситуации.

Как избавиться от «нерегулярностей» в начале последовательности

Последнее «углубление» метода рекуррентных соотношений, рассматриваемое в учебнике (п. 16.6), связано с начальными «нерегулярностями». Уже для последовательности Фибоначчи значение n-го элемента надо отдельно вычислять для $n = 1$ и для $n = 2$. Общая формула применима только начиная с $n = 3$. Соответственно, в алгоритме появляются дополнительные конструкции ветвления, что, конечно, усложняет алгоритм (в алгоритме А81 это — конструкция **если**, в которой основное содержание алгоритма записано после **иначе**).

Если рекуррентные соотношения окажутся еще сложнее, например, очередной элемент будет выражаться через пять

предыдущих, то первые пять значений надо будет задавать отдельно и нам придется написать что-то вроде:

```
выбор
| при n=1:  знач:=...
| при n=2:  знач:=...
| при n=3:  знач:=...
| при n=4:  знач:=...
| при n=5:  знач:=...
| иначе
| ..............
| ..............
все
```

От таких «начальных нерегулярностей» и, соответственно, дополнительных ветвлений в алгоритме, как правило, можно избавиться, если «продолжить последовательность влево» с сохранением рекуррентных соотношений (в математике это называется «доопределением» функции).

Это значит, что мы можем попробовать придумать, подобрать такие a_0, a_{-1}, и т. д. (подобрать элементы «влево» от обычных), чтобы все обычные (непридуманные) элементы последовательности, начиная с a_1, вычислялись через общее рекуррентное соотношение для данной последовательности.

Как это сделать? Рассмотрим пример из учебника — последовательность Фибоначчи — еще раз. Основное рекуррентное соотношение последовательности Фибоначчи — это

$$a_n = a_{n-1} + a_{n-2}$$

Следовательно,

$$a_{n-2} = a_n - a_{n-1}.$$

Используя эту формулу, легко начать вычислять элементы «влево»:

$$\begin{array}{ll} \text{при } n = 2: & a_0 = a_2 - a_1 = 1 - 1 = 0 \\ \text{при } n = 1: & a_{-1} = a_1 - a_0 = 1 - 0 = 1 \\ \text{при } n = 0: & a_{-2} = a_0 - a_{-1} = 0 - 1 = -1 \\ \text{и т. д.} & \end{array}$$

Прием называется «продолжение последовательности влево», потому что мы вычисляем недостающие слева элементы a_0, a_{-1}, и пр. через рекуррентное соотношение и из-

вестные нам элементы последовательности (расположенные «справа» от вычисляемых).

Реально надо довычислить столько «левых» элементов, через сколько предыдущих элементов выражается очередной элемент последовательности с тем, чтобы общее рекуррентное соотношение оказалось применимым для вычисления a_1. Например, для последовательности Фибоначчи достаточно продолжить последовательность влево на два элемента: a_0 и a_{-1}. После этого мы можем переписать рекуррентное соотношение в виде

$$a_n = a_{n-1} + a_{n-2}; \qquad a_{-1} = 1; \qquad a_0 = 0.$$

Чрезвычайно важно, что основное рекуррентное соотношение не изменилось. Изменились только начальные условия. Зато теперь ***любые*** «обычные» элементы последовательности, начиная с a_1, можно вычислять по общей формуле. Соответственно, нам более не нужны развилки — и алгоритм А81 превращается в алгоритм А82.

И, смотрите — опять тот же эффект от применения метода: наши рассуждения несколько усложнились и удлинились, зато алгоритм стал еще проще! Дальше надо порешать задачи, «понабивать руку», довести эти навыки до некоторого автоматизма — и вы с удивлением увидите, как целый класс задач станет для вас простым и рутинным, таким же, как нахождение корней квадратного уравнения.

Конечно, иногда бывают случаи, когда такое «продолжение влево» оказывается невозможным — в формуле возникает какое-нибудь деление на 0 или еще что-то. Но если последовательность удается продолжить влево, то это позволяет обойтись без `если`, без отдельных частных случаев и т. п., дает возможность намного упростить алгоритм, оставить только вычисления по общей формуле.

Что значит научиться методу рекуррентных соотношений

Итак, давайте подведем итоги. После всех «углублений» мы теперь можем сказать, что метод рекуррентных соотношений при решении конкретной задачи состоит в следующем. Школьник должен научиться:

(а) «видеть», распознавать, что эта задача может быть решена методом рекуррентных соотношений;

(б) применять метод и получать сами рекуррентные соотношения;

(в) «продолжать последовательность влево» так, чтобы все ее элементы, начиная с первого, вычислялись по общей формуле;

(г) записывать соответствующий алгоритм без использования таблиц и индексов.

При этом части (а) и (б) являются самыми творческими, самыми сложными в обучении. Части же (в) и (г) достаточно рутинны и просты. Если уж рекуррентное соотношение выписано, то дальше все делается известным образом и просто. Никакой смекалки, никаких «озарений» здесь уже не требуется. Явно выписанное соотношение всегда легко и быстро преобразуется в алгоритм.

Несколько слов об упражнениях (упражнения 1–12, с. 136–138 учебника)

Упр. 1 и 11 по сути являются еще одним «углублением» метода — в этих упражнениях впервые встречаются рекуррентные соотношения с ***двумя*** последовательностями, в которых ***пара*** новых элементов последовательностей выражается через их предыдущие элементы. Сами рекуррентные соотношения приведены в указании к упр. 1 и в тексте упр. 11. Естественно, бывают и тройки, четверки и т. д. последовательностей, но эти случаи в учебнике не затронуты.

Упр. 3а) и 4 практически повторяют задачу про сопротивление гирлянды. Они не должны вызвать никаких затруднений.

А вот упр. 3б) и 3в) достаточно сложны и интересны — в том числе и для сильных учеников (именно поэтому они помечены звездочкой «*»). Самое сложное — догадаться рассмотреть две последовательности одновременно: при решении упр. 3б) надо дополнить соответствующую последовательность аналогичной последовательностью из упр. 3в) и наоборот. И в обоих случаях выражать ***пару*** новых элементов этих последовательностей через ***пару*** предыдущих. Именно это, хотя и в более завуалированной форме, написано в указании к упражнениям.

Повторю, что самое сложное — догадаться применить метод для решения задачи, в которой никаких рекуррентных соотношений нет. В упражнениях эта часть работы проделана авторами учебника — ученикам предлагается составить

алгоритм, когда рекуррентное соотношение либо известно, либо выводится из понятных соображений. Тем не менее, я хочу обратить ваше внимание, что упр. 6 и 7 можно сформулировать, просто как составление алгоритма нахождения квадратного или кубического корня из заданного числа. Конечно, чтобы от этой формулировки перейти пусть даже не к рекуррентным соотношениям, а только к уравнениям, приведенным в учебнике, необходимы дополнительные математические знания. Но обратите внимание, что все же в итоге мы применяем метод рекуррентных соотношений для решения задач, внешне к рекуррентным соотношениям никакого отношения не имеющим.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Конечно, это уже выглядит, как шарлатанство. Если для гирлянды еще как-то можно было понять, почему надо применить метод рекуррентных соотношений, то в задаче «составьте алгоритм нахождения квадратного корня из заданного числа» применение метода рекуррентных соотношений кажется, как минимум, странным.

Тут, пожалуй, мне пора признаться и открыть вам маленький секрет. Честно говоря, метод рекуррентных соотношений, при желании, можно применить почти к любой задаче, в которой что-то вычисляется последовательным повторением некоторых действий (в цикле). Для этого достаточно рассмотреть новые значения величин (после выполнения тела цикла) и выразить их через предыдущие значения этих величин — это и будут рекуррентные соотношения.

Но содержание метода, конечно, не в этом. Если вы уже составили алгоритм, то никакие методы вам, видимо, не понадобятся. Но существует большой класс задач, в которых ***проще*** «выводить» алгоритм, применяя метод рекуррентных соотношений, чем придумывать, «угадывать» алгоритм «сам по себе» — без использования методов алгоритмизации.

16.2. Метод 2 — «однопроходные алгоритмы»

Какие алгоритмы называются однопроходными

Сначала — объяснение слова «однопроходный». Представьте себе, что Робот находится в горизонтальном коридоре, в клетках которого имеются какие-то уровни радиации, например, 1, 3, 7, 5, 7, 3, 1, 7. Или (если вы объясняете без Робота) задана линейная таблица из n элементов. И нас интересует число клеток (число элементов таблицы) с максимальной радиацией. В моем примере максимальный уровень равен 7, а число клеток с максимальным уровнем равно — трем.

Итак, наша задача — подсчитать, сколько клеток имеет максимальную радиацию. Естественное решение, мгновенно приходящее в голову, выглядит так: сначала пройдем по всему коридору и определим максимум; потом пройдем обратно и подсчитаем, сколько клеток совпадает с максимумом. Для таблицы это значит, что сначала надо перебрать все ее элементы и найти максимум, а потом перебрать их еще раз и подсчитать, сколько элементов совпадает с максимальным.

Каждый перебор всех элементов таблицы (так же как каждый проход Робота по коридору) в программировании, как это ни удивительно, называется «проходом», даже если никакого Робота и в помине нет. И алгоритмы обработки информации делятся на

- однопроходные — один раз что-то перебрали и получили ответ;
- двухпроходные, когда элементы таблицы (или иные объекты) для получения ответа перебираются и анализируются дважды;
- трехпроходные и т. д.

Описанный выше алгоритм нахождения числа максимумов (сначала проход по коридору и определение максимума, потом еще один проход и определение, в скольких клетках радиация совпадает с максимумом) является двухпроходным. Либо Робот у нас дважды проходит по коридору, либо элементы в таблице перебираются дважды.

Повторю, что термины «однопроходный», «двухпроходный», «трехпроходный» являются общепринятыми и не зависят ни от какого Робота. Например, компилятор называется однопроходным, если он анализирует текст программы один раз.

Почему однопроходные алгоритмы так важны

Часто бывает, что два прохода сделать либо просто невозможно, либо — по каким-то причинам — крайне нежелательно. Например, обрабатываемые «элементы» могут поступать в компьютер через антенну со спутника. Летает спутник, собирает с датчиков какую-то информацию и, скажем, 50 раз в секунду передает ее на компьютер в центр обработки. А компьютер обрабатывает эти данные по мере поступления и в любой момент должен быть готов ответить на вопрос, каков был, например, максимальный или средний уровень излучения (если измеряется излучение) на сегодняшний день или за прошедший месяц. И все это должно работать непрерывно. Тут, во-первых, никакой памяти не хватит, чтобы запомнить всю поступающую информацию. А во-вторых, на запрос надо ответить быстро — если в этот момент начать анализировать всю информацию (а параллельно продолжать принимать новую), то все это затянется на долго. Кроме того, 50 раз в секунду — это еще не очень много, но бывают ситуации, в которых числа поступают очень быстро, и их можно обработать только раз и тут же забыть (иначе не успеешь обработать следующие).

Итак, бывает необходимо составить однопроходный алгоритм, который каждое число обрабатывает только один раз. Вернемся к задаче про число максимумов, с которой мы начали. Хотя естественное решение, которое первым приходит в голову, является двухпроходным, такая задача вполне может возникнуть в ситуации, когда два раза по коридору пройти нельзя. Например, уровень радиации таков, что Робот выдержит только один проход по коридору, а коридор такой длинный, что запомнить все числа в памяти бортовой ЭВМ Робота невозможно.

В реальном программировании, впрочем, более распространена ситуация, когда необходимость однопроходного алгоритма объясняется временными факторами. Дело в том, что получение и перебор анализируемых элементов, если этих элементов достаточно много, — долгий процесс (поскольку обычно эти элементы лежат во внешней памяти ЭВМ — на диске или еще где-то и все вместе в обычную память не помещаются). В нашей учебной среде аналогичная проблема может возникнуть, если перевод Робота из одной клетки в другую требует длительного времени, является долгим. В таких ситуациях время выполнения алгоритма начинает определяться тем, сколько «проходов» мы делаем, а временем собственно обработки элементов можно пренебречь. Тогда «второй проход» вдвое увеличивает время выполнения алгоритма.

Однопроходные алгоритмы являются, как правило, самыми быстрыми. Поэтому, если в рекуррентных соотношениях мы «убирали» индексы с целью экономии памяти (говоря, что ее может не хватить), то в однопроходных алгоритмах на первом месте — скорость выполнения. Однопроходный алгоритм — это алгоритм, который обычно «проходит» быстрее других.

Пример составления однопроходного алгоритма

Итак, давайте вернемся к начальной задаче — найти число число клеток с максимальным уровнем радиации. И уточним — задачу надо решить за один проход, т. е. алгоритм должен быть однопроходным.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Попробуйте, не читая учебника, решить задачу самостоятельно — и вы увидите, что это не так просто. При изучении любого из методов алгоритмизации чрезвычайно полезно — особенно для сильных учеников — задать задачу из соответствующего класса ***до изучения метода.*** Как правило, это повышает мотивацию и способствует более глубокому усвоению.

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Вам, быть может, полезно будет знать следующее. Имеется строгое математическое доказательство того, что для любой задачи подобного рода (а) существует однопроходный алгоритм; (б) среди всех однопроходных алгоритмов решения есть минимальный в смысле размера памяти, требуемого для его выполнения, и (в) этот минимальный однопроходный алгоритм — единственный (с точностью до замены обозначений, или, как говорят математики, с точностью до изоморфизма). Доказательство этого, а также более глубокое изложение метода однопроходных алгоритмов вы можете найти в [ПдМ], в разделе «Индуктивное вычисление функций на пространстве последовательностей».

Итак, однопроходный алгоритм — самый быстрый. Минимальный однопроходный алгоритм — кроме того, использует меньше всего памяти. Таким образом, минимальный однопроходный алгоритм — это наилучший алгоритм и по быстродействию, и по памяти. Теория утвер-

ждает, что для любой задачи такой алгоритм существует и в каком-то смысле он единственный. Конечно, для преподавания в школе вам все это не понадобится. Однако полезно знать, что это очень широкий класс задач, и для любой из них можно составить наилучший алгоритм — однопроходный и с минимальными затратами памяти.

На школьном уровне для нас гораздо важнее разобраться с тем, как такие задачи решать. И, поскольку уровень школьный, это разбирательство базируется не на строгих теориях, а на простых аналогиях. В учебнике приведена аналогия с ловлей рыбы в простейшей задаче нахождения максимума. Проводим соревнование, кто выловит самую большую рыбу. Простой алгоритм: выловили новую рыбу — сравниваем с той, которая у нас в ведре. Если новая рыба больше, то старую выпускаем, а новую кладем в ведро. Если новая рыба меньше — мы ее просто выпускаем и т. д.

Вообще, все это описание нужно для того, чтобы ввести слова-аналогии, например, «ведро» — для обозначения места, где мы что-то храним (набора величин алгоритмического языка).

Рассмотренный выше пример-аналогия вошел учебник с целью пропаганды бережного отношения к живой природе: чтобы рыба по ходу соревнования не пропадала, каждому участнику выдается ведро, в котором он хранит самую большую из пойманных рыб, а все остальные («ненужные») рыбы в целости и сохранности выпускаются обратно в реку.

Я приведу сейчас еще один — менее «экологичный» — пример-аналогию, который в свое время А. Г. Кушниренко услышал от своего учителя (К.В. Ким), — пример с собиранием грибов.

Итак, та же задача на нахождение максимума (в примере — поиск самого большого гриба). Аналогия звучит так. Человек пошел в лес за грибами. Идет с корзинкой по лесу, по тропинке, и ищет грибы. Найдя новый гриб, сравнивает его с тем, который лежит в корзинке. Если новый гриб больше, то старый (меньший) из корзинки выбрасывается, а новый (больший) кладется в корзинку. Привлекательность этой аналогии в том, что человек идет, «совершает один проход» по лесной тропинке.

В учебнике, повторю, чтобы не разбрасывать бессмысленно вполне годные к употреблению грибы по лесу, мы заменили это собирание грибов на ловлю рыбы, хотя при ловле рыбы мы никуда не идем, а сидим на берегу. Но мы

решили, что природу беречь в данном случае важнее, чем идти.

Ключевое свойство однопроходного алгоритма

В нашей аналогии (в учебнике) в каждый момент происходит сравнение вновь выловленной рыбы с той, что хранится в ведре. Меньшая отпускается, а большая помещается в ведро. Понятно, что при этом к моменту, когда соревнования закончатся, у нас в ведре будет самая большая из пойманных рыб. Алгоритм поиска максимума заключается в том, что, вылавливая очередную рыбу, или получая очередное число от Робота, или взяв очередной элемент таблицы, мы сравниваем его с тем максимумом, который уже нашли до этого момента (с тем, что «хранится в ведре»). И аналогия, в основном, придумана именно для того, чтобы ввести понятие «ведра» — т. е. места, где мы что-то храним.

Итак, первое, на что нужно обратить внимание, — есть процесс, в котором к нам поступают очередные элементы: или Робот идет по коридору и узнает очередные значения радиации; или мы перебираем элементы таблицы; или вылавливаем рыбу за рыбой.

Во всех случаях есть процесс («проход»), когда мы получаем элемент за элементом, число за числом. Мы сидим с ведром на берегу и вытаскиваем очередную рыбу, или идем с корзинкой по лесу и находим очередной гриб, или читаем очередной элемент таблицы — с точки зрения информатики, это просто получение новой порции информации.

Кроме процесса, есть нечто, что в аналогии учебника можно назвать «ведром», — место, где мы что-то храним. Получив новую порцию информации (рыбу, гриб), мы смотрим на нее и на содержимое «ведра» и что-то делаем с содержимым ведра, с нашим хранилищем. Скажем, меняем рыбу в ведре на большую. Не обязательно, чтобы в «ведре», в хранилище было только одно число — там может быть и десяток чисел. Важно другое, — всякий раз, получив очередной элемент («рыбу»), мы что-то проделываем с содержимым нашего «ведра».

Здесь важны обе сущности, с которыми мы работаем, — и «рыба», и «ведро». Мы получаем очередное число из нашей последовательности, вылавливаем очередную рыбу, узнаем очередное значение радиации у Робота или очередной элемент таблицы. Это одна часть. Далее мы сравниваем этот

очередной элемент с чем-то, что мы храним, — с содержимым ведра или корзины, быть может, меняем то, что мы храним, и после этого переходим к следующему элементу.

Очень важно заметить, что при получении очередного элемента мы рассматриваем только сам этот элемент («рыбу») и содержимое нашего хранилища («ведро»), ***но никогда не рассматриваем предыдущие элементы.*** Выловив очередную рыбу, мы смотрим только в «ведро», а каких мы ловили рыб до того, нам уже не интересно, мы это можем забыть. Это самая важная составляющая, ключевое свойство всякого однопроходного алгоритма.

Соответственно, самое существенное в составлении однопроходного алгоритма состоит в придумывании «ведра» — того, что мы будем хранить.

Для задачи поиска максимума (а это простейшая из возможных задач) ответ очевиден: если мы хотим найти максимальное значение, то именно его и надо хранить в «ведре». В алгоритмическом языке информация хранится в величинах. Поэтому с алгоритмической точки зрения «ведро» — это какой-то набор величин алгоритмического языка. Для хранения максимума, например, можно завести величину «**вещ** v». Максимум найти таким образом легко (алгоритм А80). Прошли один раз, нашли максимум.

А как найти число максимумов? Здесь — ключевой момент, ключевой пример для понимания однопроходных алгоритмов. Давайте порассуждаем в терминах «ведра». Итак, у нас будет некоторое «ведро» (и пока мы не знаем, что в нем будет храниться). Пусть мы уже «прошли» какую-то часть коридора и получили значение радиации в очередной клетке, получили очередное значение x. Нам нужно сказать, сколько максимумов в пройденной части коридора, включая x (рис. 3).

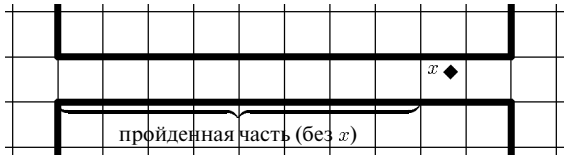


Рис. 3.

Давайте разберемся, что нам для этого нужно помнить и знать, какую информацию о пройденной части коридора (без x) нам надо хранить, чтобы ответить на вопрос, сколько максимумов в новой пройденной части коридора (включая x). Первое, что нам нужно, это старый максимум, максимум в пройденной части коридора (без x). Если x меньше этого максимума, то число максимумов не меняется. Если x равен этому максимуму, то число максимумов надо увеличить на единицу. Если x больше этого максимума, то x становится новым максимумом и он единственный, т. е. число максимумов равно 1. Обозначим k — число максимумов, а m — сам максимум. Тогда переход от старой пройденной части (без x) к новой пройденной части (включая x) запишется так:

```
выбор
| при x<m : |ничего не меняется
| при x=m : k:=k+1
| при x>m : m:=x, k:=1
все
```

Таким образом, для этой задачи надо хранить в «ведре» не одно, а два числа: старый максимум и количество этих максимумов, m и k. Встретив очередной элемент (x) и сравнив его со старым максимумом (m), мы можем изменить, при необходимости, и сам максимум (m), и количество максимумов (k). И к тому моменту, когда мы выйдем из коридора, в «ведре» будет ответ (k). Причем за один проход!

Еще раз подчеркну, в чем же, собственно, состоит метод. Метод состоит в том, чтобы при вычислении какой-то величины, например, числа максимумов в таблице или в коридоре, обрабатывать элемент за элементом последовательно и при этом всякий раз при обработке очередного элемента, анализировать и менять какую-то не очень большую информацию, которую мы храним и обновляем с самого начала («в ведре»).

Ключевой вопрос метода — что такое это «ведро», какую информацию мы должны хранить в нем. Надо рассмотреть процесс обработки очередного элемента x, процесс «удлинения» «просмотренной части» на элемент x, и понять, какая информация нужна для получения ответа на вопрос. Это и есть метод однопроходных алгоритмов. После того как вопрос с «ведром» решен, сам алгоритм — как правило — за-

писывается очень легко. Для числа максимумов в таблице алгоритм А85 приведен на с. 131 учебника.

В однопроходном алгоритме цикл всегда один — элементы перебираются один раз. Если мы ведем Робота по коридору, то получается один цикл **пока**, если перебираем элементы таблицы, то один цикл **для**, в котором перебираются все подряд элементы таблицы.

Однопроходные алгоритмы для работы со строками

Я еще раз подчеркну, что неважно, как именно мы получаем элементы. Слово «однопроходный» означает лишь, что перебор осуществляется один раз. Например, можно перебирать одну за другой буквы в литерной строке. В учебнике разбирается один подобный алгоритм — для подсчета количества слов в строке. Строка состоит из символов, и перебирать мы будем символы. Слово «однопроходный» означает, что мы должны составить алгоритм, перебирающий символы один раз. Рассмотрим, например, строку «Я (два пробела) завтра (четыре пробела) пойду (пробел) в (два пробела) кино (три пробела)!»:

Я␣␣завтра␣␣␣␣пойду␣в␣␣кино␣␣␣!

В этой строке 6 слов. (Заметьте, что восклицательный знак «!» — отдельное слово. «Словом» мы здесь называем ***любую*** последовательность символов (не пробелов) между пробелами или между пробелами и границей строки.)

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я уже говорил, что до изложения и демонстрации применения метода полезно дать задачку, чтобы школьники «помучились», решая ее «как-нибудь». Задача про число слов — очень хороший пример.

Если вы дадите такую задачу, то большинство школьников какой-то однопроходный алгоритм напишет. При этом, однако, решения почти у всех будут с ошибками. Вот типичные ошибки:

— алгоритм «пропускает» слово из одного символа в начале строки (в нашем примере — слово «Я») или
— алгоритм «пропускает» слово из одного символа в конце строки (в нашем примере — слово «!»).

Причем, если ученик допустил первую ошибку и вы просите его ее исправить, то, как правило, он устраняет первую ошибку и делает вторую. И наоборот. Я это знаю по опыту.

Обычно школьники либо ловят перепад «символ—пробел» и считают, что количество таких перепадов и есть количество слов (и тогда они допускают ошибку второго рода — «пропускают» слово в конце), либо ловят перепад «пробел—символ» — и, соответственно, пропускают слово в начале. Поняв наконец в чем ошибка, они обычно пытаются дописать **если** в начале или в конце алгоритма, чтобы учесть и крайние слова. Но после этого обычно алгоритм перестает работать, если строка начинается или кончается группой пробелов. В конце концов в попытках исправить ошибки школьники добавляют такое количество разных **если** и «циклов», что алгоритм становится совершенно запутанным, и ни они сами, ни кто-то другой уже не в состоянии понять, как он работает. А главное — на некоторых строках алгоритм все равно дает неверный ответ (т. е. по-прежнему работает с ошибками).

После этих мучений изучение метода однопроходных алгоритмов и решение задачи с его помощью кажется облегчением и обычно проходит на «ура».

Эту же задачу можно переформулировать для Робота. Робот расположен в левой клетке некоторого горизонтального ряда клеток, ограниченного стеной справа и слева, а над этим рядом расположено какое-то количество горизонтальных стенок (рис. 4):

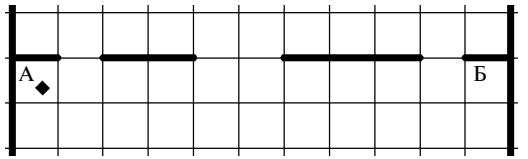


Рис. 4.

Надо перевести Робота из А в Б (слева направо) и посчитать число стенок над Роботом. Обычно, школьники «пропускают» одноклеточные стенки, прилегающие к левому или правому краю (над А и над Б). Здесь аналогом «пробела» является «сверху свободно», а аналогом «не пробела» — «сверху стена». Это та же самая задача, сформулированная в терминах Робота.

Естественно, ее можно переформулировать и для числовых таблиц, например, считать аналогом «слова» группу подряд идущих не-нулей — тогда задача переформулируется в виде «подсчитать, сколько в данной таблице встречается групп подряд идущих не-нулей».

Важно, что как бы мы ни формулировали задачу — в символьных терминах, через стены на поле Робота или в терминах чисел в таблице — это одна и та же задача, и проблемы у учеников возникают практически одни и те же.

Я эту задачу решать не буду, поскольку ее решение приведено в учебнике. Я лишь только еще раз обращаю ваше внимание, что если решать ее «по наитию», без применения метода однопроходных алгоритмов, без анализа, что у нас за «ведро», что должно быть в нем в начале, как будет меняться его содержимое, то, как правило, решение получится с ошибками.

Трудная задача на составление однопроходного алгоритма

Я же изложу задачу, которой в учебнике нет. Она мне очень нравится: это одна из самых красивых задач, хотя и достаточно сложная. Я показываю ее для вас (ибо вы должны знать больше и глубже, чем ваши ученики), а вы можете использовать ее при работе с сильными учениками — лидерами классов. Я ее сформулирую в терминах Робота, а вы сами переформулируете ее для таблиц, если захотите.

Итак, пусть у нас имеется горизонтальный коридор, в каждой клетке которого есть какая-то температура. Робот должен пройти по коридору ***(конечно же один раз!)*** и подсчитать следующую интересную величину. Обозначим температуру в клетках коридора: t_1 — в первой клетке, t_2 — во второй и т. д. Рассмотрим ***все возможные*** «куски коридора» от i-ой клетки до j-ой ($i \le j$), и для каждого такого «куска» подсчитаем сумму температур S_{ij} в его клетках:

$$S_{ij} = t_i + t_{i+1} + \cdots + t_j .$$

Если всего клеток в коридоре n, то S_{1n} — сумма температур во всех клетках коридора. S_{33} — «сумма» температур в одной 3-ей клетке, т. е. просто температура в 3-ей клетке. S_{25} — сумма температур в клетках со 2-ой по 5-ую включительно. Понятно? Так вот, рассмотрим ***все*** такие S_{ij} для всех возможных i и j и среди S_{ij} выберем максимальное число — это и есть та величина, которую нам надо найти.

Итак, еще раз, наша задача — составить алгоритм, вычисляющий максимум из всех возможных S_{ij}, где S_{ij} — сумма температур в «куске» коридора от i до j включительно. Другими словами, если мы рассмотрим ***все возможные*** «куски» коридора, в каждом «куске» просуммируем температуру всех клеток, а потом среди всех полученных сумм найдем максимальную, то это и будет то, что мы хотим найти. Если температура во всех клетках больше либо равна нулю, то ответ очевиден — это просто сумма всех температур во всех клетках. Но температура может быть и меньше нуля — и поэтому так просто задачу не решить.

Задача, на мой взгляд, замечательна тем, что по ее формулировке ни за что не догадаешься, как же, собственно, эту задачу решать, кроме как полным перебором всех случаев — прямо по определению. Но это значит, что надо писать алгоритм, который будет рассматривать все диапазоны, запоминать где-то вычисленные суммы, потом все эти суммы перебирать и искать максимум. Такой громоздкий и неэффективный алгоритм еще можно составить. Но запретите школьникам использовать в алгоритме таблицы — и они вообще не смогут эту задачу решить!!! А я вам сейчас покажу, как ее решить за один проход и очень просто! Как обычно, когда применяются методы алгоритмизации, пройдем один раз по коридору, пособираем что-то такое в «ведре» — и получим ответ!

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Я специально вам привожу такую формулировку, чтобы показать огромную мощь применения метода. Это касается всех излагаемых в учебнике методов — овладение ими действительно дает огромную мощь. На простых задачах (вроде поиска максимального элемента) это не так заметно — там требуемый алгоритм можно придумать и «так» без знания и целенаправленного применения методов алгоритмизации. Но есть задачи, в которых просто так, без применения метода, догадаться, какой будет ответ, по-моему, нельзя. По виду последней задачи ни за что не скажешь, что ответ можно получить за один «проход», просмотрев клетки коридора только один раз. Именно поэтому она так хороша для демонстрации действенности метода однопроходных алгоритмов даже в его школьной («ведерно-рыбной») формулировке. Эта задача показывает, что жизнь отнюдь не сводится к умению писать «циклы» и **`если`**.

Решим эту задачу. Вспомним, в чем состоит метод однопроходных алгоритмов. Прежде всего мы должны посмотреть на все это, как на последовательный процесс. Пусть мы уже прошли какую-то часть коридора (А) и обрабатываем очередной элемент — температуру x в клетке, где сейчас стоит Робот. Саму эту клетку мы будем обозначать также буквой x (см. рис. 5).

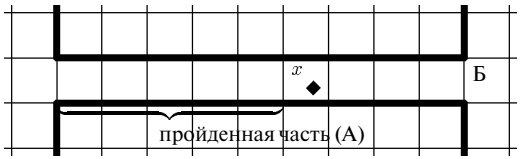


Рис. 5.

Итак, часть коридора (А) мы прошли и стоим в очередной клетке x с температурой x. В соответствии с методом, мы должны постараться выразить ответ для новой пройденной части ($A + x$), т. е. понять, каков максимум S_{ij} среди всех «кусков» от i до j, если рассматривать ***только*** новую пройденную часть ($A + x$).

Обозначим этот максимум M и будем считать, что он хранится в «ведре». Это значит, что когда мы переходим в новую клетку (x), в «ведре» (т. е. в M) хранится максимум S_{ij} для старой пройденной части последовательности (для A), а мы должны так «обработать» новую клетку, чтобы после обработки в «ведре» хранился максимум для новой пройденной части (т. е. для $A + x$). Другими словами, нам надо понять, как меняется M при переходе от A к $A + x$.

Рассмотрим ***все возможные*** «куски» внутри $A + x$. Эти куски можно разбить на две группы (рис. 6):

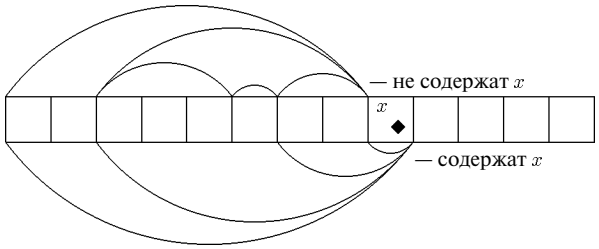


Рис. 6.

— «куски», не содержащие x, т. е. лежащие внутри A;
— «куски», содержащие x, т. е. начинающиеся где-то, а заканчивающиеся на x (допуская некоторую вольность, мы их будем обозначать S_{ix})

Соответственно, новый максимум — это максимум среди всех S_{ij}, как не содержащих, так и содержащих x. Максимум среди всех S_{ij}, не содержащих x, — это максимум среди всех S_{ij} в A, т. е. старое значение M, то, которое мы храним в «ведре» к началу обработки x. Максимум среди всех S_{ix}, содержащих x, — это величина новая, давайте мы ее как-нибудь обозначим, например, N. Тогда

$$M_{\text{новое}} = \max(M_{\text{старое}},\ N).$$

Теперь наша задача — узнать, как вычислить N, ***используя какие-нибудь характеристики, какую-нибудь информацию об*** A, которую мы сможем добавить в «ведро» и хранить, подобно тому как в задаче про число максимумов мы «добавили» в ведро сам максимум.

Итак, ***(внимание! — это ключевое место в применении метода)*** нам надо максимум среди S_{ix} (т. е. N) выразить через x и какие-то величины, зависящие только от A (от чего-то, что мы будем хранить в «ведре»). Для этого можно сами S_{ix} разбить на x и на оставшуюся часть — S_{iw}, где w — последний элемент в A, т. е. элемент перед x (рис. 7):

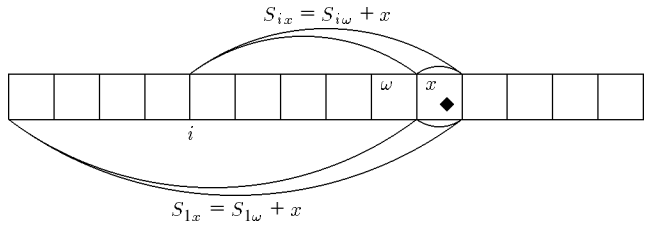


Рис. 7.

Точнее говоря, таким образом представляются все S_{ix}, за исключением S_{xx} (при $i = x$). S_{xx} просто равно x и никакой «предшествующей» части не содержит. Таким образом,

$$N = \max(S_{ix}) \text{ для всех } i = \max(S_{xx}, S_{iw} + x) \text{ для всех } i =$$
$$= \max(x, S_{iw} + x) \text{ для всех } i = \max(x, NNx)$$
$$\text{где } NNx = \max(S_{iw} + x) \text{ для всех } i =$$
$$= x + \max(S_{iw}) \text{ для всех } i$$

Уследили почему я выделил элемент x? Мы должны все выражать через то, что было раньше, должны придумать, что будем хранить в «ведре». Поэтому я отделил новый элемент x, чтобы выделить остаток, зависящий только от старой «пройденной части» — от A.

И смотрите, что получилось: «max(S_{iw}) для всех i» — это максимум среди всех S_{ij}, оканчивающихся на w, т. е. на последний элемент A. Но это просто «старое» (т.е. для A без x) значение N!

$$N\text{старое} = \max(S_{iw}) \text{ для всех } i.$$

Тогда выведенные нами выше формулы перепишутся в виде:

$$\begin{aligned} M_{\text{новое}} &= \max(M_{\text{старое}}, N_{\text{новое}}) \\ N_{\text{новое}} &= \max(x, NNx), \\ &\quad \text{где } NNx = x + N_{\text{старое}} \end{aligned}$$

Или, если убрать NNx и поставить вычисление $N_{\text{нового}}$ до его использования:

$$\begin{aligned} N_{\text{новое}} &= \max(x, x + N_{\text{старое}}) \\ M_{\text{новое}} &= \max(M_{\text{старое}}, N_{\text{новое}}). \end{aligned}$$

И — мы с удивлением можем это констатировать — теперь пара ($N_{\text{новое}}$, $M_{\text{новое}}$) выражается через пару ($N_{\text{старое}}$, $M_{\text{старое}}$) и очередной элемент x. Таким образом, если мы будем «хранить в ведре» ***всего два числа*** N ***и*** M, то мы без труда за один проход вычислим искомую величину (M для всего коридора).

Итак, в «ведре» надо хранить M (это та величина, которую требовалось найти в задаче) и N — максимум из всех сумм S_{ij}, заканчивающихся на правом краю (на последнем элементе) соответствующей части коридора.

Самый существенный прием в этом методе состоит в том, что мы не пытаемся сразу подсчитать конечную величину для всего коридора целиком. Вместо этого мы рассматриваем процесс «прохода», перебора элементов и «шаг перехода» при обработке очередного элемента. Другими словами, рассматриваем нашу задачу сначала для коридора из одной клетки, потом для коридора из двух начальных клеток и т. д., пока, последовательно «удлиняя» пройденную часть, не получим искомое значение для всего коридора целиком. Поэтому наше внимание было сосредоточено не на том, как подсчитать столь непросто заданную величину, а на том,

как она ***меняется***, когда мы «удлиняем» пройденную часть на очередной элемент x. Именно за счет этой смены акцентов нам и удалось так удивительно решить эту задачу.

Конечно, нам еще надо найти, чему равны M и N в первой клетке коридора (когда только одна клетка «пройдена»), но это уже совсем легко — для такого коридора, состоящего из одной клетки x, прямо по определению $M = x$ и $N = x$. Поэтому алгоритм в целом запишется так:

```
алг вещ максимальная подсумма
  дано |Робот в начале коридора (рис.4)
  надо |Робот вышел из коридора в клетку Б (рис.4)
       |знач = максимум из сумм температур по всем
       |"кускам" коридора от i до j, для всех
       |допустимых i и j
нач вещ N, M, x
 x:=температура |в первой клетке коридора
 M:=x; N:=x     |задание начальных значений
                |в "ведре"
 вправо         |переход в следующую клетку коридора
 нц пока снизу стена |пока не вышли из коридора
  x:=температура |получили очередной элемент x
  N:=max(x,x+N)  |вычисление N нового
  M:=max(M,N)    |вычисление M нового
  вправо         |переход в следующую клетку
 кц
 знач:=M        |ответ
кон
```

И посмотрите, какой простой у нас опять получился алгоритм! Глядя на него, даже трудно поверить, что он считает такую «заковыристую» величину — не правда ли? По существу у нас в цикле всего две содержательных команды присваивания, а задача-то вначале казалась ох какой сложной!

И это все, я еще раз подчеркиваю, — результат применения методов алгоритмизации. Когда вы знаете метод, умеете его применять, вы можете решать сложные и непонятные на первый взгляд задачи таким вот образом. И вся сложность уходит в метод, а алгоритм получается простой почти до предела.

Это, конечно, уже посложнее, чем корни квадратного уравнения. При добавлении очередного элемента мы должны разобраться с тем, что надо хранить в «ведре», понять, что делать с «ведром» и очередным элементом, чтобы двигаться дальше. Но тем не менее, действуя по общей схеме, мы через какое-то время ответ получим. И обратите внимание еще раз, что при этом мы думаем совсем не так и не над

тем, над чем думали бы без применения метода. Мы вообще не думаем, как перебрать элементы, куда записать суммы и т. п. Мы думаем над переходом от старой «обработанной части» к новой. И в наших рассуждениях алгоритмическая и математическая составляющие задействованы обе одновременно. Потому-то я и говорю, что § 16 является сложным, — здесь происходит переплетение логической и алгоритмической культур. Мы решаем алгоритмические задачи, думаем про действия и процессы, но рассуждения наши носят скорее логико-математический характер. Именно, от этого задействования математической культуры и переплетения математической и алгоритмической составляющих и возникают сложности.

Возвращаясь к учебнику, замечу, что приведенные в нем задачи, как и большинство упражнений на однопроходные алгоритмы, достаточно просты. Рассмотренную выше задачу разумно давать в воспитательных целях сильным учащимся до изучения метода. Как правило, они напишут очень сложный алгоритм, несколько вложенных друг в друга циклов, естественно с ошибками и т. п. Да и его они «добывают» с огромными мучениями. А когда после этого вы покажете метод и получившийся элементарный алгоритм — у них будет некоторый шок. И уж будьте уверены: теперь они в методе разберутся и решать задачи с его помощью научатся.

И еще одно замечание. Эта задача и предыдущая — с некоторым количеством стенок над Роботом (количеством слов в строке) — примерно одинаковой сложности. Ключевая разница состоит в том, что в случае стенок школьники алгоритм в целом пишут. Их надо ловить, указывать, что они не учли одноклеточную стенку у правого края или одноклеточную стенку у левого края. Алгоритм в целом у них вроде есть, только иногда («в особых случаях») не работает. А у задачи с суммами формулировка такая, что не надо ничего говорить про отдельные случаи, просто они ее не решают — и все. А вы можете показать, что эту задачу не только можно решить, но и более того — алгоритм получается очень простой, никакой алгоритмической сложности как бы и нет.

Ну и наконец, я еще раз скажу, что изложение в школьном учебнике — очень сильное упрощение. Собственно, все начинается и заканчивается на уровне «ведра». Если кого-то эта область интересует на более глубоком уровне, если кто-то хочет сам по-настоящему разобраться в этом методе или организовать факультатив для сильных ребят, то я рекомендую наш вузовский учебник «Программирование

для математиков» [ПдМ], раздел «Индуктивное вычисление функций на пространстве последовательностей». Там этот метод алгоритмизации изложен целиком, со всеми деталями и подробностями, строго математически, с математическими доказательствами существования и единственности минимального однопроходного алгоритма для вычисления любой функции (т. е. для любой задачи), критериями проверки минимальности и т. п. Это все, конечно, требует гораздо более высокого уровня математической подготовки, но если доказательства пропустить, то все остальное вполне можно воспринять примерно в таком стиле, как здесь изложено, — без доказательств. Поэтому, если вы захотите этим методом овладеть более глубоко и по-нормальному, можете попробовать изучить эту книгу.

Если взять задачи, встречающиеся просто «по жизни» — на практике, то наиболее применимыми методами алгоритмизации, на мой взгляд, являются два — метод однопроходных алгоритмов и «инвариант цикла». Задачи на эти два метода встречаются чаще всего, это методы с наиболее широким классом применений.

16.3. Метод 3 — «инвариант цикла»

Мы переходим к третьему методу этого параграфа, который называется «инвариант цикла». Точнее — и к великому моему сожалению — в учебнике практически изложен не столько метод «проектирования цикла с помощью инварианта», сколько просто понятие «инварианта цикла». Содержательное и практическое применение этого понятия для решения алгоритмических задач, т. е. собственно метод алгоритмизации, по большому счету остались за рамками учебника. И, даже то, что в учебнике есть сделано, конечно на достаточно простом школьном уровне. И тех, кто хочет овладеть методом по-настоящему, содержательно, я опять отсылаю к учебнику [ПдМ], раздел «Проектирование цикла с помощью инварианта».

Итак, инвариант цикла. Повторю, что в учебнике мы ограничились введением самого понятия, а метод как таковой не разбирали, хотя о каких-то простейших применениях понятия инварианта и его пользе при составлении алгоритмов немного рассказали.

Что такое инвариант цикла

Инвариант цикла — очень простое понятие. Какой бы ни был цикл, внутри него обычно, что-то меняется. Например, положение Робота, значения каких-то величин, возможно что-то еще — состояния каких-то объектов и значения каких-то величин. В примере, который мы только что обсуждали, менялось положение Робота и значения величин N, M, x. Можно попытаться описать взаимосвязь между всеми этими величинами и положением Робота. Фактически, я так и делал, когда говорил, что x — температура в той клетке, где стоит Робот; N — максимальная из сумм, которые кончаются на правом краю пройденной части коридора; M — максимальная из всех сумм вообще в пройденной части коридора. Слова «пройденная часть коридора» связывают значения этих величин с положением Робота.

Грубо и приближенно, такое описание взаимосвязи между меняющимися в цикле объектами (в данном случае между положением Робота и значениями величин M и N), такое общее описание взаимосвязи и называется «инвариантом» цикла. Главное, чтобы это описание было общим — верным для любого шага цикла (в нашем примере — для любой клетки коридора), не зависящим от того, выполнилось

тело цикла один, два, миллион раз или не выполнилось ни разу, т.е. чтобы наше описание взаимосвязи не менялось в ходе выполнения цикла, оставалось неизменным (по латыни «invariant» и значит «неизменный»), независимо от того, сколько раз выполнялось тело цикла.

Высказывание «значение величины x равно температуре в клетке, где стоит Робот» в терминах информатики — ***условие***. Оно может быть верным, если x действительно равно указанной температуре, либо ложным, если в силу каких-то причин значение величины x окажется не совпадающим с температурой в названной клетке.

Неизменное условие, описывающее взаимосвязь меняющихся в цикле объектов, и называется «инвариантом». Здесь, конечно, мы сильно упрощаем. На «настоящий» инвариант накладывается еще целый ряд ограничений, которым он должен удовлетворять. Отнюдь не любое неизменное условие называется инвариантом. Реальный инвариант должен быть еще и «полезным», но понятие «полезно» (как и стоящие за ним строгие математические рассуждения) я здесь раскрывать не буду. Мы это пропустим.

В рамках школьного курса акцент следует сделать на двух свойствах инварианта.

Первое. При составлении любого алгоритма полезно описать взаимосвязь между меняющимися в цикле объектами (т. е. явно выписать инвариант), чтобы лучше понимать, что происходит. Ведь такое описание будет статическим, неизменным, не зависящим от динамики выполнения цикла. А опыт показывает, что человек гораздо лучше воспринимает и понимает статические (неизменные) условия. Вспомните, насколько статические **дано** и **надо** проще понять, чем текст алгоритма. Кроме того, здесь фактически происходит связь и опора на другую культуру мышления.

И второе. Если уж мы такое неизменное условие — «инвариант», — придумали, если взаимосвязь между объектами выразили, то, зная, как в ходе выполнения тела цикла должен меняться один из объектов, мы можем не придумывать, а ***выводить***, как будут меняться остальные объекты. В этом состоит практическая и немедленная польза от использования инварианта. В задаче с суммами, сформулировав взаимосвязь между положением Робота и величинами N и M и зная, что Робот должен сместиться на одну клетку вправо по коридору, мы в сущности ***«выводили»***, как должны меняться N и M.

Основное практическое значение инварианта состоит в том, что, зная, как меняется одна величина, и зная взаимосвязь между величинами, можно выводить, как должны меняться другие. Не придумывать, а выводить.

Обычно инварианты достаточно просты. Например, для задачи подсчета числа стенок над Роботом (рис. 4) инвариант цикла можно сформулировать так:

> ***«N = число стенок в пройденной части, включая клетку, в которой стоит Робот (т. е. число стенок слева или над Роботом)»***

Этот инвариант связывает значение величины N и положение Робота. Теперь, имея инвариант, имея такое описание взаимосвязи, мы можем сказать, что в начальный момент N ***должно быть*** 1, если над первой клеткой есть стена, либо N ***должно быть*** 0, если стены над первой клеткой нет. Обратите внимание на это ***«должно быть»***, которое мы ***вывели*** из инварианта, ***подставив*** в инвариант в качестве места положения Робота первую (левую) клетку.

Аналогично, при переходе в следующую клетку (при «удлинении» пройденной части) мы, рассмотрев инвариант ***до*** перехода и ***после*** перехода, можем ***вывести***, как должно меняться N при таком переходе (при смещении Робота на одну клетку вправо):

$$N_{\text{новое}} = \begin{cases} N_{\text{старое}} + 1, & \text{если над Роботом первая клетка (левый край) новой стенки} \\ N_{\text{старое}}, & \text{иначе} \end{cases}$$

Мы придумали взаимосвязь (инвариант) и решили, что Робот будет двигаться вправо шаг за шагом, клетка за клеткой, каждый раз в теле цикла смещаясь на одну клетку вправо. После этого нахождение начальных значений величин и формул их преобразования в теле цикла уже не требует анализа всей задачи. Достаточно «подставить» в инвариант соответствующие положения Робота и ***вывести***, чему должны быть равны и как должны меняться эти величины, чтобы инвариант (т. е. описанная взаимосвязь объектов) сохранился.

В большинстве задач инварианты достаточно просты и, как правило, очевидны. Однако даже для одной и той же задачи, обычно можно придумать несколько разных инвариантов и, соответственно, получить разные алгоритмы.

В примере выше я полагал, что Робот расположен в правой клетке пройденной полосы — и получил выписанный выше инвариант. Если считать, что «пройденная полоса» не включает клетку с Роботом, то получится другой инвариант и другой алгоритм.

Конечно, на этом примере трудно показать полезность, осмысленность инварианта. Задачу можно решить и без инварианта, и явное выписывание инварианта в данном случае скорее проясняет смысл понятия «инвариант», чем реально помогает решить задачу (составить алгоритм). Просто в этой задаче взаимосвязь между числом препятствий и положением Робота очевидна.

Сейчас я покажу, что такая взаимосвязь не всегда очевидна и инвариант иногда здорово помогает либо понять алгоритм (первое свойство инварианта), либо составить его (второе свойство — «вывод» как должны меняться величины). Я приведу пример из учебника, в котором этот инвариант не очевиден и, более того, кажется странным.

Но до того повторю еще раз, что инвариант — это неизменное условие, описывающее взаимосвязь объектов (величин) в цикле. Смысл его состоит в том, что он: (1) помогает лучше понять, что делает цикл, когда мы анализируем уже составленный нами или другим человеком алгоритм; (2) помогает записать цикл при составлении алгоритма за счет «вывода» того, как должны меняться какие-то объекты, величины в цикле, чтобы сохранить инвариант при изменении основного объекта за один шаг выполнения тела цикла (в наших примерах — при изменении положения Робота).

Итак, задача, которая есть в учебнике — алгоритм А87:

```
алг вещ степень (арг вещ a, арг цел n)
  дано n≥0
  надо |знач=a**n
нач цел k, вещ t, b
| b:=1; t:=a; k:=n
| нц пока k≠0
| | если mod(k,2)=0
| | | то t:=t*t; k:=k/2
| | | иначе b:=b*t; k:=k-1
| | все
| кц
| знач:=b
кон
```

Утверждается, что этот алгоритм вычисляет a^n. Если непредвзято на него взглянуть, то мы вынуждены будем признать, что вообще непонятно, какое отношение этот алгоритм имеет к задаче вычисления a^n. Казалось бы, если надо a возвести в степень n, то должен появиться цикл n **раз**, в котором что-то будет n раз умножаться на a. Но приведенный алгоритм — совсем другой. Какое отношение имеет он к нашей задаче?

Зададим простейший вопрос — как проверить, возводит этот алгоритм a в степень n или не возводит? Получится в ответе a^n или не получится? Как это понять?

Используем первое свойство инварианта (ведь алгоритм уже написан, и нам надо «всего лишь» его понять!). Инвариант цикла в этом алгоритме выглядит так:

$$b \times t^k = a^n.$$

Величины a и n остаются неизменными (в цикле не меняются), а величины b, t и k меняются при выполнении тела цикла. Инвариант описывает неизменную взаимосвязь между этими меняющимися величинами — между b, t и k.

Что значит «взаимосвязь описывается инвариантом»? Это значит, что всякий раз после выполнения тела цикла условие (инвариант) выполнено.

Прежде чем двигаться дальше, нам необходимо проверить, что выписанное мною условие — действительно инвариант, что оно действительно будет выполнено до и после любого выполнения тела цикла, что выполнение тела цикла наш инвариант сохраняет, не разрушает.

Давайте это проверим. В начале величинам b, t, k присваиваются значения $b = 1, t = a, k = n$, поэтому $b \times t^k = 1 \times a^n = a^n$, т. е. в начале инвариант выполнен. Теперь нам надо показать, что, как бы ни выполнялось тело цикла, инвариант сохранится. В теле цикла у нас бывает два разных случая — для четных и нечетных k. Если k — четное, то

$$t^k = t^{2(k/2)}$$

и поэтому переход $t := t^2; k := k/2$ инвариант не меняет. Если k — нечетное, то

$$b \times t^k = (bt) \times t^{(k-1)}$$

и поэтому переход $b := b \times t; k := k - 1$ инвариант также не меняет. Таким образом, сколько бы раз и как бы ни выполнялось тело цикла, инвариант сохранится. Заметьте, что мы показали это ***чисто математически***.

Итак, выписанное нами условие — действительно инвариант. И оно сохраняется, сколько бы раз ни выполнилось тело цикла. Поэтому оно будет выполнено и после окончания выполнения всего цикла целиком. Но цикл у нас заканчивается только тогда, когда $k = 0$. Таким образом после окончания цикла (при $k = 0$) наш инвариант превратится в условие $b \times t^0 = a^n$, т. е. $b = a^n$. И снова мы чисто математически доказали, что после окончания цикла b будет равно a^n. Заметьте, что мы не говорим, ***как*** будут меняться величины при выполнении алгоритма, мы рассуждаем совершенно по-другому, в какой-то совершенно иной плоскости.

Теперь нам осталось только показать, что цикл рано или поздно кончится, что алгоритм не «зациклится». Как это доказать? (Ведь мы не разбирались и даже не знаем, что и как будет происходить с величинами при выполнении цикла!) Мы начинаем с $k \geq 0$, а цикл кончается при $k = 0$. Но ***при каждом*** выполнении тела цикла k уменьшается либо вдвое ($k := k/2$), либо на 1 ($k := k - 1$). Значит, через какое-то время k станет равным 0 и цикл закончится.

Вот теперь действительно все. Мы показали, что:

а) выписанное нами условие — действительно инвариант;
б) ***если*** цикл закончится, то в b будет ответ — a в степени n;
в) цикл обязательно закончится.

А значит в b получится ответ. Наш алгоритм действительно считает a^n. Мы это доказали, хотя при этом мы даже не интересовались ***как*** это получится, какой будет процесс, как будут меняться величины и пр. Алгоритм написан, его правильность доказана, а как он будет выполняться и что при этом будет происходить — это другой вопрос. Как выяснилось, этого мы можем даже и не понимать — и это нам совсем не мешает!

Вообще такое впечатление, как будто я вам показал какой-то фокус, не правда ли? «Ловкость рук и никакого мошенства». Вроде бы и алгоритм написали, и доказали, что он правильный и считает, что надо, а все-равно непонятно — как это? Как алгоритм это делает?

Заметьте, что этот эффект более или менее проявлялся при применении всех методов алгоритмизации. Алгоритмы получаются простые, но их связь с первоначальной постановкой задачи отнюдь не очевидна. Если бы я алгоритм подсчета максимума из сумм сразу выписал на доске, то тоже было бы не очень понятно, почему этот алгоритм дает правильный результат. А здесь мы не только сначала алгоритм

выписали (а не составили!), но к тому же и правильность его доказали, не разбираясь полностью с тем, как он будет выполняться. Это еще одна иллюстрация к замечательным результатам применения методов алгоритмизации.

На самом деле, я, конечно, должен выйти за рамки учебника и рассказать вам еще одну вещь. Я обязан это сделать, чтобы вы все-таки представляли себе реальное место и роль инварианта. Дело в том, что приведенный выше алгоритм, конечно, не «упал с неба». На самом деле я ***сначала*** придумал этот инвариант, а уж ***потом*** вывел из него алгоритм. Я обязан в этом признаться, иначе это действительно будет больше похоже на фокусы, чем на применение метода алгоритмизации при составлении алгоритмов.

Как на самом деле был составлен этот алгоритм?

Давайте, я бегло вам изложу, как на самом деле был составлен этот алгоритм. Вначале был придуман инвариант. Точнее, кроме собственно инварианта и одновременно с ним были придуманы еще две вещи (это два ключевых требования при придумывании инварианта):

1) набор таких чисел, которые ***легко*** подставить в инвариант в самом начале, чтобы инвариант оказался выполненным (это $b = 1$, $t = a$, $k = n$, при которых инвариант превращается в тождество $1 * a^n = a^n$). Наличие такого ***легко получаемого*** стартового набора — одно из основных требований к инварианту;

2) такие значения ***некоторых*** величин, при которых из инварианта получается ответ (это $k = 0$, когда инвариант превращается в равенство $b = a^n$). Получение из инварианта ответа — второе ключевое требование к инварианту.

После этого переход от набора величин 1) к состоянию 2) был представлен (воображен), ***как процесс*** последовательного уменьшения k: от $k = n$ до $k = 0$. В этот момент я еще ***не знал***, как именно мы будем уменьшать k, как будут меняться величины и пр. Я просто вообразил некоторый процесс уменьшения k, который от стартового набора величин ведет к ответу, ***никак не детализируя*** этот процесс, даже не зная еще возможно ли это все в принципе. Обратите внимание — если я ***потом*** научусь, все равно как, уменьшать k, не меняя инвариант, то в конце при $k = 0$ я получу ответ.

Описанное выше «придумывание» — безусловно творческое действие. Конечно, знание того, что такое инвариант, знание, что нужно придумать, а также некоторый опыт в придумывании инвариантов позволяют тратить на придумывание инварианта не больше творческих усилий, чем на придумывание алгоритма «в лоб». Но тем не менее, это — творческая деятельность.

А вот после этого началась «рутина» — просто последовательное (и достаточно «тупое») применение метода. Прежде всего, глядя на инвариант, надо было придумать, что будет ***шагом*** в воображаемом пока процессе перехода от старта к ответу, т. е. придумать такие действия, ***уменьшающие*** k, которые бы сохраняли инвариант. Сначала я понял, что k всегда можно уменьшить на 1, а потом, что для четных k можно уменьшать k вдвое, и при этом k будет уменьшаться намного быстрее. Глядя на инвариант и зная, что надо как-то так уменьшить k и изменить другие величины, чтобы инвариант сохранился, сделать это было уже не трудно. Параллельно я получил формулы, которые мы писали выше, показывающие, как при уменьшении k надо изменить b и t, чтобы сохранить инвариант.

А уж после этого я просто записал приведенный выше алгоритм.

Заметьте, что мы, анализируя алгоритм, шли в обратном порядке. При практическом составлении алгоритма по методу проектирования цикла с помощью инварианта ничего доказывать не надо — все эти рассуждения и «доказательства» у нас появляются по ходу придумывания инварианта и тела цикла (шага процесса), по ходу составления (точнее сказать «получения») алгоритма.

И — вспомните второе свойство инварианта («если известно, как меняется один объект, то позволяет выводить, как должны меняться другие»), — в реальности мы придумываем, кто этот «один объект» и как он должен меняться (в начале в самых общих чертах, как-то «уменьшение k»), а потом уж действительно в каком-то смысле «выводим» из инварианта, что делать с остальными объектами.

Конечно, можно придумывать разные инварианты, разные процессы получения одного и того же ответа из разных исходных «наборов» или по разным «путям» — и алгоритмы будут получаться разные.

Приведенный выше инвариант — отнюдь не «какой-нибудь», он не случаен и вообще не так прост. Это инвариант, описывающий так называемое «двоичное» возведение в степень — чрезвычайно быстрое и эффективное. Например, при $n = 1000$ в нашем алгоритме будет выполнено всего око-

ло 15 операций умножения (вы можете сами проверить это, а также — на досуге — найти связь между количеством операций умножения и двоичной записью числа n). По сравнению с тривиальным алгоритмом возведения в степень через цикл n раз (где таких операций было бы 1000) наш алгоритм почти в 70 раз быстрее! И с ростом n эта разница в скорости только увеличивается.

Вернемся к учебнику

Ну а теперь вернемся к учебнику. В рамках учебника изучение понятия инварианта цикла может происходить на двух уровнях.

Первый уровень — здраво осмысленный, наглядно-интуитивный. Просто полезно, независимо от того, как мы используем инвариант и используем ли его вообще, записать статическое (неизменное) условие, описывающее взаимосвязь между меняющимися объектами (величинами). Это полезно для понимания того, что происходит в цикле, что будет, когда он несколько раз выполнится, и т. п. Это — уровень, не содержащий никакой теории, вполне усваиваемый и достаточный для основной массы учеников.

Следующий, более высокий уровень — использование инварианта как метода алгоритмизации при составлении алгоритмов. Здесь за рамками школы оставлен вопрос о том, как научиться придумывать инварианты. Но если инвариант задан или придуман «по наитию» (что, собственно, ничем не хуже, чем придумывание «по наитию» всего алгоритма), то его можно использовать для «вывода» тела цикла при составлении алгоритма, а также для проверки правильности всего алгоритма.

Этот второй уровень — придумывание сначала инварианта, а потом вывод из него алгоритма — кажется более сложным в основном из-за непривычности. Начинать составление алгоритма с придумывания инварианта можно, но для этого надо, чтобы голова немного «повернулась», да и культура нужна другая — уже не только алгоритмическая, но и логическая (математическая).

Именно из-за существенного задействования этой логической, математической составляющей мышления § 16 и кажется таким сложным. Да он и является, пожалуй, самым сложным с учебнике (вспомните диаграмму на с. 52). Здесь не обойдешься только здравым смыслом на уровне перемещения Робота по клетчатому полю. Поэтому при преподавании в 7–8 классе или в слабых классах § 16 можно пропускать.

Упражнения на инвариант цикла (это упр. 18, 19 на с. 139) выглядят так. Задан цикл, внутри которого написаны какие-то команды присваивания, а надо написать инвариант. Поскольку про сами циклы не говориться, что именно они считают, после выписывания инварианта школьники должны сказать, что же, собственно, этот цикл делает. До выписывания инварианта это непонятно (какие-то присваивания — мало ли, что получится). А когда инвариант написан, в него можно поставить условие окончания цикла — и получится ответ на вопрос, что именно считается в цикле.

Поскольку в этих упражнениях циклы ***уже*** написаны, а инвариант только еще надо придумать, то, фактически, это упражнения на первый уровень понимания инварианта. Инвариант, как метод алгоритмизации, как метод составления алгоритмов, остается за рамками учебника и за рамками упражнений. Инвариант в учебнике — это простенький прием «давайте опишем взаимосвязь в виде неизменного условия». И все.

16.4. Метод 4 — рекурсия

Последний — четвертый — метод этого параграфа (он помечен в учебнике звездочкой) называется «Рекурсия». Звездочка показывает и некоторую сложность в восприятии этого материала, и относительно меньшую важность рекурсии, как метода алгоритмизации, по сравнению с первыми тремя методами.

Еще одно замечание «вперед»: если в случае с инвариантом кажется, что демонстрируется «ловкость рук и никакого мошенства», то с рекурсией обычно вначале возникает полное ощущение того, что дурят нашего брата — и все тут. Не понятно, ни как вообще рекурсивный алгоритм может работать, ни что происходит при его выполнении.

По сравнению с учебником я переставлю материал и начну с содержательного примера. В учебнике сначала говорится, что надо быть осторожным, а потом уже, что это полезно. Я вначале продемонстрирую полезность, а потом уже покажу, что надо быть осторожным.

Задача, которая рассматривается в учебнике, выглядит так: где-то на клетчатом поле внутри некоторой области, ограниченной закрашенными клетками, стоит Робот (рис. 8). Робот стоит в незакрашенной клетке, а «граница» области (т. е. закрашенные клетки) может быть устроена, как угодно сложно. Известно только, что вся область у нас ограничена, т. е. по незакрашенным клеткам уйти «на бесконечность» нельзя. Задача состоит в том, чтобы всю эту незакрашенную область — и только ее — закрасить.

> **МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Эту задачу очень легко проиллюстрировать на любом нормальном компьютере. Практически во всех графических редакторах есть такая возможность: когда на экране уже что-то нарисовано, есть какие-то линии, можно «ткнуть» мышкой в какую-нибудь область между линиями и «залить» ее определенной краской, определенным цветом. Причем «заливка» происходит только до границы области, до линии, как бы сложно она не извивалась на экране.
>
> Это в точности та же задача, только вместо клеток Робота здесь точки растра экрана («пикселы»), а вместо границы из закрашенных клеток — граничная линия из точек экрана, отличающихся по цвету от точки внутри области, в которую мы указали.

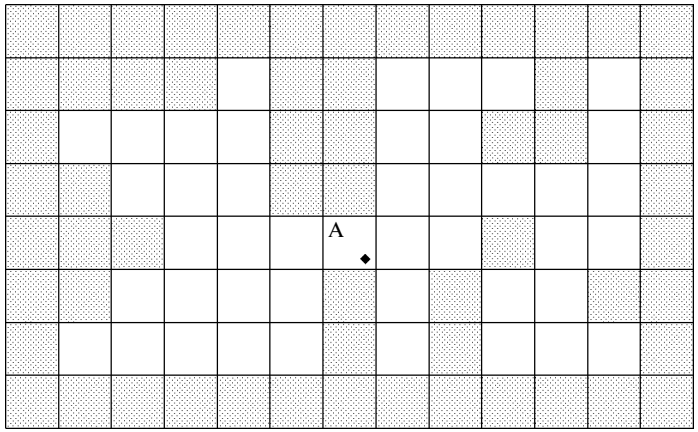


Рис. 8.

> **И ЕЩЕ ОДНО ОТСТУПЛЕНИЕ.** Как всегда, чтобы лучше «прочувствовать», что дает применение метода, полезно сначала попробовать решить эту задачу «просто так», не читая учебник и не разбираясь с рекурсией. ***Если вы впервые читаете это пособие и еще не знакомы с рекурсией — отложите все книги и попробуйте просто решить задачу***. Задайте ее ученикам (например, на дом) прежде, чем вы приступите к изучению метода. Сравнение того, что при этом получится (у тех, кто все-таки умудрится решить задачу), с решением, приведенным в учебнике, — самый сильный аргумент в пользу изучения методов алгоритмизации.

В чем сложность этой задачи? Посмотрите — в положении, изображенном на рис. 8, надо закрасить области и справа, и слева от Робота. Если мы закрасим клетку, в которой стоит Робот, и пойдем, например, направо, то — скорее всего — уже никогда не узнаем, что область слева первоначально входила в закрашиваемую зону, потому что не сможем отличить закрашенную клетку A от клеток границы.

Так же, как и с инвариантом, предлагаемое здесь рекурсивное решение в каком-то смысле «выводится» из доказательства того, что оно будет работать. Поэтому мы начнем с доказательства.

Итак, Робот стоит в некоторой клетке области. Клетки,

которые он должен закрасить, — это клетки, до которых Робот может дойти, если пойдет только по незакрашенным клеткам. На старте — в общем случае — Робот может пойти в одну из четырех сторон (вниз, вверх, вправо, влево), поэтому множество клеток, до которых он может дойти из А по незакрашенным, — это сама клетка А, а также те клетки, до которых можно дойти из четырех клеток, прилегающих к А. Естественно, если эти четыре клетки не закрашены. Если же часть из них закрашена, то надо рассматривать только незакрашенные. Например, на рис. 8 таких клеток две — слева и справа от А.

Таким образом, задача закрашивания всех клеток, до которых можно дойти из А (по незакрашенным клеткам), сводится к подзадачам закрашивания всех клеток, до которых можно дойти из соседних с А незакрашенных клеток. Если для решения подзадач использовать вызов вспомогательного алгоритма «закрасить область», закрашивающего все клетки, до которых можно дойти от той клетки, где стоит Робот, то получится алгоритм А89 со с. 135 учебника):

```
алг закрасить область
  дано |на поле Робота стен нет; число клеток
       |в которые Робот может попасть из исходного
       |положения, двигаясь только по незакрашенным
       |клеткам, конечно
  надо |закрашены все клетки, в которые Робот мог
       |попасть из исходного положения по
       |незакрашенным клеткам; Робот в исходном
       |положении
нач
| если клетка не закрашена то
|  закрасить
|  вверх; закрасить область; вниз
|  вправо; закрасить область; влево
|  вниз; закрасить область; вверх
|  влево; закрасить область; вправо
| все
кон
```

Я думаю вы уже заметили, что этот алгоритм вызывает сам себя (именно это и называется «рекурсией» или «рекурсивным вызовом» алгоритма). Что алгоритм будет работать, доказывается приведенным выше рассуждением: все клетки, которые надо закрасить, — это клетка А и клетки незакрашенных областей, вплотную примыкающих к А с одного из четырех направлений.

Поскольку алгоритм вызывает сам себя, надо отдельно

показывать и доказывать, что он не будет это делать вечно (не «зациклится»). Это можно сделать так: алгоритм сначала закрашивает незакрашенную клетку, а уж потом вызывает сам себя, поэтому перед каждым таким вызовом число незакрашенных клеток будет уменьшаться. В начале число незакрашенных клеток, которые надо закрасить, ограничено, следовательно рано или поздно вызовы алгоритмом самого себя прекратятся и, значит, выполнение алгоритма рано или поздно закончится.

Мы опять написали алгоритм, как-то логически показали, доказали, что он будет работать правильно и что все окончится. А как будет при этом Робот двигаться, куда он пойдет и в каком порядке будет красить клетки, — этого мы пока не знаем. Согласитесь, что до § 16 нам и в голову не могло прийти, что можно составлять алгоритм, абсолютно не разбираясь с тем, что он будет ***делать***, что будет происходить при его ***выполнении***, как и куда будет двигаться Робот.

Итак, «рекурсия», как понятие, — это вызов алгоритмом самого себя. Использование таких вызовов при составлении алгоритмов составляет существо рекурсии, как метода алгоритмизации. Сложность здесь обычно только в одном — в восприятии. От этих вызовов алгоритмом самого себя голова может пойти кругом и возникнет либо ощущение порочного круга, либо просто непонимание.

Но давайте вспомним, — раньше были основные и вспомогательные алгоритмы. Основной алгоритм вызывает вспомогательный алгоритм — это понятно. При вызове алгоритма в памяти отводится место, начинается выполнение алгоритма и т. д.

В случае рекурсии происходит все то же самое. При вызове из алгоритма «закрасить область» самого себя в памяти ЭВМ отводится место под ***еще один*** алгоритм «закрасить область» (принято говорить под «второй», «третий» и т. д. ***экземпляр*** алгоритма) — см. рис. 9.

То есть ЭВМ работает, как всегда, — при каждом вызове алгоритма «закрасить область», сколько бы раз он перед этим уже не вызывался, в памяти отводится ***новое*** место и начинается выполнение вызванного алгоритма (например, 25-го экземпляра), а выполнение алгоритма, из которого произошел вызов (в нашем примере — выполнение 24-го экземпляра), приостанавливается на команде вызова до полного окончания выполнения вызванного — теперь надо говорить «экземпляра» — алгоритма. Происходят такие вызовы, вызовы, вызовы и вызовы все новых экземпляров алгоритма.

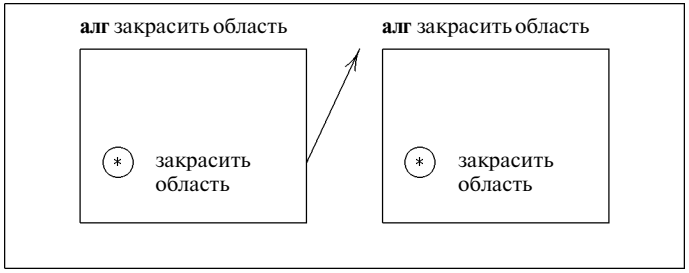


Рис. 9.

Можно представить себе каждый экземпляр алгоритма написанным на отдельном листке и кипу таких листков, с одним и тем же текстом — текстом алгоритма «закрасить область». Тогда работу ЭВМ можно описать так. При выполнении очередного вызова очередного экземпляра алгоритма ЭВМ выполняет команды по соответствующему листку, пока не доходит до вызова «закрасить область». В этот момент ЭВМ «ставит галочку» напротив этого вызова, «откладывает листок», берет ***новый*** листок с тем же текстом и начинает его выполнять с самого начала. Образуется такая стопка отложенных листков (в информатике она называется «стеком», потому что каждый следующий листок кладется сверху предыдущего). Если в какой-то момент очередной экземпляр алгоритма, очередной листок, удастся выполнить до конца (например, если клетка окажется закрашенной, — тогда наш алгоритм ничего не делает), то в таком случае этот листок выбрасывается (т. е. этот экземпляр алгоритма стирается из памяти ЭВМ), из стопки берется последний отложенный листок и ЭВМ продолжает его выполнение с места, отмеченного галочкой.

Это очень важно — представлять себе, ***как*** ЭВМ, выполняет рекурсивные вызовы алгоритмов, чтобы понимать, как вообще работает рекурсия и что при этом происходит.

И здесь я плавно перейду ко второй части — рекурсия бывает не только полезна, но и опасна. Дело в том, что обычно, как и в примере выше, при составлении рекурсивного алгоритма у нас возникает «доказательство» его правильности и того, что он рано или поздно закончится. Но при этом не возникает никакого представления о том, какие действия и как будут выполняться. В этом и состоит опасность. Ведь формальной правильности и завершаемости недостаточно. Что значит «выполнение алгоритма рано или поздно закончится»? — ведь алгоритм может оказаться ужасающе неэффективным. Вряд ли нас устроит доказательство того, что алгоритм «когда-нибудь» кончится, если реально для его выполнения понадобятся сотни лет работы на самых совершенных компьютерах.

Для демонстрации этой опасности, для демонстрации того, что очень простой рекурсивный алгоритм может оказаться ужасающе неэффективным, в учебнике используется рекурсивный алгоритм А88, вычисляющий (уже который раз!) n-ый элемент последовательности Фибоначчи (с. 134 учебника). Этот алгоритм записан в точности по определению последовательности Фибоначчи: $\Phi(n) = \Phi(n-1) + \Phi(n-2)$. Именно поэтому он гораздо проще всех наших предыдущих алгоритмов для последовательности Фибоначчи. Но как он будет выполняться? Пусть, для примера, мы стали с его помощью вычислять Ф(45). Это значит — смотрите на текст алгоритма, — что выполнятся вызовы Ф(44) и Ф(43), т. е. наш алгоритм Ф будет дважды вызван рекурсивно, но уже с другими аргументами. Соответственно, алгоритм Ф(45) будет «отложен» и начнет выполняться, скажем, Ф(44). При выполнении Ф(44) надо будет вычислить (рекурсивно вызвать) Ф(43) и Ф(42). При вычислении Ф(43) — вызвать Ф(42) и Ф(41). И т. д. (рис. 10).

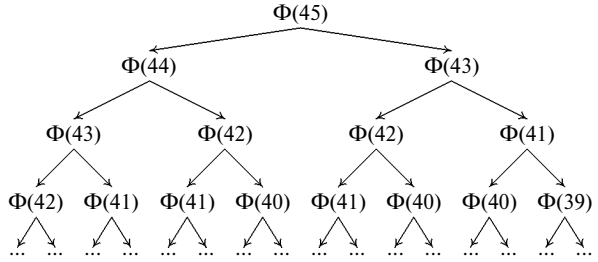


Рис. 10.

Вы видите, что Ф(42) вызывается уже три раза (в разных частях картинки). Если рисовать картинку дальше, то к моменту, когда мы дойдем до Ф(1) и Ф(2) (которые вычисляются явно, т. е. не порождают новых рекурсивных вызовов), у нас на картинке будет уже больше миллиарда разных Ф(...).

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Вы помните старинную задачу про мудреца, который попросил на первую клетку шахматной доски положить одно зерно, на следующую — два, на следующую — четыре и т. д. На

каждую следующую — в два раза больше, чем на предыдущую. Получаются степени 2 (на последнюю клетку надо положить 2^{63}), а степени двойки чрезвычайно быстро растут.

У нас почти то же самое — каждый следующий уровень на рис. 10 содержит вдвое больше вызовов Ф, чем предыдущий (более высокий) уровень.

Таким образом, если мы начнем вычислять по приведенному рекурсивному алгоритму Ф(45), то будет выполнено свыше миллиарда операций сложения. А ведь в рекуррентном алгоритме А82 этих операций было всего 45. Значит, рекурсивный алгоритм будет выполняться ***в несколько десятков миллионов раз*** медленнее! И это — не считая затрат времени на постоянное «откладывание» алгоритмов, затрат памяти ЭВМ при отведении места под вызываемые алгоритмы и т. п.

Вы видите: такая простейшая задача, такой простой алгоритм, — и какая жуткая неэффективность. Поэтому, повторю еще раз, рекурсией надо пользоваться с осторожностью. Недостаточно просто написать рекурсивный алгоритм. Надо представлять себе, как он будет выполняться, — не возникнет ли в процессе выполнения эффекта удвоения (или утроения, или учетверения) объема работы, объема вычислений.

Если рекурсивный алгоритм вызывает сам себя только один раз (в одном месте), то, как правило, это не опасно. Аналогичный рис. 10 рисунок будет содержать линейную цепочку вызовов — и никакого удвоения или утроения не произойдет. Но если алгоритм вызывает себя несколько раз (алгоритм А88 вызывает себя дважды, алгоритм А89 — четыре раза), то это потенциально очень опасно. И надо отдельно разбираться и понимать, как алгоритм будет выполняться и не начнет ли он помногу раз вычислять одно и то же.

Итак, рекурсия — это несколько экзотический, потенциально опасный, но иногда полезный метод алгоритмизации. Существуют задачи, которые рекурсивно решить легко, а без рекурсии — очень трудно. Закраска области (на поле Робота или на экране) — одна из таких задач. Есть и другие. Попробуйте, например, составить алгоритм обхода конем шахматной доски (конь должен побывать в каждой клетке ровно по одному разу) без использования и с использованием рекурсии — и, как говорится, — «почувствуйте разницу».

В целом рекурсия обычно применима и полезна, когда после первого хода, или шага, или еще чего-то мы сводим исходную задачу к другой ***точно такой же*** задаче (подзадаче), но с другими данными. Именно это мы проделали в задаче закраски области. Задачи, которые легко свести к другим точно таким же, но с другими данными, как правило, легко и просто записываются рекурсивно. Но, повторю еще раз, простота записи не означает в данном случае простоты и эффективности работы алгоритма. Это надо смотреть отдельно. Поэтому, хотя рекурсия и позволяет очень легко записывать такие алгоритмы, использовать ее надо аккуратно и с осторожностью.

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. В заключение я еще скажу несколько слов о нашем личном отношении к рекурсии. Мы включили этот метод в учебник, потому что существует такое течение в информатике, такая достаточно большая группа ученых, которые считают рекурсию основным и главным методом информатики. Считают, что даже циклы надо реализовывать через рекурсивные вызовы алгоритмов, а конструкций циклов в языке быть не должно.

Базой для этого течения является математическая теория, называемая аппаратом рекурсивных функций, в которой доказывается, что таким образом можно подсчитать все, что угодно.

Соответственно, имеются языки программирования, построенные на том, что именно так и надо работать, языки, в которых нет циклов, в которых циклы имитируются через рекурсию. Кстати небезызвестный язык ЛОГО (если отвлечься от черепашьей графики и посмотреть на сам язык программирования) — один из таких языков. Цикла **пока** в нем уже нет, а если мы хотим повторять какие-то действия, пока будет выполнено некоторое условие, то это надо делать через **если** и рекурсивные вызовы.

Поскольку это течение, считающее рекурсию одной из самых важных и базовых вещей в информатике, достаточно представительно, особенно среди математиков, мы решили этот метод в учебнике изложить — хотя бы со звездочкой.

Кроме того, когда мы обсуждали включение рекурсии в школьный курс с Андреем Петровичем Ершовым, то он сказал, что на его взгляд это очень важная составляющая

алгоритмической культуры и в каком-то виде рекурсия в школьном курсе должна быть представлена обязательно. И мы, конечно, постарались это мнение учесть.

Но, повторю, сами мы считаем рекурсию просто одним из методов алгоритмизации — методом безусловно полезным, хотя и с достаточно ограниченной областью применения. И полагаем, что пользоваться им надо осторожно и аккуратно.

На этом § 16, а с ним и вся первая глава, заканчиваются. На самом деле, если вы теперь посмотрите на первую главу целиком, то увидите, что состоит она из двух разных разделов: изложение собственно алгоритмического языка заканчивается § 15, а в § 16 содержится совершенно другой материал — методы алгоритмизации (хотя параграф назван не так, а попроще — «составление циклических алгоритмов»). В реальности это — отдельный раздел информатики, но в учебнике он не выделен.

В § 16 изложены четыре метода алгоритмизации. Эти методы различны и по сложности, и по применимости. На школьном уровне наиболее применим, по-видимому, метод рекуррентных соотношений. В школе (и в математике, и в физике) достаточно много задач, решаемых этим методом. Следующим идет метод однопроходных алгоритмов, применимый к огромному количеству типовых задач информатики. Все эти задачи вычисления максимальных, минимальных и среднеарифметических величин, как и практически любая другая задача, вычисления чего-то по таблице, могут быть решены этим методом — и за один проход. Содержательное применение метода использования инварианта цикла осталось за рамками учебника, как несколько более сложное. А на рекурсию вы можете смотреть, как на достаточно редкий, экзотический метод, применимый к некоторым специфическим задачам (и именно поэтому она поставлена со звездочкой).

Более глубокое и содержательное изложение этих, а также некоторых других методов алгоритмизации можно найти в нашем вузовском учебнике [ПдМ]. Поскольку учитель должен владеть предметом глубже, чем изложено в школьном учебнике, то я вам настоятельно рекомендую потратить свое время и разобраться в методах алгоритмизации более глубоко. Хотя это требует усилий и некоторого уровня математической культуры.

Упражнения на повторение к главе 1

Первый раздел заканчивается так называемыми «упражнениями на повторение» (с. 142 и далее). На самом деле эти «упражнения на повторение» никакими упражнениями на повторение не являются. Это просто набор упражнений, которые можно решать, пройдя всю первую главу. Среди них довольно много задач, взятых из предыдущих учебников Андрея Петровича Ершова, — задач математического плана, где надо что-то вычислить, но произвести при этом не более какого-то заданного числа операций.

Единственное мое общее замечание сводится к тому, что это не есть задачи на повторение — это просто набор задач. Вы можете вполне использовать их в процессе изучения первого раздела, выбирая и давая задачи сильным ученикам.

Кроме того, я хочу обратить ваше внимание на задачи, которые мне кажутся наиболее полезными и важными. Это задачи 30 и 31 на с. 145.

Предположим, у вас в классе сидит какой-нибудь умный мальчик и говорит: «Что мы все какой-то ерундой занимаемся, все Робота гоняем. Когда же мы наконец займемся делом?».

Вы помните, я говорил, что наш курс построен на том, что надо научить решать задачи. Совершенно не важно — с этой точки зрения — умеет ли ученик нажимать на кнопки. (Конечно, если курс компьютерный, то надо уметь это делать, но даже в этом случае умение нажимать на кнопки, знание КуМира, Бейсика, Паскаля, операционной системы не входит в основные ***цели*** курса.)

Посмотрите, сколько мы уже всего разного — и очень содержательного — прошли, а про то, как работать на компьютерах ни слова не сказали. Поэтому для изучения нашего курса чрезвычайно важно создать правильную ценностную ориентацию, чтобы школьник понимал, что умение решать задачи важнее, чем знание тех или иных операторов и скорость нажатия на кнопки. Умение думать значительно важнее знания каких-то технических деталей. Чтобы эту ценностную ориентацию создать, ученикам надо предлагать задачи, достойные их уровня, иначе ничего не получится. Если сильному ученику долго задавать простые задачи вроде «довести Робота до стены», то, конечно, он будет считать это ерундой и игрушками для маленьких.

Поэтому сильным ученикам надо предлагать достойные их задачи — это с одной стороны. А с другой стороны, задачи

должны быть на поле Робота, чтобы показать, что дело не в Роботе и не в формулировке. Надо просто уметь решать задачи. И здесь очень полезны задачи 30 и 31.

Задача 30. На поле Робота имеется область, «обнесенная забором» из стен. Известно, что Робот стоит вплотную у одной из стен «забора». Надо определить, где стоит Робот — внутри огороженного участка или снаружи. Вы видите — это обычная, простая формулировка в стиле всех задач про Робота. Однако сама задача уже вполне достойна сильных учеников.

Чем хороша эта задача? И в чем вообще отличительная черта задач для сильных? В том, что сразу не видно, как эту задачу решать. Глядя на картинку, на формулировку, нельзя сразу сказать, куда и как надо вести Робота, чтобы получить ответ.

На случай, если ученики окажутся недостаточно сильными и не смогут решить эту задачу, и вам понадобится давать им наводящие указания, я вам сейчас в общих чертах обрисую, как ее можно решить. Для этого давайте вообразим себя вместо Робота. Если мы возьмемся левой рукой за стену и пойдем вдоль «забора», все время касаясь его левой рукой и считая, сколько раз и куда (направо или налево) нам придется поворачивать, то при возвращении в исходную клетку «с другой стороны» (после обхода всего «забора») у нас будет либо левых поворотов сделано на 4 (т. е. на один полный оборот) больше, чем правых, либо, наоборот, правых поворотов будет сделано на 4 больше, чем левых. Значит, при таком обходе мы один раз повернемся либо в одном направлении (через левое плечо, левых поворотов больше — при обходе «забора» снаружи), либо в другом (через правое плечо, правых поворотов больше — при обходе «забора» изнутри). Поэтому, посмотрев каких поворотов было сделано больше, можно сказать снаружи или изнутри огороженного участка мы находимся.

Конечно, это только идея решения, которую еще надо превратить в алгоритм, — ведь у Робота нет ни рук, ни лица, он не умеет поворачиваться вокруг собственной оси ни налево, ни направо. Значит, все эти понятия придется имитировать с помощью величин алгоритмического языка, например, завести величину «ориентация» (т. е. куда направлено воображаемое «лицо» Робота) (кстати, для задания ориентации на плоскости можно использовать аналогию с часами: 0 — направление вверх, 3 — вправо, 6 — вниз, 9 — влево).

Тогда поворот направо (по часовой стрелке) запишется как «ориентация := mod(ориентация + 3,12)», а поворот налево как «ориентация := mod(ориентация – 3,12)».

Как, например, узнать, что Робот вернулся в исходную клетку? Можно считать по дороге суммарное смещение Робота от начальной клетки по горизонтали (по оси X) и по вертикали (по оси Y). При шаге вправо увеличивать смещение по X ($x := x + 1$), а при шаге влево — уменьшать ($x := x - 1$). Аналогично для шагов вверх и вниз ($y := y + 1$ и $y := y - 1$). И если мы в какой-нибудь момент обнаружим, что $x = 0$ и $y = 0$, то это и будет означать, что Робот в исходной клетке.

Придется также составить вспомогательный алгоритм «шаг вдоль стены», который при повороте забора влево заставит Робота сделать два шага (рис. 11), при повороте забора вправо (конечное положение Робота на рис. 11) просто «повернет» Робота, оставив его на той же клетке. (т. е. изменит значение величины «ориентация»). Другими словами, «шаг вдоль стены» должен так изменить ориентацию и положение Робота, чтобы воображаемая «левая рука» Робота сместилась к следующему одноклеточному участку стены:

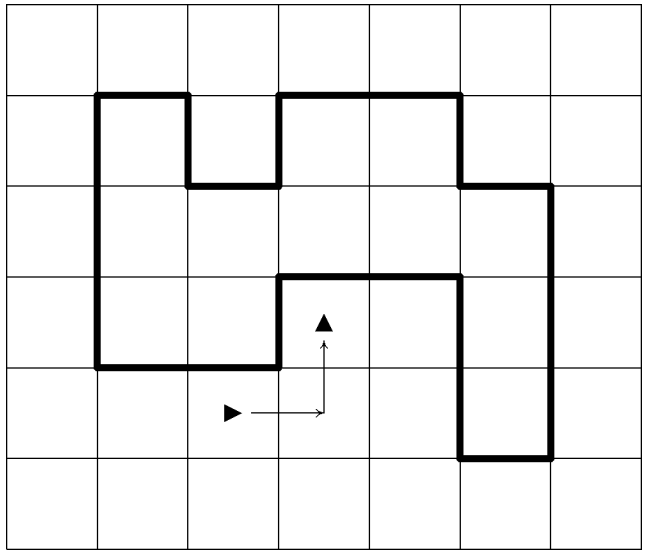


Рис. 11. Шаг «вдоль стены» при повороте «забора» влево

Задача 31 существенно сложнее она недаром помечена звездочкой. Мы обычно называем ее задачей «о налоговом инспекторе». Налоговый инспектор Робот подошел к земельному участку, огороженному неприступной стеной. Чтобы определить, правильно ли платится налог за землю, Робот должен подсчитать площадь земельного участка. Но поскольку участок частный, а частная собственность неприкосновенна (это все не у нас, конечно, происходит, а в каком-то воображаемом правовом государстве), заходить внутрь участка даже налоговый инспектор права не имеет. Поэтому надо, как-то бродя снаружи вдоль стен участка, умудриться подсчитать площадь участка.

Эта задача намного сложнее, и даже идею ее решения, как это было в предыдущей задаче, я вам кратко изложить не могу. Точнее, одну идею, идею одного из решений, я могу изложить — «надо взять интеграл по контуру и используя формулу Грина, получить площадь». Но эта идея апеллирует к теоремам и понятиям дифференциальной геометрии и потому вряд ли будет полезна в обычной школе. И хотя на основе этой идеи можно написать короткий алгоритм и даже объяснить, почему он дает правильную формулу, придумать такой алгоритм чрезвычайно трудно.

Если эту задачу задать даже сильным ученикам, то с большой вероятностью они будут мучиться, но вряд ли сами ее правильно решат. И это позволяет учителю сказать: «Видишь, тот же Робот и клетчатое поле, а задача не решается. Значит, суть не в Роботе, а в том, умеешь ты решать задачи или не умеешь».

И еще, и еще раз повторю — в нашем курсе очень важна ценностная ориентация. Умение решать задачи должно быть поставлено выше технических знаний и навыков. Если этого не сделать, если школьники сидят и ждут, когда им разрешат сыграть на компьютере, когда им расскажут про операционную или файловую систему и т. п., то у них возникнет проблема «обманутых ожиданий». Проблема несоответствия содержания курса их ожиданиям. Желательно с самого начала это погасить. Сказать, что курс содержательный, что мы будем учиться решать задачи. А чтобы не тратить время на изучение особенностей Бейсика или Паскаля, решать будем задачи про Робота на школьном алгоритмическом языке (на компьютерах — в системе КуМир).

Учитель, конечно, может преподавать и без Робота, и школьного языка. Но тогда нужно будет их чем-то заменить, причем заменить так, чтобы не потерять алгоритмическую сложность решаемых задач, чтобы решение задач не оказалось заменено на изучение чего-нибудь, типа многочисленных замечательных возможностей команды PRINT в Бейсике.

В заключение хочу еще обратить ваше внимание на задачи 32, 33 и 34. Соответствующие исполнители реализованы в системе КуМир, и вы можете прямо на компьютере решать задачи по управлению этими исполнителями. Наибольший интерес у школьников обычно вызывает «Двуног», состоящий из — естественно — двух ног и тяжелой головы на очень длинной шее. Когда голова наклонена, Двуног начинает медленно падать. В этот момент можно, скажем, выставить ногу, упасть на нее, распрямить шею, выставить вперед другую ногу, наклонить шею и опять начать падать. Типовая задача — «научить Двунога ходить», т. е. составить соответствующий алгоритм, при выполнении которого Двуног пойдет. Сначала по ровному месту, потом по лестнице, потом в низкой пещере, где нельзя полностью распрямиться, и т. п. Этот исполнитель полезен при прохождении темы вспомогательных алгоритмов (т. е. еще до конструкций циклов и ветвления).

С другой стороны, по образцу этих задач можно предложить ученикам придумать своих исполнителей, посмотреть, быть может они придумают что-то и интересное, и полезное.

Некоторые из таких исполнителей изображены на цветной вклейке в конце учебника: «Двуног», забирающийся по лестнице; «Вездеход» с локатором; «Строитель» — тот же Робот, но работающий на клетчатой площадке в пространстве и умеющий перемещать «строительные кубики»; и т. п.

Наконец, вы должны понимать, что мы ввели в учебник Робота и Чертежника потому, что с их помощью мы показали все, что хотели. Но, конечно, Робот — это не мировая константа, не число π. Исполнителей можно придумывать разных и много. Вы, наверное, знакомы с исполнителем «Муравей», который катает по клетчатому полю кубики с буквами и которого использовал при преподавании информатики Г.А. Звенигородский. В системе «Роботландия» [Первин] этих исполнителей достаточно много и разных. В свердловском учебнике [Гейн] есть Чертежник-Вычислитель, чем-то похожий на Робота с хвостом. В нашем вузовском учебнике использовались «Редактор слова», «Путник» (похож на Робота, но имеет ориентацию) и целый ряд других исполни-

телей. Я уж не говорю о всем известной Черепашке в Лого. Применение исполнителей как методический прием сейчас можно считать вполне распространенным и освоенным. Поэтому, если вы в какой-то момент будете считать, что Робот в этом классе уже надоел, то вы можете сменить исполнителя, заменить его, скажем, на Путника, у которого есть ориентация. Замечу, что при этом многие задачи типа обхода препятствия можно будет записать, не повторяя четыре раза одно и то же с шагами в разные стороны («вправо», «вниз», «вверх», «влево»), а виде цикла 4 **раз** идти «вперед» и потом повернуться. Примеры вы можете найти в [ПдМ]. Алгоритмы при этом станут более компактными. Хотя сам Путник для восприятия несколько сложнее Робота.

И последнее замечание. Я очень вам рекомендую следующую задачу (ее нет в списке «задач на повторение»). Рассмотрим какой-нибудь алгоритм управления Роботом, например, алгоритмы А35–А37 из п. 9.14. Можно выбрать также и любые другие алгоритмы. Новая задача состоит в переделке этих алгоритмов таким образом, чтобы траектория движения Робота в ходе исполнения алгоритмов оказалась запомненной в виде литерной величины (по одной букве на каждое перемещение Робота). Кроме того, необходимо составить алгоритм, который, используя эту литерную величину, без каких-либо проверок вернет Робота обратно и повторит его движение вперед. Тогда при изучении материала про компиляцию, интерпретацию и кодирование алгоритмов из § 20–21 можно будет сослаться на решение этой задачи, как на что-то уже пройденное, и облегчить восприятие нового материала.

Лекция 9.

Мы переходим к главе 2, которая называется «УСТРОЙСТВО ЭВМ» и состоит из четырех параграфов.

§ 17. Физические основы ЭВМ

Здесь преследуются две основные цели — показать, как, с помощью каких физических процессов, можно (а) обрабатывать информацию и (б) ее хранить, т. е. показать, как ***в принципе*** могут работать процессор (обработка информации, действия) и память (хранение информации, объекты). Показать на чисто физическом уровне.

Естественно, показывается это на самых простых примерах хранения и обработки одного ***бита*** информации. Например, как собрать физическую схему, которая инвертирует бит (из 0 на входе делает 1 на выходе и наоборот). И как собрать физическую схему, которая обеспечит запоминание и хранение одного бита информации.

Этот материал более или менее традиционен. В старых учебниках применялись переключатели (электромагнитные реле), из которых, собственно, и были собраны самые первые ЭВМ. Мы использовали в качестве базового элемента более современные МОП-транзисторы (описав, как такой транзистор работает, но не разбирая, как именно его делают).

Если вы хотите объяснить устройство МОП-транзистора, то имеется замечательная книга «Знакомьтесь: компьютер» [ЗК]. Эта книга замечательна по двум параметрам. Во-первых, она художественно сделана и легко читается, содержит массу разных занимательных историй. Ее можно просто с удовольствием читать на досуге, на ночь, лежа в кровати, и т. д. Во-вторых, из подобного рода занимательных книжек эта — единственная известная мне книга, в которой все изложено очень честно и которая полностью удовлетворяет нашему критерию «настоящести». Здесь нигде «не выплеснут ребенок». Все, что излагается, излагается подробно, содержательно и хорошо.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Давайте я вам сразу расскажу про эту книжку все, что думаю, хотя остальное и не имеет отношения к данному параграфу. Когда мы только начинали писать учебник, у нас была идея

сделать на нижней трети каждой страницы «подвальчики» с разными занимательными историями, интересными примерами, неформальными алгоритмами, шутками и пр. Чтобы этот материал, с одной стороны, тоже был, а с другой — чтобы он был явно отделен от изучаемого содержания курса.

Это — по разным причинам — не получилось. И в наших заготовках осталось довольно много примеров, которые мы хотели привести. Например, про самолет с обратной стреловидностью крыла (с крыльями, скошенными к носу, а не к хвосту). Такой самолет аэродинамически неустойчив, и человек без компьютера управлять им не может. Без компьютера такой самолет все равно, что обычный самолет без рулей: летит куда хочет, а то и просто валится. Были и другие интересные примеры, связанные с применениями компьютеров. А потом появилась книжка [ЗК], и мы с удивлением обнаружили в ней практически все примеры, которые хотели привести. И даже с изложением содержания применения компьютеров, например, схем управления указанным выше самолетом.

Я очень настоятельно рекомендую вам эту книгу — просто как необходимое дополнение к учебнику, чрезвычайно высококачественное, отвечающее всем требованиям нашего курса, плюс очень красочно и хорошо изданное, и весьма занимательное.

Возвратимся к теме параграфа: в [ЗК] вы можете найти и описание (с картинками), как работает МОП-транзистор (вплоть до электронной и дырочной проводимости), и описание процесса производства транзисторов (приведены фотографические пластинки, показано, что на что напыляют, очень красивые — по-американски — цветные картинки, цветные стрелки, в общем, замечательно все нарисовано). И я вас отсылаю к этой книге, если вы захотите показать, как сделать МОП-транзистор и почему он работает.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Вы, впрочем, можете взять практически любой учебник по электротехнике (электронике) и найти там детальное описание работы электромагнитного реле — по сути обычного выключателя, который включается и выключается также с помощью электричества. Такое реле даже можно собрать и продемонстрировать в классе (желательно на уроках физики).

Мы же (в учебнике) считали МОП–транзистор «базовым кирпичиком» и только описали, как он работает, не объясняя, ни как его изготовить, ни почему он так работает. Школьнику, мы полагаем, интуитивно ясно, что столь простое устройство физически как-то изготовить можно. И сконцентрировали все внимание в этом параграфе на двух важнейших вещах: показать, как компьютер может хотя бы один бит информации как-то обработать и как он, в принципе, может хотя бы один бит хранить.

Если отвлечься от замены реле на МОП-транзисторы, то этот параграф учебника вполне традиционен, соответствующий материал можно найти в любой книжке, посвященной устройству ЭВМ. В частности, в журналах «Квант» в свое время было достаточно много статей о том, например, как собрать сумматор из подобного рода переключателей. Материал учебника достаточно хорошо дополняется материалом в методпособии [Авербух], где, в частности, изложено, как из вентилей собрать сумматор и полусумматор (в учебнике этого нет), т. е. электронную схему, которая может ***складывать*** двоичные числа. Разбор работы такого сумматора, конечно, полностью демистифицирует, дегероизирует ЭВМ, показывает, что ничего «мистического», непознаваемого в ЭВМ нет. Как бы ни была сложна ЭВМ в целом, она «собрана» из частей, похожих на сумматор. И можно понять, как она в принципе работает.

Поскольку материал главы в целом достаточно традиционен, я только опишу основные цели и скажу, что в каком параграфе. А в остальном, — материал общепринятый, литературы много, мы почти ничем не отличаемся от других учебников, книжек, журнала «Квант» и прочего, разве что, может быть, последовательностью изложения материала.

Основные цели § 17

Первое — показать, что электронная схема может, в принципе, производить формальное преобразование информации (как в игре в «черные ящики»). Продемонстрировать на чисто физическом уровне, на уровне напряжений на проводах, что соответствующие биты информации могут как-то обрабатываться, что соединяя по-разному вентили, транзисторы и проч., мы можем получить разные схемы преобразования информации. Показать, как в принципе

устроена обработка информации внутри компьютера, как и за счет чего работает процессор.

Второе — показать, что электронная схема может хранить информацию. Продемонстрировать схему устройства для хранения одного бита информации — триггер (с. 150 учебника), т. е. показать, как можно собрать триггер из вентилей. И затем продемонстрировать, что если определенным образом подать напряжение, а потом снять, то триггер запомнит и будет хранить 1, если же напряжение подать подругому, то триггер запомнит и будет хранить 0. А после этого можно будет узнать («прочесть»), какой именно бит (1 или 0) хранится в триггере.

Части параграфа про «Взаимодействие процессора и памяти», «Поколения ЭВМ» и «Изготовление микросхем» менее важны. Хотя, например, понятия разрядности процессора, шины адреса и шины данных могут пролить свет на некоторые — в противном случае, загадочные — особенности некоторых компьютеров. Я, однако, хочу обратить ваше внимание только на одну мелкую, но очень важную деталь п. 17.11 «Изготовление микросхем», а именно: что микросхемы изготовляются не поодиночке; — за счет процесса, содержащего десяток операций и напоминающего фотопечать (см. [ЗК]) изготавливаются одновременно миллионы микросхем. Вот почему при совершенствовании технологии, компьютеры становятся все более мощными и в то же время все более дешевыми.

Вот и все про § 17. Теперь вспомним о «черепахе» курса — усвоить что бы то ни было можно только через работу. Поэтому для освоения материала параграфа ученики должны решать задачи и упражнения: составлять схемы из вентилей, реализующие те или иные операции обработки информации. В принципе, задачи такого типа — это логические задачи с графическим изображением результата; решать их можно в любое время, поэтому допускается смещать материал параграфа (да и главы в целом) в широких пределах вперед или назад по курсу. Например, с целью смены характера проходимого материала, создания «перерыва» в изучении алгоритмизации.

§ 18. Команды и основной алгоритм работы процессора (программирование в кодах)

§ 18 посвящен описанию работы процессора на уровне элементарных машинных команд. Это то, что обычно называют «программированием в кодах». Этот материал имелся и в самых первых учебниках А.П. Ершова (в начале второй части). Только там были и регистры, и виды адресации, и много всего остального. Мы все это убрали и на базе той же машины (PDP-11 или УК НЦ — с той же системой команд) оставили только простейшие команды, понятие счетчика команд и описание, как, за счет чего, обеспечивается автоматизм работы процессора, как он переходит от выполнения одной команды к другой и как могут быть реализованы алгоритмические конструкции, например, цикл **пока**.

По сравнению с первыми учебниками [Ершов] оставлены только 16 команд вида «добавить содержимое по адресу 100 к содержимому по адресу 94» (с. 156 учебника). Никаких регистров, никаких видов адресации, просто содержимое по некоторому адресу памяти берется и добавляется к другому содержимому другого адреса памяти. Все остальное — как технические детали — убрано.

Из всего, связанного с регистрами, остался только счетчик команд, содержащий адрес команды, которая будет выполняться следующей, и биты-условия в слове состояния процессора (ССП), хранящие информацию о том, каким был результат предыдущей команды — отрицательным, положительным или нулем.

Материал этого параграфа является упрощением материала из учебников [Ершов]. Детальное (более глубокое) изложение и описание есть в любой технической литературе по серии машин PDP-11 (к ней относятся и наши УК НЦ, ДВК, БК и СМ-4). При желании материал параграфа можно заменить на аналогичный материал на базе другого компьютера. Выбор УК НЦ обусловлен тем, что, с одной стороны, это ***реальный*** компьютер; а с другой — его система команд очень проста, стройна, логична и легко может быть изложена в школе. Команды — настоящие, машины — настоящие, приведены простейшие команды, но из реального, настоящего компьютера. Здесь нет ничего придуманного.

При изложении параграфа акценты надо сделать на две основные вещи. Первое — основной алгоритм работы про-

цессора в п. 18.2, обеспечивающий автоматизм работы процессора и переход от выполнения одной команды к выполнению другой. Этот алгоритм, правда, сразу написан в виде **нц**—**кц**, без всяких объяснений, почему мы так пишем. Я хочу подчеркнуть, что здесь это не алгоритмическая конструкция, а просто форма записи. Процессор в этом цикле работает бесконечно (пока не встретит команду «Стоп» и не остановится).

И второе — машинная реализация управляющих конструкций (**пока**, **если**) за счет команд перехода (пп. 18.5 и 18.6).

Поскольку здесь все традиционно, я подробнее останавливаться ни на чем не буду. Как и всюду в учебнике, усвоение материала предполагается через решение учениками задач и упражнений, приведенных в конце параграфа.

Итак, § 17 — это физический уровень, уровень транзисторов, схем, вентилей, высоких, низких напряжений. § 18 — это следующий уровень, уровень машинных команд и программ в кодах.

§ 19. Составные части ЭВМ и взаимодействие их через магистраль

Параграф называется «УСТРОЙСТВА ВВОДА/ВЫВОДА ИНФОРМАЦИИ», потому что традиционно компьютер делится на процессор, память и все остальное. И это «все остальное» традиционно называется устройствами ввода/вывода информации.

Начало параграфа также вполне традиционно. Говорится, что есть такое устройство «монитор», по нему бегает луч, и точки экрана вспыхивают или гаснут так, что на экране образуется картинка. Есть «клавиатура», «принтер», «дисковод», куда вставляются магнитные диски.

Кстати, мы с удивлением обнаружили, что некоторые ученики считают магнитный диск квадратным (из-за внешней упаковки) и поэтому — специально — на цветной вклейке поместили фотографию дискеты с разорванной обложкой (фото 26). Я рекомендую вам показать эту фотографию ученикам, а еще лучше — вскрыть какую-нибудь старую и ненужную дискету и использовать ее в качестве наглядного пособия.

Также более или менее традиционным является и раздел «Взаимодействие основных частей ЭВМ. Магистраль» (п. 19.5, с. 162). В определенной степени это — повторение того, что было в учебнике [Ершов]. Единственное, на что здесь следует обратить внимание, – на управление устройствами ввода/вывода через команды записи и чтения по специальным адресам — адресам этих устройств. Если нужно что-то напечатать, то в машинных командах это выглядит, просто как запись какой-то информации по специальному адресу. Когда такая команда — «записать по такому-то адресу число» — выполняется, т. е. на проводах магистрали появляются соответствующие высокие или низкие уровни напряжения, все устройства ЭВМ, включая память, анализируют этот адрес. Если указан адрес обычной памяти, то она — физическое устройство «память» — срабатывает и записывает число по соответствующему адресу. Если же, например, указан специальный адрес, приписанный принтеру, то память не срабатывает никак, а физическое устройство «принтер», обнаружив «свой» адрес, воспринимает число как команду принтера и, соответственно, что-то печатает. Аналогично, физическое устройство «терминал» («мо-

нитор»), обнаружив свой адрес, выполняет свою команду, например, выводит какой-то символ на экран, и т. д.

Одна и та же команда — «записать по такому-то адресу число» в зависимости от адреса может либо действительно записать число в память, либо (если адрес принтера) что-то напечатать или изменить параметры печати, либо (если адрес монитора) вывести что-то на экран, либо (если адрес клавиатуры) перевести клавиатуру в какой-то иной режим и т. п. Машинная команда одна и та же, а смысл и результат выполнения разный.

Дальше (с п. 19.6) начинаются особенности, связанные именно с нашим курсом и учебником, а именно, объяснение работы устройств ввода/вывода происходит с привлечением понятия «исполнитель». Вспомните схему «программного управления» (см. с. 16): человек пишет алгоритм, отдает его ЭВМ, а ЭВМ исполняет алгоритм и управляет исполнителями. Под словом «ЭВМ» в этой схеме понимается собственно ЭВМ — процессор и память. В рамках такой схемы все устройства ввода/вывода — это просто исполнители, которыми управляет ЭВМ. Исполнителями могут быть Робот, Чертежник, клавиатура, дисковод, принтер и пр.

Мы составляли алгоритмы на примерах управления Роботом. Но управление экраном, клавиатурой, принтером устроено аналогично и мало чем отличается от управления Роботом. Другая система команд, другие понятия, другие действия, но стиль взаимодействия тот же самый: мы пишем алгоритм, при выполнении которого ЭВМ отдаст необходимые команды соответствующим исполнителям и заставит их выполнить нужные действия.

Поэтому здесь чрезвычайно важно — и я хочу особо обратить на это ваше внимание — концептуальное единство построения нашего курса с точки зрения понятия «исполнителя». Важно, что все многообразие разных устройств ЭВМ — просто разные исполнители со своими системами команд. Это дает нам возможность изучать устройства не внешне (какого размера и цвета соответствующее устройство и как оно выглядит), а по содержанию (какова система команд исполнителя «Экран»? как им можно управлять из ЭВМ? и т. п.). И понимать, что управление этими исполнителями ничуть не сложнее, чем управление Роботом.

В параграфе имеется два очень важных момента для формирования информационной культуры, для понимания об-

щей картины мира, в котором появились и используются компьютеры. Это —

1) управление исполнителями «Экран», «Клавиатура» и др. непосредственно из алгоритмического языка;
2) «физическое» подключение Робота к ЭВМ (в упражнениях).

Кратко остановимся на этих моментах.

Мы сказали, что клавиатура, экран, принтер — просто исполнители, значит, с ними, можно работать так, как с Роботом, посылая им команды, управляя ими из алгоритмического языка. Подобно тому, как мы командовали Роботу «вправо», «влево», «вверх», «вниз», мы можем скомандовать экрану «вывести символ» или клавиатуре «ввести символ». И в параграфе приводятся некоторые команды исполнителей «Экран» и «Клавиатура» (пп. 19.7 и 19.8), — команды алгоритмического языка, которые позволяют работать с экраном и клавиатурой: «цвет», «точка», «позиция», «вывести символ» и т. д. Это — действительно реальные команды, и в системе «КуМир» можно писать алгоритмы с использованием этих команд, решать задачи типа «написать алгоритм, который очищает экран целиком» (помните «черепаху» — «понимание через делание»?). Или «написать алгоритм, который определяет, нажимает ли человек стрелки вправо или влево на клавиатуре и рисует ли на экране линию в соответствующем направлении». Последняя задача — это простейший прототип графического редактора, алгоритм буквально всего из 15 строчек на алгоритмическом языке: ввод символа от человека и вызов вспомогательного алгоритма, рисующего линию. Зато какое понимание сути работы настоящих — весьма сложных — графических редакторов получит ученик, однажды написавший такой алгоритм! (См. также упр. 6–9 к этому параграфу.)

Помните, я рисовал роль понятия «исполнитель» (см. рис. на с. 49)? Теперь, я думаю, вы понимаете, что я имел ввиду. Тут важно даже не столько то, что монитор, дисковод и пр. оказались просто исполнителями, сколько, я повторюсь, возможность изучать и понимать их ***содержательно*** в рамках тех понятий, которыми мы уже овладели.

В учебнике приведены некоторые команды клавиатуры и экрана. У исполнителя «внешняя память» (диск) система команд более сложная — на содержательном уровне там надо сразу вводить понятие файла и т. п., однако и это возможно. Соответствующие исполнители есть в системе «КуМир» и вы можете более сильных учеников загружать изучением

более сложных устройств ЭВМ, более сложных исполнителей.

Второй важный момент, о котором я упомянул, — «физическое подключение» Робота к ЭВМ. Этот момент, правда, разбит на две части: в самом учебнике сказано лишь, как в принципе подключить какого-то исполнителя к ЭВМ (п. 19.5 — магистраль, адрес). А все, что касается Робота, сформулировано в качестве упражнений (упр. 1 на с. 164 учебника) и предлагается придумать самим ученикам. Это вторая крайне важная составляющая для глубокого понимания понятия «исполнитель». Можно клавиатурой управлять из алгоритмического языка (первая составляющая), а для управления Роботом составлять программы в машинных кодах (вторая составляющая). Общая цель всех этих упражнений — стереть всякое различие между Роботом, клавиатурой, экраном и другими исполнителями. Показать, что Робот может оказаться «машинным» исполнителем, а клавиатура — «алгоритмическим».

Я очень бегло повторю, что значит «подключение» Робота к ЭВМ. Во-первых, это значит, что Робота надо подключить к магистрали, т. е. на рис. 81 учебника (с. 162) добавить прямоугольник «Робот». Во-вторых, по аналогии с другими устройствами, Роботу следует выделить «его» адреса: один для передачи команд Роботу, а другой для получения информации от Робота (что и сделано в формулировке упр. 1). При реальном подключении Робота его команды должны быть как-то закодированы числами, так что «запись» соответствующего числа по адресу 65352 будет означать выдачу нужной команды Роботу, а чтение числа по адресу 65354 будет получением информации от Робота.

Эту кодировку ученики должны придумать сами. Например, «вправо» — 1, «влево» — 2, «вверх» — 3, «вниз» — 4, «закрасить» — 5, «справа стена?» — 6 и т. п. При этом команда «записать по адресу 65352 число 6» будет означать вопрос «справа стена?», по которому Робот должен проанализировать, есть справа стена или нет, и запомнить ответ, который далее можно будет, например, прочесть по адресу 65354. Это значит — формулировка громоздкая, но правильная до самых деталей, — что по команде «переслать из 65354 в x» Робот, обнаружив «свой» адрес, перешлет в x, скажем, 0 если он до того запомнил **нет** (стены справа не было), либо 1, если он запомнил **да** (стена справа была). Естественно, надо будет придумать какие-то комбинации 0 и 1, кодирующие также температуру и радиацию.

После того как такая кодировка команд Робота будет придумана, алгоритмы управления Роботом (самые простые — начиная с А1) можно переписать, как ***реальную*** программу в кодах ***реальной*** ЭВМ. Единственная условность — что само «подключение Робота» у нас воображаемое, реально таких устройств, подключаемых к ЭВМ, наша промышленность пока не производит (хотя и могла бы — это совсем не сложно).

И, что самое важное, — все это «настоящее». Именно так в действительности новые устройства подключаются к компьютеру. Именно так бортовые компьютеры по заданному алгоритму управляют космическим кораблем или системой впрыска и зажигания в автомобиле. И я глубоко уверен, что не разобравшись — хотя бы в принципе, — как такое возможно, нельзя создать «адекватную информационную картину мира», которую мы сформулировали как третью цель нашего курса.

> **МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Я никогда не мог этого понять, но, как это ни странно, подобное «машинное» управление Роботом почему-то полностью снимает с Робота налет «игрушечности». Стоит управление Роботом погрузить в дебри машинного кодирования, нулей и единиц, как даже запись алгоритма А1 начинает казаться чем-то «настоящим». Хотя алгоритмическая сложность А1 от этого никак не меняется — алгоритмически задача остается абсолютно тривиальной.
>
> Этот же феномен мы наблюдаем и с Бейсиком — запись абсолютно тривиальных задач на Бейсике выглядит, как что-то содержательное, в то время как решение содержательной задачи на алгоритмическом языке с Роботом может казаться простым и «игрушечным».
>
> И я еще раз обращаю ваше внимание на этот эффект и повторяю, что наша цель — развитие алгоритмического мышления учащихся, решение содержательных задач. И именно поэтому мы сделали выбор в пользу содержательных задач на алгоритмическом языке с Роботом, а не в пользу элементарных задач в кодах или на Бейсике.

Наконец, последнее, на что я еще раз хочу обратить ваше внимание. «Подключение» Робота к ЭВМ и кодирование алгоритмов управления Роботом на уровне машинных команд дает некоторое общее представление об основах работы ЭВМ. Дело в том, что такое кодирование убирает разрыв между тем, что мы обычно пишем на языке програм-

мирования, и тем, что делает машина. Решение подобных упражнений позволяет представить себе, что же происходит «внизу» — на уровне машинных команд.

Вместе с простейшим графическим редактором (упр. 9) все это должно сформировать у учеников правильное представление об исполнителях, а также показать «как» работает ЭВМ, что происходит при работе на клавиатуре, в графическом редакторе, как микропроцессор может управлять какими-то устройствами в автомобиле и пр.

§ 20. Работа ЭВМ в целом

Это — последний параграф раздела, посвященного в целом, я повторю, демистификации ЭВМ. Демистификация происходит на четырех уровнях (по одному параграфу учебника на уровень):

(1) на уровне электрических процессов, напряжения, включения и выключения транзисторов (§ 17);
(2) на уровне машинных команд (§ 18);
(3) на уровне взаимодействия различных частей ЭВМ (§ 19);
(4) и, наконец, на уровне ЭВМ и программного обеспечения вместе, как единого аппаратно-программного комплекса (§ 20).

Последний уровень (§ 20) посвящен увязке ранее пройденных понятий с реальным использованием ЭВМ. Вот школьник приходит в компьютерный класс, на экране сразу возникает система «КуМир» или Бейсик и в них прямо можно работать. Как это? Что происходит в ЭВМ? Что она делает? Как это связано с устройством процессора, машинными командами и пр.? Какие машинные команды выполняет процессор, когда мы в системе «КуМир» на школьном алгоритмическом языке набираем алгоритм? Ответу на эти вопросы и посвящен § 20.

В самом параграфе ничего особенно содержательного или сложного нет. Параграф абсолютно традиционный и содержит только описания (слова), без каких-либо примеров программ, кодов и пр.

Говорится, что когда ЭВМ работает, процессор выполняет программу (алгоритм) написанную не нами, а кем-то другим. Когда мы нажимаем на кнопки, то, в соответствии с этой программой (скажем программой «КуМир» или Бейсик), эти нажатия соответствующим образом обрабатываются, и в памяти, на экране и пр. происходит то, что должно происходить. При работе в системе «КуМир», например, в памяти формируется совсем не то, что мы набираем, а некоторое внутреннее представление алгоритма. Если мы набрали что-то неправильно, то «КуМир» (точнее компьютер в ходе выполнения программы «КуМир») проанализирует и сразу сообщит на полях (на экране) об ошибке. Когда мы нажимаем на кнопку «выполнить», КуМир начинает строка за строкой выполнять наш алгоритм, показывая ход исполнения на экране, и т. п.

Этот механизм, когда на самом деле компьютер выполняет не наш алгоритм, а некоторую другую программу (например, КуМир), и в ходе ее исполнения «изображает», «имитирует» выполнение нашего алгоритма, называется интерпретацией. Интерпретация — это когда в памяти хранится не машинная программа для нашего алгоритма, а лишь некоторая информация об алгоритме, которая интерпретируется («понимается и обрабатывается») с помощью программы-интерпретатора, выполняемой ЭВМ.

Другой подход — компиляция — состоит в том, что наш алгоритм с помощью программы-компилятора переводится прямо в выполняемую программу (в машинные коды), а потом полученная программа выполняется «сама по себе».

Эти два понятия, компиляция и интерпретация, объясняют, как вообще программа на языке программирования выполняется ЭВМ. Обычно Паскаль является компилирующей системой, а Бейсик — интерпретатором.

Интерпретация и компиляция — это две крайних, «чистых» позиции. Естественно, в жизни часто бывают разнообразные «смеси», но мы их рассматривать не будем. «КуМир», например, система более сложная, и в реальности она сделана по технологии следующего уровня, называемой «синтезом программ», но в рамках учебника ее тоже можно считать интерпретатором.

Что еще можно сказать про интерпретацию и компиляцию? При интерпретации выполняется (реально, «машинно») программа-интерпретатор, которая «имитирует» выполнение нашего алгоритма. При компиляции алгоритм сначала переводится в машинную программу, а потом выполняется. При компиляции на этапе выполнения ничего лишнего не делается. Поэтому обычно скорость выполнения при интерпретации существенно (раз в сто) ниже, чем при компиляции. Алгоритм на Бейсике выполняется в сотню раз медленнее, чем такой же алгоритм на Паскале.

Для полного завершения формирования картины мира осталось объяснить самую малость, откуда и как в памяти ЭВМ появляются КуМир или Бейсик. И что вообще происходит при включении компьютера в сеть. Какую программу и почему он начинает выполнять?

Ответу на этот вопрос посвящены два оставшихся раздела (пп. 20.5 и 20.6). Важно, что описание программы начальной загрузки, функций операционной системы производится в уже освоенных терминах машинных программ и процессора, который занят беспрестанным (бесконечным)

их выполнением. Скажем, что значит: начинает выполняться программа из постоянного запоминающего устройства? Это значит, что на заводе в эту память поместили соответствующую машинную программу, а сам компьютер сделали так, что при включении питания в Счетчик Команд попадает адрес начала этой программы. Что значит «считывается с диска»? Это значит, посылаются соответствующие команды исполнителю «диск» (или, на более низком уровне, на проводах магистрали появляются соответствующие высокие и низкие напряжения). И т. д. Главное, теперь за всеми этими словами стоит (должно стоять) реальное понимание, реальное наполнение — что именно и как делает компьютер.

Формирование такого целостного (без разрывов) восприятия от выполнения алгоритма на алгоритмическом языке до включений и выключений транзисторов под действием электрического напряжения и является целью всей главы.

Сам § 20, я повторяю, является чисто описательным. Но, памятуя о «черепахе», для реального усвоения этого материала школьники должны не просто читать и слушать, а сами решать какие-то задачи. Полезно заставить их писать компиляторы и интерпретаторы. Конечно, не настоящие (это слишком сложно и трудоемко), а какие-нибудь простенькие, учебные. Именно этому и посвящены упражнения на с. 168 учебника.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Честно говоря, я не знаю других учебников, в которых этот материал подкреплялся бы какими бы то ни было упражнениями вообще. Тут вы можете ярко наблюдать следствие основных целей и методики преподавания нашего курса, изложенных в начале пособия.

Рассмотрим простейший алгоритм — упр. 2 на с. 168 учебника. Это маленький, но настоящий интерпретатор, который анализирует некоторую информацию (строку из букв «В», «Н», «Л», «П», «К») и заставляет Робота выполнить соответствующую последовательность команд. Вспомните схему программного управления (с. 16). Человек может описать некоторый алгоритм управления Роботом в виде строки из этих букв и «отдать» программно-аппаратному комплексу ⟨ЭВМ + КуМир + алгоритм из упр. 2⟩, а дальше эта строка выполнится, и Робот получит соответствующие команды уже без участия человека. То есть это — абсолютно «настоящий», хотя и очень простой, интерпретатор.

Точно так же программа-интерпретатор с Бейсика анализирует программу на Бейсике и выдает команды, скажем, терминалу и клавиатуре. Точно так же система КуМир анализирует алгоритм на алгоритмическом языке и командует, скажем, экраном, изображая и сам алгоритм, и Робота. Другими словами, в плане «воды и ребенка» все в порядке — никакое содержание не потеряно. А алгоритм (простейший интерпретатор) при этом содержит меньше десятка строк!

Поэтому, даже когда мы говорим о таких понятиях, как интерпретация и компиляция, все эти понятия можно подкрепить простыми упражнениями, десятистрочными простыми алгоритмами. Единственное, что от нас требуется, — выделить содержание изучаемого явления «в чистом виде», сконструировать такую учебную обстановку, чтобы можно было формулировать и решать задачи на компиляцию и интерпретацию.

Я еще раз хочу подчеркнуть, что хотя сами эти компиляторы и интерпретаторы маленькие (10, 15, максимум 20 строк), тем не менее это — настоящие компиляторы, настоящие интерпретаторы.

Конечно, если мы в интерпретируемый текст добавим хотя бы возможность написать цикл n **раз**, то это сильно все усложнит. Можете предложить сильным ученикам сделать так, чтоб в строке можно было написать «К9(НЛ)К» и чтобы это означало

```
закрасить
нц 9 раз
| вниз
| влево
кц
закрасить
```

Интерпретатор станет намного сложнее. После этого можно сказать «а представьте себе, что разрешены ***ВСЕ*** изученные нами алгоритмические конструкции. Какой получится интерпретатор? Конечно, очень и очень сложный и большой. Но ***в принципе*** понятно, как это делать. Написать такой интерпретатор — большая и трудная работа, но ***в принципе*** понятная. Ничего сверхъестественного в этом нет.» Я также хочу обратить ваше внимание на алгоритм А92 (с. 176 учебника), который является интерпретатором для специальным образом закодированных программ управления Чертежником. При желании вы можете пройти п. 21.6

§ 21 прямо здесь, в рамках изучения интерпретации и компиляции. А потом лишь сослаться на него при изучении информационных моделей, как на уже пройденный материал.

Ну и наконец, прежде чем перейти к следующей главе, я сделаю еще одно замечание ко всей главе 2. Эта глава — в каком-то смысле — дань программе курса «основ информатики ***и вычислительной техники***». На наш взгляд, этот материал правильнее было бы изучать отдельно. Физические основы ЭВМ, транзисторы, вентили, триггеры и пр. — это изложение конкретного материала, развитие кругозора учеников. Значительная часть главы вообще относится скорее к физике и к логике, чем к развитию алгоритмического стиля мышления.

Если вся первая глава была в точности выдержана в соответствии с целями курса, то глава 2 в этом смысле несколько «привнесенная». Она важна для формирования мировоззрения, для формирования целостного представления о том, что происходит, когда мы включаем компьютер и пр., но она как бы из другого курса. В идеале, такой курс мог бы читаться независимо и, может быть, даже параллельно с курсом информатики. Его можно перенести и в конец (после главы 3), но мы поставили его в середину, между главами 1 и 3, чтобы разбить, разнообразить материал. Всю первую главу шел алгоритмический язык, потом — перерыв: вентили, коды, работа ЭВМ. Отдохнули, отвлеклись — и снова алгоритмический язык (глава 3) на новом, более высоком уровне. Чтобы курс был не совсем уж монотонный, чтобы были какие-то смены материала и рода деятельности, разнообразие.

Поэтому главу 2 можно спокойно двигать по курсу так, чтобы она попала на нужное место, туда, где вам нужен такой перерыв. Либо, если у вас курс сжатый и времени мало, главу 2 можно и вовсе опустить. При этом все остальное можно пройти. В главе 3 есть отдельные ссылки на материал главы 2, но их легко заменить на непосредственное изложение нужного материала в нужном месте.

Лекция 10.

Глава 3. Применения ЭВМ

Глава 3 — это в каком-то смысле то, ради чего затевалось все предыдущее.

Глава 1 была посвящена изучению школьного алгоритмического языка, базовым навыкам составления алгоритмов и простейшим (за исключением § 16) методам алгоритмизации. Это уровень «письма и счета»: мы не обсуждаем, почему мы решаем те или иные задачи. Так, на уроках арифметики, когда мы учим складывать, никого не волнует, почему мы складываем, например, карандаши, а не груши. Мы в этот момент учим складывать, а не анализируем сравнительные достоинства карандашей перед грушами. Точно так же в главе 1 не обсуждается, почему мы решаем одни задачи, а не другие. Их просто надо решать, надо «набивать руку» и учиться составлять алгоритмы.

Глава 3 посвящена «Применениям ЭВМ», т. е. задачам из реальной жизни. И здесь уже мы будем объяснять и откуда берутся задачи, и как в реальной жизни они решаются.

Выбор такой последовательности изложения — это выбор методики обучения. Бывают и другие подходы. Например, в учебнике свердловчан [Гейн] практически все разделы, начиная с самого начала, построены так: сначала идет мотивационная жизненная постановка задачи, а уж потом строятся модели (математические) и алгоритмы.

И у нашего, и у свердловского подхода имеются свои достоинства и недостатки. Достоинство свердловчан — наличие мотивации с самого начала, разнообразие постановок задач и, как следствие, более разнообразный и занимательный курс. Но, на первых порах, когда еще мало освоенных понятий, модели получаются куцыми и иногда неадекватными. И это — основной недостаток.

У нас в первой части создание мотивации возложено на учителя. Учебник ничего такого не содержит, является сухим и сжатым. Но зато при рассказе о применениях ЭВМ мы можем использовать весь круг понятий и демонстрировать полноценные информационные модели, т. е. работать на достаточно глубоком уровне, адекватном содержанию задачи.

Глава называется «Применения ЭВМ», и, как кажется, должна была бы сразу начинаться с рассмотрения каких-то

областей применения ЭВМ, каких-то жизненных ситуаций. На самом деле это не так. Глава начинается с §§ 21 и 22, которые по сути продолжают главу 1 — продолжают развитие базовых понятий и методов алгоритмизации.

§ 21 посвящен понятию информационной модели, а § 22 — четвертому, самому главному фундаментальному понятию информатики, четвертому фундаментальному понятию алгоритмической культуры — понятию информационной модели исполнителя.

Действительное изложение применений ЭВМ начинается только с § 23 и продолжается до § 27. Здесь описываются информационные системы, текстовые редакторы, моделирование на ЭВМ и пр.

Все эти параграфы устроены одинаково: в начале идет беллетристика, т. е. обычное описание очередной области применения ЭВМ, например, системы продажи авиабилетов или системы, управляющей конвейером на Волжском автозаводе. После такого описания идет вторая — гораздо более содержательная — часть: разбираются учебные информационные системы, демонстрирующие методы представления и обработки информации в соответствующей области. Грубо говоря, если первая часть параграфа отвечает на вопрос «что?» могут делать ЭВМ в данной области (как это выглядит «снаружи»), то вторая часть пытается ответить на вопрос «как?» это делается — как представляется информация, каковы алгоритмы ее обработки, т. е. как это выглядит «изнутри».

Такой подход (кажется, с легкой руки Зайдельмана) получил название «принцип корыта». В самом учебнике, описывая структуру главы (с. 169), мы написали, что, рассказав о реальных применениях ЭВМ, мы на учебных информационных системах покажем, как такие системы могут работать. И привели аналогию: рассказав об океанских лайнерах, мы покажем «корыто» и объясним, почему оно плавает. Конечно, мы имели ввиду, что мы не будем показывать и обсуждать детали, а объясним только самое на наш взгляд важное — «почему оно плавает». Но народ немедленно обозвал этот подход «принципом корыта», и, ввиду образности и легкости запоминания, это название так и «пошло».

Хочу обратить ваше внимание, что первые (описательные) части §§ 23–27 являются традиционными и аналогичный (и даже более интересный и увлекательный) материал вы можете найти практически в любой популярной или учебной книге, посвященной ЭВМ. Но в отношении содер-

жательной второй части §§ 23–27 это не так. По сути, мне неизвестны ни учебники, ни научно-популярные издания, которые бы содержательно отвечали на вопрос «как?», объясняли бы «почему же оно все-таки плавает», демонстрировали бы применения ЭВМ «изнутри».

Замечу, впрочем, что в § 27 эта вторая содержательная часть приведена в весьма ограниченном и усеченном виде: показана только информационная модель без примеров алгоритмов ее обработки.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Я честно признаюсь, что это отнюдь не замысел авторов. В действительности все куда проще и прозаичней. По техническим причинам учебник обязан был иметь объем, кратный трем печатным листам — какие-то машины в типографии почему-то сшивали только такие книги. А три печатных листа — это 48 страниц учебника! И мы должны были подгонять объем учебника под это требование. Поэтому нам пришлось выкинуть из учебника некоторое количество «применений», и, в частности, в § 27 остался только маленький — на наш взгляд, самый важный — фрагмент от раздела «как?». Никаких методических, педагогических или иных содержательных соображений за этим не стоит.

Наконец, последний § 28, хотя и описывает некоторые применения ЭВМ, в целом является скорее заключением ко всему учебнику, чем к главе 3. Этот параграф, на мой взгляд, важен, поскольку он единственный, где говорится, что ЭВМ не панацея и их применение может быть не только полезным, но и вредным.

Такова структура главы 3: первые два параграфа — самые важные и посвящены четвертому фундаментальному понятию информатики — информационным моделям исполнителей; § с 23 по 27 посвящены применениям ЭВМ и их описанию «снаружи» и «изнутри»; наконец, § 28 является заключением ко всему учебнику и посвящен роли и месту применений ЭВМ в нашей жизни.

§ 21. «Информационные модели» или, по-другому, «кодирование информации величинами алгоритмического языка»

Сказать по-правде, кодированием информации мы занимались непрерывно в течение всего курса, только раньше это никак не называли.

Самый последний пример (из § 20) — кодирование буквами «В», «Н», «Л», «П», «К» команд Робота, а строками из этих букв — алгоритмов по управлению Роботом. Да и до этого много похожего встречалось. Один из самых ярких примеров — занесение в линейную таблицу информации об уровнях радиации в коридоре (см. с. 110 учебника — алгоритм А62).

Но в главе 3 для нас уже важна «жизненность» примеров, так что Робот тут не годится. И поэтому параграф начинается с построения информационной модели кинозала.

> **ОПРЕДЕЛЕНИЕ:** информационной моделью объекта, явления и пр. называется набор величин языка программирования (для нас — школьного алгоритмического языка), с помощью которого мы задаем этот объект, явление и пр. Когда мы задаем какой-то объект с помощью некоторого набора величин алгоритмического языка, этот набор величин называется информационной моделью объекта.

Если мы хотим написать программу, продающую билеты на поезда, то набор величин, задающий все возможные маршруты, все проданные и непроданные места, цены билетов и пр. это и есть информационная модель. Для программы на школьном языке информационная модель — это набор величин школьного алгоритмического языка. Для программы на Паскале информационная модель — набор переменных Паскаля, используемых в программе. Таким образом, информационная модель — просто набор величин выбранного языка программирования.

Итак, глава 3 начинается с информационной модели кинозала. Пусть надо обеспечить продажу билетов в один кинозал, который содержит 28 рядов по 20 мест. Мы собираемся продавать билеты всего на один сеанс. Цена билета зависит от ряда. Какую информацию нам придется хранить? Про каждое место надо знать, продан или не продан на не-

го билет (т. е. свободно это место или уже продано). Надо также где-то хранить информацию о ценах на билеты, чтобы знать, почем их продавать. Все остальное для продажи билетов вроде бы неважно.

В учебнике вводятся две величины для хранения этой информации первая — логическая таблица «продано[1:20, 1:28]» — прямоугольная таблица, в точности соответствующая кинозалу, — в каждой клетке будет **да**, если место продано, и **нет**, если место не продано. Другими словами, «продано[i, j] =**да**» означает, что i-ое место в j-ом ряду продано. И вторая величина — целочисленная таблица «цена[1:28]», которая по номеру ряда говорит, сколько стоит билет в этом ряду. Например, «цена[5]» — это цена билета в 5-ом ряду, «цена[28]» — это цена билета в 28 ряду.

Эти две величины — и есть информационная модель кинозала (обозначенная в учебнике, как модель М1). Алгоритмы обозначаются буквой «А» и номером; информационные модели — буквой «М» и номером. М1 — первая информационная модель нашего учебника. Всего в учебнике их будет 20. Итак, эти две величины (М1) — информационная модель кинозала для поставленной нами задачи продажи билетов на один сеанс — я об этом еще скажу.

Но сейчас я хочу еще и еще раз подчеркнуть: что информационная модель — это набор величин. Заметьте — самих величин, а не их значений, что очень важно понимать. Информационная модель — это сами величины «продано» и «цена». Мы пока не знаем, что там внутри, каковы значения величин, какие билеты проданы, какие нет, и каковы цены — все это неважно. В информационную модель значения величин не входят. Другое дело, что потом мы можем использовать эту модель для представления (кодирования, задания) той или иной конкретной ситуации. Но это будет уже использование информационной модели. Сама же информационная модель — это просто набор величин, безотносительно значений этих величин.

А теперь вернемся к тезису «модель ***для*** поставленной нами задачи». Пусть у нас где-то есть настоящий, реальный кинозал, в котором 28 рядов по 20 мест. Когда мы придумываем информационную модель для представления информации об этом кинозале, наш выбор в первую очередь определяется тем, какие задачи мы собираемся решать с помощью информационной модели. Ведь реальный кинозал содержит огромную массу информации, которая может быть важна для решения одних задач и совершенна не нужна для других. Любая модель будет содержать не всю информацию о кинозале, а лишь некоторую «избранную». В нашей модели М1, например, нет никакой информации о цвете стен, расстоянии между рядами, наличии поломанных кресел и т. п. Ничего этого в нашей модели нет. Модель, как и всякая модель, математическая или любая другая, является отражением лишь некоторых сторон реальности, а не всего ее разнообразия. Реальность всегда богаче.

В модели М1 мы учитываем только то, что, по нашему мнению, нам нужно знать для продажи билетов. И мы считаем, что высота потолка в кинозале, цвет стен, расположение выходов и проходов, высота ряда над уровнем моря и другие характеристики не важны при продаже билетов. Это наше допущение, или, если угодно, это наша научная гипотеза. Быть может в реальном кинозале места расположены с подъемом, «веером», с проходами или еще как-то. Но это все мы не учитываем. Именно поэтому информационная модель и называется моделью. Во всякой модели всегда отражается только та часть реальности, которую мы считаем важной для наших задач.

Примерно на этом переходе от реальности к модели и сделан основной упор в свердловском учебнике информатики [Гейн]. Упор на анализе того, что теряется и что остается при таком переходе, что мы должны и что не должны учитывать при построении моделей. Однако у свердловчан строится ***математическая*** (а не информационная) модель. Например, моделью поверхности стола является прямоугольник.

У нас модели информационные, а их осознанное построение начинается только в главе 3. «Алгоритмическая составляющая» здесь не очень велика, и, с алгоритмической точки зрения, деятельность по составлению самих моделей проще, чем деятельность по составлению алгоритмов. Тем не менее, построение информационной модели — первый шаг на пути от неформальной постановки задачи к работающему на ЭВМ алгоритму. Поэтому § 21 целиком посвящен построению информационных моделей и составлению алгоритмов с их использованием.

Итак, информационная модель кинозала. Построив модель, мы можем начать решать задачи из соответствующей области (по продаже билетов). Например, имея некоторое ***состояние модели*** (значения величин) в какой-то момент, можно узнать, сколько мест уже продано, а сколько свободно, или какова общая выручка от продажи билетов. Эти фрагменты приведены в учебнике (с. 170). Вы также може-

те попросить учеников написать массу других фрагментов, например, «продать три места рядом», «продать два места рядом в последнем ряду» и т. п. Здесь вполне можно обыгрывать все жизненные ситуации. Причем фрагменты алгоритмов получаются очень простые, много проще тех, которые рассматривались в конце первой главы (т. е. их «алгоритмическая составляющая» не очень велика).

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я еще раз хочу подчеркнуть, что простота нашей модели и наших алгоритмов определяется тем, что значительную часть информации о кинозале мы не учитываем. Пусть например, пришел парень с девушкой в кино, попросил два места рядом в последнем ряду и получил места номер 10 и 11. Соседние номера? Да, соседние. А соседние ли места? Увы, неизвестно. Если в зале один центральный проход, то наша пара будет разочарована, получив места с соседними номерами, но разделенные проходом. Так что, если мы собираемся с помощью компьютера продавать билеты парам (на соседние места), то в модель придется включить информацию о проходах.

Можно также слегка варьировать постановку задачи, например, потребовать, чтобы все билеты в 13 ряду и на 13 место в любом ряду продавались со скидкой. Здесь можно придумать много разных подобных задач, вплоть до таких, которые потребуют внести изменения в саму информационную модель. Например, если места по краям ряда стоят дешевле, чем в середине, то таблицу «цена» приходится делать прямоугольной. Если число мест в разных рядах будет разным, то для задания зала придется придумать еще какие-то величины. И т. п.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Поскольку вы учителя, я сразу обращаю ваше внимание, что в этом параграфе всюду пишутся ***фрагменты***, а не алгоритмы. Посмотрите — нигде нет заголовков **алг**, **дано**, **надо**, **нач**, **кон**. И это не случайно. В этом месте нельзя написать алгоритм целиком. Потому что, если мы вздумаем написать **алг**, то встанет вопрос, куда девать величины модели (М1). Мы вынуждены будем включить эти величины в параметры (аргументы и результаты) алгоритма. (Иначе после завершения алгоритма вся информация о состоянии модели пропадет.) Ведь нам надо, чтобы и алгоритм, который продает билеты, и алгоритм, который считает количество проданных билетов, и алгоритм, который считает общую выручку, и другие алгоритмы работали с одной и той же информационной моделью и чтобы ее состояние сохранялось между разными вызовами разных алгоритмов.

В рамках того, что мы уже прошли, есть только один путь — вписать величины модели в аргументы и результаты ***абсолютно всех*** алгоритмов, работающих с данной моделью, и вызывать эти алгоритмы из одного «главного» алгоритма. Для модели М1 это — пусть громоздко и некрасиво — еще можно сделать. А как быть с моделью М16, где 9 величин и из них 7 таблиц? А как быть с реальными моделями, где число величин может измеряться сотнями?

В § 21 мы этот вопрос мягко обходим, записывая только ***фрагменты*** алгоритмов без заголовков.

Вопрос из зала: А как на компьютере выполнять?

Ответ: И на компьютере эти фрагменты не выполняются. Мы их пишем в тетрадях и на доске.

Реально мы можем оформить эти фрагменты в алгоритмы после прохождения § 22, который именно этому вопросу и посвящен. Информационная модель исполнителя и есть набор алгоритмов, работающих с общей информационной моделью. Но это — тема следующего параграфа.

Вы можете пойти по нашему пути — пишем фрагменты, а почему — объясним потом. Можете сразу объяснить проблему, как я ее только что вам описал, и сказать, что именно поэтому мы пишем фрагменты, а как выходить из этой ситуации разберемся потом. Наконец, можете перемешать материал §§ 21 и 22, введя сразу конструкцию **исп**—**кон**, а потом решать задачи, уже имея все необходимые средства алгоритмического языка.

Я еще раз обращаю ваше внимание, что мы пишем только фрагменты алгоритмов без начала и без конца. В § 21 подругому нельзя. И если кто-то из учеников начинает писать **алг**, **дано**, **надо**, **нач** (что вполне естественно для них к этому времени), то я рекомендую просто сказать, что пока этого писать не надо, а почему не надо, разберемся в следующем параграфе.

Вот, собственно, и все про информационные модели. Понятие введено. Примеры вы посмотрите в учебнике — они все очень простые.

Итак, что же такое информационная модель? Если есть некоторая реальность, с которой мы собираемся работать, то мы придумываем величины алгоритмического языка, которые эту реальность отражают. Точнее, отражают то, что нам нужно для решения задач. Такое придумывание всегда проводится в рамках некоторого класса задач.

В примере выше мы не рассматривали задачу уборки кинозала и поэтому не интересовались величиной проходов (что может оказаться критически важным для машинной уборки зала). Мы также не рассматривали задачу продажи билетов на соседние места и потому не интересовались расположением проходов. Мы собирались продавать билеты, так сказать, по одному, и потому отразили кинозал в виде модели М1. Если бы мы рассматривали другие задачи, например управление системой вентиляции кинозала в зависимости от того, на какие места проданы билеты, то у нас, по-видимому, получилась бы совсем другая информационная модель.

Дальше в параграфе начинается наша «черепаха»: чтобы усвоить материал, школьники должны много работать, и поэтому дальше идет просто модель за моделью для разных объектов и задач.

Следующая модель — модель транспортной сети. 100 городов связаны между собой воздушными линиями так, что из любого города в любой другой можно добраться, хотя бы и с пересадками. Какая тут может быть информационная модель? Правильно, пока — никакой. Потому что, напоминаю, без формулировки класса задач, которые мы собираемся решать, никакой модели построить нельзя. Нас интересует класс задач, связанный с добиранием из одного города в другой, с выбором маршрута. Скажем, как добраться из Москвы в Париж с минимальным числом пересадок. Или есть ли маршрут из Парижа в Архангельск не более, чем с одной пересадкой. Кроме того, нас будет интересовать длина маршрута (расстояние). Например, если мы летим из Москвы в Париж по выбранному маршруту, то сколько километров мы при этом пролетим. Вот класс задач, исходя из которого мы будем строить модель.

После того как класс задач сформулирован, можно строить информационную модель. Мы построили модель М2 (с. 171 учебника), я ее повторять не буду. Важно лишь понимать, что здесь нет никакой единственности — можно построить много разных «правильных» информационных моделей. М2 — одна из них.

Далее, как всегда, можно решать разные задачи, т. е. писать разные фрагменты обработки информации этой информационной модели. У нас приведен фрагмент алгоритма, который ищет, есть ли из одного города в другой маршрут не более, чем с одной пересадкой. И опять фрагменты алгоритмов получаются достаточно простые. Вы можете усложнять задачи, например, попросить найти все маршруты из одного города в другой не более, чем с одной (или двумя) пересадками. И т. п.

Я лишь хочу обратить ваше внимание, что информация о расстояниях в приведенном в учебнике фрагменте не задействована. Но можно загрузить сильных учеников вполне содержательной задачей «найти длину кратчайшего маршрута» между двумя заданными городами. При всей простоте модели эта задача уже достаточно сложна.

Примеры информационных моделей, которые идут дальше, призваны продемонстрировать, что информационную модель можно построить почти для чего угодно.

Информационные модели геометрических объектов

В разделе 21.4 приводятся информационные модели для точки, отрезка, прямоугольника, квадрата, окружности и т. д.

На примере информационной модели точки на плоскости очень хорошо демонстрировать разницу между математической и информационной моделями. «Точка на плоскости» — геометрическое понятие. Ее можно задать координатами — и это будет алгебраическое (математическое) представление (или модель) точки. А вот если мы напишем «**вещ** x, y», т. е. две вещественные величины алгоритмического языка, то получим информационную модель точки. Точка в математическом смысле — это одна точка, конкретная пара чисел — координат точки. Информационная модель точки — это две величины, значения которых могут меняться с течением времени, в ходе исполнения алгоритма. Значения «алгоритмической» точки могут меняться. Информационной моделью точки называются сами величины x и y, а не их значения.

И еще. Координаты точки можно задать в виде «**вещтаб** x[1:2]» (модель М13) — и это будет ***другая*** информационная модель. Обратите внимание: с математической точки зрения ничего не изменилось — точка по-прежнему

задается той же парой своих координат. Но теперь вместо двух величин у нас таблица из двух элементов, т. е. ***другой*** набор величин алгоритмического языка, ***другая*** информационная модель.

Из информационных моделей геометрических объектов следует обратить внимание на информационную модель квадрата (М11). Задав квадрат в виде координат центра (a, b) и координат вектора (u, v) из центра в одну из вершин, легко выразить и координаты вершин квадрата, — это будут $(a + u, b + v)$, $(a + u, b - v)$, $(a - u, b + v)$, $(a - u, b - v)$.

Конечно, геометрические объекты — это не объекты из реальной жизни. Поэтому мы построили их информационные модели без труда, не уточняя «класс задач». Точнее, эти модели годятся для любых геометрических задач. Вот если бы мы рассматривали не геометрические, а «реальные» точки, то нам понадобилось бы уточнить, что мы собираемся решать геометрические задачи, и потому размеры точки, чем и на чем она нарисована, какого она цвета и пр., нас не интересуют. Однако эту часть работы за нас выполнил Евклид (и его предшественники). Евклид в начале своего курса геометрии объяснил, что у точки нет никакого размера и никаких различимых частей. Этому мы и следовали, создавая информационную модель точки.

И, как всегда, когда модели построены, можно решать разнообразные задачи, писать соответствующие фрагменты алгоритмов. В учебнике приводится фрагмент для нахождения периметра n-угольника, алгоритмическое условие (формула), соответствующая геометрическому высказыванию «окружность, заданная моделью М4, находится внутри прямоугольника, заданного моделью М10» и т. п. Я еще и еще раз подчеркиваю, что всюду в этих фрагментах мы используем величины модели, получаем формулы и алгоритмы, независимо от значений этих величин. Значения могут меняться в ходе выполнения алгоритма, и это будет означать, что в ходе выполнения алгоритма обрабатываются ***разные*** геометрические объекты, заданные ***одной и той же*** информационной моделью. Это связка геометрических понятий и информационных моделей.

Эта часть параграфа, начиная с модели кинозала и кончая моделями геометрических объектов, чрезвычайно проста. За исключением собственно понятия информационной модели в ней нечего понимать и объяснять. А вот дальше происходит некоторое усложнение, которое ученые назвали бы «метапереходом».

Информационные модели поля Робота

По сути дела материал и дальше ничуть не сложнее «кинозала». Вся сложность состоит в том, чтобы преодолеть инерцию мышления и взглянуть на то, что мы уже проходили раньше, с совершенно другой точки зрения. Ибо дальше строятся информационные модели того, к чему школьники уже привыкли, — информационные модели обстановки на поле Робота и даже информационные модели алгоритмов! «Алгоритмы обработки информации для информационных моделей алгоритмов управления Роботом» — тут, конечно, голова кругом может пойти. Поэтому давайте разберем это постепенно, шаг за шагом.

Вспомните вещественную таблицу с информацией об уровнях радиации в коридоре (А62). Так вот, вещественная таблица «a[1 : n]» — это, как мы теперь знаем, информационная модель коридора для класса задач, в которых нам важна только радиация и ничего больше. Если, однако, мы захотим решать задачи типа «определить количество боковых выходов из коридора», то нам понадобится более сложная информационная модель коридора, чтобы мы смогли задать и информацию о стенках тоже. Такая полная модель коридора приведена в учебнике на с. 174 (модель М15). Слово «полная» означает, что мы можем решать любые задачи про Робота в коридоре. Конечно, такие «полные» модели возможны только, если моделируемые объекты не из реальной жизни, а сами в каком-то смысле являются моделями (как геометрические объекты или поле Робота).

Дальше происходит переход от модели «Робот в коридоре» к модели «Робот на ограниченном поле размером 16 на 32 клетки» (модель М16). Никакого принципиального усложнения здесь нет. Но в этот момент, и по-моему это очень важно, можно отвлечься от темы информационных моделей и объяснить школьникам, как они работают с Роботом на компьютере. Мы все время говорили о Роботе, как об устройстве, действительно существующем. Однако при работе на компьютере в системе КуМир на экране ***изображается*** небольшое по размерам поле, ***изображаются*** стенки на поле, сам Робот и пр. Мы ***воображаем***, что Робот существует, что он как-то подключен к ЭВМ, а ***на самом деле*** наблюдаем все это на экране. Как это получается? А ответ очень прост: в действительности мы работаем не с Роботом, а с его ***информационной моделью***, которая и изображается на экране компьютера.

Здесь происходит раскрытие того инструмента (системы КуМир), с которым работают школьники. Раньше мы говорили просто «Робот», да и не могли сказать по-другому. А теперь говорим, что на самом деле вы работали не с Роботом, а с информационной моделью, т. е. просто с набором величин алгоритмического языка. И вы можете ***сами*** написать, что значит «вправо» (в модели М16 это команда $x := x + 1$), что значит «закрасить», и т. п. Все эти «команды Робота» превращаются просто во фрагменты алгоритмов использующих ***и меняющих*** состояние информационной модели М16. Здесь, конечно, ничего не сказано про изображение модели М16 на экране. Но все остальное представлено и целиком соответствует реальности.

Это чрезвычайно глубокое и важное место, я к нему еще вернусь. А сейчас запомните, пожалуйста, хорошенько, что мы все время воображали Робота существующим, а работали с его информационной моделью. И это было удобно. И облегчало составление алгоритмов управления Роботом, поскольку позволяло вообще не думать, как Робот работает и как он устроен внутри.

Повторю, что материал этот очень простой, совсем как «кинозал» и вся сложность состоит в новом взгляде на старое и давно знакомое понятие. Если бы мы взяли не Робота, а другого исполнителя, например «Уборщика кинозала», то не было бы никаких проблем. А когда мы берем понятие, к которому уже давно привыкли, с которым много работали, и пытаемся представить его совсем по-другому, то это немного сложно. Сложно отказаться от уже привычного и взглянуть на это понятие по-новому. Хотя это трудности скорее психологического плана, чем содержательного.

Информационные модели алгоритмов

Но модель для Робота и его поля — это еще «цветочки». «Ягодки» — это модели для алгоритмов управления Роботом. Ибо кинозал и поле Робота — объекты сходной природы. Алгоритм — что-то совсем другое. И следующий раздел должен продемонстрировать, что информационные модели — универсальное средство, информационно моделировать можно все, что угодно. Это очень важный момент. Информационную модель можно построить для любого объекта, для любой реальности, ибо информация присутствует всюду. И алгоритм — тоже объект, содержащий какую-то информацию. А значит его можно задать какой-то информационной моделью, обработать эту модель и т. п.

Но для этого нужно совершить «метапереход»: если раньше объектом был Робот на поле и мы составляли алгоритмы по управлению этим объектом, то теперь сам такой алгоритм мы должны рассмотреть как объект, задать его какой-то моделью и составлять алгоритмы по обработке этой модели, т. е. алгоритмы, обрабатывающие алгоритмы.

В учебнике для этого используется кодировка простейших алгоритмов управления Роботом с помощью букв «В», «Н», «Л», «П», «К» (см. главу 2). Я вам напоминаю, что если вы раздел 2 пропускаете, то в этом месте надо вместо ссылки на главу 2 ввести эту кодировку непосредственно, благо это очень просто. Величина **пит** t» из алгоритма А91 главы 2 — это и есть информационная модель (М17) алгоритмов из соответствующего класса, т. е. алгоритмов, состоящих только из команд «вверх», «вниз», «вправо», «влево», «закрасить». (Каждая команда кодируется одной буквой, а весь алгоритм — последовательность команд — литерной строкой, состоящей из этих букв.)

Как всегда, когда информационная модель задана, можно решать разные задачи по обработке модели. Например: «определить, на сколько клеток по горизонтали и вертикали от начального положения сместится Робот в результате выполнения алгоритма, заданного моделью М17». Это — простая задача. Надо завести величины для подсчета смещения, например, «**цел** x, y»: x — смещение по горизонтали, y — по вертикали. Вначале присвоить $x := 0$; $y := 0$, а потом анализировать модель М17 («**пит** t»), буква за буквой, и на каждое «В» (вверх) прибавлять единицу к y; на «Н» (вниз) — вычитать единицу из y; на «П» (вправо) — прибавлять единичку к x; а на «Л» (влево) — вычитать единицу из x. В итоге у нас в x, y получится искомое смещение конечного положения Робота от начального — ***если выполнить алгоритм, заданный строкой*** t ***(модель М17)***.

Заметьте, что я вам описал простенький алгоритм (вы легко можете сами записать его целиком), который по состоянию модели М17 вычисляет, на сколько сместится Робот. Хотя нету ни самого Робота, ни алгоритма по управлению Роботом на алгоритмическом языке, ни процесса его выполнения. Мы просто анализируем какую-то информацию, строку из букв и вычисляем смещение Робота, т. е. работаем с ***информационной моделью алгоритма по управлению Роботом***.

Или более сложная задача — определить сколько клеток закрашивает Робот (имея ввиду, что если одна и та же клетка красится несколько раз, то ее надо учесть только единожды). Упрощенный вариант этой задачи приведен в упр. 18 к параграфу.

Еще раз обращаю ваше внимание, что при решении этих задач алгоритм, заданный моделью М17, лишь анализируется, — но отнюдь не исполняется! Никаких команд Роботу вообще не отдается. Мы лишь смотрим, какие буквы записаны в строке, и что-то в каких-то величинах считаем!

Вся сложность здесь — в переходе: мы привыкли, что объектом всегда был Робот, но не алгоритм. Алгоритм — это то, что мы писали, а не то, над чем работали. А тут все меняется местами: алгоритм становится объектом, Робот вообще превращается в элемент описания, поскольку ему никто никаких команд уже не отдает, — он лишь «имеется ввиду», придает некий воображаемый смысл командам, а мы пишем алгоритм по обработке алгоритма.

Здесь важно, что какой бы объект мы ни рассматривали, будь то кинозал, обстановка на поле Робота, алгоритм управления Роботом или что-то еще, на все можно смотреть с информационной точки зрения. Мы можем закодировать величинами алгоритмического языка (т. е. представить в виде информационной модели) информацию и о кинозале, и об обстановке на поле Робота, и об алгоритме. Для любой реальности, любой действительности можно построить информационную модель. И, кстати, информационная модель — это тоже информация, и для нее тоже можно построить информационную модель информационной модели. И т. д.

Но если вы считаете алгоритмы обработки алгоритмов чересчур сложными для восприятия, можете их опустить (вместе со смыслом понятия «интерпретация»), либо ограничиться изложением моделей для алгоритмов управления Чертежником (п. 21.6), которые, на мой взгляд, психологически воспринимаются проще, чем модели для алгоритмов управления Роботом. Соответствующая модель (М18), приведенная на с. 175 учебника, очень проста, и я ее повторять не буду.

Обратите внимание на алгоритм А92 (с. 176 учебника), который, анализируя состояние модели, выдает соответствующие команды Чертежнику. Это еще один пример программы-интерпретатора, которая, анализируя некоторую

информацию, некоторое представление алгоритма, этот «закодированный» алгоритм выполняет. В рамках связи с главой 2 и устройством ЭВМ обратите внимание на замечание на с. 176 учебника, непосредственно перед упражнениями. Индекс i в ходе исполнения алгоритма перемещается от одной команды Чертежника к другой точно так же, как счетчик команд перемещается от одной машинной команды к другой, каждый раз показывая, какую следующую команду следует выполнить. Здесь происходит еще одна связка с устройством ЭВМ, поскольку интерпретация алгоритмов и выполнение машинных программ, в общем, очень похожи друг на друга.

И последнее. Я уже говорил, что при работе в системе «КуМир» школьник видит на экране поле Робота и вообще работает не с «настоящим» Роботом, а с его информационной моделью. Но ведь школьник видит на экране и свой алгоритм по управлению Роботом, а компьютер, как мы говорили, всегда выполняет некоторую машинную программу. Какова же связь между изображаемым на экране алгоритмом и работой компьютера? Как вообще все происходит ***на самом деле***? На мой взгляд, глубокое реальное понимание этих процессов возможно только после изучения главы 2, информационных моделей, понятия информационной модели алгоритма и понятия программы-интерпретатора. Именно поэтому, несмотря на некоторые сложности восприятия (повторю, при простом — по сути — материале), эти вопросы попали в школьный учебник, как необходимый элемент демистификации ЭВМ и формирования адекватного представления о современной информационной реальности. Мы обязаны, хотя бы в принципе, сформировать у школьников адекватное представление о том, как ***на самом деле*** работает ЭВМ.

Резюме

Итак, резюме по этому параграфу. Во-первых, материал параграфа очень прост. Для изложения понятия информационной модели вообще достаточно кинозала. Все возможные сложности связаны не с понятием информационной модели как таковой, а с приложением этого понятия к новым объектам: обстановке на поле Робота и — особенно — к алгоритмам.

Во-вторых, с методической точки зрения крайне важно понимание универсальности информационного моделирования. Если это будет понято, то станет понятным фактически все: почему на экране изображается лабиринт (потому что это — изображение информационной модели, с которой работает ЭВМ) как ЭВМ выполняет алгоритм, (она строит внутри себя информационную модель алгоритма и потом его «интерпретирует»). Здесь же следует повторить про начальную загрузку, операционную систему и пр. из § 20.

Если это не будет понято, то останется пробел — не в понятии информационной модели, а в общей картине мира, в представлении о том, как на самом деле работает компьютер. И этот пробел знаний могут занять мифы, формирующие неадекватное, неверное представление о реальности.

Упражнения параграфа можно разбить на три класса:

1) в рамках заданной информационной модели написать фрагмент какого-то алгоритма (типа «продать три места в одном ряду» в модели кинозала);

2) изменить информационную модель так, чтобы учитывалась какая-то дополнительная информация, чтобы можно было решать более широкий класс задач (скажем, продавать места на два сеанса или еще что-нибудь).

3) придумать «с нуля» информационную модель. Например, информационную модель прямоугольной пирамиды, расписания уроков, паспорта или свидетельства о рождении. В последних задачах имеется в виду обработка информации, ***записанной*** в паспорте или свидетельстве о рождении (т. е. внешний вид, толщина бумаги, наличие жирных пятен и пр. для нас не важны). Сложность модели будет зависеть от детальности представления информации и ваших требований. Например, если для информационной модели камеры хранения на вокзале вы потребуете, чтобы через трое суток хранения ячейка помечалась, как просроченная (гудела и не открывалась), то модель будет чуть сложнее, чем если вы такого условия ставить не будете.

Я вам рекомендую использовать упражнения разных типов. Кроме того, вы без труда можете дополнить приведенный после параграфа набор упражнений своими, например, попросить учеников составить алгоритмы для выполнения простейших геометрических преобразований над теми или иными объектами планиметрии (модели М3–М11): симметрия относительно координатных осей, параллельный перенос, поворот на 90 градусов, произвольный поворот, гомотетия относительно начала координат и пр.

§ 22. Информационные модели исполнителей, или исполнители в алгоритмическом языке

В начале изложения этого материала необходимо отметить два момента.

Первый — исходящий из фундаментальной сущности информационного моделирования. Поскольку информационные модели можно строить для чего угодно, давайте построим информационную модель Робота как исполнителя, целиком, т. е. вместе со всеми его командами.

Раньше мы строили информационные модели только для данных: кинозал, города, расстояния между ними, фигуры на плоскости и пр. (И даже алгоритмы в п. 21.6 мы рассматривали, как данные, как объекты.) А теперь давайте рассмотрим исполнителя Робот — целиком. У него кроме данных, обстановки, поля, существуют еще и команды: он умеет ходить вправо и влево, закрашивать клетки, сообщать температуру и радиацию и пр., т. е. совершать ***действия***. Как нам отразить, как задать эти действия? Действие (команду),«вправо» лучше всего отразить алгоритмом — мы это уже знаем, мы уже писали $x := x + 1$. Но до сих пор у нас все было кусочками и по отдельности. Отдельно — информационная модель М16. Отдельно — команда $x := x + 1$. Как это все собрать воедино? Как сказать, что $x := x + 1$ — это и есть команда «вправо» в нашей модели?

Подойдем к этому, как к информационному моделированию некоторой реальности, — есть исполнитель Робот, как промоделировать его целиком вместе с обстановкой (что уже было) и с командами (чего пока не было)?

Второй момент — чисто прагматический, или, если хотите, технический. Даже для модели кинозала нам пришлось писать фрагменты алгоритмов, потому что было непонятно, что делать с информационной моделью, куда помещать ее величины. Что же касается модели М16 для поля Робота, то очевидно, что повторять описание всех величин модели в качестве аргументов и результатов алгоритмов «вправо», «влево» и т. д. просто какое-то безумие. Поэтому нам нужны какие-то новые конструкции языка программирования, которые позволят разным алгоритмам работать с одними и теми же величинами. Чтобы значения этих величин не пропадали от вызова одного алгоритма до вызова другого. Что-

бы алгоритм, считающий выручку в модели кинозала, учел результаты работы и алгоритма по продаже билетов, и алгоритма по возврату билетов. Вы можете также повторить здесь аналогию со строительными бригадами (с. 29), которая является абсолютно точной.

Для задания группы некоторых общих величин и группы алгоритмов, работающих с использованием этих величин, т. е. для задания и информационной модели, и алгоритмов обработки информации в этой модели, в алгоритмическом языке есть специальная конструкция **исп**—**кон**, называемая информационной моделью исполнителя, или просто исполнителем. Это и есть то четвертое фундаментальное понятие алгоритмизации, то четвертое фундаментальное понятие информационной культуры, о котором я говорил в самом начале.

Обратите внимание, что информационная модель исполнителя — это новое понятие, не укладывающееся в понятие информационной модели, как набора величин. Точнее, информационная модель исполнителя — более сложное понятие, содержащее внутри себя и набор величин (информационную модель в смысле § 21 — модель «обстановки» исполнителя), и алгоритмы обработки этих величин («команды» исполнителя). Именно поэтому понадобилась и новая конструкция алгоритмического языка **исп**—**кон** («исполнитель — конец исполнителя»).

Используя это понятие мы можем от записи фрагментов перейти к полной записи алгоритмов и (впервые!) к полной ***записи исполнителей на алгоритмическом языке***. Рассмотрим, например, исполнителя «кинозал». Обратите внимание, не «информационная модель кинозала», а «исполнитель кинозал». В чем разница? «Информационная модель кинозала» — это набор величин алгоритмического языка (М1). «Исполнитель кинозал» — это и величины модели М1, и алгоритмы ее обработки те самые алгоритмы, фрагменты которых мы приводили в предыдущем параграфе, но не могли записать.

Исполнитель «кинозал» на алгоритмическом языке можно записать так:


```
исп Кинозал
| лог таб продано [1:20,1:28] | продано[i,j] ⟺
|                               | место i ряда j продано
| цел таб цена[1:28]          | цена[j]=цена билета в j-м ряду

| цел i,j
| нц для j от 1 до 28          | для каждого ряда
| | нц для i от 1 до 20        | для каждого места в ряду
| | продано[i,j]:=нет          | запомнить, что место свободно
| | кц
| | выбор                      | установить цену билета
| | | при j< 5 : цена[j]:=1000  | в зависимости от
| | | при j<10 : цена[j]:=3000  | номера ряда
| | | при j<20 : цена[j]:=5000  |
| | | иначе    : цена[j]:=8000  |
| | все
| кц
кон

алг цел число проданных
  надо | знач=число проданных билетов
нач
| знач:=0
| нц для j от 1 до 28    | для каждого ряда
| | нц для i от 1 до 20  | для каждого места в ряду
| | | если продано[i,j]
| | | то знач:=знач+1
| | | все
| | кц
| кц
кон

алг цел выручка
 надо | знач=выручка от продажи билетов
нач
| знач:=0
| нц для j от 1 до 28    | для каждого ряда
| | нц для i от 1 до 20  | для каждого места в ряду
| | | если продано[i,j]
| | | то знач=знач+цена[j]
| | | все
| | кц
| кц
кон
.....(остальные алгоритмы исполнителя).....

кон
```

После строки **исп** (исполнитель) задается информационная модель кинозала (модель М1), т. е. описываются величины алгоритмического языка. Потом идет текст на алго-

ритмическом языке, задающий начальные значения величин модели. В нашем примере в качестве начальных значений запоминается, что все места свободны, а также задаются цены билетов. После этого, один за другим, пишутся все алгоритмы — все «команды» исполнителя. Завершает все служебное слово **кон** (конец исполнителя). Это — та самая новая конструкция алгоритмического языка, (четвертое фундаментальное понятие алгоритмизации), которая за последние 20 лет (по разному называемая) появилась во всех современных языках программирования.

Можно представлять себе конструкцию **исп**—**кон** образно, в виде ромашки (рис. 12). Сердцевина — информационная модель, набор величин и задание их начальных значений. Лепестки — алгоритмы, работающие с величинами данной модели.

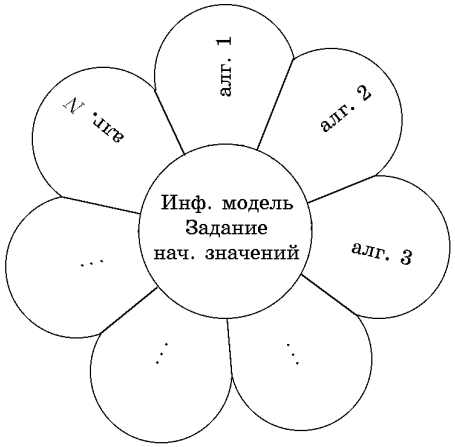


Рис. 12.

Величины информационной модели принято называть ***общими величинами исполнителя***. Это новое понятие, новый ***вид*** величин алгоритмического языка: это и не аргументы, и не результаты, и не промежуточные величины.

Обратите внимание, что я уже перестал говорить «информационная модель исполнителя» и говорю просто «исполнитель». Давайте я еще раз поясню, почему это так, почему

мы не делаем между этими понятиями никакой разницы. Т.е. почему мы не делаем разницы между внешним исполнителем («в железе»), подключенным к ЭВМ, и информационной моделью исполнителя, обрабатываемой внутри ЭВМ.

Вспомните, — на протяжении всей первой главы мы рассматривали Робота как внешнего исполнителя, существующего «в металле», подключенного к ЭВМ. И мы писали алгоритмы по управлению этим исполнителем, считая Робота «внешним». Лишь совсем недавно я вам признался, что ***на самом деле*** мы работали с информационной моделью исполнителя Робот, входящей в состав системы «КуМир». Мы считали Робота «внешним», и это нам совсем не мешало. Наоборот, мы могли не задумываться о том, как Робот «реализован», какими величинами задаются стены, какими величинами задается положение Робота и пр. Очень трудно, почти невозможно думать одновременно и над тем, как составить алгоритм по управлению Роботом, и над тем, как это все внутри информационной модели Робота будет выполняться.

Поэтому первое утверждение звучит так: человеку при решении задач ***удобно*** представлять себе исполнителей, как внешних, подключенных к ЭВМ, даже если они ***на самом деле*** всего лишь информационные модели, обрабатываемые внутри ЭВМ.

В чем, собственно, разница между этими ситуациями, если говорить об использовании исполнителей, об управлении ими? В случае «внешнего» исполнителя ЭВМ, выполняя написанный нами алгоритм, командует «настоящему» Роботу, который исполняет команды. В случае информационной модели ЭВМ, выполняя написанный нами алгоритм, вызывает соответствующие алгоритмы информационной модели Робота («команды» исполнителя Робот), как вспомогательные, т. е. сама же их и исполняет.

Вся разница в том, как и кем исполняются «команды» Робота. Но сам алгоритм управления Роботом в обоих случаях ***абсолютно один и тот же***. При составлении этого алгоритма совершенно неважно, есть ли Робот на самом деле или он будет имитироваться (моделироваться) с помощью ЭВМ. Именно этим мы и воспользовались, когда говорили, что пишем алгоритмы управления «настоящим» Роботом, а сами выполняли алгоритмы и наблюдали за полем Робота на экране ЭВМ.

А поскольку представлять себе «внешних» исполнителей проще, чем информационные модели, и для «внешних» ис-

полнителей можно вообще не думать, как они устроены внутри, мы и используем более простое слово «исполнитель» при управлении как «внешними», так и «информационными», «внутренними» исполнителями. Это первое.

Кстати, для задания информационной модели исполнителя часто используется термин «реализация исполнителя». «Реализация исполнителя» — это текст на алгоритмическом языке от **исп** до **кон**. Так, например, на с. 273 приведена ***реализация исполнителя Кинозал***. В учебнике (с. 180) показана ***реализация исполнителя «Маленький Робот»***.

И второе, — самое важное. Это понятие информационной модели исполнителя, эта конструкция алгоритмического языка **исп**—**кон**, это четвертое фундаментальное понятие алгоритмической культуры может (и должно) использоваться нами, как ***инструмент*** алгоритмизации, как способ структуризации нашего мышления при решении алгоритмических задач. Основная ценность понятия исполнителя состоит в том, что это — способ структуризации мышления, способ накопления алгоритмических «знаний». Ученикам достаточно овладеть конструкцией **исп**—**кон** и научиться ее применять, что достаточно просто. Вы же, как учителя и методисты, должны еще и понимать место и роль этого понятия в информатике в целом. И именно поэтому я все время повторяю про четвертое фундаментальное понятие алгоритмической культуры.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Скорее всего вы уже слышали про объектно-ориентированные языки программирования. В этих языках наше четвертое фундаментальное понятие алгоритмической культуры, понятие объекта (в нашей терминологии — исполнителя) поставлено на первое место. И успех объектно-ориентированных языков, или точнее, объектно-ориентированного подхода к составлению алгоритмов, как раз и объясняется тем, что в основу и методов придумывания, и способов записывания алгоритмов положено понятие исполнителя (объекта).

Я сейчас раскрою роль понятия «исполнителя», как способа алгоритмизации. Но прежде, я хочу напомнить вам рисунок (с. 49), демонстрирующий роль понятия «исполнитель» в нашем курсе и, соответственно, в учебнике. Где-то к § 12 «исполнители» практически сошли на «нет». Мы решали задачи, ни в постановке, ни в решении которых никаких исполнителей не было вообще. И вот мы снова вернулись к

исполнителям, но на совершенно другом уровне (как иногда говорят, «на новом витке диалектической спирали»). Вернулись к исполнителям, как к внутреннему понятию алгоритмического языка, как к способу решения алгоритмических задач. Именно это и было отражено на графике на рис. С2 (с. 49).

Прежде чем перейти к анализу понятия «исполнитель», как способа алгоритмизации, давайте уточним, что теперь представляет собой программа на алгоритмическом языке в целом, т. е. что именно мы отдаем на исполнение ЭВМ в схеме программного управления. Вначале это был один, отдельно взятый алгоритм. Потом — алгоритм и набор вспомогательных алгоритмов. Теперь, после введения конструкции **исп**—**кон** программа кроме «основного» и вспомогательных алгоритмов, может содержать еще и какое-то количество исполнителей (или, точнее, реализаций исполнителей), внутри которых могут быть свои алгоритмы — алгоритмы команд исполнителя.

И вот это представление о программе, как о наборе алгоритмов и исполнителей, является полным в смысле современного состояния информатики.

> **ПРОГРАММНОЕ ОТСТУПЛЕНИЕ**. В реальности — в системе «КуМир» — «основной» и вспомогательные к нему алгоритмы, которые мы раньше представляли, как находящиеся «вне» всяких исполнителей, на самом деле считаются расположенными внутри специального — безымянного и никак специальным образом не оформленного — «основного» исполнителя. Соответственно, можно задать общие величины этого исполнителя, доступные всем его алгоритмам, написать фрагмент, устанавливающий их начальные значения (и выполняющийся до начала выполнения «основного» алгоритма) и т. п.

> **ЕЩЕ ОДНО ПРОГРАММНОЕ ОТСТУПЛЕНИЕ**. В производственных объектно-ориентированных языках программирования внутри исполнителей, как правило, могут быть не только алгоритмы, но и другие исполнители. У нас это не так, поскольку никаких новых фундаментальных понятий такое «упрятывание» одного исполнителя внутрь другого не добавляет.

А теперь представьте себе программу, состоящую из двух частей:

1) некоторый алгоритм управления Роботом, за которым следует

2) реализация исполнителя «Маленький Робот» со с. 180 учебника.

Как будет выполнятся такая программа? При выполнении алгоритма по управлению Роботом, встретив, например, команду «вправо», ЭВМ вызовет вспомогательный алгоритм «вправо» исполнителя «Маленький Робот».

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Обратите внимание, что на с. 180 учебника дана ***упрощенная*** реализация исполнителя Робот. И дело не только в ограниченном размере поля Робота. Например, в приведенном алгоритме «вправо» реализован (промоделирован) только сам шаг вправо. Если при выполнении этого алгоритма справа от Робота будет стена или край поля, то Робот спокойно «пройдет сквозь стену» или выйдет за край поля. Хотя при правильном моделировании в этой ситуации должен возникать отказ. Вы можете указать на эту ***ошибку реализации*** школьникам и попросить их исправить алгоритм. Скорее всего, школьники при этом напишут что-то типа:

```
алг вправо
нач
| если справа стена или x=32
| | то отказ
| | иначе x:=x+1
| все
кон
```

Вы можете привести им более «правильный» алгоритм с проверкой в **дано**:

```
алг вправо
  дано справа свободно и x<32
нач
| x:=x+1
кон
```

После чего пояснить им, чем второй алгоритм лучше первого. Вы можете также сказать, что проверки, находится ли Робот на краю поля, можно всюду исключить, если «обнести» поле по краю сплошной стеной. И попросите их записать соответствующий фрагмент установки начальных значений общих величин исполнителя «Маленький Робот».

И еще раз обратите внимание, что ***вызов*** алгоритма «вправо» ничем не отличается от ***вызова*** команды «вправо» «настоящего» Робота. Для ***использования***, для вызова команд, совершенно все равно, подключен ли исполнитель к ЭВМ реально или же только реализован на алгоритмическом языке. Мы не в состоянии различить такие ***вызовы*** по внешнему виду. Написано «вправо» — значит ЭВМ скомандует «вправо», а будет ли при этом выполнен вспомогательный алгоритм или же подана команда «железному» Роботу, зависит от того, как на самом деле устроена жизнь, т. е. подключен ли к ЭВМ реальный Робот или в ЭВМ введена реализация (информационная модель) Робота.

Исполнитель «И1»

Первый содержательный пример исполнителя в учебнике — исполнитель «И1» из п. 22.3 (с. 181–183). Почему я говорю «содержательный»? Потому что ценность понятия «исполнитель» не в том, чтобы промоделировать Робота, а в том, чтобы в задаче, не имеющей, казалось бы, никакого отношения ни к каким исполнителям, воспользоваться понятием исполнителя, как методом алгоритмизации, т. е. использовать конструкцию **исп**—**кон** для облегчения жизни, как метод решения задачи.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Естественно, наш первый «содержательный» пример на самом деле тоже будет простым, учебным. В реальной жизни потребность в использовании исполнителей возникает только при решении достаточно сложных задач. Мы же пытаемся приводить примеры попроще. Поэтому и исполнители у нас будут упрощенными, учебными.

По этой же причине при изложении темы исполнителей мы отошли от проблемного подхода, иначе нам пришлось бы формулировать слишком трудоемкие «проблемы» и тратить слишком много времени, прежде чем потребность в исполнителях (как конструкциях алгоритмического языка) была бы осознана. В частности, наш первый исполнитель «И1» возникает не в процессе решения задач, а «сам по себе», — он сначала описывается, а уж потом используется. Хотя при реальной работе обычно сначала возникает потребность в исполнителе, формулируется его система команд и проч., и только потом

пишется реализация исполнителя на языке программирования.

Итак, в учебнике (с. 181) приводится пример исполнителя «И1», — абсолютно «чистого» внутримашинного исполнителя, не имеющего отношения ни к каким Роботам, Чертежникам и пр. Название исполнителя первоначально было как-то связано с фразой «***И1 — в поле воин***» (и один в поле воин) из книги «Физики шутят». Впрочем, можно считать, что «И1» — это просто первый исполнитель учебника.

Как и всякий исполнитель, «И1» содержит некоторую информационную модель и алгоритмы-«лепестки». У «И1» три «лепестка», три команды которые можно выполнить, три алгоритма: «запомнить», «число больших» (с ударением на букву «о») и «минимум». Исполнителя «И1» можно использовать, как и любого другого исполнителя, отдавая ему команды. Например, мы можем скомандовать ему «запомни(a)» — и он запомнит число «a»; потом скомандовать «запомни(b)» и т. д. А после того, как запомнено какое-то количество чисел, мы можем скомандовать «число больших (7.5)» — и «И1» выдаст нужный ответ. Или можем с помощью команды «минимум» получить минимальное из запомненных чисел.

Обратите внимание, что:

— с одной стороны, исполнитель «И1» — совершенно «надуманный» (придуманный), абсолютно информационный и не «железный» по сути;
— с другой стороны, я описал его (а, сказать по-правде, и придумал), как некоторое внешнее («железное») устройство — этакий прибор для запоминания чисел. Как обычного «внешнего» исполнителя. Как устройство, про которое совершенно неизвестно, каким образом оно запоминает числа и выполняет другие команды. Я придумал систему команд и описал, что эти команды значат, но я совершенно пока не представляю себе, как «И1» будет устроен, как он будет эти команды выполнять, какого он будет размера и веса, и сколько электроэнергии он будет потреблять. Тем не менее, мы вполне можем уже этим исполнителем ***пользоваться***. И в учебнике, на с. 182 приведен алгоритм А96, который «управляя» исполнителями «Робот» и «И1», подсчитывает число опасных клеток в коридоре.

Как создать исполнителя «И1»

Теперь важный сущностный момент. Предположим, для решения каких-то задач нам нужен исполнитель «И1», т. е. исполнитель, который умеет запоминать числа, а потом выдавать их минимум и отвечать, сколько чисел среди них больше, чем некоторая заданная величина. Что делать, чтобы в действительности выполнить алгоритм А96 и получить ответ? Есть два пути.

(1) Обратиться к инженерам и попросить их изготовить нам такое устройство. Они, видимо, повозятся со своими микросхемами, паяльниками и проводами, соберут нужное устройство, подключат его к ЭВМ, и у нас появится исполнитель И1 (именно таких исполнителей я называю «внешними», «железными», «настоящими»). Это один подход.

(2) Ни к кому не обращаться. Вместо «настоящего» исполнителя И1 написать его реализацию (информационную модель исполнителя) на алгоритмическом языке. Для этого придется придумать, где и как будут запоминаться числа, составить алгоритмы, реализующие команды исполнителя, написать соответствующую конструкцию **исп**—**кон** и ввести все это в ЭВМ. т. е. «научить» ЭВМ делать (моделировать, имитировать) работу исполнителя «И1». ЭВМ — универсальная машина для переработки информации и потому может переработать любую информацию. (А значит, в состоянии сама выполнить любую работу, в которой не требуется что-либо штамповать, резать или иным образом делать что-то в реальном мире.) Это второй подход.

В курсе информатики (и в учебнике) нас, естественно, интересует второй подход. Мы придумали исполнителя «И1», сказали, что он нам нужен, и теперь нам надо его ***реализовать*** (т. е. написать на алгоритмическом языке). А потом им можно будет пользоваться — совсем как «настоящим», и не придется обращаться к инженерам и закупать дорогостоящие материалы.

Поэтому мы будем описывать реализацию (информационную модель) исполнителя И1 на алгоритмическом языке. Она приведена целиком на с. 182–183 учебника. Этот текст от **исп** до **кон** — и есть изготовление исполнителя «И1». Сверху заданы общие величины исполнителя «И1»:

— вещественная таблица «a[1:100]», где будут запоминаться числа (т. е. наш «И1» будет уметь запоминать не более 100 чисел), и

— целочисленная величина «n» — количество реально запомненных чисел.

Далее следует установка начальных значений: $n := 0$, т. е. в начале ни одного числа не запомнено. А потом идут алгоритмы-«лепестки» нашего исполнителя. Их у нас три: алгоритм «запомнить», алгоритм «число больших» и алгоритм «минимум».

Важно, что теперь, когда исполнитель «И1» реализован, мы можем поместить в ЭВМ программу, которая содержит и алгоритм А96, и исполнителя «И1», что позволит нам реально выполнить алгоритм А96. При этом ЭВМ, выполняя «команды» И1, запомнит соответствующие числа в таблице $a[1:100]$, сравнит их с уровнем радиации в конечной точке и выдаст ответ. Таким образом, вместо изготовления реального железного исполнителя И1 мы заставили ЭВМ его проимитировать, изготовили его информационную модель.

Но поскольку при написании алгоритма А96 нет абсолютно никакой разницы между «железным» и «информационным» исполнителем (алгоритм А96 вообще был написан ***до*** того, как мы решили, каким образом изготавливать «И1»), в дальнейшем мы можем говорить не «информационная модель», а просто «исполнитель». Нас совершенно не интересует, работает при этом ЭВМ или специальное устройство, спаянное инженерами. Главное, чтобы выполнялись команды исполнителя. А уж будут ли они выполнятся «прибором», подключенным к ЭВМ, или программой, помещенной в память ЭВМ, — совсем неважно.

Конечно, при ***реализации*** исполнителя надо знать и понимать, как записывается информационная модель исполнителя, что значат и как понимаются и выполняются в ЭВМ различные элементы «ромашки» исполнителя, где и как размещаются общие величины исполнителя, когда они появляются в памяти ЭВМ и когда исчезают из нее, и т. д., и т. п. Но при ***использовании*** исполнителя все это совершенно не должно нас интересовать.

Как я уже говорил, один из методических приемов учебника — показывать и описывать новые понятия со всех возможных сторон. И следующий пункт параграфа (п. 22.5) посвящен описанию того, как ЭВМ размещает общие величины в своей памяти, а также содержит пример их изменения в ходе выполнения алгоритма А96. Важно отметить, что общие величины исполнителей заводятся в памяти ЭВМ ***до начала выполнения*** каких-либо алгоритмов. Также ***до начала выполнения алгоритмов*** выполняются фрагменты, которые я называл «установкой начальных значений», — то, что расположено в конструкциях **исп**—**кон** перед текстами алгоритмов. И только ***после*** этого начинается исполнение «основного» алгоритма. Если внутри алгоритма промежуточные величины появляются при начале выполнения алгоритма и исчезают, когда выполнение алгоритма заканчивается, то общие величины исполнителей существуют все время (появляются до начала выполнения чего бы то ни было и исчезают после окончания всего).

А теперь главное — понятие «исполнитель», как способ алгоритмизации, как метод мышления при решении алгоритмических задач

Исполнителей можно использовать, даже если задача чисто информационная, если никаких исполнителей в постановке задачи нет.

В алгоритме А96 все-таки кроме «И1» фигурировал и наш старый знакомый — Робот. В алгоритме А99 (п. 22.6, с. 185 учебника) решается задача, именуемая в науке «минимаксом»: дана прямоугольная таблица (в А99 — «**вещ таб** $t[1:60, 1:60]$»), в каждом столбце надо найти максимум (это будет 60 чисел) и среди них выбрать минимум.

Эта задача из так называемой «теории игр». Я не буду вдаваться в детали (при желании вы можете найти их в литературе), но, грубо говоря, результат работы алгоритма «минимакс» — это результат игры, когда и вы, и ваш противник играют наилучшим возможным образом: вы стараетесь максимизировать свой выигрыш, а противник стремится его минимизировать.

А теперь забудем про всю эту теорию игр и посмотрим на нашу задачу просто, как на упражнение по программированию. Итак, надо по прямоугольной таблице вычислить «минимакс». Никакого Робота, никаких исполнителей вообще, чисто информационная задача. Можно решать и на Бейсике. Но приведенные в учебнике алгоритмы А99–А100 решают эту задачу с использованием исполнителя «И1»! После нахождения максимума в очередном столбце А99 командует И1 «запомнить», а после запоминания всех максимумов от И1 получается минимум из запомненных чисел, т. е. «минимакс».

Решение, возможно, чуть-чуть искусственное, но обратите внимание, ни в постановке задачи, ни в процессе ее решения, ни в ответе никаких предопределенных исполнителей нет. Мы используем И1 не как нечто заданное, а как привнесенное, придуманное нами в ходе решения задачи. Как метод алгоритмизации. Как достояние нашего знания, нашей головы, независимо от исходной постановки задачи.

Понятие исполнителя стало инструментом алгоритмизации. В отличие от Робота, который был задан извне, был предопределен и выступал, как прием для освоения базовых понятий алгоритмизации, здесь произошла очень существенная замена. Мы используем придуманных нами исполнителей для решения задач, в которых и в помине нет никаких исполнителей, в совершенно информационных, в чисто вычислительных задачах. Используем их так же, как вспомогательные алгоритмы, «циклы» и `если`, как ***средство*** решения задач.

Именно в этом смысле — как средство алгоритмизации — понятие исполнителя является тем четвертым фундаментальным понятием современной информатики, про которое я говорил. Так же, как вспомогательные алгоритмы используются для структуризации действий, исполнители используются для одновременной структуризации совокупности и объектов, и действий, связанных с этими объектами.

Этого понятия — исполнителя, как конструкции языка и как средства алгоритмизации — пока нет в большинстве школьных учебников, поскольку оно появилось существенно позже понятия вспомогательных алгоритмов (подпрограмм), и поскольку реальная потребность в этом понятии возникает только при решении достаточно сложных задач. Хотя, повторю, примерно к 1980 году это понятие вошло в абсолютно все языки программирования («пакет» в Аде, «модуль» в Модуле, «класс» в Симула, «экземпляр объекта» в Smalltalk и т. д.). По-разному называемое, оно, тем не менее, есть во всех современных языках программирования.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ.** Мы называем это понятие «исполнителем» с легкой руки Владимира Борисовича Бетелина, который первоначально использовал слово «исполнитель» при русскозначном описании понятия «класс» в языке «Параллельный Паскаль». Потом, в ходе преподавания курса программирования на мехмате МГУ это слово стало использоваться в его нынешнем значении (см. [ПдМ]). Как всякое удачное слово оно начало распространяться и было задействовано Г.А. Звенигородским для описания устройств вроде Дежурика, Черепашки и пр. при преподавании программирования школьникам (т. е. в том значении, в котором мы его использовали в главе 1). Мы, в свою очередь, заимствовали у Звенигородского термины «предписание» и «система предписаний», использующиеся в курсе программирования на мехмате МГУ вместо перегруженных «команда» и «система команд». Исполнитель, как конкретная конструкция языка программирования, возник также в курсе программирования на мехмате МГУ (см. [ПдМ]) и оттуда, лишь слегка видоизменившись, перекочевал в школьный курс и школьный алгоритмический язык.

Возвращаясь к понятию исполнителя, как инструменту алгоритмизации, я еще раз хочу отметить, что нужда в этом инструменте возникает только при решении достаточно сложных, больших задач. Подобно тому, как для составления алгоритма из двух действий не требуется понятие вспомогательного алгоритма, для составления простых алгоритмов из 10–15 строк не требуется понятие исполнителя. И даже алгоритм из 100 строк можно составить не пользуясь этим понятием.

Поэтому мы не можем продемонстрировать в полной мере ту чрезвычайную важность, которую это понятие играет при решении сложных задач. И в учебнике мы ограничились лишь несколькими словами (п. 22.7), констатирующими, что метод последовательного уточнения с использованием исполнителей — ***основной метод алгоритмизации сложных задач***. Этот пункт (22.7) помечен звездочкой, поскольку не поддержан упражнениями и, соответственно (вспомните «черепаху»!), нормально усвоен быть не может. Тем не менее, поскольку это ***основной*** метод, совсем мы его опустить не могли и привели словесное описание.

Дело в том, что реальные программные системы создаются так.

Во-первых, исходная задача обычно формулируется, как требование создать исполнителя, который будет уметь выполнять некоторый набор команд. (Вместо слова «исполнитель» часто говорят «система».) Например, система продажи билетов должна уметь отвечать на вопросы про проданные и непроданные билеты и наличие рейсов: продать билет и учесть возврат билета, подсчитать сколько всего денег должен сдать кассир в конце смены, выдавать сред-

нюю загрузку рейсов, чтобы можно было принимать решения, на каких направлениях надо вводить дополнительные рейсы, и т. д. и т. п. Получается очень много разных команд.

Во-вторых, основной метод решения состоит в том, чтобы сказать, что «если бы у нас были такие-то исполнители, умеющие делать то-то и то-то, то тогда мы нашу задачу могли бы решить, и соответствующие алгоритмы (и исполнители) записываются так-то». Важно, что в этот момент можно ***придумать*** себе в помощь таких исполнителей, использование которых позволяет достаточно легко решить исходную задачу. Такой переход называется «шаг декомпозиции» или «шаг детализации», он сводит исходную задачу к нескольким более простым задачам, для каждой из которых, в свою очередь, придумываются еще чуть более простые вспомогательные исполнители. И так постепенно, шаг за шагом, все упрощая и упрощая исполнителей, доходят до совсем простых, которые уже можно просто написать на языке программирования.

Обычно при этом происходит еще и разделение труда. Тот, кто отвечает за создание всей системы целиком, делает первый шаг детализации, получает описание, скажем, 10 подисполнителей попроще и раздает их на реализацию 10 разным людям (или даже 10 разным отделам). В этих отделах происходит следующий шаг детализации — и описания под-под-исполнителей раздаются на реализацию подотделам и т. п.

Чтобы вы представили себе о чем идет речь, замечу, что трудозатраты на создание системы продажи авиабилетов «Сирена» оцениваются в 300 человеко-лет. Это значит, что коллектив из 60 человек работал над системой в течение 5 лет или коллектив из 100 человек — в течение 3 лет. Конечно, в это время входит и поиск ошибок, и исправления, и переделки. Но все равно, любая реальная система — это, как правило, большая и сложная работа. Продемонстрировать методы решения таких задач мы можем только на очень упрощенных — школьных — примерах. И именно этому и будут посвящены § 23–27 (применения ЭВМ).

Самое главное в этом методе алгоритмизации с использованием исполнителей то, что при проведении шага детализации люди придумывают не отдельные вспомогательные действия (алгоритмы), а целиком вспомогательных исполнителей. И лишь потом — в ходе их реализации — уточняют, какие в исполнителях будут общие величины и что с этими величинами будет происходить при выполнении команд.

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Вы, наверное, знаете, что в описанной выше картинке кипучей бюрократической (в хорошем смысле этого слова) деятельности (когда начальник расписывает подчиненным, что они должны делать, распределяет между ними работу) в реальной жизни также используются слова «исполнитель», «подисполнитель» (или «субисполнитель») и пр. И это, конечно, не случайно. Это — хорошая аналогия, хорошее пояснение, и оно вам может пригодиться.

Как, например, делается какое-то новое большое дело? Создается организация, директору поручается все сделать, и он начинает думать, кто, какие отделы, люди ему для этого нужны, кто чем должен заниматься и т. п. Дальше директор утверждает «штаты», принимает на работу сотрудников, поручает им дела, и организация начинает как-то функционировать. Слово «исполнитель» в этом смысле взято из обыденной или, если хотите, «организационно-бюрократической» жизни. Мы придумываем, какие нам нужны вспомогательные исполнители, нанимаем их на работу и поручаем им сделать то или иное дело. Они, в свою очередь поручают более мелкие дела своим подисполнителям, отделам и подотделам и т. д. Дальше все это выполняется и результаты передаются вверх по цепочкам к начальству.

Правда, в отличие от обыденной жизни, в информатике все очень строго формализовано — ведь алгоритмы выполняет ЭВМ. Поэтому все команды выполняются ***в точности*** так, как мы их записали. Конечно, мы можем ошибиться при записи алгоритмов, но ЭВМ всегда выполняет именно то, что мы ей написали.

На этой же житейской аналогии я сейчас бегло поясню смысл термина «параллельные процессы». Когда один исполнитель-человек просит что-либо сделать другого, то первому вовсе необязательно сидеть сложа руки и ждать, пока второй завершит свою работу и сообщит результат. В реальной жизни первый человек вполне может заняться чем-то полезным, хотя бы, например, раздавать поручения другим своим подисполнителям.

В информатике же, как вы помните, при вызове вспомогательного алгоритма (в том числе, и при вызове алгоритма из вспомогательного исполнителя) выполнение основного алгоритма приостанавливается и возобновляется только после полного выполнения вспомогательного алгоритма. Ведь все эти алгоритмы и исполнители по-

мещаются в памяти ЭВМ, и процессор выполняет их последовательно — в каждый момент времени выполняется только какой-то один алгоритм.

Принцип последовательного выполнения, однако, не догма. Существует отдельный раздел информатики, посвященный методам организации параллельного (одновременного) исполнения нескольких алгоритмов (это могут быть несколько ЭВМ исполняющих разные алгоритмы, или одна ЭВМ, с несколькими процессорами, или даже одна ЭВМ с одним процессором, переключающимся с выполнения одного алгоритма на выполнение другого и обратно). Каждый такой параллельно (одновременно) исполняемый алгоритм называется ***процессом***, а соответствующая область информатики — теорией «параллельных процессов». Существуют и специальные языки программирования, специальные понятия и языковые конструкции, предназначенные для организации таких процессов (сигналы, флаги, семафоры, критические области, захват и освобождение ресурсов, прерывания и т. п.). Ведь если «начальник», отдав команду исполнителю и не ожидая результата, продолжает заниматься «своими делами», — т. е. при вызове вспомогательного алгоритма выполнение основного не приостанавливается, — то требуются некие специальные системы связи, чтобы подчиненный мог сообщить, что он закончил работу, или чтобы начальник мог поинтересоваться не окончена ли уже порученная работа и каков результат.

В учебнике, впрочем, эта область совершенно не затронута. Учебник посвящен исключительно и только так называемому «последовательному» программированию.

Возвращаясь к учебнику, хочу заметить, что с изучением § 22 мы прошли все четыре фундаментальных понятия информатики, и вышли на «плато», на котором проложено много разных дорог в разные стороны. И теперь мы можем двигаться в любую сторону, углубляться в любую конкретную область информатики. Одна из дорог, например, ведет в сторону «параллельных процессов», другая — в «конструирование типов и структур данных», третья — в «методы алгоритмизации и доказательства правильности программ» и т. д.

В упражнениях после параграфа имеются самые разные задачи как на дополнение и изменение уже заданных исполнителей, так и на реализацию новых. Я обращаю ваше

внимание на упр. 8 (и рекомендую его сделать), где требуется, используя в качестве вспомогательного исполнителя «Чертежник», реализовать исполнителя «Черепашка», система команд которого взята из черепашьей графики в языке Лого.

Теперь мы наконец можем перейти к заключительной части нашего курса, а именно, к применениям ЭВМ. §§ 21 и 22 были подготовкой, необходимой для того, чтобы мы имели язык, на котором можно описывать, ***как*** представляется и обрабатывается информация в тех или иных применениях ЭВМ. Соответственно, эти применения (§§ 23–27) можно рассматривать, как примеры, поясняющие и углубляющие понятия, изложенные в параграфах 21–22. Никаких новых понятий больше в курсе вводиться не будет. Остались только «применения» уже изученных понятий к разным областям использования ЭВМ и изложение некоторых специфических методов обработки информации в конкретных областях (как, например, метод дискретизации непрерывных процессов в § 26).

Лекция 11. Применения ЭВМ

Теперь, усилив наш алгоритмический инструментарий понятиями информационной модели и информационной модели исполнителя (конструкция **исп**—**кон**), мы наконец можем заняться собственно применениями ЭВМ, которым посвящены §§ 23–27. Наиболее содержательными из них являются §§ 23, 24 и 26. Причем § 23 — самый содержательный и самый важный: в нем ярче всего показано применение введенного нами только что понятия исполнитель при решении конкретных прикладных задач. Да и материал — информационные системы — достаточно простой и благодатный.

§ 23. Информационные системы

Информационные системы — это системы, предназначенные для хранения, поиска и обработки той или иной информации. Как и всюду в этой части (§ 23–27), параграф начинается с беллетристики, с описаний конкретных систем.

Первым приводится описание системы Экспресс для продажи железнодорожных билетов с указанием реальных данных (1990 года) про количество задействованных мегабайтов памяти, число продаваемых за сутки билетов и пр. Эти фактические сведения, как мне кажется, достаточно интересны (во всяком случае, я с большим интересом все это узнал, когда мы получали информацию в процессе написания учебника). Узнать, сколько памяти нужно для представления информации обо всей железнодорожной сети и обо всех поездах страны — это достаточно интересно.

Итак, в начале параграфа описываются система Экспресс, система управления конвейером волжского автозавода, система учета междугородных переговоров, и — чуть-чуть — базы данных. Естественно, все это — очень компактно.

А потом, начинается содержание, т. е. изложение методов представления и обработки информации в описанных системах. Конечно, «настоящую» информационную систему нам здесь не описать — ведь это десятки и сотни тысяч строк на алгоритмическом языке. Но мы воспользуемся принципом «корыта» (а также нашим методическим приемом «очистки от всего лишнего») и покажем школьникам учебные информационные системы, очень простые, но поясняющие, как могут быть в принципе реализованы описанные выше системы («объясним, почему оно плавает»).

В параграфе рассматривается две такие учебные системы.

Первая — прямой учебный аналог «Сирены» — учебно-информационная система продажи билетов в один купейный вагон, следующий без остановок.

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. При изложении материала младшеклассникам (что вполне возможно) требуются образные постановки задач. И если задачи про Робота могли начинаться словами «В некотором царстве, в некотором клетчатом государстве жил-был Робот», то материал про применения ЭВМ мы планировали изучать на примере сказочной страны «Лилипутии», где все очень маленькое. В этой сказочной стране живет меньше тысячи лилипутов (но у каждого есть телефон); там всего два города, между которыми совсем недавно лилипуты проложили железную дорогу и пустили один самодвижущийся купейный вагон (рис. 13).

Рис. 13.

Лилипутия, однако, очень прогрессивная страна и лилипуты, чтобы быть в ногу со временем, решили компьютеризировать свою жизнь — продавать билеты в свой единственный вагон, учитывать, у кого какой номер телефона, и решать другие задачи с помощью компьютеров. Но на беду лилипуты пока не проходили информатику, поэтому они решили позвать нас (точнее, школьников) и попросили написать им все их «лилипутские» информационные системы. И вот теперь наша задача —

помочь лилипутам, реализовать для них на алгоритмическом языке информационную систему для продажи билетов «Вагон» (п. 23.5 учебника), систему для учета абонентов лилипутской телефонной сети «Телефонная книжка» (п. 23.6 учебника) и т. п.

В порядке разрядки такое изложение может быть проведено и в старших классах. Но для старшеклассников мы пустили наш единственный вагон без паровоза и без других вагонов по вполне взрослому маршруту «Москва-Петушки». Итак, простая учебная система «Вагон», которая должна уметь продавать билеты в этот вагон, узнавать, продано или не продано место, продать два места в одном купе и т. д. Маленькая учебная модель, перефраз системы Экспресс.

Естественно, раз мы собираемся продавать билеты, то нам нужна информационная модель вагона (для класса задач, связанного с продажей билетов), а также исполнитель, который будет хранить эту информацию и выполнять команды типа «продай место», «верни билет» и пр. Все это вместе, в соответствии с § 22, мы назовем исполнителем «Вагон». Модель, которая тут получается (п. 23.5), оказывается даже проще, чем модель кинозала. В отличие от кинозала (и именно для того, чтобы показать, что проданные места можно задавать по-разному), здесь вместо логической таблицы используется целочисленная таблица «М[1 : 36]» (М — от слова «место»: М[i] — это i-е место). Если М[i] = 1 — место продано, если М[i] = 0, — свободно.

Обратите внимание, что в учебнике не приведен фрагмент задания начальных значений таблицы «М[1 : 36]». Вы можете попросить школьников либо написать соответствующий фрагмент сразу после описания величины «М», либо реализовать команду (алгоритм) «начать работу» (некоторым больше нравится название «очистить вагон»), которая должна устанавливать, что все места свободны.

Из других содержательных элементов обратите внимание на **дано** алгоритма А102. Записанное в **дано** условие «$1 \le i \le 36$ **и** М[i] = 0» — абсолютно формальное и содержательное условие на алгоритмическом языке. Это значит, что всякий раз перед началом исполнения тела алгоритма А102 «учесть продажу места» ЭВМ проверит, что условие «$1 \le i \le 36$ **и** М[i] = 0» истинно, т. е. что i-ое место существует и свободно. Если условие окажется невыполненным, то значит, мы где-то при написании основного алгоритма

допустили ошибку — возникнет отказ и надо будет менять программу.

Все остальное здесь — проще некуда. Такими алгоритмами можно было заниматься с самого начала они вполне доступны даже младшеклассникам. Вы можете как угодно разнообразить их, требуя продать два места в одном купе или место на нижнюю полку и т. п. Соответствующие задачи сформулированы в упражнениях, и, — вспомните «черепаху» — именно в процессе решения таких задач школьники должны действительно («через делание») усвоить и понятие исполнителя, и простейшие методы представления и обработки информации в информационных системах. Вы можете легко повысить сложность этой задачи, усложняя представляемый объект, например, предусмотрев возможность бронировать места, введя разные типы вагонов (плацкартные, общие, вагон-ресторан и пр.), промежуточные станции и возможность продавать билеты на промежуточных станциях, и т. п.

Реальная система продажи билетов «Сирена» достаточно сложна, но в принципе устроена так же. Изучение учебной системы «Вагон» и решение задач в рамках этой системы формирует достаточно глубокое и адекватное представление о том, как устроены настоящие системы. Именно это я и имел в виду, когда говорил, что наша задача показать не только «что» могут делать ЭВМ, но и «как» они могут это делать.

И еще один повтор — обратите внимание, насколько естественна здесь формулировка «реализовать исполнителя», насколько естественна реализация алгоритмов не самих по себе, а в рамках заданного исполнителя. В реальной жизни крайне редко требуется написать один отдельно взятый алгоритм. Как правило, задачи формулируются, как задачи создания программных систем (исполнителей) с некоторым набором команд.

Вторая учебно-информационная система учебника — система «телефонная книжка» (п. 23.6). Если не говорить о Лилипутии, то проблему можно сформулировать так: реализовать систему для хранения информации об абонентах телефонной сети в небольшом поселке (меньше 1000 телефонных номеров). Это значит, что надо уметь отвечать на вопросы типа «Какой телефон у Иванова Петра Ивановича?», или наоборот, «Кому принадлежит номер 555?». Надо уметь получить свободный номер для установки новому

абоненту или поменять номер, если он уже кому-то надоел. Такой вот обычный набор действий.

Приведенная в учебнике информационная модель состоит из целочисленной величины n, в которой хранится число телефонов в поселке, и литерной таблицы «ФИО[1:999]», в которой хранятся фамилии, имена и отчества абонентов. В тексте параграфа можно было бы обойтись и без величины n: эта величина добавлена в модель, чтобы удобно решался ряд упражнений. Алгоритмы исполнителя, реализующие команды установки телефона и смены номера также достаточно просты, и вы их посмотрите сами.

Алгоритмы поиска

Я же хочу обратить ваше внимание на алгоритм «справка о номере телефона» и на вызываемый из него вспомогательный алгоритм «поиск абонента». «Поиск абонента» — именно то, ради чего приведен в учебнике весь этот исполнитель. Дело в том, что задача поиска является очень частой задачей в информатике. Настолько частой, что вначале мы даже хотели посвятить таким задачам отдельный параграф в алгоритмической части (в главе 1). Потом, однако, мы решили ограничиться алгоритмом поиска в рамках темы «Информационные системы» — это алгоритм А109 и ряд упражнений после параграфа.

Напомню, что информационные системы — это системы, предназначенные для хранения, изменения и поиска информации. Как хранить большие объемы информации, мы уже знаем — с помощью таблиц и информационных моделей. Изменение информации достаточно продемонстрировано в системе «Вагон». А вот поиск — одна из ключевых функций информационных систем — демонстрируется (в простейшем виде) именно алгоритмом «поиск абонента» (А109).

Алгоритм А109 учебника (с. 192) — простейший. Это так называемый «последовательный» поиск, когда мы просто перебираем элементы таблицы один за другим, пока не обнаружим нужный. Алгоритмы поиска — это отдельная тема информатики, которой вполне можно посвятить полгода или год занятий на факультативе. Здесь же мы лишь обозначили эту тему.

Если вы желаете затронуть ее хоть чуть более глубоко, можете воспользоваться упражнением 4.б), в котором требуется, чтобы абоненты в таблице ФИО были упорядочены не по номерам телефонов, а по именам, как в обычных телефонных книжках: сначала — все на «А», потом — на «Б» и т. д. В этом случае индекс таблицы уже нельзя использовать, как номер телефона, и поэтому в соответствующей модели в упр. 4 появляется еще одна таблица, содержащая номера телефонов абонентов. В этой модели вы можете попросить учеников реализовать поиск абонента не последовательным перебором, а методом деления пополам: сначала взять середину заполненной части таблицы ФИО и посмотреть, левее или правее этой середины находится фамилия искомого абонента (ведь элементы таблицы упорядочены по алфавиту!), потом, соответственно, взять середину левой или правой части — и т. п. Этот — бинарный — алгоритм поиска, а также разные другие алгоритмы поиска вы можете найти почти в любых книжках по программированию, в частности, в [ПдМ].

Заключение

В упражнениях после параграфа есть стандартные, требующие дописать или изменить что-то в исполнителях «Вагон» и «Телефонная книжка», Наиболее интересным из них, помоему, является упр. 5. Но я хочу обратить ваше внимание на мое любимое упр. 6 на с. 193 учебника — хранение информации о владельцах автомобилей и поиск всех владельцев белых «Волг», в номере которых есть буква «М» и цифра «7». У школьников тут возникает масса всяких гипотез (шпионских и прочих) о происхождении задачи, и поэтому обычно ее решают с большим интересом. Вы можете и сами формулировать эту задачу вполне жизненным образом: «Поступила информация, что напротив нашей школы белая «Волга» сбила человека и скрылась. Свидетели запомнили только, что в номере есть буква «М» и цифра «7». Надо найти все такие машины, чтобы их проверить.»

Итак, резюме. Параграф состоит из двух частей. Вначале идет беллетристика (в хорошем смысле этого слова), т. е. описание конкретных областей применения информационных систем, демонстрирующих, «что» могут ЭВМ. Потом, во второй части, излагаются учебные информационные системы, в очень упрощенном виде показывающие, «как» хранится, изменяется, ищется информация в информационных системах. С алгоритмической точки зрения, материал здесь

очень простой, — по-моему, гораздо проще цикла **пока** — и вполне доступен для изучения, начиная с 6–7 класса.

Повторю, что § 23 является фундаментальным с точки зрения описания применений ЭВМ: он посвящен непосредственно хранению, изменению, переработке и поиску информации, т. е. практически предмету информатики (вспомните определение информатики на с. 3 учебника). По этой же причине он наилучшим образом иллюстрирует введенное в § 22 понятие исполнителя, как конструкции алгоритмического языка.

Поэтому, я вам рекомендую в случае нехватки времени лучше пропустить остальные параграфы про применения ЭВМ (§§ 24–27), но в полном объеме пройти § 23.

§ 24. Обработка текстовой информации

При работе в системе «КуМир» нет необходимости пользоваться каким-то отдельным текстовым редактором — соответствующие возможности встроены непосредственно в систему «КуМир».

Но существуют системы, в которых создание текста программы отделено от его обработки и выполнения. Кроме того, многие люди используют ЭВМ для редактирования обычных текстов, не имеющих отношения ни к каким алгоритмам и программам, например, для ввода и изменения писем друзьям или текстов научных статей по географии. Говорят, что (если исключить компьютерные игры) редактирование текстов составляет чуть ли не 80% всех применений ЭВМ в мире. Поэтому наш второй (после «чисто» информационных систем) параграф посвящен именно обработке текстовой информации.

Естественно, сначала идет беллетристика под названием «Системы обработки текстов», где говорится, что тексты можно обрабатывать на ЭВМ, и это хорошо. Что можно исправить только неправильное место и напечатать весь текст заново. И т. д. Поскольку все вы работали с теми или иными системами обработки текстов (текстовыми редакторами), вам легко рассказать школьникам об используемой вами системе, показать ее и дать поработать. Это — вещи, известные всем.

Задача, которую перед собой поставили мы, — показать, как устроен внутри, как реализуется современный экранный текстовый редактор. Создать у ребят представление, что происходит, когда они нажимают на клавиши, почему текст на экране начинает смещаться и пр. Здесь мы снова от вопроса, «что» может делать текстовый редактор, переходим к вопросу «как» он это делает.

Для этого вначале (п. 24.2) вводятся понятия «текст», «окно», «курсор». А потом (п. 24.3) рассматривается реализация учебного редактора текстов.

Я не буду повторять материал учебника. Отмечу лишь, что «курсор» в п. 24.2 соответствует системам обработки текстов, в которых позиция в тексте (курсор) отмечается квадратиком поверх символа (как в «КуМире» и «МикроМире») или черточкой под символом. Если в вашей системе обработки текстов курсор — вертикальная черточка ***между*** символами, то в п. 24.2 и 24.3 следует внести соответствующие изменения, либо продемонстрировать школьникам си-

стему, в которой «курсор» будет совпадать с «курсором» из п. 24.2 учебника.

Кроме того, я обращаю ваше внимание, что в п. 24.2 ошибочно указана ссылка на фото 18 вклейки. Здесь должна быть ссылка на фото 5 вклейки, на котором изображено много «окошек» с разными текстами. Впрочем, это — фото, сделанное с экрана «Ямахи». Если есть возможность, то вместо него лучше продемонстрировать любую современную многооконную систему, например Windows.

После введения всех трех понятий в п. 24.3 разбирается учебная модель редактора текстов, рассчитанная на 100 символов в строке и текст максимум из 200 строчек. Эти параметры, впрочем, без труда можно заменить на другие. Для хранения и представления текста, а также окна и курсора нам необходимо построить информационную модель этих понятий, задать их величинами алгоритмического языка.

В качестве информационной модели текста на с. 195 учебника задается символьная таблица «T[1:200,1:100]», а также целочисленная величина N, в которой будет хранится реальное число строк в тексте, поскольку текст может содержать и меньше, чем 200 строк.

Информационная модель окна — это две величины «**цел** YO, XO», содержащие смещение окна от верхнего (YO) и левого (XO) краев текста. (Ширина и высота окна считаются фиксированными.)

Информационная модель курсора — также две величины «**цел** Yk, Xk», задающие координаты курсора внутри окна, т. е. смещение курсора от верхнего (Yk) и левого (Xk) крев окна.

Таким образом, YO — это количество строк над окном, XO — количество символов в строке левее окна, Yk — номер строки с курсором в окне (считая сверху), Xk — номер позиции курсора в окне по горизонтали (считая по символам слева).

Все это вместе взятое — **сим таб** T[1:200,1:100], **цел** N, YO, XO, Yk, Xk — и есть информационная модель текста, окна и курсора.

Теперь, имея информационную модель, мы можем начинать писать алгоритмы исполнителя «Простейший редактор». Например, перемещение курсора вправо на один символ или алгоритм вставки пустой строки. А также любые другие, обычные для текстового редактора команды.

В учебнике реализованы алгоритмы (команды) «вниз на строку» и «удалить символ». Еще некоторое количество команд сформулировано в упр. 1 (с. 198 учебника).

Обратите внимание, что приведенные в учебнике алгоритмы меняют только информационную модель текста, окна, курсора. Чтобы изменения в тексте отобразились на экране компьютера, нужно провести отдельную работу — составить соответствующий алгоритм (алгоритмы А112 и А113 учебника). Здесь приведено простейшее решение — переизображение всего экрана целиком. Обычно в экранных редакторах меняют только часть экрана, в зависимости от того, какое действие было выполнено. Мы же — для простоты — написали общий алгоритм изображения экрана целиком. Этого хватает для демонстрации принципов и может оказаться достаточно для реальной работы, если у нас быстрый компьютер. Если же компьютер медленный, то при выполнении каждой команды редактора будет видно, как переизображается экран.

Конечно, написать команды «вниз на строку», «удалить символ» и даже алгоритм переизображения экрана — еще не достаточно. Ведь мы хотим реализовать текстовый редактор целиком. Поэтому надо еще написать основной алгоритм, который будет ожидать нажатия человеком на клавишу; после нажатия — анализировать, какая именно клавиша нажата и, в зависимости от клавиши, вызывать соответствующую команду (например, «удалить символ»); а после всего этого — еще и переизображать состояние информационной модели на экране (вызывать алгоритм А112 «переизобразить экран и курсор»).

Этот основной алгоритм в учебнике не приведен, а лишь описан (примерно как я его сейчас описал) на с. 197. Мы не стали приводить его по следующим причинам. После ввода клавиши в этом алгоритме идет огромный выбор, в котором последовательно перебираются разные коды клавиш, и если была нажата клавиша с данным кодом, то вызывается соответствующая команда редактора. Например, если код нажатой клавиши равен 12 (это код стрелки влево), то вызывается команда «влево на символ». Другими словами, здесь должны быть выписаны все коды клавиш. А они на разных компьютерах разные и в любом случае требуют каких-то пояснений. Поэтому мы основной алгоритм приводить не стали.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. В комплект поставки системы «КуМир» входят и «Вагон», и «Телефон-

ная книжка», и учебный экранный редактор. Поэтому, хотя в самом учебнике текст основного алгоритма не приведен и школьникам его составлять не нужно, посмотреть этот текст на экране в системе «КуМир» они могут. При реальном добавлении новой команды в редактор им надо будет подправить и основной алгоритм. Таким образом, внутренности экранного редактора будут им предъявлены полностью. С этим можно разобраться, понять, как редактор работает, и даже самому добавить в редактор те или иные команды. В частности, составив очередной алгоритм из упр. 1 (с. 198 учебника) и добавив его в исполнитель «Простейший редактор», школьник может превратить редактор, который только что не понимал, например, команду «удалить конец строки» (упр. 1.б), в редактор, понимающий эту команду, и затем немедленно проверить, как она будет работать и пр.

§ 25. Научные расчеты на ЭВМ

Этот параграф информационных моделей не содержит и посвящен исключительно методам алгоритмизации при решении конкретных вычислительных задач. Если не говорить о § 16 (который слишком многоплановый, чтобы его с чем-то сравнивать), то § 25 является самым математизированным параграфом учебника. Его можно пропустить, если уровень математической подготовки учеников недостаточен, например, если речь идет о 7 классе. Хотя учебник предназначен для 10–11 классов, по нему часто преподают в 7, 8 или 9 классах. Это сейчас уже достаточно распространенная практика. Мы в учебнике постарались весь математизированный материал локализовать (это § 16, § 25 и, в меньшей степени, § 26), и эти параграфы можно пропустить при преподавании в младших или ослабленных классах.

§ 25, как и любой другой параграф «Применений», начинается с беллетристики, описывающей специфику этой области применений ЭВМ. В данном случае рассказывается про томографию, т. е. получение «срезов» человеческого тела. Этот пример вы найдете и в книге [ЗК]. Впрочем, есть и другие примеры, требующие огромных вычислений. Например, вы можете рассказать (в учебнике этого нет), что для запуска ракеты на нужную орбиту необходимо учитывать не только сопротивление воздуха, но и то, что это сопротивление падает с набором высоты. Надо учитывать изменение веса ракеты по мере расхода топлива. Надо учитывать не только притяжение Земли и его изменение, но и вращение Земли, силу Кориолиса. И еще целый ряд факторов.

В отличие от предыдущих параграфов, раздел «как» этого параграфа практически не имеет никакого отношения к первой части (к беллетристике). Математические основы, используемые для построения картинки плотности тела в томографе или для точного расчета запуска ракеты на нужную орбиту, слишком сложны и громоздки для школьного учебника. Поэтому здесь разбираются традиционные методы приближенных вычислений: вычисление корня функции методом деления отрезка пополам (п. 25.4), вычисление интеграла методом трапеций (п. 25.5) и метод Монте-Карло (п. 25.6 и 25.7). Изложение первых двух методов практически дословно повторяет соответствующий материал из учебников [Ершов]. (Тогда этот материал рассматривался как основной, поскольку все изложение строилось на математическом материале.)

Я думаю, что все это вам хорошо знакомо и не нуждается в повторениях.

Метод Монте-Карло

Третий метод — метод Монте-Карло. Хорошее изложение этого метода вы можете найти в учебнике [Гейн]. Там же имеется замечательный образ, объясняющий суть метода: представьте себе, что на квадратной детской площадке расположена песочница неправильной формы. Представьте себе далее, что прошел снегопад и выпало много снега. Понятно, что количество снега (снежинок), попавшего в песочницу, будет относится ко всему снегу, выпавшему на детскую площадку, так же, как площадь песочницы относится к площади всей площадки. Поэтому для подсчета площади песочницы мы можем подсчитать, какая часть снега попала в песочницу, и умножить эту величину на площадь всей площадки.

В нашем учебнике фигурка на плоскости (внутри квадрата) засыпается ровным слоем «песка» и считается, сколько «песчинок» попало внутрь фигурки. Конечно, и «снег», и «песок» — не более, чем образы. Важно, что «снежинок» («песчинок») много и что они покрывают весь квадрат (детскую площадку) равномерно.

При вычислениях на ЭВМ «снег» («песок») надо проимитировать, т. е. следует случайным образом «бросать» точки («снежинки», «песчинки») в квадрат так, чтобы в целом они оказались равномерно распределенными по квадрату. Для этого у нас используется специальная функция алгоритмического языка «rnd» (от англ. random — случайный).

Тогда, если мы умеем про каждую точку быстро вычислить, внутри она или снаружи нашей фигурки (т. е. площадь фигурки считать не умеем, а определить, находится точка внутри или снаружи фигурки, можем), — то можно поступить следующим образом. «Накидаем» достаточно много точек и подсчитаем, какая часть (k) из них попала внутрь фигурки. Если точек достаточно много и они распределены равномерно, то площадь фигурки можно приближенно вычислить, умножив k на площадь квадрата. Соответствующий алгоритм — алгоритм А117 — приведен в учебнике на с. 204.

В пункте 25.7 этот метод применяется для приближенного вычисления числа π. Предположим, что мы не знаем, чему равно π, но знаем, что площадь круга равна πR^2. Мы можем

взять круг единичного радиуса (его площадь равна π) и воспользоваться методом Монте-Карло, т. е. накидать случайных точек («снежинок», «песчинок») и подсчитать приблизительно площадь круга (т. е. число π). Соответствующий алгоритм А118 приведен в учебнике на с. 204. На с. 205 даны результаты выполнения этого алгоритма, где при 5000 точек π получается равным уже 3.1424. Естественно, если мы выполним алгоритм еще раз, то при том же количестве точек (5000) число π может получиться другим, ведь точки «бросаются» случайно и результаты каждый раз могут меняться.

Специфика вычислений на ЭВМ

Самое важное в этом параграфе — п. 25.3 на с. 200 — специфика вычислений на ЭВМ и их отличие от того, что принято в математике.

Главное, что надо понять, — это то, что машина считает:

а) приближенно — мы получаем не точный ответ, а приблизительный;
б) за счет того, что мы считаем приближенно, мы можем подсчитать что-то, не имея точных формул и не выводя их; это — совершенно другой, отличный от школьной математики, подход к решению задач.

Ни для томографа, ни для запуска ракеты невозможно написать ***математическую формулу*** получения результата. Например, при расчете запуска ракеты, невозможно написать формулы, выражающие местонахождение и скорость ракеты через начальные величины (вес топлива, направление старта и т. д.), с учетом вращения Земли, изменения сопротивления воздуха, изменения массы топлива и пр. Для получения ответов в обоих случаях требуются вычисления — миллионы и миллиарды операций.

Имеется одна очень распространенная ошибка, часто встречающаяся при преподавании информатики, когда учитель приводит, например, формулу корней квадратного уравнения, а потом просит составить по этой формуле алгоритм вычисления корней. Естественно возникает вопрос, зачем это надо. Ведь по формуле можно и так подсчитать. Информатика при этом как бы лишается своего предмета, спускается до уровня умения пользования калькулятором. Конечно, и калькулятором надо уметь пользоваться. Но не может же это быть предметом школьного курса.

Здесь очень важно подчеркнуть, что мы можем использовать компьютер для нахождения величин, для которых мы никаких формул не знаем, и что алгоритмы при этом могут получаться достаточно простые.

Возьмем хотя бы нахождение корня уравнения $\cos(x) = x$ методом деления отрезка пополам (п. 25.4 учебника). Мы не знаем и не можем вывести ***никакой*** формулы для корня этого уравнения. В курсе школьной математики такую задачу вообще решить нельзя. А приближенно, можем подсчитать корень с любой заданной точностью. Именно на этот нюанс надо обратить внимание: вычисления на ЭВМ не точные, а приближенные, но зато мы их проводим, не имея точных формул, не имея математического аппарата для нахождения корней, интегралов или площадей и т. д. Важно, что это — другая область, не охватываемая обычной математикой, обычными формулами.

§ 26. Моделирование и вычислительный эксперимент на ЭВМ

Это третий (после 23 и 24) содержательный параграф про применения ЭВМ. Начинается он, как всегда, с беллетристической части, на этот раз совсем небольшой, которая, фактически, служит просто введением. Единственная мысль этого введения состоит в том, что всегда, когда нам нужна информация о некотором процессе или явлении, мы можем попытаться составить информационную модель этого явления, проанализировать ее на ЭВМ и получить желаемую информацию. Особенно это применимо к таким явлениям, как исследование последствий взрывов, катастроф, влияния жестких излучений и пр., которые «в реальности» лучше не проводить. Что будет с Землей, если озоновый слой исчезнет? Мы не можем для ответа на этот вопрос провести полномасштабный эксперимент. Но можем попробовать смоделировать интересующие нас явления на ЭВМ и просчитать возможные последствия.

Вообще говоря, все, что мы делаем на ЭВМ, — это всегда моделирование. Но в данном случае, речь идет о более конкретных, физических задачах. Моделирование физических явлений (вычислительный эксперимент на ЭВМ) рассматривается в учебнике на примере падения парашютиста с учетом сопротивления воздуха. Вы можете в качестве преамбулы к параграфу повторить также эссе про расчет запуска ракеты на нужную орбиту (с. 301 пособия), где надо учитывать сопротивление воздуха, изменение этого сопротивления и уменьшение массы ракеты с ростом высоты и т. п.

Расчет запуска ракеты — сложная задача. А вот падение парашютиста с нераскрытым парашютом с учетом сопротивления воздуха — задача, с одной стороны достаточно простая, а с другой — привычными школьными физическими формулами не описываемая.

Формулировка этой задачи обычно вызывает оживление, поскольку, фраза «падение парашютиста с нераскрытым парашютом» естественно предполагает печальный конец. Потом в упражнениях (упр. 1 и 2 на с. 211) появятся более «щадящие» парашютиста формулировки, когда парашютист падает с нераскрытым парашютом первые 700 м, а потом парашют все-таки раскрывает. Но это в упражнениях. А в тексте параграфа мы исследуем падение с нераскрытым парашютом до самой земли.

Итак, парашютист падает с нераскрывшимся парашютом. Наша задача — промоделировать этот процесс на ЭВМ. Самое главное, существо всего параграфа здесь — это метод дискретизации непрерывных процессов (п. 26.1). Все содержание параграфа — ровно один метод, метод дискретизации непрерывных процессов.

Что значит «промоделировать» на ЭВМ процесс падения? Такого у нас еще не было. Мы моделировали кинозал, транспортную сеть и прочие ***объекты***, — это понятно. А что значит ***модель процесса*** падения? Как промоделировать на ЭВМ такой непрерывный (плавный) процесс? Ответу на этот вопрос и посвящен § 26.

Как всегда, когда мы хотим пояснить что-то новое, лучше всего использовать аналогию с чем-нибудь хорошо знакомым. Тогда, даже если полного понимания и не появляется, то, во всяком случае, появляются иллюзия понимания, успокоенность и возможность решать задачи. В данном случае в качестве такой аналогии для метода дискретизации непрерывных процессов используется киносъемка.

Дискретизация — это замена непрерывного, плавного процесса последовательностью его состояний. Представьте себе, что мы засняли падение бедного парашютиста на кинопленку (обычную старинную кинопленку, которую можно рассматривать на свет, в которой один кадр сменяет другой). Такая кинопленка — это некоторая информация о процессе падения. Если ее прокрутить на киноаппарате, то процесс падения можно воссоздать на экране, причем с вполне приличной точностью. Но если рассмотреть саму кинопленку как объект, то она никакого плавного процесса не содержит, а состоит из последовательности кадров. На одном кадре — парашютист в одной точке, на другом — в другой, а между ними ничего нет. Плавный процесс падения парашютиста оказался заменен на последовательность его состояний. Этот переход и называется дискретизацией непрерывных процессов.

Как и в примере с кинопленкой, при дискретизации непрерывных процессов, как правило, состояния процесса фиксируются с некоторым шагом по времени, например, с шагом 0.01 с. Мы можем считать, что снимали кино с частотой 100 кадров в секунду. Тогда два соседних кадра на кинопленке — это два состояния процесса, отличающиеся друг от друга на 0.01 с. по времени. А непрерывный процесс падения в итоге заменен на последовательность его состояний с шагом 0.01 с. по времени.

Переходя теперь от аналогии с кинопленкой к собственно моделированию на ЭВМ мы должны отметить две вещи

(1) Мы не можем «заснять» процесс падения — ведь никакого реального процесса у нас нет. Мы, наоборот, хотим промоделировать этот процесс на ЭВМ.
(2) Мы не умеем моделировать на ЭВМ непрерывные процессы, но можем попытаться рассчитать последовательность состояний парашютиста с некоторым шагом по времени. При этом в «состояние» парашютиста мы можем включить любую интересующую нас информацию: высоту, скорость, ускорение и пр. Каждое состояние парашютиста в алгоритмическом языке можно задать набором значений соответствующих величин алгоритмического языка. Последовательные состояния парашютиста могут получаться как последовательные состояния (значения) этого набора величин в ходе выполнения некоторого алгоритма.

Таким образом, метод дискретизации непрерывных процессов как метод алгоритмизации, состоит в том, что:

(1) мы представляем (воображаем) себе непрерывный процесс как некоторую дискретную последовательность его состояний, обычно с каким-то шагом по времени;
(2) далее мы задаем информационную модель (набор величин) для описания одного, отдельно взятого ***состояния*** процесса;
(3) наконец, мы пишем алгоритм, в ходе выполнения которого происходит переход из одного состояния в другое (т. е. меняются значения величин информационной модели) и вычисляются интересующие нас характеристики всего процесса в целом.

Обратите внимание, что информационная модель (как набор величин алгоритмического языка) при этом строится для представления ***одного состояния***, а не всего моделируемого процесса. Моделью самого процесса можно считать ***ход выполнения алгоритма***, который переводит значения величин от одного состояния к другому, как бы воспроизводя (моделируя) процесс.

Самым главным в методе дискретизации непрерывных процессов является то, что переход от одного состояния к другому в этом методе производится дискретно — «скачком». Другими словами, по набору значений для одного состояния вычисляется набор значений в следующем состо-

янии. А это уже хорошо нам знакомое, обычное явление в информатике.

Возвращаясь к модели процесса падения, состояние парашютиста можно задать моделью М19 (с. 208 учебника), т. е. тройкой величин (t, h, v), задающих время, расстояние до земли и скорость парашютиста. Для перехода к следующему состоянию нужно выразить новые значения этих величин через предыдущие:

$$\begin{aligned} t_{\text{новое}} &= t_{\text{старое}} + dt \\ h_{\text{новое}} &= h_{\text{старое}} - v_{\text{старое}} \times dt \\ v_{\text{новое}} &= v_{\text{старое}} + a_{\text{старое}} \times dt \end{aligned}$$

где dt — шаг по времени. Надо понимать, что при написании этих формул мы считаем, что между последовательными состояниями нашей модели (между соседними кадрами кино) ничего не происходит, а значит, ничего и не меняется. На протяжении всего шага по времени (например, 0.01 с.) ни время, ни высота, ни скорость не меняются. Потом, по истечении этого шага по времени, время, высота и скорость меняются «скачком»: время сразу увеличивается на шаг по времени, высота и скорость также скачком устанавливаются в новые значения. Затем, на протяжении следующего интервала времени, опять ничего не меняется. И т. д.

Вопрос: А где формула для $a_{\text{новое}}$?

Ответ: По условиям нашей задачи $a = g - kv^2$, т. е. ускорение однозначно определяется по значению скорости. Эта формула носит «абсолютный» характер, верна для любого момента времени и не зависит от «старых» и «новых». Поэтому $a_{\text{старое}} = g - kv^2_{\text{старое}}$, $a_{\text{новое}} = g - kv^2_{\text{новое}}$. А поскольку ускорение вычисляется через значение скорости, величина для ускорения в информационную модель М19 не входит.

Написанные выше соотношения, выражающие новые значения (t, h, v) через их старые значения, — это и есть (в терминологии § 16) рекуррентные соотношения, задающие последовательность состояний в моделируемом процессе падения парашютиста. Соответствующий алгоритм далее записывается очевидным образом (алгоритм А120 на с. 209 учебника).

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Здесь используется ссылка на метод рекуррентных соотношений из § 16. Если при прохождении курса вы § 16 пропустили и метод рекуррентных соотношений не проходили, то алгоритм А120 все равно можно изложить (он очень прост), но вместо ссылки на метод рекуррентных соотношений надо соответствующий материал (про «исчезновение индексов») повторить здесь явно.

ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Несколько слов для вас, как учителей, которые должны знать больше, чем ученики. Если нарисовать графики функций $h(t)$ и $v(t)$, задающих высоту и скорость парашютиста в зависимости от времени, то для для реального процесса падения это будут плавные (непрерывные) кривые. В нашей дискретной модели эти кривые ***аппроксимируются*** (т. е. задаются приближенно) с помощью ступенчатых функций, которые постоянны на протяжении шага по времени, потом скачком меняются и опять постоянны на протяжении следующего шага (рис. 14). С математической точки зрения «дискретизация» — это замена непрерывной функции такого рода ступеньками.

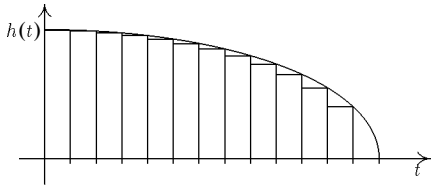


Рис. 14.

На этом собственно изложение метода дискретизации непрерывных процессов заканчивается и остаток параграфа посвящен двум небольшим уточнениям (разъяснениям), касающимся этого метода, а именно: сравнению приближенного и точного решений (п. 26.4) и выбору величины шага по времени (п. 26.5).

Сравнение приближенного и точного решений

Если сопротивление воздуха отсутствует ($k = 0$), то получается обычная задача из школьного курса физики, для которой можно написать точные формулы, вычислить все положения «парашютиста» точно. В то же время мы можем

подставить $k = 0$ в наш алгоритм А120 и посмотреть, насколько приближенное решение, полученное методом дискретизации непрерывных процессов, отличается от точного. Именно это и проделано в п. 26.4.

Мы, конечно, понимаем, что поскольку мы разбили непрерывный процесс на дискретные состояния, ответ (например, через сколько времени тело ударится о землю) получится приближенный. Но насколько мы при этом ошибемся на минуту — или на год? Надо же хотя бы возможный порядок ошибки себе представлять. Именно для этого и рассматривается случай $k = 0$ — это единственный случай, для которого мы знаем точные формулы.

Самый главный вывод, который надо сделать из этого пункта: при уменьшении шага по времени dt, например, с $dt = 1$ с. до $dt = 0.5$ с. наше приближенное решение приближается к настоящему. Уменьшение шага по времени приводит к более точным решениям.

И последнее замечание. Конечно сравнение с точным решением носит чисто учебный характер. В реальных задачах, да и в нашей начальной задаче никакое «точное» решение неизвестно. Мы потому и моделируем эти процессы на ЭВМ, что не знаем для них точных формул. Как я уже говорил, мы пытались всюду в учебнике приводить в качестве примеров только такие задачи, которые не сводятся к задачам из школьной математики или физики, которые без информатики, без компьютеров в каком-то смысле «не решаются» вообще.

Выбор шага по времени

После осознания, что уменьшение шага по времени приводит к более точным решениям, немедленно возникает вопрос, почему бы сразу не взять какой-нибудь достаточно маленький шаг, например, $dt = 0.000001$, чтобы решение получилось достаточно точным.

Здесь надо отметить два момента.

Во-первых, в каких-то задачах и шаг $dt = 0.000001$ может оказаться «слишком большим». Является ли шаг «достаточно маленьким» или все еще «большим», зависит от решаемой задачи.

Во-вторых, в других задачах шаг $dt = 0.000001$ будет «чересчур маленьким». Если, например, требуемую точность решения можно достигнуть уже при $dt = 1$, то запуск алгоритма сразу с $dt = 0.000001$ приведет к тому, что алгоритм будет работать в 1000000 (миллион!) раз дольше, чем это необходимо. Ведь чем меньше шаг по времени, тем больше состояний придется вычислить в алгоритме А120 и тем дольше будет работать алгоритм.

Поэтому стандартный практический прием состоит в том, что сначала алгоритм выполняют с каким-то, кажущимся соответствующим задаче шагом. (В задаче падения парашютиста я бы лично начал с шага $dt = 0.1$). Полученные в результате вычислений значения запоминают, потом уменьшают шаг вдвое, выполняют алгоритм снова и запоминают вычисленные значения. Если разница между полученными таким способом результатами невелика, то считается, что шаг достаточно мал. Если же разница велика, то шаг уменьшают еще вдвое и повторяют вычисления.

При таком подходе к выбору шага по времени:

— с одной стороны, шаг очень быстро уменьшается, поэтому мы очень скоро достигнем «достаточно малого» для данной задачи шага;

— с другой стороны, все «предварительные» вычисления, вместе взятые, занимают не больше времени, чем одно «итоговое» вычисление, т. е. мы тратим максимум в два раза больше времени, чем если бы провели только одно «итоговое» вычисление (если бы из каких-то иных соображений нам было точно известно, каким должен быть шаг по времени).

Пункт 26.5 про выбор шага по времени — это некий заключительный нюанс, заключительный штрих к тому, как на самом деле моделируются процессы на ЭВМ. Этот пункт, в принципе, можно опустить. Проблема выбора шага по времени останется не раскрытой, но это уже менее важная тема, чем сам метод дискретизации непрерывных процессов. Что же касается сути метода дискретизации, то тут ничего потеряно не будет.

§ 27. Компьютерное проектирование и производство

Как и все остальные параграфы про применения ЭВМ, этот параграф начинается с беллетристической части: использование ЭВМ для подготовки чертежей; замена реальных экспериментов (когда, например, автомобиль с чучелом вместо человека, со всей скорости врезается в неподвижную стену) на моделирование эксперимента на ЭВМ; использование ЭВМ для управления станками с программным управлением и т. п.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Преобразование Чертежника в устройство для резки металла (п. 27.3 учебника) идейно восходит к исполнителю «Резчик металла» из нашего вузовского учебника [ПдМ], откуда и взяты команды «поднять резак», «опустить резак» и пр. Хотя Резчик металла и проще Робота, он, ввиду его очевидной «производственности» и кажущейся непридуманности, — лишен того ореола «игрушечности», который так мешает нормально работать с Роботом.

Поэтому, если вам не удалось преодолеть «игрушечность» Робота, то вы можете попробовать простые алгоритмы проработать с «Резчиком металла» (систему команд и примеры алгоритмов можно найти в [ПдМ]), а к Роботу вернуться после прохождения команд циклов.

В содержательной части параграфа — п. 27.5 — изложена лишь простейшая информационная модель (М20) поверхности сложной формы, например, капота или крыла автомашины. В [ПдМ] и в учебниках [Ершов] есть картинки, изображающие различные детали автомобиля, представленные методом конечных элементов (т. е. треугольниками, четырехугольниками и пр.). В учебнике ни картинок, ни алгоритмов обработки таких моделей не приведено. Как я уже говорил (см. с. 256), соответствующий материал просто не уместился в отведенный размер учебника. Мы лишь привели модель М20 и сформулировали несколько упражнений, из которых упр. 1 и 2 мне кажутся обязательными, если эту тему вообще проходить.

§ 28. Заключение

Заключение начинается с описания еще нескольких областей применения ЭВМ: электронные деньги, штриховые коды, компьютерные игры и проч. Все это можно прочесть в газетах, в книжках и т. д. Материал является беллетристическим и изложен в том же стиле, что и беллетристические части § 23–27.

Основное содержание параграфа сосредоточено в п. 28.3 «Ошибки в применениях ЭВМ» на страницах 215–217. Этот материал кажется мне чрезвычайно важным, особенно с точки зрения формирования общего мировоззрения учеников. И самое главное, пожалуй, написано в подпункте 1 п. 28.3, где говорится, что ЭВМ не столько развивают научно-технический прогресс, сколько тормозят его. Компьютеры настолько окружены ореолом «современных технологий», ореолом чего-то новейшего, что их реальная роль в научно-техническом прогрессе осознается очень редко. Дело в том, что мы всегда передаем ЭВМ рутину, только то, что уже осознано, то, что делать не сложно, а все наше творческое начало остается при нас.

Конечно, замечательно, что мы, тем самым, можем освободиться от рутинных дел и заняться творческими задачами. Но есть и вторая сторона медали. Когда мы перестаем справляться с потоком рутины, у нас есть два пути. Первый — попытаться осознать, почему рутины так много и изменить жизнь таким образом, чтобы ее не было. Это и есть развитие, прогресс. Другой путь — поставить ЭВМ и заставить ее в больших объемах перерабатывать всю рутину, никак не меняя нашу жизнь по существу. В этом смысле ЭВМ всегда являются своего рода консерваторами существующего состояния дел. Когда человек не справляется с огромными потоками рутины, применение ЭВМ позволяет некоторое время выжить, не меняя существа дела. Если говорить о привнесении чего-то принципиально нового, то ЭВМ этому не способствуют — они сохраняют уже существующие технологии и делают потребность в изменениях менее насущной.

В учебнике приведено несколько примеров таких ситуаций. Например, при усилении транспортного потока через перекресток можно пытаться поставить ЭВМ, которая бы учитывала количество транспорта на разных направлениях и соответствующим образом переключала светофор с целью оптимизации дорожного движения (и такие проекты существовали в действительности). Можно, однако, вместо этого

построить дорожную развязку, чтобы исчез и перекресток, и светофоры, и необходимость регулировать движение. Другой пример — переход на порошковые технологии изготовления металлических деталей делает ненужной чистовую обработку детали (и ЭВМ, управляющие такой обработкой).

Все сказанное здесь о роли ЭВМ как консерваторов существующих технологий, — почти дословный пересказ из книги Дж. Вейценбаума «Возможности вычислительных машин и человеческий разум», которая приведена под номером «2» в списке рекомендуемой литературы на стр. 224 учебника. Лично я считаю, что это — лучшая книга всех времен и народов о компьютерах, о месте компьютеров в мире и об их роли в развитии человечества. Хотя она написана довольно сложным языком, насыщена философией и потому читается очень медленно, я настоятельно рекомендую вам эту книгу прочесть. Я ее перечитывал раз десять — и каждый раз как новую, открывая для себя что-то, чего не заметил или не понял при предыдущем прочтении.

Конечно, есть технологии, которые без компьютера просто невозможны. Один пример такой технологии — томография — есть в учебнике. Очень яркий пример принципиально новых возможностей, которые может дать компьютер. Раньше при подозрении на опухоль мозга врач должен был вскрывать пациенту череп — нередко, лишь для того, чтобы убедиться, что никакой опухоли нет. С появлением томографа стало возможным просто просветить череп и выяснить, что там внутри.

Но, с другой стороны, часто (и даже, «как правило») компьютеры применяются для выполнения рутинных действий в рамках уже существующих технологий и в этом смысле ***тормозят*** научно-технический прогресс. Поэтому стремление просто «внедрить ЭВМ», превращение компьютеров из средства в самоцель чрезвычайно опасно.

Подпункты 2–4 пункта 28.3 менее фундаментальны и лишь демонстрируют, что ЭВМ не всемогущи и при их применении возможны ошибки и в алгоритмах, и в исходных данных — и к этому надо быть готовым. Более подробное изложение приведенных в учебнике и других примеров ошибок в работе ЭВМ можно найти в книге Майерса «Надежность программного обеспечения» [Майерс].

А вот заключительный подпункт 5 опять содержит чрезвычайно важное замечание, что компьютеры и их применения становятся частью нашей жизни. Поэтому вопрос о том, надо или не надо применять компьютеры при решении

каких-то задач, должен в первую очередь решаться, исходя из того, как изменится наша жизнь. В учебнике это не очень удачно названо «социальными последствиями». Более глубокое изложение материала вы можете найти также в книге [Вейнценбаум].

На этом глава 3, а с ней и весь учебник заканчиваются.

Лекция 12.

D. Заключение

D1. Методики преподавания курса

В методпособии [Авербух] приведены (с. 297–301) почасовые разбивки, рассчитанные на традиционный машинный курс. Считается, что в школе или рядом со школой есть кабинет и, при желании, учитель имеет возможность провести урок на ЭВМ. Общий объем курса может составлять 102, 68 или 32 часа — и показана рекомендуемая разбивка этих часов: сколько занятий в классе без ЭВМ и сколько — на ЭВМ для каждой темы.

По сути, эта разбивка отчасти отражает картинку, которую я рисовал (с. 52), хотя почасовая разбивка больше зависит от сложности, а столбики на картинке были интегральные — отражали и сложность, и важность одновременно. Важная или не важная тема — часы выделяются в зависимости от сложности, чтобы тема была нормально усвоена.

Вы можете использовать эти почасовые разбивки вместе с картинкой на с. 52 для понимания относительной ценности разного материала учебника — по мере уменьшения общего объема курса в нем остается время только на самый важный материал, а остальное постепенно выпадает. Если вы почему-либо не успеваете, не укладываетесь во времени, то оцените, что можно пропустить, а что надо обязательно оставить.

Повторю, что названные почасовые разбивки относятся к так называемому традиционному машинному курсу. «Традиционный» означает, что как учебник написан, так мы его и изучаем — параграф за параграфом.

Если говорить о более общем подходе, то можно выделить четыре основных методики преподавания курса.

1) БЕЗМАШИННАЯ — компьютеры вообще недоступны, организовать уроки с использованием ЭВМ нет никакой возможности. Этот случай требует от учителя наибольших усилий, потому что он должен придумывать «черные ящики», разыгрывать спектакли и иным образом компенсировать отсутствие ЭВМ. Учитель сам (за счет своей личности) должен сделать предмет интересным. Для учителя это наиболее тяжелый курс.

Но важно понимать, что с точки зрения методики преподавания такой безмашинный курс ничем не хуже математики. Вы вполне можете излагать новый материал, задавать задачи, проверять домашние задания, проводить самостоятельные, контрольные и т. д., то есть делать все то, что делают учителя математики. Другая предметная область, но тот же самый стиль, та же методика.

Конечно, безмашинный курс менее интересен и, соответственно, менее эффективен, чем машинный. Когда есть интерес, учить лучше. Но, повторю, даже в этом случае учиться нужно и есть чему, и математика является хорошим аналогом. В курсе математики ученики тоже решают задачи, пишут в тетрадях, сдают их, а учитель проверяет. Конечно, при этом учителю надо сидеть с тетрадками, проверять и т. д. Методика и стиль преподавания безмашинного курса информатики почти не отличаются от преподавания математики. Во всяком случае, тот курс и тот учебник, про который я рассказываю, вполне можно излагать в таком стиле, задавать задачи, проверять решения, устраивать самостоятельные и контрольные, проверять, выставлять оценки, задавать новые задачи и т. п.

2) МАЛОМАШИННАЯ. Это такой промежуточный вариант, когда есть возможность эпизодически водить школьников куда-нибудь, грубо говоря «на экскурсию». Нельзя планировать определенные уроки на ЭВМ, но можно, скажем, раз в месяц, посетить вычислительный центр. (У меня есть знакомые учителя, которые водили школьников даже в агентство по продаже авиабилетов, а потом на уроках обсуждали, как эти системы и машины работают, как построить информационную модель того, что они видели и пр.)

Маломашинный курс практически ничем не отличается от безмашинного. Единственное, что для повышения мотивации иногда бывают экскурсии. Если при этом есть возможность что-то выполнить на компьютерах, то желательно, чтобы школьники заранее подготовились, решили какие-то задачи, которые они в ходе экскурсии выполнят.

Если удается выполнять алгоритмы на компьютерах регулярно — скажем, раз в месяц, то ***маломашинный*** курс фактически превращается в ***полумашинный***: можно месяц изучать материал в безмашинном математическом стиле, а потом проводить самостоятельную или контрольную работу с выходом на компьютеры.

3) ТРАДИЦИОННАЯ МАШИННАЯ. Слово «машинная» мы применяем в ситуации, когда учитель может планиро-

вать, какие уроки ему проводить на ЭВМ, а какие — без. Другими словами, когда учитель свободен в этом планировании. В маломашинном или полумашинном курсе компьютеры не очень доступны, и надо загодя договариваться, о занятиях школьников. Машинным же мы называем курс, если компьютерный класс есть либо прямо в школе, либо рядом, если учитель может составить расписание, спланировать занятия на ЭВМ тогда, когда это нужно по логике преподавания. Именно для такого традиционного машинного курса приведены все почасовые разбивки в методическом пособии [Авербух].

4) ИНТЕНСИВНАЯ. Наконец, есть еще курс, называемый интенсивным. Довольно много педагогов считает, что если компьютеры есть, то абсолютно все занятия надо проводить на ЭВМ. Обычно интенсивный курс проводится, когда компьютеры стоят прямо в классе и их в любой момент можно включить.

Я сейчас более подробно остановлюсь на машинном и интенсивном курсах (про безмашинный уже все сказано — есть материал, его можно задавать, спрашивать, проверять в тетрадях и т. д.).

Машинный курс. При изложении материала параграф за параграфом, одной из лучших является, видимо, — схема «лекция, семинар, практика». По каждой теме одно занятие проводится в форме «лекции» — учитель задает простые задачи (проблемный подход), рассказывает, излагает новый материал и пр. Второе занятие — «семинар», когда школьники без машин решают задачи, а учитель смотрит, подправляет и консультирует. Третье занятие — «практика»: на нем школьники уже решенные (и записанные в тетради) задачи набирают и выполняют на ЭВМ. «Практики» может быть не одно занятие, а два или три. Авербух, как правило, рекомендует схему 1/1/2 — «лекция, семинар, практика, практика». Если материал сложный, то соответственно увеличиваются лекционные или семинарские часы.

Для такого традиционного курса в методпособии имеется не только почасовая разбивка, но и поурочное планирование для первых 24 уроков, с указанием, какие упражнения надо решать на уроке, какие задавать на дом и пр. Конечно, это не догма, а всего лишь пример. Если вы преподаете курс в первый раз, это может оказаться полезным. Потом, по мере роста понимания и опыта, вы сможете поурочное планирование менять.

Кроме традиционной, имеются и другие формы проведения машинного курса. Например, есть подход, при котором любая тема занимает ровно два урока. На первом уроке учитель рассказывает новый материал и задает упражнения на дом, а на втором — принимает на ЭВМ домашнее задание, после чего переходит к следующей теме. Если параграф сложный (типа § 9), то при таком подходе его просто разбивают на две части, говорят, что здесь две темы, а расчасовка остается стандартной — 1/1.

Интенсивный курс. Крайним выражением таких подходов является интенсивный курс, когда учитель вообще ничего не рассказывает на уроках. Ученикам на дом задается прочесть следующий параграф и подготовить какие-то упражнения. (Учебник это, вообще говоря, позволяет — т. е. можно просто задать школьникам прочитать материал, который написан достаточно просто.) Поэтому в интенсивном курсе говорят так: «На дом: решить то-то и то-то, прочесть очередной параграф или очередную тему, часть параграфа.» Когда школьники приходят на урок, их усаживают за ЭВМ решать то, что им задавали на дом, а преподаватель отвечает на вопросы. И так урок за уроком — школьники работают на ЭВМ, а все остальное делают дома сами. Изредка, по мере надобности, учитель может проводить не урок, а своего рода консультацию, отвечая на накопившиеся по всему прошлому материалу вопросы или обращая внимание учеников на какие-то нюансы, которые они могли пропустить при чтении.

В таком интенсивном курсе все уроки проводятся на ЭВМ, изложения материала как такового вообще нет, а школьникам говорят «читайте учебник». Изредка устраиваются консультации для того, чтобы уровень всех подогнать до того, что учитель считает нужным. Мы обычно, такую форму курса называем режимом «практикума», поскольку школьники все время работают за ЭВМ.

Из других форм интенсивных курсов я отмечу только одну, о которой уже говорил, — это прохождение всей начальной части до «величин» путем «шоковой терапии», когда школьникам за один урок излагаются все управляющие конструкции, а потом начинается практика на ЭВМ. В отличие от последовательного изложения материала в учебнике, при «шоковой терапии» значительная часть учебника (теоретический материал) дается на одном уроке, а потом начинается решение задач из всего этого раздела. А уж школьники сами

должны разбираться, какие им конструкции использовать и как решать задачи.

Дело в том, что последовательное и неторопливое — шаг за шагом — введение алгоритмических конструкций не всегда оказывается удачным. Если класс сильный, то ученикам может стать скучно. Тогда лучше изложить все управляющие конструкции на первом уроке, а потом уж их обсуждать и решать разные задачи. Так одним залпом, можно пройти весь материал до «величин». Это хорошо работает, если класс достаточно сильный — существенно экономится время и сохраняется интерес. Зато потом, в третьей главе, можно дать больше применений ЭВМ.

И, наконец, последнее. Конечно, учитель вовсе не обязан выбирать какой-то один стиль и строго его придерживаться. Вы вполне можете, например, пройти управляющие конструкции методом «шоковой терапии», устройство ЭВМ — вообще в беллетристическом безмашинном стиле, что-то еще (скажем, раздел про величины) — в режиме практикума и т. д. В машинном курсе полезно сходить на экскурсию в агентство по продаже билетов, а потом попытаться составить информационную модель этой системы. Все эти стили, методики, возможности построения курса можно комбинировать между собой в соответствии с вашим ощущением класса, в соответствии с тем, как вы считаете нужно работать с вашими учениками. Другими словами, смотрите на все это, как на возможности, из которых вы можете выбирать, набирать и строить свои собственные курсы.

Я, например, считаю, что даже в режиме практикума учителю следует периодически проверять тетради, а в чрезвычайно машинном курсе полезно с учениками говорить, писать что-то в их тетрадях и т. п. Все можно комбинировать, строить и искать что-то свое. Ведь это — всего лишь набор возможностей и методов, из которых можно выбирать.

Сложные части учебника

Теперь я хочу обрисовать наиболее сложные части учебника (они же — первые кандидаты на пропускание в слабых классах или при преподавании в 7–8 классах), а также то, что с ними связано. На мой взгляд, в целом в учебнике есть ровно две относительно сложные темы:

а) Методы алгоритмизации (§ 16) и наиболее математизированный § 25, а также частично § 26, поскольку в нем используются рекуррентные соотношения, выражающее следующее состояние парашютиста через предыдущее. Если § 16 и § 25 опустить, то § 26 надо либо тоже опускать, либо несколько переделать, чтобы не использовать рекуррентные соотношения. Эту часть, я считаю, в 7–8 классах следует пропускать безусловно.

б) Рассмотрение алгоритмов, как информации (кодирование алгоритмов величинами, компиляция, интерпретация и пр.). Кодирование стартует в § 20 (буквы «П», «Н», «Л», «В», «К», понятие компиляции и интерпретации, и пока не называемая так информационная модель алгоритма). Это используется далее в § 21. Если вы всю главу 2 пропустили, то можете либо ввести кодировки сами по себе в нужных местах (они простые, и трудности не должно быть), либо — если вы считаете материал сложным или не хотите в него углубляться — пропустите эту часть § 21. Здесь достаточно материала, скажем, с тем же кинозалом, транспортной сетью и пр., — есть что изучать и без кодирования алгоритмов. Сложность кодирования алгоритмов скорее психологического плана и связана исключительно с тем, что мы начинаем рассматривать алгоритмы, как информацию и писать другие алгоритмы, которые эту информацию должны обрабатывать. Такой метапереход — алгоритмы выполнения закодированных алгоритмов.

Других сложных тем и разделов я, в курсе не вижу. Соответственно в почасовых разбивках в методпособии эти темы при уменьшении общего объема курса исчезают первыми. И в целом разбивки для разного числа часов полезны при составлении собственной расчасовки.

Ослабленный курс

И наконец, ко всему этому я еще добавлю несколько слов про ослабленный курс. Если вместо нормального общеобразовательного школьного курса надо просто «ликвидировать компьютерную безграмотность», например, прочитать курс для «директоров предприятий», то это тоже можно сделать. Первая часть сильно ослабляется и заканчивается перед «величинами» — здесь можно либо Робота «погонять» либо порешать задачи с другими исполнителями (например, с «резчиком металла» и пр.). Конечно, и эту часть надо облегчить — циклы с **если** внутри можно не давать, а решать простенькие задачки типа «довести Робота до стенки» или

«прорезать дырку в металле». «Введение» и начало курса (до величин) может быть изложено в ослабленном курсе, примерно в объеме по трети урока на параграф в очень ослабленном виде.

Дальше все до конца главы 1 пропускается, на главу 2 тратится один час, в стиле «вот это — клавиатура, это — экран, это — дисковод». Если есть возможность, то хорошо разобрать компьютер и показать, как он выглядит внутри. Чистый показ — никаких задач и никакого содержания. И, аналогично, из главы 3 берется также лишь беллетристическая часть про области применения ЭВМ. Можно показать, как работает текстовый редактор или как продаются билеты, но только внешне.

Все задачи и упражнения даются в таком курсе в начале (управление исполнителями), а курс в целом укладывается в 1–2 дня. Получается ослабленный миникурс, ликвидация компьютерной безграмотности.

Тем не менее, даже после такого курса у человека возникает ощущение понимания, что же такое алгоритм. Это пожалуй единственное содержание, которое дает ослабленный курс, — представление об алгоритме и противопоставление программирования схеме непосредственного управления (первый «кит» курса).

Еще несколько слов о методике построения курса

Помимо всего прочего, я хочу повторить, что методика нашего курса, построена еще и на том, что, образно говоря, «навыки устной речи должны предшествовать изучению грамматики» или — если привлечь пример из другой предметной области — сначала надо научиться складывать и вычитать («набить руку»), а уж потом решать содержательные задачи. Применительно к информатике, мы считаем, что сначала надо научиться писать простейшие алгоритмы и освоить алгоритмические конструкции (глава 1 учебника) и только потом приступать к решению содержательных прикладных задач (глава 3 учебника — «Применения ЭВМ»). При этом в первой части — при освоении основных алгоритмических конструкций и получении базовых навыков алгоритмизации — требуется чтобы материал, способствовал усвоению этих навыков и по возможности не содержал «ничего лишнего». Соответственно, на старте у нас нет никаких информационных систем и ничего в этом духе.

Есть курсы, построенные по иному. Например, у свердловчан [Гейн] с самого начала изложение строится на «жизненных» проблемах и примерах. И первым шагом любой такой деятельности считается построение математической модели исследуемого объекта или явления. Соответственно, с самого начала можно пытаться как-то мотивировать, почему решаются те или иные задачи.

У нас в первой главе абсолютно не обсуждается ***почему*** надо решать ту или иную задачу. Образно говоря, мы учимся складывать, а не обсуждаем преимущества задачи складывания груш перед задачей складывания карандашей. Все вопросы создания мотивации в первой главе возлагаются на учителя. Учебник ничего такого не содержит.

Наконец, сама эта первая часть — освоение основных понятий и базовых навыков алгоритмизации — может быть условно разбита на два больших блока.

Первый блок — алгоритмические конструкции (§§ 4–10 учебника): вспомогательные алгоритмы, циклы, ветвления. Этот блок можно назвать «программирование до величин». Наличие такого блока, отработка основных алгоритмических конструкций на наглядных задачах нематематического характера — существенная особенность нашего курса. Как я уже сказал, можно либо изучать этот блок следуя учебнику, либо менять порядок изложения, либо (особенно в сильных классах) сначала изложить сразу все конструкции, а потом долго решать задачи. Но какой бы ни была методика, важно, чтобы этот блок был, т. е. чтобы алгоритмические конструкции были освоены до величин.

Многие курсы начинаются с понятия величины и команды присваивания. Такой курс можно построить и на основе алгоритмического языка, но у него будет другая идеология, и, несмотря на единство языка, это будет совсем другой курс.

Второй блок — работа с величинами (§§ 11–15). Здесь рассматриваются понятия величины, присваивания, различные типы величин (числовые, логические, табличные, литерные), обмен информацией между алгоритмами. Этот блок наиболее близок к традиционным курсам, и задачи для него можно найти практически в любой книге по программированию.

D2. Место курса в «большой информатике»

Вы знаете, что есть курсы, в которых упор делается на Бейсик; есть наш курс; есть курсы, опирающиеся на Пролог или Лого; есть какие-то загадочные функциональные и параллельные языки и т. д. И в общем не очень понятно, как то, что здесь рассказано, связано, например, с параллельными процессами, логическими языками и пр., о чем речи не было. Сейчас я коротко попробую это изложить.

Процедурная, функциональная и логическая традиции

С некоей совсем глобальной точки зрения в современной информатике (computer science) существует три основных направления, три ветви, почему-то называемые «традициями»: процедурная, функциональная и логическая.

«Процедурная» традиция предполагает описания того, что компьютер должен сделать, т. е.задаются и описываются ***действия.*** Бейсик, Паскаль, школьный алгоритмический язык, Фортран, Алгол, Ада, Си, Ассемблер, как и большинство других известных языков программирования, относятся к процедурной традиции. Это основное течение, основное направление современной информатики.

Для контраста рассмотрим другие «традиции», например, «функциональную». Поскольку любая программа преобразует набор исходных данных в некоторый результат, то программу можно считать функцией, которая исходные данные отображает в результаты. Эту функцию можно пытаться задать, как композицию других, более простых функций: к примеру, $(x-1)**2$ можно записать в виде композиции $f(x) = g(x)**2$, где $g(x) = x - 1$. Примерно так в функциональных языках программирования программу в целом рассматривают как преобразование исходных данных в результаты, т. е. как некоторую сложную функцию, которую пытаются задать в виде композиции более простых функций. Те, в свою очередь пытаются задать в виде композиции еще более простых функций и т. д. При этом нет места никаким циклам, ветвлениям и прочим «процедурным» вещам, поскольку используется совсем другой подход. Место циклов, например, занимают рекурсивные функции — вспомните последовательность Фибоначчи, задаваемую соотношением $f(n) = f(n-1) + f(n-2)$.

Типичными представителями функциональных языков являются Лисп и Рефал, хотя, скажем, Лисп содержит и некоторые элементы из процедурной традиции. В функциональных языках, повторю, программа задается как некоторая функция, которая выражается через другие более простые функции, и т. д. — до каких-то базовых функций. Роль вспомогательных алгоритмов (накопленных знаний) при этом играют ранее составленные функции. Такова — функциональная традиция, функциональные языки программирования. Ни в одном из распространенных школьных курсов про это нет ни слова. Хотя Лисп является популярным языком в задачах, именуемых «системы искусственного интеллекта», а Рефал часто используется в системах обработки текстов: преобразование одного текста в другой задается как некоторая функция (для тех, кто знаком с теорией алгоритмов Маркова, добавлю, что Рефал можно считать программной реализацией этого подхода).

Обращаю ваше внимание, что функциональная традиция опирается на математический стиль мышления: одну функцию представляем в виде композиции других, другие в свою очередь, в виде композиции третьих и т. д.

Наконец, «логическая традиция». Я не могу привести вам названия широко распространенных «логических» языков. Обычно, — это специальные языки для управления базами данных, которые описывают данные и связи между ними. Чтобы понять о чем идет речь, вполне достаточно примера из учебника [Каймин] (язык Пролог): задается, что «мама мамы есть бабушка» (связь), что «Катя — мама Оли», а «Таня — мама Кати» (факты) и компьютер должен сделать заключение, что «Таня — бабушка Оли». Подобного рода описания каких-то фактов и связей, или, говоря другим языком, аксиом и правил вывода (правил, по которым из одних фактов можно получать другие) составляют основу логических языков.

В логических языках мы описываем только факты и только связи, а потом задаем вопросы «верно ли, что. . . » или «какова связь между. . . ». В чисто логической традиции компьютер должен перебрать все возможные варианты и установить, есть ли такие факты, из которых по заданным правилам вывода можно вывести то, что мы спросили. Соответственно, мы либо получим ответ (вместе с «доказательством» — выводом), либо ответ, что таких фактов и выводов нет.

Пролог, впрочем, не чисто логический язык — в нем также есть элементы процедурной традиции, т. е. возможность

явно указать, что надо ***делать*** (чтобы не перебирать уж совсем все).

Процедурная традиция является сейчас абсолютно доминирующей. Функциональная достаточно развита в некоторых специальных областях. Что же касается логической, то ее область применения пока чрезвычайно узка — работа с базами данных. Я думаю, это происходит из-за того, что в «чистом» виде «логическая» традиция жутко неэффективна. Дело в том, что в чисто логической традиции не предполагается вообще задание программы ни в виде алгоритма, ни в виде функциональной взаимосвязи. Машина просто должна перебрать все возможные варианты, а перебор вариантов — дело долгое и изначально неэффективное.

> **ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ**. Японцы пытались эту проблему обойти, создав ЭВМ с большим количеством процессоров. Всякий раз, когда задача разбивается на подзадачи, когда надо проверить вариант «а» и вариант «б» (а ведь там свои подслучаи, скажем, «а1», «а2», «а3» и т.д.), они на каждой развилке включали новые процессоры, для параллельного анализа разных вариантов, случаев и подслучаев (обычно именно эту идею у нас называют «японским компьютером 5-го поколения»). Другими словами, алгоритмическую неэффективность подхода пытались компенсировать огромным количеством процессоров, работающих каждый над своим случаем. За счет этого перебор (пока процессоров хватает на новые случаи и подслучаи) не должен приводить к затяжке времени ответа.
>
> Я вам скажу свою глубоко субъективную точку зрения, — вы ее можете принимать или не принимать, — но лично я считаю, что этот проект был затеян для того, чтобы тянуть деньги из японского правительства. Было взято изначально неэффективное решение и под него начали разрабатывать какую-то жуткую аппаратуру. Конечно, проект крайне дорогостоящий, сюда можно вбухать уйму денег, а проект все будет тянуться и тянуться, пока идея себя полностью не дискредитирует.
>
> Конечно, моя точка зрения не отрицает возможности каких-то открытий и достижений в рамках этого проекта. Когда вы варите все подряд, пытаясь получить золото, у вас случайно может получиться резина. Я лишь призываю вас с некоторым скепсисом относится собственно к идее варки всего подряд с целью получения золота и не строить школьный курс, полагаясь на безответственные высказывания о том, что в будущем нас ждет исключительно и только Пролог и машины «5-го поколения».

В заключение обзора логической традиции я бы хотел заметить, что у этой традиции есть вполне эффективная область применения — компактификация информации в базах данных (такие базы называются обычно «базами знаний»), когда хранятся не только факты, но и некоторые связи, и большой пласт информации может не храниться, а генерироваться из фактов и связей.

Если вы еще раз взглянете на эти традиции, то увидите, что процедурная традиция связана с алгоритмическим стилем мышления, а функциональная и логическая опираются на математический или логический стиль мышления. Поскольку основной целью нашего курса является развитие алгоритмической составляющей мышления, мы пошли по «процедурной» ветви, являющейся, впрочем, основной и доминирующей и в «большой» информатике. Это — первая развилка.

Базовые понятия процедурной традиции

Вторая развилка будет уже внутри процедурной традиции. Языки процедурной традиции можно делить по тому, какие основные понятия алгоритмизации они содержат или не содержат. Первое, что мы можем сделать, — разбить языки в зависимости от того, какие из четырех базовых понятий информатики (с. 32) они содержат или не содержат. На самом деле, впрочем, во всех процедурных языках есть циклы и таблицы (массивы), и рассматривать мы должны лишь наличие вспомогательных алгоритмов с параметрами и исполнителей.

Я вам говорил, что в последние годы эти понятия, включая «исполнителя», появились во всех современных языках программирования.

Но если мы говорим о традиционном Бейсике, то в нем нет ни того, ни другого. (Кстати, авторы Бейсика объясняли появление языка True Basic примерно так: «Некоторые думают, что Бейсик — конкретный язык с номерами строчек и прочими особенностями. Они ошибаются. Мы, авторы, считаем, что Бейсик — это когда удобно. Жизнь изменилась, и сейчас удобно совсем не то, что было удобно двадцать лет назад. Поэтому настоящий Бейсик (True Basic) сейчас — это

язык без номеров строк, в котором есть вспомогательные алгоритмы с параметрами и т. д.»).

Но Бейсик, обычно используемый в наших школах — традиционный Бейсик 30-тилетней давности — не содержит ни вспомогательных алгоритмов с параметрами, ни, тем более, понятия исполнителя.

К сожалению, исполнителей нет и в традиционном, классическом Паскале. (Как и с другими языками, новые версии обычно существенно отличаются от классических, например, понятие исполнителя было введено в UCSD Паскале). Но возвращаясь к классическим версиям этого языка, обычно используемым в школах, мы должны констатировать, что Паскаль не содержит понятия исполнителя. У Паскаля есть и другие недостатки (например то, что называется «блочной структурой»), но нас сейчас интересует только наличие в языке базовых понятий информатики.

Именно отсутствие фундаментальных понятий и определило наш отказ от использования Бейсика или Паскаля. Что же касается школьного алгоритмического языка, то тут картина следующая. К моменту написания учебника школьный алгоритмический язык все еще был языком, «придуманным» Андреем Петровичем Ершовым. А поскольку он был «придуманным», то мы без труда «допридумали» к нему конструкцию исполнителя. И получилось уже то, что надо. Полноценные реализации этого языка (система «КуМир») появились позже и сразу содержали все конструкции, включая «исполнителя».

Замечу, что кроме языков, явно содержащих понятие исполнителя, таких как Simula-67, Smalltalk, Ада, Модула, школьный алгоритмический, C++ и др., есть языки, в которых такой явной конструкции нет, но она достаточно легко имитируется (подобно тому, как с помощью «*goto*» можно реализовать цикл **пока**). К таким языкам относятся, в частности, язык «Си» и — как это ни удивительно — очень старый, но бывший в свое время очень популярным язык «Фортран» (в котором исполнителей можно смоделировать через механизм COMMON-блоков).

Итак, мы использовали школьный алгоритмический язык А.П. Ершова, «допридумав» в него конструкцию «исполнителя», примерно так же, как авторы UCSD Паскаля «допридумывали» Паскаль, авторы C++ «допридумывали» язык C, а авторы Бейсика сами «допридумывали» его до True Basic.

Я еще раз хочу обратить ваше внимание, что наш школьный алгоритмический язык лежит в той же ветви, где находятся все современные процедурные языки — это основное и доминирующее течение в «большой информатике». И в школьном курсе мы идем по основному пути, по которому идет весь мир.

Конструирование типов и структур данных

И последняя развилка, которую я вам обязан показать, — единственная развилка, где мы не пошли тем путем, «которым идет весь мир».

Есть еще одна классификация, по которой делят языки программирования, — это наличие в языке способов конструирования типов, возможности определять свои типы и структуры данных.

Такие механизмы были впервые предложены Н. Виртом в Паскале (и это та причина, по которой язык Паскаль столь знаменит). В Паскале я могу написать, что у меня в программе будут объекты (величины) ***типа***, например, «день недели». Что величина этого типа может принимать значения «понедельник», «вторник», «среда», «четверг», «пятница», «суббота», «воскресенье». Таков простейший способ конструирования нового типа — явное перечисление всех возможных значений. Есть и другие способы — вы можете с ними познакомиться, изучив Паскаль.

Естественно, во всех современных языках, которые я неоднократно называл выше, имеются способы создания, описания, конструирования новых типов данных.

Школьный алгоритмический язык (в том виде, как он используется в учебнике) таких способов конструирования данных не содержит. Да и вся область «конструирования типов и структур данных» в учебнике совершенно не затронута. Учебник не содержит даже намеков на то, что такая область существует. В этом смысле школьный алгоритмический язык попал в группу старых языков, таких как Бейсик, Фортран, Алгол. И это единственная развилка, я подчеркиваю, где мы пошли не по основному пути. Пошли отчасти сознательно, решив, что конструирование данных — отдельная область, которая в заданный нам объем школьного курса просто не уместилась, поскольку в число четырех фундаментальных понятий информатики не входит.

Но при любом расширении курса, увеличении его объема, при преподавании любого более углубленного курса информатики или при проведении факультатива по информатике конструирование типов и структур данных — кандидат номер один на включение.

И чтобы вы себе представляли чуть больше, я замечу, что в этой области сейчас происходит такое «раскручивание», что ли. Нулевой уровень — нет никаких способов конструирования, типы данных фиксированы (как «целый», «вещественный», «целочисленная таблица» и пр. в школьном языке). Первый уровень — есть способы конструирования данных, но сами способы (как «перечисление», «отрезок», «запись» в Паскале) фиксированы в языке. Следующий уровень — языки, в которых есть способы конструирования способов конструирования, т. е. можно придумывать новые способы построения типов. И так далее. Разница примерно такая же, как между (а) сбором мебели из готовых деталей; (б) сбором мебели из деталей, которые мы можем сами изготавливать с помощью фиксированного набора инструментов; (в) сбором мебели из деталей, которые мы можем изготавливать с помощью инструментов, которые мы в свою очередь можем изготавливать с помощью «инструментов для изготовления инструментов» и т. п. Тут имеется такое естественное развитие с переходом с уровня на уровень, пока дело не дойдет до «универсальных инструментов», с помощью которых можно изготавливать все, что угодно, в том числе и новые универсальные инструменты.

Наш курс соответствует нулевому уровню в этой классификации. Мы считаем, что первый уровень (уровень типов данных в Паскале) также может быть включен в школьный курс при увеличении его объема. Остальные уровни — это предмет специальной (уже не общеобразовательной) подготовки.

Повторю, что если в других вопросах мы пошли по основному, «магистральному» пути развития информатики, то в данном вопросе дело обстоит иначе. Мы эту область вообще не затронули, хотя такое направление довольно бурно развивается и уже имеет массу своих «поднаправлений», ответвлений и нюансов.

Другие развилки

Наконец, существует целый ряд, скажем так, «специальных» направлений и «развилок». Например, языки процедурной традиции можно классифицировать по тому, содержат ли они средства организации параллельных процессов (см. описание параллельных процессов на с. 287). В современных языках, как правило, есть способы организации и обработки исключительных ситуаций (называемых также «прерываниями»), о которых мы не упомянули вовсе (см., например, язык «Ада»).

Да и вообще методы алгоритмизации, способы описания действий, конструкции языков программирования непрерывно совершенствуются и развиваются. Многое из того, что здесь излагается, уже через несколько лет может показаться архаичным или, наоборот, широко известным и не заслуживающим специального внимания.

Но с точки зрения целей нашего курса, все эти развилки второстепенны. Конечно, мы, при всем желании, не можем охватить все стороны и аспекты информатики, даже столь фундаментальные и существенные, как, например, параллельные процессы. Поэтому, повторяя, что в курсе изложены все четыре фундаментальных понятия информатики, я хочу, чтобы вы тем не менее прекрасно отдавали себе отчет, что «настоящая» информатика — огромная область со множеством даже не затронутых и не названных нами подразделов, направлений и пр.

Таково место нашего курса в информатике вообще. Кстати, если вы посмотрите на учебник [Каймин], то увидите, там предпринята попытка слегка захватить логическую традицию. А свердловчане [Гейн], как и мы остались в рамках процедурной традиции, но сделали акценты на построении математических моделей. Соответственно, они сочли возможным использовать Бейсик, поскольку для их задач он ничем не был плох. Наоборот, те, кто преподает на Паскале, как правило достаточно глубоко затрагивают область конструирования типов и структур данных — область, безусловно, во всех отношениях достойную изучения. Но наш курс преследует другие цели — я вам их формулировал с самого начала, и построен он соответственно под эти цели.

D3. Место курса в школе

Теперь о месте нашего курса в школе. Мы как авторы учебника, считаем, что у него нет развития. Сам учебник можно переписать, сделать лучше, более интересным, расширить и т. д., но по набору понятий он закончен. После нашего курса можно двигаться дальше, но это движение, на наш взгляд, будет носить не фундаментальный, не общекультурный, а специальный (обычно «программистский») характер.

Вот стандартное продолжение нашего курса в специальные области (для факультативов, более глубоких курсов, для вузов и пр.).

1) Языковые курсы, расширяющие кругозор учащихся за счет изучения других языков (Паскаль, Си, Бейсик, Модула, Алгол-68 и т. д.). Если продолжать говорить о развитии мышления, то я бы особо выделил Алгол-68 (язык, на котором, по-моему, невозможно программировать, но который крайне полезен с точки зрения заложенных в нем идей, и который был прообразом многих других языков), а также C++ и Smalltalk. Я вам говорил, что существует объектно-ориентированное программирование. Сейчас это очень популярное словосочетание, и все обычно ссылаются на Smalltalk, как на пример того, что понимается под объектно-ориентированным программированием.

Есть языки не из процедурной традиции — Лисп, Рефал; в процедурной традиции интересен язык под названием Форт. Все это языки, безусловно достойные изучения.

Но изучение языков — скорее профессиональная подготовка, либо факультатив. Я не думаю, что это необходимо в базовой школе, даже если говорить только о Паскале.

2) Второе направление — способы конструирования типов и структур данных. Тут можно начать с Паскаля и закончить стеками, деками, деревьями, графами и пр. Эта часть безусловно входит в общее программистское образование. Ее можно найти в нашем вузовском учебнике [ПдМ], а также в массе другой литературы. Обычно, в названиях таких книг фигурируют слова «структуры», «данные» или что-нибудь в этом духе. Это либо материал для факультатива, или более глубокого программистского курса, либо часть вузовского курса.

3) Следующее направление — развитие § 16. Я могу назвать его «методы алгоритмизации» — это огромная область, которая почти не связана с языками программирования.

Материал для этого направления можно взять из нашего вузовского учебника [ПдМ]. Кроме того, есть разные другие хорошие и очень хорошие книги, например, [Вирт], [Грис], целый ряд книг [Дейкстра] и др.

В основном, это область, как-то примыкающая к так называемому «доказательству правильности программ», где математические методы применяются для доказательства правильности алгоритмов и программ. Я вам отчасти это демонстрировал, (помните «ловкость рук и никакого мошенства» (с. 217)) когда мы какими-то рассуждениями доказали, что программа правильная и получится то, что надо, но так и не поняли, как же она будет работать.

Это отдельная, достойная изучения область, и книжка [Вирт] очень хороша для введения. Но, как вы могли почувствовать, область сильно математизированна. И наше «Программирование для математиков» — хорошее название, отражающее существо дела. Для тех, кто силен в математике, изучение этой области (методов алгоритмизации и доказательств правильности программ) может привести к огромному продвижению вперед, даже если говорить о чисто практической (технической) работе по составлению алгоритмов и программ.

4) И, наконец, четвертое направление — способы конструирования языков программирования, управляющих и иных языковых конструкций. Это уже не для факультатива, а чтобы вы себе представляли направления развития науки информатики. Сейчас пока это научное направление — книжек я вам порекомендовать не могу (их просто нет), и преподавать эту область, как какой-то курс, еще рано.

Но посмотрите, что происходит. Когда у нас есть какие-то действия и мы их часто используем, мы один раз задаем вспомогательный алгоритм с параметрами, а потом пишем только его вызов, только имя. Что это имя значит — его «определение» — записано в виде вспомогательного алгоритма. Другими словами, у нас есть средства конструирования ***своих*** действий. Аналогично в Паскале есть средства конструирования ***своих*** типов данных. В школьном языке у нас есть средства конструирования ***своих*** исполнителей. А как насчет ***своих*** управляющих конструкций и ***своих*** конструкций алгоритмов и тому подобных элементов языка программирования? Ведь до сих пор мы считали заданными и неизменными все эти **нц**—**пока**—**кц**, **если**—**то**—**иначе**—**все**, **алг**—**нач**—**кон** и пр. Как, впрочем, до Паскаля считали неизменными в языке типы данных.

Оказывается, можно предложить такие способы конструирования управляющих конструкций языка, что можно будет задавать ***свои*** конструкции, например **выбор**, в котором варианты анализируются снизу вверх или в случайном порядке; циклы, в теле которых записываются варианты и выполняется какой-то один (эти две конструкции вы можете найти в книге [Дейкстра-1]), а также и любые другие. И задавать такие новые конструкции мы сможем примерно так, как задавали вспомогательные алгоритмы, а «условие окончания цикла» или «тело цикла» при этом становятся параметрами, как аргументы или результаты для алгоритмов.

Это как если бы мы с вами строили разные здания из строительных блоков, а потом обнаружили, что и сами блоки можно собирать из более мелких «кирпичиков». Или как в физике от молекул переходят к атомам и кваркам. Оказывается, что и в программировании уже обнаружены свои «атомы», «кварки» и прочие частички, из которых можно собирать все эти **если**—**то**—**иначе**, «циклы» и пр.

Думаю, лет через 10 это все станет нормой, войдет во все языки, появиться литература, учебники и пр. Сейчас же — это просто одна из зон развития. Но мне кажется полезным, чтобы вы посмотрели на конструкции школьного алгоритмического языка с этой точки зрения — просто как на удобные «блоки», собираемые из каких-то «кирпичиков». Чтобы вы подумали над возможностями программирования, если вы можете, например, определить такую конструкцию цикла:

```
нц для каждой пары (x,y), удовлетворяющей условию u(x,y)
| тело цикла (x,y)
кц
```

и потом использовать ее по мере надобности с разными условиями и разными «телами» так же, как вы использовали цикл **пока**. Чтобы вы подумали о том, какие новые возможности «накопления знаний» открывает такой подход.

Как вы видите, все три основных продолжения, развития нашего курса носят программистский, профессиональный или пред профессиональный характер. Именно это я имел ввиду, когда говорил, что наш курс, как базовый общеобразовательный курс информатики в школе, полон. Кроме него, в базовой школе можно использовать компьютеры в других предметах. Но сам курс закончен и полон.

D4. О программном обеспечении курса

Я надеюсь, что у вас уже сформировалось достаточно полное представление о курсе, о его целях, об учебнике и общей методике построения курса. Я бы хотел обратить ваше внимание, что учебник и курс построены без привязки к конкретным компьютерам и конкретному программному обеспечению. Учебник написан, исходя из того, каким он должен быть. В момент его написания никакого программного обеспечения под этот учебник вообще не было.

С одной стороны, как мне кажется, замечательно, что учебник написан «как надо», без подгонки под какие-нибудь особенности программного обеспечения.

С другой стороны, когда учебник впервые вышел из печати, он оказался в школах без какого-либо программного обеспечения вообще. Это, конечно, проблема, поскольку между выходом учебника и выходом соответствующего программного обеспечения проходит некоторое время.

Если курс опирается на Бейсик, то такой проблемы, конечно, не возникает. Бейсик есть у всех — и еще до начала написания учебника. Но я еще раз хочу напомнить вам основную цель нашего курса — развитие алгоритмического стиля мышления. Мы не могли пожертвовать этой целью в угоду какому бы то ни было программному обеспечению.

Программное обеспечение, разработанное специально для поддержки нашего курса, называется «КуМир» (название это историческое и первоначально означало набор учебных исполнителей — «Комплект Учебных Миров»). Сейчас оно используется и для самой системы программирования на основе школьного алгоритмического языка.

Хотя система «КуМир» разрабатывалась специально как современная реализация школьного алгоритмического языка, эта система — «настоящая», в отличие от старого «Е-практикума», который являлся чисто учебной системой. Если в старом «Е-практикуме» имелась возможность лишь ввести и выполнить простейший алгоритм из 30–50 строк, то при разработке «КуМира» были поставлены (и достигнуты) совершенно другие цели. КуМир — это полноценная, полномасштабная система программирования, в которой можно создать программу и в тысячу строк, в которой — на школьном алгоритмическом языке — можно реализовать и экранный редактор текстов, и простенькую базу данных, и электронные таблицы, и учебный компилятор, и приклад-

ные системы по другим школьным предметам (например, геометрии или биологии).

Методическая цель, которую мы при этом преследовали, — дать школьникам возможность не только поработать с подобного рода системами, но и познакомиться с текстом реализации соответствующей системы, посмотреть «как» это делается. Чтобы им можно было задать задачу типа: добавьте в редактор текста команду «вставить строку»; измените так-то и так-то базу данных и пр. И чтобы при этом они работали в той среде, к которой привыкли, на обычном школьном алгоритмическом языке.

Наконец, в каждом классе есть лидеры, решающие какие-то свои (не учебные) задачи. КуМир — это система, позволяющая решить любую такую задачу (даже если понадобится на физическом уровне управлять какими-то устройствами). В КуМире можно сделать все, что можно сделать на Бейсике, и даже больше того. И при этом работать в КуМире намного удобнее и эффективнее, чем на Бейсике. На мой взгляд чрезвычайно важно, чтобы лидеры класса предпочли КуМир другим системам программирования при решении конкретных задач.

Вам, наверное, будет интересно узнать, что на механико-математическом факультете МГУ, школьный алгоритмический язык используется весь первый курс. По расписанию на первом курсе лекций нет, а есть только практикум на ЭВМ, посвященный решению задач из алгебры, анализа, дифференциальных уравнений и других предметов (например, «найти все делители нуля в кольце вычетов по модулю P»). Поэтому бывшим школьникам на первом же занятии говорят «Решать будете на школьном алгоритмическом языке». Примерно 20–30 процентов спрашивают: «Что это такое?». Мы им отвечаем: «Это вам должны были объяснить в школе». И все — они успокаиваются (независимо от того, учили их, или не учили) — если в школе, значит что-то простое.

Дальше они получают пособие [Валединский], где в начале приводятся конструкции алгоритмического языка, а потом просто идет список примерно из 200 задач. Все задачи не из информатики,а из курсов алгебры, анализа и пр. Обычно мы тратим 45 минут, чтобы написать на доске все конструкции алгоритмического языка, сказать, какими кнопками они вызываются, а также объяснить, как следует входить в систему, выходить из нее и т. д., после чего студенты просто весь семестр сидят за терминалами и решают задачи.

Это может быть вам интересно, потому что, как вы понимаете, мы делаем это не ради школьного языка. У нас совершенно другие цели — научить студентов решать задачи по алгебре, анализу и пр. Мы просто не можем тратить время на изучение языка — для этого надо какие-то лекции читать, еще что-то, а так мы говорим «школьному алгоритмическому языку вас должны были научить в школе. Не научили — берите в библиотеке школьный учебник или спрашивайте у товарищей. Ваше дело — уметь решать задачи, а на каком языке — никого не волнует.»

Некоторые при этом спрашивают: «А можно я буду программировать на Си?» или «А можно я буду программировать на Бейсике?». Я, обычно, говорю: «Можно, но вы просто по времени не успеете решить все задачи, потому что на запуск, поиск и исправление ошибки и следующий запуск вы потратите времени в десять раз больше, чем при работе на школьном языке в Е-практикуме».

Стиль такой, что если они в течение года сдают, скажем, 20–30 задач по выбору преподавателя, то получают зачет-автомат. А если они зачет-автомат не получают, то на «настоящем» зачете они тянут билет с тремя задачами и через час все задачи должны правильно работать на ЭВМ. Не успел пусть даже одну букву набрать или какую-то ошибку исправить — значит не сдал, приходи в следующий раз, тяни новый билет и сдавай заново. В стандартных Бейсике, Паскале или Си мы просто не вправе были ставить такие требования.

Это немного в сторону, но я хочу еще раз подчеркнуть, что выбор языка и программной системы, как ***цели обучения***, сильно отличается от выбора ***средств обучения***. Если цель — решать задачи по высшей алгебре, то средство должно быть удобным и не требовать времени на изучение. Если цель — изучить интересный язык программирования, то язык должен быть интересным и может быть сколь угодно сложным и неудобным.

Кроме того, уж коль скоро я заговорил о механико-математическом факультете МГУ, я хочу сказать, что наш курс «Программирование для математиков» [ПдМ] появился на мехмате до появления информатики в школах. Поэтому он стартовал с исполнителей «Путник», «Резчик металла», «Редактор слова» и простейших программ управления этими исполнителями. Сейчас этот материал спустился в школьный

курс и, конечно, из вузовского курса исчез. Поэтому сейчас считается, что студенты уже умеют программировать — неважно на каком языке, а в курсе программирования им излагаются структуры данных (стеки, деки, деревья и пр.), а также решаются сложные математизированные задачи — например, задача удаления невидимых линий при изображении конечноэлементных моделей [ПдМ, с. 167–188].

И в заключение.

Всеми поставками программного обеспечения к нашему курсу, занимается объединение «ИнфоМир» (тел. 9391786, факс 9390713, e-mail: infomir@math.msu.su). Системы, как вы видели, называются КуМир, АльфаМир, ФортранМир, ПаскальМир, МикроМир и т. д. Окончание «Мир» (отражающее принцип непосредственного редактирования информации «Человек в Мире объектов» [Лебедев]) является торговой маркой, признаком всех наших систем.

Программное обеспечение для УК НЦ, Корвета и другие «старые» версии объявлены свободно копируемыми, т. е. любая школа, институт усовершенствования учителей и пр. могут бесплатно получить старые версии и свободно их копировать. Новые системы, в частности, КуМир на IBM PC, пока продаются за деньги.

КОНЕЦ КУРСА ЛЕКЦИЙ

E. Послесловие (разные замечания, отступления, рекомендации и пр.)

E1. Рекомендуемая литература

Учитель должен знать больше того, что он преподает школьникам. Поэтому, прежде всего я настоятельно рекомендую вам потратить время и силы на изучение нашего вузовского учебника [ПдМ]. Я думаю, что те из вас, у кого хватит на это сил и упорства, как минимум будут абсолютно полно представлять себе, что и для чего написано в школьном учебнике, какие цели преследуют те или иные параграфы и пр. Конечно, наш вузовский учебник достаточно сильно математизирован, недаром он называется «Программирование для математиков». Но вы можете пропустить доказательства и математические схемы в разделах, посвященных методам алгоритмизации (индуктивное вычисление функций на пространстве последовательностей, проектирование цикла с помощью инварианта и пр.), а также пролистать решения математизированных задач (построение выпуклой оболочки последовательно поступающих точек плоскости, «удаление» невидимых линий при изображении конечноэлементной модели и пр.).

Зато все, что есть в школьном учебнике, вы найдете и в вузовском (напомню, что это старый учебник, написанный до введения информатики в школах) — причем на более глубоком уровне. Кроме того, темы, затронутые в школьном учебнике бегло (как-то инвариант цикла или использование и придумывание **дано**/**надо** при составлении алгоритмов), а также не затронутые вовсе (как-то конструирование типов и структур данных) в вузовском учебнике изложены достаточно подробно и, на мой взгляд, достаточно просто.

Наконец, вы найдете там массу примеров других исполнителей (Путник, Стековый калькулятор, Резчик металла и др.) и массу примеров программ разной сложности, включая такие, где используется 3–4 уровня вспомогательных исполнителей, придуманных в ходе решения задачи (т. е. вполне содержательных задач по управлению простейшими исполнителями). Этот материал вы можете использовать и на обычных уроках в ваших классах.

Я так подробно и долго рекомендую вам [ПдМ], поскольку это тот же самый (по целям и по подходу) курс, что и в школьном учебнике, но изложенный на гораздо более глу-

боком и содержательном уровне. (Хотя чисто полиграфически учебник [ПдМ] издан ужасно). Все остальные книги — как бы полезны и замечательны они ни были — посвящены чему-то другому. Их можно и нужно изучать. Из них можно и нужно заимствовать то, что вы сочтете возможным. Но это другие книги, посвященные другим темам или основывающиеся на ином подходе к информатике.

Кроме школьного учебника [Инф], вузовского учебника [ПдМ] и настоящего пособия непосредственно нашему курсу посвящены ряд статей в журнале «Информатика и образование» [И&О] и пособие для учителя [Авербух]. Как я уже говорил, в этом пособии хорошо дополнена глава 2 учебника (Устройство ЭВМ), а также почти полностью приведены решения упражнений из главы 1 учебника. В частности и в особенности я обращаю ваше внимание на решение упражнений 6–8 на с. 175–177 пособия и замечание о «подводных камнях» (с. 176). На мой взгляд этот материал следует безусловно задействовать на уроках. Но что касается главы 3 учебника, то в этом отношении методическое пособие [Авербух] очень слабое и содержит, в основном, рекомендации, а не решения.

Существуют также многочисленные комментарии, пособия и пр. по отдельным темам нашего учебника, написанные разными учителями, из которых я, пожалуй, выделю только [Зайдельман]. Кроме того, появились и некоторые книги, связанные с учебником опосредованно, либо раскрывающие отдельные его темы, например, [ДРОФА 9, 10].

Наконец, очень близкий к нашему учебнику подход используется в курсе «Алгоритмика» [Шень] для учащихся младших классов, а также в замечательной книге [Шень-2]. Кстати, те из вас, у кого есть компьютер, подключенный к Интернет, последнюю книгу могут найти и бесплатно загрузить к себе из Интернета.

Все вышеназванные книги так или иначе связаны с нашим учебником или нашим курсом. Учитель — конечно же — должен быть знаком со всеми курсами. И уж как минимум с [Гейн], [Каймин]. Я уже неоднократно хвалил вам учебник [Гейн]. Про качество учебника [Каймин] я вам (как автор конкурирующего учебника) ничего говорить не буду. Но, на мой взгляд, вы все равно должны его прочесть и — где возможно — понять.

Даже если вы уже знакомы со всеми учебниками, я рекомендую вам еще раз их все перечитать, задавая себе один и

тот же вопрос: «Каковы цели и методы данного курса? Что должен усвоить ученик и как это достигается?».

Ну и, конечно, учитель должен быть знаком не только с учебной литературой. Здесь я прежде всего рекомендую вам книги, указанные в конце нашего учебника: «Знакомьтесь: компьютер» [ЗК] и «Возможности вычислительных машин и человеческий разум» [Вейценбаум]. Первая из этих книг — популярная и очень легка для чтения. Вторая, наоборот, имеет философский характер и достаточно трудно читается. О самих книгах я уже говорил (см. с. 127, 314).

Из других книг я назову еще раз [Вирт], [Грис], [Дейкстра]. Учителю также следует быть знакомым с основными понятиями современных языков программирования, таких как C++, Ада, Smalltalk.

Еще я хочу обратить ваше внимание, что в середине 90-х годов в жизни человечества произошло событие, роль которого трудно переоценить. Мировая сеть Интернет перестала быть достоянием узкого круга ученых и инженеров и стала доступна сотням миллионов рядовых граждан развитых и развивающихся стран. Компьютерное сообщество мгновенно, за 1–2 года, признало ряд важных общемировых стандартов, из которых прежде всего следует отметить:

(1) язык подготовки гипертекстов HTML,
(2) объектно-ориентированный язык программирования Java и
(3) простенький язык программирования гипертекстов, по сложности сравнимый с Бейсиком, — JavaScript.

По этой тематике в сети Интернет есть много свободно-копируемых материалов, а к моменту выхода нашей книжки, я надеюсь, появятся уже и книги на русском языке. Поэтому желательно, чтобы помимо традиционных языков, указанных выше, учитель познакомился также с:

— языком и системой программирования Visual Basic фирмы Микрософт;
— языком подготовки гипертекстов HTML (на уровне, достаточном для составления гипертекста из головной страницы, ссылающейся на пару других страниц);
— языком программирования гипертекстов JavaScript (на уровне достаточном для составления контрольных вопросов к простейшему учебному гипертексту);
— языком программирования Java (основные понятия объектно-ориентированных языков и основные понятия параллельного программирования — потоки и процессы).

Е2. Как возник Робот

Это отступление будет посвящено истории возникновения школьного «Робота» в том виде, в котором он используется в учебнике. История эта началась в 1980 году, когда мы еще только-только приступали к преподаванию нового курса программирования на механико-математическом факультете МГУ (раньше в качестве курса программирования на мехмате преподавался ассемблер для ЕС ЭВМ (аналог IBM 360) и язык Фортран).

Надо сказать, что мы к тому времени уже осознали роль исполнителей (этим термином мы тогда обозначали пакет, программ работающих над общими данными, а вовсе не «игрушечных» внешних исполнителей типа Робота). Более того, мы понимали, что это понятие первично и более важно, чем понятие подпрограммы, величины (переменной, объекта) и пр.

С другой стороны, мы считали, что студенты должны работать — составлять программы. И мы поставили перед собой фантастическую по тем временам цель — студент должен уйти с ***первого*** занятия с выполненной программой. Напомню, что речь идет о времени, когда программы еще набивались на перфокартах, потом отдавались на обработку на ЭВМ в так называемом ***пакетном режиме***, а результаты распечатывались на АЦПУ. Обычно, набитую на перфокартах программу студент получал на следующий день после сдачи текста, отдавал программу на выполнение, после чего на следующий день получал результаты, сдавал в набивку тексты перфокарт, в которые надо было внести изменения, и т. п. Над дверью перфораторной висел лозунг: «Если вам не то набили — обратитесь, вам добьют». На все эти технические сложности накладывались содержательные — ведь для составления даже самой простой программки на Фортране надо пройти массу материала, который в один урок никак не поместится.

Поэтому мы решили создать пакет (библиотеку) программ на Фортране, который можно будет быстро и легко объяснить, с тем чтобы студенты составили лишь основную программу, состоящую только из вызовов подпрограмм. В качестве такого пакета программ на Фортране — исключительно для простоты объяснения и постановки задач (по принципу «ничего лишнего») — мы придумали пакет программ «Путник», описание которого вы можете найти в [ПдМ], и который отличался от Робота тем, что имел ориентацию (шагал вперед, поворачивал налево и направо). Кроме того, препятствия у Путника, в отличие от стен на поле Робота, занимали клетку целиком (иначе их было невозможно напечатать на АЦПУ). Кроме того, мы решили заранее заготовить перфокарты с базовыми командами Путника (на Фортране), т. е. перфокарты вида

```
CALL STEP       ; Путник.сделать шаг
```

чтобы студенту достаточно было набрать из уже готовых, набитых перфокарт свою программу и отнести ее на ЭВМ. Заметьте, что текст справа — всего лишь комментарий для человека. ЭВМ будет понимать и выполнять вызов подпрограммы на Фортране. Но мы решили, что «существо» дела — как ЭВМ выполняет такие программы — мы объясним потом. Пока же главное — дать студентам возможность выполнить программу и унести домой распечатку с ответом.

Это все было осознанным построением курса. А вот дальше, как говорят, «не было бы счастья, да несчастье помогло». Многое в жизни происходит случайно или по каким-то техническим причинам. Чтобы пропустить большой поток студенческих программ, мы использовали в то время систему АСФОР, созданную студентами факультета под руководством В.Б. Бетелина, которая работала в сотни раз быстрее стандартного программного обеспечения (в рамках стандартного обеспечения наши цели были в принципе недостижимы). И то ли АСФОР оказался недостаточно хорош для введения новых библиотечных подпрограмм, то ли у меня в ту пору не хватило знаний и умений, но я, как сейчас, помню вечер накануне первого выхода студентов на ЭВМ, когда я — после двух суток борьбы с АСФОРОМ — с абсолютной ясностью понял, что создать библиотеку программ Путника до утра я не успею. А ведь в 9 часов утра первая группа придет на первый семинар — и что мы будем делать? Я думаю, вы очень живо можете представить себе мое состояние в тот момент. Апатию, отчаяние и лихорадочное «ну что же делать?».

Отход от Фортрана

И именно тогда — под воздействием обстоятельств и понимая полную уже несовместимость наших целей и реального программного обеспечения — я вдруг понял, что для выполнения программ, которые наши студенты будут составлять на первом занятии, никакой Фортран не нужен — я

могу написать программу-интерпретатор, и она все выполнит. Поскольку речь шла о линейных программах без управляющих конструкций, мне пришлось написать программу-интерпретатор типа алгоритма А91 из учебника, реализовать информационную модель исполнителя Путник и написать подпрограммы, реализующие ввод студенческой программы с перфокарт и распечатку программы и результата ее работы на АЦПУ (на последнее ушло больше всего времени: потому что надо было сделать так, чтобы на каждого студента тратился один и ровно один лист АЦПУ, чтобы он останавливался на сгибе бумаги, чтобы его было удобно отрывать и т. п.).

Сейчас, по прошествии времени, я думаю, что именно тогда впервые мы пожертвовали существующим программным обеспечением ради сохранения целей курса. Т. е. в ситуации выбора и цейтнота были выбраны и сохранены цели курса, а не Фортран.

Дальше ситуация развивалась стремительно и абсолютно естественно. Коль скоро программа перестала быть программой на Фортране, из текста перфокарт немедленно исчезли все «лишние» символы, все эти «CALL STEP» и прочие фортрановские штучки. Осталось лишь самое необходимое — «Сделать шаг». Программа приобрела простой читаемый русский вид, ***перестала быть набором вызовов библиотечных подпрограмм*** на Фортране, а превратилась просто в программу управления неким внешним устройством.

Аналогичным образом были реализованы исполнители «Стековый калькулятор», «Редактор слова» и «Резчик металла».

Появление Робота в его нынешнем виде

Позже, когда мы начали адаптировать этот подход — управление внешними исполнителями — для школьников, в ходе написания промежуточного учебного пособия [Ершовмы-Шень] А.Х. Шень предложил упростить Путника, оставив ему для движения команды Резчика металла (вправо, влево, вверх, вниз), убрав ориентацию и стороны света. В соответствии с принципом «ничего лишнего» этот подход был принят. Одновременно, ввиду развития техники, появления персональных компьютеров, экранов и пр., мы решили расположить препятствия (стены) между клетками (чего раньше сделать не могли из-за технических ограничений печати на АЦПУ).

Так родился «Робот». И именно в этом виде, уже без всяких изменений, он перекочевал в наш школьный учебник [Инф] и в «Алгоритмику» [Шень].

Если сравнивать Робота с Путником, то я должен заметить, что и сам Робот, и алгоритмы, управляющие Роботом, выглядят проще. Путник — более сложный исполнитель, имеющий ориентацию и умеющий действовать относительно этой ориентации (например, узнавать, стена или свободно ***справа*** от Путника). С другой стороны (и как следствие) алгоритмы решения одних и тех же задач при использовании Путника записываются часто компактнее и красивее, чем при использовании Робота. Поэтому, если Робот начинает казаться слишком простым и слишком игрушечным, то можно заменить его Путником и несколько повысить класс решаемых задач.

И в заключение — еще несколько слов о Путнике, Роботе, Резчике Металла и других клетчатых исполнителях. Я думаю, что любая попытка обучить основам алгоритмизации, не пользуясь математическим материалом и числами, с неизбежностью приводит к использованию того или иного исполнителя, обитающего в «клетчатом мире». До начала перестройки мы мало знали о том, что делается в преподавании информатики за рубежом, в частности не были знакомы с роботом по имени Карел [Паттис]. По-видимому, все, кто пытался решить эту задачу, приходили примерно к одному и тому же результату.

Е3. Как возник школьный алгоритмический язык

Сказать по правде, если бы я выбирал язык для преподавания основ информатики, и если бы до этого никакого языка в школах не использовалось, то я бы выбрал «псевдокод» из нашего вузовского учебника [ПдМ]. Дело в том, что «псевдокод» возник так же, как и Путник. Начав писать на первых занятиях «сделать шаг» без всякого ФОРТРАНа, мы просто вынуждены были в момент появления управляющих конструкций написать по-русски:

```
если впереди свободно
| то сделать шаг
конец если
```

или

```
цикл пока впереди свободно
|   выполнять
| сделать шаг
конец цикла
```

Другими словами, как только мы стали писать, «как надо», чтобы поменьше объяснять студентам и чтобы все было само понятно, у нас совершенно естественно возник русский «псевдокод». И лишь после этого мы реализовали сначала простой язык программирования без переменных «Мини» (это сделали — тогда второкурсники — братья А. и Ю. Прилипко под руководством А.А. Веденова), а потом и более содержательные системы программирования на основе «псевдокода».

Школьный алгоритмический язык родился по-другому. Когда в 1985 году было принято решение ввести курс информатики в школах, Андрей Петрович Ершов выбрал в качестве основы для преподавания широко распространенный в то время в научной литературе язык Алгол-60 (что и отразилось в названии: «алгоритмический язык» — это дословный перевод английского «Algorithmic Language», сокращением от которого является Algol). Поскольку, однако, настоящий Алгол достаточно сложен и не приспособлен для школы, было выбрано очень небольшое подмножество языка (например, без понятия блочной структуры), которое кроме того было русифицировано (так появились **алг**, **нач**, **кон** и пр.). Получился достаточно простой и понятный, но несуществующий язык.

В первых школьных учебниках [Ершов] этот язык, впрочем, и не считался языком программирования. Он рассматривался как некоторая нотация для записи алгоритмов, причем сами алгоритмы могли быть совершенно неформальными, формулы можно было записывать как в математике (безо всякой линейной записи) и пр. Считалось, что алгоритмический язык — это язык для записи алгоритмов (в первых учебниках под алгоритмом понималось существо вычислений) и что при работе на ЭВМ алгоритм надо будет перевести (переписать) на один из языков программирования, как-то Бейсик, Паскаль, Рапира. Таким образом, школьный алгоритмический язык (как язык для записи алгоритмов) противопоставлялся языкам программирования. Правила записи алгоритмов были достаточно расплывчаты (ведь язык был неформальным) и требовали соблюдения формы только управляющих конструкций.

Летом 1985 года (напомню, что курс информатики был введен в школах с 1 сентября 1985 года) мы проводили на мехмате МГУ курсы подготовки учителей информатики, т. е. излагали содержание информатики (в том виде, в каком оно было в первых учебниках А.П. Ершова) учителям математики и физики, которых «призвали» в преподаватели информатики. Поскольку к этому времени мы уже, с одной стороны, не представляли себе интенсивного курса информатики без массового решения обучаемыми задач на ЭВМ, а, с другой стороны, считали обычным делом создание специального программного обеспечения для поддержки курса, мы реализовали алгоритмический язык на ЭВМ, создав систему, названную «Е-практикум» [Е-практикум]. («Е» — в честь автора языка А.П. Ершова. В отличие от реализации Путника, которая заняла одну ночь, «Е-практикум» оказался серьезной работой, и на его реализацию была потрачена целая неделя.) Конечно, нам при этом пришлось формализовать те части языка, которые не были формализованы в учебнике.

В результате статус алгоритмического языка резко изменился: теперь он стал ***одним из*** языков программирования, исчезла ненужная работа по ручному «переводу» алгоритмов на «настоящий» язык программирования, надуманное противопоставление алгоритмического языка языкам программирования, а алгоритмов — программам. Соответственно, в более поздних учебниках [Ершов-мы-Шень], [Инф] алгоритмический язык рассматривается как один из языков программирования. К 1987 году более развитые «Е-практикумы» (Е-86 и Е-87) были реализованы на всех школьных ЭВМ (Д.В. Варсанофьев, А.Г. Дымченко, А.Г. Леонов, М.Г. Эпиктетов). А затем на IBM PC была реализована полноценная и полномасштабная система программирования на школьном алгоритмическом языке, получившая название «КуМир» (реализация на IBM PC — М.Г. Эпиктетов, адаптация на УКНЦ — М.В. Лебедев).

Е4. История возникновения системы «КуМир»

К 1987 году учебная система программирования «Е-практикум» и ее более поздние версии «Е-86» и «Е-87» были реализованы на мини-ЭВМ СМ-4 и на всех школьных ЭВМ: на Ямахе, Корвете, УК НЦ и даже на БК-0010. Системы эти позволяли учителю работать по учебникам [Ершов], организуя на ЭВМ практикум по составлению программ на школьном алгоритмическом языке. Однако все эти «Е-практикумы» были чисто учебными — отсутствовали файловый ввод-вывод и графика, а длина программы была ограничена тремя десятками строк. Школьники, интересующиеся информатикой и программированием, быстро выполняли в «Е-практикуме» учебные задания и переключались на Бейсик (реже Паскаль), чтобы написать свои собственные программы, выгодно отличающиеся от учебных тем, что они

а) имели длину аж до 100 и более строк,
б) позволяли нарисовать что-то графическое на экране,
в) давали возможность в начале работы что-то прочесть из файла, а по окончании работы что-то записать в файл.

Желание писать не только учебные, но и миниатюрные «настоящие» программы вполне естественно. Мы сами по опыту использования «Е-практикума» во вводном курсе программирования на 1 курсе мехмата МГУ поняли, что для поддержки курса информатики в школе и начального курса программирования в вузе больше подходит не чисто учебная, а учебно-производственная система, которая

— построена на развитии школьного алгоритмического языка;
— допускает динамическое подключение Робота, Чертежника и других внешних исполнителей;
— наследует удобный интерфейс «Е-практикума»: ввод конструкций нажатием одной-двух клавиш, мгновенная диагностика ошибок при при наборе программы, отслеживание ошибок в процессе выполнения программы, пошаговая отладка с выдачей на поля результатов присваиваний и пр.;
— не уступает Бейсику по производственным характеристикам.

К этому моменту для курса «Основы программирования» на мехмате МГУ уже была разработана и эксплуатировалась учебно-производственная система программирования «ФортранМир» на основе языка Фортран 77 для ЭВМ Электроника-85 и PDP-11/70 (В.В. Борисенко, Д.В. Варсанофьев, А.Г. Дымченко). Система унаследовала и улучшила интерфейс «Е-практикума» и на ЭВМ с памятью 128К выигрывала у Бейсика по производственным показателям. Словом, было ясно, что учебно-производственный «Е-практикум», этакий «Русский Бейсик», в принципе сделать можно. В то время в образовании только-только начали появляться ЭВМ типа IBM PC, и мы решили разрабатывать новую систему на IBM PC, но так, чтобы потом ее можно было «перенести» на «массовые» школьные ЭВМ «Ямаха» и «УК НЦ». (Перенос, а точнее переделка системы на УК НЦ позже действительно состоялись, однако производственной системы на УК НЦ не получилось. Перенос на Ямаху даже не рассматривался — пришли другие времена).

Вначале новая система называлась «Гамма-практикум» — продолжение мехматовской серии, состоящей из «Альфа-практикума», «Бета-практикума» и «С-практикума» (практикум по структурам данных), разработанных неформальной группой «Аттик» мехмата МГУ (из которой позже и возникло объединение «ИнфоМир»). Потом новую систему стали называть «Мега-Е», чтобы подчеркнуть преемственность с «Е-практикумом». Затем тогдашний директор объединения «ИнфоМир» А.И. Левенчук в рамках приведения «к общему окончанию» всех поставляемых «ИнфоМиром» систем предложил распространить название “КуМир” — Комплект Учебных Миров — не только на набор исполнителей (Робот, Двуног, Вездеход и пр.), но и на саму систему программирования. А.Г. Кушниренко некоторое время выступал против, но название быстро распространилось и прижилось.

На первом курсе мехмата МГУ системой «КуМир» пользуются, начиная с 1992 г., для составления программ по алгебре, анализу, геометрии и программированию. «КуМир» применяется также во многих школах и педвузах для поддержки либо всего курса школьной информатики, либо темы «основы алгоритмизации». С 1995 года многие школы используют «КуМир-Гипертекст» в курсе «Информационная культура». Основные черты и особенности «КуМира» таковы:

— простые программы легко набрать и просто отладить;
— лексика полностью совпадает с учебником;
— есть ввод/вывод, в том числе и файловый;
— есть графика;

— ученики могут написать свою игровую программу и играть с ней;
— объем программы — до 1000 строк;
— имеется возможность задания тела алгоритма в машинных кодах;
— есть команда «выход» для неструктурного завершения конструкции;
— есть полная рекурсия;
— есть конструкция «исполнитель», общие величины исполнителей;
— возможно гибкое подключение исполнителей с полным контролем правильности вызовов;
— есть возможность реализации исполнителей на «Си» и их подключения к «КуМиру».

Е5. КуМир — внешние исполнители

Уже на самом старте проектирования и разработки КуМира было ясно, что в дальнейшем понадобится как совершенствовать курс информатики, так и разрабатывать учебное программное обеспечение для математики, физики и других предметов.

Чтобы КуМир можно было использовать в этой деятельности, а также чтобы обеспечить возможность развития системы (и даже языка), в КуМире была реализована концепция внешних пакетов программ, внешних исполнителей. Пакет программ реализуется на производственном языке «Си» или «C++» по определенным правилам и подсоединяется к системе «КуМир» при ее запуске. Вот несколько примеров таких пакетов — внешних исполнителей, реализованных в комплекте «КуМир»:

1) «Вездеход» — учебный исполнитель, моделирующий вездеход с лазерным дальномером, перемещающийся по ровной местности с произвольно расположенными стенами (задаются ломаными линиями);
2) «Двуног» — учебный исполнитель, которого я уже подробно описывал (см. стр. 314);
3) «FIO» — файловый ввод/вывод;
4) «Гратекс» — пакет программ для работы с графикой и текстами;
5) «Комплексные числа»;
6) «Функции и графики» (см. ниже);
7) «ПланиМир» (см. ниже).

Что дает возможность подключения внешних пакетов с методической точки зрения

Первый аспект — «ничего лишнего»

Допустим на первом занятии учитель работает с одним исполнителем Роботом. Тогда он может организовать запуск КуМира только с Роботом. При получении справок ученик будет видеть только информацию о командах Робота, — ничего лишнего. По одной-двум первым буквам команды КуМир сможет автоматически «дописать» команду целиком, и при этом названия команд других исполнителей не будут «мешать».

Для одновременной работы с Роботом и Чертежником к КуМиру при запуске подключаются оба этих исполнителя. Соответственно меняются подсказки, автоматическое продолжение команд и пр.

Если на уроке решается задача о росте суммы вклада (алгоритм А57 учебника), то можно запустить КуМир без внешних исполнителей — простейшие команды ввода-вывода «встроены» в язык. Если же необходимо прочесть какую-то начальную информацию из файла, то можно при запуске КуМира загрузить исполнитель «FIO».

Второй методический аспект — возможность менять обстановку и создавать новых исполнителей

Допустим, учитель решил на уроке геометрии использовать Вездеход, но при подготовке к уроку понял, что имеющаяся в комплекте КуМир программа расстановки стен на поле Вездехода не соответствует целям его урока, что ему нужна другая расстановка. Поскольку программа расстановки стен для Вездехода (как и для Робота) написана на алгоритмическом языке, она может быть легко изменена одним из учеников по указанию учителя. Причем можно модифицировать уже имеющуюся программу, а не писать ее «с нуля».

А как быть, если у учителя возникла идея нового учебного исполнителя? Одно решение — написать на алгоритмическом языке информационную модель нового исполнителя на базе существующих. Это может быть сделано прямо в системе «КуМир», и она даже поможет (если вы этого пожелаете) скрыть детали вашей реализации от учеников. Другое решение — реализовать нового исполнителя на языке «Си»

с соблюдением определенных правил так, чтобы КуМир мог подключить его при запуске.

Третий методический аспект — введение новых типов величин

А теперь предположим, что учитель решил поработать с комплексными числами. Можно, конечно, задавать комплексные числа в виде пары вещественных. Но можно — и это будет гораздо лучше — запустить КуМир с подключением исполнителя «Комплексные числа». При этом в справках (в help-файлах) появится информация о типе «комплексное число» и допустимых операциях над величинами этого типа; можно будет использовать алгоритмы с комплексными аргументами и результатами и др. При вводе алгоритмов будет обеспечен мгновенный полный синтаксический контроль, при пошаговом выполнении результаты присваиваний будут выводиться на поля и т. д. Но вся эта информация и все эти возможности появятся в КуМире только при условии загрузки исполнителя «Комплексные числа». При работе над другими темами она не будет мешать.

Исполнитель «Комплексные числа» входит в состав комплекта «КуМир». Однако учитель может вводить и свои типы величин. Например, для работы с вычетами целых чисел по модулю два можно ввести соответствующий новый тип. Правда, в этом случае придется программировать на Си с соблюдением определенных требований (средствами КуМира новый тип величин ввести нельзя). Но создавать придется только нового исполнителя («Вычеты по модулю 2»), система «КуМир» потом настроится автоматически.

Е6. КуМир — реализация учебной системы с нуля

Простенькая учебная система может быть создана в Кумире с нуля. Вот описание одной такой системы, которую вы можете предложить реализовать школьникам, используя исполнитель «Гратекс». (А.Г. Кушниренко и С.Ю. Оревков в свое время написали такую систему за вечер. Ее объем — меньше 300 строк на алгоритмическом языке.) Система называется «Фазовое пространство» и предназначена для иллюстрации некоторых понятий аналитической геометрии и алгебры.

Описание системы «Фазовое пространство»

На экране постоянно изображены две плоскости:
- плоскость ***точек*** с координатами (x, y) в левой половине экрана,
- плоскость ***прямых*** с координатами (a, b) в правой половине экрана.

Если указать мышкой точку (A, B) на плоскости прямых и нажать правую кнопку, то в плоскости точек (x, y) появится нарисованная прямая y = A $\times$ x + B. Если двигать мышку при нажатой правой кнопке, то прямая на плоскости точек будет меняться в соответствии с изменением координат (A, B). При отпускании кнопки прямая исчезнет.

При нажатии левой кнопки точка (A, B) в правой плоскости и соответствующая ей прямая в левой плоскости будут нарисованы (и останутся при отпускании кнопки).

Если указать мышкой точку (X, Y) на плоскости точек и нажать правую кнопку, то в плоскости прямых (a, b) будет нарисована прямая с уравнением $b = -a \times X + Y$. Если двигать мышку при нажатой правой кнопке, то прямая будет меняться в соответствии с изменением (X, Y). При отпускании кнопки прямая исчезнет.

При нажатии левой кнопки точка (X, Y) в левой плоскости и соответствующая прямая в правой плоскости будут нарисованы (и останутся при отпускании кнопки).

Эта система позволяет проиллюстрировать тот факт, что множество невертикальных прямых в заданной плоскости само может рассматриваться, как плоскость с координатами (a, b). Множество всех возможных состояний какого-то объекта в математике и физике называется ***фазовым пространством*** этого объекта. В нашем случае объект — это невертикальная прямая на плоскости (x, y), а фазовое пространство — плоскость с координатами (a, b).

Работа с системой состоит в решении нескольких приведенных ниже задач. Перед решением каждой из них по нажатии на клавишу F10 обе плоскости очищаются. Следующие клавиши позволяют изобразить на экране данные, необходимые для решения разных задач:

F1 — в плоскости точек появляется случайная точка;
F2 — в плоскости точек появляется случайная пара точек;
F3 — в плоскости прямых появляется случайная точка;
F4 — в плоскости прямых появляется пара точек;
F5 — в плоскости точек рисуется график $y = x^2$;
F6 — в плоскости точек рисуется график $y = abs(x)$;

F7 — в плоскости точек рисуется график $y = -x^3$.

Задачи, решаемые в системе «Фазовое пространство»

1. Задана точка в левой плоскости (F1). Отметьте на правой плоскости несколько точек так, чтобы соответствующие прямые проходили через заданную точку на левой плоскости. Что можно сказать про расположение этих точек?

2. Задана пара точек в левой плоскости (F2). Нажмите правую кнопки мыши в правой полуплоскости и переместите мышку так, чтобы соответствующая прямая в левой плоскости прошла через заданные две точки. Проделайте это для нескольких пар точек. Сформулируйте эмпирический алгоритм, как надо действовать мышкой в правой плоскости, чтобы провести прямую через две заданные точки в левой плоскости.

3–4. Аналог задач 1–2 с использованием клавиш F3 и F4 и перемещением мышки в левой плоскости.

5. Задана парабола в левой плоскости (F5). Ведите мышку вдоль параболы и периодически нажимайте левую кнопку мыши. В правой плоскости окажется «заштрихованным» прямыми некоторое множество М. Опишите границу этого множества. Переместите мышь в правую плоскость и нажмите на левую кнопку мыши на границе множества М, внутри М, снаружи М. Как будут расположены соответствующие прямые в левой плоскости по отношению к параболе?

6. Аналог задачи 5 для функции $y = abs(x)$ (клавиша F6).

7. Задан график функции $y = -x^3$ (F7). Нажмите правую кнопку мыши в правой плоскости и попробуйте переместить мышку так, чтобы соответствующая прямая в левой плоскости касалась кривой $y = -x^3$. В этом положении нажмите левую кнопку мыши. В правой плоскости будет отмечена точка (A, B), а в левой — прямая $y = A \times x + B$, касающаяся графика $y = -x^3$. (В этой точке уравнение $x^3 + A \times X + B = 0$ имеет корень.) Найдите еще несколько таких точек. Попробуйте составить уравнение кривой, на которой расположены все эти точки.

(Ответ: $4 \times a^3 + 27 \times b^2 = 0$. Это выражение называется дискриминантом кубического уравнения $x^3 + a \times x + b = 0$. Когда дискриминант положителен, уравнение имеет один корень, а когда отрицателен — три).

Е7. КуМир — система «функции и графики»

Большую и сложную программную систему в КуМире — из-за ограничений на объем программы — реализовать нельзя. Но можно разбить систему на две части: базовую и интерфейсную. Базовую часть можно реализовать на Си как внешнего исполнителя, подключаемого к Кумиру при запуске. После реализации и отладки эта базовая часть уже не нуждается в изменении. Интерфейсная же часть может быть реализована на алгоритмическом языке в самом КуМире с тем, чтобы потом ее было легко менять. Именно таким образом была реализована система «Функции и Графики», поддерживающая одноименную книгу [Гельфанд].

Е8. КуМир — система «КуМир-гипертекст»

Гипертексты, то есть тексты с дополнительной структурой, использовались задолго до появления компьютеров. Примерами гипертекстов могут служить толковый словарь, в котором одни словарные статьи ссылаются на другие, или энциклопедии, где набраны специальным шрифтом слова, значение которых можно найти в этой же энциклопедии. Совокупность научных статей всех математических журналов также представляет собой гипертекст — в каждой статье есть ссылки на другие статьи. С появлением компьютеров работа с гипертекстами настолько упростилась, что гипертекстовое представление информации стало преобладающим.

Компьютерный гипертекст — это обычный текст с картинками, в котором выделены — как правило цветом — отдельные знаки, слова, фразы или картинки, называемые ***полями.*** С каждым полем в гипертексте (при нажатии на кнопку мышки) связано определенное действие, например:

— переход в другое место того же гипертекста;
— переход в определенное место другого гипертекста;
— проигрывание фрагмента звукозаписи или видеозаписи;
— запуск на выполнение заданной программ и т.д.

Снабдив гипертекст некоторым количеством программ, можно подготовить так называемый ***активный гипертекст***, содержание и поведение которого зависят от работы человека с этим гипертекстом. Для записи и программирования гипертекстов используются специальные языки. Система

«КуМир-Гипертекст» позволяет создавать активные гипертексты, используя школьный алгоритмический язык и систему программирования «КуМир».

В комплекте учебных гипертекстов «НовоМир» есть гипертексты, знакомящие с Роботом и Чертежником, обучающие конструкциям алгоритмического языка. Есть и небольшой задачник: школьник решает задачу, а специальная программа (также написанная на алгоритмическом языке в «КуМире») проверяет решение и сообщает об ошибках.

В 1997 году в рамках развития системы «КуМир-Гипертекст» начаты работы по переводу этой системы на мировой стандарт HTML.

Е9. КуМир — система «ПланиМир»

Основным объектом любой геометрической конструкции, задачи или теоремы является чертеж. «ПланиМир» позволяет ученику строить чертежи на экране компьютера (т. е. на плоскости — планиметрия) с помощью мышки. Результаты построения запоминаются, но в виде программы, а не картинки. Эта программа управляет внешним исполнителем «PlaniMir», который собственно и рисует картинку на экране. Система «ПланиМир» устроена так, что программа выполняется несколько десятков раз в секунду. Когда ученик с помощью мышки меняет положение каких-то элементов чертежа, мгновенно изменяется программа, а следовательно — и картинка на экране. Если, например, ученик построил чертеж треугольника с тремя медианами, то при изменении (с помощью мышки) положения одной из вершин треугольника он увидит «мультфильм» — непрерывное изменение треугольника, при котором три медианы продолжают пересекаться в одной точке.

Таким образом, результатом решения задачи на построение является программа, описывающая построение. Правильность этой программы может быть проверена с помощью заранее подготовленной программы на школьном алгоритмическом языке.

Именно так организован практикум по решению задач на построение с использованием систем «КуМир-Гипертекст» и «ПланиМир».

Е10. Алгоритмы и программы. Алгоритмизация и программирование

В заключение, от вопросов, связанных с конкретным программным обеспечением, вернемся к еще одному часто задаваемому фундаментальному вопросу о взаимосвязи алгоритмизации и программирования в нашем курсе.

Я уже говорил, что в первых школьных учебниках [Ершов] алгоритмы противопоставлялись программам. Алгоритмизацией называлась творческая деятельность по придумыванию алгоритма, т. е. «существа» вычислений. Программированием называлась техническая работа по формальной записи на одном из языков программирования уже придуманного и неформально записанного (на алгоритмическом языке) алгоритма.

С превращением школьного алгоритмического языка в один из языков программирования вопрос о сходстве и различии понятий «алгоритм» и «программа» возник вновь. После долгого и тщательного изучения этого вопроса мы вынуждены были констатировать, что «неформальных», или «никак не записанных» алгоритмов не бывает. Есть разные способы записи алгоритмов, но нет алгоритмов вне всяких форм записи.

Я, конечно, имею ввиду формальное понятие алгоритма, а не «идею алгоритма», понимаемую на бытовом уровне. (Точно так же «энергия» в физике — это строго определенная величина, которую можно измерить. Говоря в быту «энергичный человек», мы имеем в виду отнюдь не количество «физической» энергии, которое может быть извлечено из жировых запасов человека.)

Так вот, если говорить о строгом формальном понятии алгоритма, то это понятие ***в точности совпадает*** с понятием программы. Термин «алгоритм», правда, возник задолго до термина «программа» и поэтому применяется обычно не к любой программе, а лишь к программам с ***ярко выраженной идеей***, как правило, ***математической***. Так, например, можно говорить об алгоритме Евклида, алгоритме нахождения корня непрерывной функции методом деления отрезка пополам и т. п.

Конечно, можно попробовать записать алгоритм Евклида в разных нотациях, в разных системах обозначений, а потом сказать, что “алгоритм” — это то общее, что останется, если отрешиться от формы записи. В наши дни большинство философов, логиков и математиков верит в так называемый

тезис Черча, который состоит в том, что такое понятие алгоритма законно, что все такие интуитивные понятия последовательного детерминированного алгоритма совпадают.

Но для наших целей — для преподавания информатики — такое понятие бесполезно. Ведь именно курс информатики призван выработать у ученика то интуитивное представление об алгоритме, на которое опирается этот подход. А не зафиксировав нотацию, не зафиксировав форму записи, невозможно ни показать пример алгоритма, ни составить алгоритм, ни проверить решение.

Поэтому в нашем курсе и в нашем учебнике, в отличие от первых учебников А.П. Ершова, алгоритмом называется просто соответствующая конструкция алгоритмического языка от **алг** до **кон**. Или, другими словами, ***алгоритм — это программа на алгоритмическом языке***. Слова «алгоритмизация» и «программирование» мы считаем ***синонимами***. А технический перевод программы (или алгоритма) с одного языка на другой мы называем «кодированием» (если это делает человек) или «компиляцией» (если перевод осуществляет компьютер).

Естественно, нас часто обвиняют в подмене информатики и/или алгоритмизации программированием. При этом в слово «программирование» вкладывается негативный оттенок «технической» деятельности из первых учебников А.П. Ершова.

Беда в том, что до сих пор очень часто в разных курсах учат конкретному языку программирования, а не программированию как таковому. В наших аналогиях — это все равно, как если бы мы пытались научить человека решать математические задачи, изучая устройство ручки и бумаги, мела и доски. Я не буду здесь повторять все, что говорил о целях и методах построения нашего курса, но, надеюсь, вы уже почувствовали разницу между изучением конкретного языка и обучением основам алгоритмизации.

Что же касается призыва «не подменять информатику программированием», то им, к сожалению, слишком часто прикрывают элементарное незнание предмета, помноженное на желание выглядеть специалистом в этой области. Ибо для того, чтобы рассуждать об алгоритмах типа перехода улицы или заварки кофе, говорить о «понятности» алгоритма исполнителю или изучать такое свойство информации как «своевременность», вообще ничего знать не надо. Это даже проще, чем перечислять операторы Бейсика, не говоря уже о том, чтобы учить думать и решать задачи, учить основам алгоритмизации. Лично мне до сих пор не встретилось ни одного учебного пособия, в котором «информатика» бы сильно отличалась от «алгоритмизации» (или «программирования» в нашем смысле слова) и при этом про разницу между «информатикой» и «алгоритмизацией» было бы сказано хоть что-нибудь содержательное, имеющее отношение к какой-нибудь науке.

Конечно, для работы на компьютере в наши дни человеку приходится применять много сложных программных систем. Для эффективного использования таких систем, нужно определенное время на их изучение, на привыкание, на тренинг. Бывают нужны и специальные курсы по изучению конкретного программного обеспечения.

Но наша точка зрения состоит в том, что образование в области информатики, не может сводится к таким специализированным курсам, к изучению особенностей конкретных систем, а должно обязательно включать курс основ алгоритмизации, который и обсуждался в этой книге.

Литература

[Ершов] Под ред. А.П. Ершова, М.В. Монахова. Основы информатики и вычислительной техники. ч. I и II — М.: Просвещение. 1986 г.

[Ершов-мы-Шень] А.П. Ершов, А.Г. Кушниренко. Г.В. Лебедев, А.Л. Семенов, А.Х. Шень. Основы информатики и вычислительной техники. — М.: Просвещение. 1988.

[Инф] А.Г. Кушниренко, Г.В. Лебедев, Р.А. Сворень. Основы информатики и вычислительной техники. — М.: 1990, 1991, 1993, 1996.

[ПдМ] А.Г. Кушниренко, Г.В. Лебедев. Программирование для математиков. — М.: Наука, 1988.

[Авербух] А.В. Авербух, В.Б. Гисин, Я.Н. Зайдельман, Г.В. Лебедев. Изучение основ информатики и вычислительной техники. Пособие для учителя: — М.: Просвещение, 1992.

[Зайдельман] Я.Н. Зайдельман — газета "1 сентября"

[ДРОФА 9, 10] Под ред. А.Г. Кушниренко, М.Г. Эпиктетова. Кодирование информации. 9–10 класс. — М.: ДРОФА, 1996.

[Гейн] А.Г. Гейн, В.Г. Житомирский, Е.В. Линецкий и др. Основы информатики и вычислительной техники. М.: Просвещение, 1991.

[Каймин] В.А. Каймин и др. Основы информатики и вычислительной техники. Пробный учебник для 9–10 классов средних школ. — М.: Просвещение, 1988–1989.

[Шень] А.К. Звонкин, С.К. Ландо, А.Л. Семенов, А.Х. Шень. Алгоритмика. — М.: Институт новых технологий образования, 1994.

[Шень-2] А.К. Звонкин, С.К. Ландо, А.Л. Семенов, А.Х. Шень. Алгоритмика. — М.: НМУ, 1995.

[Звенигородский] Г.А. Звенигородский. Вычислительная техника и ее применение. М.: Просвещение 1987.

[Валединский] В.Д. Валединский, А.Г. Кушниренко, Г.В. Лебедев и др. Методическая разработка по курсу "Программирование" — М.: Изд-во МГУ, 1991.

[Е-практикум] В.Д. Варсанофьев, А.Г. Кушниренко, Г.В. Лебедев. Е-практикум — программное обеспечение школьного курса информатики и вычислительной техники. — Микропроцессорные средства и системы, N 3, с. 27–32, 1985.

[ЗК] Знакомьтесь: компьютер. — М.: Мир, 1989.

[Майерс] Г. Майерс. Надежность программного обеспечения. — М.: Мир, 1980.

[Вейнценбаум] Дж. Вейценбаум. Возможности вычислительных машин и человеческий разум. — М.: Радио и связь, 1982.

[Вирт] Н. Вирт. Систематическое программирование. Введение. — М.: Мир, 1977.

[Грис] Д. Грис. Наука программирования. — М.: Мир, 1984.

[Дейкстра] Э. Дейкстра. Дисциплина программирования. — М.: Мир, 1978.

[Дейкстра-1] У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. М.: Мир, 1975.

[Кнут] Д. Кнут. Искусство программирования. Том II. — М.: Мир, 1976.

[Лебедев] Г.В. Лебедев. Разработка интерактивных программ на основе принципа непосредственного редактирования информации. — Микропроцессорные средства и системы, N 1, с. 44–46, 58, 1986.

[И&О] Д.В. Варсанофьев, А.Г. Дымченко. Е-86. — Информатика и образование, N 1, с. 72, 1986.

[Лебедев-1] Г. В. Лебедев. О новом учебнике информатики. — Информатика и образование, N 5, 1990.

[Бурцев] С. Бурцев, А. Ефремов, Д. Ефремов, А. Зорич. Комплект учебных миров "КуМир". — № 2, 1992, стр. 15–20; № 3–4, 1992, стр. 11–16.

[КуМир] А. Г. Кушниренко, М. Г. Эпиктетов. КуМир — новое семейство учебных программ. — Информатика и образование, N 1, 1993.

[гипертекст] А. Г. Кушниренко, М. Г. Эпиктетов. Активные гипертекстовые среды на уроках информатики. — Информатика и образование, N 1, 1994.

[Ада] П. Вегнер. Программирование на языке Ада. — М.: Мир, 1983.

[Гельфанд] И.М. Гельфанд. Функции и графики. Изд. 5-е М.: Наука, 1973.

[Паттис] Pattis R.E. Karel — the robot, a gentle introdiction to the art of programming with Pascal — New-York, Wiley, 1981.