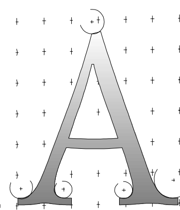
№ 6 (199) февраль 1999

еженедельное
ПРИЛОЖЕНИЕ К ГАЗЕТЕ
«ПЕРВОЕ СЕНТЯБРЯ»

ИНФОРМАТИК

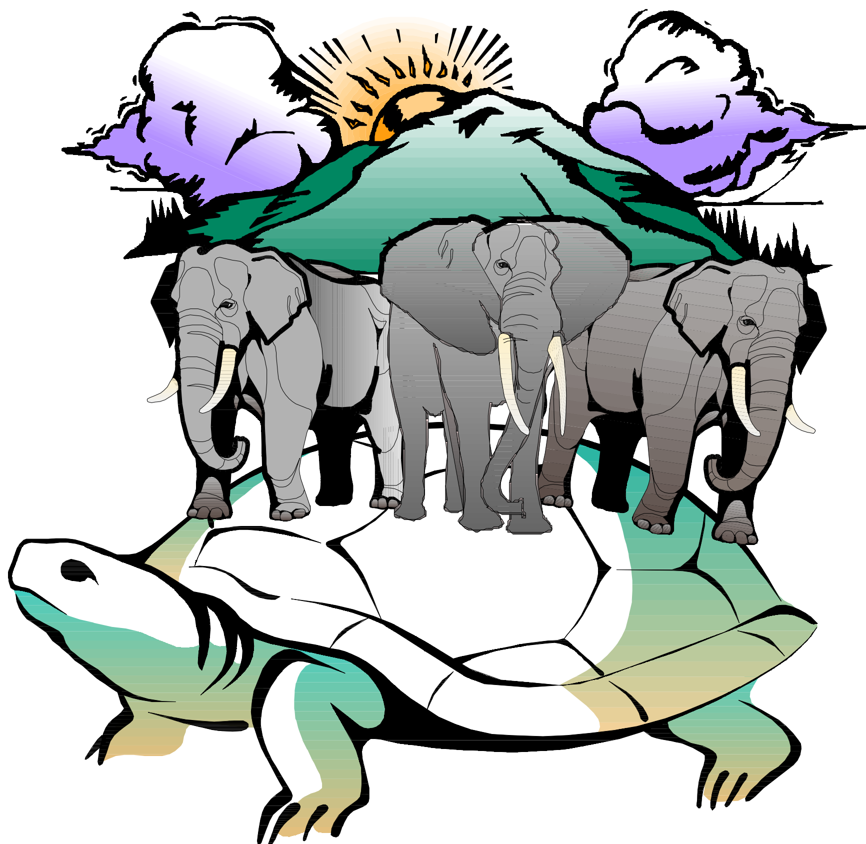


12 лекций

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

Лекции 4, 5

А.Г. КУШНИРЕНКО,
Г.В. ЛЕБЕДЕВ



Введение,
Лекции 1, 2, 3
были опубликованы
в № 1, 3, 5/99

Книга готовится к изданию в издательстве “Дрофа”.
Выражаем признательность издательству “Дрофа” за содействие в подготовке публикации.

© ИнфоМир. Печатается с сокращениями.

СОДЕРЖАНИЕ

Предисловие

Введение

Лекция 1
- А. Основные цели, или Три “кита” курса
- А1. Главная цель курса – развитие алгоритмического стиля мышления
- А2. Курс должен быть “настоящим”
- А3. Курс должен формировать адекватное представление о современной информационной реальности

Лекция 2
- В. Методика построения курса
- В1. “Черепаха” курса – все познается через работу
- В2. Проблемный подход
- В3. Выделение алгоритмической сложности “в чистом виде”
- С. Общий обзор учебника
- С1. Распределение материала в учебнике
- С2. Понятие исполнителя в курсе и учебнике
- С3. Относительная важность и сложность материала в учебнике
- С4. Несколько слов о месте курса в школьном образовании

Лекция 3
- Введение
- § 1. Информация и обработка информации
- § 2. Электронные вычислительные машины
- § 3. Обработка информации на ЭВМ
- § 4. Исполнитель Робот. Понятие алгоритма
- § 5. Исполнитель Чертежник и работа с ним

Лекция 4
- **§ 6. Вспомогательные алгоритмы и алгоритмы с аргументами** 3
- **§ 7. Арифметические выражения и правила их записи** 8
- **§ 8. Команды алгоритмического языка. Цикл n раз** 8

Лекция 5
- **§ 9. Алгоритмы с “обратной связью”. Команда пока** 13
- **§ 10. Условия в алгоритмическом языке. Команды если и выбор. Команды контроля** 26

Лекция 6
- § 11. Величины в алгоритмическом языке. Команда присваивания
- § 12. Алгоритмы с результатами и алгоритмы-функции
- § 13. Команды ввода-вывода информации. Цикл для
- § 14. Табличные величины
- § 15. Логические, символьные и литерные величины

Лекции 7–8
- § 16. Методы алгоритмизации

Лекция 9
- § 17. Физические основы вычислительной техники
- § 18. Команды и основной алгоритм работы процессора (программирование кода)
- § 19. Составные части ЭВМ и взаимодействие их через магистраль
- § 20. Работа ЭВМ в целом

Лекция 10
- § 21. “Информационные модели”, или по-другому — “Кодирование информации величинами алгоритмического языка”
- § 22. Информационные модели исполнителей, или Исполнители в алгоритмическом языке

Лекция 11. Применения ЭВМ
- § 23. Информационные системы
- § 24. Обработка текстовой информации
- § 25. Научные расчеты на ЭВМ
- § 26. Моделирование и вычислительный эксперимент на ЭВМ
- § 27. Компьютерное проектирование и производство
- § 28. Заключение

Лекция 12
- D. Заключение
- D1. Методики преподавания курса
- D2. Место курса в “большой информатике”
- D3. Место курса в школе
- D4. О программном обеспечении курса
- E. Послесловие (разные замечания, отступления, рекомендации и пр.)
- E1. Рекомендуемая литература
- E2. Как возник Робот
- E3. Как возник школьный алгоритмический язык
- E4. История возникновения системы КуМир
- E5. КуМир – внешние исполнители
- E6. КуМир – реализация учебной системы с нуля
- E7. КуМир – система “Функции и графики”
- E8. КуМир – система “КуМир-гипертекст”
- E9. КуМир – система “Планимир”
- E10. Алгоритмы и программы. Алгоритмизация и программирование

Литература

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.: Лекция 1, № 1/99, с. 11).

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 для усвоения, но вы легко можете эту сложность снять, приведя какой-нибудь бытовой аналог. Например, вы можете сказать, что основной и вспомогательный алгоритмы — 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. пример в п. 6.3 учебника).

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

> "В общем случае если в записи алгоритма *X* встречается вызов алгоритма *У*, то алгоритм *У* называется вспомогательным для *X*, а алгоритм *X* называется основным для *У*".

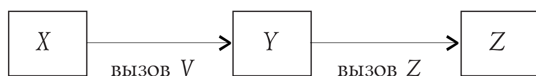
Догматический подход нуждается в подобного рода опорных точках: определение можно продиктовать, заставить выучить наизусть, повесить на большом плакате в кабинете информатики. Но вы должны понимать, что это будет всего лишь "реклама" понятия "вспомогательный алгоритм", демонстрация его важности. Освоение 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 заготовлен еще и алгоритм "буква У", что позволяет составлять разные основные алгоритмы, например, "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, *У* может быть вспомогательным по отношению к *X*. Но если внутри *У* мы вызываем *Z*, то *Z* будет вспомогательным для *У*, а *У* — основным для *Z*. Если рисовать в виде стрелочки, что *X* вызывает *У* и *У* вызывает *Z*, то получится такая картинка:

---

* *Кушниренко А.Г., Лебедев Г.В., Сворень Р.А.* Основы информатики и вычислительной техники. М.: Просвещение, 1990, 1991, 1993, 1996.



В вызове $X \rightarrow Y$ X является основным, а Y — вспомогательным, но в вызове $Y \rightarrow Z$ 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 [1], 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 "закрашивание блока" и "обход" и задача на составление второго реализована в гипертексте "Вспомогательные алгоритмы" в системе КуМир. Ниже приведены копии экрана, соответствующие головной странице гипертекста "Вспомогательные алгоритмы" и одной из задач описанного выше типа:

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 представить себе, как конкретно проводить эти крупные шаги (как их реализовать, т.е. записать). Нам достаточно лишь предполагать, что они выполнимы, что ***как-то*** нам их ***потом*** записать удастся.

Основной выигрыш тут в том, что после записи общего плана можно думать независимо над каждым крупным шагом. 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 пропедевтикой метода последовательного уточнения.

Разделение труда между ЭВМ и исполнителями.

Тут нам нужно снова вернуться к схеме программного управления, к этапу выполнения 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 ЭВМ выполняет этот текст и командует одним или несколькими исполнителями (например, Роботом и Чертежником).

ЭВМ + алгоритм → команды → Исполнители

Исполнители → обратная связь → ЭВМ + алгоритм

Самое важное здесь — сказать, что это новое понятие вспомогательного алгоритма относится к левому квадратику 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).

```
алг квадрат
┌─────────────────────────────┐
│ алг вещ а                   │
│ ┌─────────────────┐         │
│ │        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 5*а* и 5*б*, если делать их без вспомогательных алгоритмов, приходится писать слишком много команд. За счет введения вспомогательных алгоритмов объем текста можно значительно сократить. В задаче 5*в* можно использовать как вспомогательные алгоритмы, полученные при решении задач 5*а* и 5*б*.

Задача 6*а* (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 6*б* 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 неоднозначен. Например, в задаче на *рис.* 24*а* 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.* 24 *б*) 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 неизвестно 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.

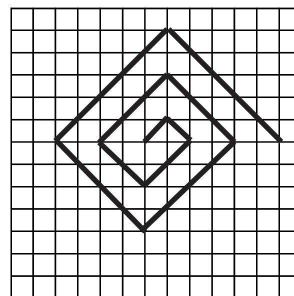
Это чрезвычайно важная задача, потому что для ее решения необходимо использовать понятия основного и вспомогательного алгоритмов, но нельзя задействовать общее представление о задаче (которого нет). Кроме того, задача очень наглядно показывает, что мы можем заставить ЭВМ 25 раз сделать нечто, даже не 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 [2]. Я лишь скажу, что в задачах 23 *е, ж, з* есть хитрости. В 23 *е* и *з* картинка получается наложением пересекающихся фрагментов, как в задаче 6. Кроме того, в задачах 23 *ж, 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 для 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. 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 понадобится.

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.

Чтобы записывать формулы "в строчку", имеются несложные правила — вы их можете найти в учебнике. Практически по тем же правилам записываются формулы и во всех других языках программирования. Различия очень незначительны. Например, в Бейсике возведение в степень обозначается знаком "^" (уголок вверх), а не двумя звездочками "**", как у нас и в большинстве других языков.

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. Если задействован компьютер, то и цель линейной записи становится понятной, и правила записи компьютер проконтролирует. Впрочем, и без компьютера материал проходится и легко, и с интересом. Да и ошибок школьники, как правило, не делают. За исключением единственной 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 <u>раз</u>**

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

Методика введения цикла **n <u>раз</u>**.

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 6*а* 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 правило, 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

$$(x + (x+y)^5)^{10}$$

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 будем считать, 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 вы-

полнении 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 не много, 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 (арг цел 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 в первоначальное 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.

It’s staying anyway!

ИнфоМир

Добро пожаловать на сервер

http://www.math.msu.su/InfoMir

- *Бесплатные программы*
- *Демонстрационные версии*
- *Обновление для зарегистрированных пользователей*
- *Условно-бесплатное (Shareware) программное обеспечение*
- *Новый редактор МикроМир для Windows 95*

Заявка

Я хочу приобрести для индивидуального использования

КуМир+НовоМир (250 руб.) ____экз.

МикроМир (150 руб.) ____экз.

Мой адрес:______________________________

Информатика. Зима-99

Заявки направляйте по адресу: 125167, Москва, а/я 4. “ИнфоМир”

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 (см. Лекция 1, № 1/99, с. 6), лампочку здесь надо представлять себе одноцветной. Скажем, нажимаем на кнопку **снизу стена**? — если лампочка зажглась (**да**) — 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 жерт-

вовать 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.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. Вот как выглядит экран при работе в системе КуМир в режиме, когда поле Робота не изображается:

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

Я немного забегу вперед и сразу скажу, что “оснащение” Робота “термометром” и “счетчиком Гейгера” — хороший методический подход при решении большого класса стандартных задач информатики, обыч-

но 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 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 его изучении методикой 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 вообще), 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 начале параграфа 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 и этого параграфа был использован мною при изложении методических основ курса (см.: Лекция 2, № 3/99).

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 говорят: “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.

ИСТОРИЧЕСКОЕ 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. Этот 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 — то**), 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 встретятся 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 Робота.

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.

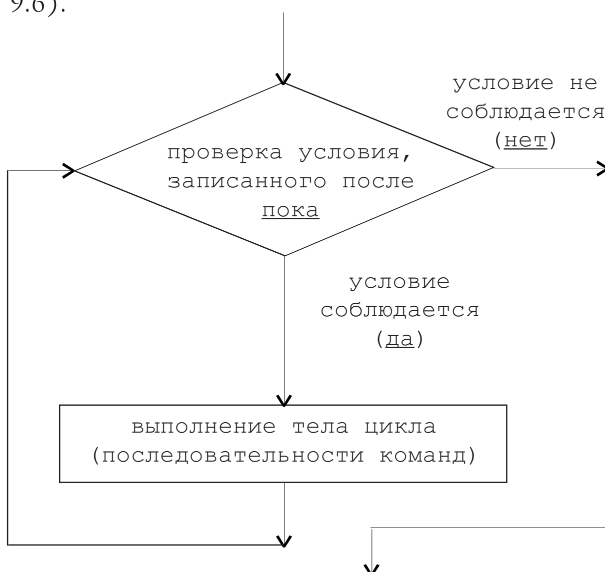
Заметьте, что раньше ученик был автором (впрочем, нигде не записанного) алгоритма, мог гордиться, если алгоритм отработал правильно, и нес ответственность, если по его вине Робот разбивался о стену. Теперь же — в роли ЭВМ — ученик выполняет механическую работу, гордиться ему нечем и отвечать он ни за 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, 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 выполниться 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 вначале 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 А.П. 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 [3], 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 "Знакомьтесь: компьютер" [4] 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) содержит, кроме примеров алгоритмов, 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 демонстрируются применение и метода последовательного 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 четыре 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 неизвестно, 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 закончиться. Обращаю ваше внимание на то, что каково бы ни было тело цикла, что бы мы ни написали внутри, цикл закончится, когда 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 алгоритм, записываем цикл **пока** для исполнения на ЭВМ, 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", понять, 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.

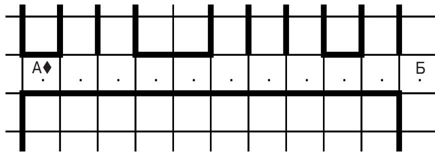
4) Ну и, наконец, нужно убедиться, что не произойдет "зацикливания", что после некоторого числа повторений тела цикла условие, записанное после **пока**, перестанет выполняться, т.е. цикл рано или поздно закончится.

Эта методика составления алгоритмов с циклом **пока** сначала ***иллюстрируется*** на очень простом примере с закрашиванием коридора (п. 9.12), а потом ***применяется*** для составления 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 соответствующее условие (точнее, его отрицание) на алгоритмическом языке. Если бы нам это не удалось, то 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 [1], с. 60.

Теперь (этап 3) надо формально записать тело цикла, т.е. ту последовательность команд, которую нужно 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 сместится 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.

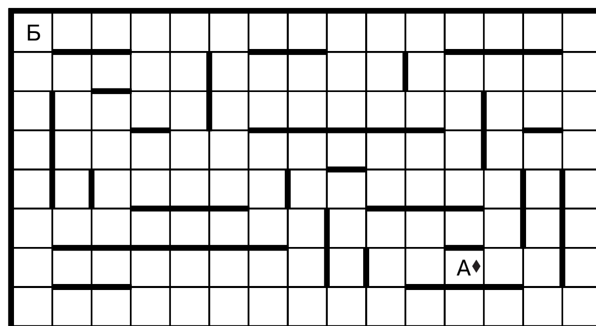
Пример содержательной задачи — выход в левый верхний угол лабиринта.

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 самим. 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 сместится 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 “намного”. 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 [2]. 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 7*а*—7*г*.

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 [1].

В то же время задачи на закраску прямоугольника (упр. 2, 7*д*) 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 являются 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 я сказать не могу. 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 неформализованного)

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 ***иллюстративные*** 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≤ x≤ 1** и др. То есть неравенства в алгоритмическом языке записываются так же, 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 перемене) 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 применять Робота, а использовать 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, **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 упражнений есть в методическом пособии [2]. 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.

Список литературы

1. *Кушниренко А.Г., Лебедев Г.В.* Программирование для математиков. М.: Наука, 1988.
2. *Авербух А.В., Гисин В.Б., Зайдельман Я.Н., Лебедев Г.В.* Изучение основ информатики и вычислительной техники. М.: Просвещение, 1992.
3. Под ред. *А.П. Ершова, М.В. Монахова.* Основы информатики и вычислительной техники. ч. I и II. М.: Просвещение. 1986 г.
4. Знакомьтесь: компьютер. М.: Мир, 1989.

Новый учебник

“ИНФОРМАТИКА 7–9”

(авторы: *Кушниренко А.Г., Лебедев Г.В., Зайдельман Я.Н.*)

Новый учебник информатики, подготовленный авторским коллективом под руководством А.Г. Кушниренко, продолжает линию курса школьной информатики, начатую в 1985 году академиком А.П. Ершовым. Он во многом опирается на предыдущий учебник этого коллектива (*Кушниренко А.Г., Лебедев Г.В., Сворень Р.А.* Основы информатики и вычислительной техники. М.: Просвещение, 1990) и развивает его основные идеи с учетом новых реалий сегодняшнего дня.

Учебник в основном соответствует программе курса, разработанной его авторами (см. № 39/98).

Основное внимание в учебнике уделено общим принципам обработки информации, независимым от конкретной технической базы. Подробно рассмотрены понятия алгоритма, исполнителя, информационной модели, составляющие ядро информатики как научной дисциплины. Это делает учебник машинно-независимым и позволяет работать с ним, используя любые компьютеры, имеющиеся в российских школах.

По сравнению с предыдущим вариантом учебника существенно расширена часть, посвященная алгоритмизации. Более подробно рассмотрены вопросы анализа и тестирования алгоритмов, увеличено количество упражнений, в текст включены примеры решения задач.

В то же время авторы стремились не противопоставлять информатику как науку и прикладные информационные технологии, а объединить их в рамках единого курса и учебника. В учебнике рассмотрены вопросы компьютерной обработки текстов, построения электронных таблиц и баз данных, составляющие основу современных информационных технологий. Большое место уделено работе компьютерной сети Интернет. При этом пользовательский подход сочетается с рассмотрением базовых теоретических понятий, основной упор делается не на конкретные особенности отдельных программных средств, а на общие принципы, лежащие в основе любых систем сходного назначения.

Отличительная особенность учебника – ориентация на активную самостоятельную работу школьников. Каждая тема сопровождается упражнениями – от простых учебных заданий, позволяющих сформировать и закрепить знания, умения и навыки, до сложных творческих задач, некоторые из которых по уровню приближаются к олимпиадным задачам.

Работа с учебником предполагает активное усвоение информатики, причем уровень сложности заданий может регулироваться в соответствии с реальными возможностями учеников – учебник предоставляет достаточный объем учебных заданий для такого выбора.

В настоящее время учебник готовится к публикации в издательстве “Дрофа” и выйдет в свет в 1999 году.

Конференции и семинары для преподавателей

(сообщение Министерства образования РФ)

Дата проведения	Мероприятие	Телефон, электронный адрес
ежегодно		
март	6-я Международная научно-техническая конференция “Новые информационные технологии в университетском образовании” в Новосибирском гос. университете и НИИ математико-информационных основ обучения	(3832) 35-62-37 факс 35-78-08
март	Конференция-выставка “Геоинформатика и образование” в ИОСО РАО	(095) 246-16-59 e_kapral@gisa.msk.ru
апрель	Региональная научно-практическая конференция “Информационные технологии в школьном образовании” в гимназии г. Переславля-Залесского	(08535) 2-06-08 kon@gorono.botik.ru
апрель	Международный научно-практический семинар “Информационные технологии — пути внедрения в районной образовательной системе” в Санкт-Петербургском межшкольном учебном комбинате	(812) 232-24-04 nadya@nin.emissia.spb.su
апрель	Международный научно-технический семинар “Школьная информатика и развитие” в Санкт-Петербургском гос. университете аэрокосмического приборостроения	(812) 210-70-94 bob@aanet.ru
апрель	Международный научно-технический семинар “Новые информационные технологии в образовании” в Белгородском гос. университете	(0722) 34-00-17
апрель	Электронная Всероссийская научно-практическая конференция “Проблемы информатизации” в Воронеже, ГПУ и ВШ МВД	kravets@avs.vsu.ru
май	Международный конгресс-выставка “Образование-99” в “Совинцентре”. По вопросу участия обращаться: вузам — в ЦИ, педагогическим вузам, остальным учреждениям образования — в РОСФОКОМП	(095)237-62-85 root@interphys.msk.ru
май	Международная научная конференция “Информатика — современное состояние и перспективы развития” в рамках 52-х Герценовских чтений кафедры информатики и вычислительной техники Санкт-Петербургского гос. пед. университета	(812) 219-17-46
май	Международная конференция “Дистанционное образование”. Проводит Московский гос. университет экономики, статистики и информатики в г. Твери	(095) 442-84-33
июнь	Ежегодная конференция Российской академии информатизации образования	(095) 925-45-49
июнь	Всероссийская конференция “Телематика-98”. Проводится Вузтелекомцентром в Санкт-Петербургском государственном региональном образовательном центре	(812) 238-87-79 tm98@spb.runnet.ru
июнь — июль	Международная научно-практическая конференция “Применение новых технологий в образовании”, г. Троицк Московской области	(095) 334-03-67 cue@bytic.troitsk.ru

В следующем номере читайте

ТЕЛЕКОММУНИКАЦИИ

• ИНТЕРНЕТ ДЛЯ НАЧИНАЮЩИХ

А.А. ДУВАНОВ

В этом учебном году в рамках Роботландского сетевого университета открылся курс “Введение в Интернет” (о чем сообщалось в № 35/98).

Мы публикуем фрагмент учебника “Интернет для начинающих” (первоначально задуманного как учебник для Роботландского университета).

В № 36, 40, 44, 46/98; 2, 5/99 были помещены первые шесть параграфов книги, теперь представлен следующий параграф с рассказом о том, как посылать и принимать “вложения” (вложением может быть любой файл, а значит, и текст, и картинки, и музыка, и видео, и игры, и другие программы).

Продолжение следует

МАТЕРИАЛЫ К УРОКУ

• ЛОГОМИРЫ В ЕКАТЕРИНБУРГЕ: ГОД РАБОТЫ

Н.А. ЮНЕРМАН, Н.А. БАРХАТОВА, Л.Б. БЕЛКИНА, В.И. КАДОЧНИКОВА, И.П. ТОКАРЕВА, И.А. ШЕЛЕГОВА

Обучающая программная среда ЛогоМиры разработана на базе языка Лого, созданного С.Пейпертом для “комплексного развития образного, логического и алгоритмического компонентов мышления”.

О методике использования среды ЛогоМиры для занятий с учащимися разных “возрастных категорий” (младших школьников и учащихся среднего звена) и для разных форм учебной работы.

КРОССВОРДЫ

Составил *В.Г. ФЕДОРИНОВ*

“КРУГЛЫЙ СТОЛ”

• НЕСКОЛЬКО СЛОВ ОБ ИНФОРМАТИКЕ, “COMPUTER SCIENCE”, ИТ И ШКОЛЬНЫХ УЧЕБНИКАХ

Ю.А. ШАФРИН

Продолжение дискуссии “О судьбах школьной информатики”, которая, конечно, не потеряла актуальности.

Спецвыпуск

Информатика для малышей

Пермская версия курса информатики для начальной школы

“Информатика для малышей” будет состоять из следующих модулей:

1. Информация.
2. Словари. Каталоги. Организация текста.
3. Организация информации (таблицы, диаграммы, карты и пр.).
4. Введение в логику.
5. Введение в системологию.
6. Принятие решений. Учет противоречивых требований.

Модуль 2 относительно автономен, в том смысле, что он не опирается непосредственно на материал модуля “Информация”. Это дает возможность изучать его независимо, причем такое автономное изучение возможно как в рамках курса “Информатика для малышей”, так и позднее, в средней школе.

Модуль

СЛОВАРИ. КАТАЛОГИ. ОРГАНИЗАЦИЯ ТЕКСТА

М.А. ПЛАКСИН

Книга для учителя

“Введение”, “Содержание модуля”, “Некоторые организационные моменты”, “Примерное поурочное планирование”, уроки 1—5.

Тетрадь для ученика

Задания и упражнения, соответствующие первым пяти урокам.

Продолжение следует

Гл. редактор
Е.Б.Докшицкая
Зам. гл. редактора
С.Л.Островский
Редакция:
Л.Н.Картвелишвили,
Ю.А.Соколинский,
Н.Л.Беленькая,
Н.П.Медведева
Дизайн и компьютерная верстка:
Н.И.Пронская
Корректоры:
Е.Л.Володина,
С.М.Подберезина

Windows95

©ИНФОРМАТИКА 1999
выходит четыре раза в месяц
При перепечатке ссылка на ИНФОРМАТИКУ обязательна, рукописи не возвращаются

121165, Киевская, 24
тел. 249 4896
Отдел рекламы
тел. 240 1041

ИНДЕКС ПОДПИСКИ
для индивидуальных подписчиков 32291
комплекта приложений 32744

Internet: infosef@glasnet.ru
Fidonet: 2:5020/69.32
WWW: http://www.glasnet.ru/~infosef
FTP: ftp://ftp.glasnet.ru/users/infosef

Объединение педагогических изданий "ПЕРВОЕ СЕНТЯБРЯ"

Отпечатано в типографии “Пресса”

125865, Москва, ул. Правды, 24
Тираж 7000 экз.
Заказ №