Владимир ПАРОНДЖАНОВ

ЯЗЫК ДРАКОН

КРАТКОЕ ОПИСАНИЕ

В краткой форме изложены наиболее важные идеи языка ДРАКОН.

В данном материале использованы главы из еще не опубликованной книги.

Если Вы желаете прочитать более полное описание, скачайте книгу «В.Д. Пароджанов. Как улучшить работу ума: Алгоритмы без программистов — это очень просто! — М.: Дело, 2001. — 360с».

http://forum.oberoncore.ru/viewtopic.php?p=21078#p21078

Обсуждение языка ДРАКОН смотрите на форуме (там же можно скачать дракон-редактор):

http://forum.oberoncore.ru/viewforum.php?f=62

СОДЕРЖАНИЕ

Глава 1

ИЗЮМИНКИ ЯЗЫКА ДРАКОН

Графический язык является главным средством достижения наглядности

Константин Гомоюнов

ВВЕДЕНИЕ

В этой части (главы 1—5) дано популярное (неформальное) описание языка ДРАКОН. Этот язык построен путем значительной доработки и улучшения блок-схем алгоритмов (flowcharts) [1].

Для обозначения блок-схем, построенных по правилам языка ДРАКОН, используется термин «дракон-схемы».

ПРЕИМУЩЕСТВА ДРАКОН-СХЕМ

Чем же отличаются дракон-схемы от блок-схем?

Блок-схемы [1] не обеспечивают автоматическое преобразование алгоритма в машинный код. Дракон-схемы, напротив, пригодны для формализованной записи, автоматического получения кода и исполнения его на компьютере.

Однако более важным является второе (когнитивное) отличие. Хотя блок-схемы порою действительно улучшают понимаемость программ, однако это происходит не всегда, причем степень улучшения невелика. Кроме того, есть немало случаев, когда неудачно выполненные блок-схемы запутывают дело и затрудняют понимание. В отличие от них дракон-схемы удовлетворяют критерию *сверхвысокой понимаемости.*

Благодаря использованию специальных формальных и неформальных когнитивных приемов дракон-схемы дают возможность изобразить решение любой, сколь угодно сложной процедурной проблемы в предельно ясной, наглядной и доходчивой форме.

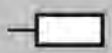
Это позволяет значительно сократить интеллектуальные усилия персонала, необходимые для разработки и отладки алгоритмов и программ.

ИКОНЫ И МАКРОИКОНЫ

Графоэлементы (графические буквы) языка ДРАКОН называются *иконами* (рис. 4). Подобно тому, как буквы объединяются в слова, иконы объединяются в составные иконы — *макроиконы* (рис. 5).

Соединяя иконы и макроиконы по определенным правилам, можно строить разнообразные алгоритмы, примеры которых показаны на рис. 6—9, 13, 15, 17—20, 22, 23, 25.

Шампур-блок — часть дракон-схемы, имеющая один вход сверху и один выход снизу, расположенные на одной вертикали. Примерами шампур-блоков являются иконы И3, И5 — И10, И13 — И16, И18, И20 — И22 (рис. 4) и макроиконы 2—20 (рис. 5).

	Икона	Название иконы
И1		Заголовок
И2		Конец
И3		Действие
И4		Вопрос
И5		Выбор
И6		Вариант
И7		Имя ветки
И8		Адрес
И9		Вставка
И10		Полка
И11		Формальные параметры
И12		Начало цикла ДЛЯ
И13		Конец цикла ДЛЯ

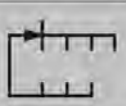
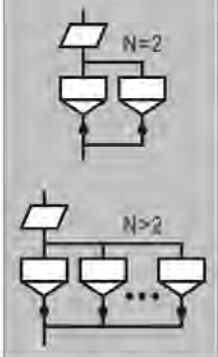
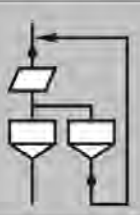
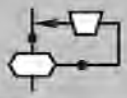
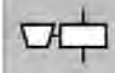
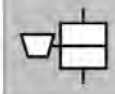
	Икона	Название иконы
И14		Вывод
И15		Ввод
И16		Пауза
И17		Период
И18		Пуск таймера
И19		Синхронизатор (по таймеру)
И20		Параллельный процесс
И21		Комментарий
И22		Правый комментарий
И23		Левый комментарий
И24		Петля цикла
И25		Петля силуэта

Рис. 4. Иконы языка ДРАКОН

	Макроикона	Название макроиконы
1		Заголовок с параметрами
2		Развилка
3		Переключатель (число вариантов N>2)
4		Обычный цикл
5		Переключающий цикл
6		Цикл ДЛЯ
7		Цикл ЖДАТЬ
8		Действие по таймеру
9		Полка по таймеру

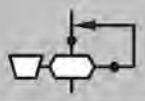
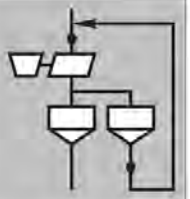
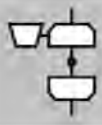
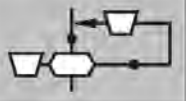
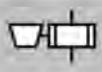
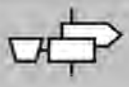
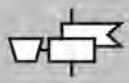
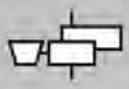
	Макроикона	Название макроиконы
10		Развилка по таймеру
11		Переключатель по таймеру
12		Обычный цикл по таймеру
13		Переключающий цикл по таймеру
14		Цикл ДЛЯ по таймеру
15		Цикл ЖДАТЬ по таймеру
16		Вставка по таймеру
17		Вывод по таймеру
18		Ввод по таймеру
19		Пуск таймера по таймеру
20		Параллельный процесс по таймеру

Рис. 05. Макроиконы языка ДРАКОН

ЗАЧЕМ НУЖНА ВЕТКА?

Когда принцесса Анна развелась с маркизом Ришелье, возник спор о разделе имущества. Судья потребовал указать: какие покупки принцесса сделала до замужества, а какие после.

А теперь забудем об этой семейной драме и сравним между собой рис. 6 и 7. Легко видеть, что рис. 6 не позволяет ответить на вопрос судьи. Зато рис. 7, наоборот, содержит нужную информацию. Более того, алгоритм на

рис. 7 нарочно нарисован так, что покупки, сделанные до и после замужества, четко разделены на два столбика. Такой прием называется делением алгоритма на смысловые части. А сами части называются *ветками.*

Чтобы лучше вникнуть в суть дела, разберем еще один пример. На рис. 8 представлен алгоритм «Сборы на рыбалку», содержащий довольно большую последовательность действий. Иной раз такая последовательность может оказаться ужасно длинной и утомительной для чтения. Можно ли облегчить восприятие и анализ подобных «длиннющих» алгоритмов? Можно ли сделать «долговязый» алгоритм обозримым и удобным для быстрого понимания?

Да, можно. Чтобы облегчить работу читателя и сделать алгоритм дружелюбным, разработчик дракон-схемы должен заблаговременно разрезать «длинную кишку» и разбить ее на смысловые части.

Сделать это нетрудно. «Сборы на рыбалку» (рис. 8) — это алгоритмический рассказ, в котором можно выделить три крупных куска, три самостоятельных темы.

- Подъем и завтрак
- Укладка вещей
- Поездка

Каждую тему можно нарисовать в виде ветки. Результат изображен на рис. 9. Назначение икон пояснено в таблице.

Икона	Название иконы	Пояснение
	Имя ветки	Обозначает начало ветки
	Адрес	Обозначает конец любой ветки, кроме последней

Вопрос. Где начало и конец у первой ветки на рис. 9?

Ответ. Начало — икона «Подъем и завтрак». Конец — «Укладка вещей». Между началом и концом размещается тело ветки. Оно содержит команды «Встань пораньше» и «Позавтракай».

Что такое ветка

Это смысловая часть алгоритма, которая содержит:

икону «имя ветки» (в ней пишут название смысловой части)

тело ветки, состоящее из команд

одну или несколько икон «адрес» (в любой ветке, кроме последней)

икону «конец» (в последней ветке)

Итак, зачем нужна ветка? Чтобы помочь алгоритмисту, программисту и разработчику технологии формализовать смысловое разбиение алгоритма, программы или техпроцесса на части. И, что очень важно, дать частям удобные смысловые названия.

При этом разделение проблемы на *N* смысловых частей реализуется путем разбиения алгоритма на *N* веток.

Следует подчеркнуть, что ветка — составной оператор языка ДРАКОН, который не имеет аналогов в известных языках.

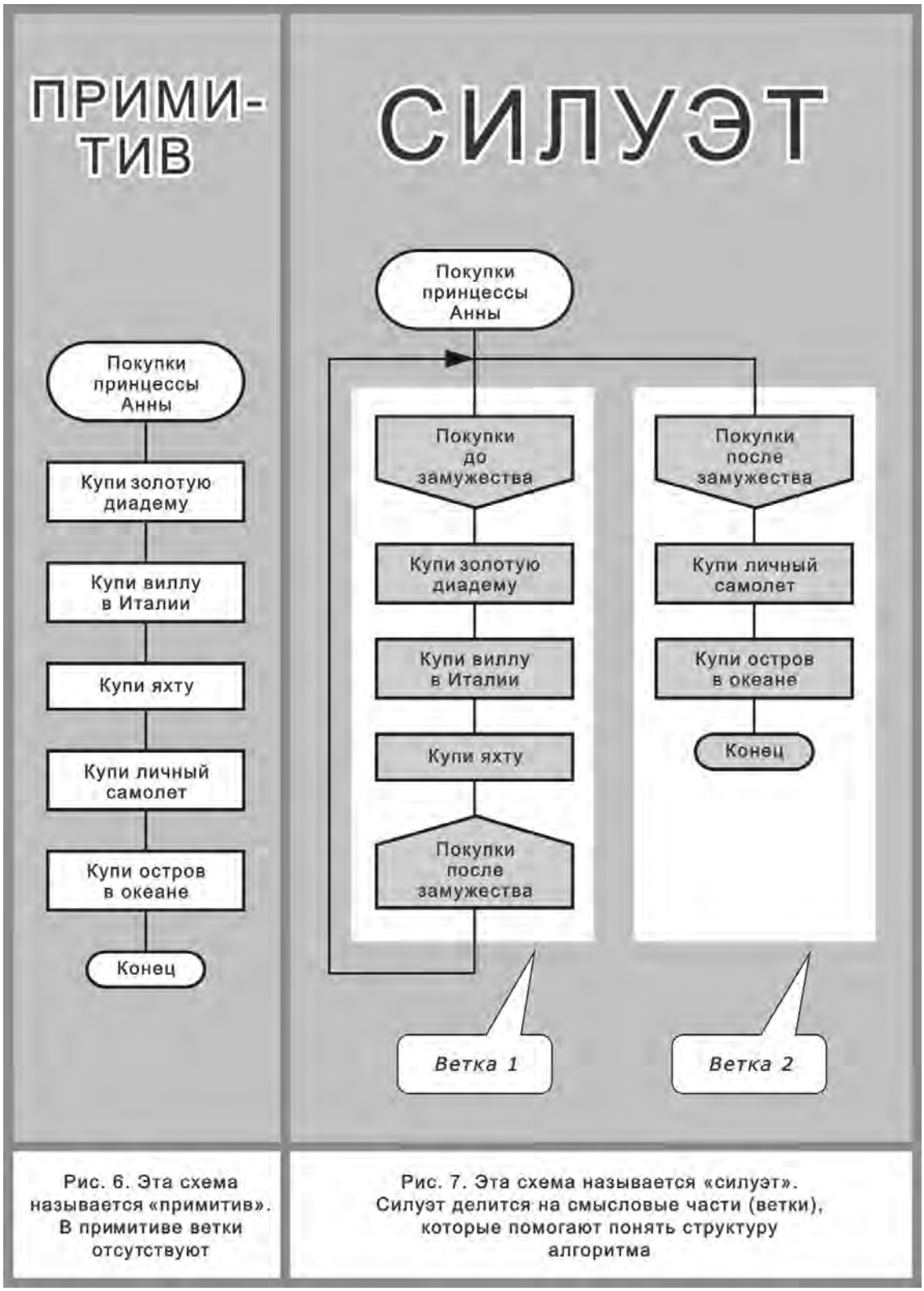


Рис. 6. Эта схема называется «примитив». В примитиве ветки отсутствуют

Рис. 7. Эта схема называется «силуэт». Силуэт делится на смысловые части (ветки), которые помогают понять структуру алгоритма

Рис. 8. Алгоритм «Сборы на рыбалку»

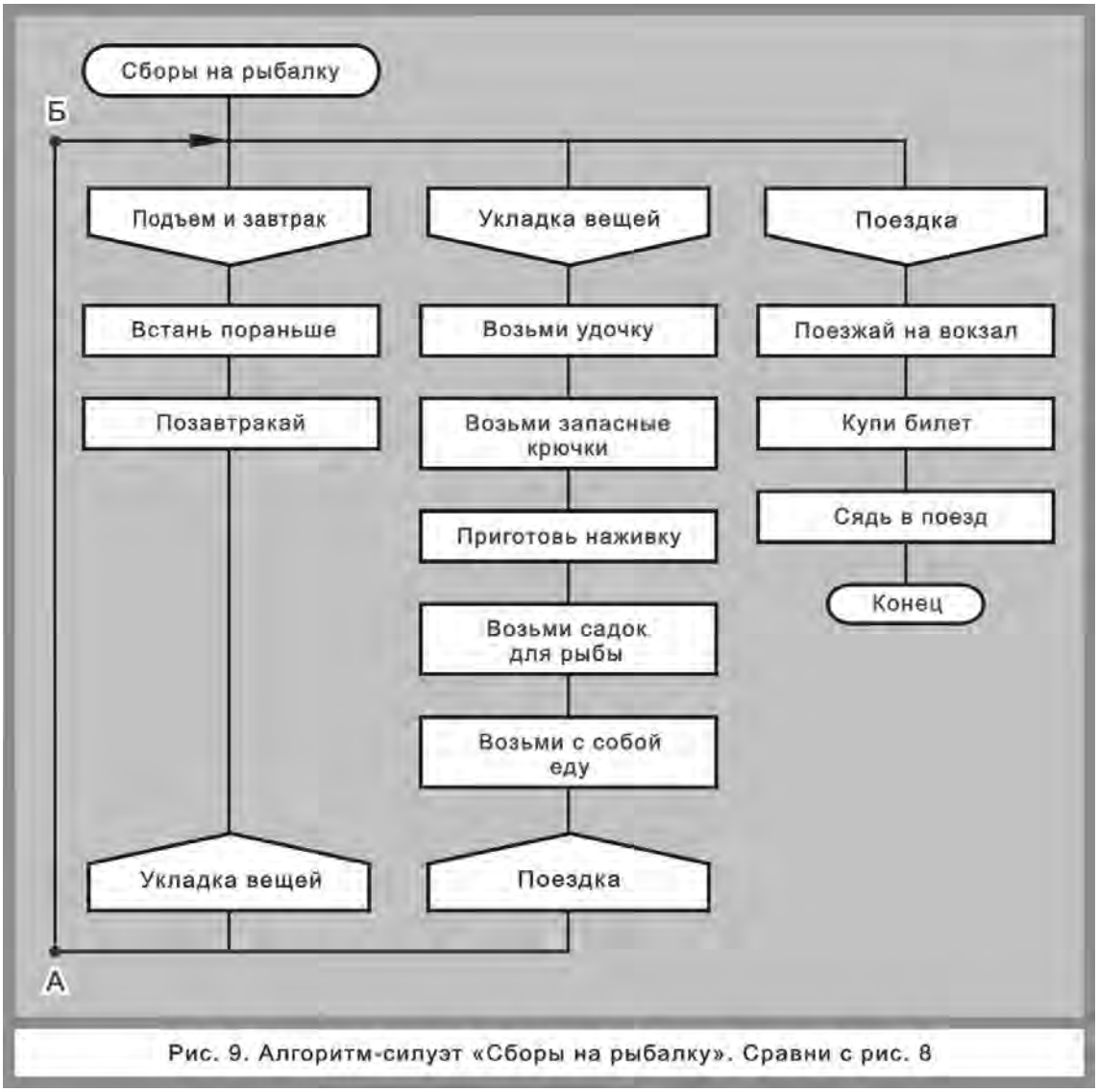


Рис. 9. Алгоритм-силуэт «Сборы на рыбалку». Сравни с рис. 8

КАК РАБОТАЕТ ВЕТКА?

Ветка имеет один вход и один или несколько выходов. Входом служит икона «имя ветки», содержащая идентификатор ветки. Графический оператор «имя ветки» не выполняет никаких действий. Это всего лишь метка, объявляющая название смысловой части алгоритма. Исполнение дракон-программы всегда начинается с крайней левой ветки (рис. 7, 9).

Выходом из ветки служит икона «адрес», в которой записывается имя следующей по порядку исполнения ветки. Икона «адрес» — это замаскированный оператор перехода *(goto).* Однако он передает управление не куда угодно, а только на начало выбранной ветки.

Вход в ветку возможен только через ее начало. Выход из последней ветки осуществляется через икону «конец».

КАК СЛЕДУЕТ РАСПОЛАГАТЬ ВЕТКИ В ПОЛЕ ЧЕРТЕЖА?

Ветки упорядочены двояко: логически и пространственно. *Логическая* последовательность исполнения веток определяется метками, записанными в иконах «адрес».

Однако логический порядок — это еще не все. На рис. 10, 11, 12 показаны три разных способа пространственного расположения веток, которые имеют один и тот же логический порядок.

Чтобы устранить пространственную неоднозначность и облегчить понимание смысла дракон-схемы, вводится

Правило. **Чем правее — тем позже**.

Оно означает: ветка, нарисованная правее, работает позже всех веток, находящихся левее нее.

Алгоритм, нарисованный согласно правилу «чем правее — тем позже», считается хорошим, эргономичным (рис. 12). Схемы, где это правило нарушается, объявляются плохими (рис. 10, 11). Их использование запрещено.

В разрешенных (эргономичных) алгоритмах имеет место следующий порядок работы (рис. 7, 9, 12):

- первой работает крайняя левая ветка, последней — крайняя правая;
- остальные ветки передают управление друг другу слева направо (при этом может случиться так, что некоторые ветки будут пропущены);
- иногда образуется так называемый «веточный цикл». Это происходит, когда в иконе «адрес» указано имя собственной или одной из левых веток. На рис. 13 веточный цикл помечен черными треугольниками.

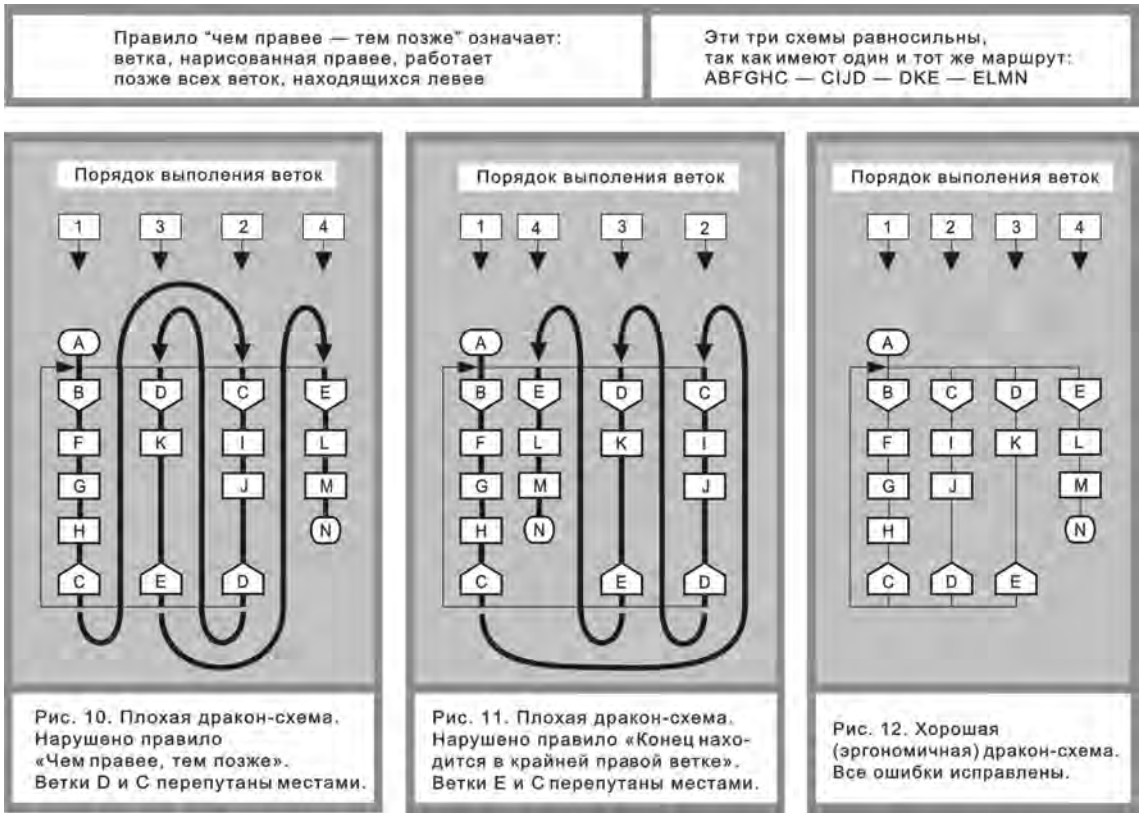


Рис. 10. Плохая дракон-схема. Нарушено правило «Чем правее, тем позже». Ветки D и C перепутаны местами.

Рис. 11. Плохая дракон-схема. Нарушено правило «Конец находится в крайней правой ветке». Ветки E и C перепутаны местами.

Рис. 12. Хорошая (эргономичная) дракон-схема. Все ошибки исправлены.

ЧТО ТАКОЕ ШАПКА?

Многие алгоритмы чрезвычайно сложны. Чтобы разобраться в хитросплетениях такого алгоритма, нужно прилагать огромные усилия и тратить слишком много времени.

Подобную практику следует признать порочной. Алгоритмы можно и нужно сделать эргономичными, легкими для восприятия. Для этого нужно

во время давать читателю маленькие, но умные подсказки, проглотив которые, он мог бы легко сориентироваться в задаче и быстрее понять материал.

Одной из таких подсказок служит «шапка». Название объясняется тем, что шапка находится вверху, «на голове» у алгоритма.

Шапка — верхняя часть дракон-схемы (рис. 9), которая включает заголовок алгоритма и комплект икон «имя ветки».

Назначение шапки — помочь читателю мгновенно (не более чем за несколько секунд или минут) сориентироваться в задаче. Расчленить ее на части, увидеть смысловую структуру.

Причем увидеть не в фигуральном смысле слова, не с помощью воображения, не духовным оком, а своими двумя глазами — на бумаге или экране.

Но как это сделать? Трудность в том, что ни один из существующих языков не предоставляет читателю, изучающему сложный алгоритм или технологию, эффективной помощи, позволяющей моментально (за несколько секунд) уяснить ее структуру, деление на смысловые блоки.

В языке ДРАКОН имеются специальные средства, обеспечивающие решение задачи.

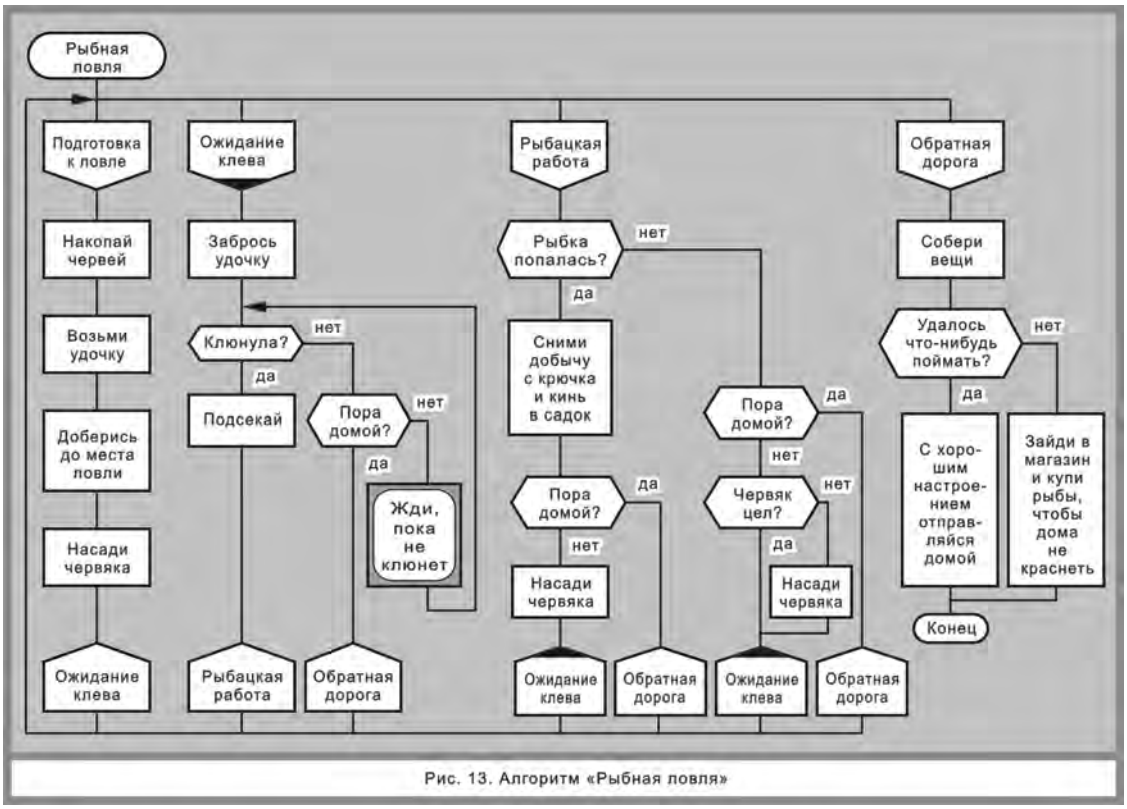


Рис. 13. Алгоритм «Рыбная ловля»

ТРИ ЦАРСКИХ ВОПРОСА

Когда мы сталкиваемся с новой незнакомой задачей, в первую очередь мы желаем получить ответ на три царских (наиболее важных) вопроса:

1. Как называется задача?
2. Из скольких частей она состоит?
3. Как называется каждая часть?

Именно с этих вопросов начинается наше знакомство с любой задачей при рациональном подходе к делу.

Эргономическая хитрость состоит в том, что шапка, угадывая тайное желание читателя, дает ему мощную подсказку — ответ на все царские вопросы.

Вот ответы для рис. 9.

- Как называется задача? *(Читаем заголовок алгоритма).* Сборы на рыбалку.
- Из скольких частей она состоит? *(Считаем иконы «имя ветки»).* Из трех.
- Как называется каждая часть? *(Читаем текст в иконах «имя ветки»).* 1. Подъем и завтрак. 2. Укладка вещей. 3. Поездка.

ТРЕХЭТАПНЫЙ МЕТОД ПОЗНАНИЯ АЛГОРИТМА

Дополнительные эргономические удобства связаны с тем, что шапка занимает «парадное» место в верхней части чертежа. Названия смысловых частей помещены внутри особых рамок уникальной формы, которые легко отыскать взглядом.

Благодаря этому шапка моментально приковывает к себе внимание читателя без всяких усилий с его стороны. Это очень важно, так как читателю не приходится рыскать глазами по темным закоулкам алгоритма, пытаясь выудить нужную информацию.

Таким образом, ДРАКОН предоставляет читателю эффективный трехэтапный метод познания незнакомого или забытого алгоритма.

На первом этапе, анализируя шапку, читатель узнает назначение алгоритма и его деление на смысловые части (ветки). На втором — осуществляет углубленный анализ каждой ветки. На третьем производит разбор взаимодействия веток.

КАК РАБОТАЕТ АЛГОРИТМ-СИЛУЭТ?

Выполнить алгоритм — значит пройти путь от начала до конца алгоритма. Применим это правило к рис. 9. Будем считать, что существует воображаемый бегунок, который при работе алгоритма пробегает алгоритмическую дорожку от иконы «заголовок» до иконы «конец».

Вернемся к вопросу: как работает алгоритм-силуэт? Выехав из иконы «заголовок», бегунок мчится вниз по крайней левой ветке. Он движется через станции (рис. 9):

- Подъем и завтрак
- Встань пораньше
- Позавтракай
- Укладка вещей

Икона «адрес» — последняя станция первой ветки. Куда ехать дальше? Ответ содержится внутри самой иконы. Эта икона потому и зовется «адрес», что сообщает адрес (название) следующей станции. В данном случае она сообщает — следующая станция называется «Укладка вещей».

Из рис. 9 видно, что данная станция находится в начале второй ветки. Но как бегунок доберется туда? По какой линии?

Ответ прост. Выехав из иконы «адрес», бегунок сворачивает налево и попадает в точку *А* (рис. 9). Потом движется вверх к точке *Б*. Затем едет направо по стрелке и въезжает в верхнюю икону «Укладка вещей».

Дальше все происходит аналогично. Бегунок скользит вниз по второй ветке. Добравшись до иконы «адрес», узнает адрес следующей станции («Поездка»). Затем огибает схему по линии *АВ*, попадает в начало третьей ветки и спускается по ней до конца. На этом выполнение алгоритма заканчивается.

В ЧЕМ СЕКРЕТ ИКОНЫ «АДРЕС»?

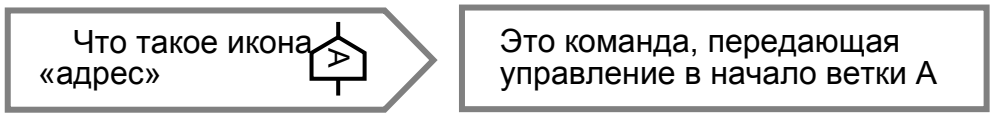
Сейчас мы поступим, как чеховский злоумышленник — будем разбирать рельсы. Имеются в виду линии, окаймляющие алгоритм на рис. 9. Сотрем линию *АБ.* Уберем также горизонтальные линии, проходящие через точки *А* и *Б.* Результат представлен на рис. 14.

Интересно, как будет работать алгоритм после этих исправлений?

К нашему удивлению, отсутствие «рельсов» никак не сказывается на работе алгоритма. В этом легко убедиться. Выехав из иконы «заголовок», бегунок движется вниз по крайней левой ветке. Опустившись до конца ветки, бегунок попадает в икону «адрес».

Казалось бы, это тупик. Рельсы кончились, и дальше ехать некуда. Но это не совсем так. Ведь в иконе «адрес» записан адрес следующей станции («Укладка вещей»). Зная адрес, бегунок прыгнет туда, куда нужно — в начало второй ветки. И поедет вниз. Добравшись до конца второй ветки, он совершит второй прыжок. И попадет в третью ветку. И так далее.

Таким образом, икона «Адрес *А*» — это команда «Прыгни в начало ветки *А*». Или, выражаясь по-научному, «Передай управление в начало ветки *А»*. Проще говоря, данная команда передает приказ: «Брось данную ветку и начни выполнять ветку *А*». (Выполнять ветку — значит исполнять записанные в ней команды).



Переходя от рис. 9 к рис. 14, мы для «упрощения» стерли несколько линий. Теперь мы убедились, что для работы алгоритма они совершенно не нужны. Потому что маршрут бегунка определяют не они, а указания, записанные в иконе «адрес».

Тем не менее, указанные линии не следует удалять *по эргономическим соображениям.* Дело в том, что обрамляющие линии зрительно «склеивают» разрозненные куски алгоритма. Они превращают их в приятный для глаза целостный зрительно-смысловой образ. И наоборот, устранение скрепляющих линий приводит к тому, что схема зрительно рассыпается на части, что мешает увидеть целостный образ. И сбивает с толку читателя (рис. 14).

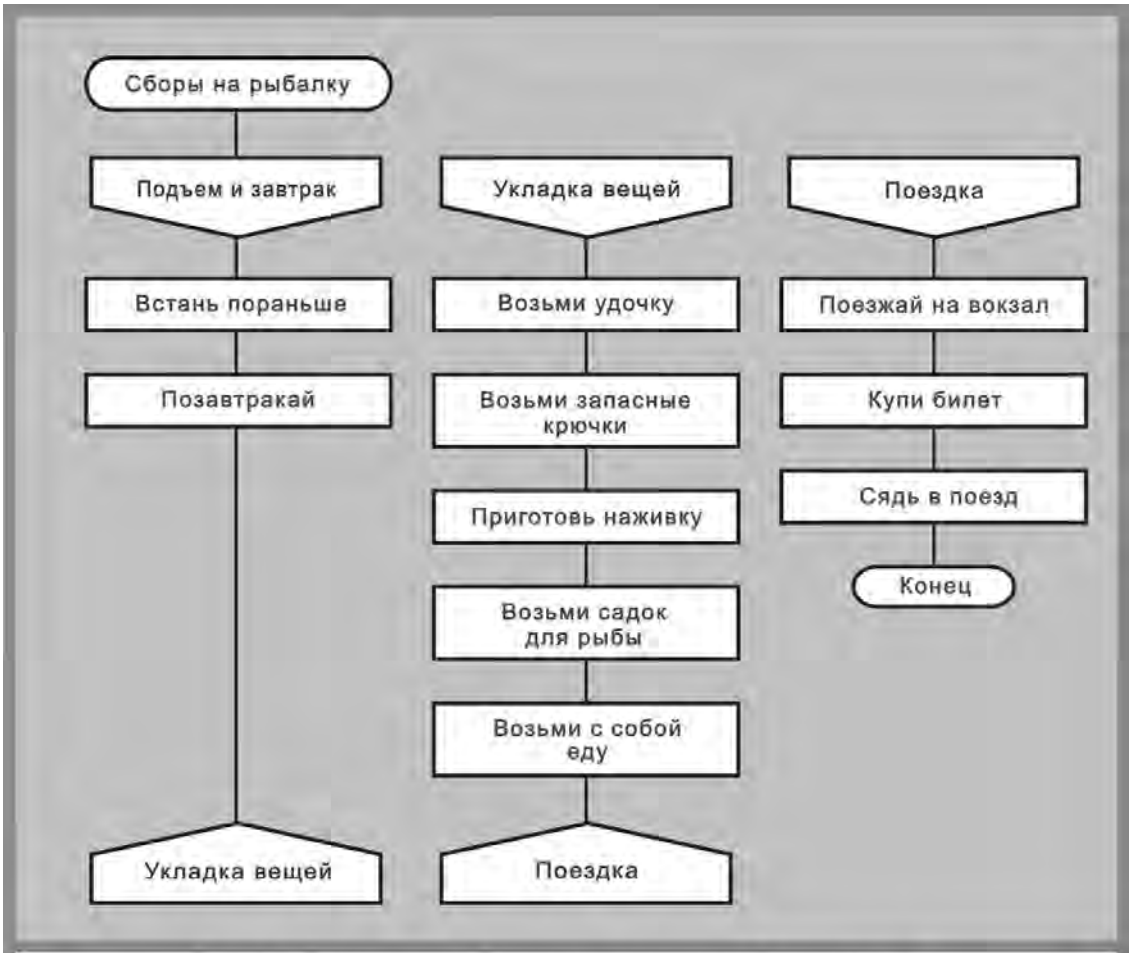


Рис. 14. Схема «с разобранными рельсами». Сравни с рис. 9

ЕЩЕ РАЗ О РАСПОЛОЖЕНИИ ВЕТОК НА ЧЕРТЕЖЕ

В начале главы мы затронули вопрос: как следует располагать ветки в поле чертежа? Учитывая важность темы, коснемся ее еще раз.

Давайте мысленно перетасуем ветки на рис. 14, как колоду карт. И расположим их на чертеже в произвольном порядке. Легко сообразить, что подобное «перепутывание» веток никак не отразится на работе алгоритма. Потому что очередность работы веток зависит только от команд «адрес». И совсем не зависит от расположения веток на листе бумаги. Словом, сколько ветки ни тасуй, получим тот же самый алгоритм.

Здесь есть одна тонкость. Перестановка веток не отражается на правильности алгоритма. Однако алгоритм должен быть не только правильным, но и понятным, доходчивым. Хаотичное расположение веток затрудняет понимание, что недопустимо.

Поэтому нужно обязательно упорядочить ветки в пространстве чертежа. Удобнее всего расположить их слева направо в той последовательности, в какой они включаются в работу. Для этого служит уже известное нам правило: «Чем правее, тем позже».

Отсюда вытекает важная эргономическая

> *Рекомендация.* Чтобы дракон-схема была наглядной и удобной, ветки следует упорядочить слева направо. Более правая ветка должна работать позже, чем любая ветка, расположенная левее нее.

ЧТО ТАКОЕ ШАМПУР?

Шампур — вертикальная линия, соединяющая икону «заголовок» и икону «конец». Между этими иконами обычно помещается несколько других икон. Все они, словно кусочки мяса, оказываются нанизанными на шампур (рис. 15).

Правило шампура. **Выход иконы «заголовок» и вход иконы «конец» должны лежать на одной вертикали.**

Если это правило выполняется, дракон-схема становится более упорядоченной, эргономичной, легкой для чтения (рис. 15). И наоборот, нарушение данного правила делает схему корявой, изломанной, неудобной для глаза (рис. 16).

Рис. 15. *Правильно* нарисованный алгоритм. Иконы «заголовок» и «конец» лежат на одной вертикали. Эта вертикаль называется «шампур» (жирная линия).

Рис. 16. *Неправильно* нарисованный алгоритм. Ошибка в том, что иконы «заголовок» и «конец», лежат на разных вертикалях. Эта ошибка называется «Нет шампура».

ЕСТЬ ЛИ В АЛГОРИТМЕ ЦАРСКАЯ ДОРОГА?

Маршрут — это путь, ведущий от начала до конца алгоритма.

На рис. 8 и 9 показан неразветвленный алгоритм. В нем всего один маршрут. В разветвленном алгоритме на рис. 17 таких маршрутов два.

Если тропинок несколько, среди них можно выделить главный и побочные маршруты.

Главный маршрут алгоритма — путь, который ведет к наибольшему успеху. Например, на рис. 17 главный маршрут проходит по левой вертикали, и дела обстоят хорошо. Суп не пролился на скатерть, все довольны. Другой (правый) маршрут ведет к неудаче (скатерть испачкалась).

Рассмотрим задачу. В запутанном лабиринте, соединяющем начало и конец сложного алгоритма, нужно выделить один-единственный маршрут — «путеводную нить». С ней можно зрительно сравнивать все прочие маршруты, чтобы легко сориентироваться в проблеме и не заблудиться в путанице развилок. Путеводная нить должна быть визуально легко различимой. Бросив беглый взгляд на дракон-схему, мы должны обнаружить четкие ориентиры, позволяющие сразу и безошибочно увидеть царский маршрут и упорядоченные относительно него остальные маршруты. Для этого вводится

> *Правило главного маршрута.* **Главный маршрут алгоритма должен идти по шампуру.**

Это значит, что царский маршрут не может оказаться где-то на задворках дракон-схемы, где его днем с огнем не сыскать. Нет, он всегда должен находиться на самом почетном месте — на крайней левой вертикали. Соблюдение этого правила делает схему зрительно упорядоченной, предсказуемой и интуитивно ясной.

Если правило главного маршрута по каким-то причинам оказалось нарушенным (рис. 18), ошибку надо исправить. Для этого нужно поменять местами слова «да» и «нет» в развилках, а также присоединенные к ним гирлянды икон. Действуя таким путем, всегда можно добиться, чтобы на царском пути оказался тот выход иконы «Вопрос», который ведет к наибольшему успеху.

Рис. 17. Хорошая (эргономичная) схема. Главный маршрут идет по шампуру

Рис. 18. Плохая схема. Нарушено правило «Главный маршрут должен идти по шампуру»

ПОБОЧНЫЕ МАРШРУТЫ НЕЛЬЗЯ РИСОВАТЬ КАК ПОПАЛО

Побочный маршрут — любой маршрут разветвленного алгоритма за исключением главного.

> *Правило побочных маршрутов.* **Побочные маршруты алгоритма нужно рисовать справа от шампура по принципу «Чем правее, тем хуже».**

Это значит: чем правее нарисован побочный маршрут, тем более неприятную ситуацию он описывает.

Вот пример из жизни: «О, ужас! Я, кажется, потерял деньги!»

Кому случалось делать подобное открытие, знает, что степень огорчения зависит от потерянной суммы. Рассмотрим пять возможных ситуаций, и дадим им оценку.

	Ситуация	Оценка
1	Я ничего не потерял	Хорошо
2	Я потерял 100 рублей	Плохо
3	Я потерял 500 рублей	Очень плохо
4	Я потерял 1000 рублей	Совсем плохо
5	Я потерял всю получку	Хуже некуда

На рис. 19 показана дракон-схема, описывающая эту грустную историю. По каждой вертикали идет свой маршрут.

Самая первая, крайняя слева вертикаль — это шампур. По ней идет главный маршрут, имеющий оценку «хорошо», потому что все деньги целы. Чуть правее находится вторая вертикаль (потеряны 100 рублей) с оценкой «плохо». Еще правее идет третья вертикаль (пропали 500 рублей) с оценкой «очень плохо». И так далее. Крайняя справа — пятая вертикаль описывает самую скверную ситуацию, когда потеряна вся получка.

Таким образом, четыре побочных маршрута, идущие по вертикалям 2, 3, 4, 5, расположены не случайно, а со смыслом. Они выстроены слева направо по принципу «Чем правее, тем хуже».

Чтобы алгоритм был понятным, он должен быть стройным, красивым, упорядоченным, то есть эргономичным. Он не должен содержать непредсказуемые и хаотичные хитросплетения линий и икон.

Язык ДРАКОН был разработан, в частности, потому, что традиционные блок-схемы алгоритмов, рекомендованные стандартом [1], с эргономической точки зрения, не выдерживают никакой критики. Они напоминают непроходимые джунгли, в которых легко запутаться и почти ничего нельзя понять. ДРАКОН выгодно отличается тем, что его графический узор подчиняется жестким и тщательно продуманным правилам, которые дисциплинируют мышление, облегчают разработку и отладку алгоритмов.

Чтобы убедиться в этом, взглянем еще разок на рис. 19 и проведем взглядом по всем вертикалям слева направо. Мы обнаружим не хаос, а строгий порядок. Потому что маршруты нарисованы не как попало, а по правилам.

В результате чертеж алгоритма обретает четкую зрительно-смысловую структуру, которая облегчает работу мысли. Читая такой чертеж, человек не станет плутать в потемках. Ведь он заранее знает, что схема алгоритма составлена по мудрому плану: *все хорошее — слева, все плохое — справа.* Про такой алгоритм можно сказать: «Все ясно, как на ладони!».

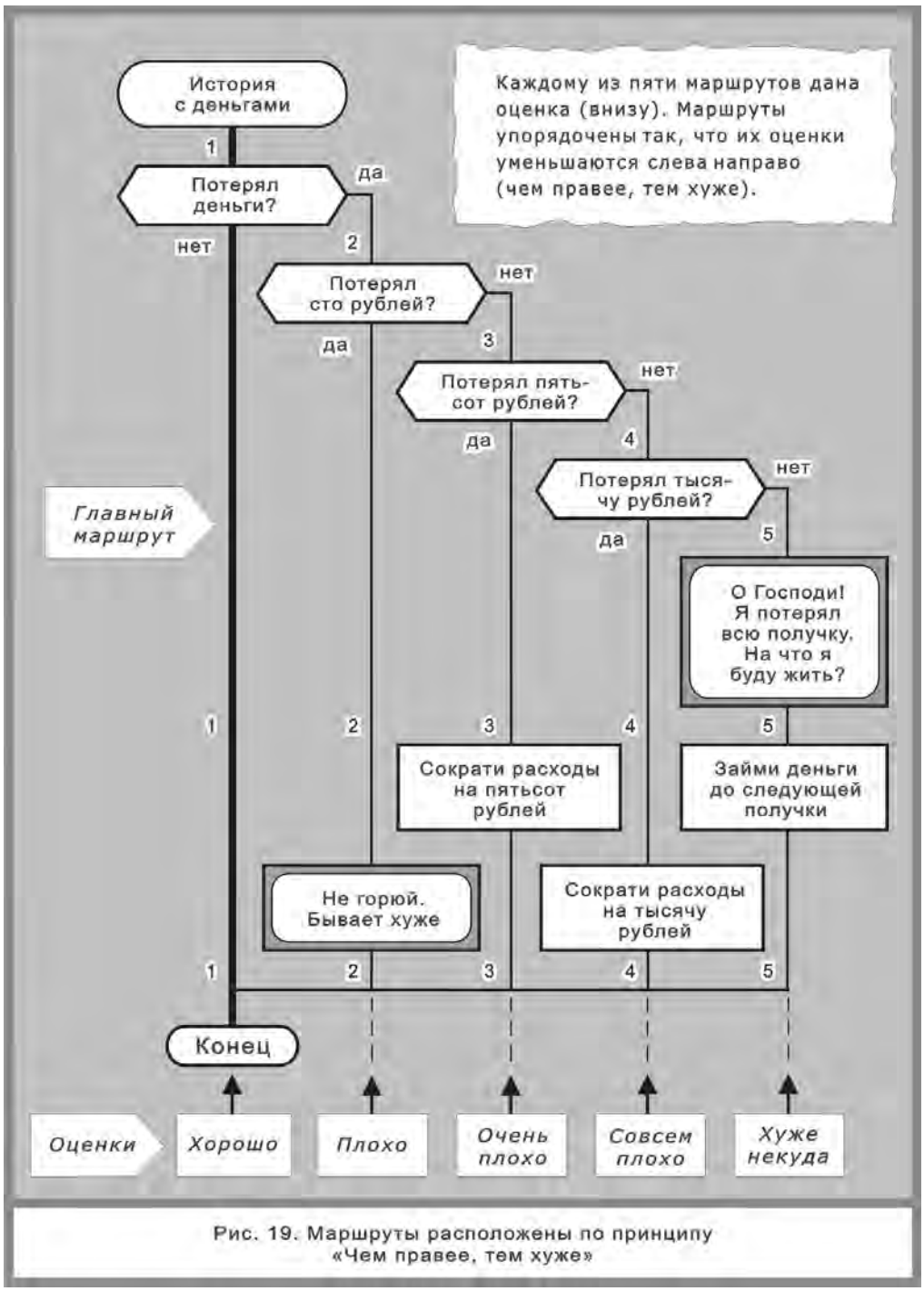


Рис. 19. Маршруты расположены по принципу «Чем правее, тем хуже»

ЧТО ДЕЛАТЬ, ЕСЛИ ПРАВИЛО «ЧЕМ ПРАВЕЕ, ТЕМ ХУЖЕ» НЕ РАБОТАЕТ?

Из физики известно: если начальная скорость ракеты меньше 7,8 километров в секунду (км/с), она непременно упадет на Землю. Если же разогнать ее чуть сильнее, но не больше 11,2 км/с, она станет спутником Земли. При дальнейшем увеличении скорости ракета станет спутником Солнца. А если скорость превысит 16,4 км/с, ракета навсегда простится с Солнечной системой и умчится в безбрежные просторы космоса.

Алгоритм этой задачи показан на рис. 20.

Мы знаем, что каждой вертикальной линии на дракон-схеме соответствует свой маршрут, причем вертикали следует рисовать не хаотично, а упорядоченно. До сих пор мы пользовались правилом «Чем правее, тем

хуже». Однако здесь оно не имеет смысла. Поэтому на рис. 20 выбрано другое правило «Чем правее, тем больше скорость ракеты».

Чтобы понять смысл правила, проведем взглядом по схеме слева направо. Мы увидим, что от маршрута к маршруту скорость неуклонно возрастает. Первый слева маршрут самый медленный. На втором маршруте скорость больше. На третьем — еще больше. Наконец, четвертый (самый правый) маршрут описывает ситуацию, когда ракета с огромной скоростью улетает за пределы Солнечной системы.

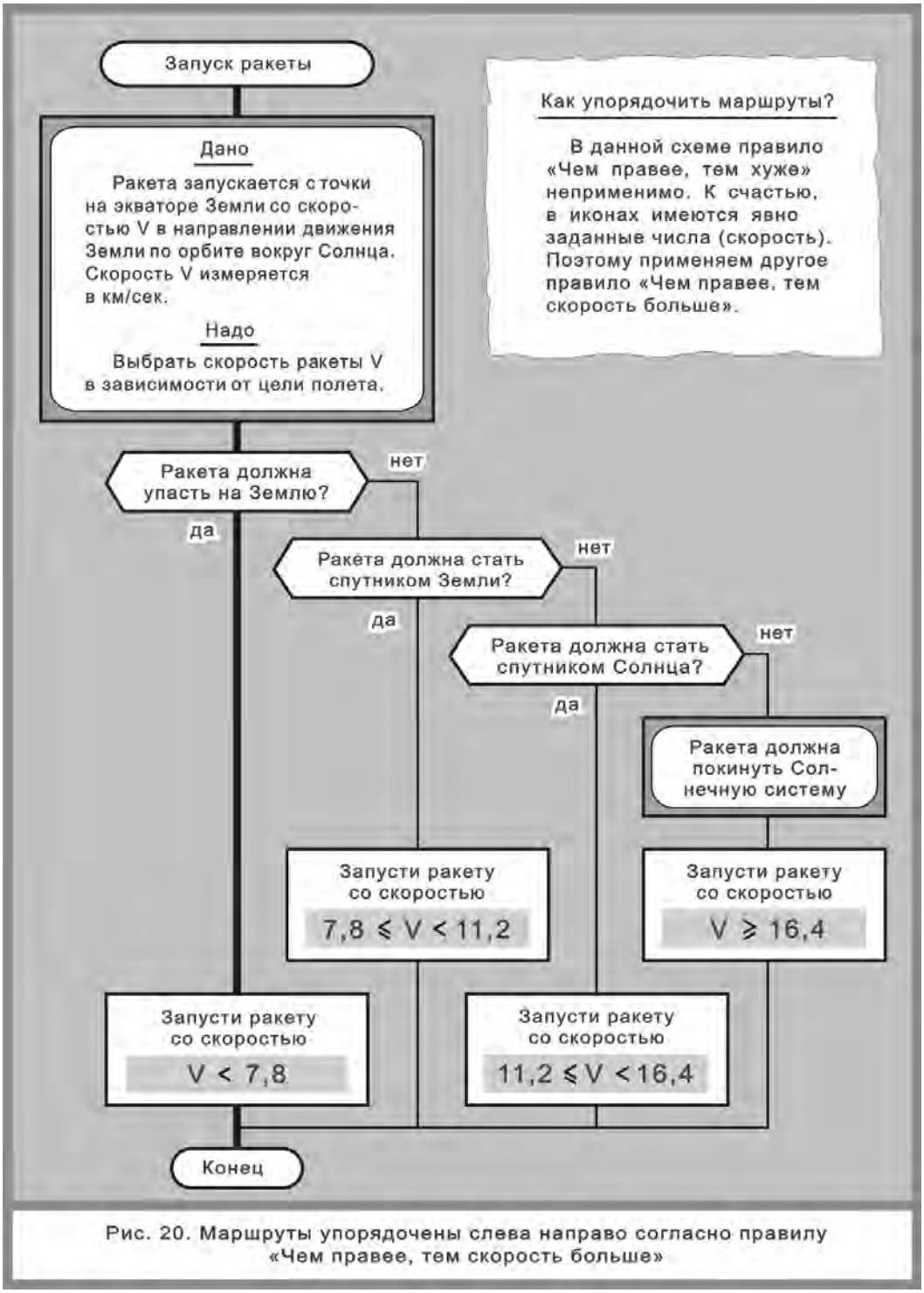


Рис. 20. Маршруты упорядочены слева направо согласно правилу «Чем правее, тем скорость больше»

КАК РИСОВАТЬ ПОБОЧНЫЕ МАРШРУТЫ?

Правило. **Смещение вправо от главного маршрута должно быть не произвольным и хаотичным, а продуманным и логичным.**

Например, при решении математических задач вертикали можно расположить в порядке увеличения или уменьшения математической величины (числа), соответствующей этим вертикалям.

Можно придумать и другие правила, позволяющие сделать схему упорядоченной. Для разных задач могут понадобиться разные правила (рис. 21).

Но у всех правил есть общая черта. Главное, чтобы в схеме был не хаос, а порядок. Здесь действует

Правило хорошей хозяйки. **Если постараться, порядок всегда можно навести.**

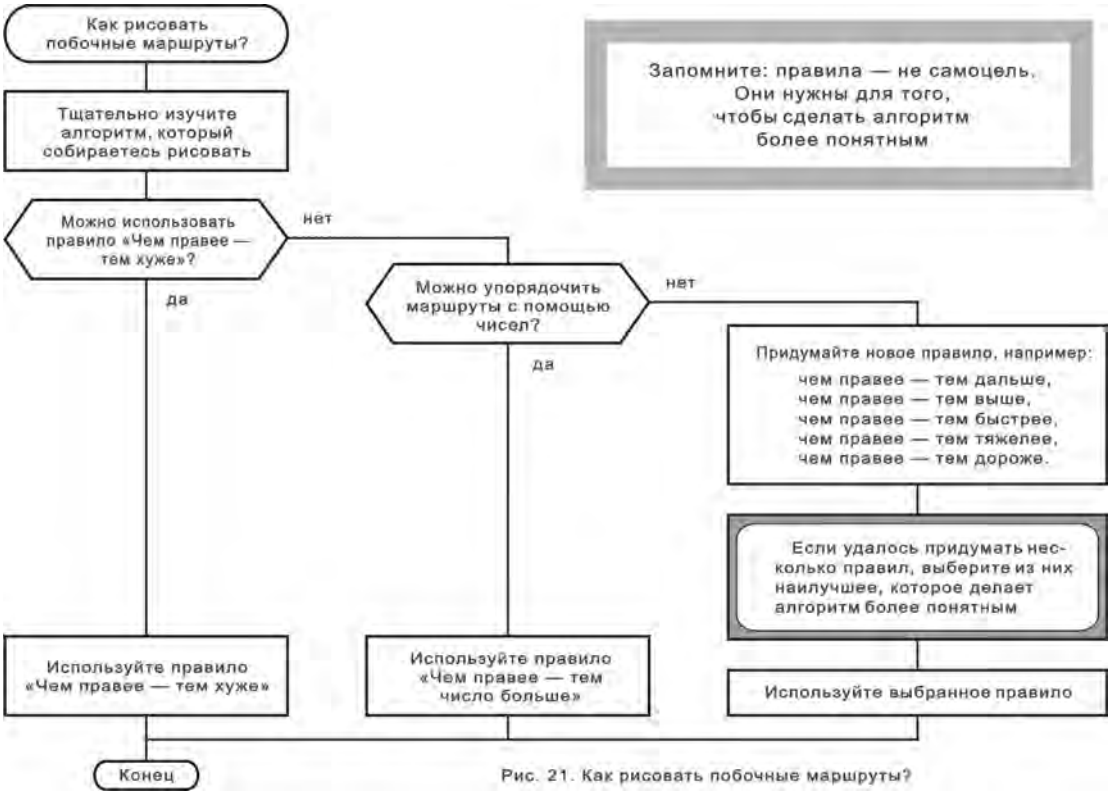


Рис. 21. Как рисовать побочные маршруты?

ЧТО ЛУЧШЕ: ПРИМИТИВ ИЛИ СИЛУЭТ?

Напомним определения понятий, о которых читатель уже мог догадаться из рис. 7 и 8.

Силуэт — дракон-схема, разделенная на ветки.

Примитив — дракон-схема, не имеющая веток.

Сравним их между собой. Какая схема лучше? Какая легче для понимания?

Чтобы уяснить суть дела, возьмем один и тот же алгоритм (Поездка на автобусе). И изобразим его двумя способами: в виде силуэта (рис. 22) и в виде примитива (рис. 23).

Легко сообразить, что силуэт обладает серьезными преимуществами. В самом деле, примитив на рис. 23 не позволяет увидеть деление задачи на

смысловые части. Иное дело рис. 22. Тут сразу ясно — алгоритм состоит из четырех частей:

- Поиск автобуса
- Ожидание посадки
- Посадка в автобус
- Поездка

Таким образом, в силуэте смысловые части алгоритма четко выделены и сразу бросаются в глаза. Они зрительно и пространственно разнесены в поле чертежа, делая его более понятным. А в примитиве смысловые части не выделены и перемешаны (все в одной куче), что затрудняет чтение и анализ сложных алгоритмов.

> *Правило.* **Примитив рекомендуется применять, если дракон-схема очень простая (примитивная) и содержит не более 10 икон. В более сложных случаях выгоднее использовать силуэт.**

Нарушение этого правила обычно чревато неприятностями, ибо мешает читателю выявить сущность решаемой проблемы. И, следовательно, затрудняет и замедляет понимание смысла программы.

Например, алгоритм на рис. 23 выглядит громоздким и неоправданно сложным для восприятия. Это вызвано тем, что он содержит 19 икон, однако изображен в виде примитива. Криминал в том, что схема на рис. 23 не позволяет читателю мгновенно (за несколько секунд) распознать зрительно-смысловую структуру алгоритма.

В самом деле, из скольких частей состоит решаемая проблема? Глядя на рис. 23, ответить на этот вопрос довольно трудно. А быстро ответить — невозможно.

Положение в корне меняется, когда мы смотрим на рис. 22, где тот же самый алгоритм изображен в виде силуэта. Тут деление алгоритма на части «само бросается в глаза». Однако это не все. Не менее важно, что запутанность зрительного рисунка полностью исчезла. И схема приобрела новое эстетическое (эргономическое) качество: элегантность, ясность и прозрачность.

Таким образом, в сложных случаях силуэт позволяет существенно уменьшить интеллектуальные усилия, затрачиваемые на понимание алгоритма. В силуэте крупные структурные части программы (ветки) четко выделены, образуя легко узнаваемый, стабильный, предсказуемый и целостный зрительный образ.

А в примитиве структурные части не выделены и перемешаны, что затрудняет чтение и анализ сложных алгоритмов.

Однако для простых случаев (менее 10 икон) примитив, как правило, оказывается более предпочтительным.

В этой главе рекомендации по использованию силуэта и примитива даны в упрощенном виде.

Окончательные рекомендации даны в конце главы 5 в такой же рамке.

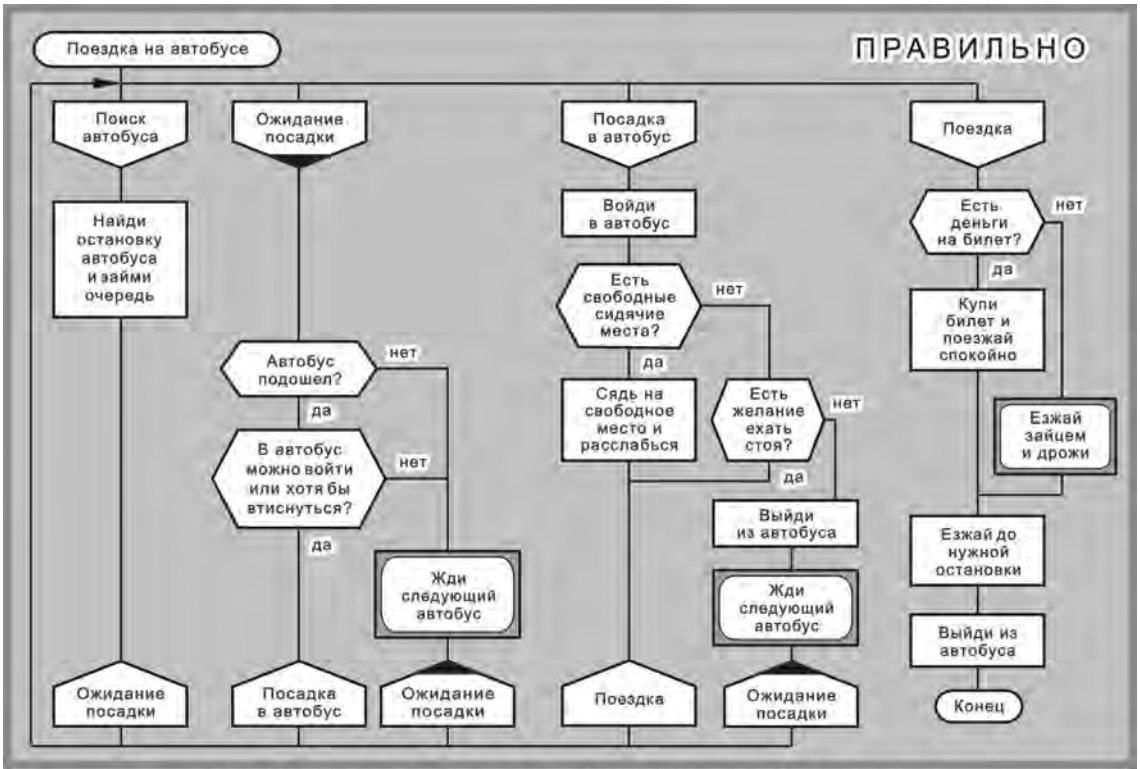


Рис. 22. Алгоритм-силуэт «Поездка на автобусе». Сравни с рис. 23. В чем отличие?

ГЛАВНЫЙ МАРШРУТ СИЛУЭТА

Выше мы узнали, как упорядочить маршруты примитива. Теперь настала очередь силуэта.

Шампур ветки — это вертикаль, соединяющая икону «имя ветки» с иконой «адрес», а если у ветки несколько выходов — с левым из них.

Каждая ветка силуэта имеет свой шампур. Например, на рис. 22 четыре ветки. Следовательно, этот силуэт имеет четыре шампура ветки.

Для ветки сохраняют силу оба «царских» правила:

- главный маршрут ветки должен идти по шампуру ветки;
- побочные маршруты ветки следует упорядочить слева направо по какому-либо критерию.

Предположим, в качестве критерия выбран принцип «Чем правее, тем хуже». В этом случае для каждой ветки силуэта действует

> *Правило.* **Чем правее (чем дальше от шампура данной ветки) расположена очередная вертикаль, тем менее успешные действия она выполняет.**

Например, на рис. 22 ветка «Посадка в автобус» имеет три вертикали.

Левая вертикаль (главный маршрут) описывает наибольший успех, так как вы будете ехать в автобусе сидя.

Правая вертикаль означает наименьший успех, поскольку вы вышли из автобуса и поездка откладывается.

Средняя вертикаль (расположенная выше иконы «Есть желание ехать стоя?») занимает промежуточное положение. Потому что — в зависимости от ответа — может иметь место либо частичный успех (вы будете ехать, но не сидя, а стоя), либо неудача, поскольку вы выходите из автобуса несолоно хлебавши.

Главный маршрут силуэта — последовательное соединение главных маршрутов поочередно работающих веток.

Таким образом, ДРАКОН позволяет читателю моментально увидеть главный маршрут любого, сколь угодно сложного и разветвленного алгоритма. Кроме того, смещение всех побочных маршрутов относительно «царского» оказывается не случайным, а осмысленным и предсказуемым, то есть легким для восприятия.

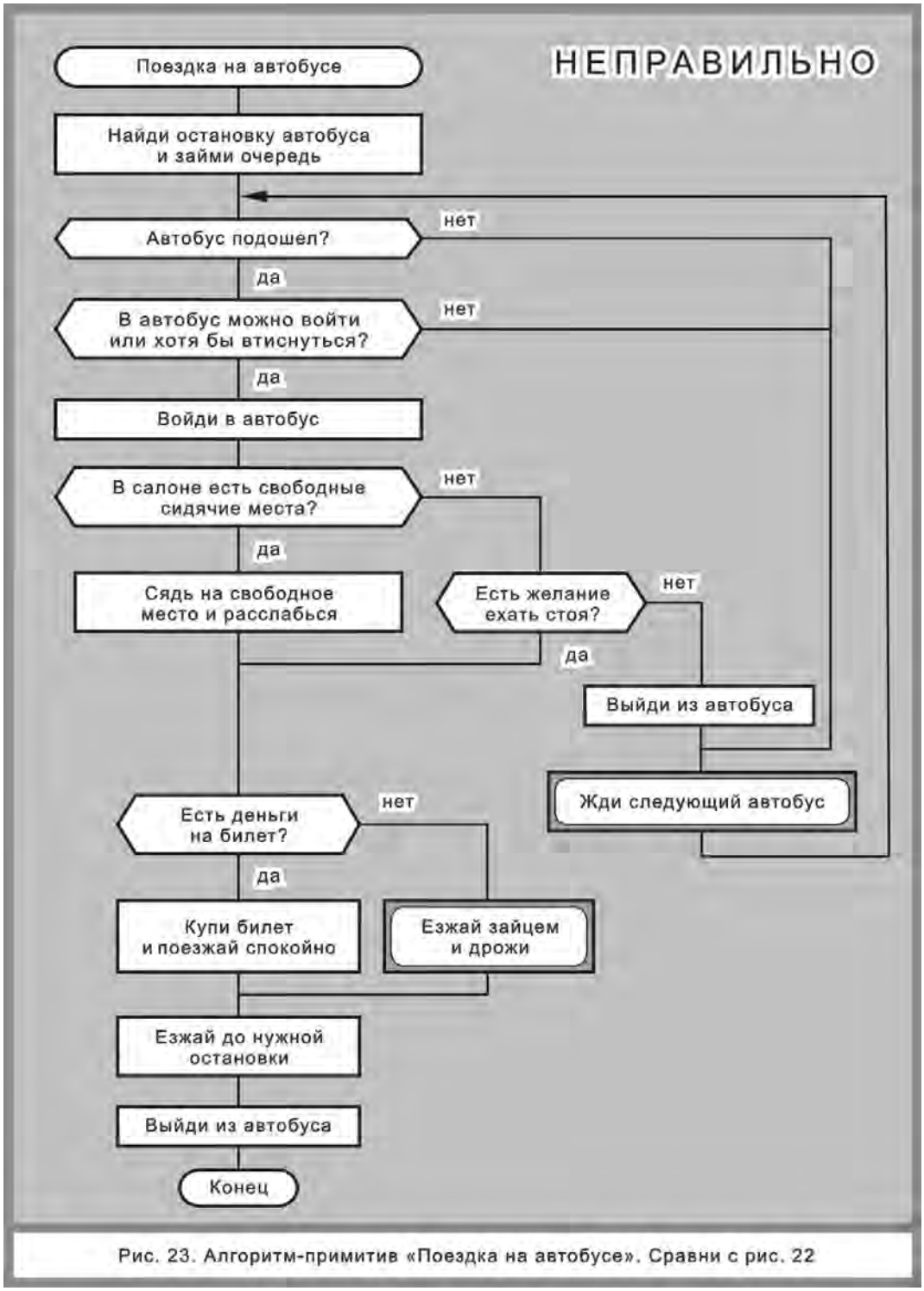


Рис. 23. Алгоритм-примитив «Поездка на автобусе». Сравни с рис. 22

ПЕРЕСЕЧЕНИЯ ЛИНИЙ? — БОЖЕ УПАСИ!

Некоторые специалисты, склонные к резким выражениям, называют традиционные блок-схемы алгоритмов [1] «помоечными блок-схемами». Потому что в этих схемах царит беспорядок. Запутанная паутина соедини-

тельных линий не помогает, а наоборот, затрудняет работу читателя, который пытается понять суть алгоритма.

ДРАКОН выгодно отличается тем, что его графический узор имеет строгое математическое и когнитивно-эргономическое обоснование и подчиняется жестким и тщательно продуманным правилам. Среди них особое место занимает

> *Правило.* **Пересечения и обрывы соединительных линий запрещены.**

При вычерчивании обычных блок-схем допускаются два типа пересечения линий:

- явное, изображенное крестом линий,
- замаскированное, выполняемое с помощью так называемых соединителей.

Как известно, соединитель «используется для обрыва линии и продолжения ее в другом месте... для избежания излишних пересечений» [1].

Следует подчеркнуть, что эффективность соединителей для борьбы с пересечениями близка к нулю. Хотя соединители действительно уменьшают число пересечений, но они не улучшают понимаемость блок-схем. То есть не позволяют решить наиболее важную, приоритетную задачу.

В языке ДРАКОН все перечисленные ухищрения (пересечения, обрывы, соединители) по эргономическим соображениям считаются вредными и категорически запрещены. Потому что они засоряют поле чертежа ненужными деталями, создают визуальные помехи для глаз и отвлекают внимание от главного.

Поскольку запрет пересечений является серьезным топологическим ограничением, возникает вопрос: можно ли произвольный алгоритм изобразить в виде дракон-схемы?

Теорема 1. Любая структурная программа может быть изображена на языке ДРАКОН двумя способами: в виде примитива и в виде силуэта.

Теорема 2. Произвольная (неструктурная) программа в ряде случаев не может быть изображена в виде примитива. Однако с помощью эквивалентных преобразований, допускающих введение дополнительных переменных (идентификаторов ветки), она всегда может быть изображена в виде силуэта.

Чтобы прояснить вопрос, обратимся к примерам. На рис. 24 (слева) приведена запрещенная дракон-схема — примитив, в котором имеется неустранимое (без введения дополнительных переменных) пересечение.

На рис. 24 (справа) изображен силуэт, который, как нетрудно убедиться, эквивалентен упомянутому примитиву и вместе с тем не содержит ни одного пересечения. Таким образом, пример на рис. 24 подтверждает справедливость теоремы 2[1].

Подведем итоги. Язык ДРАКОН обладает важным достоинством: он позволяет изобразить *любой* алгоритм, полностью отказавшись от таких эргономически неудачных приемов, как пересечения, обрывы, соединители. Отсутствие «паразитных элементов» создает дополнительные удобства для читателя, делает дракон-схему прозрачной, легкой для понимания.

---

[1] Доказательство теорем 1 и 2 предоставляем читателю. *Указание:* необходимо опереться на теорему о структурировании и метод Ашкрофта — Манны [2, 3].

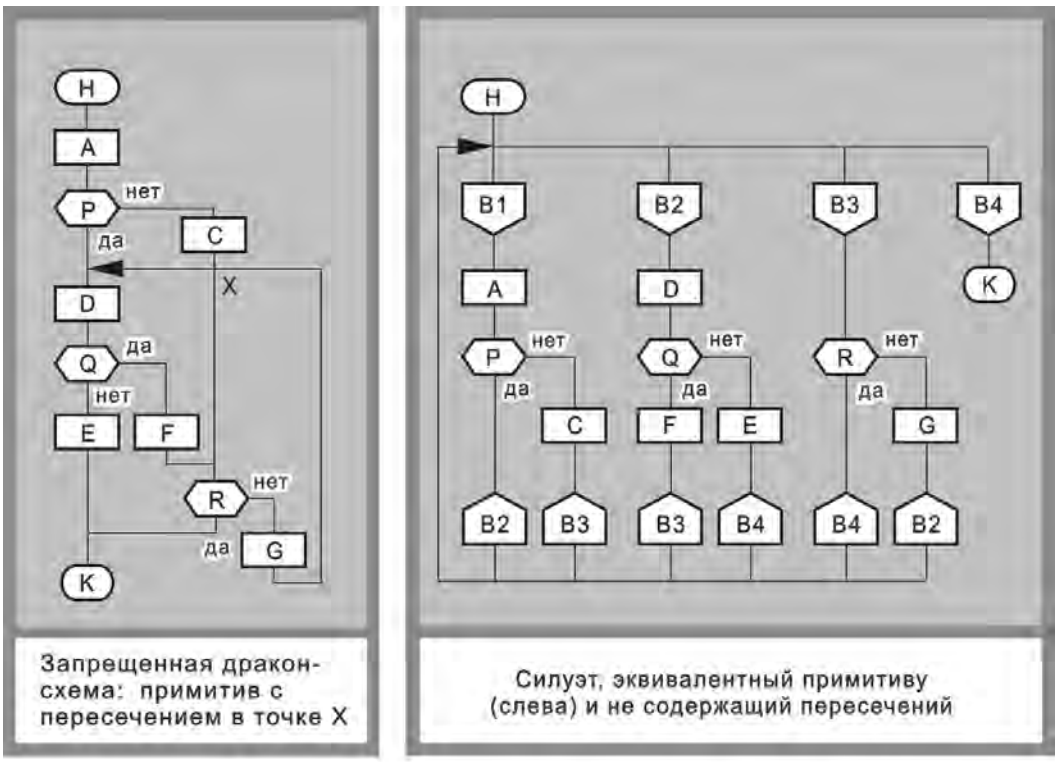


Рис. 24. **Замена примитива на эквивалентный ему силуэт позволяет устранить любые пересечения линий**

КАК ОПИСАТЬ СИЛУЭТ С ПОМОЩЬЮ ТЕКСТОВОГО ЯЗЫКА?

Из рис. 25 видно, что для описания веток в текстовый язык пришлось внести ряд изменений. В частности, появились два новых текстовых оператора, отсутствующие в традиционных языках:

ВЕТКА < идентификатор ветки >
АДРЕС < идентификатор ветки >

Оператор текстового языка ВЕТКА объявляет название ветки (записываемое на графическом языке внутри иконы «имя ветки»). Оператор АДРЕС безусловно передает управление на текстовый оператор ВЕТКА, имя которой совпадает с записью в операторе АДРЕС.

Сравнивая два языка: графический и текстовый, можно заметить, что соответствующие алгоритмы (рис. 22 и 25) эквивалентны[1]. Однако графический язык (язык дракон-схем), несомненно, более нагляден и доходчив.

Второе преимущество состоит в том, что графика позволяет полностью исключить избыточные (паразитные) элементы, каковыми в текстовом языке оказываются почти все ключевые слова: АЛГОРИТМ, ВЕТКА, АДРЕС, КОНЕЦ ВЕТКИ, ЕСЛИ, ТО, ИНАЧЕ, КОНЕЦ ЕСЛИ, ЦИКЛ ЖДАТЬ, КОНЕЦ ЦИКЛА, КОММЕНТАРИЙ, ПЕРЕХОД НА, а также метки.

[1] Два алгоритма называются *эквивалентными*, если они дают одинаковые результаты для одних и тех же исходных данных.

```
АЛГОРИТМ Поездка на автобусе
ВЕТКА Поиск автобуса
        ВЫПОЛНИТЬ Найди остановку автобуса и займи очередь
        АДРЕС Ожидание посадки
КОНЕЦ ВЕТКИ
ВЕТКА Ожидание посадки
        ЕСЛИ Автобус подошел ? = ДА
                ЦИКЛ ЖДАТЬ
                        КОММЕНТАРИЙ Происходит посадка пассажиров
                        ЕСЛИ Твоя очередь подошла ? = НЕТ
                                КОММЕНТАРИЙ Жди, пока подойдет очередь
                КОНЕЦ ЦИКЛА
                ЕСЛИ В автобус можно войти ? = ДА
                        АДРЕС Посадка в автобус
        М1:     ИНАЧЕ КОММЕНТАРИЙ Жди прихода следующего автобуса
                        АДРЕС Ожидание посадки
                КОНЕЦ ЕСЛИ
        ИНАЧЕ ПЕРЕХОД НА М1
        КОНЕЦ ЕСЛИ
КОНЕЦ ВЕТКИ
ВЕТКА Посадка в автобус
        ВЫПОЛНИТЬ Войди в автобус

        и так далее
```

Рис. 25. Текстовая запись алгоритма, соответствующая дракон-схеме на рис. 22 (описаны только две ветки из четырех)

ВИЗУАЛЬНЫЙ И ТЕКСТОВЫЙ СИНТАКСИС ДРАКОНА

ДРАКОН — графический язык, в котором используются два типа элементов:

- графические фигуры (*графоэлементы*),
- текстовые надписи, расположенные внутри или снаружи графических фигур (*текстоэлементы*).

Следовательно, синтаксис ДРАКОНА распадается на две части.

Графический синтаксис охватывает алфавит графоэлементов, правила их размещения в поле чертежа и правила связи графоэлементов с помощью соединительных линий.

Текстовый синтаксис задает алфавит символов, правила их комбинирования и привязку к графоэлементам. (Привязка необходима потому, что внутри разных графических фигур используются разные типы выражений).

Оператором языка ДРАКОН является графоэлемент или комбинация графоэлементов, взятые вместе с текстовыми надписями.

Одновременное использование графики и текста говорит о том, что ДРАКОН адресуется не только к словесно-логическому мышлению автора и читателя программы, но сверх того активизирует интуитивное, образное, правополушарное мышление, стимулируя его не написанной, а именно нарисованной программой, то есть программой-картинкой.

СЕМЕЙСТВО ДРАКОН-ЯЗЫКОВ

ДРАКОН — не один язык, а целое семейство, которое может включать практически неограниченное число языков.

Все языки семейства имеют одинаковый графический синтаксис (что зрительно делает языки почти близнецами). Каждый язык семейства отличается тем, что имеет свой собственный текстовый синтаксис.

ДРАКОН-1 — графический псевдоязык, графический аналог обычного текстового псевдокода.

В этом языке в качестве текстоэлементов используется естественный язык. Почти все примеры в этой книге даны на языке ДРАКОН-1. Именно этот язык предлагается в качестве универсального средства взаимопонимания. ДРАКОН-1 служит для описания структуры деятельности, создания технологий, алгоритмов и проектов программ, используется в методе декомпозиции и пошаговой детализации, а также при формализации профессиональных знаний.

ДРАКОН-2 — графический алгоритмический язык для разработки алгоритмов и программ реального времени.

Упрощенная версия этого языка используется в качестве *CASE*-технологии «ГРАФИТ-ФЛОКС» в Научно-производственном центре автоматики и приборостроения им. академика Н.А. Пилюгина. Она применяется для разработки программного обеспечения систем управления ракет-носителей и разгонных блоков космических аппаратов.

После доработки инструментальных программ язык может быть использован при разработке систем управления реального времени для атомных электростанций, нефтехимических и металлургических заводов, биотехнологических производств и т. д.

Кроме того, семейство включает гибридные графические языки программирования: ДРАКОН-БЕЙСИК, ДРАКОН-ПАСКАЛЬ, ДРАКОН-СИ++ и т. д.

Чтобы получить гибридный язык, например, ДРАКОН-БЕЙСИК, необходимо взять графический синтаксис ДРАКОНА и присоединить к нему по определенным правилам текстовый и визуальный синтаксис языка БЕЙСИК.

Строгое разграничение графического и текстового синтаксиса позволяет в максимальной степени расширить сферу применения языка, обеспечивая его гибкость и универсальность.

При этом *единообразие* правил графического синтаксиса семейства ДРАКОН-языков обеспечивает их концептуальное единство.

А *разнообразие* текстовых правил (то есть возможность выбора любого текстового синтаксиса) определяет гибкость языка и легкую настройку на различные предметные области.

В настоящей книге основное внимание уделяется графическому псевдоязыку ДРАКОН-1 (использующему естественный человеческий язык). Что касается остальных языков ДРАКОН-семейства, даются лишь краткие пояснения.

ВЫВОДЫ

Приведем сводку эргономических правил, позволяющих улучшить когнитивное качество дракон-схем и сделать алгоритмы, программы и технологии более понятными.

1. Сложные алгоритмы следует рисовать в виде системы вложенных друг в друга силуэтов. При этом примитивы используются крайне редко, только как исключение.
2. В иконе «заголовок» запрещается писать слово «начало». Вместо этого следует указать понятное и точное название алгоритма.
3. Разбейте сложный алгоритм на части, каждую часть изобразите в виде ветки. Дайте частям доходчивые и четкие названия и запишите их в иконах «имя ветки».
4. Вход в ветку возможен только через ее начало.
5. В иконе «адрес» разрешается писать имя одной из веток, другие надписи запрещены.
6. Ветки следует располагать в пространстве согласно правилу «Чем правее, тем позже». Наличие веточного цикла модифицирует это правило.
7. Примитив обязательно имеет шампур. Это значит, что у примитива иконы «заголовок» и «конец» всегда лежат на одной вертикали, которая и называется «шампур».
8. Каждая ветка обязательно имеет шампур. У правой ветки шампур — это вертикаль, соединяющая иконы «имя ветки» и «конец». У остальных веток шампуром служит вертикальная линия, соединяющая иконы «имя ветки» и «адрес». А если адресов несколько — с левым из них.
9. Алгоритм всегда имеет главный маршрут, который должен идти по шампуру.
10. Побочные маршруты должны быть упорядочены слева направо согласно одному из выбранных критериев, например, «Чем правее, тем хуже».
11. В иконе «конец» следует писать слово «конец».
12. Соединительные линии могут идти либо горизонтально, либо вертикально. Наклонные линии не допускаются.
13. Пересечения линий запрещены.
14. Обрывы линий запрещены.
15. Использование соединителей запрещено.

Глава 2

ЭРГОНОМИЧНЫЕ АЛГОРИТМЫ

На ошибках мы горим! —
Мне сказал Алеха.
Непонятный алгоритм —
Это очень плохо.

Мы ошибки победим!
Надо сделать вот чего —
Надо сделать алгоритм
Ясным и доходчивым!

Юрий Примашев

ПРОВЕРКА АЛГОРИТМОВ ЗА СТОЛОМ

При решении сложных задач, таких как предсказание погоды, управление войсками, управление большой электростанцией или нефтехимическим заводом, приходится создавать крупномасштабные алгоритмы, насчитывающие сотни тысяч и даже миллионы команд.

Практика показывает: чем крупнее алгоритмы, тем больше в них ошибок. Тем сложнее их найти.

Исправлять ошибки в огромных и сложных алгоритмах — трудное и дорогостоящее занятие, причем цена ошибки тем выше, чем позже она обнаружена.

Наименьший ущерб приносят ошибки, которые удается обнаружить в самом начале работы, до генерации кода и исполнения программы на компьютере — в ходе мозговой (зрительной) проверки программы за столом.

Мы называем проверку *мозговой*, если основную работу по поиску ошибок выполняет человеческий мозг, а компьютер играет вспомогательную роль. При такой проверке человек тщательно изучает технические задания, алгоритмы и программы, представленные на бумаге или экране, стараясь обнаружить как можно больше ошибок и слабых мест. Однако сегодня такая проверка является не только дорогостоящей, но и крайне неэффективной.

Серьезный недостаток мировой практики программирования

Низкая эффективность проверки за столом является важным недостатком существующих методов разработки программного обеспечения. Причина в том, что технические задания, тексты алгоритмов, исходные коды программ и другие документы начального этапа разработки не приспособлены для решения этой задачи. Устранение данного недостатка является актуальной задачей.

КАК ПОВЫСИТЬ ЭФФЕКТИВНОСТЬ ПРОВЕРКИ ЗА СТОЛОМ?

Мозговую проверку, как и любую другую деятельность, нужно грамотно проектировать. Критерием ее эффективности служит выявление максимального числа ошибок за минимальное время. Или, что одно и то же, минимизация интеллектуальных усилий мозга, затрачиваемых в среднем на выявление одной ошибки.

Нейронная конструкция мозга такова, что он может эффективно проводить проверку за столом отнюдь не при любых условиях. А только в том случае, если проверяемые документы обладают высоким когнитивным качеством.

Чтобы минимизировать удельные интеллектуальные усилия, затрачиваемые на обнаружение ошибок, необходимо, чтобы когнитивно-значимые характеристики документов были хорошо согласованы с конструктивными характеристиками мозга. Это значит, что технические задания, алгоритмы и другие документы должны быть специально приспособлены для быстрой и надежной проверки, для легкого и вместе с тем глубокого понимания.

НУЖНА НОВАЯ ТЕОРИЯ

Введем понятие «критерий сверхвысокого понимания». Считается, что алгоритмический язык удовлетворяет этому критерию, если написанные на нем алгоритмы обладают наивысшим когнитивно-эргономическим качеством.

Можно сказать и по-другому. Критерий сверхвысокого понимания требует, чтобы форма записи алгоритмов была максимально удобной. И позволяла человеку читать и анализировать любой алгоритм с максимальной легкостью и глубиной понимания.

Чтобы создать подобную легкость, нужны новые, скажем прямо, непривычные для математиков, но понятные «для народа» правила записи алгоритмов. Более того, нужна новая теория — *теория эргономичных алгоритмов.*

ЧТО ТАКОЕ ЭРГОНОМИЧНЫЙ АЛГОРИТМ?

В этой книге выражения «дружелюбный алгоритм» и «эргономичный алгоритм» употребляются как синонимы.

Однако, здесь есть важные оттенки. Слово «дружелюбный» — всего лишь метафора. А выражение «эргономичный алгоритм» мы склонны рассматривать как новое фундаментальное научное понятие. Разумеется, это утверждение надо тщательно обосновать. Данная книга как раз и является таким обоснованием.

Эргономичный алгоритм — это алгоритм, удовлетворяющий критерию сверхвысокого понимания. То есть алгоритм, специально сконструированный таким образом, чтобы обеспечить выявление ошибок за столом без лишней траты умственных сил.

Преимущество эргономичных алгоритмов в том, что они намного понятнее, яснее, нагляднее и доходчивее, чем обычные. Если алгоритм непонятный, в нем трудно или даже невозможно заметить затаившуюся ошибку. И наоборот, чем понятнее алгоритм, тем легче найти дефект.

Поэтому более понятный, эргономичный алгоритм намного лучше обычного. Лучше в том смысле, что он облегчает выявление ошибок, а это очень важно. Ведь чем больше ошибок удастся обнаружить при про-

верке за столом, тем больше вероятность, что вновь созданный алгоритм окажется правильным, безошибочным, надежным.

Кроме того, эргономичные алгоритмы удобнее для изучения, их проще объяснить другому человеку.

ЭРГОНОМИЧНЫЙ АЛГОРИТМИЧЕСКИЙ ЯЗЫК

Эргономичный алгоритмический язык — это язык, позволяющий создавать эргономичные алгоритмы.

Особенность такого языка состоит в следующем. Эргономичные алгоритмы делают проблему более ясной и прозрачной. Поэтому использование эргономичных алгоритмических языков облегчает и ускоряет творческий процесс создания новых алгоритмов.

Иными словами, эргономичный язык повышает производительность труда при разработке и отладке алгоритмов и программ.

НУЖЕН НОВЫЙ ПОНЯТИЙНЫЙ АППАРАТ

В предыдущей главе мы рассмотрели несколько способов, позволяющих улучшить эргономические характеристики алгоритмов.

В этой главе мы продолжим тему. И попытаемся ответить на вопрос: можно ли повысить эргономичность алгоритмов, используя формальный метод *эквивалентных преобразований алгоритмов?*

Для этого понадобится особый понятийный аппарат, заметно отличающийся от того, которым обычно пользуются программисты. Дело в том, что традиционные понятия плохо приспособлены для решения проблемы понимания. И совершенно не учитывают специфику зрительных образов.

ИКОНА «ВОПРОС»

Да-нетный вопрос — это вопрос, на который можно ответить либо «да», либо «нет». Все другие ответы запрещены.

Вот примеры да-нетных вопросов: утюг сломался? Тетя приехала? Вася купил хлеб? Преступника арестовали? Эта лужа больше, чем та? Температура выше нуля? Прошла команда на включение двигателя?

На рис. 4 (позиция И4) изображена икона «вопрос». Она называется так, потому что внутри нее пишут да-нетный вопрос. Другие надписи не допускаются.

Икона «вопрос» имеет один вход сверху и два выхода: вниз и вправо. Выход влево запрещен и никогда не используется. Возле выходов обязательно пишут слова «да» и «нет».

ЧТО ТАКОЕ РАЗВИЛКА?

На рис. 5 (позиция 2) показана макроикона «развилка».

Развилка — часть дракон-схемы, внутри которой маршрут сначала раздваивается, а затем соединяется в точке слияния. Развилка имеет один вход сверху и один выход снизу Она представляет собой шампур-блок, который содержит:

- икону «вопрос»,
- левое плечо развилки,
- правое плечо развилки
- точку слияния (рис. 26).

Суть дела ясна из примера на рис. 27.

Левое плечо развилки есть путь от нижнего выхода иконы «вопрос» (точка *А*) до точки слияния *Д.* Оно содержит ответ «да», три иконы и соединительные линии.

Правое плечо развилки начинается у правого выхода иконы «вопрос» и заканчивается в точке слияния. На рис. 27 оно содержит ответ «нет» и линию *БВГД.*

Таким образом, плечо имеет в своем составе надпись «да» или «нет», соединительные линии, иконы и точку слияния. Одно из двух плеч может быть *пустым* (не содержать икон).

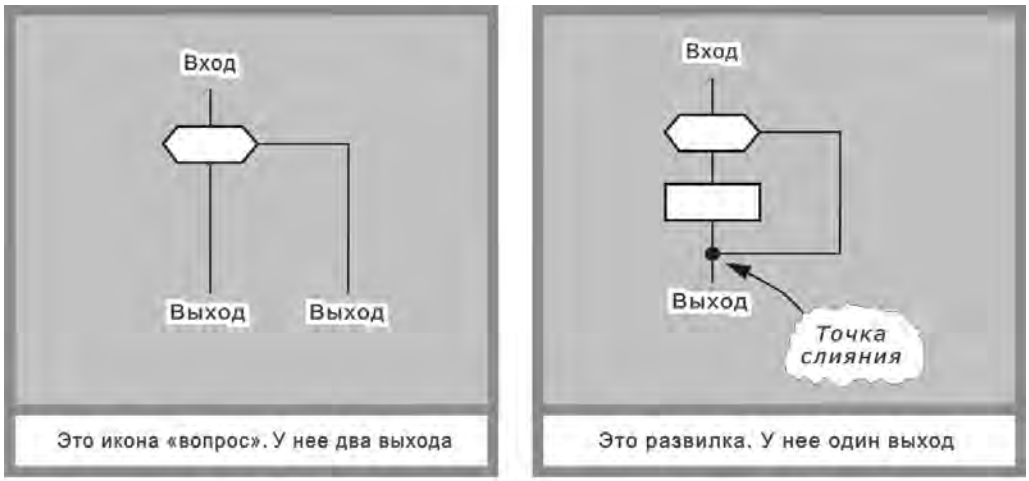


Рис. 26. Чем различаются развилка и икона «вопрос»?

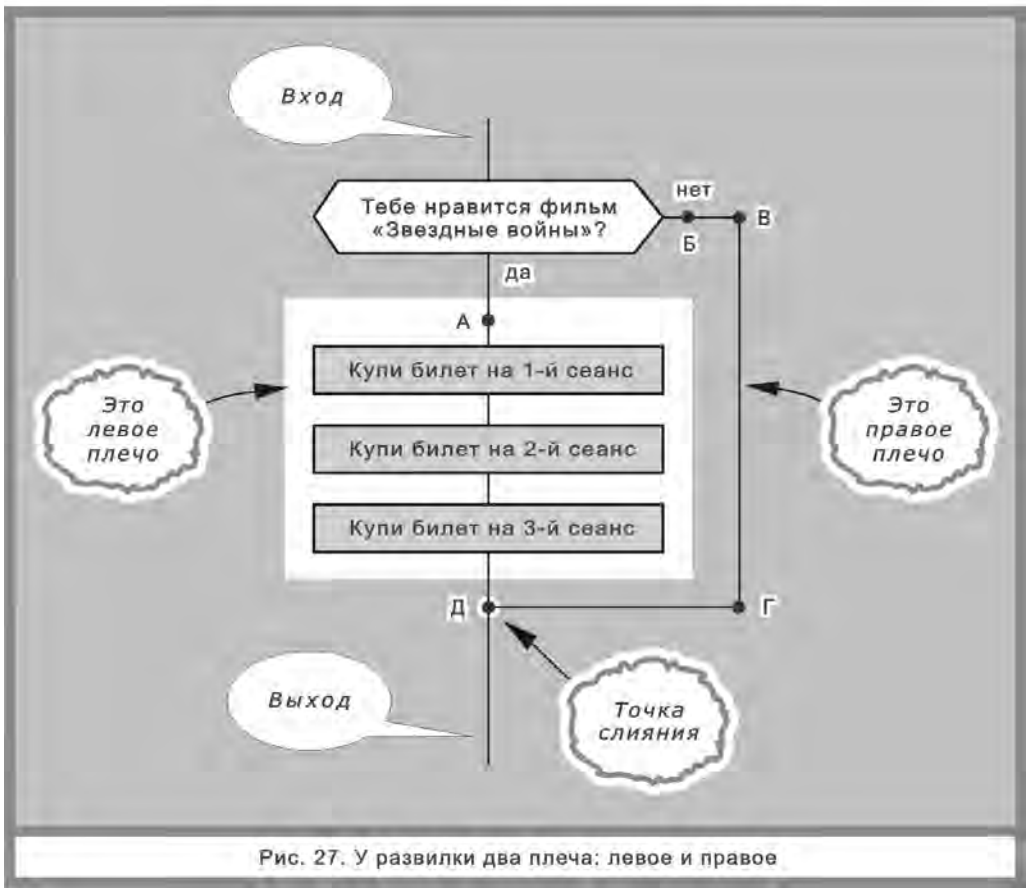


Рис. 27. У развилки два плеча: левое и правое

ПРОСТЫЕ И СЛОЖНЫЕ РАЗВИЛКИ

Развилки бывают простые и сложные.

Простая развилка содержит только одну икону-вопрос. Примеры простых развилок показаны на рис. 28.

Развилка называется *сложной*, если в ее плечах имеется по крайней мере одна простая или сложная развилка.

На рис. 19 показаны три сложные развилки. Например, развилка «Потерял 500 рублей?» сложная, так как ее правое плечо содержит простую развилку «Потерял 1000 рублей?». Развилки «Потерял деньги?» и «Потерял сто рублей?» тоже сложные.

Другие примеры сложных развилок показаны на рис. 29.

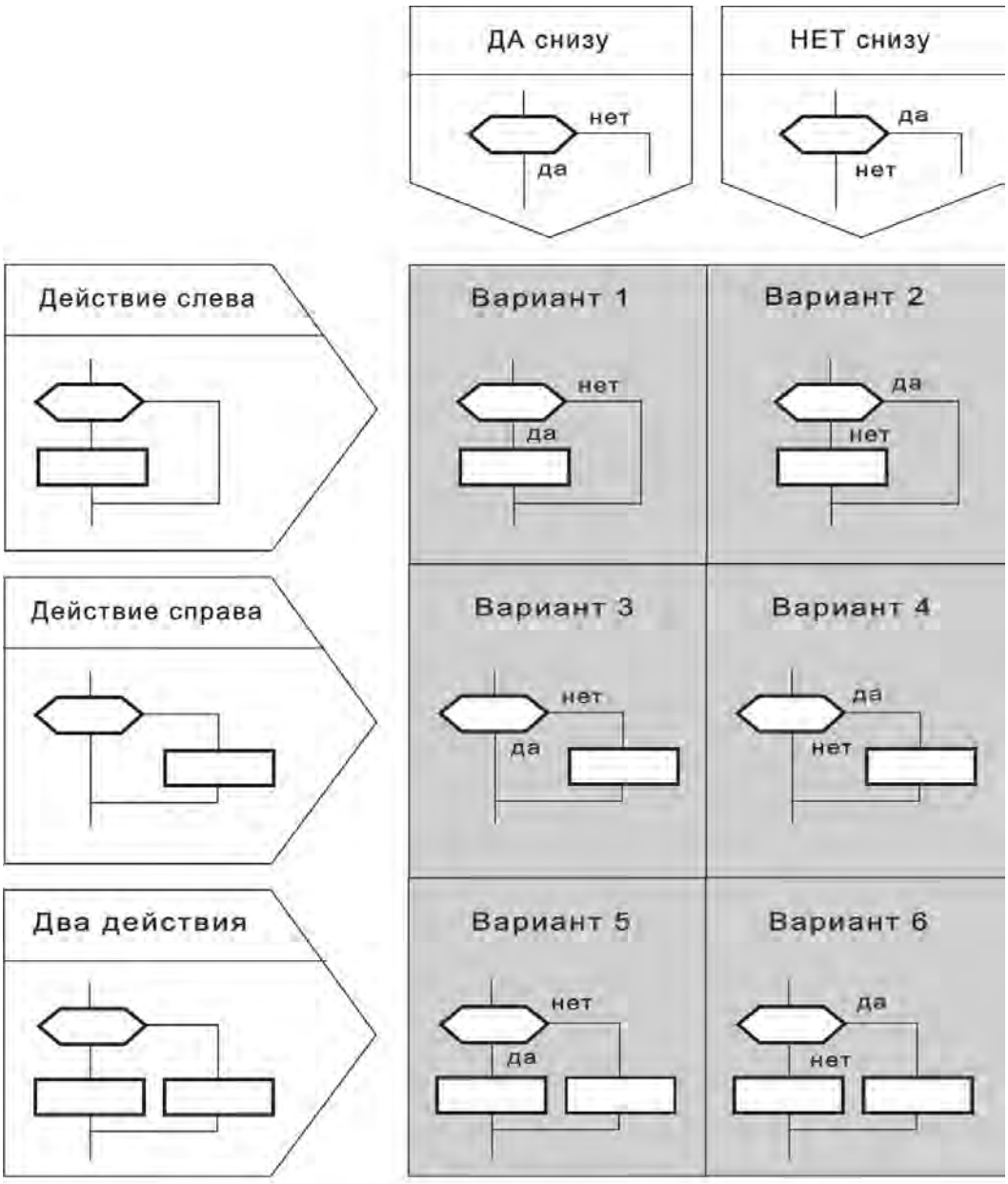


Рис. 28. Шесть вариантов изображения развилки

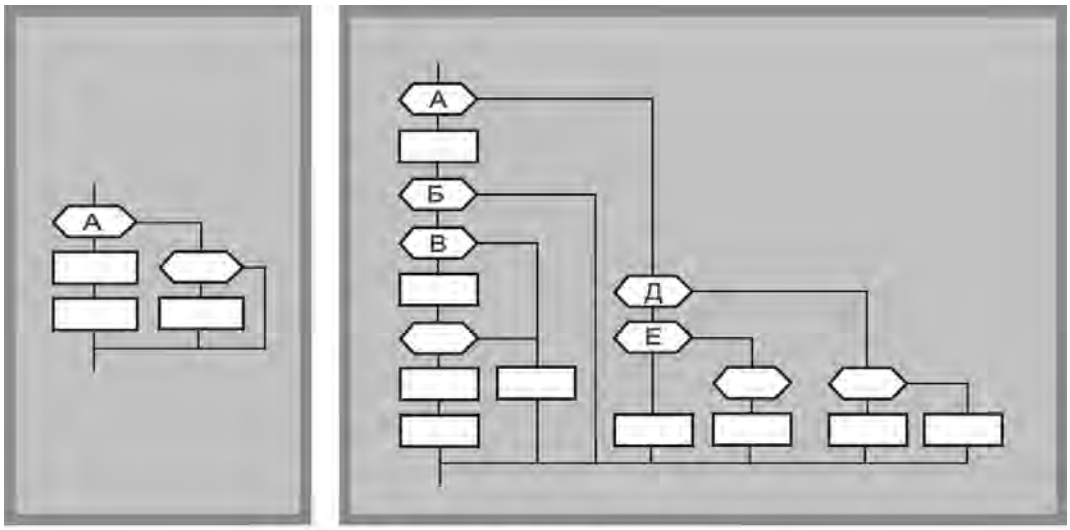


Рис. 29. Примеры сложных развилок

МАРШРУТЫ И ФОРМУЛЫ МАРШРУТОВ

На рис. 30 (слева) представлена дракон-схема «Охота на мамонта». Заменим текст внутри икон буквами. Вместо «Охота на мамонта» запишем букву А. Вместо «Поймай мамонта» — букву Б и т. д. В результате получим буквенную (литеральную) дракон-схему на рис. 30 справа.

Буквенные схемы удобно использовать для описания маршрутов.

Маршрут — это графический путь от начала до конца алгоритма, проходящий через иконы и соединительные линии. Маршрут можно описать с помощью *формулы*, которая представляет собой последовательность букв, обозначающих иконы. Все иконы, включая одинаковые, обозначаются разными буквами.

Линейный (неразветвленный) алгоритм имеет только один маршрут и одну формулу. Например, схема на рис. 30 (справа) описывается формулой

АБВГД

Разветвленный алгоритм имеет несколько (два или более) маршрутов, причем у каждого маршрута своя, отличная от других формула (рис. 31, 32).

В формулах разветвленных алгоритмов наряду с буквами, обозначающими иконы, используются слова «да» и «нет» (отделяемые пробелами).

Для графического алгоритма справедливо

> *Правило*. **Выполнить графический алгоритм — значит пройти путь от начала до конца алгоритма по одному из возможных маршрутов.**

Будем считать, что существует воображаемый «бегунок», который при работе алгоритма пробегает алгоритмическую дорожку (маршрут) от иконы «заголовок» до иконы «конец».

Рис. 30. Как преобразовать обычную дракон-схему в буквенную?

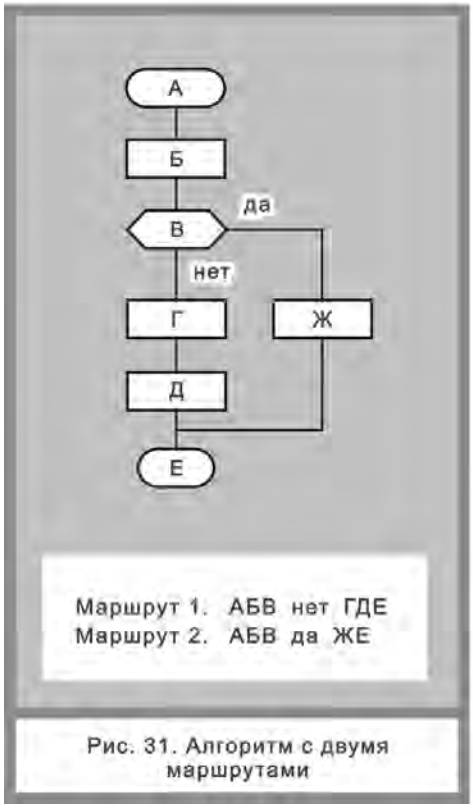


Рис. 31. Алгоритм с двумя маршрутами

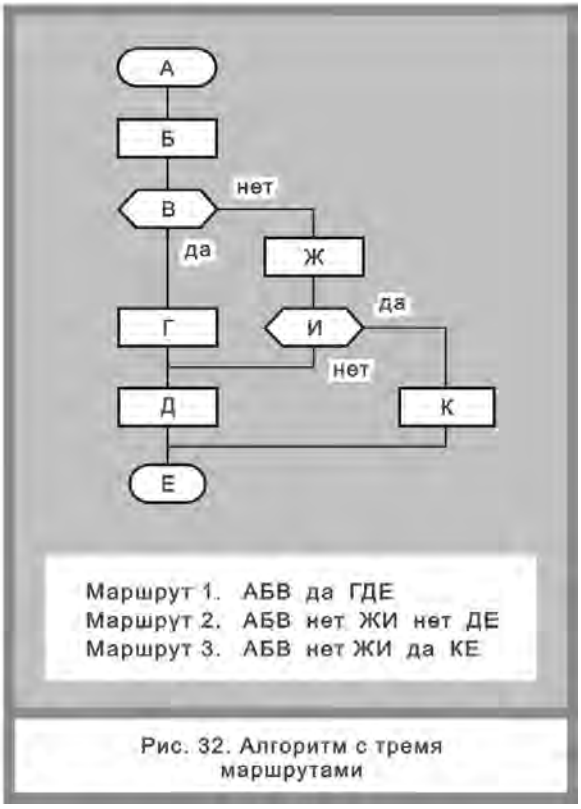


Рис. 32. Алгоритм с тремя маршрутами

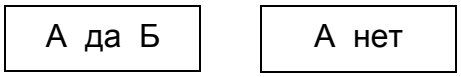
ЧТО ТАКОЕ РОКИРОВКА?

Рокировка — преобразование алгоритма, при котором левое и правое плечо развилки меняются местами. При этом слова «да» и «нет» также меняются местами. Простейшие примеры рокировки показаны на рис. 17, 18 и 33, 34.

Говорят, что два алгоритма имеют одинаковый набор маршрутов, если для каждого маршрута первого алгоритма можно найти парный маршрут второго алгоритма, причем для каждой пары маршрутов их формулы совпадают.

Если два алгоритма имеют одинаковый набор маршрутов, они эквивалентны.

Обратимся к рис. 34. Легко убедиться, что схемы на рис. 34 (слева) и 34 (справа) имеют одинаковый набор маршрутов:



Следовательно, указанные дракон-схемы эквивалентны.

Формальное преобразование алгоритма А1 в алгоритм А2 называется равносильным, если алгоритмы А1 и А2 эквивалентны. Отсюда вытекает

Следствие. Рокировка является равносильным преобразованием алгоритмов. При рокировке смысл алгоритма не меняется.

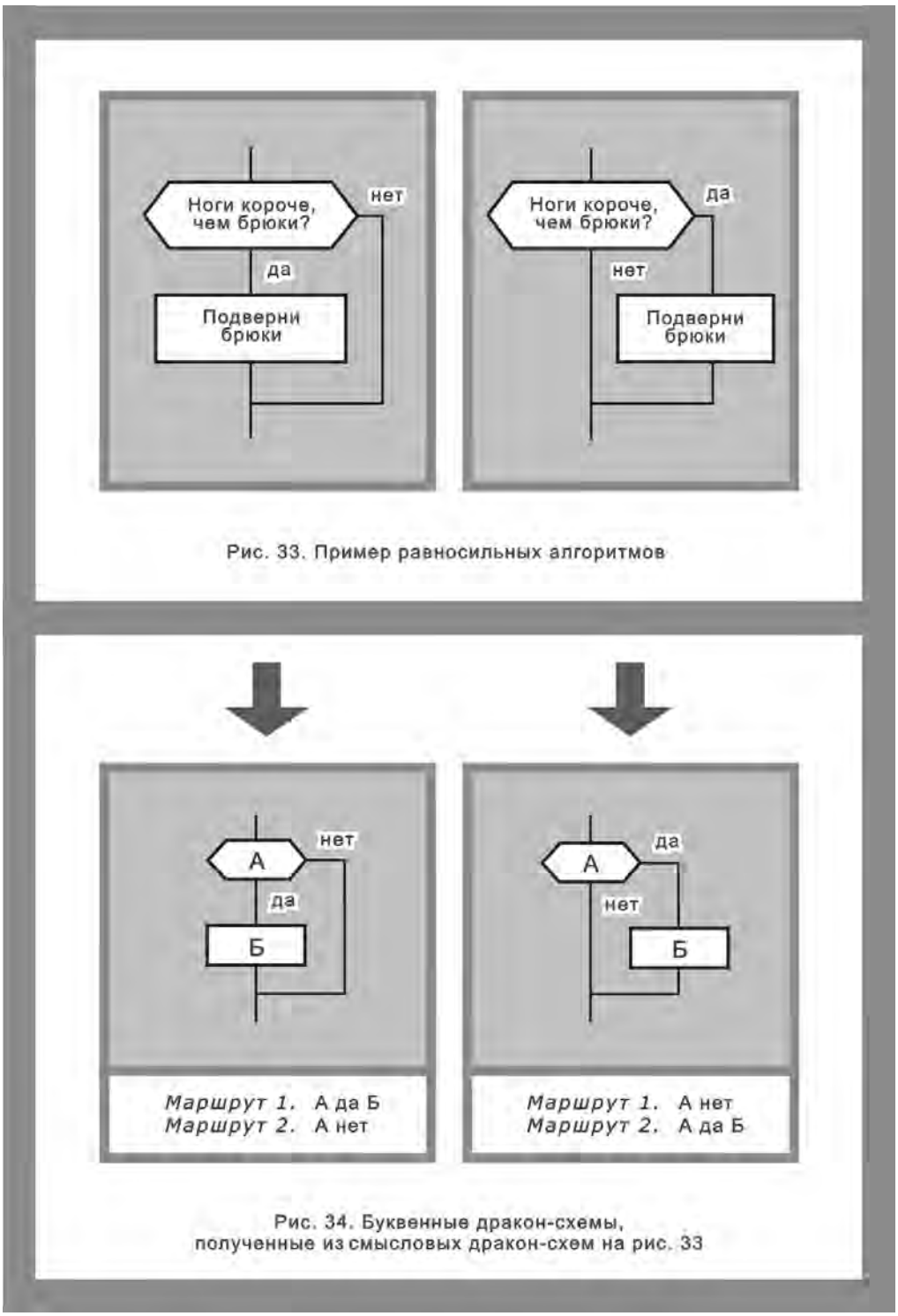


Рис. 33. Пример равносильных алгоритмов

Рис. 34. Буквенные дракон-схемы, полученные из смысловых дракон-схем на рис. 33

РОКИРОВКА И ЭРГОНОМИЧНОСТЬ: ОБСУЖДЕНИЕ ВОПРОСА

Полученный результат чрезвычайно важен. Ведь *рокировка позволяет улучшить наглядность и понятность алгоритмов.*

Попытаемся обосновать этот вывод для рис. 33, рассмотрев последовательно все промежуточные шаги рассуждений.

Шаг 1. Выдвигаем гипотезу: для сравнения двух маршрутов на рис. 33 можно использовать признак «лучше—хуже».

Шаг 2. Проверяем гипотезу с помощью следующих рассуждений. Если брюки впору — это хорошо, если их приходится подворачивать —

это плохо. Следовательно, использование в данном алгоритме признака «лучше—хуже» правомерно.

Шаг 3. Определяем главный маршрут, который по соглашению соответствует признаку «хорошо». Убеждаемся, что главный маршрут на рис. 33 (слева) идет через правое плечо развилки, что соответствует хорошей ситуации «брюки впору — их не надо подворачивать».

Шаг 4. Констатируем, что главный маршрут на рис. 33 (слева) не идет по шампуру. Делаем вывод: данный алгоритм является плохим, неэргономичным. Однако его можно исправить с помощью рокировки.

Шаг 5. Выполняем рокировку и получаем более эргономичный алгоритм на рис. 33 справа. На этом процедура заканчивается[1].

РОКИРОВКА МОЖЕТ УЛУЧШИТЬ ЭРГОНОМИЧНОСТЬ АЛГОРИТМОВ

На рис. 35 показана плохая дракон-схема. Согласно правилу главный маршрут (жирная линия) должен быть прямым, как стрела, и идти точно по шампуру. А он вместо этого превратился в ломаную-переломаную линию, которая делает невообразимые скачки и путает читателя. Чтобы исправить ошибку, нужно три раза сделать рокировку.

Первый раз делаем рокировку в развилке «В меню есть ваш любимый салат?». Это позволяет пустить главный маршрут по шампуру на верхнем участке. Однако внизу главный маршрут по-прежнему совершает неоправданные зигзаги.

Затем делаем рокировку в развилке «Борщ очень вкусный?». Тем самым улучшаем среднюю часть схемы.

Нам осталось выпрямить главный маршрут на нижнем участке. Для этого переставляем плечи у развилки «Жаркое как подошва?».

Таким образом, в результате трех рокировок неэргономичная схема на рис. 35 превратилась в хорошую (эргономичную) схему на рис. 36.

Подведем итоги. Выпрямляя главный маршрут, мы делаем алгоритм более наглядным, легким для понимания. Потому что главный маршрут — путеводная нить алгоритма, позволяющая быстрее уяснить суть дела.

А теперь — самое главное. Мы осуществили выпрямление главного маршрута не случайно, не по принципу «Что хочу, то и ворочу!», а на основании строгого математического закона — *закона рокировки.* Напомним суть закона: *рокировка — равносильное преобразование алгоритма.*

Наличие такого закона придает нашим *эргономическим* действиям (позволяющим выпрямить «кривой» главный маршрут) *математическую* строгость и точность.

Мы убедились, что рокировка алгоритма на рис. 35 позволила получить алгоритм на рис. 36, который имеет более высокие эргономические характеристики. Это означает, что операция «рокировка» в примере на рис. 35 и 36 действительно улучшает эргономичность алгоритма.

---

[1] Правило «Главный маршрут идет по шампуру» — это необходимое, но отнюдь не достаточное условие эргономичности алгоритма. Другое условие — эргономизация текста, то есть превращение запутанного и невразумительного текста в ясный и понятный.

Вопрос «Ноги короче, чем брюки?» звучит вычурно, противоестественно и сбивает с толку читателя. Вместо него следует написать: «Брюки слишком длинные?». В итоге получим действительно понятный и хороший алгоритм.

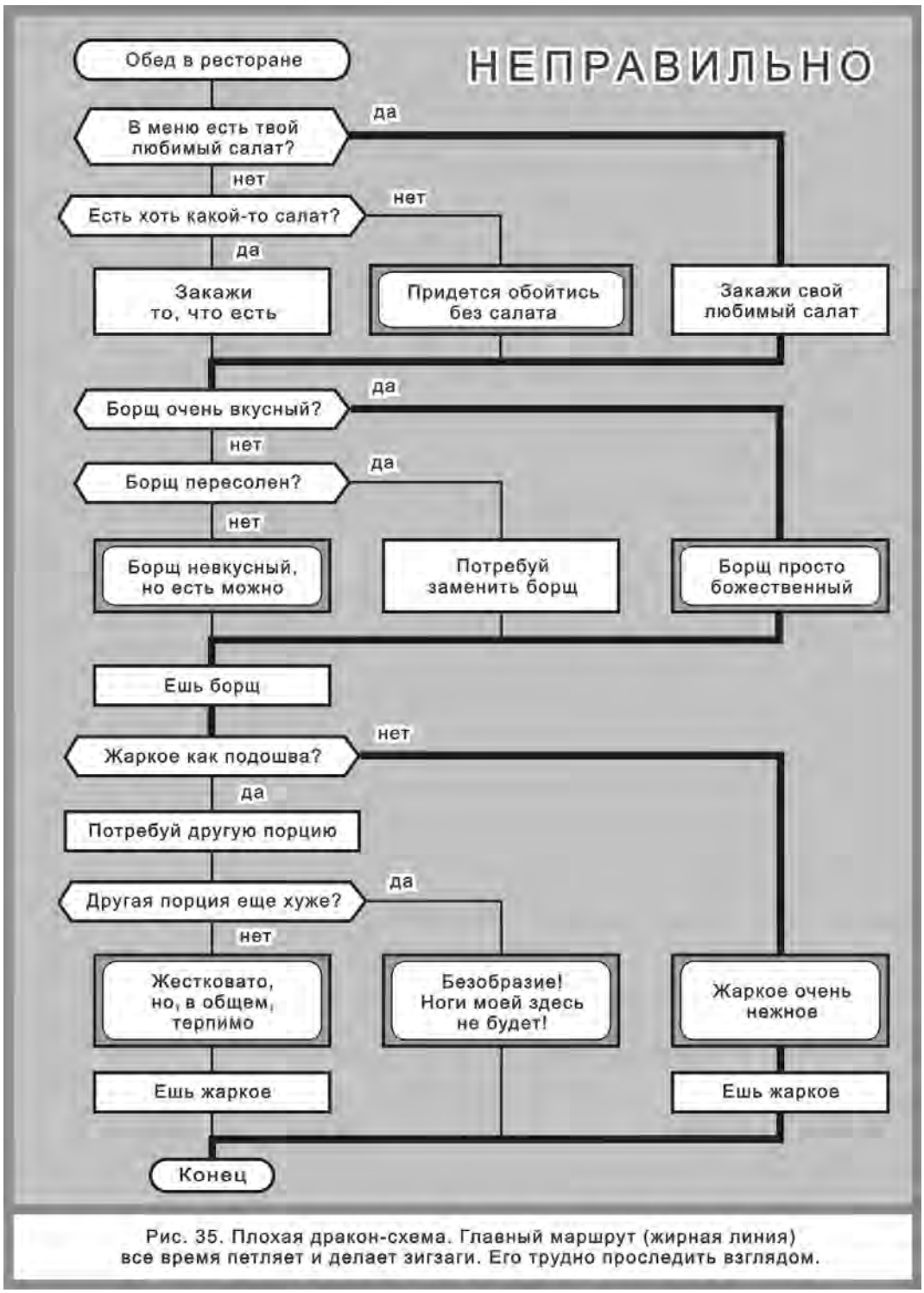


Рис. 35. Плохая дракон-схема. Главный маршрут (жирная линия) все время петляет и делает зигзаги. Его трудно проследить взглядом.

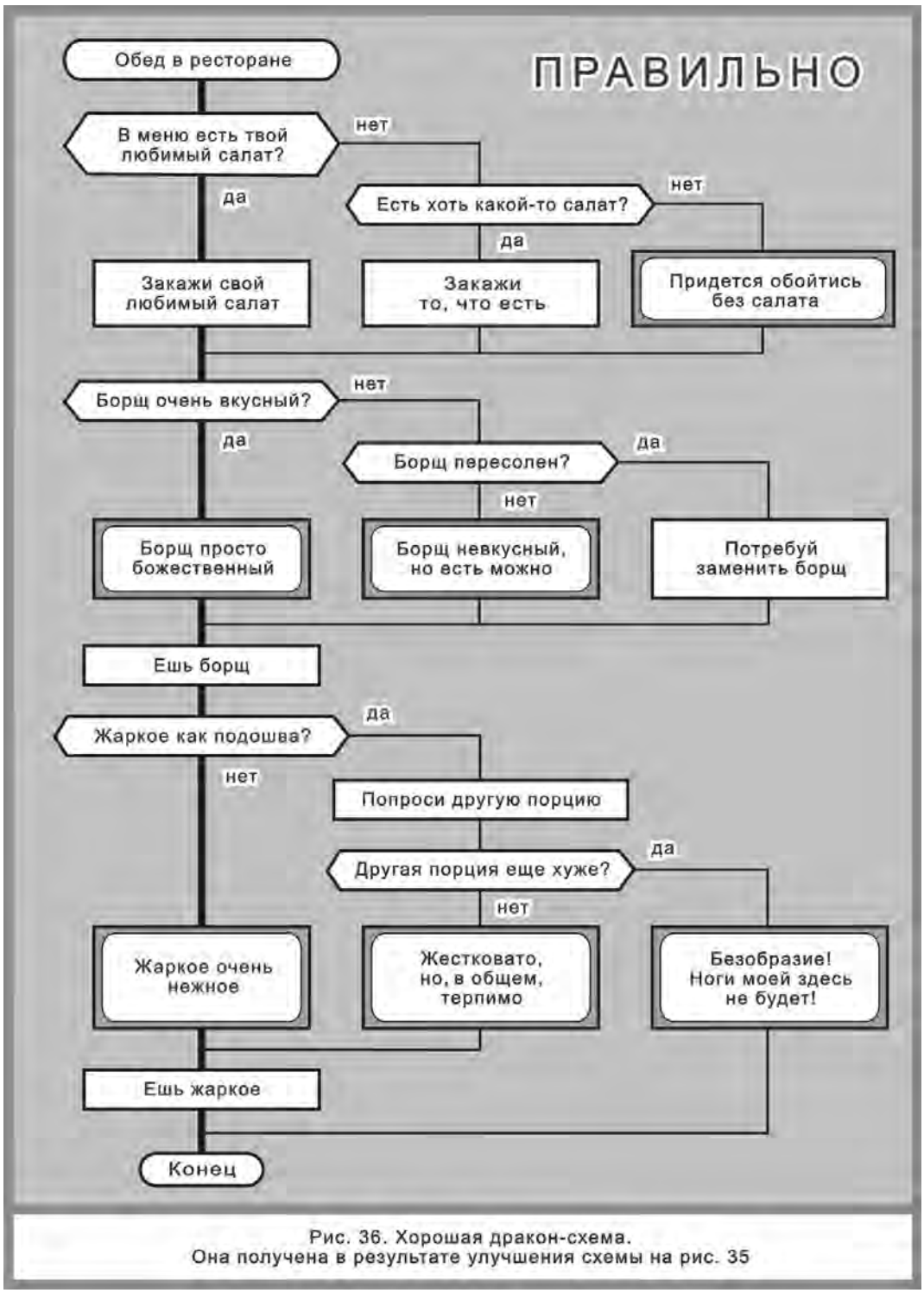


Рис. 36. Хорошая дракон-схема.
Она получена в результате улучшения схемы на рис. 35

ЕЩЕ ОДИН ПРИМЕР

На рис. 37—40 представлены четыре схемы, на которых описана история с чернильницей. На всех схемах изображен один и тот же алгоритм. Однако схемы выглядят по-разному.

Зададим вопрос, какие из них начерчены правильно, а какие нет?

Сначала нужно найти главный маршрут. В развилке «Чернильница упала?» главный маршрут идет через «нет». Потому что, когда вещи падают, это плохо. А когда не падают — хорошо. Значит, две схемы — на рис. 37 и 38 — нарисованы неверно.

В чем ошибка? Согласно правилу, главный маршрут должен идти по шампуру. А он вместо этого петлят по задворкам, как заяц.

Проверим правило побочных маршрутов. На рис. 37—40 их два. Один описывает ситуацию, когда чернильница упала, но уцелела. Во втором случае она разбилась. Первой ситуации выставим оценку «плохо», второй — «очень плохо».

Отсюда делаем вывод, что на рис. 39 маршруты не упорядочены. Значит, схема нарисована неверно. Почему? Потому что самый плохой маршрут (с оценкой «очень плохо») должен быть крайним справа. А он по ошибке затесался в середину.

Таким образом, правильно нарисована только одна, самая последняя, схема (см. рис. 40). В ней нет ни одной ошибки. Главный маршрут идет по шампуру, и все маршруты упорядочены по правилу «Чем правее, тем хуже».

Чтобы превратить плохую схему на рис. 37 в хорошую на рис. 40, достаточно сделать всего две рокировки в обеих развилках.

Таким образом, на основании анализа ряда примеров мы убедились: равносильное преобразование «рокировка» позволяет улучшить эргономичность алгоритмов.

Однако этот вывод относится только к смысловым алгоритмам (где можно указать главный маршрут).

Он совершенно неприменим к буквенным алгоритмам (где понятие главного маршрута теряет силу). Отсюда вытекает, что применение рокировки к буквенной схеме на рис. 34 бессмысленно, так как в данном случае рокировка не влияет на эргономичность.

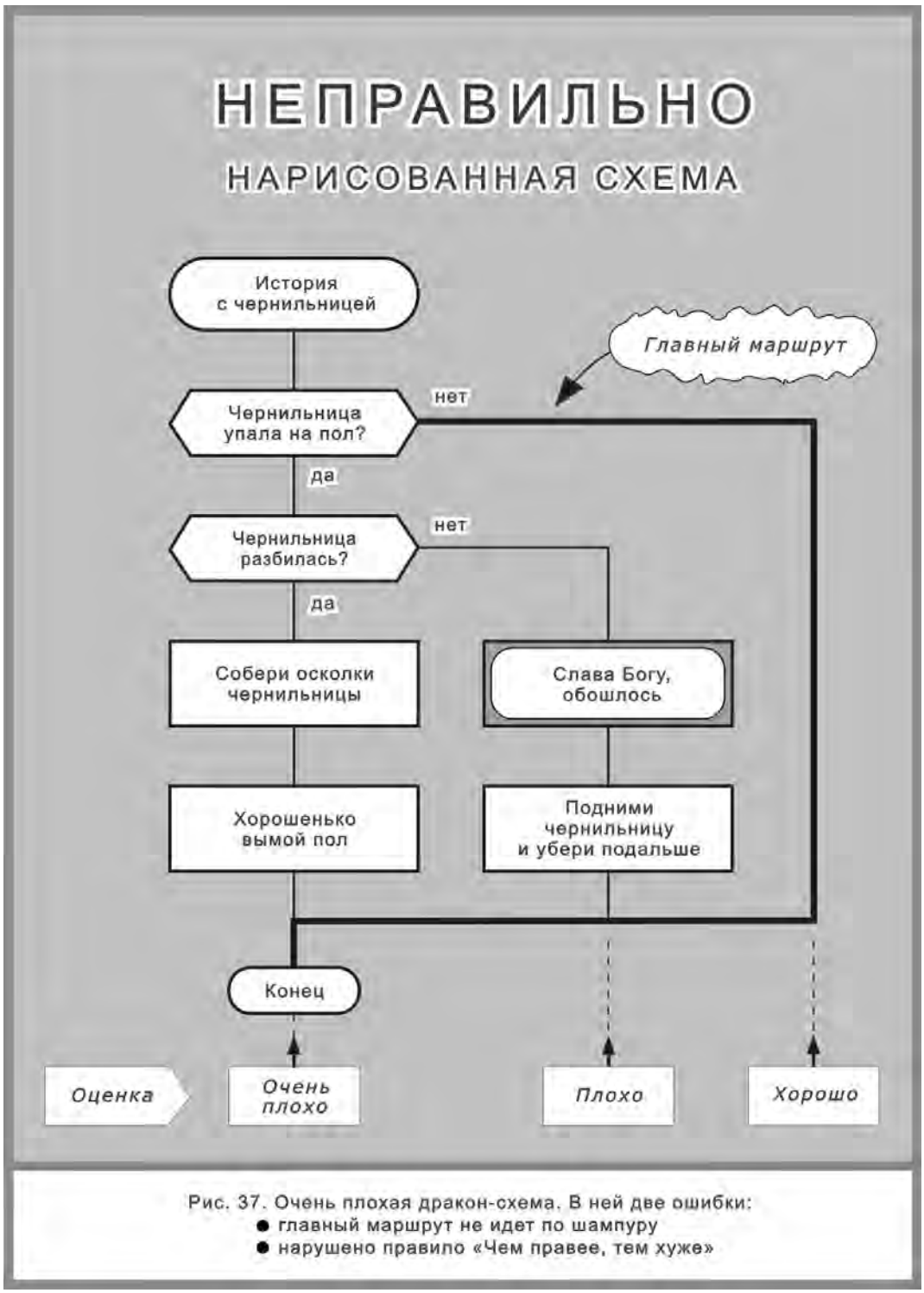


Рис. 37. Очень плохая дракон-схема. В ней две ошибки:
- главный маршрут не идет по шампуру
- нарушено правило «Чем правее, тем хуже»

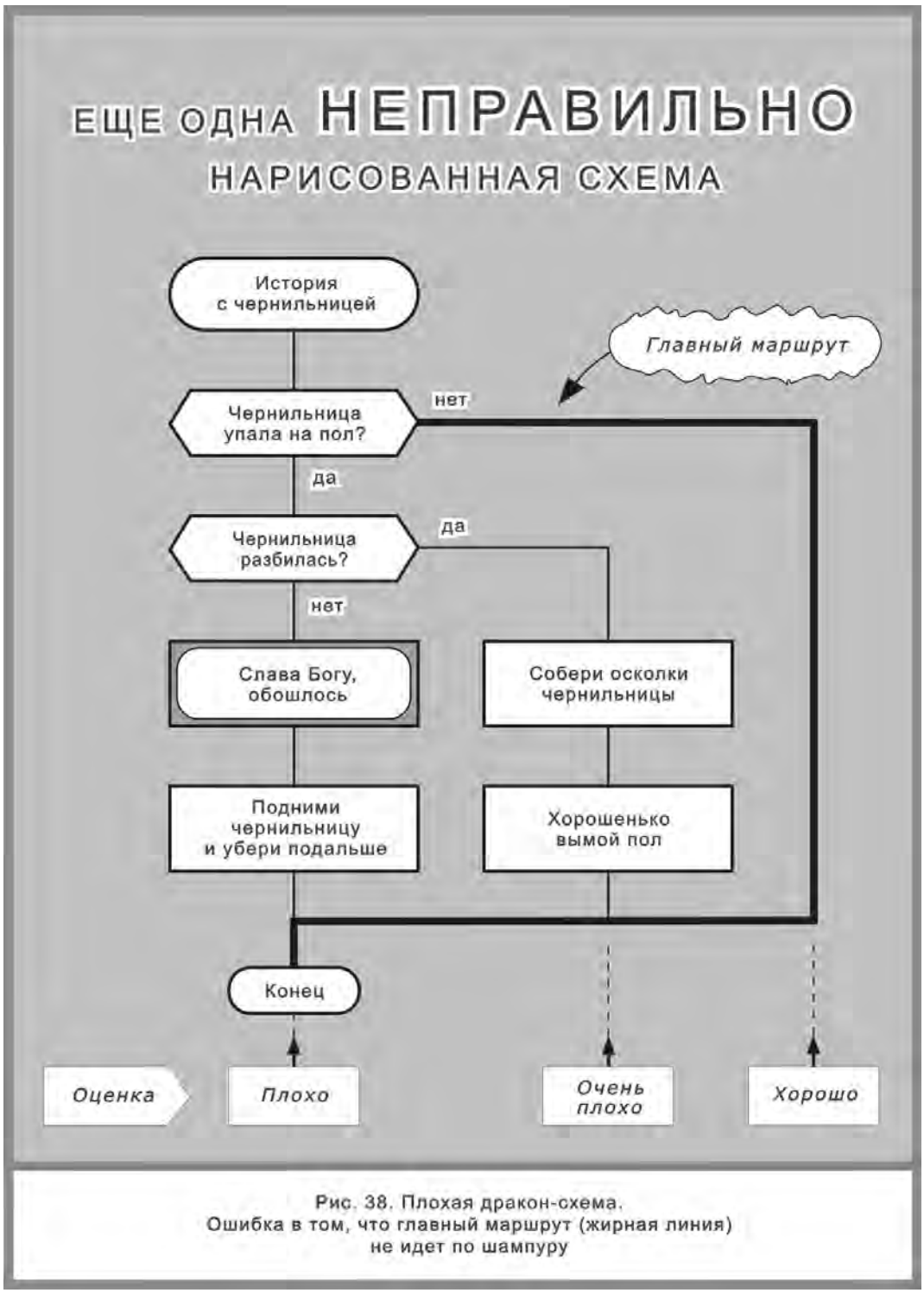


Рис. 38. Плохая дракон-схема.
Ошибка в том, что главный маршрут (жирная линия)
не идет по шампуру

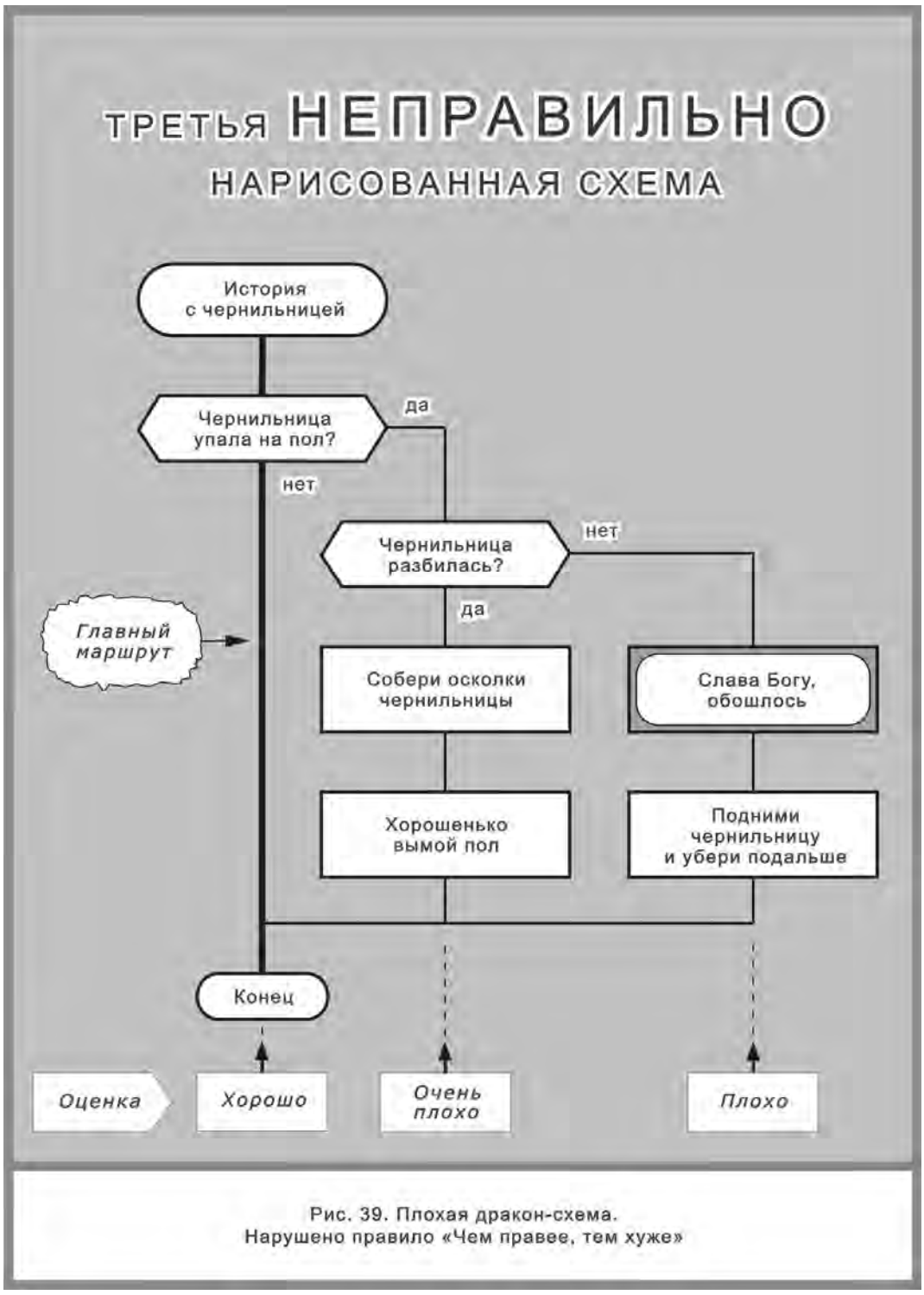


Рис. 39. Плохая дракон-схема.
Нарушено правило «Чем правее, тем хуже»

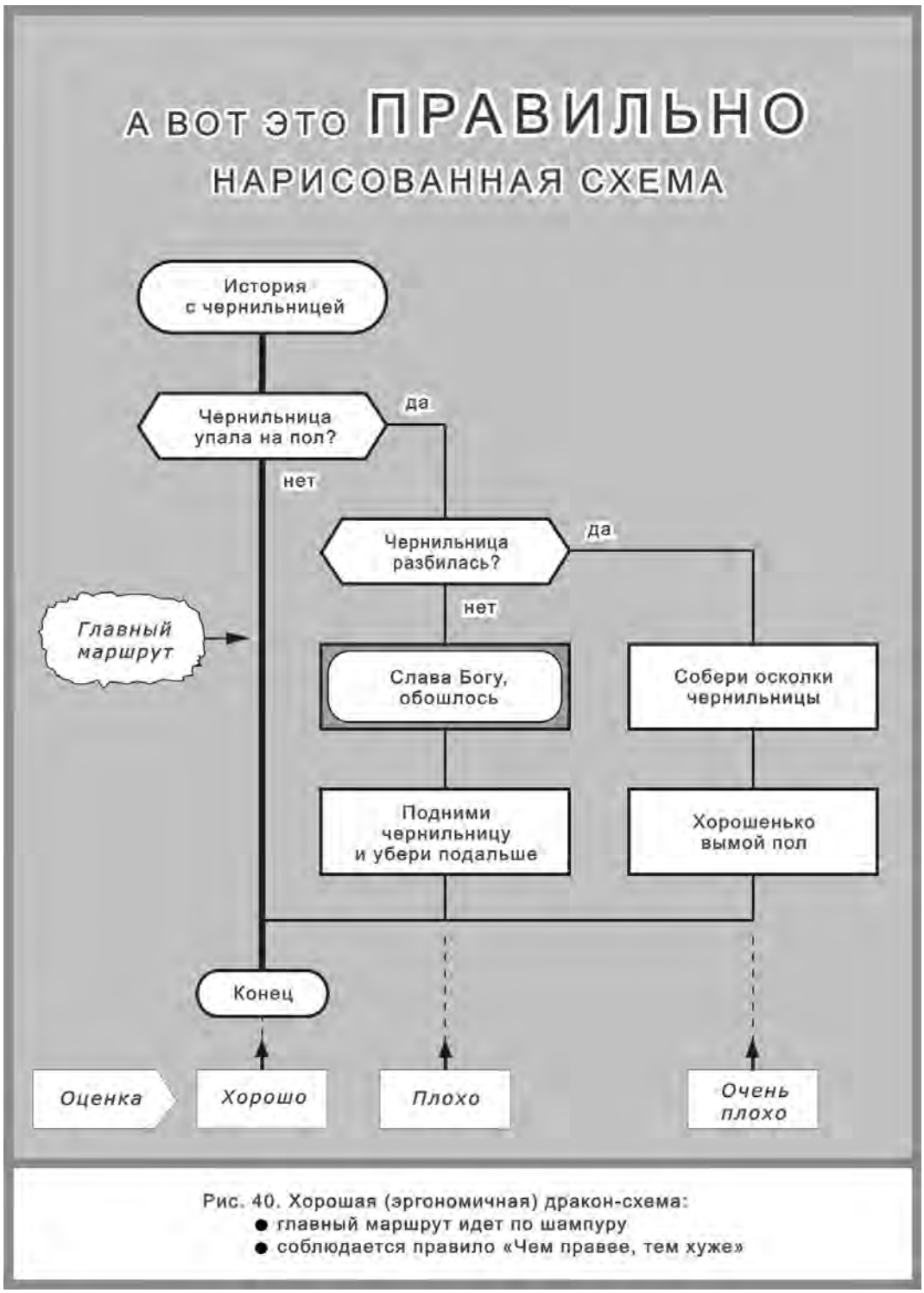


Рис. 40. Хорошая (эргономичная) дракон-схема:
- главный маршрут идет по шампуру
- соблюдается правило «Чем правее, тем хуже»

ВЕРТИКАЛЬНОЕ ОБЪЕДИНЕНИЕ

Иногда бывает, что в дракон-схеме иконы повторяются. Например, на рис. 41 икона «Отдай мотоцикл в ремонт и впредь будь умнее» встречается три раза. Это плохо. Навязчивые повторения раздражают читателя, которому приходится тратить лишнее время и несколько раз читать одно и то же.

К счастью, некоторые повторы можно устранить. Такая возможность появляется, если одинаковые иконы находятся рядом, причем их выходы соединены между собой (рис. 41). В этом случае действует

Правило. **Повторы запрещены.**

Устранение повторов производится с помощью вертикальной линии (рис. 42) и называется *вертикальным объединением*.

При этой операции несколько икон объединяются в одну. Это делается так:

- сначала входы икон объединяются вертикальной линией,
- затем удаляются все одинаковые иконы, кроме крайней левой (рис. 41).

Нетрудно доказать, что операция «вертикальное объединение» есть равносильное преобразование алгоритмов.

Сравнивая алгоритмы на рис. 41 и 42, легко заметить, что схема стала более компактной, ясной и наглядной, поскольку освободилась от бессмысленного повторения икон.

Отсюда проистекает

Вывод. Равносильное преобразование «вертикальное объединение» позволяет улучшить эргономичность алгоритмов.

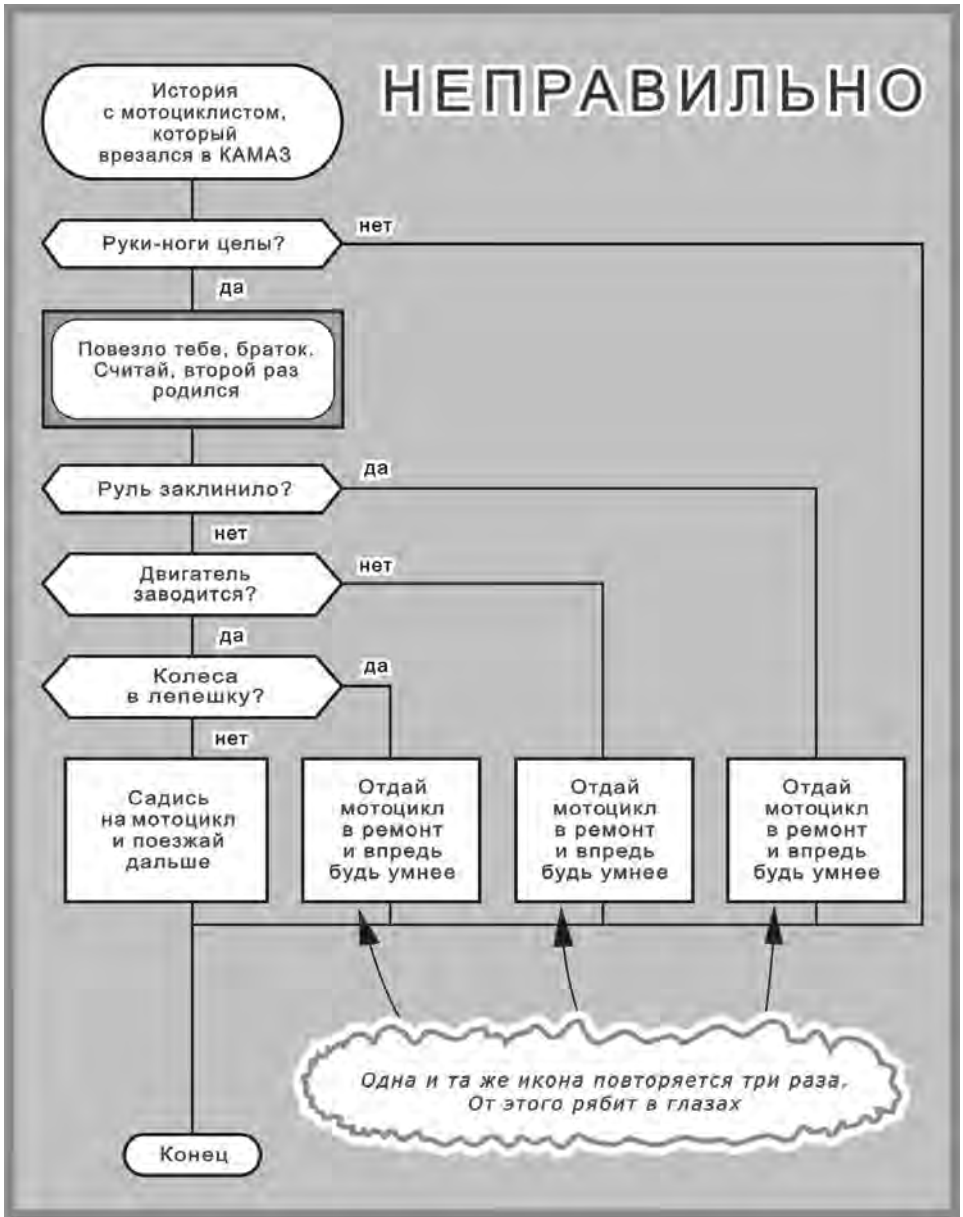


Рис. 41. Плохая дракон-схема. Нарушено правило «Повторы запрещены». К счастью, повторы можно убрать (см. рис. 42)

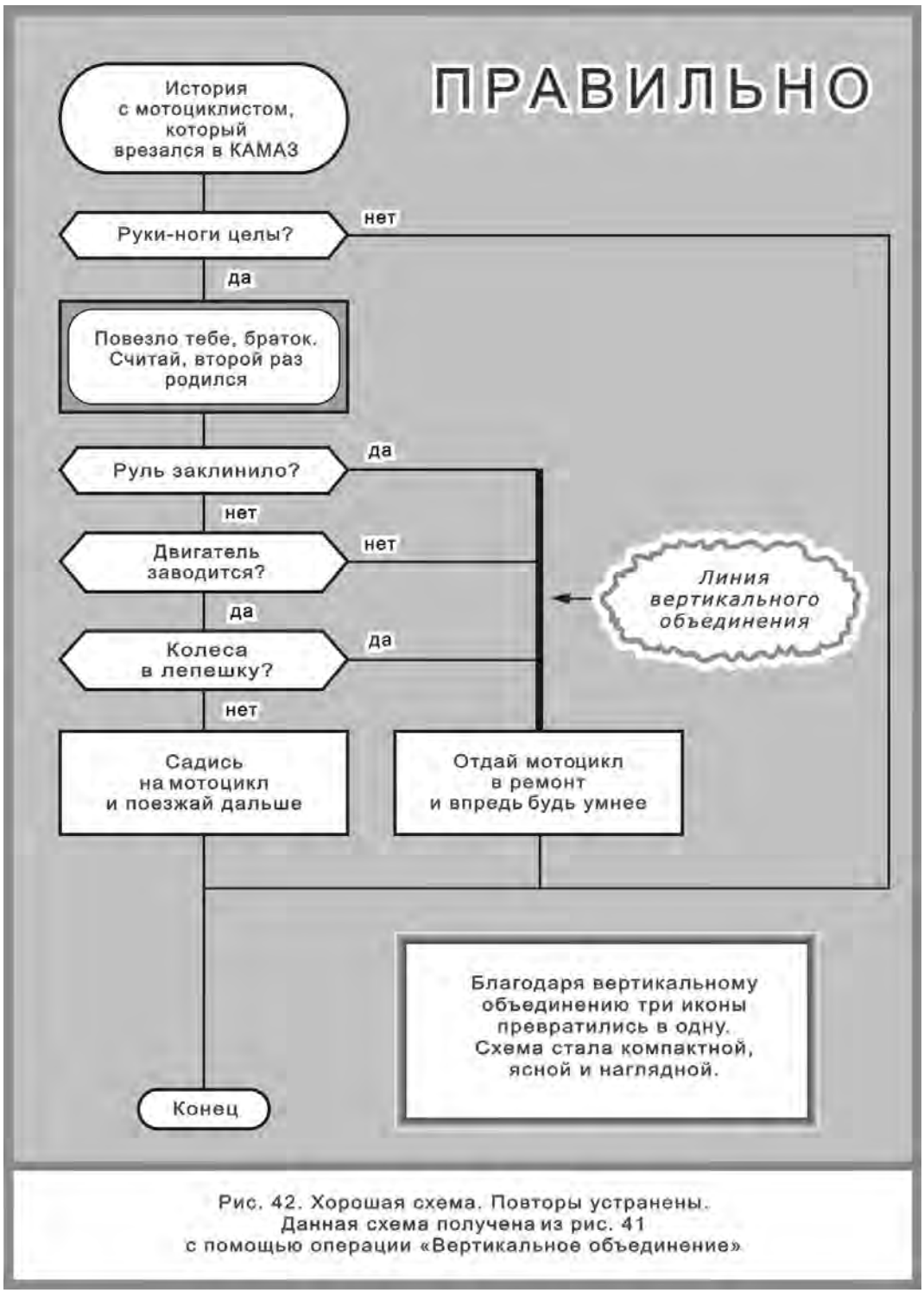


Рис. 42. Хорошая схема. Повторы устранены. Данная схема получена из рис. 41 с помощью операции «Вертикальное объединение»

ГОРИЗОНТАЛЬНОЕ ОБЪЕДИНЕНИЕ

Иногда для исключения повторов используется не вертикальная, а горизонтальная линия. Соответствующая операция называется *горизонтальным объединением*. Она является равносильным преобразованием алгоритмов.

Рассмотрим более сложную задачу на рис. 43, где также имеются повторы, подлежащие удалению.

Сначала устраним повторение иконы «Съешь кашу» и получим схему на рис. 44. В данном случае для исключения повторов используется горизонтальное объединение.

Затем применим вертикальное объединение и исключим повторение развилки «Есть можно?».

Окончательный результат показан на рис. 45. Полученная схема более удобна и занимает меньше места, чем исходная схема на рис. 43. Самое главное, она стала проще и намного понятнее.

Разобранный пример позволяет сделать

Вывод. Равносильное преобразование «горизонтальное объединение» улучшает эргономичность алгоритмов.

П р е д о с т е р е ж е н и е: выполняя вертикальное и горизонтальное объединение, нужно следить, чтобы не появились пересечения соединительных линий.

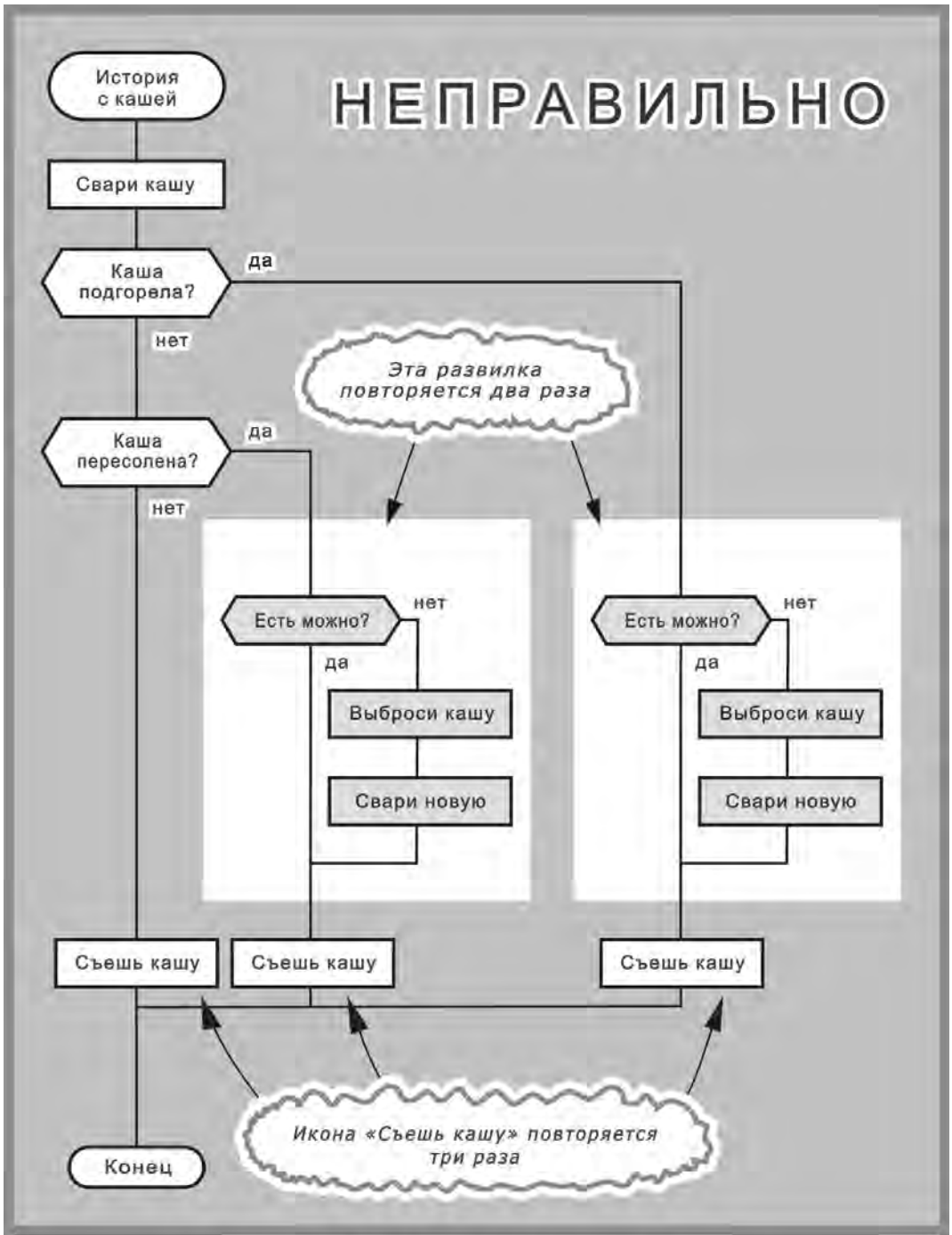


Рис. 43. Очень плохая дракон-схема.
Правило «Повторы запрещены» нарушено в двух местах

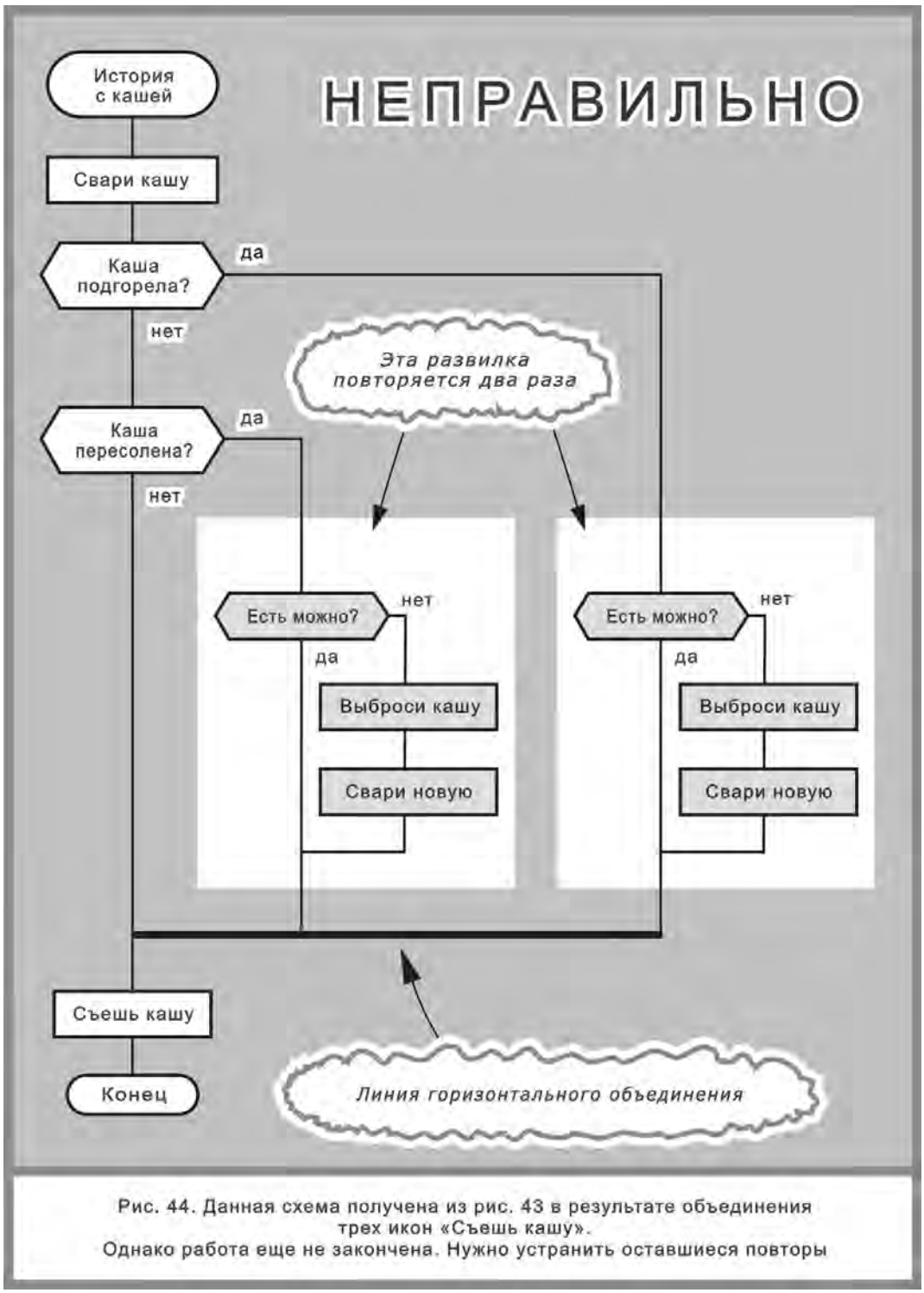


Рис. 44. Данная схема получена из рис. 43 в результате объединения трех икон «Съешь кашу».
Однако работа еще не закончена. Нужно устранить оставшиеся повторы

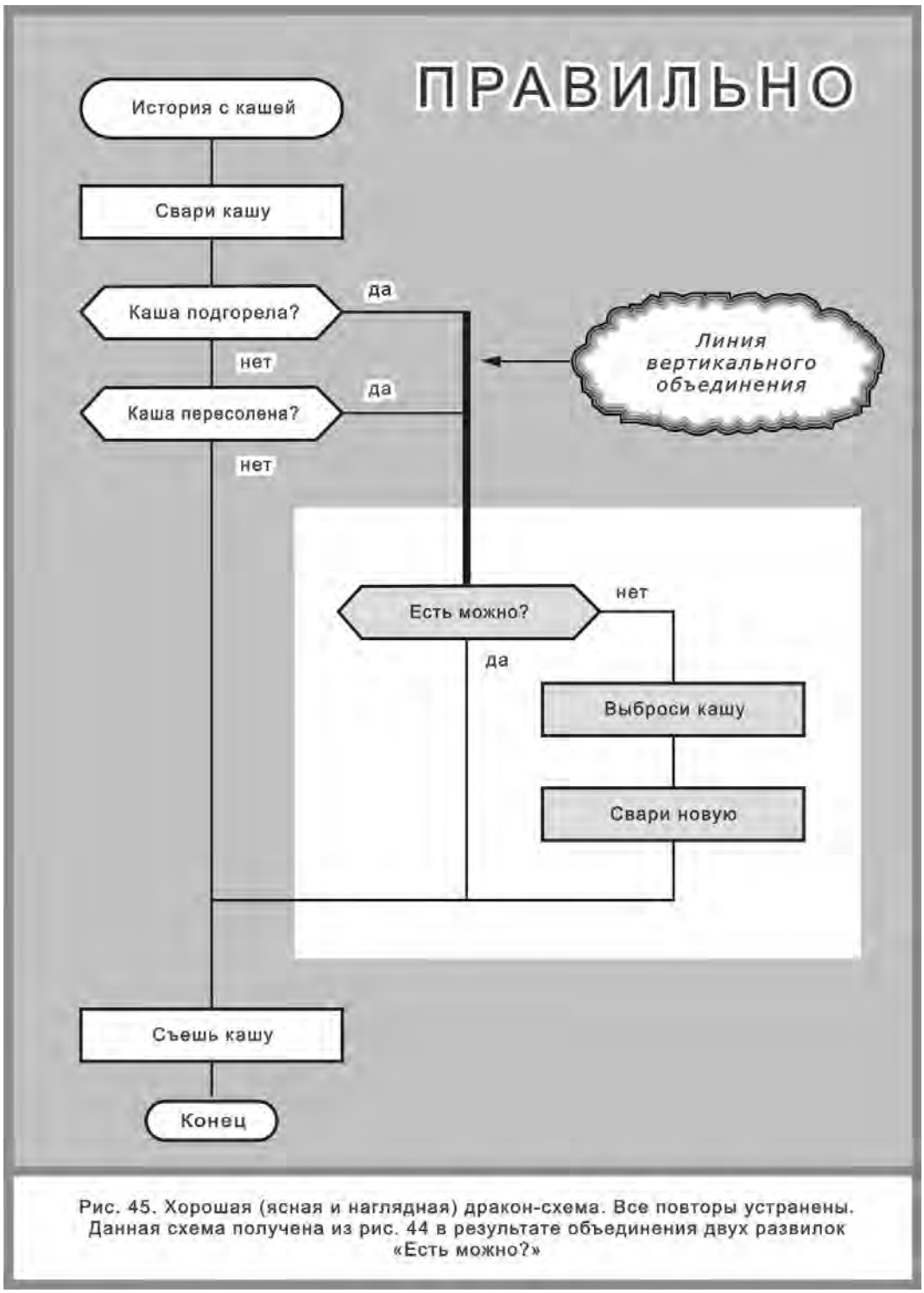


Рис. 45. Хорошая (ясная и наглядная) дракон-схема. Все повторы устранены. Данная схема получена из рис. 44 в результате объединения двух развилок «Есть можно?»

ОПАСНОСТЬ ПЕРЕСЕЧЕНИЙ

На рис. 46 в точке X линии пересекаются. Это очень плохо! Ведь пересечение — визуальная помеха. Она затрудняет восприятие и анализ алгоритмов. Обилие пересечений мешает читателю думать. Практика показывает: пересечение линий — это источник опасности, который может привести к ошибке.

Поэтому в языке ДРАКОН действует правило: *пересечения запрещены.*

Существуют специальные приемы позволяющие устранить пересечения. Самый простой из них показан на рис. 46 и 47.

ПРОСТОЕ ДОКАЗАТЕЛЬСТВО

Можно доказать, что схемы на рис. 46 и 47 эквивалентны (имеют один и тот же смысл).

На рис. 46 имеются три маршрута:

Номер маршрута	Описание маршрута
Маршрут 1	А—Б—В
Маршрут 2	А—Б—Д
Маршрут 3	А—Г

Легко убедиться, что схема на рис. 47 имеет точно такие же маршруты. Совпадение маршрутов говорит о том, что на обеих схемах представлен один и тот же алгоритм. Значит, схемы 46 и 47 равносильны.

КАКАЯ ОШИБКА ПОДСТЕРЕГАЕТ НАС ПРИ ОБЪЕДИНЕНИИ?

На рис. 48 (слева) икона *Е* повторяется три раза. На первый взгляд кажется, что две иконы *Е* можно убрать с помощью операции «горизонтальное объединение». Действуя подобным образом, получим результат на рис. 48 (в центре).

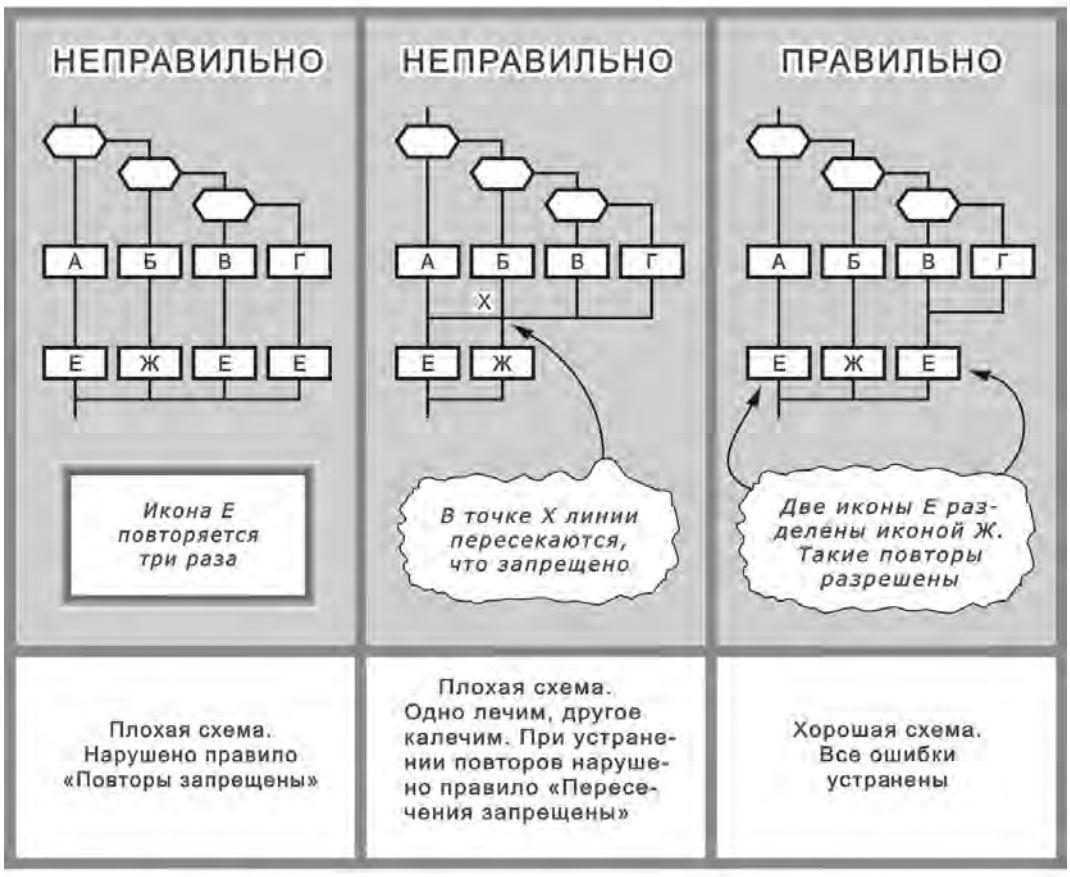
Мы допустили грубую ошибку! Вспомним, что при рисовании драконсхем пересечения запрещены. А у нас получилось, что две линии пересекаются в точке *Х*. Отсюда следует

> *Правило.* **Разрешается объединять не любые одинаковые иконы, а только соседние.**

Обратите внимание: на рис. 48 (слева) из трех одинаковых икон *Е* только две правые являются соседними. А третья (крайняя левая) отделена от них иконой *Ж*. Поэтому она не может участвовать в объединении. Правильный ответ показан на рис. 48 справа.

Рис. 46. Плохая схема.
Есть пересечение в точке Х

Рис. 47. Хорошая схема.
Пересечение устранено



Плохая схема.
Нарушено правило
«Повторы запрещены»

Плохая схема.
Одно лечим, другое калечим. При устранении повторов нарушено правило «Пересечения запрещены»

Хорошая схема.
Все ошибки устранены

Рис. 48. Устраняя повторы, следите, чтобы не появились пересечения

ЧТО ДЕЛАТЬ, ЕСЛИ ЭРГОНОМИЧЕСКИЕ ТРЕБОВАНИЯ ПРОТИВОРЕЧАТ ДРУГ ДРУГУ?

До сих пор мы рассматривали простейшие случаи, когда различные эргономические требования не вступали в конфликт. Однако конфликты возможны. Вот два эргономических критерия, которые могут противоречить друг другу:

- правило главного и побочных маршрутов,
- минимизация числа вертикалей.

Чтобы исключить конфликт, следует соблюдать

Принцип приоритета. **Правило главного и побочных маршрутов имеет более высокий приоритет, нежели стремление уменьшить число вертикалей.**

В качестве иллюстрации сравним эквивалентные схемы на рис. 49 и 50. По критерию «минимизация числа вертикалей» выигрывает схема на рис. 49. У нее на одну вертикаль меньше.

А как насчет порядка в маршрутах? Попробуем разобраться.

На рис. 49 правило маршрутов грубо нарушено, причем сразу в двух местах. Во-первых, главный маршрут (когда человек здоров) не совмещен с шампуром. Во-вторых, маршруты не упорядочены слева направо. Действительно, самое тяжелое заболевание (когда человек вынужден лечь в больницу) находится на средней вертикали. Это неправильно, так как слева и справа от нее находятся более легкие недомогания.

Таким образом, правило «Чем правее, тем хуже» не соблюдается. Поэтому схема на рис. 49 является плохой, неэргономичной.

Чтобы исправить недостаток, необходимо выполнить:

- *вертикальное разъединение* в точке *C* (эта операция обратна вертикальному объединению);
- рокировку развилки «Заболел?»,
- рокировку развилки «Врач помог?».

В итоге получим безукоризненно четкую схему на рис. 50. Здесь все маршруты упорядочены по принципу «Чем правее — тем хуже».

В самом деле, левая вертикаль означает, что дела идут хорошо, ибо человек здоров. Значит, главный маршрут идет по шампуру. Вторая вертикаль описывает легкое недомогание, которое можно снять таблеткой. Третья вертикаль говорит: самочувствие ухудшилось, нужен врач. Наконец, четвертая (крайняя правая) вертикаль означает, что дела плохи — пришлось лечь в больницу.

Как уже упоминалось, схема на рис. 50 тоже не без греха — в ней на одну вертикаль больше, чем на рис. 49. Тем не менее, мы признаем ее наилучшей, поскольку выполняется более приоритетное правило маршрутов.

Отсюда вытекает, что (благодаря наличию приоритетов) комплекс эргономических требований является взаимоувязанным и непротиворечивым.

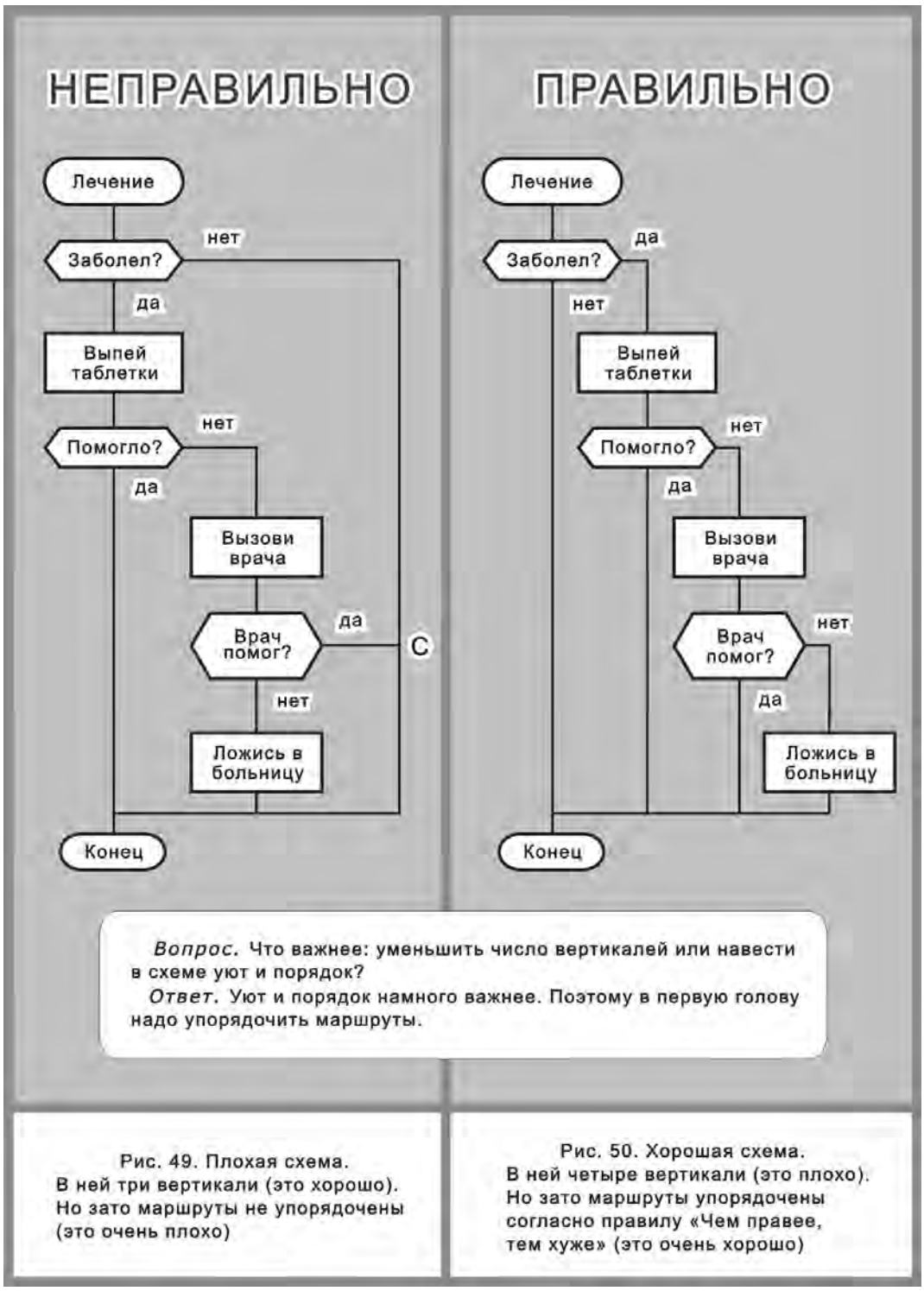


Рис. 49. Плохая схема. В ней три вертикали (это хорошо). Но зато маршруты не упорядочены (это очень плохо)

Рис. 50. Хорошая схема. В ней четыре вертикали (это плохо). Но зато маршруты упорядочены согласно правилу «Чем правее, тем хуже» (это очень хорошо)

ЭРГОНОМИЧНОСТЬ АБСТРАКТНЫХ АЛГОРИТМОВ

Можно ли улучшить эргономичность абстрактных (буквенных) дракон-схем с помощью равносильных преобразований? Мы уже знаем, что рокировка в этом случае бесполезна. Однако вертикальное и горизонтальное объединение позволяют заметно повысить эргономичность буквенных схем.

Чтобы убедиться в этом, обратимся к рис. 51 и 52. В самом деле, схема на рис. 51 выглядит неоправданно громоздкой. Она содержит семь вертикалей, тринадцать икон и заставляет читателя шесть раз читать идентификатор *B*, чтобы убедиться, что правые шесть икон одинаковые.

А равносильная ей схема на рис. 52 (полученная методом вертикального объединения) свободна от этого недостатка.

Она наглядна, проста и изящна. Содержит только две вертикали и восемь икон. Занимает втрое меньше места на листе бумаги (на экране). И к тому же имеет всего два прямоугольных излома линий (на рис. 51 — семь изломов).

Таким образом, буквенных схема на рис. 52 более эргономична, чем ее соседка.

Еще более громоздкой выглядит абстрактная схема на рис. 53, в которой насчитывается 14 вертикалей.

А эквивалентная ей схема на рис. 54 (полученная методом вертикального и горизонтального объединения) снова выигрывает. Она позволяет уменьшить число вертикалей в три раза (с 10 до 3). Сокращает число икон более чем в два раза (с 45 до 19). Обеспечивает более экономное топологическое упорядочивание маршрутов. Заметно сокращает суммарную длину соединительных линий.

Проведенный анализ позволяет сделать

Вывод. В отличие от рокировки, которая полезна только для смысловых дракон-схем, вертикальное и горизонтальное объединение улучшают эргономичность не только смысловых, но и абстрактных алгоритмов.

Рис. 51. Плохая схема.
В ней слишком много вертикалей (семь)
и одинаковых икон (шесть)

Рис. 52. Хорошая схема. Число
вертикалей и одинаковых икон
удалось значительно сократить

Рис. 53. Плохая схема. В ней слишком много икон и соединительных линий, без которых вполне можно обойтись

Рис. 54. Хорошая схема. Число икон и соединительных линий удалось значительно сократить

ИКОНА-ВСТАВКА КАК ЭРГОНОМИЧЕСКИЙ ПРИЕМ

Мы уже знаем, что язык ДРАКОН запрещает применять пересечения, обрывы и соединители. Отсюда вытекает жесткое ограничение: любая дракон-схема должна целиком размещаться на одном листе бумаги. А если она слишком большая и вылезает за рамки прокрустова ложа?

Тогда можно взять бумагу больших размеров, например формата А1. А если схема громадная и все равно не помещается? На этот случай предусмотрены специальные приемы, позволяющие уменьшить габариты дракон-схемы и разрубить ее на удобные куски. Эти приемы позволяют разместить дракон-схему на листах желаемого размера. Рассмотрим проблему на «миниатюрном» примере.

Предположим, линейный алгоритм состоит из двенадцати икон, а бумажный лист небольшой, так что на нем можно разместить по вертикали не более восьми икон. Как быть?

На рис. 55 и 56 показаны два варианта решения проблемы. Алгоритм на рис. 55 не годится, так как на участке *АБ* бегунок (рабочая точка) движется вверх, что запрещено правилами языка ДРАКОН.

Преодолеть затруднение можно двумя способами:

- применить конструкцию «силуэт» (о которой шла речь в главе 1);
- примнить прием «разрежь великана» и разделить алгоритм на части.

Рассмотрим последний способ. Для этого удалим из алгоритма несколько связанных по смыслу икон. А вместо них нарисуем икону-заместитель, которая называется *вставкой* (рис. 56). Вставка нужна, чтобы напомнить об изъятых иконах. Вставка занимает мало места — намного меньше, чем выброшенные иконы, поэтому алгоритм становится короче. Разумеется, выброшенные иконы не пропадают — они образуют новый алгоритм — *процедуру*.

На рис. 57 показано, как взаимодействуют основной алгоритм и процедура.

Икона-вставка — это команда «Передай управление в процедуру».

Икона «конец» процедуры означает: «Верни управление в основной алгоритм». При этом управление возвращается в точку, расположенную после иконы-вставки.

На языке программистов икона-вставка — это оператор «Вызов процедуры».

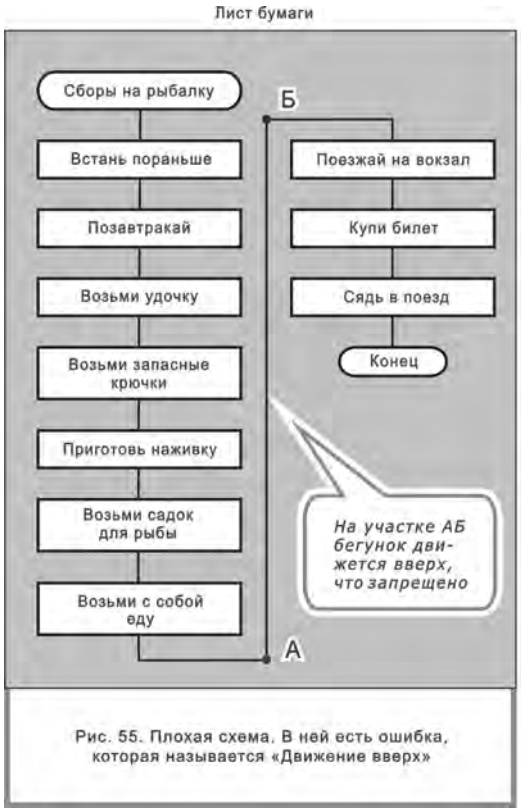


Рис. 55. Плохая схема. В ней есть ошибка, которая называется «Движение вверх»

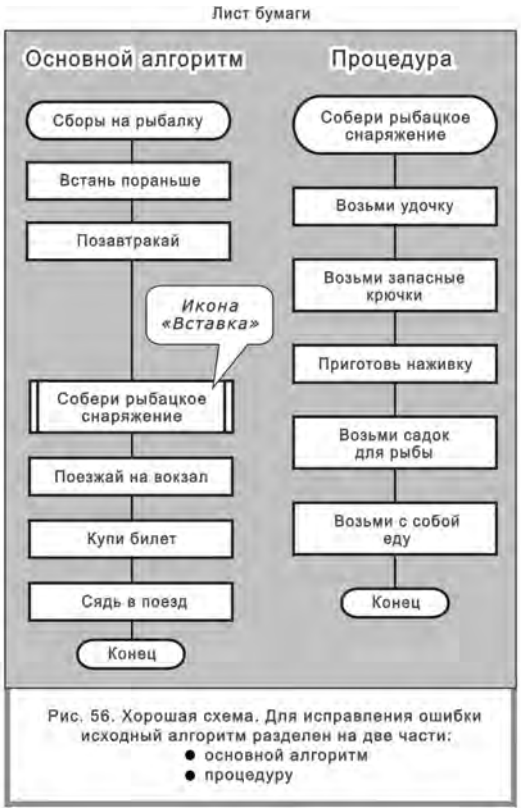


Рис. 56. Хорошая схема. Для исправления ошибки исходный алгоритм разделен на две части:
- основной алгоритм
- процедуру

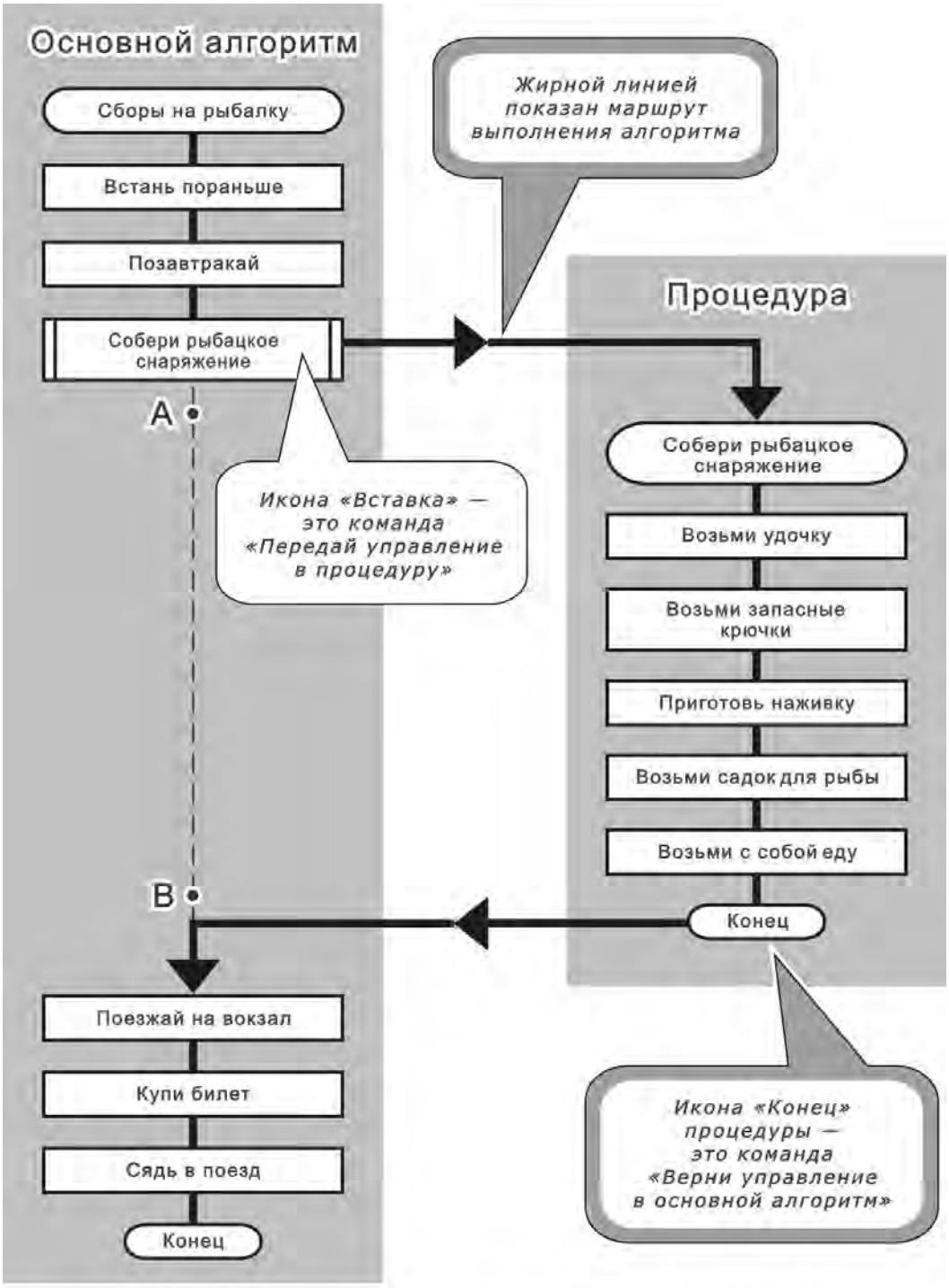


Рис. 57. Как взаимодействуют основной алгоритм и процедура?

ЧТО ТАКОЕ ПОДСТАНОВКА?

Операция «подстановка» связана с использованием иконы-вставки. Она выполняется за три шага.

Шаг 1. Из дракон-схемы удаляется фрагмент, имеющий один вход и один выход.

Шаг 2. Вместо него подставляется икона-вставка с именем *X*.

Шаг 3. К удаленному фрагменту добавляется икона-заголовок с тем же именем *X* и икона-конец. В результате получается процедура.

Два алгоритма на рис. 55 и 56 неравносильны, так как их формулы не совпадают. Ведь маршрут на рис. 55 содержит 11 икон, а на рис. 56 — 15 икон (см. также рис. 57).

Вместе с тем нетрудно убедиться, что подстановка — эквивалентное преобразование алгоритмов, так как исходный и преобразованный алгоритмы дают одинаковые результаты для одних и тех же исходных данных. Попутно заметим, что равносильные алгоритмы всегда эквивалентны (обратное неверно).

УЛУЧШЕНИЕ ЭРГОНОМИЧНОСТИ АЛГОРИТМОВ С ПОМОЩЬЮ МАТЕМАТИКИ

Когда рисуешь алгоритм, нужно стремиться, чтобы он с самого начала удовлетворял правилам и был эргономичным. А если это не получилось? Если первый блин комом, как на рис. 37? В этом случае необходимо довести алгоритм до ума с помощью изложенных выше приемов.

Мы рассмотрели ряд операций, позволяющих выполнить эквивалентное преобразование алгоритмов (при котором смысл алгоритма не меняется). К их числу относятся: рокировка, вертикальное объединение, горизонтальное объединение и подстановка. Подчеркнем еще раз, что эти операции являются математически строгими. С другой стороны, они позволяют улучшить эргономическое качество алгоритмов. Отсюда следует фундаментальный

Вывод. Понимаемость (эргономичность) алгоритмов можно повысить с помощью строгих математических методов.

Для удобства читателя на рис. 58 дана общая сводка эквивалентных преобразований.

КРАСОТА И ИЗЯЩЕСТВО АЛГОРИТМОВ

Алгоритм, представленный в письменном виде, предназначен для зрительного восприятия человеком. Следовательно, алгоритм представляет собой зрительную сцену. Или, если угодно, — зрительный образ, зрительную картину.

Красота алгоритма — это красота его зрительного образа, в частности, красота дракон-схемы. Алгоритм можно назвать *красивым* в том случае, если процесс зрительного восприятия, понимания и постижения алгоритма протекает с максимальной скоростью, наименьшими усилиями и максимальным эстетическим наслаждением.

Чем красивее алгоритм, тем быстрее и легче можно его понять. Отсюда вытекает, что красота и элегантность алгоритмов открывают путь к экономии умственных усилий. Но не только.

Чем красивее зрительные образы частей алгоритма, чем изящнее они соединены в общую (алгоритмическую) картину, тем приятнее на них смотреть. Чем точнее и элегантнее зрительный «пейзаж» обнажает глубинный смысл алгоритма, тем плодотворнее мышление.

Чем больше красоты, тем глубже понимание алгоритмов. Чем скорее течет наша алгоритмическая мысль, тем легче мы постигаем суть дела. Тем быстрее и качественнее протекает важнейший производственный процесс, играющий немалую роль в мировой экономике, — процесс массовой разработки алгоритмов и программ.

И наоборот, если зрительный образ алгоритма кажется некрасивым, неприятным, отталкивающим и запутанным, процесс понимания и обдумывания неизбежно замедляется, что снижает производительность умственного труда.

ДРАКОН — графический язык, язык зрительных образов. С учетом сказанного можно уточнить: ДРАКОН — язык красивых зрительных образов. Но красота ДРАКОНА — не самоцель. Она позволяет ощутимо повысить производительность труда при создании алгоритмов.

Мы исходим из того, что зрительные образы алгоритмов следует сознательно проектировать. Для этой цели можно (в разумных пределах) использовать *средства художественного конструирования.* Уместно напомнить слова видного психолога Бориса Ломова:

> «Средства художественного конструирования в конечном счете направлены на то, чтобы вызвать тот или иной эффект у работающего человека… Применяя средства художественного конструирования, мы создаем положительные эмоции, облегчаем операцию приема информации человеком, улучшаем концентрацию и переключение внимания, повышаем скорость и точность действий. Короче говоря, мы пользуемся этими средствами для управления поведением человека в широком смысле слова, для управления его психическим состоянием» и умственной работоспособностью [1].

ПРАВИЛА, РОЖДАЮЩИЕ АЛГОРИТМИЧЕСКУЮ КРАСОТУ

О каких правилах идет речь? Приведем несколько примеров.

> *Правило лаконичности.* **Зрительный образ алгоритма должен быть лаконичным. Все ненужные, лишние детали должны быть отсечены.**

Поясним. Блок-схема алгоритма должна содержать лишь те элементы, которые необходимы для сообщения читателю существенной информации, точного понимания ее значения и стимулирования правильных решений и разумных действий. Пустые украшения, избыточные, затемняющие детали должны быть удалены.

Характеризуя это правило, Ф. Эшфорд пишет:

> «Бесполезно стремиться направить внимание на важнейшие характеристики, если они окружены лишними, не относящимися к ним визуальными раздражителями, мешающими восприятию главного» [2].

ДЬЯВОЛ ТАИТСЯ В ПОДРОБНОСТЯХ

Чтобы найти и изгнать «дьявола», надо засучить рукава и заняться чисткой авгиевых конюшен. Надо заглянуть во все углы, внимательно рассмотреть мельчайшие детали и устранить затаившиеся в них недостатки. Это грязная и неблагодарная работа. Однако обойтись без нее нельзя. Чтобы добиться «великой победы», надо устранить тысячи вредных мелочей. Одна из таких мелочей — ненужные изломы соединительных линий.

> *Правило устранения изломов.* **Чтобы алгоритм был удобным для чтения, количество изломов соединительных линий должно быть минимальным.**

Из двух схем лучше та, где число изломов меньше. Сравним две схемы на рис. 59 и 60. На рис. 59 показана обычная блок- схема, заимствованная из технической литературы [3]. На рис. 60 изображена эквивалентная ей дракон-схема.

Эти рисунки позволяют выявить различия между уродливой блок-схемой и красивой дракон-схемой. С точки зрения правил, рождающих алгоритмическую красоту, блок-схема на рис. 59 имеет следующие недостатки.

- Неоправданно большое число изломов линий (в блок-схеме 12 изломов, а в дракон-схеме только 4).
- Большое число паразитных элементов: 14 стрелок и 3 кружка, которые в дракон-схеме отсутствуют (поскольку они совершенно не нужны и представляют собой визуальные помехи, затемняющие суть дела).

Мы рассмотрели два эргономических закона — *правило лаконичности и правило устранения изломов.* Мы убедились, что в дракон-схеме эти правила строго соблюдены, а в блок-схеме грубо нарушены.

КРИТИЧЕСКИЙ АНАЛИЗ

Обратимся снова к рис. 59 и 60. Мы сделали лишь первый шаг к расчистке авгиевых конюшен. На рис. 59 осталось еще немало «грязи», которую необходимо отмыть.

Итак, вооружимся микроскопом и продолжим наш критический анализ. Блок-схема на рис. 59 имеет следующие недочеты.

- Для обозначения развилки используется ромб, который занимает слишком много места и не позволяет поместить внутри необходимое количество удобочитаемого текста, состоящего из строк равной длины. В дракон-схеме верхний и нижний углы ромба «отпилены». Поэтому схема становится компактной и удобной как для записи текста, так и для чтения.
- Функционально однородные иконы Д, Е, Ж, И хаотично разбросаны по всей площади чертежа, занимая три разных горизонтальных уровня (что путает читателя). В дракон-схеме они расположены на одном уровне, что служит для читателя наглядной подсказкой об их функциональной однородности.
- Ромбы имеют выход влево, что разрушает шампур и не позволяет применить правило главного маршрута. В дракон-схеме выход влево не допускается.
- Икона Д и ее вертикаль расположены слева от шампура (в дракон-схеме это запрещено).
- Ниже икон Ж и И находятся три уровня горизонтальных линий, которые имеют паразитный характер. В дракон-схеме три уровня сведены в одну линию, что делает схему еще более наглядной и компактной.

Да, конечно, каждое из этих улучшений является маленьким и не делает погоды. Но когда мелкие улучшения исчисляются тысячами и становятся массовыми, ситуация может измениться. Количество переходит в качество. В этом случае облегчение умственного труда может стать значительным.

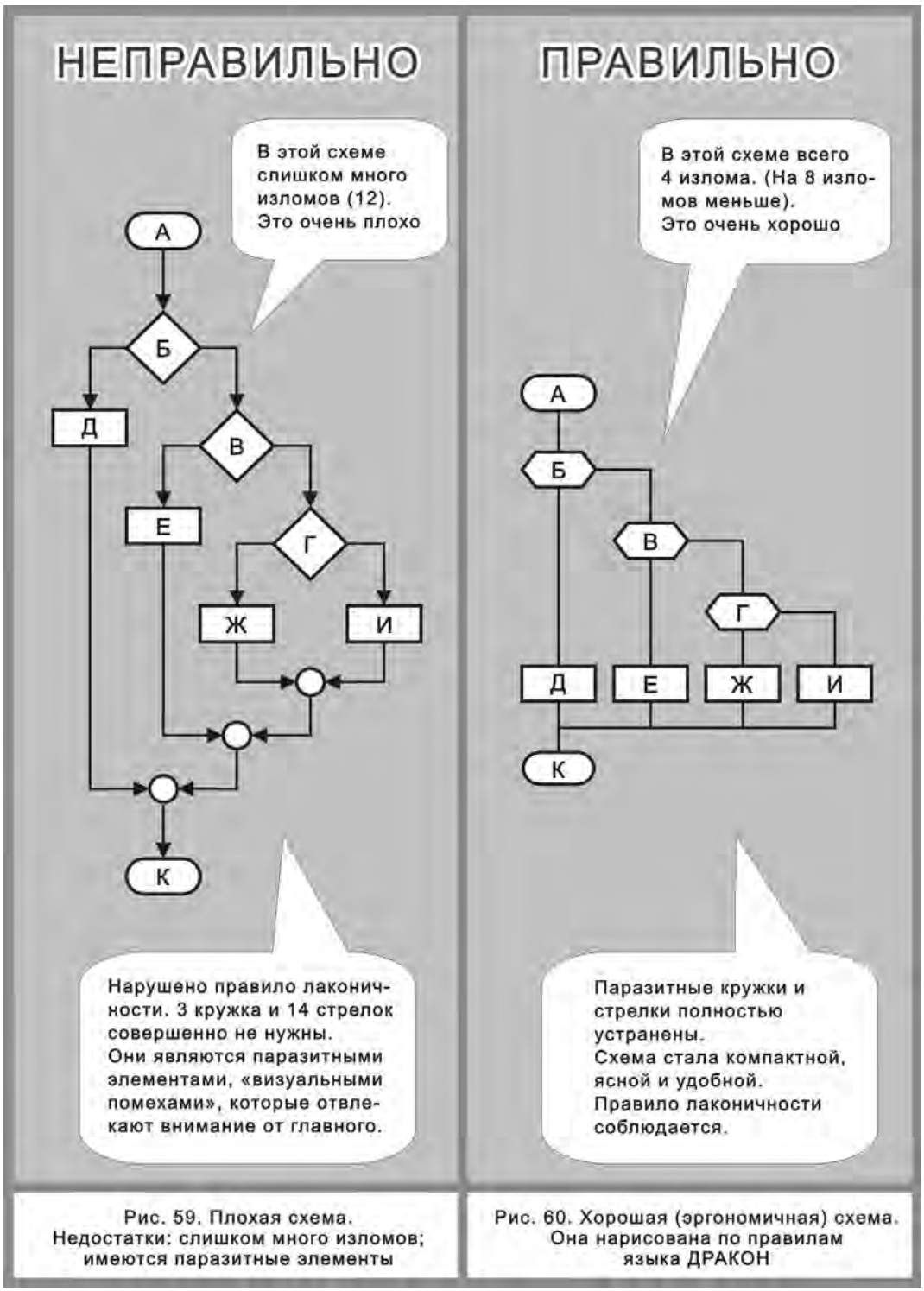


Рис. 59. Плохая схема. Недостатки: слишком много изломов; имеются паразитные элементы

Рис. 60. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

«АЛГОРИТМИЧЕСКИЕ МЯТЕЖНИКИ»

В последнее время появились алгоритмические «мятежники», которые затеяли восстание против вчерашних авторитетов. Они называют традиционные блок-схемы «кучей мусора».

Образчик подобной кучи мусора представлен на рис. 61 [4]. Этот «мусорный» алгоритм мы подвергли косметической операции в «салоне алгоритмической красоты». И превратили в изящную дракон-схему (рис. 62). Сравним, что было и что стало.

Схема на рис. 61 имеет множество изъянов.

- Слева от иконы Ж есть пересечение линий (в дракон-схеме пересечения запрещены).
- Возле иконы Е имеется линия под углом 45° (в дракон-схеме наклонные линии не допускаются).
- Иконы Д, Е и Ж имеют более одного выхода (в дракон-схеме это запрещено).
- Иконы В, Д, Е, Ж имеют входы сбоку, что придает схеме неряшливый вид. В дракон-схеме вход разрешается только сверху, что упорядочивает алгоритм и создает в нем четкую ориентацию «сверху вниз»).
- Отсутствует шампур, так как выход иконы «заголовок» и вход иконы «конец» не лежат на одной вертикали. Исчезновение шампура означает, что в схеме отсутствует зрительный остов, художественно-композиционная главная вертикаль. Тем самым уничтожается основа для наглядного выделения главного маршрута.

Таким образом, последняя блок-схема (рис. 61), как и предыдущая (рис. 59), по всем параметрам проигрывает дракон-схеме.

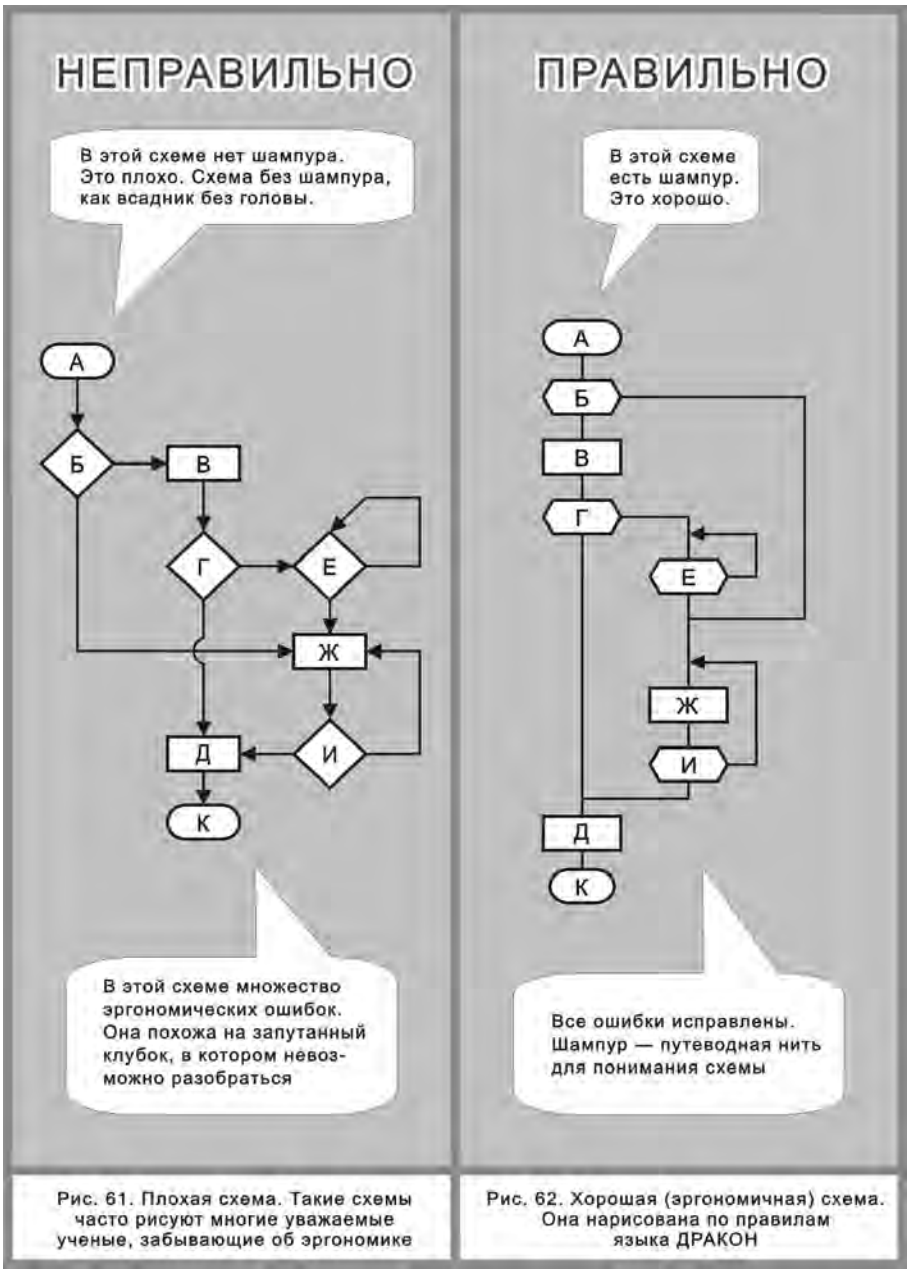


Рис. 61. Плохая схема. Такие схемы часто рисуют многие уважаемые ученые, забывающие об эргономике

Рис. 62. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

ЗАЧЕМ ЗДЕСЬ СТРЕЛКИ? РАЗВЕ ОНИ ВНОСЯТ ЯСНОСТЬ?

Предыдущие примеры плохих блок-схем были случайным образом взяты из технической литературы. Следующий пример (рис. 63) скопирован из источника [5], где он характеризуется как «стандартная блок-схема ANSI» (Американский национальный институт стандартов). Блок-схема, выполненная по этому стандарту, также имеет многочисленные дефекты:

- Ниже иконы G имеет место разрыв шампура (нарушено правило, согласно которому один из путей, идущих от входа к выходу, должен проходить по главной вертикали).
- Икона G имеет два входа (в дракон-схеме разрешается только один вход).
- Икона G имеет вход сбоку (в дракон-схеме это запрещено).
- У иконы G выход находится слева (в дракон-схеме он должен быть снизу).
- Две петли обратной связи обычного цикла находятся слева от шампура и закручены по часовой стрелке (в дракон-схеме они расположены справа от шампура и закручены против часовой стрелки).
- Используются неудобные ромбы (в дракон-схеме их заменяют эргономичные иконы “вопрос”).
- Ромб L имеет выход слева (в дракон-схеме он должен быть справа).
- Используются 12 стрелок, из которых 10 — паразитные (в дракон-схеме всего 2 стрелки).
- Имеется один избыточный излом линии (в блок-схеме 9 изломов, в дракон-схеме только 8).

Таким образом, американская блок-схема, как и предыдущие примеры, по всем параметрам проигрывает дракон-схеме (рис. 64).

Мы еще раз убедились, что алгоритмическая красота достигается благодаря совокупному действию многих правил, каждое из которых, взятое по отдельности выглядит скромным и будничным.

Нетрудно заметить, что красота дракон-схем во многом достигается за счет стандартизации графических узоров. В «плохих» блок-схемах входы и выходы икон изображаются как угодно. Стандартизация отсутствует, превращая блок-схемы в царство анархии.

В отличие от них язык ДРАКОН задает строгие эргономичные стандарты, направленные на достижение алгоритмической красоты. Это значит, что графические символы, обозначающие одни и те же объекты или явления, должны быть унифицированы. Они должны иметь единое графическое решение.

Это касается не только формы икон и правил присоединения к ним входов и выходов. Этому закону подчиняются и более глобальные правила, регламентирующие графический узор всего алгоритма, как целостного зрительного образа.

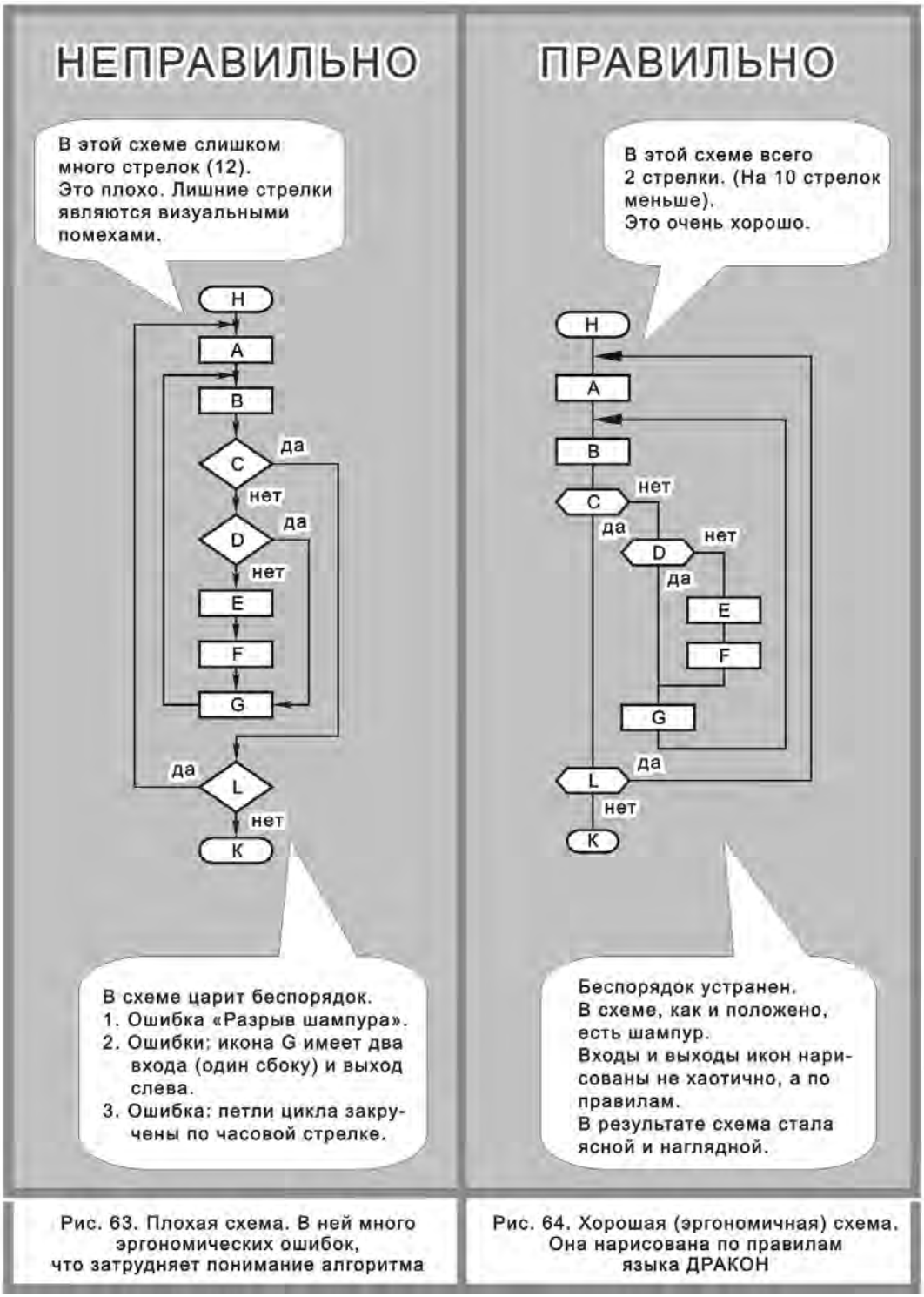


Рис. 63. Плохая схема. В ней много эргономических ошибок, что затрудняет понимание алгоритма

Рис. 64. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

ЭПОХА ПОНЯТНЫХ АЛГОРИТМОВ И ВСЕОБЩАЯ АЛГОРИТМИЧЕСКАЯ ГРАМОТНОСТЬ

До сих пор одним из важнейших недостатков теории алгоритмов была ее недостаточная связь с практикой программирования. Это приводило к тому, что разработка и отладка алгоритмов и программ превратилась в сверхтрудный интеллектуальный процесс. В результате производительность труда алгоритмистов и программистов значительно отстает от потребностей практики. Причина этого недостатка — низкая понимаемость алгоритмов и программ. Причем этот недостаток невозможно устранить чисто математическими методами.

Но есть и хорошая новость. Объединение усилий математики и эргономики является плодотворным для решения задачи.

Сказанное позволяет предположить, что объявленная выше стратегическая цель — построение ясных, понятных и доходчивых алгоритмов, пригодных для более глубокого взаимопонимания между соисполнителями работ — вполне достижима.

Если это верно, то мы находимся на пороге новой эпохи — *эпохи понятных алгоритмов.* Впервые в истории во всем мире сложные алгоритмы станут легкими для понимания! Это значит, что будет реализована заветная мечта многих алгоритмистов, программистов и заказчиков программных комплексов.

Вместо нынешних «уму непостижимых» алгоритмических джунглей повсюду засияют волшебным светом «удивительно наглядные описания алгоритмов и процессов». Перед нашими восхищенными очами откроется новый мир — мир дружелюбных алгоритмов, в котором будет царить необыкновенная легкость и глубина понимания.

Можно надеяться, что дальнейшее развитие теории и практики эргономизации алгоритмов проложит широкий путь ко *всеобщей алгоритмической грамотности* (в тех пределах, в которых подобная задача в принципе может быть решена).

ВЫВОДЫ

1. Предполагается, что понятие «эргономичный алгоритм» должно стать фундаментальным понятием теории алгоритмов.
2. Применение достижений эргономики к теории алгоритмов позволяет значительно улучшить понимаемость алгоритмов и программ, усилить творческую продуктивность алгоритмистов и программистов.
3. Понятие эргономичного алгоритма задается с помощью конечного набора четко определенных правил и признаков, таких, как «главный маршрут должен идти по шампуру» и т. д. Следовательно, это понятие является строгим.
4. Эргономичность алгоритмов можно улучшить с помощью простых и ясных методов, в частности, с помощью математически строгих равносильных преобразований алгоритмов (рокировки, вертикального и горизонтального объединения и подстановки).

Глава 3

ВИЗУАЛИЗАЦИЯ ЦИКЛИЧНЫХ АЛГОРИТМОВ

> Успешность принятия решения во многом зависит от способности человека «визуализировать проблемную ситуацию», наглядно представлять ее и оперировать наглядными образами.
>
> *Наталья Завалова, Борис Ломов, Владимир Пономаренко [1]*

ПРОБЛЕМА ЦИКЛИЧНЫХ АЛГОРИТМОВ

В этой главе речь пойдет о графических циклах языка ДРАКОН. Проблема в том, что программисты привыкли к текстовой записи циклов, сжились с ней. Для тех, кто всю жизнь использовал текстовую запись, графическая запись (точнее, графический способ мышления о циклах) может показаться непривычным.

Чтобы исключить трудности, мы будем рассказывать о циклах на самых простых бытовых примерах, не избегая юмористических приемов. Как говорил великий Блез Паскаль, «предмет математики настолько серьезен, что полезно не упускать случая сделать его немного занимательным».

Теперь о самом главном. Существующие циклы, используемые во всем мире, имеют серьезный недостаток. Они накладывают на творческую мысль алгоритмиста неоправданные ограничения. Графика позволяет снять многие из этих ограничений. В результате алгоритмическая мысль становится более естественной и плодотворной.

ОБЫЧНЫЙ ЦИКЛ

В языке ДРАКОН имеется следующий ассортимент циклов:

- обычный цикл;
- переключающий цикл;
- цикл ДЛЯ;
- веточный цикл;
- цикл ЖДАТЬ.

Первые четыре цикла рассматриваются в этой главе, цикл ЖДАТЬ — в главе 5.

Составной графический оператор «обычный цикл» (рис. 5, макроикона 4) содержит иконы «вопрос» и «петля цикла» (рис. 4, иконы И4, И11). Он охватывает циклы трех типов (рис. 65—67):

- цикл ДО (do-while),
- цикл ПОКА (while-do),
- гибридный цикл (do-while-do).

Визуально отличить их очень легко. У цикла ДО вопрос рисуют внизу, а действие вверху (рис. 65). У цикла ПОКА — все наоборот (рис. 66). Гибридный цикл — это «помесь» цикла ДО и цикла ПОКА (рис. 67).

Рис. 65. Цикл ДО

Рис. 66. Цикл ПОКА

Рис. 67. Гибридный цикл

ЦИКЛ «ДО»

В цикле ДО действие выполняется до вопроса. Это значит, что бегунок сначала пробегает через одну или несколько икон «Действие», потом — через икону «Вопрос». Например, в цикле на рис. 68 сначала выполняются два действия:

- Покрась одну доску
- Шагни вправо на ширину доски

И только после этого задается вопрос: «Все доски покрашены?». При ответе «нет» описанные два действия выполняются снова и снова.

Когда все доски будут покрашены, бегунок выходит из иконы «вопрос» через «да». И алгоритм заканчивает работу.

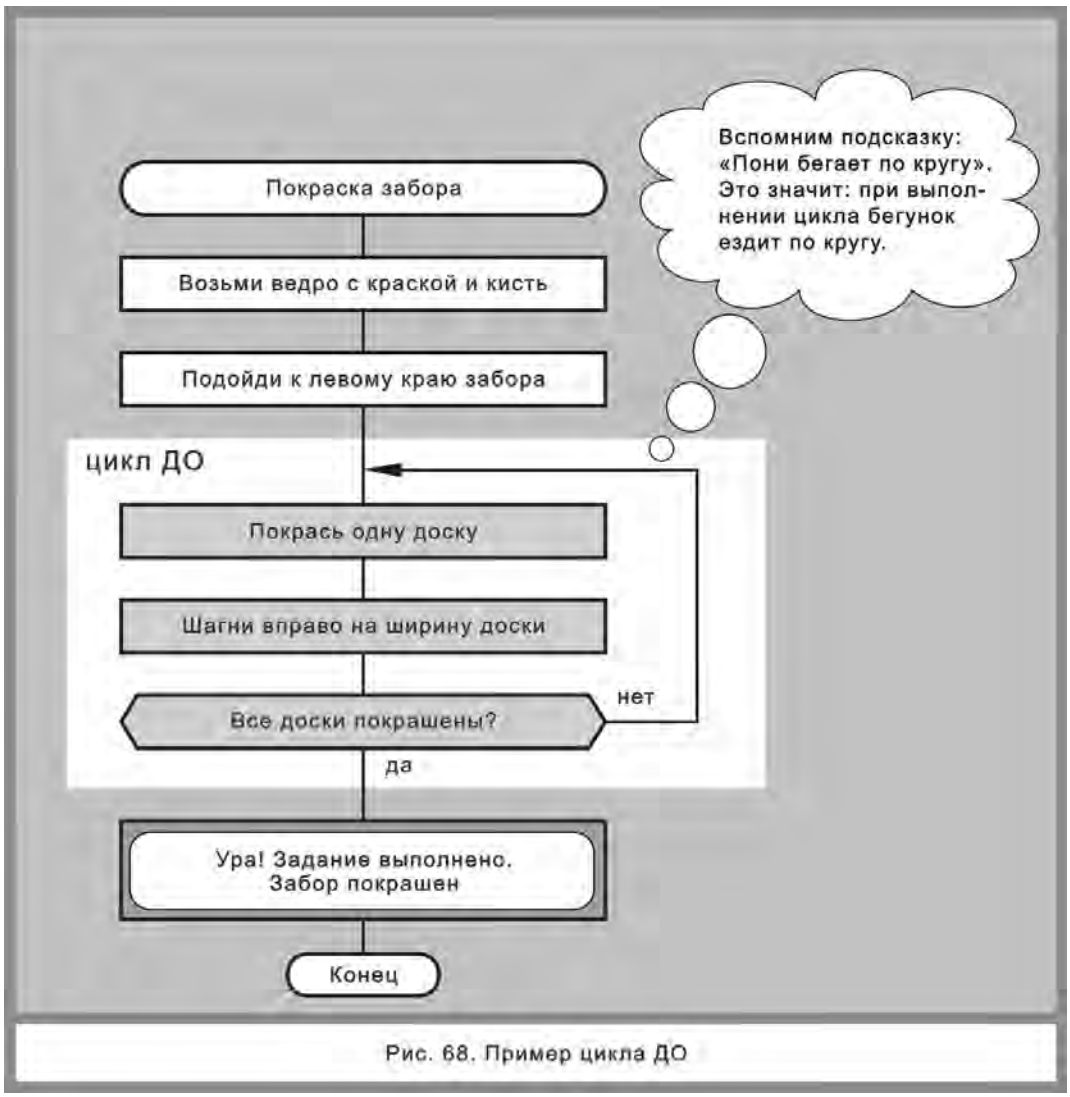


Рис. 68. Пример цикла ДО

ЦИКЛ «ПОКА»

В цикле ПОКА иная картина. Действие либо вообще не выполняется, либо выполняется после вопроса. Бегунок сначала движется через икону «вопрос», а затем (если ответ благоприятный) — через икону «действие».

Обратимся к примеру на рис. 69. Однажды Карлсон, который живет на крыше, нашел кошелек и открыл его. А что случилось дальше? Здесь возможны варианты.

Вариант 1. Кошелек оказался пустым. Поэтому Карлсону не удалось купить плюшку.

Вариант 2. В кошельке всего одна денежка, так что Карлсон смог купить только одну плюшку.

Вариант 3. В кошельке целая куча монет. Поэтому Карлсон купил гору плюшек и наелся до отвала.

Рассмотрим дело подробнее. Цикл на рис. 69 начинается с вопроса: «В кошельке есть денежки?».

Если денег нет, из иконы «вопрос» бегунок выходит через «нет», и алгоритм сразу заканчивается. Следовательно, действие «Возьми из кошелька денежку» не выполняется ни разу.

Если же деньги есть, бегунок выходит через «да» и начинает кружить по маршруту АБВГДЕЖА. При этом выполняются действия, образующие тело цикла:

- Возьми из кошелька денежку
- Купи себе плюшку
- Съешь плюшку

Когда деньги кончатся, бегунок выходит из иконы «вопрос» через «нет». И алгоритм заканчивается.

Чем различаются циклы ПОКА и ДО?

Цикл ПОКА может либо ни разу не выполняться, либо выполняться один раз, либо много раз (два и более). А цикл ДО обязательно выполняется хотя бы один раз.

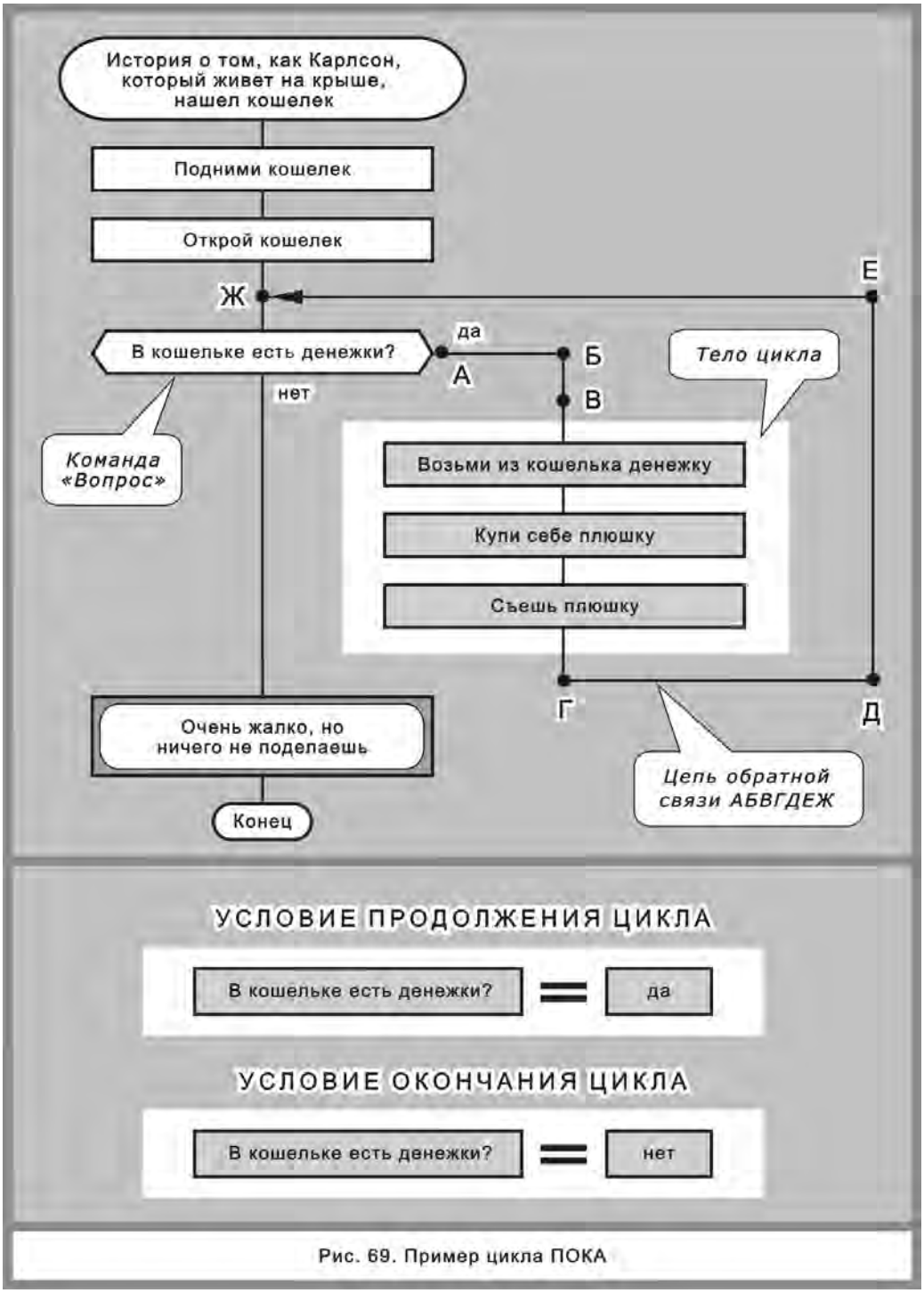


Рис. 69. Пример цикла ПОКА

УСЛОВИЕ ПРОДОЛЖЕНИЯ И ОКОНЧАНИЯ ЦИКЛА

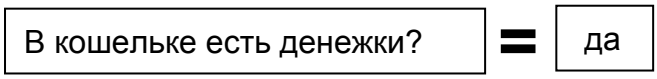
Введем два новых понятия. Вообще говоря, эти понятия существовали всегда (без них цикл просто не может работать), но их описание было либо скомканным, либо вообще оставалось за кадром.

Уже говорилось, что в иконе «вопрос» записан да-нетный вопрос, то есть логическая переменная величина, принимающая значение «да» или «нет».

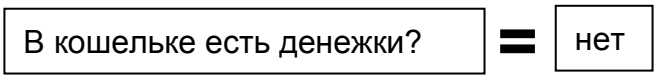
Условие продолжения цикла — условие, определяемое значением да-нетного вопроса («да» или «нет»), при котором цикл продолжает работать.

Условие окончания цикла — условие, определяемое значением да-нетного вопроса («да» или «нет»), при котором цикл заканчивает работу.

На рис. 69 условие продолжения цикла имеет вид



Когда деньги кончатся, логическая переменная изменит свое значение с «да» на «нет». В этот момент условие продолжения цикла исчезнет (станет неистинным). И появится условие окончания цикла:



В обычных языках значения указанных условий жестко регламентируются. Например, в языке Паскаль в цикле ДО *(repeat-until)* выход из цикла производится только тогда, когда логическое выражение принимает значение *true* (да). Чтобы выполнить столь жесткое условие, иногда приходится делать противоестественную и вычурную запись.

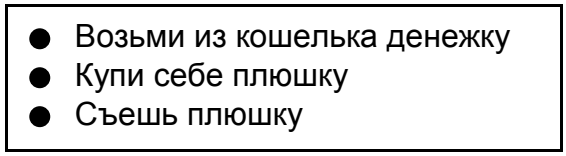
В языке ДРАКОН подобные надуманные ограничения полностью отсутствуют, что (в сочетании с другими эргономическими приемами) заметно облегчает разработку алгоритмов.

ДОСРОЧНЫЙ ВЫХОД ИЗ ЦИКЛА «ПОКА»

Карлсон, который живет на крыше, может съесть очень много плюшек. Наверное, штук сто. Или даже двести. Но не больше! Иначе он просто лопнет. А теперь предположим, что в кошельке, на его счастье (или беду), оказалось пятьсот монет.

Зададим вопрос: как в этой ситуации будет работать алгоритм на рис. 69?

Бедный Карлсон! Ему не позавидуешь. Алгоритм заставит его съесть пятьсот плюшек. Все до единой! И он наверняка умрет от обжорства. Почему? Потому что из цикла на рис. 68 нельзя выйти раньше времени. Вспомним — в кошельке пятьсот монет. Значит каждая команда цикла



будет исполнена ровно пятьсот раз. И лишь затем на вопрос: «В кошельке есть денежки?» — мы получим ответ «нет» и сможем уйти из цикла.

Отсюда следует вывод. Чтобы спасти Карлсона, нужно организовать *досрочный* выход из цикла. Для этого в алгоритм нужно ввести дополнительную команду-вопрос: «Карлсон уже наелся?» (рис. 70).

Что это даст? Допустим, в кошельке пятьсот монет, а Карлсон может съесть всего двадцать плюшек. После того как цикл повторится двадцать раз, на вопрос: «Карлсон уже наелся?» — будет получен ответ «да». И мы благополучно выйдем из цикла.

Обратите внимание: цикл на рис. 70 имеет не один, а два выхода: *основной* и *досрочный*. Через первый мы выходим, когда в кошельке кончились деньги. Через второй — когда Карлсон наелся.

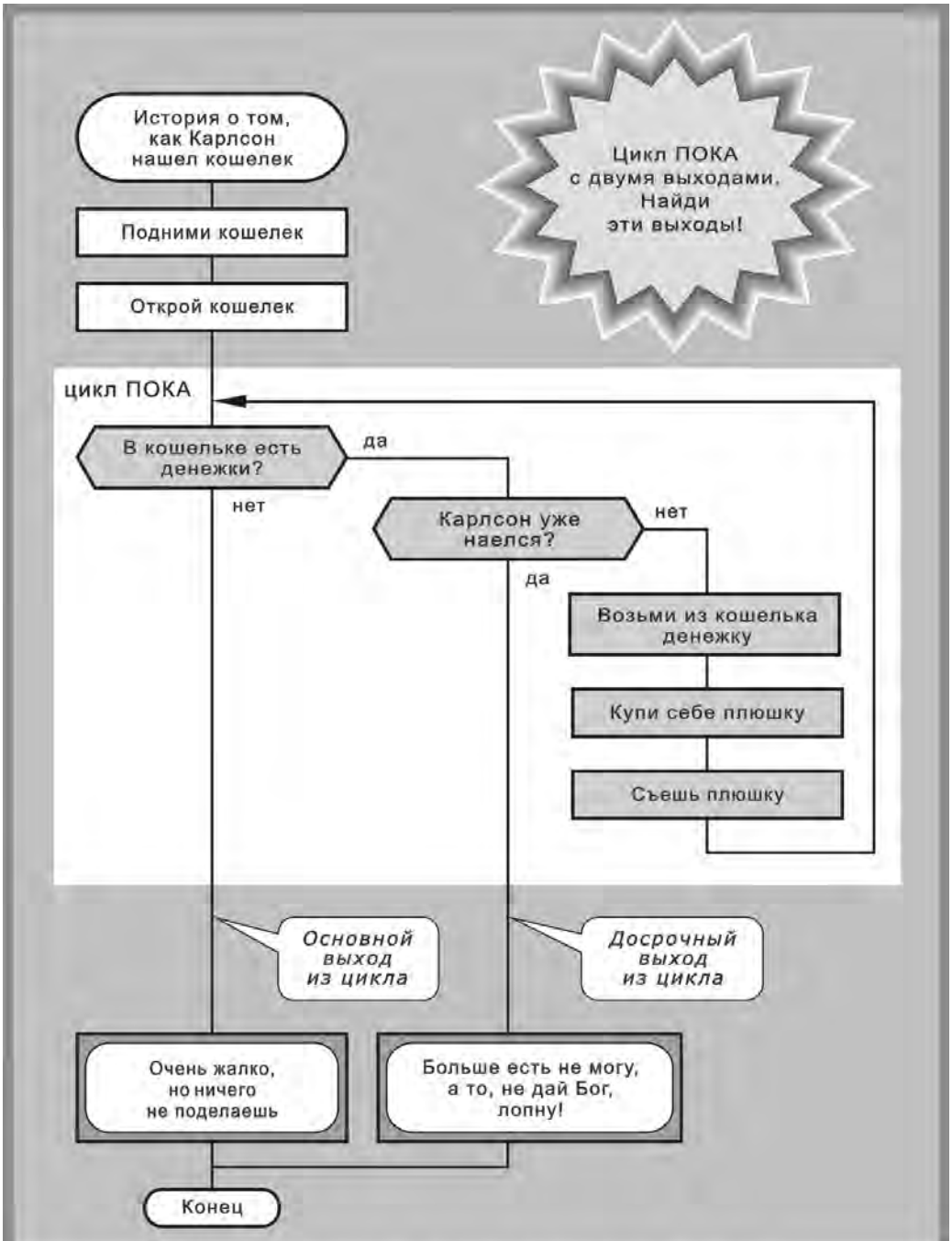


Рис. 70. Досрочный выход из цикла происходит потому, что Карлсон больше не хочет есть. Сравни с рис. 68.

ДОСРОЧНЫЙ ВЫХОД ИЗ ЦИКЛА «ДО»

Папино слово — закон! А папа сказал: сегодня нужно покрасить забор. На рис. 68 показан случай, когда все идет по плану и работа успешно доводится до конца.

Однако в жизни вечно случается то одно, то другое. Например, ни с того ни с сего кончилась краска. Этот случай представлен на рис. 71. Алгоритм на рис. 71 имеет два выхода из цикла: *основной* (забор удалось покрасить) и *досрочный* (забор остался недокрашенным).

Однако, что значит «кончилась краска»? Одно дело, если краски нет в ведре — тогда ее можно взять в сарае. И совсем другое, если в сарае краски тоже нет. Последний случай показан на рис. 72.

Жизнь полна неожиданностей. Кому охота красить дурацкий забор, если все нормальные люди уже играют в футбол? Этот полезный для футбола и вредный для забора случай отражен на рис. 73. Данный алгоритм интересен тем, что в нем три выхода из цикла: основной и два досрочных. В первом случае забор будет покрашен как надо. Во втором дело не ладится из-за нехватки краски. В третьем — из-за любви к футболу.

Сказанное можно обобщить и записать

Правило. **В цикле всего один вход. А выходов может быть много. Один выход — основной, остальные — досрочные.**

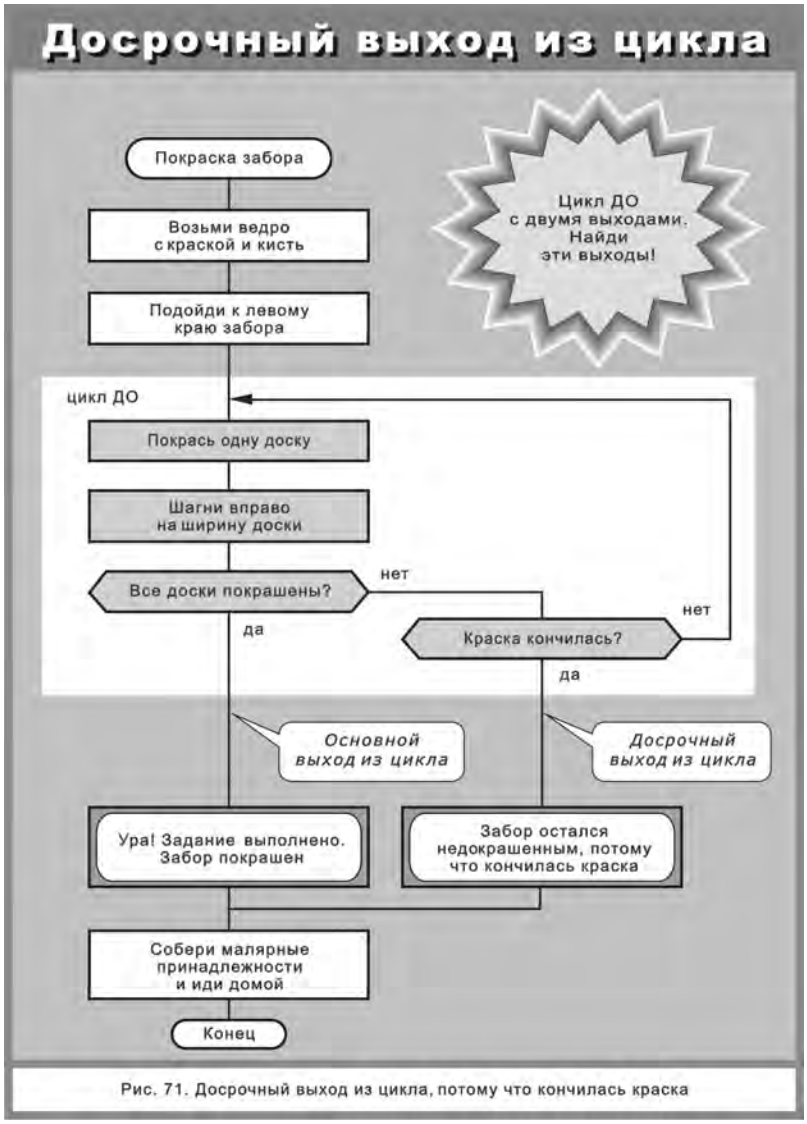


Рис. 71. Досрочный выход из цикла, потому что кончилась краска

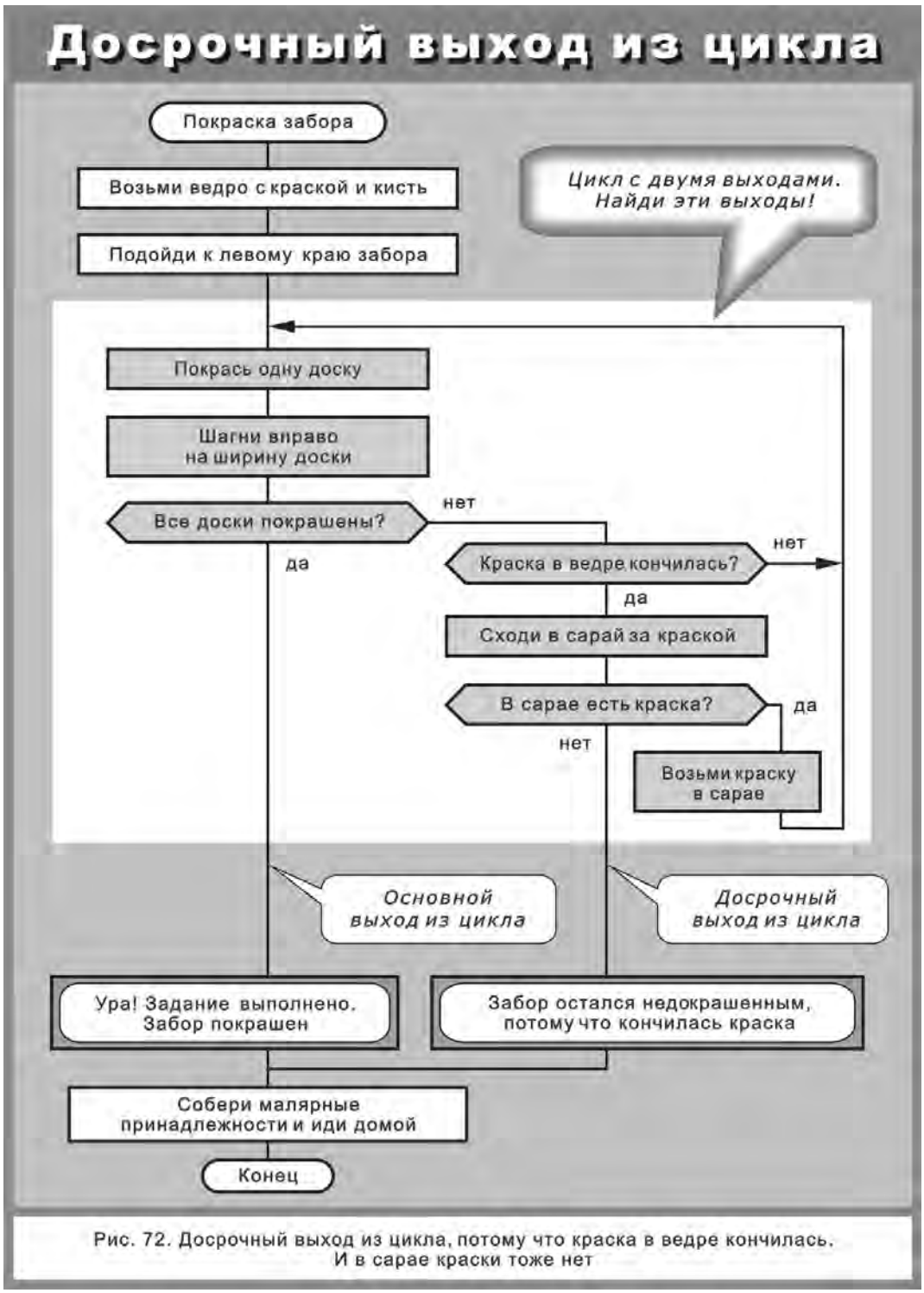


Рис. 72. Досрочный выход из цикла, потому что краска в ведре кончилась. И в сарае краски тоже нет

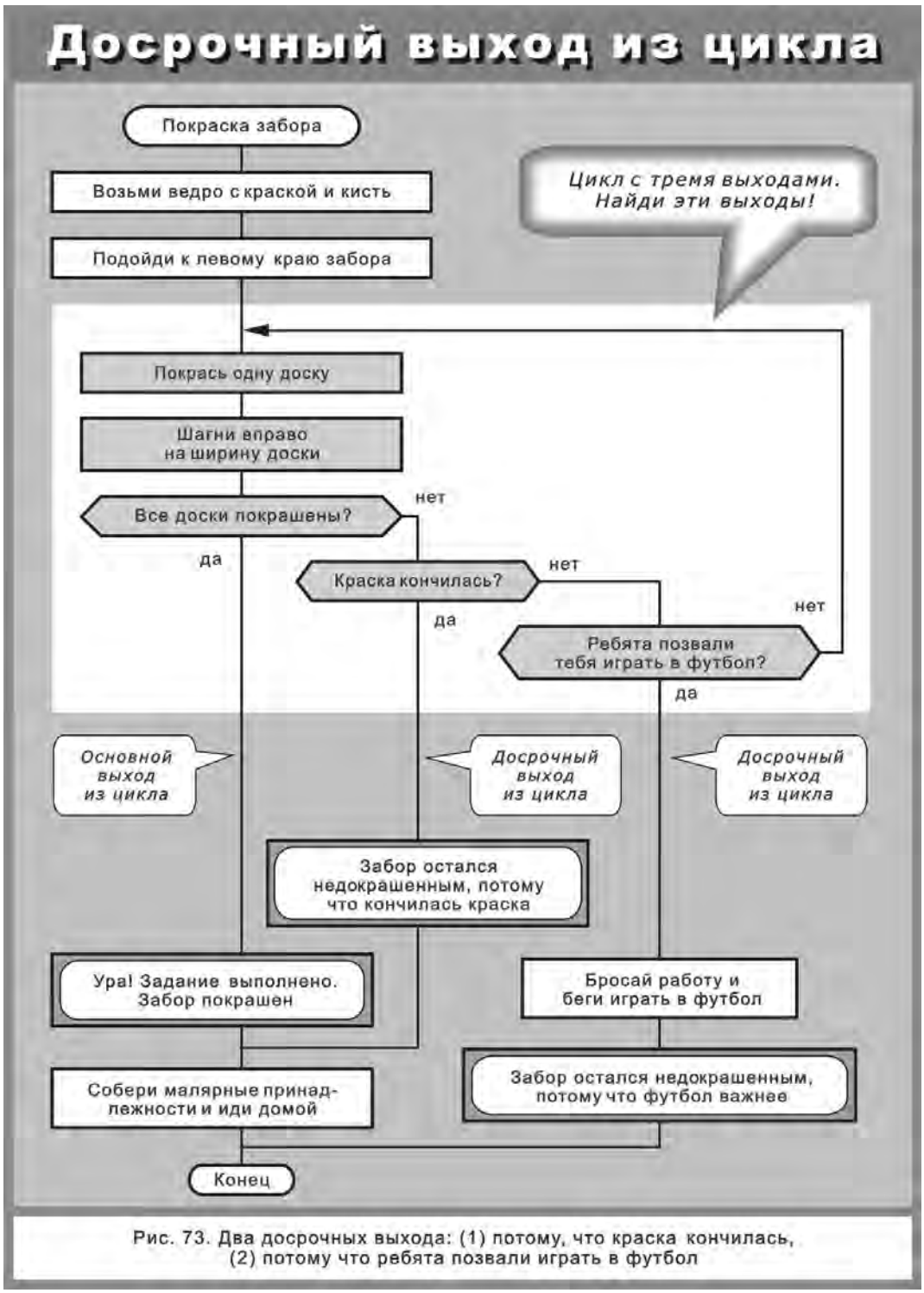


Рис. 73. Два досрочных выхода: (1) потому, что краска кончилась, (2) потому что ребята позвали играть в футбол

ЦИКЛ В ЦИКЛЕ

Рассмотрим графическую конструкцию «цикл внутри другого цикла». Пусть это будет цикл ДО внутри цикла ДО.

На рис. 68 есть команда «Покрась одну доску». Взглянем на нее «под микроскопом». Чтобы покрасить доску, надо сделать несколько операций: макнуть кисть в краску, сделать мазок, потом еще и еще — до тех пор пока вся доска не станет окрашенной. Эти действия можно изобразить в виде цикла (рис. 74).

Полученный цикл поместим на рис. 68 вместо команды «Покрась одну доску». Результат показан на рис. 75. Мы получили алгоритм, в котором есть конструкция «цикл в цикле».

Этот алгоритм работает так. Предположим, забор красит Том Сойер. Сначала Том выполняет две команды (рис. 75):

- Возьми ведро с краской и кисть
- Подойди к левому краю забора

Дальше нужно покрасить самую первую доску. Для этого Том исполняет команды, содержащиеся в цикле ДО (2):

- Обмакни кисть в краску
- Сделай мазок кистью по доске
- Покраска доски окончена?

Если не окончена, Том продолжает красить до тех пор, пока не будет выполнено условие окончания цикла ДО (2):

Покраска доски окончена?	=	да

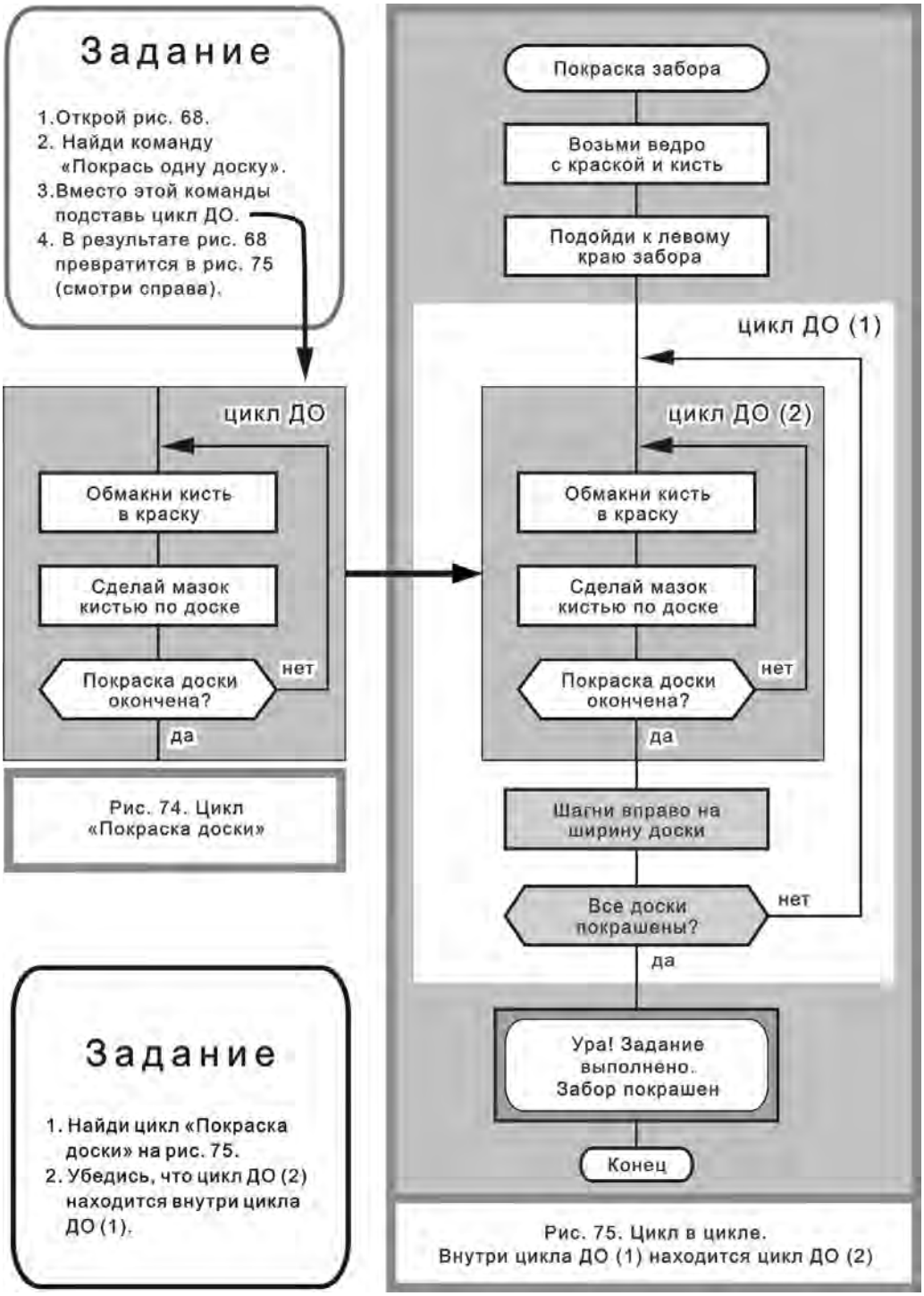
Рассмотрим числовой пример. Допустим, чтобы покрасить одну доску, нужно семь раз макнуть кисть в краску и сделать семь мазков. Значит цикл ДО (2) будет выполнен ровно семь раз.

После этого Том выполняет команды цикла ДО (1):

- Шагни вправо на ширину доски
- Все доски покрашены?

Если нет (не все доски покрашены), Том примется за следующую доску.

Давайте проследим маршрут. Из иконы «Все доски покрашены?» выходим через «нет» направо. По стрелке попадаем на вход цикла ДО (1). Затем Том начинает красить вторую доску. Он семь раз макнет кисть и сделает семь мазков. Разделавшись со второй доской, Том шагнет вправо на ширину доски и возьмется за третью доску. И так далее — пока весь забор не будет покрашен (рис. 75).



ЦИКЛ «ДО» ВНУТРИ ЦИКЛА «ПОКА»

В алгоритме на рис. 69 заменим икону «Съешь плюшку» на цикл ДО, состоящий из икон «Откуси кусок плюшки» и «Плюшка съедена?» (рис. 76). В итоге снова получим цикл в цикле. Цикл ПОКА, построенный с помощью иконы «В кошельке есть денежки?», является внешним. В нем «прячется» внутренний цикл ДО (рис. 76).

Поясним. При выполнении цикла ДО бегунок кружит по внутренней петле ВИКЛВ. За это время Карлсон откусывает один кусок плюшки, потом другой и так далее. Когда он покончит с первой плюшкой, будет выполнено условие окончания цикла ДО.

Плюшка съедена?	=	да

После этого маршрут бегунка меняется. Если в кошельке по-прежнему есть деньги, бегунок побежит по внешней петле ЖАБВГДЕЖ.

Предположим, в кошельке 50 монет, а Карлсон съедает плюшку за три приема. Это значит, что каждая команда цикла ПОКА будет выполнена 50 раз, а каждая команда цикла ДО — 150 раз. (Чтобы съесть 50 плюшек, откусив каждую 3 раза, нужно сделать 50 x 3 = 150 «откусываний»).

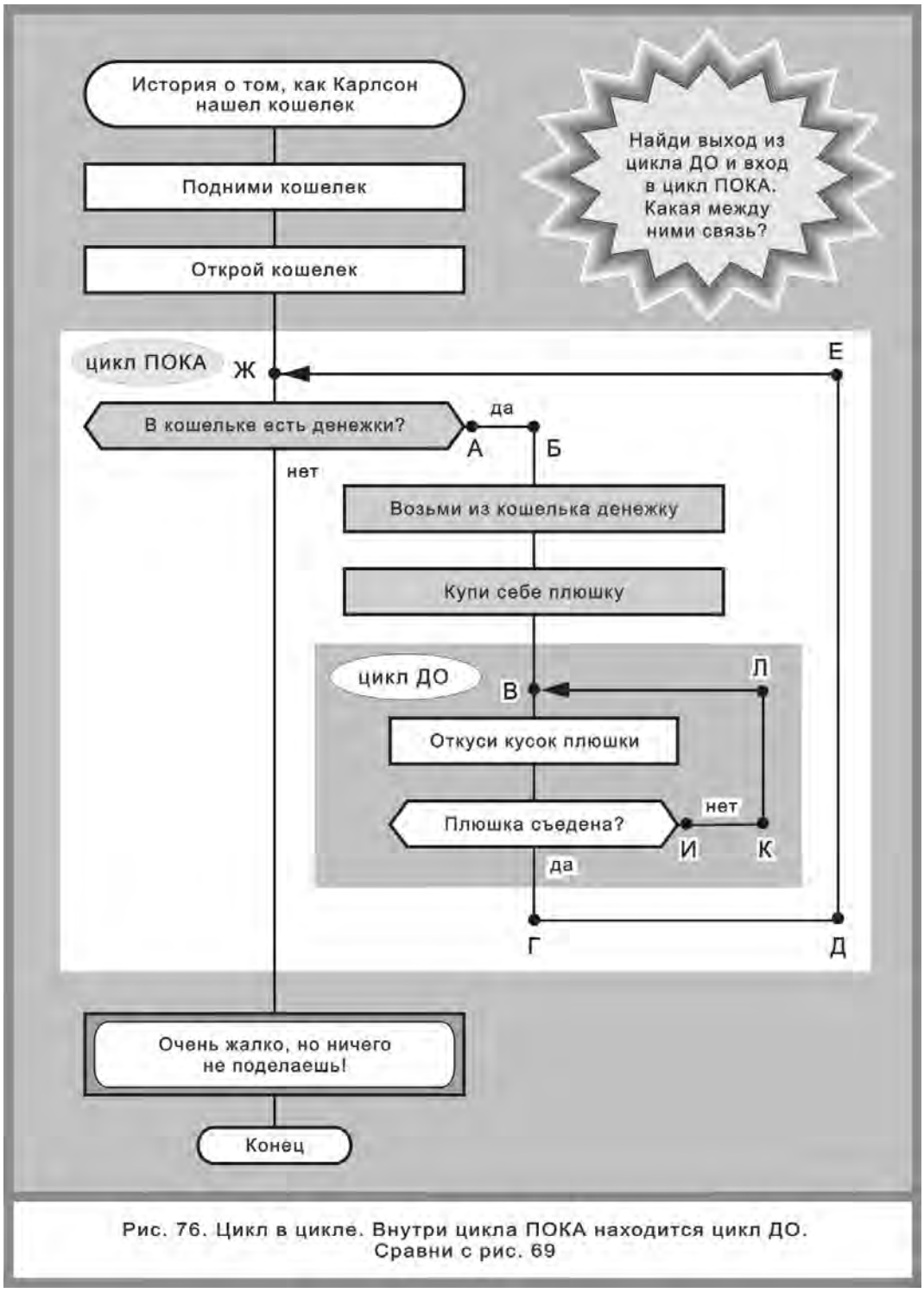


Рис. 76. Цикл в цикле. Внутри цикла ПОКА находится цикл ДО. Сравни с рис. 69

АБСТРАКТНАЯ КОНСТРУКЦИЯ «ЦИКЛ В ЦИКЛЕ»

На рис. 77 показано построение абстрактной (буквенной) конструкции «цикл в цикле».

На рис. 78 изображены четыре варианта конструкции «цикл в цикле». Этот рисунок очень важен — он задает эргономичные правила графического синтаксиса, используемые при создании конструкции «цикл в цикле».

Рис. 77. Как построить цикл в цикле

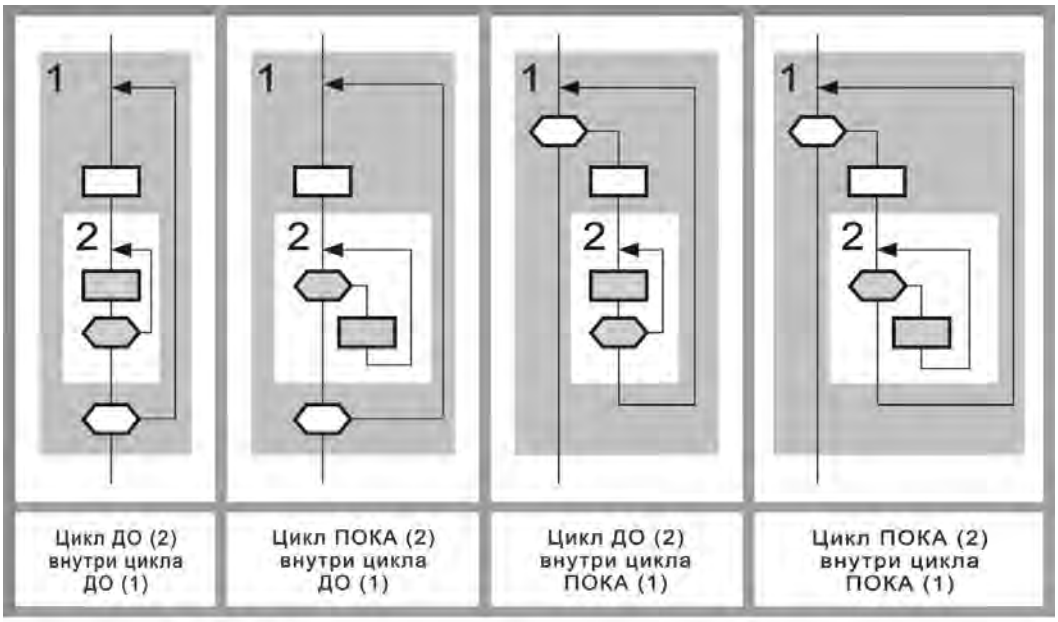


Рис. 78. Четыре варианта конструкции «цикл в цикле».

ОСОБЕННОСТИ ОБЫЧНОГО ЦИКЛА

Анализируя рисунки 65—78, можно сделать следующие замечания.

- Оператор «обычный цикл» имеет один вход и один или несколько выходов.
- Цикл с одним выходом представляет собой шампур-блок (вход и выход находятся на одной вертикали).

- Если цикл имеет более одного выхода, основной выход размещается на главной вертикали, дополнительные (досрочные) — правее ее.
- Шампур цикла проходит через икону «вопрос».
- Тело цикла ПОКА находится справа от шампура.
- Петля цикла находится правее главной вертикали и закручена против часовой стрелки.
- Икона «вопрос» задает *условие цикла*, которое распадается на две части: условие продолжения цикла и условие окончания цикла (рис. 69).
- *Условие продолжения цикла* соответствует правому выходу иконы «вопрос», *условие окончания цикла* — нижнему.
- Условие окончания цикла может помечаться как словом «нет», так и словом «да». То же самое относится и к условию продолжения цикла.

СТИХОТВОРЕНИЕ И АЛГОРИТМ

Наша ближайшая цель — познакомиться с понятием «переключающий цикл». Это цикл, внутри которого находится алгоритмическая конструкция «переключатель».

План рассказа таков. Сначала мы познакомимся с переключателем. Чтобы сделать материал приятным для чтения, воспользуемся стихотворением Генриха Гейне, в котором «спрятан» переключатель.

Вставай, слуга! Коня седлай!
Чрез рощи и поля
Скачи скорее ко двору
Дункана-короля.

Зайди в конюшню там и жди.
И, если кто войдет,
Спроси: «Которую Дункан
Дочь замуж выдает?»

Коль чернобровую — лети
Во весь опор назад.
Коль ту, что с русою косой,
Спешить не надо, брат.

Тогда ступай на рынок ты,
Купи веревку там,
Вернися шагом и молчи —
Я догадаюсь сам.

Это стихотворение легко превращается в алгоритм. (Те, кто любит забегать вперед, могут взглянуть на рис. 81).

Мы же начнем по порядку.

ПЕРЕКЛЮЧАТЕЛЬ

Предположим, в алгоритме нужно организовать разветвление на несколько направлений. Задачу можно решить двумя способами: с помощью иконы «вопрос» (рис. 79) и с помощью переключателя (рис. 80).

Переключатель — составной графический оператор (рис. 5, макроикона 3), имеющий один вход и один выход, содержащий одну икону «выбор» и несколько (две и более) икон «вариант» (рис. 4, иконы И5, И6).

Внутри иконы «выбор» делается надпись, обычно в утвердительной форме, которая обозначает вопрос, имеющий строго определенное число ответов (два и более). Ответы записываются в иконах «вариант». Таким образом, число вариантов равно числу ответов.

Говоря формально, в иконе «выбор» записывается переменная, в иконах «вариант» — ее значения.

Пример 1. На рис. 81 в иконе «выбор» записан вопрос: «Какую дочь Дункан выдает замуж?». На этот вопрос есть два ответа, записанные в иконах «вариант»:

- чернобровую,
- с русою косой.

Пример 2. На рис. 80 картина иная. На первый, взгляд там нет вопроса. Однако на самом деле вопрос есть, правда неявный. Чтобы убедиться, слово «светофор» прочитаем так: *какой сигнал светофора сейчас горит?* Вот три ответа:

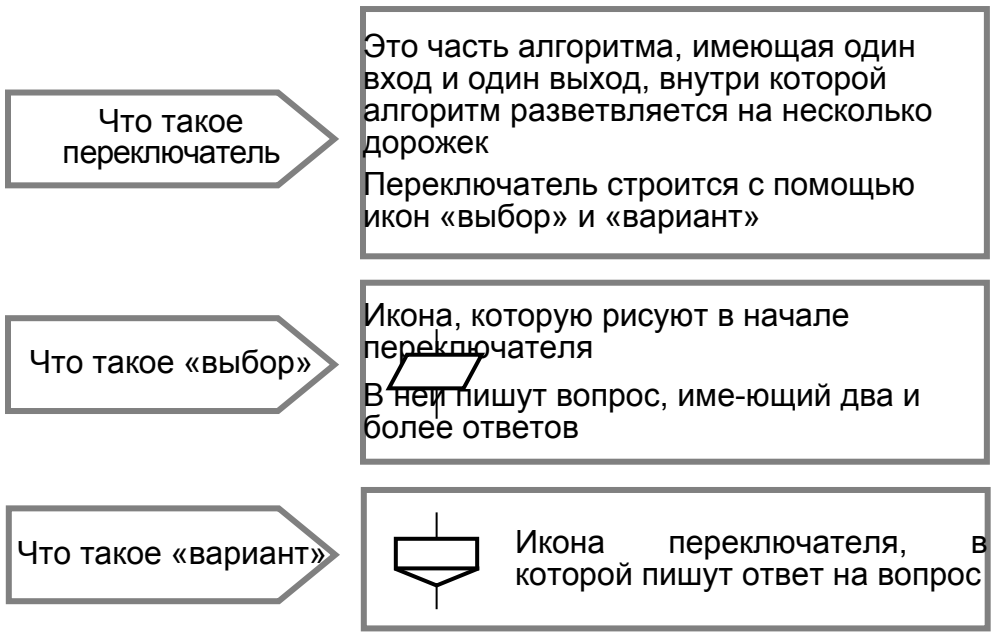
- зеленый,
- желтый,
- красный.

Формально это означает: переменная «Светофор» принимает три значения: зеленый, желтый, красный.

КАК РАБОТАЕТ ПЕРЕКЛЮЧАТЕЛЬ?

Ответ всегда начинается со слова «если».

- Если Дункан выдает замуж чернобровую дочь — лети во весь опор назад.
- Если Дункан выдает замуж дочь с русою косой — ступай на рынок, купи веревку, вернись шагом и молчи (рис. 81).
- Если светофор зеленый — жми на газ.
- Если светофор желтый — притормози.
- Если светофор красный — стой (рис. 80).



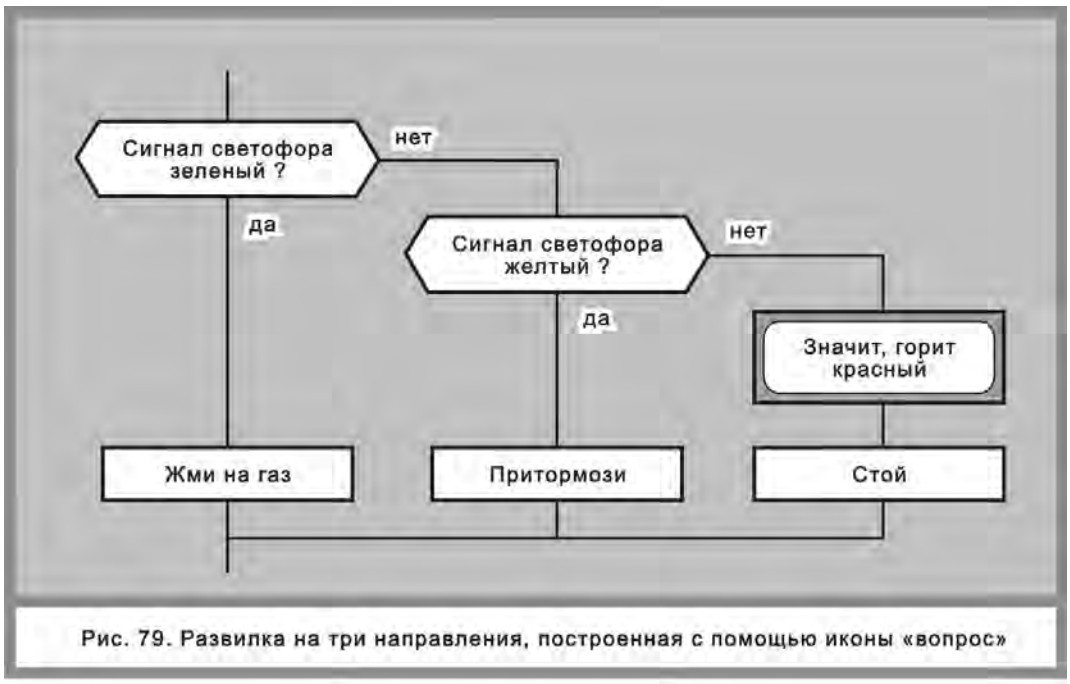


Рис. 79. Развилка на три направления, построенная с помощью иконы «вопрос»

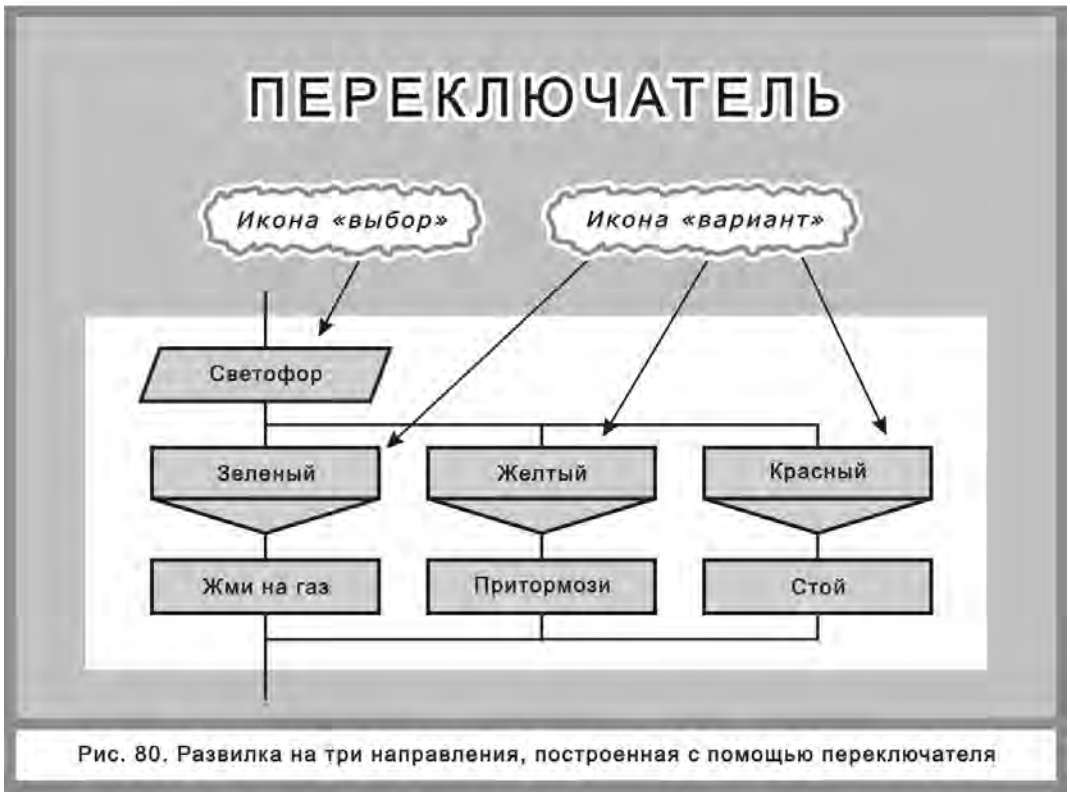


Рис. 80. Развилка на три направления, построенная с помощью переключателя

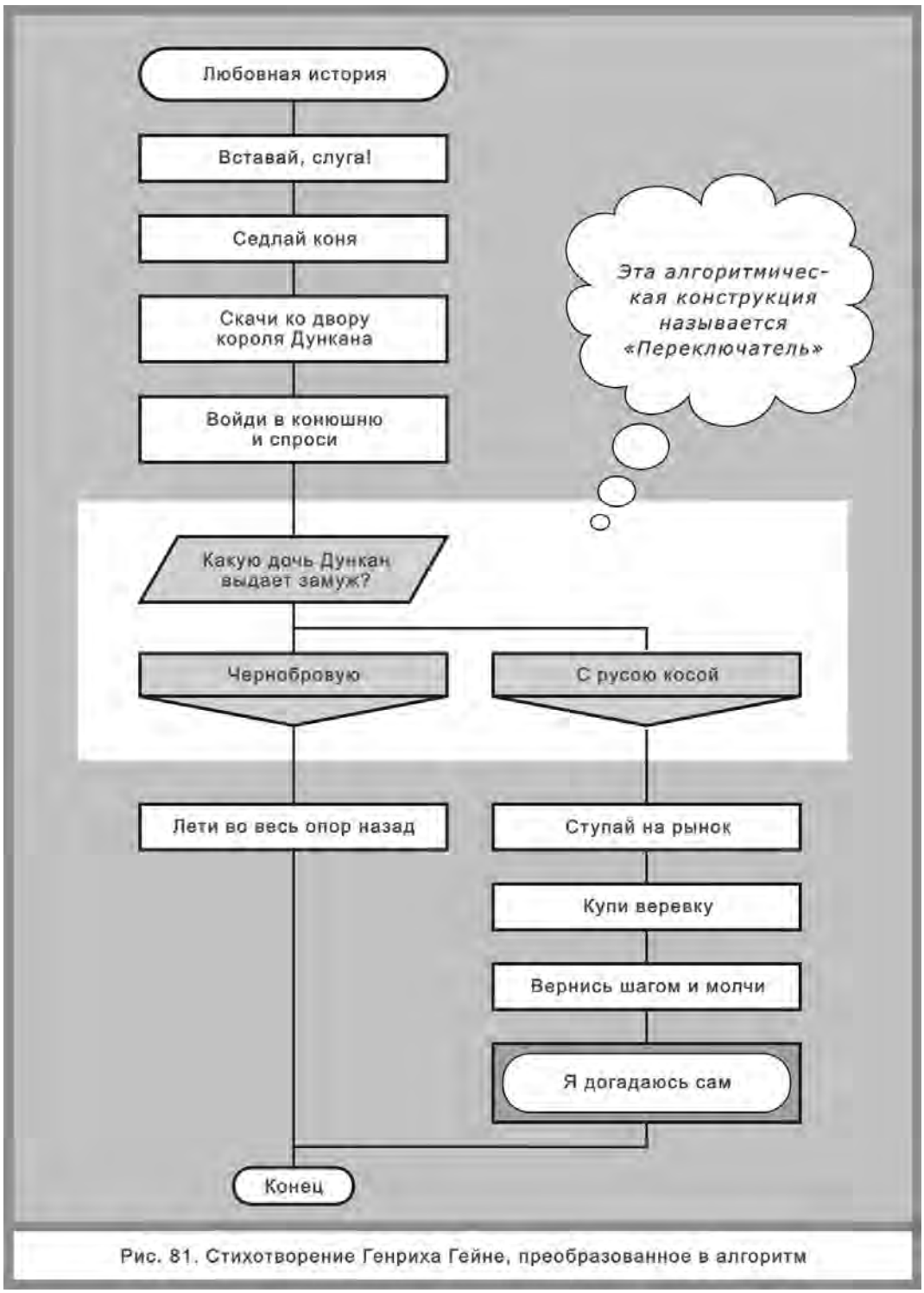


Рис. 81. Стихотворение Генриха Гейне, преобразованное в алгоритм

ПЕРЕКЛЮЧАЮЩИЙ ЦИКЛ

Переключатель позволяет создать особый тип цикла — *переключающий цикл* (рис. 5, макроикона 5). Для этого нужно оторвать выход правой ветви переключателя, загнуть его вверх и присоединить стрелку в нужное место (рис. 82).

На рис. 83 изображен цикл с переключателем, однако это не переключающий цикл, а обычный.

Как их отличить? В первом случае переключатель имеет два выхода, во втором — только один. Есть еще одно отличие. Если вверх загибается выход иконы «вопрос» — это обычный цикл (ДО, ПОКА или гибридный). А

если кверху идет выход переключателя — перед нами переключающий цикл.

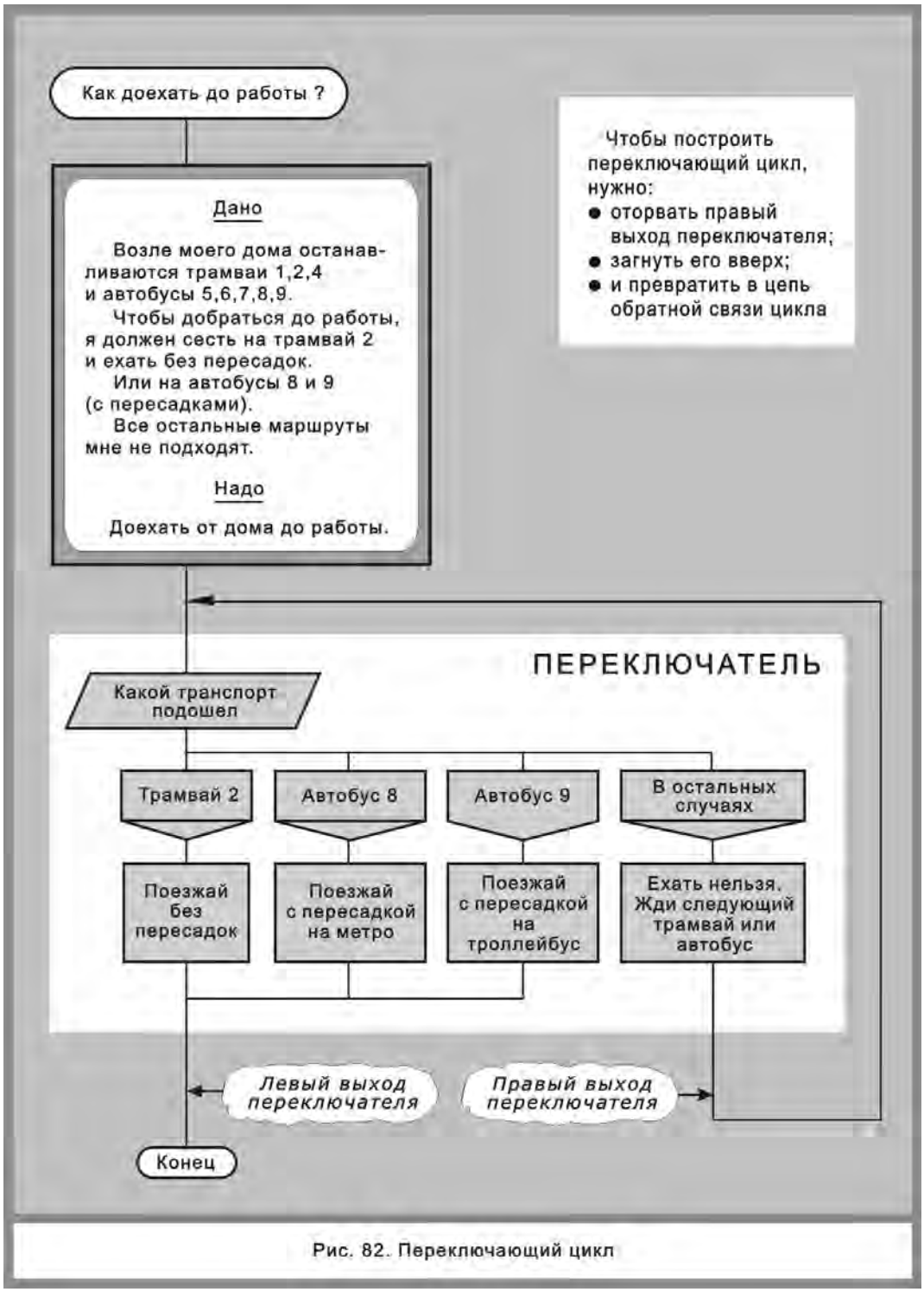


Рис. 82. Переключающий цикл

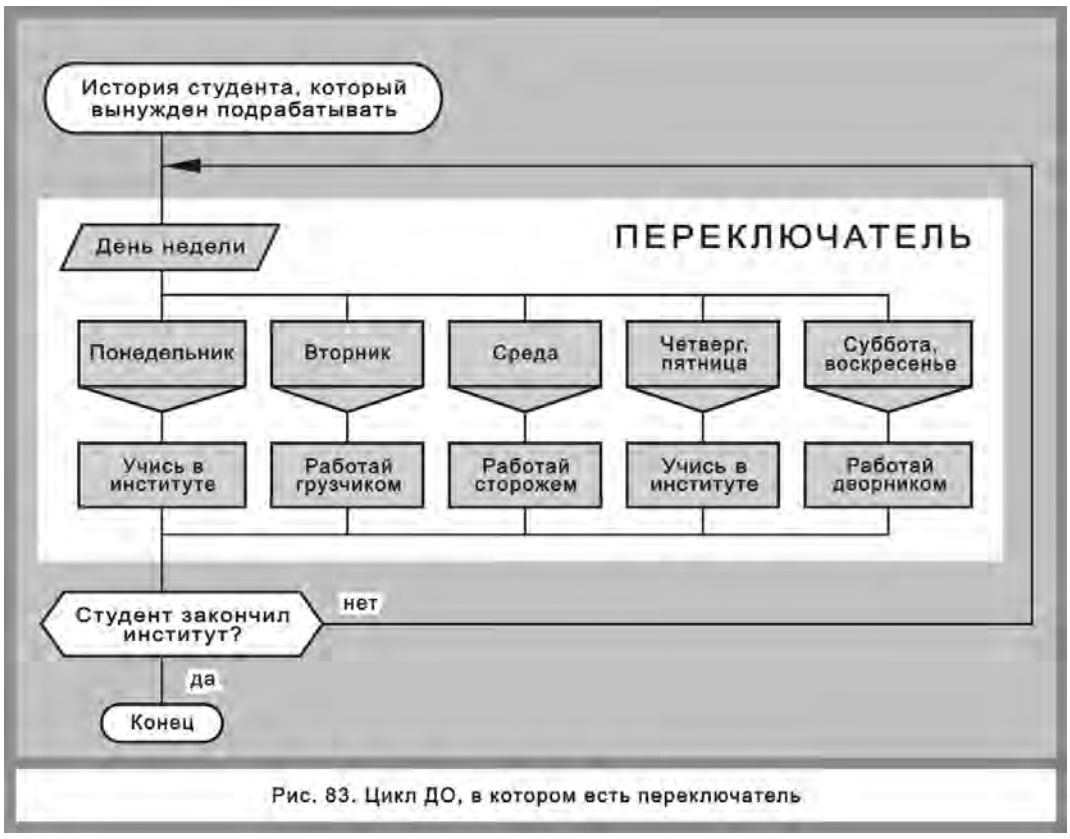


Рис. 83. Цикл ДО, в котором есть переключатель

ЦИКЛ ДЛЯ

На рис. 84 и 85 показаны два варианта решения простой математической задачи. В первом случае используется цикл ДО, во втором — цикл ДЛЯ.

Цикл ДЛЯ — составной графический оператор (рис. 5, макроикона 6). Он содержит иконы «начало цикла ДЛЯ» и «конец цикла ДЛЯ» (рис. 4, иконы И13, И14), между которыми располагаются одна или несколько других икон.

Внутри иконы «начало цикла ДЛЯ» указываются переменная цикла, ее начальное и конечное значения и шаг. Порядок записи этих величин определяется выбранным вариантом текстового синтаксиса. На рис. 85 изображен вариант, по умолчанию принимающий, что шаг равен 1.

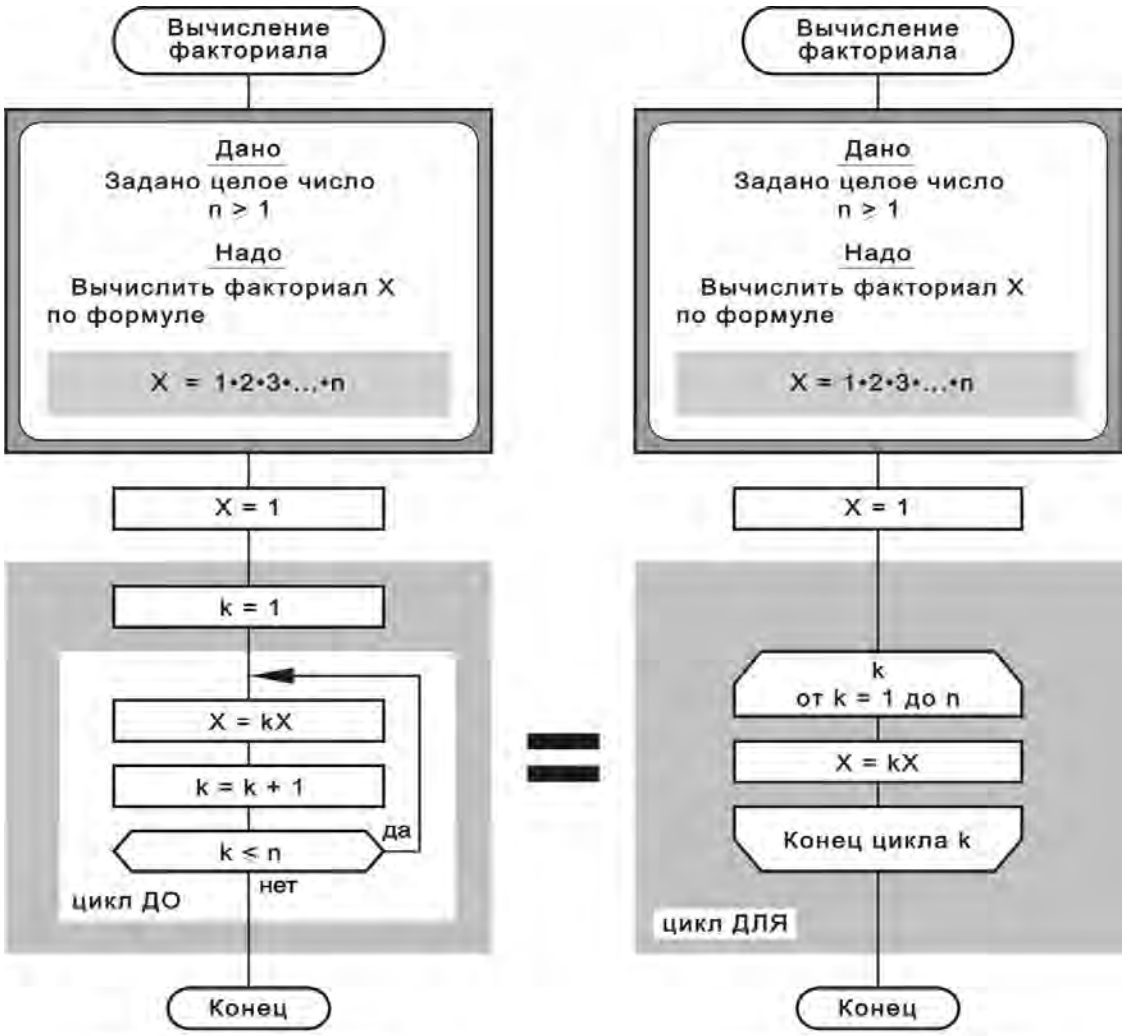


Рис. 84. Как вычислить факториал X = n! с помощью цикла ДО?

Рис. 85. Как вычислить факториал X = n! с помощью цикла ДЛЯ?

ВЕТОЧНЫЙ ЦИКЛ

Циклы, описанные выше, могут использоваться как в примитиве, так и в силуэте. В этом параграфе речь пойдет о веточном цикле, который встречается только в силуэте.

Веточный цикл — это повторное исполнение одной и той же ветки. Чтобы построить веточный цикл, нужно написать в иконе «адрес» название данной ветки (или более левой ветки).

На рис. 68 изображен циклический алгоритм «Покраска забора». Можно ли нарисовать его в виде силуэта? Да, можно. Результат показан на рис. 86.

Во второй ветке слово «Покраска» встречается дважды: вверху и внизу. Это значит, что перед нами *веточный* цикл. Бегунок, доехав до адреса «Покраска», тут же вернется к началу ветки и будет «утюжить» ее вновь и вновь.

Циклическое движение по ветке «Покраска» будет продолжаться, пока выполняется условие продолжения цикла:

Все доски покрашены?	=	нет

Когда Том Сойер кончит красить забор, появится условие окончания цикла:

Все доски покрашены?	=	да

После этого бегунок, проходя через икону «вопрос», повернет направо и через адрес «Завершение» попадет в ветку «Завершение». На этом алгоритм закончит работу (рис. 86).

Другой пример веточного цикла показан на рис. 87. Как и любой цикл, веточный цикл может иметь основной и досрочный выходы (рис. 88 и 89).

Веточный цикл можно использовать в сочетании с циклами ПОКА и ДО. Например, на рис. 90 изображена конструкция «цикл в цикле». Внутри веточного цикла «Покраска» находится цикл ДО, содержащий иконы:

- Обмакни кисть в краску
- Сделай мазок кистью по доске
- Покраска доски закончена?

Вопрос. Зачем нужны маленькие черные треугольники в иконах «Покраска» и «Покупка плюшек» (рис. 86—90)?

Ответ. Это флажки. Они привлекают внимание и позволяют легко заметить веточный цикл даже при беглом взгляде.

Рис. 86. Силуэт с веточным циклом. Сравни с рис. 68

Рис. 87. Силуэт с веточным циклом. Сравни с рис. 69

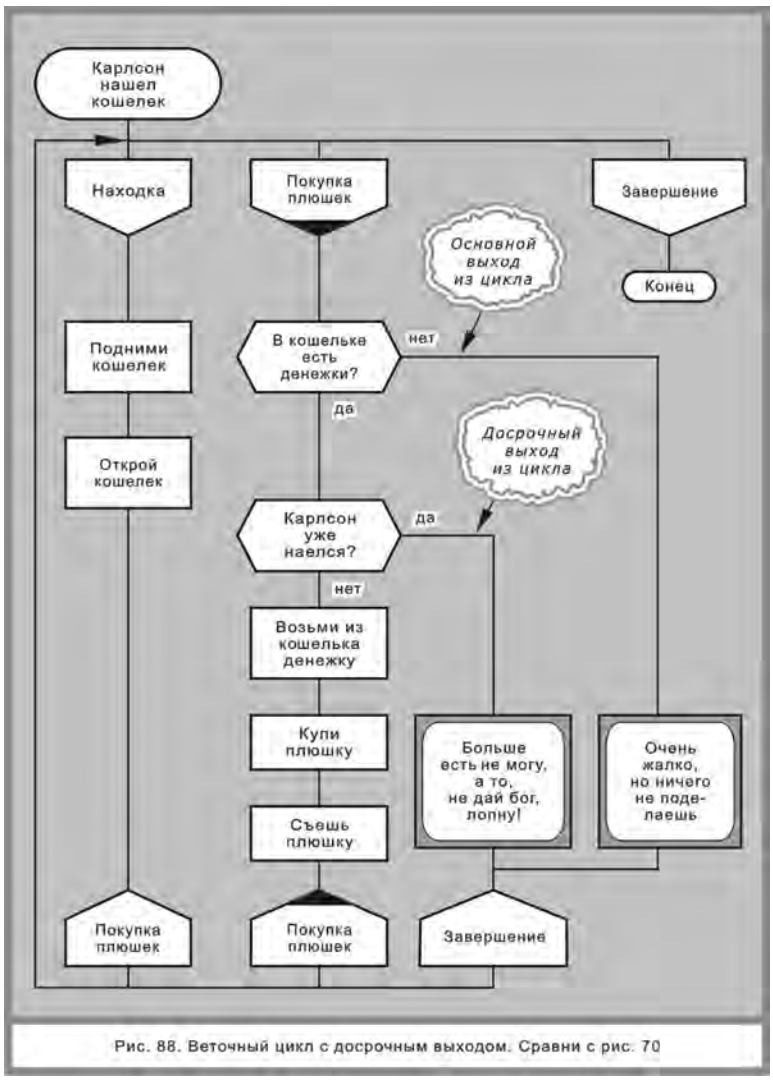


Рис. 88. Веточный цикл с досрочным выходом. Сравни с рис. 70

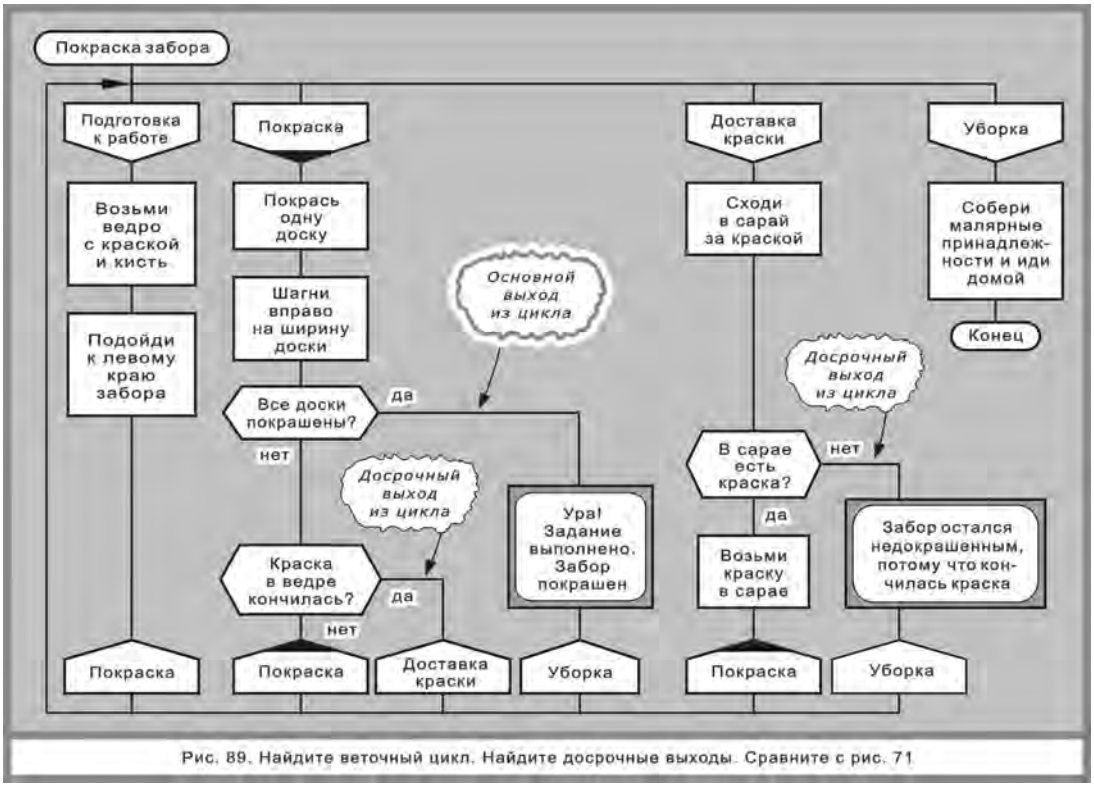


Рис. 89. Найдите веточный цикл. Найдите досрочные выходы. Сравните с рис. 71

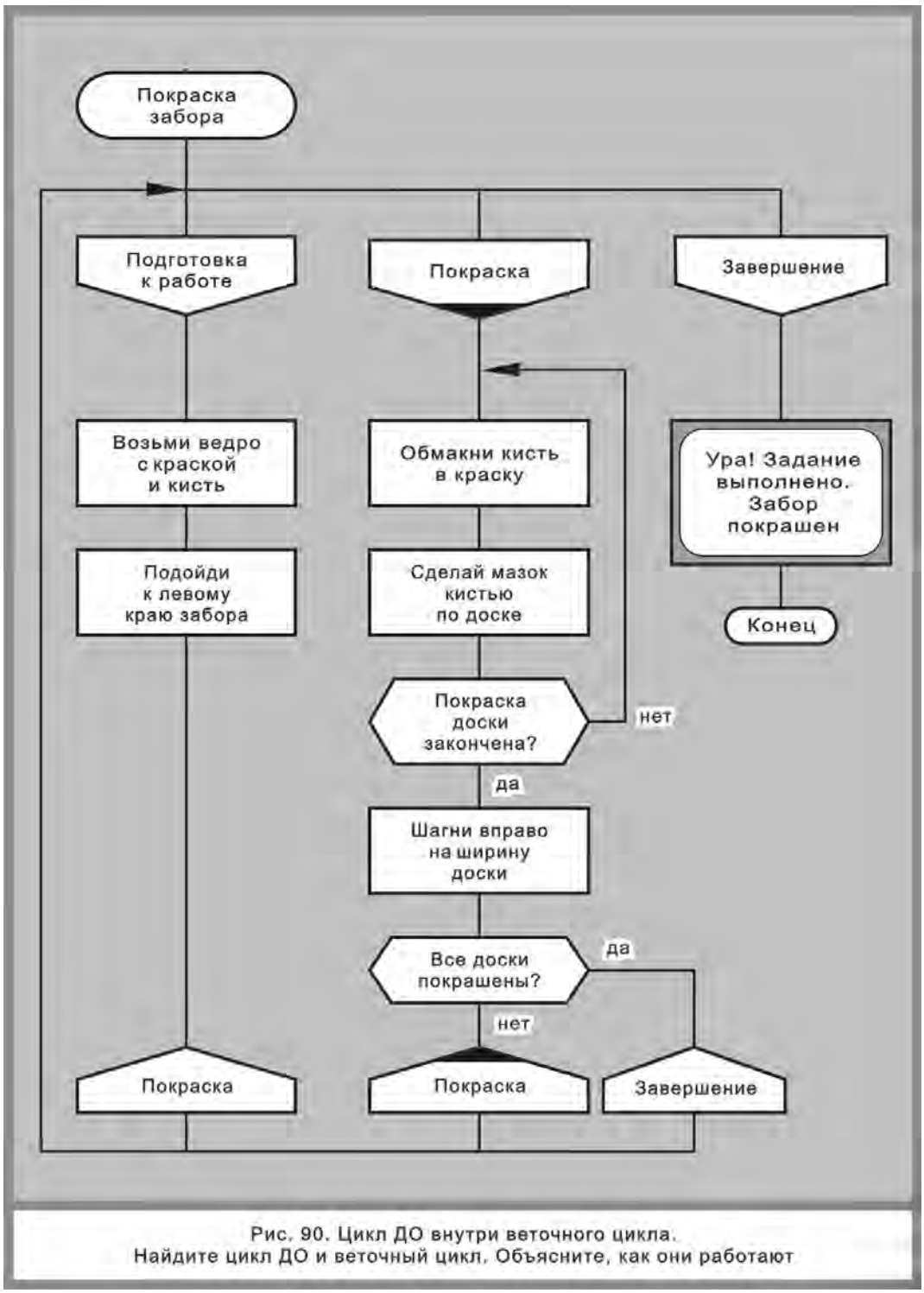


Рис. 90. Цикл ДО внутри веточного цикла.
Найдите цикл ДО и веточный цикл. Объясните, как они работают

ПРЕЖНИЙ ПОДХОД ИСЧЕРПАЛ СЕБЯ

Описание циклов с помощью текста следует признать устаревшим. Визуальные циклы гораздо удобнее и эффективнее.

Существующая «текстовая» традиция возводит в ранг закона «насилие» над мыслью программиста, заставляя его, в частности, вводить никому не нужные и нередко опасные инверсии логических условий.

Например, язык Си требует: «выход из цикла производится только по *false*». Данное требование заставляет «искалечить» пример на рис. 68 и заменить утвердительный вопрос «Все доски покрашены?» на отрицательный «Доски не покрашены?».

Это явно противоречит рекомендациям эргономики, согласно которым отрицательные вопросы нежелательны, ибо провоцируют ошибки [2].

ВЫВОДЫ

1. В различных текстовых языках при описании циклов применяются разные наборы ключевых слов, имеющих к тому же разную семантику. Неразбериху усугубляют отличия в логике окончания цикла. Например, в языке Си для циклов *while* и *do-while* условие окончания цикла соответствует значению *false* или 0, условие продолжения — значению *true* или 1. В языке Паскаль картина иная: в цикле *while-do* выход из цикла соответствует значению *false*, а в цикле *repeat-until* по каким-то загадочным причинам применяется диаметрально противоположный принцип: выход из цикла производится, когда логическое выражение принимает значение *true*. Все эти путаные правила программист обязан знать и неукоснительно выполнять.
2. Отсутствие унификации ключевых слов и разнобой в определении условий выхода из цикла является серьезным недостатком. Программисты вынуждены зубрить ключевые слова и значения условий, причем освоение каждого следующего языка требует новой зубрежки.
3. С точки зрения визуального программирования, указанные трудности являются надуманными и легко устраняются. Надо лишь отказаться от сложившихся привычек и устаревших стереотипов мышления, связанных с текстовым программированием. Визуализация качественно меняет ситуацию, поскольку текст больше не является единственным носителем информации.
4. Визуальные образы уменьшают нагрузку на память программиста. Ликвидируют ошибки, вызванные неправильным пониманием семантики ключевых слов. Отменяют ненужные ограничения. Предоставляют пользователю богатую палитру выразительных средств. И в конечном итоге обеспечивают более высокую понимаемость алгоритмов и программ.
5. Визуализация циклов — весьма полезный инструмент, так как сложные вложенные циклы со многими выходами часто бывают источником трудных ошибок. Многие из них возникают из-за путаницы, связанной с устаревшей привычкой описывать циклы словами. Сегодня никто не пытается заменить конструкторские и строительные чертежи словесными описаниями. По мнению автора, текстовая форма записи циклов во многих случаях является таким же анахронизмом, как словесное описание механического чертежа или электрической схемы.

Глава 4

ВИЗУАЛИЗАЦИЯ ЛОГИЧЕСКИХ ФОРМУЛ

Существует форма представления информации наглядная, броская, понятная всем с детства. Такой формой является графика.

Валерий Венда [1]

ЩЕПОТКА МАТЕМАТИКИ

В большинстве случаев математики записывают алгоритмы в виде текста (математического текста). В последнее время бурно развивается новое направление — графическое представление алгоритмов и программ. Однако известные чертежи алгоритмов обладают важным недостатком — они слишком трудны для понимания. Язык ДРАКОН выгодно отличается от конкурентов тем, что удовлетворяет критерию сверхвысокого понимания и облегчает работу с алгоритмами.

Тем не менее, есть одно «но». Читатель-математик вправе усомниться: обеспечивают ли дракон-схемы необходимую математическую строгость? Является ли графическая запись алгоритмов столь же точной, как и текстовая математическая запись?

На этот вопрос следует дать утвердительный ответ. Дракон-схемы нисколько не уступают тексту. Чтобы пояснить суть дела, рассмотрим частный случай — алгоритмическую конструкцию **Если–то–иначе** и равносильную ей дракон-схему (рис. 91). Из рисунка видно, что обе формы алгоритма (текст и схема) математически равносильны, т.е. выражают в точности одинаковое содержание.

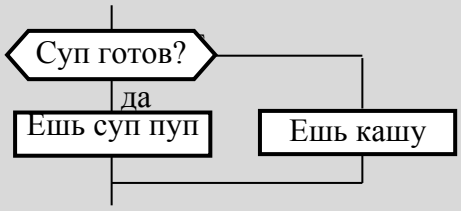
Текст	**Дракон-схема**
Если (Суп готов? = да) **то** Ешь суп **иначе** Ешь кашу	

Рис. 91. Содержательный алгоритм.
(Текст и схема математически равносильны)

Введем обозначения:

- A = Суп готов?
- B = Ешь суп
- C = Ешь кашу

Подставив новые обозначения в рис. 91, получим «буквенный» алгоритм на рис. 92.

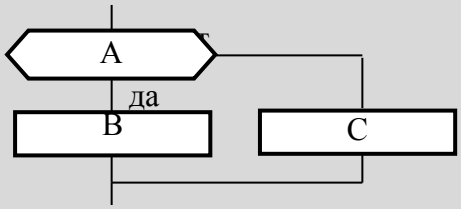
Текст	Дракон-схема
Если A = да **то** B **иначе** C	

Рис. 92. Буквенный алгоритм.
(Текст и схема математически равносильны)

Применим рокировку к дракон-схеме. Для этого в правой части рис. 92 поменяем местами:

- иконы B и C;
- слова «да» и «нет».

Аналогичные изменения произведем и в левой части рис. 92. Результат представлен на рис. 93.

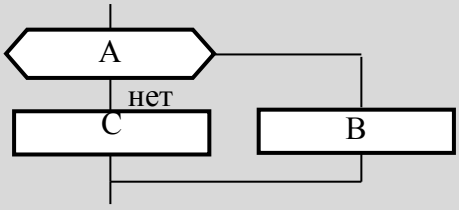
Текст	Дракон-схема
Если A = нет **то** C **иначе** B	

Рис. 93. Буквенный алгоритм после рокировки.
(Текст и схема математически равносильны)

Заметим, что все четыре формы записи алгоритма, показанные на рис. 92 и 93, равноценны. Можно сказать и по-другому: один и тот же алгоритм записан четырьмя разными способами.

Сделаем следующий шаг рассуждений. Уберем из рис. 93 символы A, B, C, да, нет, =. В итоге получим рис. 94.

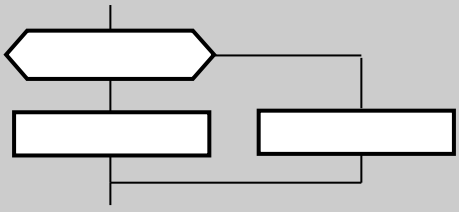
Текст	Дракон-схема
Если **то** **иначе**	

Рис. 94. Абстрактный алгоритм.
(Текст и схема математически равносильны)

Абстрактная дракон-схема — схема, из которой удалены все текстовые надписи. На рис. 94 справа показан частный случай такой схемы — *абстрактная развилка.* Ее математический смысл выражается формулой **Если то иначе** (**If Then Else**).

Сделанные пояснения убедительно подтверждают, что графический узор дракон-схемы является четко определенным математическим объектом.

Сочетание математической строгости и богатых возможностей когнитивно-эргономического подхода позволили превратить язык ДРАКОН в чрезвычайно удобный инструмент, обеспечивающий беспрецедентный рост производительности умственного труда при решении широкого класса задач.

ВИЗУАЛИЗАЦИЯ ФУНКЦИИ «И»

— Где можно купить щенка?

— В нашем городке они продаются на рынке, но сегодня рынок закрыт. К тому же щенков продают не каждый день. Щенки довольно дорогие и какие-то невзрачные — не знаю, понравятся ли они вам.

Упростим ситуацию. Будем считать, что покупка щенка возможна в том и только в том случае, когда выполняются три условия (рис. 95):

- у покупателя деньги есть (*Q*);
- щенки есть в продаже (*R*);
- щенок понравился (*S*).

В итоге получаем логическую функцию

$$X = Q \text{ и } R \text{ и } S$$

где *X* означает «Можно купить щенка» (рис. 95).

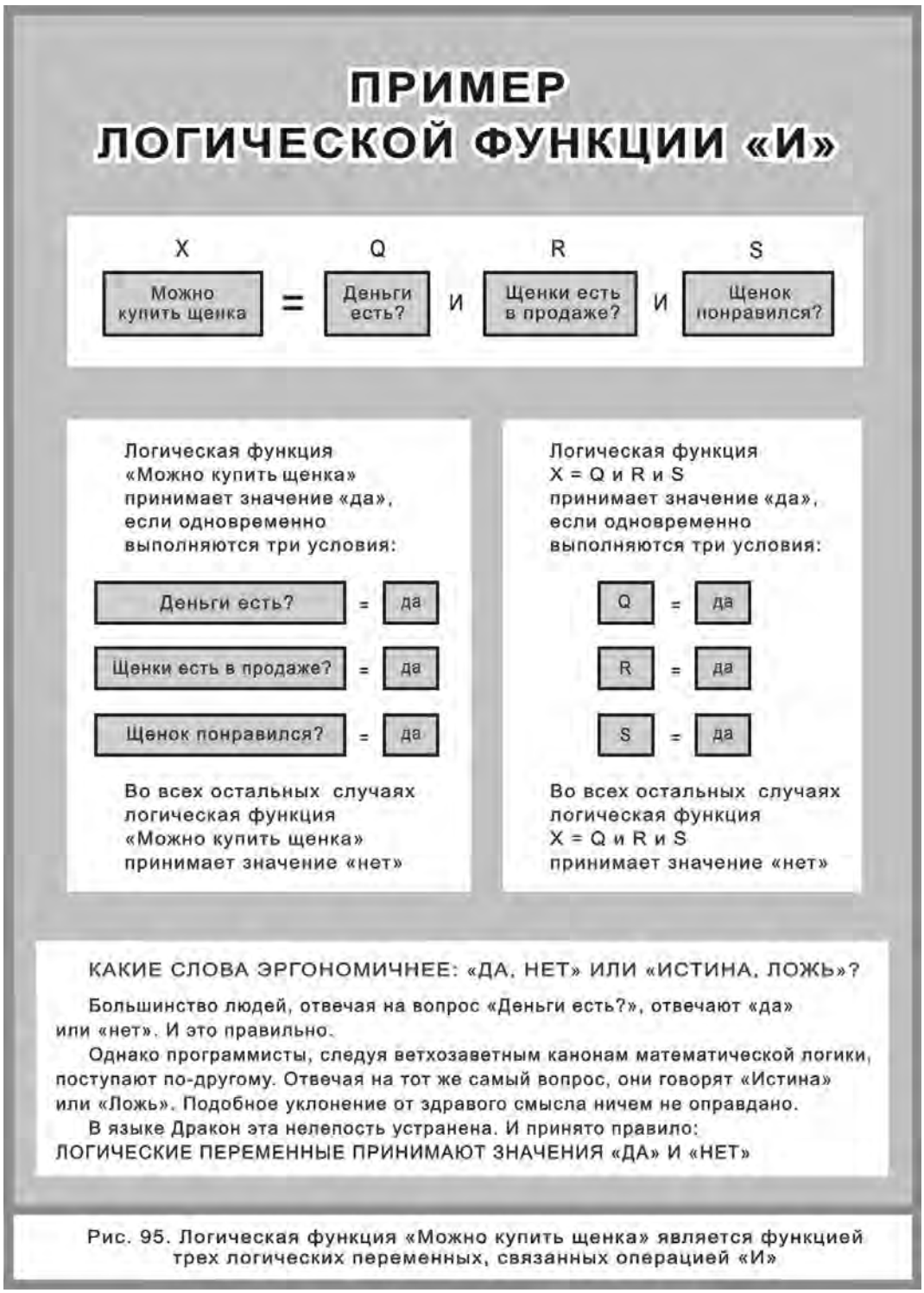


Рис. 95. Логическая функция «Можно купить щенка» является функцией трех логических переменных, связанных операцией «И»

БУДЬТЕ ПРОЩЕ — К ВАМ ПОТЯНУТСЯ!

В традиционных языках программирования значениями логических переменных считаются пары (ИСТИНА, ЛОЖЬ) или (1, 0). С эргономической точки зрения, такой подход нельзя признать удачным.

В самом деле, использование «шибко мудреных» слов ИСТИНА и ЛОЖЬ или таинственных цифр 1 и 0 в примере о щенках (как и в любом другом конкретном примере) является надуманным и дезориентирующим. Оно не содействует пониманию сути дела, а наоборот, затемняет картину.

Чтобы поправить дело, надо убрать никому не нужные сложности.

В качестве значений логических переменных и логических функций гораздо лучше выбрать простые и ясные слова «да» и «нет». Смысл этих

слов не требует пояснений. Он понятен любому. Даже ребенок знает, что такое «да» и «нет».

Поэтому в языке Дракон используются ключевые слова «да» и «нет». А логические функции, переменные и выражения рассматриваются как да-нетные вопросы или утверждения.

ЛОГИЧЕСКАЯ ФУНКЦИЯ «И»

Логическая функция И — это функция, которая принимает значение «да», если все логические переменные имеют значение «да».

В остальных случаях функция приобретает значение «нет» (рис. 95).

АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ФУНКЦИЮ «И»

На рис. 96 приведены два примера алгоритмов. Слева описан рассказ о покупке щенков. Справа тот же самый алгоритм представлен в абстрактной математической форме.

Попытайтесь в правом алгоритме найти и выделить логическую функцию «И». Полученный результат пригодится при анализе рис. 97.

Рис. 96. Примеры алгоритмов с операцией «И»

Рис. 97. Рисуйте дракон-схему «И», как показано справа.
Избегайте нерекомендуемых схем

ДВА СПОСОБА ЗАПИСИ ФУНКЦИИ «И»

На языке ДРАКОН существуют два способа записи функции И: *текстовый* и *визуальный*.

В первом случае используют одну икону «вопрос», внутри которой пишут логическое выражение, состоящее из логических переменных, соединенных знаками логической операции И (рис. 97 слева).

В другом случае на одной вертикали рисуют *N* икон «вопрос»,

где *N* — число логических переменных, причем в каждой иконе записывают одну логическую переменную (рис. 97 справа).

КАКОЙ СПОСОБ ЛУЧШЕ: ТЕКСТОВЫЙ ИЛИ ВИЗУАЛЬНЫЙ?

Визуальная формула на рис. 97 показывает, что оба способа эквивалентны.

Для практического использования рекомендуется визуальный способ, так как он более нагляден и позволяет быстрее найти ошибку в сложном алгоритме. Следует подчеркнуть, что текстовый способ не является запрещенным, но пользоваться им следует с осторожностью и лишь в тех случаях, когда пользователь убежден в своих способностях гарантировать отсутствие ошибок.

Опыт показывает, что большинство людей выбирает визуальный способ как более легкий. Однако подготовленные специалисты, знакомые с основами математической логики, иногда предпочитают текстовый метод. Таким людям можно посоветовать освоить оба метода.

СРАВНЕНИЕ МАТЕМАТИЧЕСКОЙ ФОРМУЛЫ И ДРАКОН-СХЕМЫ

На рис. 98 приведена математическая формула и равносильная ей дракон-схема. Какую из них следует предпочесть? Какая является более эргономичной?

Слева представлена традиционная формула, понятная далеко не всем.

Формула справа, написанная на языке ДРАКОН, намного легче для понимания. Она становится еще более наглядной, если заменить абстрактные буквы Q, R, S, B на конкретные производственные понятия. Например:

Q = норма подачи топлива
R = норма зажигания
S = норма электропитания;
B = включить двигатель.

Два объекта (текстовый слева и графический справа на рис. 98) математически равносильны. Это значит, что графическая дракон-схема является МАТЕМАТИЧЕСКОЙ ФОРМУЛОЙ.

Отсюда вытекает, что математические формулы бывают не только текстовые, но и графические.

И последнее. Анализируя формулу слева на рис. 98, обычно приходится вникать в сложные подробности, например:

X = [Q & R & S = да] = [(Q = да) & (R = да) & (S = да)]

Дракон-схема (рис. 98, справа) хороша тем, что полностью избавляет читателя от подобной ненужной работы.

Рис. 98. Математическая формула и дракон-схема

ВИЗУАЛИЗАЦИЯ ФУНКЦИИ «ИЛИ»

Пете не повезло — он заболел. Что с ним случилось?

На этот счет может быть тьма ответов. Но медициной мы заниматься не будем. Ограничимся логикой.

Для простоты будем считать, что Петя болен, если выполняется хотя бы одно из трех условий (рис. 99):

- у Пети грипп (L);
- у Пети ангина (M);
- у Пети ушиб (N).

В итоге получаем логическую функцию

$$X = L \text{ или } M \text{ или } N$$

где *X* означает «Петя заболел» (рис. 99).

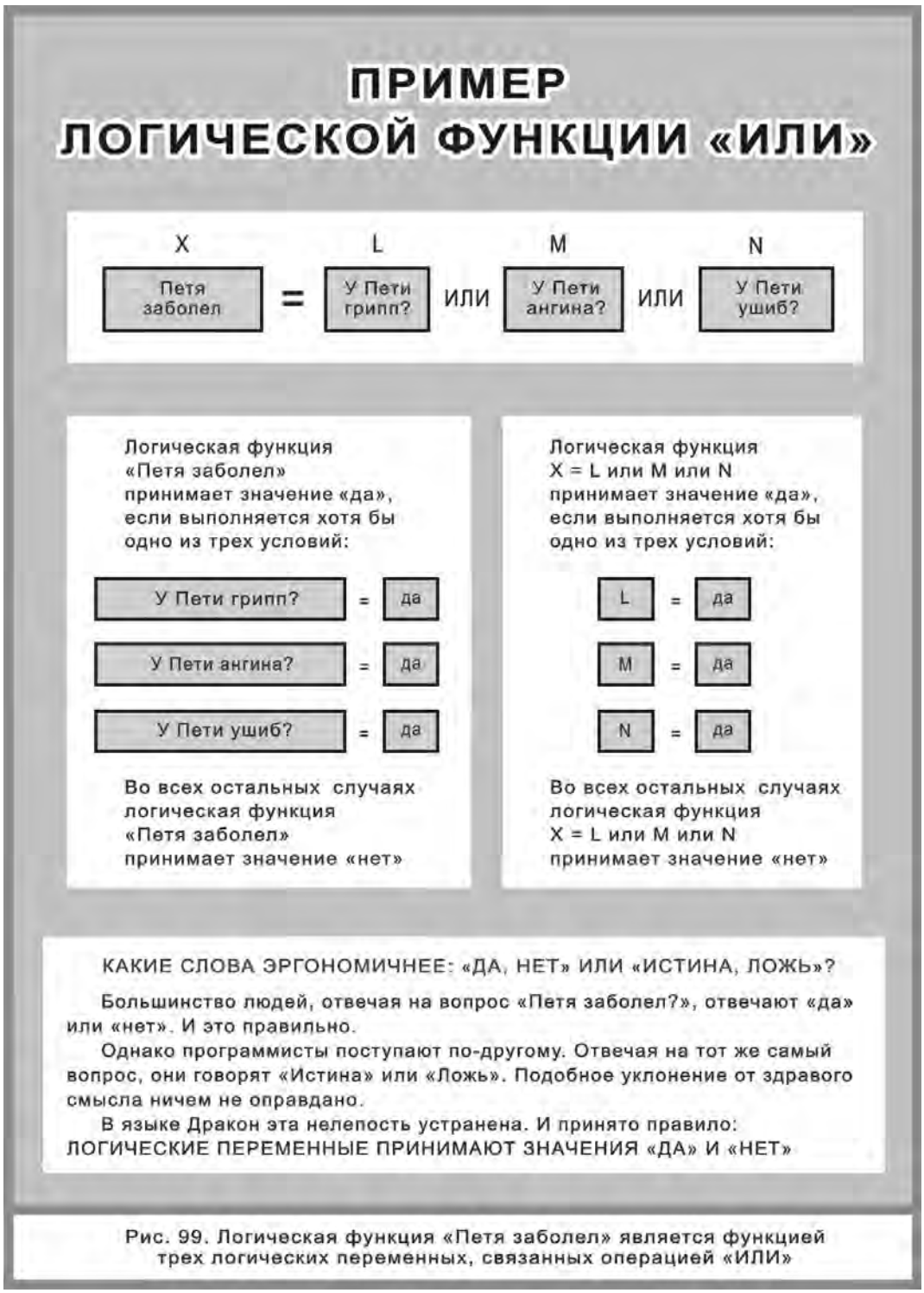


Рис. 99. Логическая функция «Петя заболел» является функцией трех логических переменных, связанных операцией «ИЛИ»

ЛОГИЧЕСКАЯ ФУНКЦИЯ «ИЛИ»

Логическая функция ИЛИ — это функция, которая принимает значение «да», если хотя бы одна логическая переменная имеет значение «да». Функция принимает значение «нет», если все логические переменные имеют значение «нет» (рис. 99).

АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ФУНКЦИЮ «ИЛИ»

На рис. 100 приведены два примера алгоритмов. Слева рассказ о Петиных недугах. Справа тот же самый алгоритм представлен в абстрактной математической форме.

Попытайтесь в правом алгоритме найти и выделить логическую функцию «ИЛИ». Полученный результат пригодится при анализе рис. 101.

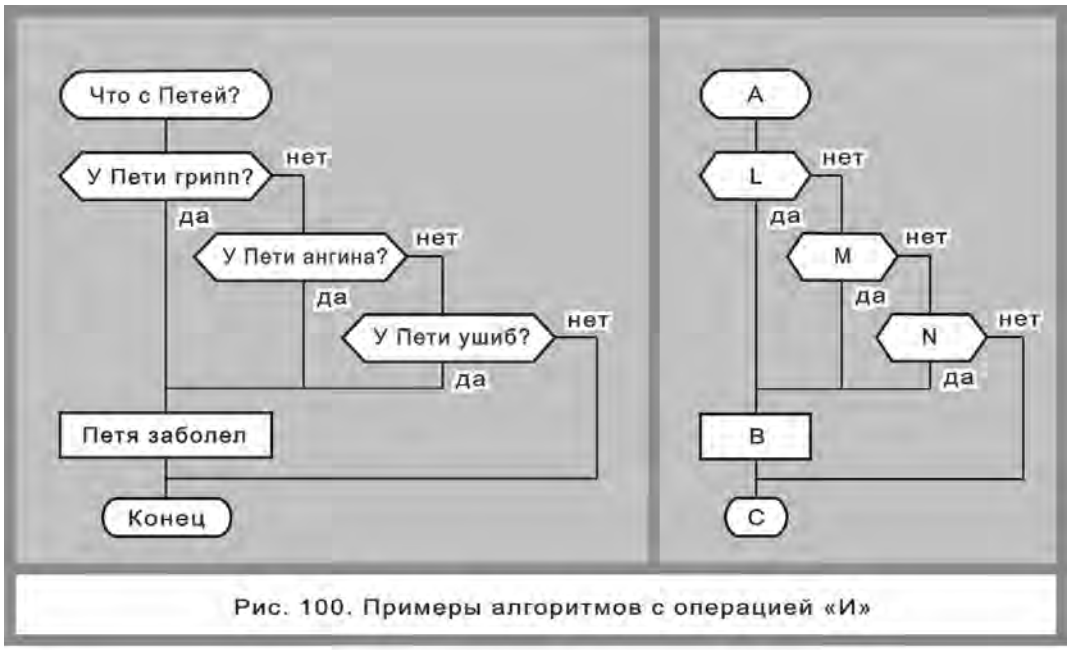


Рис. 100. Примеры алгоритмов с операцией «И»

Рис. 101. Рисуйте дракон-схему «ИЛИ», как показано справа. Избегайте нерекомендуемых схем

ДВА СПОСОБА ЗАПИСИ ФУНКЦИИ «ИЛИ»

На языке ДРАКОН существуют два способа записи функции ИЛИ: *текстовый* и *визуальный*.

В первом случае используют одну икону «вопрос», внутри которой пишут логическое выражение, состоящее из логических переменных, соединенных знаками логической операции ИЛИ (рис. 101 слева).

В другом случае лесенкой рисуют *N* икон «вопрос»,

где *N* — число логических переменных, причем в каждой иконе записывают одну логическую переменную (рис. 101 справа).

КАКОЙ СПОСОБ ЛУЧШЕ: ТЕКСТОВЫЙ ИЛИ ВИЗУАЛЬНЫЙ?

Визуальная формула на рис. 101 показывает, что оба способа эквивалентны.

Для практического использования рекомендуется визуальный способ. Он более нагляден и позволяет быстрее понять суть сложного алгоритма.

Следует подчеркнуть: ДРАКОН *не запрещает* работать с левой формулой. Но тем, для кого она трудна, он предлагает более гуманный и легкий вариант.

СРАВНЕНИЕ МАТЕМАТИЧЕСКОЙ ФОРМУЛЫ И ДРАКОН-СХЕМЫ

На рис. 102 приведена математическая формула и равносильная ей дракон-схема.

Слева — традиционная текстовая формула, понятная узкому кругу математиков и программистов.

Справа — «демократическая» графическая формула, записанная на языке ДРАКОН. Она понятна значительно более широкому кругу работников. Как показывает практика, правая формула доступна даже тем людям, которые испытывают непреодолимые затруднения при работе со сложной левой формулой.

Анализируя формулу слева на рис. 102, нередко приходится вникать в сложные подробности, например:

$$X = [L \vee M \vee N = \text{да}] = [(L = \text{да}) \vee (N = \text{да}) \vee (N = \text{да})]$$

Дракон-схема (рис. 102, справа) хороша тем, что полностью избавляет читателя от подобной ненужной работы.

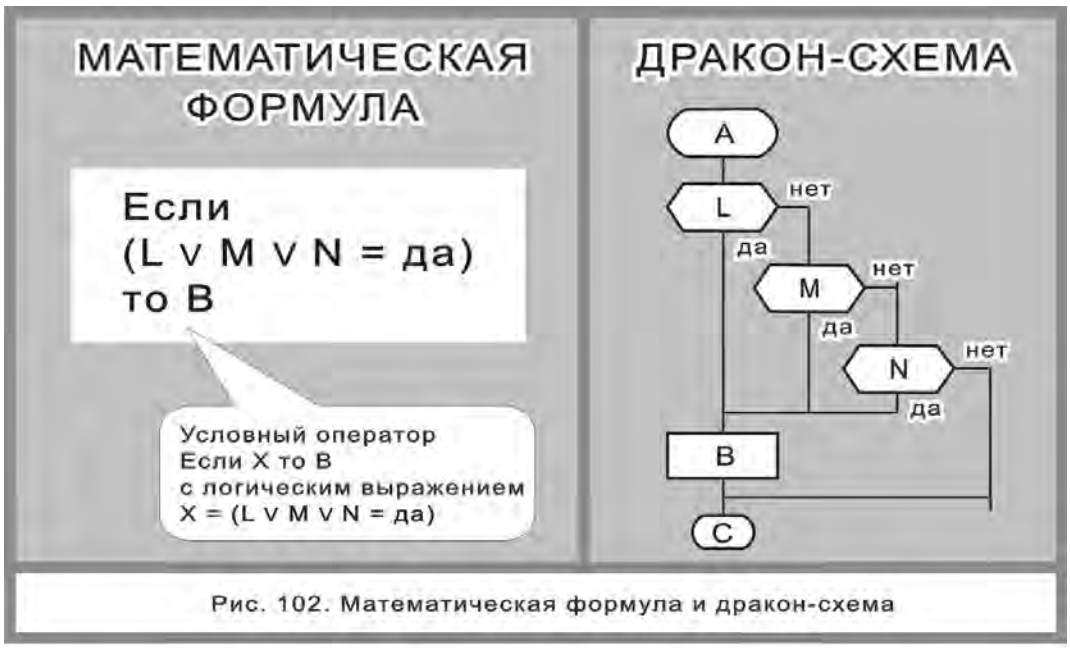


Рис. 102. Математическая формула и дракон-схема

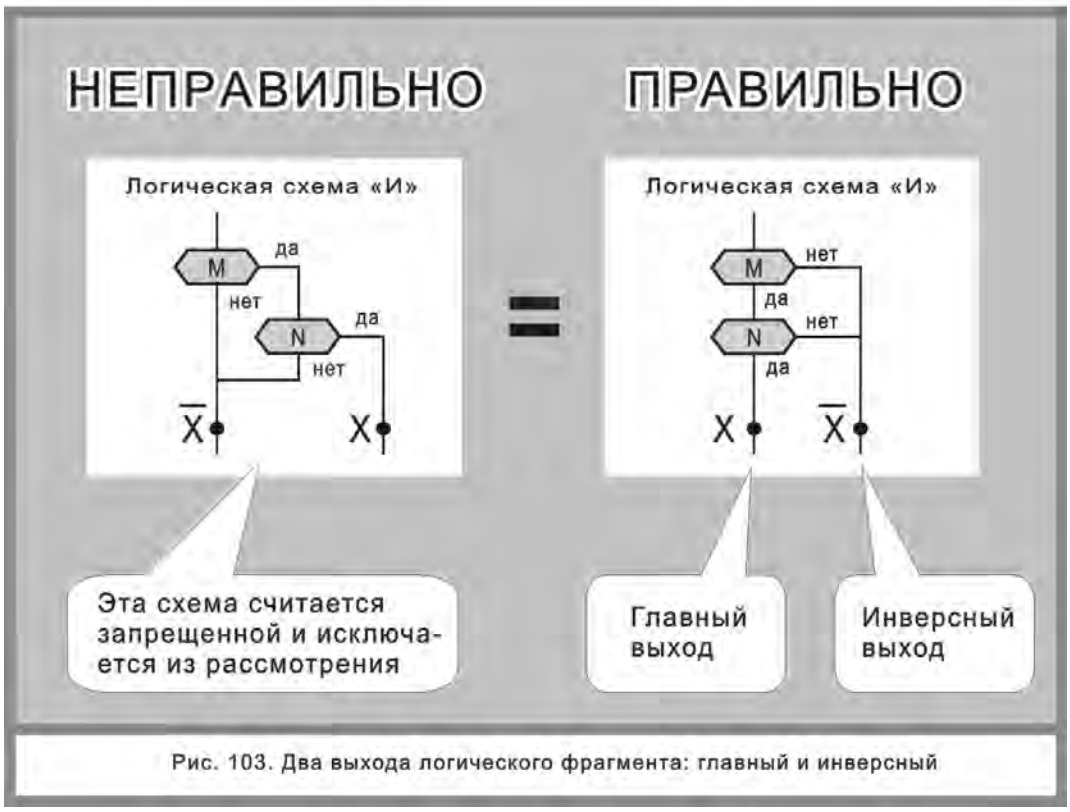


Рис. 103. Два выхода логического фрагмента: главный и инверсный

ЛОГИЧЕСКИЙ ФРАГМЕНТ ДРАКОН-СХЕМЫ

На рис. 103 схема И представлена двумя способами: слева и справа. Подобная неоднозначность зрительных образов нежелательна, так как может привести к путанице. В связи с этим левая схема считается «незаконной» и исключается из рассмотрения. Весь дальнейший текст относится исключительно к «законной» правой схеме.

Определение. **Фрагмент дракон-схемы называется логическим, если он имеет один вход, два выхода и содержит только иконы «вопрос».**

Левый выход логического фрагмента называется главным, правый — инверсным. Главный выход лежит на шампуре и обычно обозначается буквой X. Инверсный выход находится справа от шампура и обозначается буквой $\overline{X}$ (рис. 103).

В реальных дракон-схемах буквы X и $\overline{X}$ обычно не пишут, а подразумевают.

ЛОГИЧЕСКАЯ ФУНКЦИЯ «НЕ»

Логическая функция НЕ — это функция $W = \overline{Z}$, где логические переменные *Z* и *W* принимают инверсные значения, то есть удовлетворяют условиям:

если *Z* = да, то *W* = нет;
если *Z* = нет, то *W* = да.

ВИЗУАЛИЗАЦИЯ ФУНКЦИИ «НЕ»

Как известно, логическое отрицание представляет определенную трудность для понимания. В связи с этим Эдвард Йодан советует:

> «Если это возможно, избегайте отрицаний в булевых выражениях. Представляется, что их понимание представляет трудность для многих программистов» [2].

Учитывая сказанное, ниже будет показано, что логическое отрицание (а также логические связки «И» и «ИЛИ») можно безболезненно изъять из логических выражений.

> *Правило.* **Знак логического отрицания (верхнюю черту) всегда можно исключить из дракон-схемы. Для этого надо поменять местами слова «да» и «нет» на выходах иконы вопрос.**
> (При этом иконы, находящиеся в плечах развилки, следует оставить на своих местах).

Визуальные формулы на рис. 104 иллюстрируют это правило.

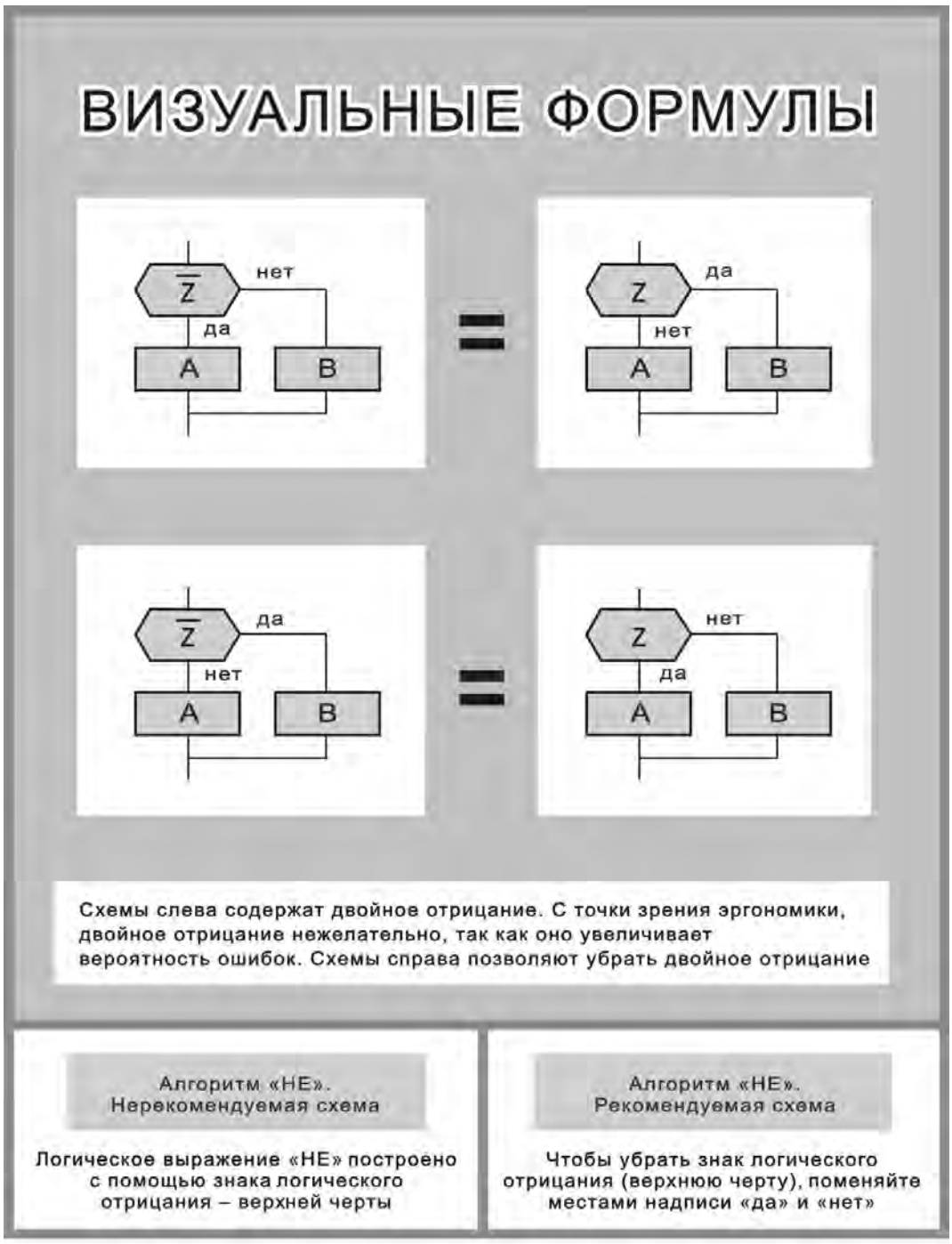


Рис. 104. Визуальные формулы, позволяющие освободиться от знака логического отрицания (верхней черты).

КАНОНИЧЕСКИЙ ВИД ДРАКОН-СХЕМЫ

Определение. Дракон-схема имеет канонический вид, если она:

- не содержит логических связок И, ИЛИ, НЕ;
- не содержит повторяющихся частей, которые можно удалить с помощью операций горизонтального и вертикального объединения;
- иконы «вопрос», образующие логическую схему И расположены на шампуре, (см. рис. 97, справа);
- иконы «вопрос», образующие логическую схему ИЛИ расположены лесенкой, (см. рис. 101, справа).

Теорема. Дракон-схему всегда можно привести к каноническому виду с помощью цепочки равносильных преобразований.

Приведение дракон-схемы к каноническому виду показано для частного случая (рис. 105).

Упражнения на рис. 106 помогут читателю закрепить материал.

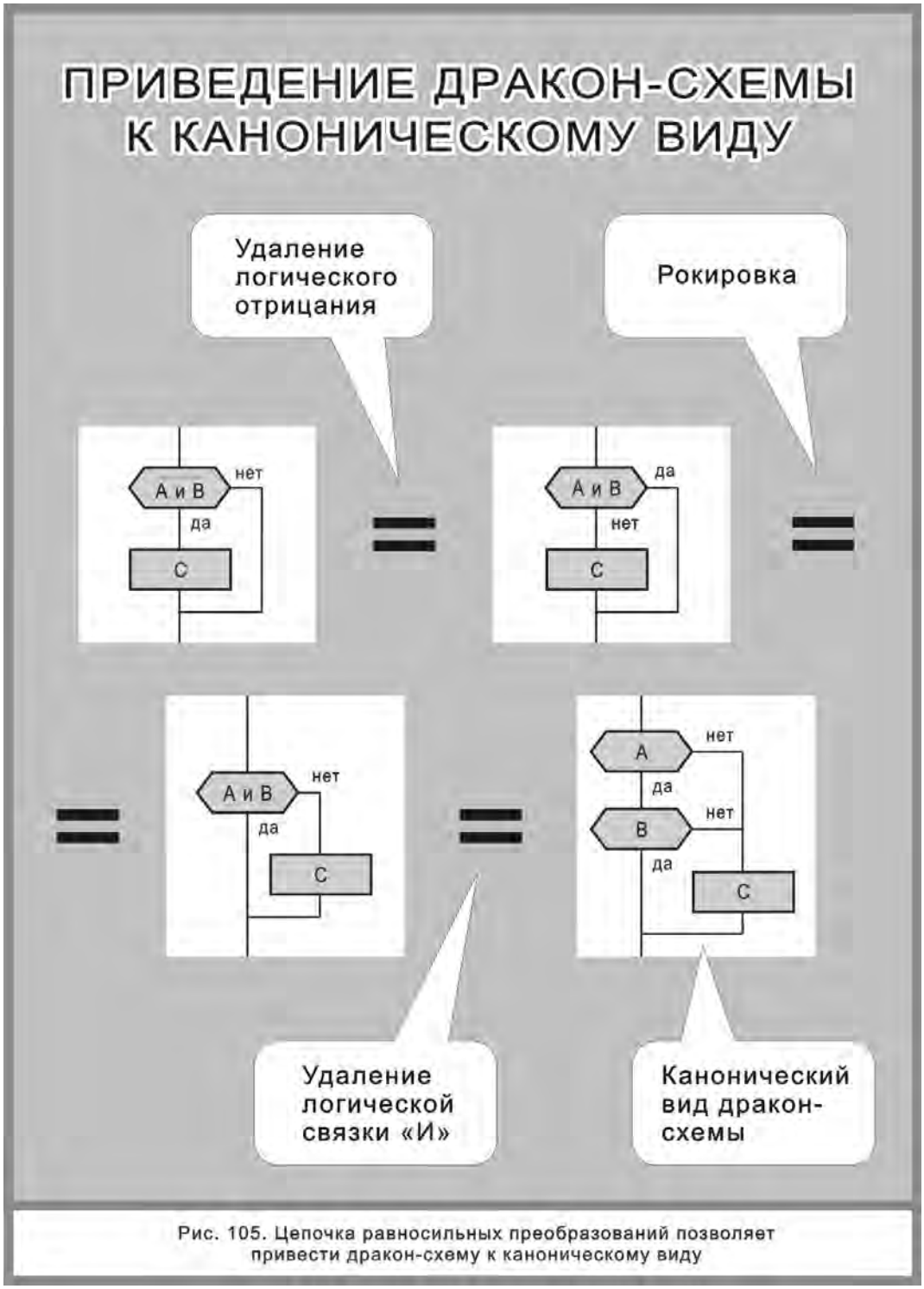


Рис. 105. Цепочка равносильных преобразований позволяет привести дракон-схему к каноническому виду

ЗАПРЕЩЕННЫЕ ДРАКОН-СХЕМЫ

На рис. 103 был показан пример запрещенной схемы «И». Рассмотрим вопрос в более общем виде.

Правило. **Выход логической функции X должен находиться на шампуре.**

Понятие «канонический вид» подразумевает запрет использовать дракон-схемы на рис. 107 слева. Запрет объясняется тем, что в этих схемах выход логической функции X удален от шампура.

Теорема. Если главный выход логической дракон-схемы есть результат вычисления логической функции X, то инверсный выход вычисляет ее логическое отрицание $\overline{X}$.

Доказательство теоремы предоставляем читателю.

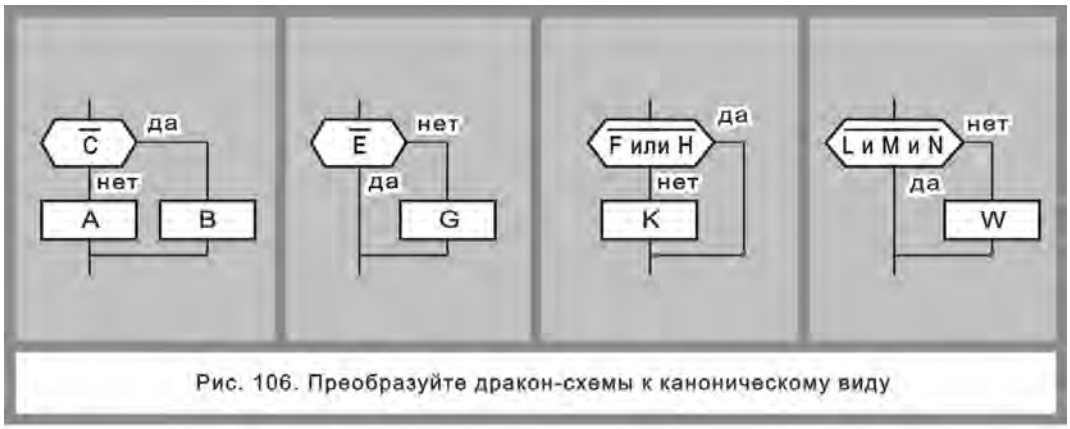


Рис. 106. Преобразуйте дракон-схемы к каноническому виду

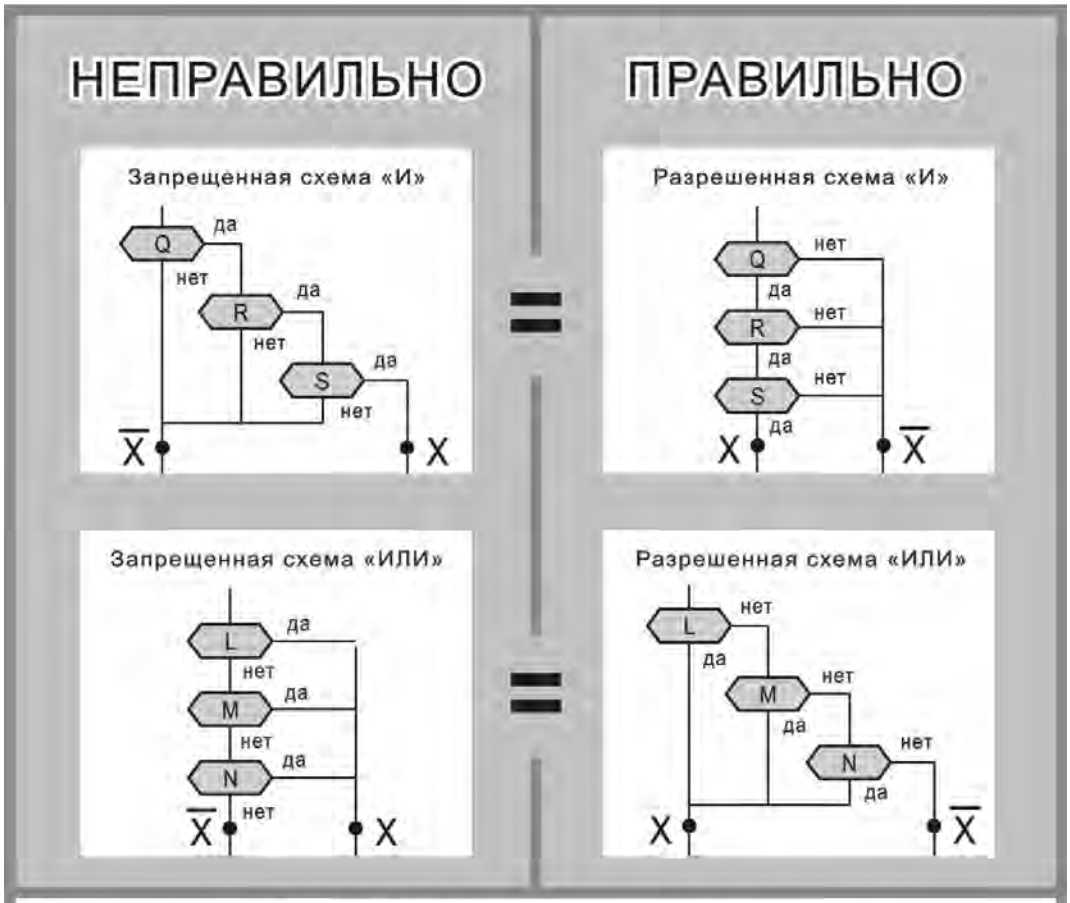


Рис. 107. Рисуйте схемы, как показано справа.
Схемы слева логически правильны; тем не менее они ЗАПРЕЩЕНЫ,
чтобы исключить неоднозначность

ВИЗУАЛИЗАЦИЯ СЛОЖНЫХ ЛОГИЧЕСКИХ ФУНКЦИЙ

Рассмотрим функцию

$$X = (A \text{ и } \overline{B} \text{ и } C) \text{ или } (D \text{ и } E \text{ и } \overline{F}) \qquad (1)$$

На рис. 108 показан визуальный способ записи этой функции. Из рисунка видно, что формула (1) разбивается на три части:

- *A* и $\overline{B}$ и *C*;
- *D* и *E* и $\overline{F}$;
- Операция «или».

Функция *A* и $\overline{B}$ и *C* изображается с помощью трех икон *A*, *B*, *C*, расположенных на одной вертикали. Аналогично рисуют функцию *D* и *E* и $\overline{F}$. Связка «или» реализуется с помощью линий, объединяющих нижние выходы икон *C* и *F* в точке *K* (рис. 108).

В формуле (1) некоторые члены записаны без логического отрицания (*A*, *B*, *D*, *E*), другие — с отрицанием ($\overline{B}$, $\overline{F}$). Члены без отрицания превращаются в иконы *A*, *B*, *D*, *E*, у которых нижний выход помечен словом «да». Членам с отрицанием соответствуют иконы *B* и *F*, где нижний выход помечен словом «нет» (рис. 108).

Другие примеры алгоритмов, вычисляющих сложные логические функции, представлены на рис. 109, 110.

Опираясь на изложенные соображения, можно показать, что справедлива

***Теорема*.** Дракон-схему, содержащую логические связки И, ИЛИ, НЕ внутри икон «вопрос», всегда можно преобразовать в эквивалентную дракон-схему, не содержащую указанных связок.

Доказательство теоремы, как всегда, оставляем читателю.

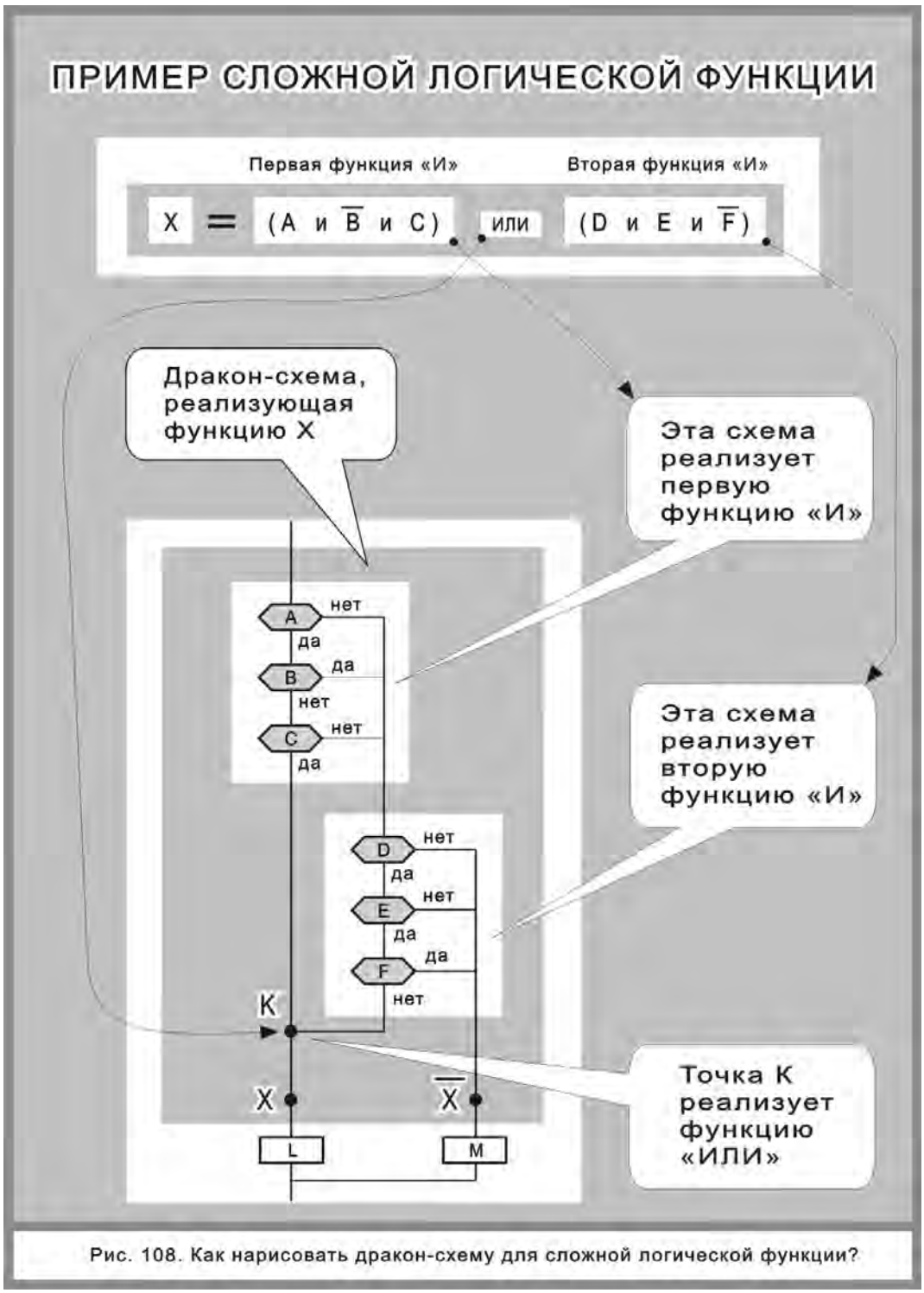


Рис. 108. Как нарисовать дракон-схему для сложной логической функции?

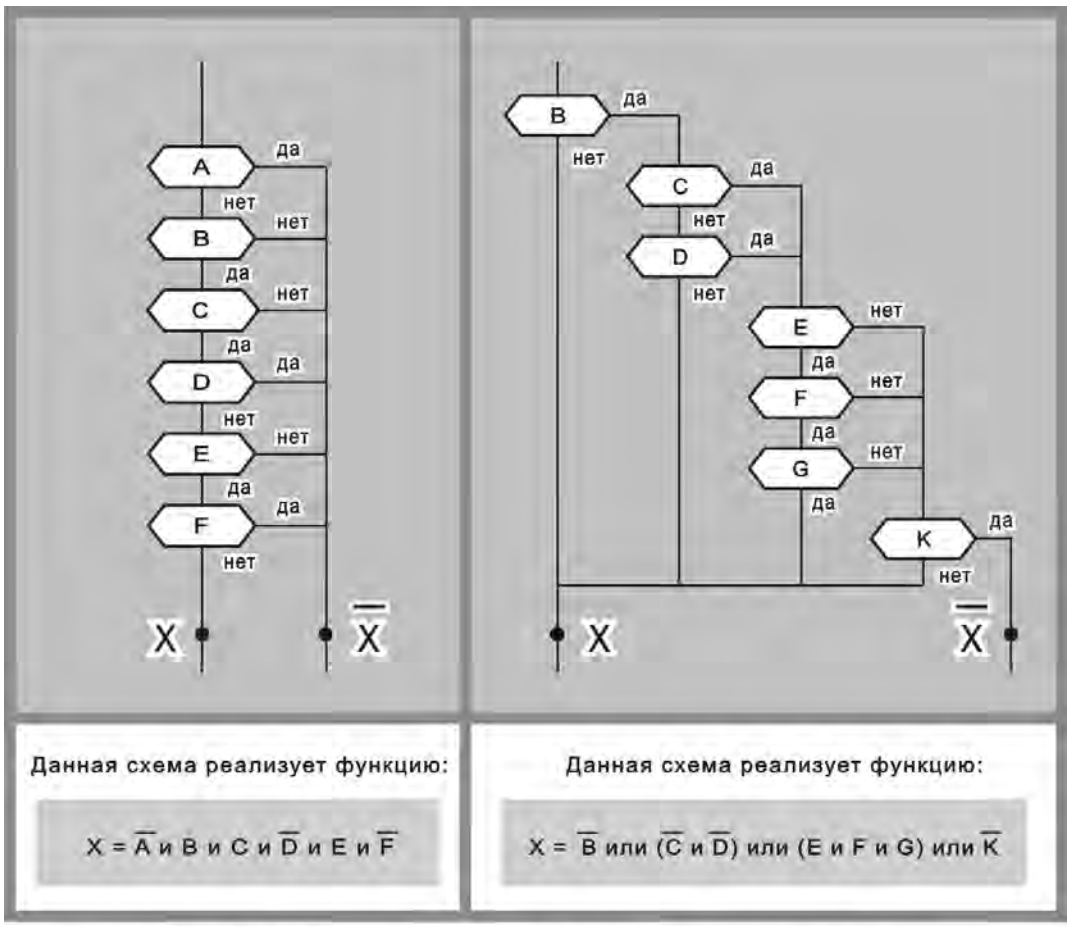


Рис. 109. Преобразование дракон-схемы в логическую функцию

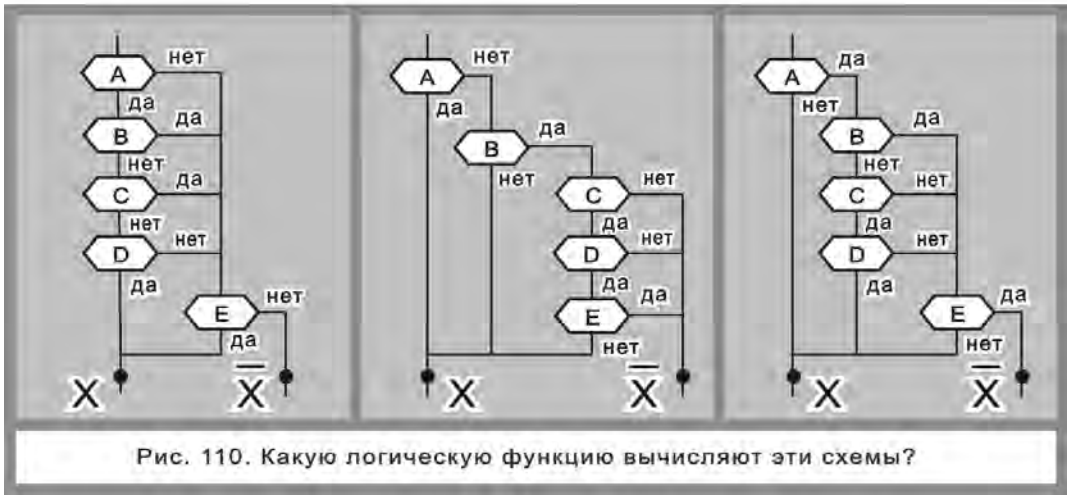


Рис. 110. Какую логическую функцию вычисляют эти схемы?

ВЫВОДЫ

1. В алгоритмах со сложной логикой часто используются условные операторы с логическими выражениями. Опыт показывает, что такие операторы во многих случаях трудны для понимания, что нередко приводит к ошибкам.
2. В языке ДРАКОН используются визуальные логические выражения, позволяющие при желании полностью исключить логические связки И, ИЛИ, НЕ из условных операторов.
3. Визуализация логических формул во многих практически важных случаях заметно облегчает их понимание и уменьшает вероятность ошибок.

Глава 5

ВИЗУАЛЬНЫЕ ОПЕРАТОРЫ РЕАЛЬНОГО ВРЕМЕНИ

> Удачный рисунок иногда не только позволяет сделать наглядной и понятной суть сложного вопроса, но нередко способен подсказать принципиально новое соображение, идею, гипотезу, которые без такого рисунка просто, что называется, не приходят в голову.
>
> *Александр Зенкин*

СПИСОК ОПЕРАТОРОВ РЕАЛЬНОГО ВРЕМЕНИ

В языке ДРАКОН имеется пять икон реального времени (рис. 4, иконы И16 — И20):

- пауза;
- период;
- пуск таймера;
- синхронизатор (по таймеру);
- параллельный процесс.

Три из них (пауза, пуск таймера и параллельный процесс) — простые операторы. Две другие иконы (период и синхронизатор) служат «кирпичиками» для построения составных операторов и вне последних не используются.

Икона «период» является принадлежностью цикла ЖДАТЬ (рис. 5, макроикона 7). Икона «синхронизатор» служит для образования тринадцати составных операторов (рис. 5, макроиконы 8—20).

Назначение операторов поясним, как всегда, на примерах.

ОПЕРАТОРЫ ВВОДА-ВЫВОДА

В языке ДРАКОН предусмотрены два визуальных оператора ввода-вывода: «вывод» (рис. 4, икона И14) и «ввод» (рис. 4, икона И15). Они не относятся к операторам реального времени и рассматриваются здесь только потому, что встречаются в ближайшем примере.

Из рис. 4 видно, что иконы ввода-вывода имеют мнемоническую форму. Икона И14 содержит полую стрелку, направленную наружу, что символизирует «вывод», а икона И15 — стрелку, направленную внутрь (ввод).

Оба оператора «двухэтажные». На верхнем этаже пишется ключевое слово или ключевая фраза. На нижнем (в прямоугольнике) — содержательная информация, подлежащая вводу и выводу (рис. 122, 123).

Рис. 122. Пример использования оператора «пауза»

Рис. 123. Пример использования операторов «пуск таймера» и «синхронизатор»

ОПЕРАТОР «ПАУЗА»

Предположим, управляющий компьютер должен выдать серию электрических команд, которые по линиям связи передаются в исполнительные органы и вызывают срабатывание электромеханических реле. В результате происходит открытие трубопровода, включение насоса и другие операции, необходимые для функционирования управляемого объекта.

Такая ситуация может встретиться во многих системах управления реального времени. Например, при заправке топливом баллистических ракет, на атомных электростанциях, нефтеперерабатывающих заводах и т. д.

Рассмотрим пример. Предположим, управляющий компьютер должен:

- выдать команду ОТКРЫТЬ.ТРУБОПРОВОД;
- подождать две минуты;
- выдать две команды: ВКЛЮЧИТЬ.НАСОС и ОТКРЫТЬ.ЗАСЛОНКУ;
- подождать 45 секунд;
- выдать команду ПОДАЧА.ТОПЛИВА;
- подождать три минуты;
- выдать команду ПУСК.АГРЕГАТА.

Соответствующая программа представлена на рис. 122. Задержка выдачи команд реализуется с помощью иконы «пауза». Внутри последней указывается время необходимой задержки. Например, 2 мин (2 минуты), 45 с (45 секунд) и т. д.

Если говорить более точно, верхний оператор «пауза» на рис. 122 работает так. После выдачи команды ОТКРЫТЬ.ТРУБОПРОВОД в управляющем компьютере запускается виртуальный счетчик времени на 2 ми-

нуты. По истечении этого времени компьютер выдает в линию связи команды ВКЛЮЧИТЬ.НАСОС и ОТКРЫТЬ.ЗАСЛОНКУ.

ОПЕРАТОРЫ «ПУСК ТАЙМЕРА» И «СИНХРОНИЗАТОР»

Вернемся еще раз к задаче, описанной в предыдущем параграфе, и слегка изменим ее. Будем считать, что разработчик управляемого объекта хочет указать время выдачи команд не по принципу «задержка после предыдущей команды», а по принципу секундомера. Это значит, что все времена отсчитываются от единого начального момента (совпадающего с пуском секундомера).

Исходя из этого, сформулируем задачу управляющего компьютера. Он должен:

- включить «секундомер», т. е. обнулить и запустить виртуальный таймер;
- выдать команду ОТКРЫТЬ.ТРУБОПРОВОД;
- когда таймер отсчитает две минуты, выдать пару команд ВКЛЮЧИТЬ.-НАСОС и ОТКРЫТЬ.ЗАСЛОНКУ;
- когда таймер отсчитает 2 минуты 45 секунд, выдать команду ПОДАЧА.ТОПЛИВА;
- когда таймер отсчитает 5 минут 45 секунд, выдать команду ПУСК. АГРЕГАТА.

Программа, реализующая описанный алгоритм, изображена на рис. 123. В ней используются операторы «пуск таймера» и «синхронизатор», совместная работа которых обеспечивает нужный эффект.

Оператор «пуск таймера» порождает, обнуляет и запускает виртуальный таймер и присваивает ему имя *А*. Оператор «синхронизатор» задерживает выполнение размещенного справа от него визуального оператора до наступления момента, указанного в иконе «синхронизатор».

Например, синхронизатор *А* = 2мин 45с на рис. 83 задерживает выдачу команды ПОДАЧА.ТОПЛИВА до момента, когда таймер *А* отсчитает 2 минуты 45 секунд.

Сравнивая программы на рис. 122 и 123, можно заметить, что они почти эквивалентны. Почему почти?

Чтобы разобраться, рассмотрим идеальный случай. Представим, что время, необходимое для выдачи одной команды равно нулю. В этом случае обе программы будут выдавать команды синхронно.

Однако в действительности идеальные случаи встречаются далеко не всегда. Иногда бывает, что время выдачи одной команды больше нуля. В таком случае программы работают по-разному.

Практика разработки систем управления показывает, что в некоторых ситуациях предпочтительным является *принцип паузы*, а в других — *принцип таймера.* Поэтому оба инструмента оказываются в равной степени необходимыми и полезными.

АЛГОРИТМ РЕАЛЬНОГО ВРЕМЕНИ

На рис. 124 представлен более сложный алгоритм, в котором используются операторы «пауза», «пуск таймера» и «синхронизатор».

В средней ветке изображена икона «пауза» с записью 2мин48с. Это означает, что после завершения процедуры ВОЛШЕБНЫЙ РЕМОНТ ТА-

РЕЛКИ отсчитывается пауза длительностью 2 минуты 48 секунд. И только после этого производится снятие признака АВАРИЯ ТАРЕЛКИ.

Еще одна 4-секундная пауза предусмотрена в левой ветке.

В правой ветке есть икона «пуск таймера» с записью *A* = 0. Данный оператор порождает, обнуляет и запускает виртуальный таймер *A*.

В той же ветке установлены три иконы «синхронизатор по таймеру» с записями *A* = 3мин, *A* = 5мин и *A* = 8мин. При этом вызов процедуры ВКЛЮЧИТЬ ТЕЛЕПОРТАЦИЮ произойдет не сразу, а только после того, как таймер *A* отсчитает 3 минуты. Соответственно включение в работу процедур ОТКЛЮЧИТЬ ГРАВИТАЦИЮ и ВЫХОД ИЗ АСТРАЛЬНОГО ТЕЛА будет задержано до тех пор, пока таймер *A* не примет значения 5 и 8 минут соответственно.

Из рис. 124 видно, что оператор «пуск таймера» можно применять двумя способами:

- во-первых, совместно с иконой «синхронизатор» (этот случай мы обсудили);
- во-вторых, совместно с иконой «вопрос».

Последний случай рассмотрен в следующем параграфе.

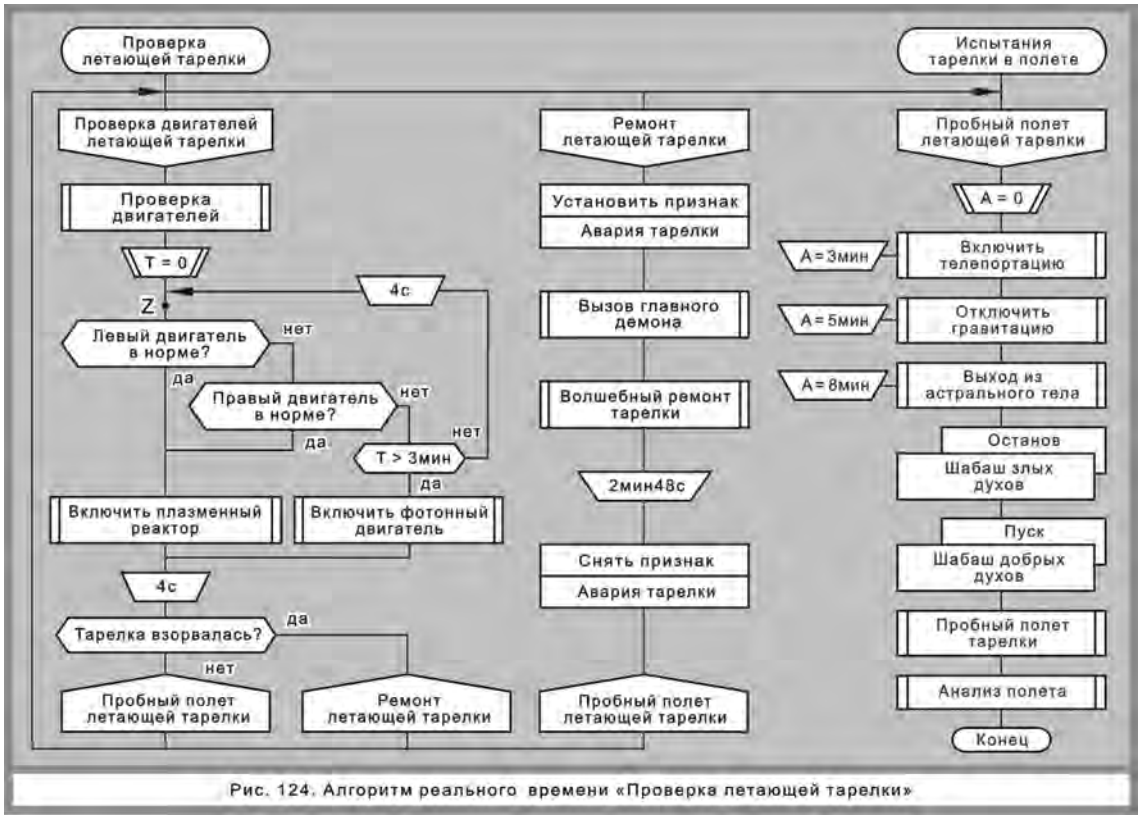


Рис. 124. Алгоритм реального времени «Проверка летающей тарелки»

ЦИКЛ ЖДАТЬ

Предположим, нужно в течение 3-х минут ждать появления хотя бы одного из двух признаков ЛЕВЫЙ ДВИГАТЕЛЬ В НОРМЕ и ПРАВЫЙ ДВИГАТЕЛЬ В НОРМЕ. При наступлении этого события (появлении одного из признаков) необходимо включить плазменный реактор. Если же названные признаки отсутствуют, по истечении трех минут следует включить фотонный двигатель.

Для решения задачи на рис. 124 используются два оператора:

- пуск таймера *T*, отсчитывающего три минуты;

- цикл ЖДАТЬ.

В состав последнего входит икона «период» и три иконы «вопрос». В последних размещены надписи:

- ЛЕВЫЙ ДВИГАТЕЛЬ В НОРМЕ?
- ПРАВЫЙ ДВИГАТЕЛЬ В НОРМЕ?
- *T* > 3мин

Последний оператор проверяет: значение таймера *T* больше трех минут?

Если оба признака отсутствуют, а значение таймера не превышает 3-х минут, опрос условий периодически повторяется. При этом период опроса указывается в иконе «период». В данном примере он равен 4 секундам.

Как явствует из рисунка, работа цикла ЖДАТЬ закончится в момент обнаружения одного из ожидаемых признаков, а если они так и не появятся, — через 3 минуты.

ЦИКЛ «ЖДАТЬ» В ОБЩЕМ ВИДЕ

В общем виде цикл ЖДАТЬ показан на рис. 125. Он позволяет организовать режим ожидания признаков *B*, *C*, *D*, ..., *E*. Если первым появится признак *B*, выполняется действие *F*. Если *B* отсутствует и первым придет признак *C*, реализуется действие *G*. И так далее. Операторы *A* и *L* обычно не используются.

Задача ожидания нескольких признаков (когда система должна по-разному реагировать на каждый признак) является одной из наиболее типичных при разработке систем управления реального времени.

Цикл ЖДАТЬ предлагает чрезвычайно простое, удобное, наглядное и эффективное средство для ее решения, удовлетворяя тем самым важную потребность практики.

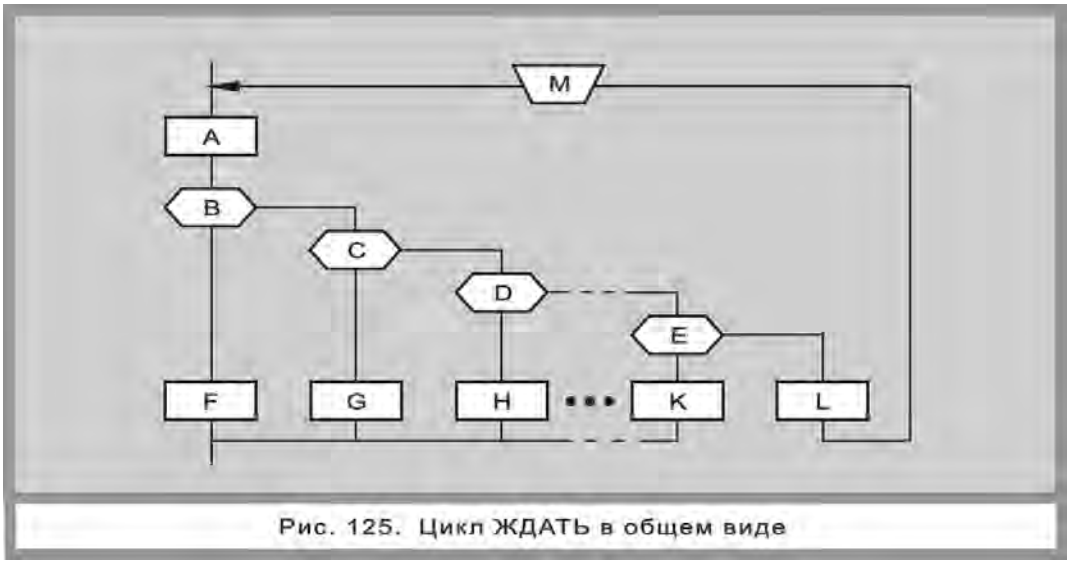


Рис. 125. Цикл ЖДАТЬ в общем виде

ОПЕРАТОР «ПЕРИОД»

Сравнивая макроиконы 4 и 7 на рис. 5 (обычный цикл и цикл ЖДАТЬ), мы видим, что они очень похожи. Поэтому во избежание путаницы нужно иметь какой-то различительный признак. Эту функцию выполняет икона «период». Если она есть в петле цикла — перед нами цикл ЖДАТЬ. Если нет — обычный цикл.

Человек, который стоит на остановке и ждет появления трамвая, воспринимает ожидание как нечто непрерывное. Однако программа реального времени организует ожидание как дискретный процесс и запускает цикл ЖДАТЬ периодически. Отсюда вытекает, что период — важная характеристика цикла ЖДАТЬ.

А теперь зададим самый интересный вопрос: как работает оператор «период»? Фокус в том, что на этот вопрос придется дать два совсем разных ответа.

С точки зрения человека, читающего алгоритм на рис. 124, все обстоит очень просто. Цикл ЖДАТЬ «крутится» по своей петле с периодичностью 4 секунды, пока не выполнится одно из трех условий, после чего произойдет выход из цикла. Таким образом, оператор «период» задает период повторения цикла ЖДАТЬ.

С точки зрения функционирования программы реального времени, дело обстоит иначе. Суть в том, что длительность периода отсчитывает не прикладная программа на рис. 124, а дракон-диспетчер, входящий в состав операционной системы реального времени.

Оператор «период» означает выход из прикладной программы. Управление переходит к дракон-диспетчеру (с одновременной передачей параметра 4с). Через каждые 4 секунды дракон-диспетчер передает управление в начало цикла ЖДАТЬ (точка Z на рис. 124). Если все три условия дают ответ «нет», оператор «период» всякий раз возвращает управление в дракон-диспетчер. Таким образом, функционирование цикла ЖДАТЬ обеспечивается совместными усилиями прикладной программы и дракон-диспетчера.

Нередко имеет место ситуация, когда разработчик программы реального времени использует цикл ЖДАТЬ, но считает, что для его программы конкретное значение периода не играет роли. В этом случае икону «период» следует оставить пустой; система по умолчанию присвоит периоду максимальное значение из того ассортимента, которым располагает дракон-диспетчер.

ОПЕРАТОР «ПАРАЛЛЕЛЬНЫЙ ПРОЦЕСС»

Пусть заданы два алгоритма *A* и *B*, причем *A* — основной алгоритм, а *B* — вспомогательный. Алгоритмы *A* и *B* могут работать последовательно (рис. 126) или параллельно (рис. 127).

Чтобы организовать *последовательную* работу, необходимо в дракон-схеме основного алгоритма *A* нарисовать икону-вставку с надписью *B*. В этом случае алгоритм B называется *процедурой.*

Например, на рис. 124 в основном алгоритме ПРОВЕРКА ЛЕТАЮЩЕЙ ТАРЕЛКИ имеется процедура ПРОВЕРКА ДВИГАТЕЛЕЙ. Эти алгоритмы действуют последовательно. Основной алгоритм передает управление процедуре ПРОВЕРКА ДВИГАТЕЛЕЙ и прекращает работу. Возобновление работы алгоритма ПРОВЕРКА ЛЕТАЮЩЕЙ ТАРЕЛКИ произойдет только тогда, когда процедура ПРОВЕРКА ДВИГАТЕЛЕЙ закончится. В общем виде ситуация показана на рис. 126.

Отличие параллельного режима состоит в том, что после начала вспомогательного алгоритма *B* основной алгоритм *A* не прекращает работу и действует одновременно с алгоритмом *B* (рис. 127).

Чтобы организовать параллельную работу, нужно в дракон-схеме основного алгоритма *А* нарисовать икону «параллельный процесс» (рис. 4, икона И20).

Икона «параллельный процесс» двухэтажная. На верхнем этаже пишут ключевое слово, обозначающее команду, изменяющую состояние параллельного процесса, например, «Пуск», «Останов» и т. д. На нижнем этаже помещают идентификатор (название) параллельного процесса.

Обратимся к примеру на рис. 124. В правой ветке находятся два оператора управления параллельными процессами. После окончания процедуры ВЫХОД ИЗ АСТРАЛЬНОГО ТЕЛА производится останов параллельного процесса ШАБАШ ЗЛЫХ ДУХОВ и пуск процесса ШАБАШ ДОБРЫХ ДУХОВ.

При этом предполагается, что до начала алгоритма ПРОВЕРКА ЛЕТАЮЩЕЙ ТАРЕЛКИ некий третий алгоритм выдал команду «Пуск» и запустил параллельный процесс ШАБАШ ЗЛЫХ ДУХОВ. Последний работает одновременно с алгоритмом ПРОВЕРКА ЛЕТАЮЩЕЙ ТАРЕЛКИ вплоть до момента выдачи команды «Останов» (см. последнюю ветку на рис. 124).

Указанная команда ликвидирует параллельный процесс ШАБАШ ЗЛЫХ ДУХОВ. В этот момент одновременная работа заканчивается.

Однако следующая команда «Пуск» запускает другой параллельный процесс — ШАБАШ ДОБРЫХ ДУХОВ, который начинает работать одновременно с алгоритмом ПРОВЕРКА ЛЕТАЮЩЕЙ ТАРЕЛКИ.

Рис. 126. Последовательная работа алгоритмов

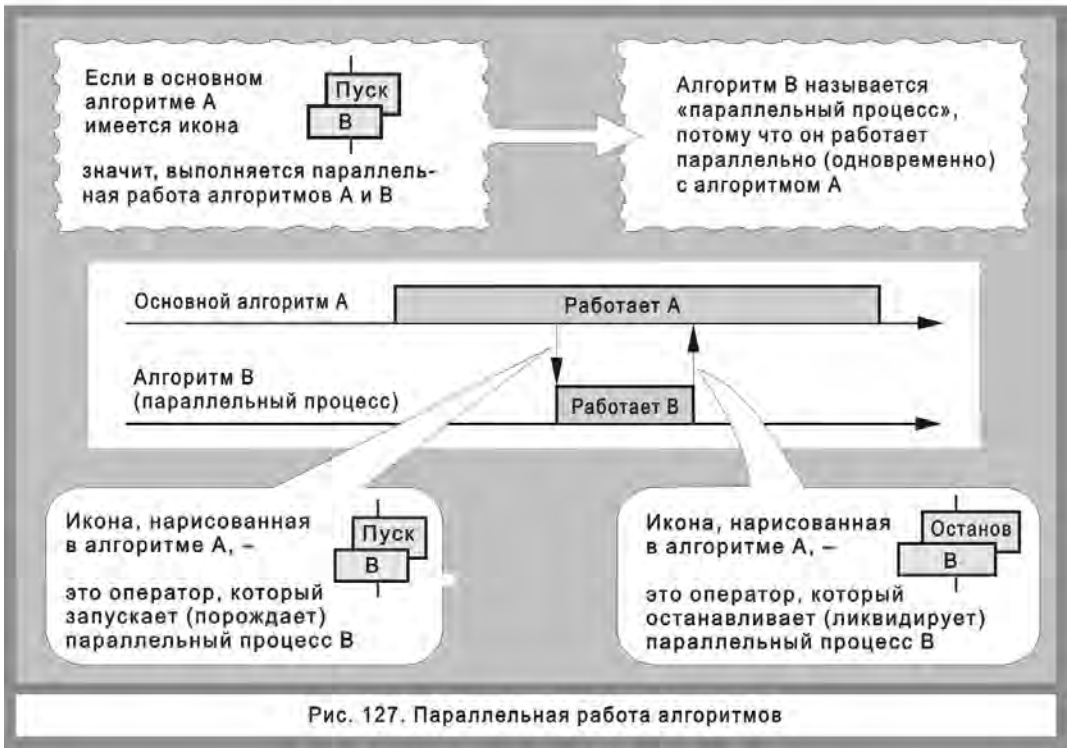


Рис. 127. Параллельная работа алгоритмов

ОСОБЕННОСТИ ОПЕРАТОРОВ РЕАЛЬНОГО ВРЕМЕНИ

Уже говорилось, что цикл ЖДАТЬ выполняется прикладной программой при участии дракон-диспетчера. Этот вывод относится ко всем операторам реального времени. Следует подчеркнуть, что данное утверждение относится не к языку, а к реализации системы и для разных реализаций может быть различным.

Операторы реального времени — это формальные операторы языка визуального программирования ДРАКОН-2. Однако их можно использовать и в языке ДРАКОН-1 при неформальном изображении алгоритмов.

Например, для построения наглядных «картинок», позволяющих легко объяснить ту или иную идею, относящуюся к системам реального времени.

Примеры таких картинок представлены на рис. 128 и 129. При этом в цикле ЖДАТЬ икону «период» обычно опускают, чтобы не загромождать рисунок (см. последнюю ветку на рис. 128). Однако если длительность периода нужна для понимания, икону «период» можно сохранить (рис. 129).

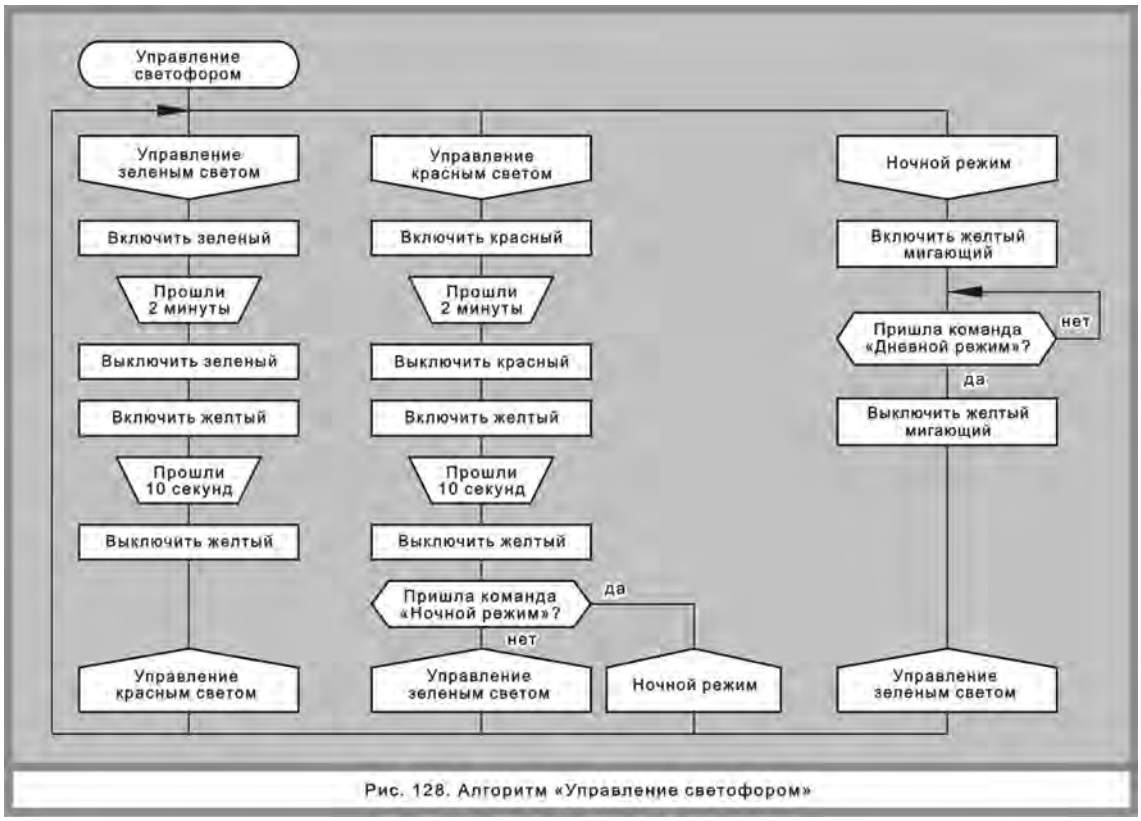


Рис. 128. Алгоритм «Управление светофором»

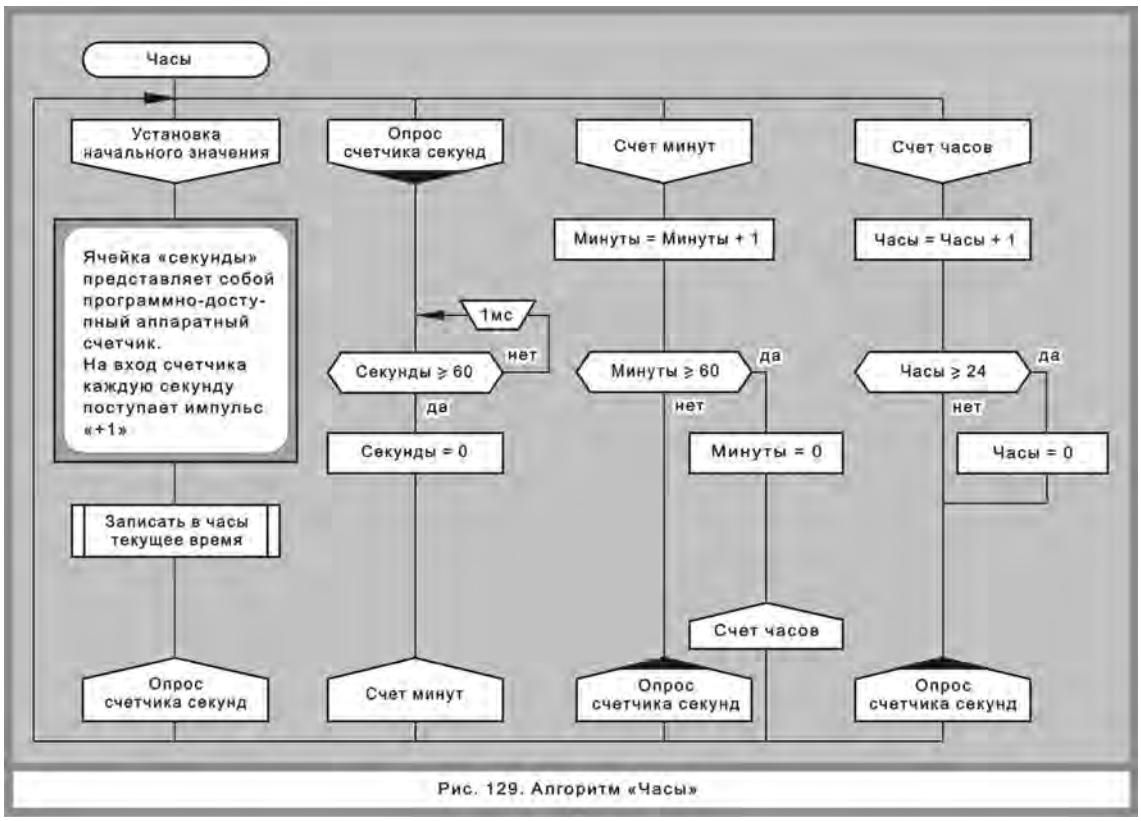


Рис. 129. Алгоритм «Часы»

БЕСКОНЕЧНЫЕ АЛГОРИТМЫ

В отличие от обычных вычислительных и информационных программ в программах реального времени икона «конец» может отсутствовать. Это имеет место, когда нужно организовать бесконечный цикл, который прекращается особой внешней причиной, например выключением питания системы (рис. 128, 129).

НЕСКОЛЬКО ВХОДОВ В ДРАКОН-ПРОГРАММУ

Дракон-программа может иметь более одного входа. Чтобы организовать дополнительный вход, нужно поместить икону «заголовок» над иконой «имя ветки», как показано на рис. 124 справа.

Таким образом, любая ветка может быть объявлена дополнительным входом. Однако есть исключение: если несколько веток образуют веточный цикл, вход разрешается только в начало цикла. Остальные ветки конструкции «веточный цикл» не могут являться входами в программу.

Разумеется, созданием нескольких входов в программу не следует злоупотреблять. Этот прием следует использовать лишь в особых случаях.

ВЫВОДЫ

1. Наличие операторов реального времени резко расширяет изобразительные возможности языка ДРАКОН и позволяет использовать его при проектировании и разработке не только информационных, но и управляющих систем. Это обстоятельство существенно увеличивает область применения языка.
2. Дополнительным преимуществом является лаконичность выразительных средств, их универсальность. В языке всего пять икон реального времени, однако их алгоритмическая мощь — в сочетании с другими возможностями языка — позволяет охватить обширный спектр задач, связанных с созданием программного обеспечения для управляющих систем.
3. Важную роль играет эргономичность операторов реального времени. Как и другие операторы языка ДРАКОН, они имеют визуальный характер, что позволяет сделать операции реального времени более наглядными и легкими для понимания по сравнению с традиционной текстовой записью.
4. Четыре иконы (пауза, период, пуск таймера и синхронизатор) — «близкие родственники» в том смысле, что внутри каждой из них указывается значение времени. Эта родственная связь находит свое эргономическое отражение в том, что перечисленные операторы имеют визуальное «фамильное сходство». Все они построены (с вариациями) на основе одной и той же геометрической фигуры — перевернутой равнобедренной трапеции.
5. Операторы реального времени порождают сложные действия компьютера, связанные с частыми передачами управления между прикладной программой и операционной системой (дракон-диспетчером). Эргономическая изюминка состоит в том, что эти передачи намеренно скрыты от читателя программы, чтобы не загромождать ее текст (чертеж) второстепенными подробностями. Благодаря этому внимание читателя не отвлекается на мелочи, и он имеет возможность сосредоточиться на главном, поскольку дракон-схема предоставляет ему ясную, четкую и целостную картину алгоритмического процесса, очищенную от «мелкого мусора».

ЧТО ЛУЧШЕ: СИЛУЭТ ИЛИ ПРИМИТИВ?

Этот вопрос в упрощенном виде обсуждался в главе 1. Сейчас мы дадим окончательные рекомендации.

1. Силуэт — главное достоинство языка Дракон. Более того, силуэт — его ЕДИНСТВЕННОЕ достоинство.

2. Примитив не в счет, так как его вообще не следует использовать (почти).

3. Алгоритм (или программу) надо рассматривать как последовательную декомпозицию силуэтов. В том смысле, что каждый силуэт может содержать много вставок, каждая из которых раскрывается как силуэт. Примитивы при этом не используются (совсем или почти совсем).

4. Тем не менее, полностью отказываться от понятия «примитив» не следует по двум причинам.

5. Первая причина — педагогическая. Примитив — это прообраз (зародыш) ветки. Основные понятия и правила Дракона удобно объяснять на самой простой модели. То есть на примитиве. И только после этого переходить к рассказу о силуэте.

6. Вторая причина — необходимость описания «мелких огрызков». Откуда берутся «мелкие огрызки»? В процессе декомпозиции силуэта может случиться (очень редко), что какая-нибудь вставка окажется очень простой, элементарной. Настолько простой, что ее неудобно представлять в виде силуэта. Такую вставку можно назвать «мелким огрызком». Вот в этом (исключительном) случае полезно использовать примитив.

7. Добавим еще одну мысль. **Ни в коем случае нельзя представлять программу как систему примитивов.** Потому что в этом случае НЕВОЗМОЖНО БЫСТРО УВИДЕТЬ ГЛАЗАМИ, как эти примитивы логически связаны между собой. Чтобы понять эту связь, нужен трудоемкий мыслительный анализ, который требует усилий, отнимает время и снижает производительность труда.

8. Систему примитивов всегда можно превратить в силуэт. При этом каждый примитив превращается в ветку силуэта. (Иногда часть примитивов превращается в ветки силуэта более низкого уровня на лестнице декомпозиции).

9. В заключение повторим еще раз основные рекомендации.

- Алгоритмы и программы следует изображать как силуэты.
- Сложные алгоритмы и программы следует изображать как силуэты, в которых многократно используются иконы «вставка». Последние, в свою очередь, раскрываются как силуэты и т.д.
- Примитивы рекомендуется не использовать совсем или использовать только в крайних случаях.

ЛИТЕРАТУРА

ГЛАВА 7

1. ГОСТ 19.701–90. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. М.: Изд-во стандартов, 1991.
2. *Лингер Р., Миллс Х., Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982. С. 124–126, 139–146.
3. *Йодан Э.* Структурное проектирование и конструирование программ. М.: Мир, 1979. С. 185–196.

ГЛАВА 8

1. *Ломов Б.Ф.* Эргономические (инженерно-психологические) факторы художественного конструирования. В кн.: Учебно-методические материалы по художественному конструированию. М., 1965.
2. *Венда В.* Предисловие к русскому изданию. В кн.: Боумен У. Графическое представление информации. М.: Мир, 1971. С. 8—14.
3. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию. М.: Мир, 1980. С. 24, 73, 80.
4. *Криницкий Н. А.* Алгоритмы вокруг нас. М.: Наука, 1984. С. 102.
5. *Питерс Л. Дж.* Методы отображения и компоновки программных средств // ТИИЭР. 1980. Т. 68, № 9. С. 60.

ГЛАВА 9

1. Завалова Н.Д., Ломов Б.Ф., Пономаренко В.А. Образ в системе психической регуляции деятельности. М.: Наука, 1986. С. 21.
2. Человеческий фактор. В 6-и т. Т. 6. Эргономика в автоматизированных системах. М.: Мир, 1992. С. 216.

ГЛАВА 10

1. Венда В.Ф. Предисловие к русскому изданию. В кн: У. Боумен. Графическое представление информации. М.: Мир, 1985. С. 5.
2. Йодан Э. Структурное проектирование и конструирование программ. М.: Мир, 1979. С. 252.