

**Владимир
ПАРОНДЖАНОВ**

УЧИСЬ ПИСАТЬ, ЧИТАТЬ И ПОНИМАТЬ АЛГОРИТМЫ

**АЛГОРИТМЫ ДЛЯ ПРАВИЛЬНОГО
МЫШЛЕНИЯ**

Основы алгоритмизации



Москва
2012

УДК 004.438ДРАКОН:004.021
ББК 32.973.26-018.2
П18

Паронджанов В. Д.

П18 Учись писать, читать и понимать алгоритмы. Алгоритмы для правильного мышления. Основы алгоритмизации. – М.: ДМК Пресс, 2012. – 520 с.: ил. 272

ISBN 978-5-94074-800-7

Излагаются новые полезные для практики идеи и достижения, помогающие легко и быстро освоить алгоритмы. Дается систематизированное изложение основных понятий и методов алгоритмизации. Книга содержит общедоступный практический курс, позволяющий существенно ускорить разработку, анализ и проверку алгоритмов, облегчить проектирование сложной деятельности. Ведется наглядное обучение на примерах. Читатель быстро привыкает к самостоятельному осмысленному составлению алгоритмов.

Использованы доходчивые и привлекательные чертежи алгоритмов (дракон-схемы), значительно облегчающие усвоение материала. Книга богато иллюстрирована. Почти триста наглядных схем и рисунков, выполненных по принципу «Посмотрел – и сразу понял!», окажут читателю неоценимую помощь.

Книга предназначена для начинающих и профессионалов, а также для самостоятельного изучения.

УДК 004.438ДРАКОН:004.021
ББК 32.973.26-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-5-94074-800-7

© Паронджанов В. Д., 2012
© Оформление, издание ДМК Пресс, 2012

СОДЕРЖАНИЕ

Введение	6
-----------------------	----------

Часть I. Алгоритмы (облегченный материал для первого знакомства)	13
---	-----------

Глава 1. Алгоритмы – это очень просто!	15
Глава 2. Алгоритмы и процедурные знания	23

Часть II. Алгоритмический язык ДРАКОН и удобные чертежи алгоритмов (дракон-схемы)	35
--	-----------

Глава 3. Иконы и макроиконы языка ДРАКОН	37
Глава 4. Алгоритмическая структура «силуэт»	41
Глава 5. Алгоритмическая структура «примитив»	56
Глава 6. Сравним силуэт и примитив	67
Глава 7. Как улучшить понятность алгоритмов?	78
Глава 8. Простые циклические алгоритмы	113
Глава 9. Особенности циклических алгоритмов	124
Глава 10. Сложные циклические алгоритмы. Структура «цикл в цикле»	140
Глава 11. Логические формулы, используемые в алгоритмах	153
Глава 12. Что такое эргономичный текст?	184
Глава 13. Алгоритмы реального времени	205
Глава 14. Параллельные алгоритмы	222
Глава 15. Дракон-схемы и блок-схемы	242
Глава 16. Коротко о программировании	255

Часть III. Алгоритмы практической жизни (примеры) ...	267
--	------------

Глава 17. Алгоритмы в медицине	269
Глава 18. Алгоритмы в промышленности	277
Глава 19. Алгоритмы в торговле	286

Глава 20. Алгоритмы бухгалтерского учета	293
Глава 21. Алгоритмы в атомной энергетике	301
Глава 22. Алгоритмы в биологии	312
Глава 23. Алгоритмы в сельском хозяйстве	325
Глава 24. Алгоритмы в средней школе.....	331
Глава 25. Алгоритмы государственного и муниципального управления	344
Часть IV. Математические алгоритмы (примеры)	349
Глава 26. Простые математические алгоритмы	351
Глава 27. Алгоритмы с массивами	356
Глава 28. Алгоритмы поиска данных	360
Глава 29. Рекурсивные алгоритмы	365
Часть V. Заключительные рекомендации по созданию дракон-схем.....	369
Глава 30. Рекомендации по использованию алгоритмических структур «силуэт» и «примитив»	371
Глава 31. Как улучшить понятность веток?	377
Часть VI. Конструктор алгоритмов и формальное описание языка	393
Глава 32. Конструктор алгоритмов (помощник человека).....	395
Глава 33. Графический синтаксис языка ДРАКОН	416
Часть VII. Теоретические основы языка ДРАКОН	425
Глава 34. Исчисление икон	427
Глава 35. Метод Ашкрофта-Манн и алгоритмическая структура «силуэт».....	436
Глава 36. Визуальный структурный подход к алгоритмам и программам (шампур-метод)	449
Часть VIII. Какую роль играют алгоритмы в человеческой культуре?	473
Глава 37. Алгоритмическое мышление	475
Глава 38. Алгоритмы и улучшение работы ума	480
Глава 39. Алгоритмическое мышление и две группы людей	488
Глава 40. Как ликвидировать алгоритмическую неграмотность?.....	493
Глава 41. Необходимость культурных изменений	497

Алгоритмы должны быть понятными (вместо заключения)	504
Литература	508
Основная литература по языку ДРАКОН	514
Применение языка ДРАКОН в ракетно-космической отрасли	515
Предметный указатель	516

ВВЕДЕНИЕ

ЧТО МЫ ЗНАЕМ ОБ АЛГОРИТМАХ?

Многие думают, что алгоритмы нужны только программистам и математикам. Это не так. Алгоритмы могут пригодиться всем или почти всем. Начиная от врача и агронома и кончая топ-менеджером.

Почему? Потому что мы живем в мире алгоритмов, хотя зачастую не догадываемся об этом. Современная цивилизация – это цивилизация алгоритмов. Они окружают нас повсюду.

К сожалению, большинство людей не умеют читать, писать и понимать алгоритмы. Впрочем, это дело поправимое. Прочитав книгу, вы быстро получите нужные знания. Как известно, один рисунок стоит тысячи слов. К вашим услугам – четкие, кристально ясные и забавные рисунки. Сделанные так, чтобы читатель «посмотрел – и сразу понял!». Они помогут легко открыть заветную дверь в увлекательное царство алгоритмов.

В ЧЕМ ПРОБЛЕМА?

Трудность в том, что алгоритмы, как правило, очень сложны. Понять их не просто. Нужно изрядно попотеть, затратить много труда и времени.

А нельзя ли найти обходную дорогу и сэкономить время?

Конечно, можно! Секрет в том, что алгоритмы надо сделать *дружелюбными*. Это позволит превратить головоломки в наглядные алгоритмы-картинки, обеспечивающие быстрое и глубокое понимание. Глубина понимания сложных проблем – как раз то, чего всем нам (от студента до министра) ой как не хватает!

Почему алгоритмы трудны для понимания? Потому, что существующий способ записи алгоритмов (принятый во всем мире) выбран неудачно. Он устарел и превратился в досадное препятствие. Он удовлетворяет требованиям математической строгости, но не учитывает требования науки о человеческих факторах – эргономики. Этот устаревший способ упускает из виду психофизиологические характеристики человека-алгоритмиста. И тем самым затрудняет и замедляет работу с алгоритмами.

ЛЕГКИЕ ДЛЯ ПОНИМАНИЯ И УДОБНЫЕ ДЛЯ РАБОТЫ

В книге излагается новый взгляд на алгоритмы. Мы покажем, что алгоритмы должны быть не только правильными, *но и дружелюбными*. То есть легкими для понимания и удобными для работы.

Этой благородной цели служат *эргономичные* алгоритмические языки. Они создают повышенный интеллектуальный комфорт, увеличивают продуктивность труда. С их помощью вы научитесь легко и быстро, затратив минимум усилий, решать сложнейшие проблемы. Вы сможете проектировать сложную деятельность и бизнес-процессы. Формализовать свои профессиональные знания. И составлять алгоритмы самостоятельно, без помощи программистов.

Алгоритмы – важная часть человеческой культуры. Умение составлять алгоритмы позволяет улучшить работу ума. Несколькими преувеличивая, можно сказать: «Алгоритмы – вторая грамотность!».

БЛОК-СХЕМЫ АЛГОРИТМОВ

Существует несколько способов для записи алгоритмов. Широкую известность получили блок-схемы алгоритмов. Они используются в системе образования для первоначального ознакомления с алгоритмами (см. например, [1–3]). Международная организация стандартизации *ISO* выпустила стандарт на блок-схемы *ISO 5807–85* [4]. Ему соответствует отечественный стандарт *ГОСТ 19.701–90* [5].

Тем не менее, многие специалисты убеждены, что блок-схемы – это вчерашний день. Можно сказать, что блок-схемы постигла печальная участь. Истина в том, что они обладают поистине удивительными достоинствами, которые остаются нераскрытыми. Без преувеличения можно сказать, что блок-схемы алгоритмов – это бесценный бриллиант, который по воле мачехи-судьбы до сих пор не огранен и не вставлен в золотую оправу.

Хотя имеются отдельные попытки приспособить блок-схемы к современным нуждам (язык *UML* и др.), однако в целом блок-схемы явно оказались на обочине бурно развивающегося процесса визуализации алгоритмов. А их громадные потенциальные возможности фактически не используются.

Данная книга посвящена блок-схемам. В ней изложен *новый подход* к решению проблемы блок-схем. Наш труд можно рассматривать как дальнейшее развитие плодотворной идеи изобретателя блок-схем Джона фон Неймана.

Мы покажем, что именно блок-схемы могут придать алгоритмам новую чарующую силу, небывалую наглядность и другие полезные свойства.

ДРАКОН-СХЕМЫ И ЯЗЫК «ДРАКОН»

Блок-схемы нужно значительно улучшить, сделать математически строгими и эргономически привлекательными. Во избежание путаницы мы присвоили новым блок-схемам название «дракон-схемы».

Дракон-схемы отличаются от традиционных блок-схем, как небо от земли. Они позволяют построить семейство дружелюбных алгоритмических языков под общим названием ДРАКОН (Дружелюбный Русский Алгоритмический язык, Который Обеспечивает Наглядность) [6, 7].

Визуальный язык ДРАКОН обладает уникальными эргономическими характеристиками. Он позволяет легко и быстро создавать дружелюбные и наглядные алгоритмы.

ДРАКОН – оружие интеллекта. Он помогает ясно и четко мыслить. И значительно облегчает творческий процесс создания алгоритмов, делая его доступным для широкого круга работников.

КТО РАЗРАБОТАЛ ЯЗЫК ДРАКОН?

Язык ДРАКОН разработан совместными усилиями Федерального космического агентства России (Научно-производственный центр автоматизации и приборостроения им. академика Н. А. Пилюгина, г. Москва) и Российской академии наук (Институт прикладной математики им. академика М. В. Келдыша, г. Москва).

ДРАКОН РОДИЛСЯ В КОСМИЧЕСКОЙ КОЛЫБЕЛИ

ДРАКОН возник как обобщение опыта работ по созданию космического корабля «Буран». На базе ДРАКОНа построена автоматизированная Технология проектирования алгоритмов и программ под названием «ГРАФИТ-ФЛОКС». Она успешно используется во многих крупных ракетно-космических проектах: «Морской старт», «Фрегат», «Протон-М» и др.

ДРАКОН В СИСТЕМЕ ОБРАЗОВАНИЯ

В 1996 году Государственный комитет по высшему образованию Российской Федерации включил изучение языка ДРАКОН в программу курса информатики высшей школы. Этот факт нашел отражение в официальном документе Госкомвуза под названием:

«Примерная программа дисциплины «Информатика» для направлений:

510000 – Естественные науки и математика

540000 – Образование

550000 – Технические науки

560000 – Сельскохозяйственные науки

Издание официальное.

М.: Госкомвуз, 1996. – 21 с.

Авторы программы: Кузнецов В. С., Падалко С. Н., Паронджанов В. Д., Ульянов С. А.

Научные редакторы: Падалко С. Н., Паронджанов В. Д.» [8].

Примечание. Этот документ официально узаконил изучение языка ДРАКОН в системе образования.

ЧТО ТАКОЕ ЭРГОНОМИЧНЫЙ?

В этой книге важную роль играет слово «эргономичный». Что оно означает? Чтобы не тратить лишних слов, мы сочиним краткий словарь, который, хотя и нарушает все каноны научной строгости, зато вполне понятен новичкам.

Словарик

Эргономика – наука о том, как превратить сложную и трудную работу в простую, легкую и приятную.

Когнитивная эргономика – наука о том, как облегчить и улучшить умственную работу.

Эргономичный – дружелюбный, легкий для понимания, удобный для работы.

Эргономичный алгоритм – дружелюбный, легкий для понимания алгоритм.

Эргономичный алгоритмический язык – дружелюбный, легкий для понимания и удобный в работе алгоритмический язык.

Эргономизация алгоритма – превращение недружелюбного алгоритма в дружелюбный.

Эргономизация алгоритмического языка – превращение недружелюбного алгоритмического языка в дружелюбный.

ЦЕЛЬ КНИГИ

Наша цель – научить читателя самостоятельно создавать дружелюбные, то есть легкие для понимания алгоритмы. И показать, что это простое и даже приятное дело. На многочисленных примерах читатель убедится, что дружелюбные алгоритмы имеют огромные преимущества.

Чем понятнее алгоритм, тем легче уяснить его смысл. Чем прозрачнее смысл, тем лучше взаимопонимание между людьми. Чем меньше ошибок, тем выше производительность труда при разработке алгоритмов. Чем меньше усилий затрачивают алгоритмисты, тем быстрее они выполняют свою работу.

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

В этой книге мы не будем касаться вопросов программирования. И сосредоточим внимание исключительно на алгоритмах.

Алгоритмизация и программирование – разные вещи. Подчеркнем главное.

Число людей, которым необходимо знать алгоритмы, во много раз превышает число людей, которым надо знать программирование.

Разумеется, современное общество нуждается в программистах. Но обучение программистов должно быть не массовым, а экономически обоснованным.

И наоборот, обучение алгоритмизации должно быть очень широким или даже массовым. Потому что алгоритмы нужны не только программистам, но и многим другим людям.

В обществе знаний во многих случаях возникает необходимость формализовать собственные процедурные профессиональные знания специалистов. Отсюда проистекает

Вывод. Умение читать, писать и понимать алгоритмы – настоятельная необходимость. Такое умение должно стать частью профессиональной культуры специалистов почти всех специальностей. Эта мысль проходит красной нитью через всю книгу.

СОДЕРЖАНИЕ КНИГИ

Книга состоит из восьми частей.

Часть I (главы 1, 2) носит вводный характер. Приводятся забавные примеры алгоритмов, описывающих бытовую человеческую деятельность.

В части II (главы 3–16) изложен эргономичный алгоритмический язык ДРАКОН. Рассмотрены графический алфавит языка, алгоритмические структуры «силуэт» и «примитив». Представлены математические методы, позволяющие улучшить понятность алгоритмов. Показан богатый ассортимент визуальных (графических) циклических алгоритмов, визуальная логика, системы реального времени и параллельные алгоритмы. Для удобства читателя изложение основных идей дается на наглядных примерах.

Часть III (главы 17–25) содержит большое число алгоритмов на языке ДРАКОН, взятых из практической жизни. Примеры демонстрируют универсальность языка, показывают широкий спектр его возможностей для различных отраслей и предметных областей. Сюда относятся медицина, промышленность, сельское хозяйство, торговля и многое другое.

В части IV (главы 26–29) читатель вкратце знакомится с математическими алгоритмами. Примеры демонстрируют работу с массивами, поиск данных и др.

В части V (главы 30, 31) даются заключительные рекомендации по созданию дракон-схем. Даны рекомендации по использованию алгоритмических структур «силуэт» и «примитив». Описывается метод дробления веток, позволяющий улучшить понятность алгоритмической структуры «силуэт».

В части VI (главы 32, 33) описывается компьютерная программа «конструктор алгоритмов». Она представляет собой рабочий инструмент, помогающий человеку придумывать и конструировать алгоритмы. Приводится формальное описание языка ДРАКОН.

Часть VII (главы 34–36) – наиболее сложная часть книги. Здесь даны теоретические основы языка ДРАКОН.

Часть VIII (главы 37–41) посвящена социальным, гуманитарным и культурным аспектам алгоритмизации. Обсуждается вопрос: как ликвидировать алгоритмическую неграмотность?

ГДЕ СКАЧАТЬ КОНСТРУКТОР АЛГОРИТМОВ?

Ответ дан на стр. 414.

В ДОБРЫЙ ПУТЬ!

Прочитав книгу, читатель сможет легко ориентироваться в империи алгоритмов. И убедиться, что язык ДРАКОН – удобное средство, помогающее создавать наглядные и понятные алгоритмы.

Конструктор алгоритмов – надежный помощник человека. Он умело подсказывает, как нужно составлять алгоритмы. Кроме того, он осуществляет 100%-е автоматическое доказательство правильности дракон-схем, гарантируя принципиальную невозможность ошибок визуального синтаксиса. Иными словами, конструктор алгоритмов полностью исключает ошибки графического синтаксиса.

Безошибочное проектирование графики дракон-схем – важный шаг вперед, повышающий производительность труда при практической работе.

По мнению специалистов, управляющая графика ДРАКОНа является мощным инструментом, причем ее мощь легка в освоении и легко применима на практике.

Часть I

**АЛГОРИТМЫ
(ОБЛЕГЧЕННЫЙ МАТЕРИАЛ
ДЛЯ ПЕРВОГО ЗНАКОМСТВА)**

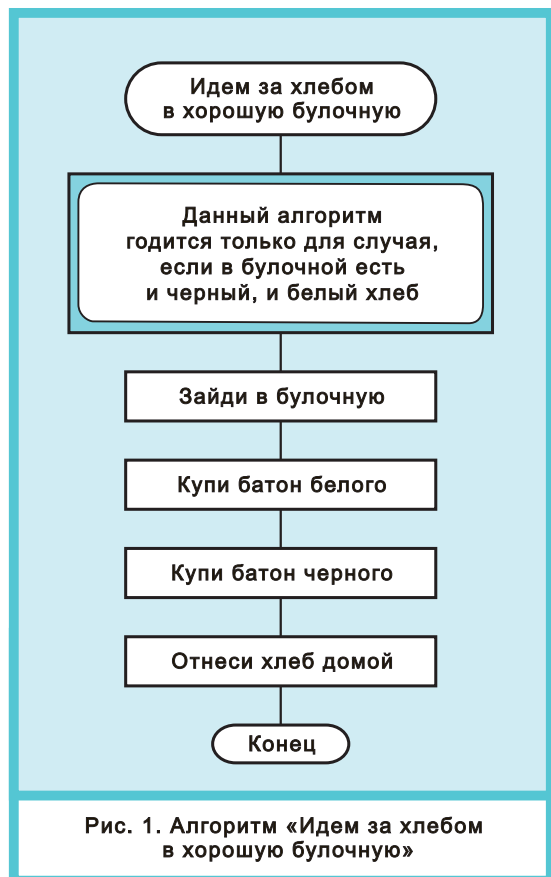
АЛГОРИТМЫ – ЭТО ОЧЕНЬ ПРОСТО!

§1. ПЕРВОЕ ЗНАКОМСТВО С АЛГОРИТМОМ

Предположим, мать говорит сыну: «Сходи в булочную. купи батон белого и батон черного». Слова матери – алгоритм, показанный на рис. 1. Данный алгоритм нельзя признать удачным. Он не отвечает на вопрос: что делать, если в магазине нет хлеба.

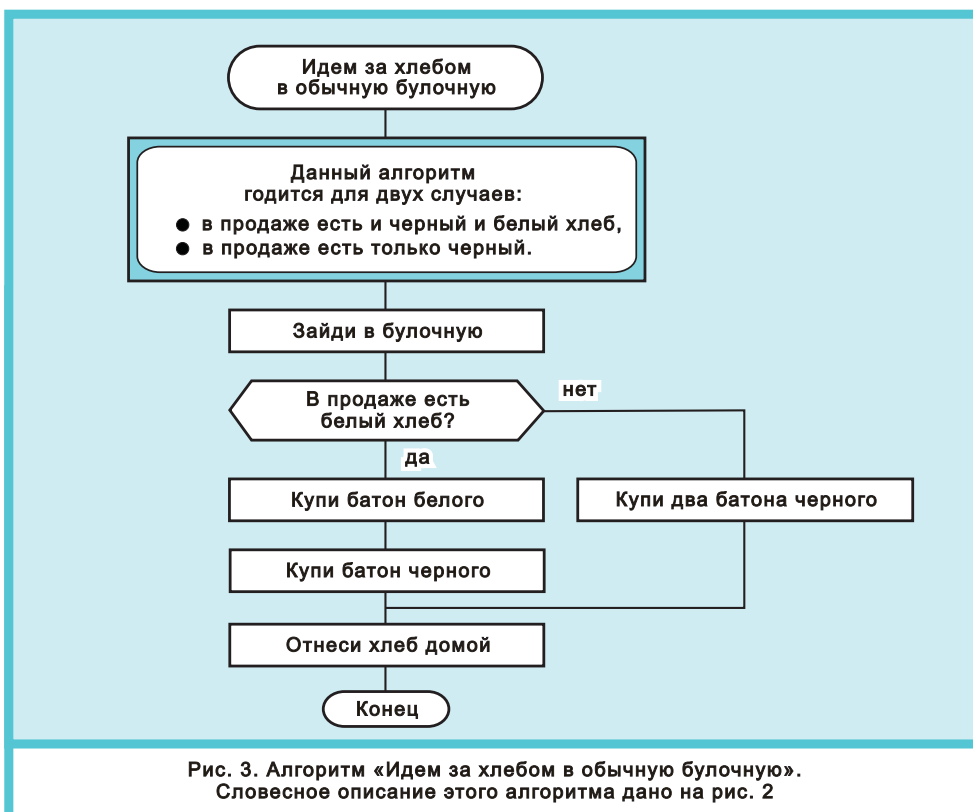
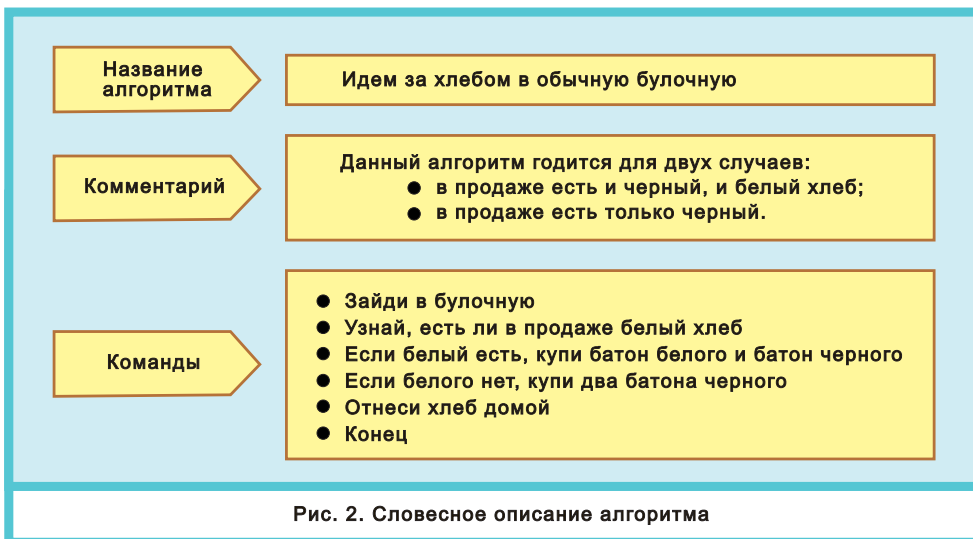
Чтобы исправить оплошность, нужно придумать другой алгоритм, который учитывает реальные условия. Предположим, в булочной черный хлеб водится всегда, а с белым случаются перебои. В таком случае мать могла бы сказать: «Купи батон белого и батон черного. Если белого не будет, возьми два черного». Словесная формулировка этого алгоритма показана на рис. 2, а блок-схема – на рис. 3.

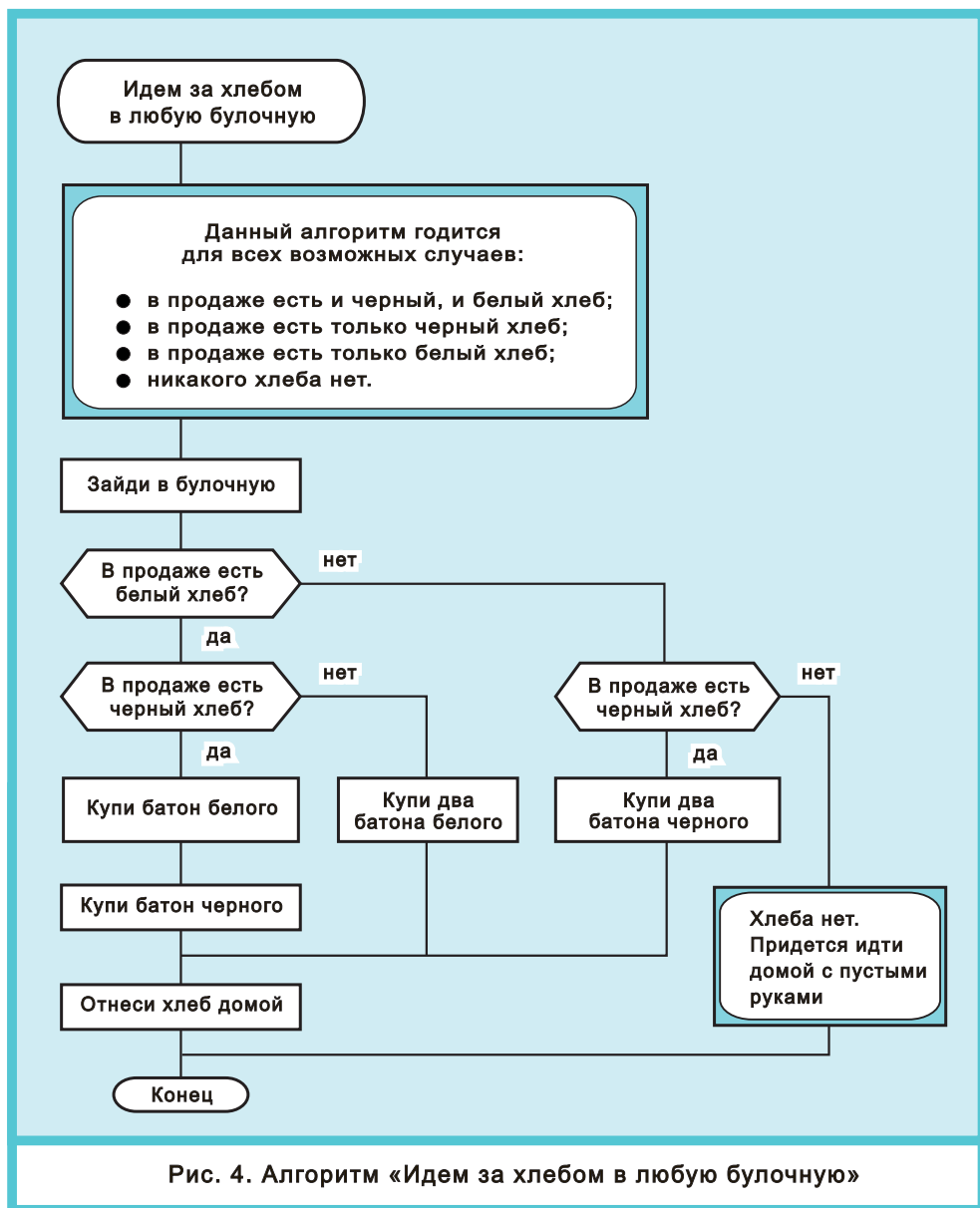
К последнему алгоритму тоже можно придаться. Как быть, если в продаже нет черного хлеба? Чтобы учесть все варианты, мать должна дать сыну более сложную инструкцию: «Сходи за хлебом. Один батон белого и один черного.



Белого не будет – купи два черного. И наоборот. А если никакого нет, обойдемся».

Соответствующая блок-схема показана на рис. 4.





§2. СРАВНЕНИЕ АЛГОРИТМОВ

Взглянем на три алгоритма на рис. 1, 3, 4. Какой из них следует считать хорошим, а какой – плохим? Алгоритм считается хорошим, если он правильно работает при всех реальных условиях. В противном случае алгоритм считается плохим.

Применим это правило к нашим алгоритмам. Легко видеть, что только один алгоритм (рис. 4) будет работать при любых условиях.

В самом деле, если в булочной есть и белый, и черный хлеб, воображаемый «бегунок» побежит от начала к концу алгоритма по левой вертикальной линии.

Если белый есть, а черного нет, бегунок помчится чуть правее – через икону «Купи два батона белого».

Если белого нет, а черный есть, бегунок побежит еще правее – через икону «Купи два батона черного».




Наконец, если в булочной вообще нет хлеба, бегунок выберет крайний правый путь – через икону «Хлеба нет. Придется идти домой с пустыми руками».

Что же мы узнали? Алгоритм на рис. 4 – хороший алгоритм, так как он выполняет свою работу *при любых условиях*. И наоборот, алгоритмы на рис. 1 и 3 – это плохие, «неполноценные» алгоритмы.

Можно сказать и по-другому. Алгоритм на рис. 4 – правильный, безошибочный, работоспособный алгоритм. А схемы на рис. 1 и 3 не отвечают на вопрос: что делать, если в булочной нет хлеба. Значит, эти алгоритмы неработоспособны; они содержат ошибку. На рис. 4 ошибка исправлена.

§3. ЧТО ТАКОЕ ДРАКОН-СХЕМЫ? ЗАЧЕМ НУЖНЫ ИКОНЫ?

Блок-схемы, нарисованные по правилам языка ДРАКОН, называются *дракон-схемы*. Они состоят из простых рамок – *икон*, соединенных между собой. На рис. 1 таких икон семь. Верхняя называется «заголовок». Та, что пониже – «комментарий». Следующие четыре – «действие». А самая нижняя – «конец».

Что такое «Заголовок»		Овальная икона, расположенная в самом начале схемы. Внутри нее пишут название алгоритма
Что такое «Действие»		Прямоугольная икона, внутри которой пишут команды алгоритма
Что такое «Комментарий»		Прямоугольная икона с двойной рамкой, внутри которой пишут не команды, а пояснения

Что такое
«Конец»



Маленькая овальная икона, расположенная в самом конце алгоритма. Внутри нее пишут слово «Конец»

§4. АЛГОРИТМ

Каждый человек, находясь на работе, ведет себя отнюдь не случайно. Он стремится к четко определенной цели и выполняет нужные для этого действия. Таким образом, любую работу (за исключением сложных творческих операций) можно рассматривать как хорошо определенный каскад действий, то есть алгоритм. Нетрудно заметить, что алгоритмы окружают нас повсюду.

Алгоритм – последовательность шагов, ведущих к цели. Можно сказать и по-другому. Алгоритм – последовательность команд, помогающих решить задачу.

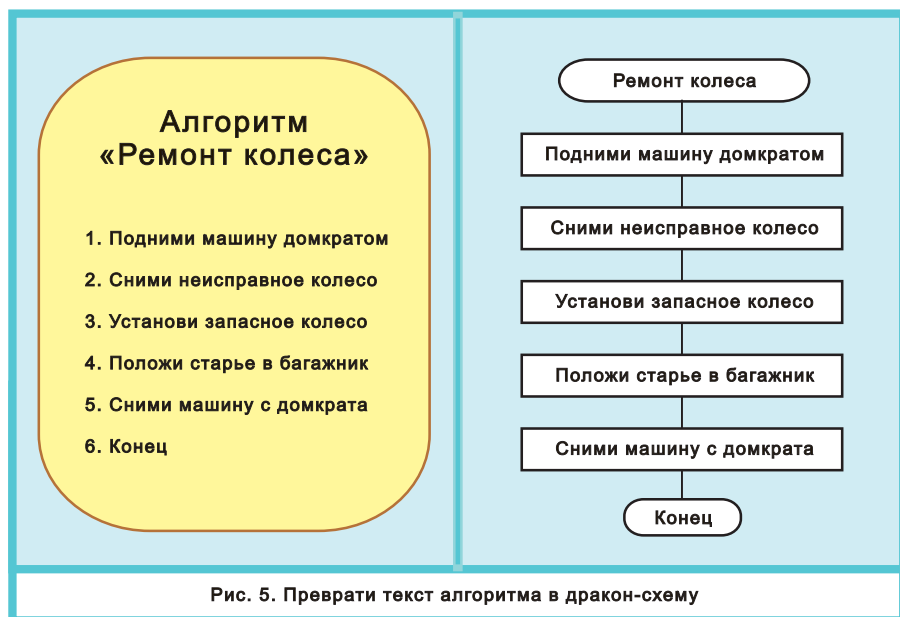
Что такое
алгоритм

Это последовательность действий, ведущих к поставленной цели

Команды алгоритма можно записать по-разному, например:

- в виде дракон-схемы;
- в виде текста.

Чтобы убедиться в этом, взгляните на рис. 5.



§5. КАЖДАЯ КОМАНДА АЛГОРИТМА ДОЛЖНА ВЫПОЛНЯТЬСЯ ТОЧНО И ОДНОЗНАЧНО

Рассмотрим алгоритм на рис. 6. Предположим, алгоритм выполняет робот.

Робот – типичный буквоед. Обычно педантов и буквоедов не любят. Но в мире алгоритмов свои законы. Там буквоеды в большом почете. Чтобы робот понял алгоритм, каждая команда должна быть написана «побуквоедски». Точно и однозначно.

На рис. 6 алгоритм написан именно так – точно и однозначно. Там ясно сказано – надо купить килограмм картошки и кочан капусты. Это значит, что любые другие покупки запрещены.

А что мы видим на рис. 7? Тут ситуация иная. Команда «Купи продукты» – плохая, неоднозначная команда. Ее невозможно исполнить. Потому что нигде не сказано, какие именно продукты нужно купить.

Такие умолчания в алгоритмах недопустимы. Они рассматриваются, как ошибки.

Выполните упражнение на рис. 8.

§6. ИКОНА «ВОПРОС»

В верхней части рис. 9 изображена икона «вопрос». Она называется так потому, что внутри нее пишут «да-нетный вопрос». То есть вопрос, на который можно ответить либо «да», либо «нет». Все другие ответы запрещены.



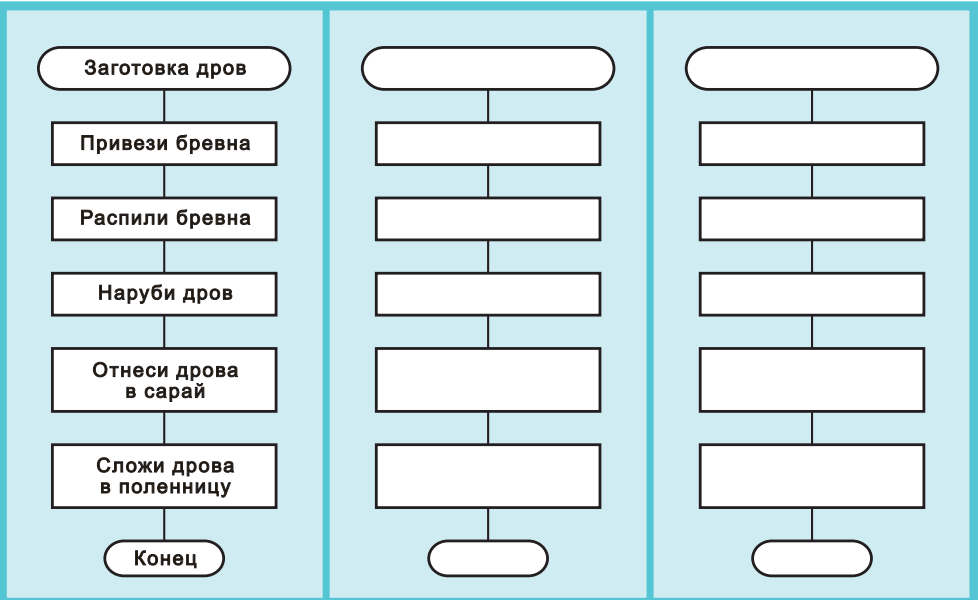


Рис. 8. Придумай два алгоритма по образцу, показанному слева

Вот примеры да-нетных вопросов: утюг сломался? Вася купил хлеб? Поезд пришел? Преступника арестовали? Тетя приехала? «Спартак» выиграл? Эта лужа больше, чем та? На улице температура выше нуля?

Икона «вопрос» имеет один вход сверху и два выхода: вниз и вправо. Выход влево запрещен и никогда не используется.

Бывают случаи, когда нужно выбрать одно действие из двух. В этом случае удобно использовать икону «вопрос». При ответе «да» выполняется одно действие, при «нет» – другое. Икона «вопрос» нужна, чтобы сделать в алгоритме развилку. В этом легко убедиться, взглянув на рис. 9.

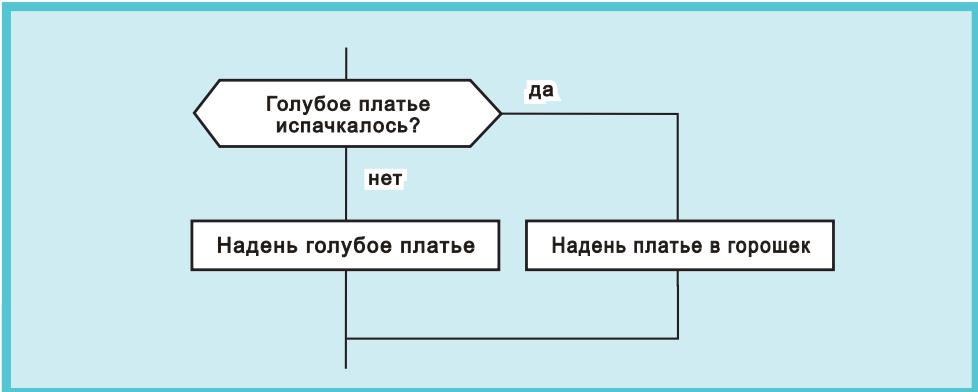
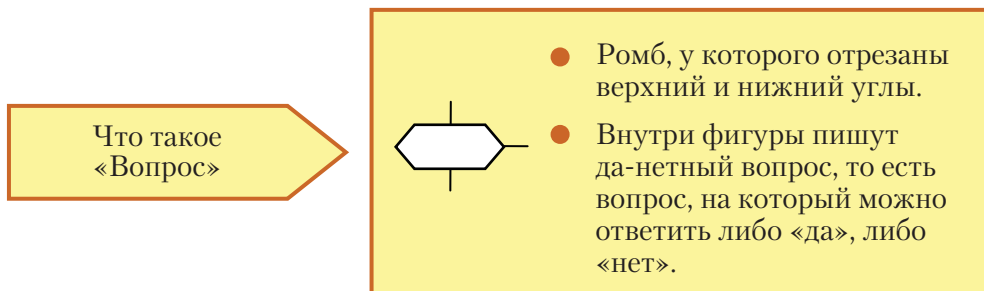


Рис. 9. История с голубым платьем

Иногда в алгоритме надо сделать не одну, а много развилок. Для этого используют несколько икон «Вопрос» (рис. 4).



§6. ВЫВОДЫ

1. Умение выразить свою мысль в виде алгоритма имеет много преимуществ. Такое умение пригодится почти каждому – ведь оно делает интеллект более гибким и эффективным.
2. Алгоритмы играют огромную роль в жизни общества. Они оказывают заметное влияние на эффективность экономики, уровень жизни, а также на умственное развитие цивилизации.

АЛГОРИТМЫ И ПРОЦЕДУРНЫЕ ЗНАНИЯ

§1. ПРОЦЕДУРНЫЕ И ДЕКЛАРАТИВНЫЕ ТЕКСТЫ

Давайте совершим экскурсию в библиотеку. На полках стоят тысячи книг. Толстых и тонких. Художественных и научных. Всяких.

Все тексты в книгах можно разделить на две части:

- процедурные,
- декларативные.

Чтобы уяснить, чем они различаются, обратимся к таблице 1.

Таблица 1

Это процедурный текст	Это декларативный текст
Скинь мантилью, ангел милый, И явись, как яркий день! Сквозь чугунные перилы Ножку дивную продень!	По синим волнам океана, Лишь звезды блеснут в небесах, Корабль одинокий несется, Несется на всех парусах.

Текст в левом столбце при желании можно превратить в набор команд:

1. Скинь мантилью.
2. Явись, как ясный день.
3. Продень ножку сквозь перила.

Текст в правом столбце такому преобразованию не поддается. Его невозможно превратить в команды.

Рассмотрим еще один пример (табл. 2).

Таблица 2

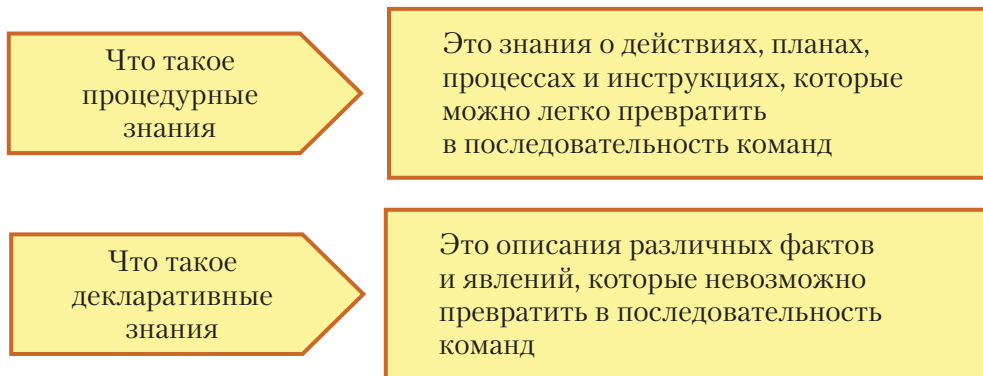
Это процедурный текст	Это декларативный текст
Сходи к бабушке, возьми у нее телевизор и отнеси в ремонт.	Телевизор был старый, покрытый толстым слоем пыли. Видимо, им уже давно не пользовались.

Левый текст без труда превращается в команды:

1. Сходи к бабушке.
2. Возьми у нее телевизор.
3. Отнеси его в ремонт.

Правый текст имеет принципиально иную логическую структуру. Как ни старайся, как ни хитри, а в команды его не превратишь!

Если текст содержит достоверные и обоснованные сведения, такие сведения называются знанием.



§2. КАК ПРЕВРАТИТЬ ПРОЦЕДУРНОЕ ЗНАНИЕ В АЛГОРИТМ?

В верхней части рис. 10 находится текст, содержащий процедурное знание. Это знание можно превратить в дракон-схему (алгоритм).

Пробежим взглядом по схеме сверху вниз. Мы обнаружим, что алгоритм нарисован не хаотично, а упорядоченно. Тут действует

Правило. Чем ниже нарисована икона, тем позже выполняется записанная в ней команда. (Для краткости: «Чем ниже – тем позже»).

Данное правило выполняется во всех алгоритмах.

Здесь нас поджидает сюрприз. Переведем взгляд со схемы на исходный текст. Налицо разительный контраст – в тексте данное правило не соблюдается! Таким образом, алгоритм сильно отличается от исходного текста. В чем отличие?

А вот в чем – порядок записи команд коренным образом изменился. Это хорошо видно на рис. 11.

- Первая команда текста (*Сходи к бабушке*) в алгоритме стала не первой, а второй.
- Вторая команда текста (*возьми у нее телевизор*) переехала совсем в другое место.

- Третья команда текста (*и отнеси его в ремонт*) стала предпоследней.
- Четвертая команда текста (*На случай, если бабушки не будет дома*) сохранила свое место – осталась четвертой.
- И наконец, самое удивительное! Последняя команда текста (*возьми с собой ключ от ее квартиры*) перескочила из конца в начало. В алгоритме она стала самой первой.

Как можно объяснить эти чудесные превращения? Ответ довольно прост.

Исходный текст, представленный на рис. 10 и 11, написан на естественном языке. В этом языке закон «Чем ниже – тем позже» не действует. А в алгоритме ситуация обратная. Там этот закон – святая святых и обязательно исполняется.

Отсюда вытекает

Правило преобразования процедурного текста в алгоритм

Чтобы превратить процедурное знание, записанное на естественном языке, в дракон-схему, необходимо:

- расчленить исходный текст на фрагменты, чтобы в каждом фрагменте содержалась одна команда;
- записать команды в виде упорядоченной последовательности согласно принципу «Чем ниже – тем позже»;
- добавить иконы «заголовок» и «конец»;
- при необходимости внести дополнительные команды, устраняющие пробелы исходного текста и превращающие дракон-схему в логически законченный алгоритмический рассказ.

Примечание. На рис. 10 и 11 дополнительными командами являются:

- Позвони в дверь.
- Открой дверь своим ключом.
- Войди в квартиру.

§3. МАТЕМАТИЧЕСКИЕ И «БЫТОВЫЕ» АЛГОРИТМЫ

Во многих учебниках описаны математические алгоритмы. Это хорошо, но мало. Разумеется, студенты должны назубок знать математику. Но в жизни бывают неожиданности. Некоторые студенты, овладев математикой, затрудняются составлять алгоритмы, в которых отсутствуют математические формулы.

Чтобы избежать подобных неприятностей, студенты должны уметь распознавать «нематематические» алгоритмы, которые постоянно встречаются в жизни.

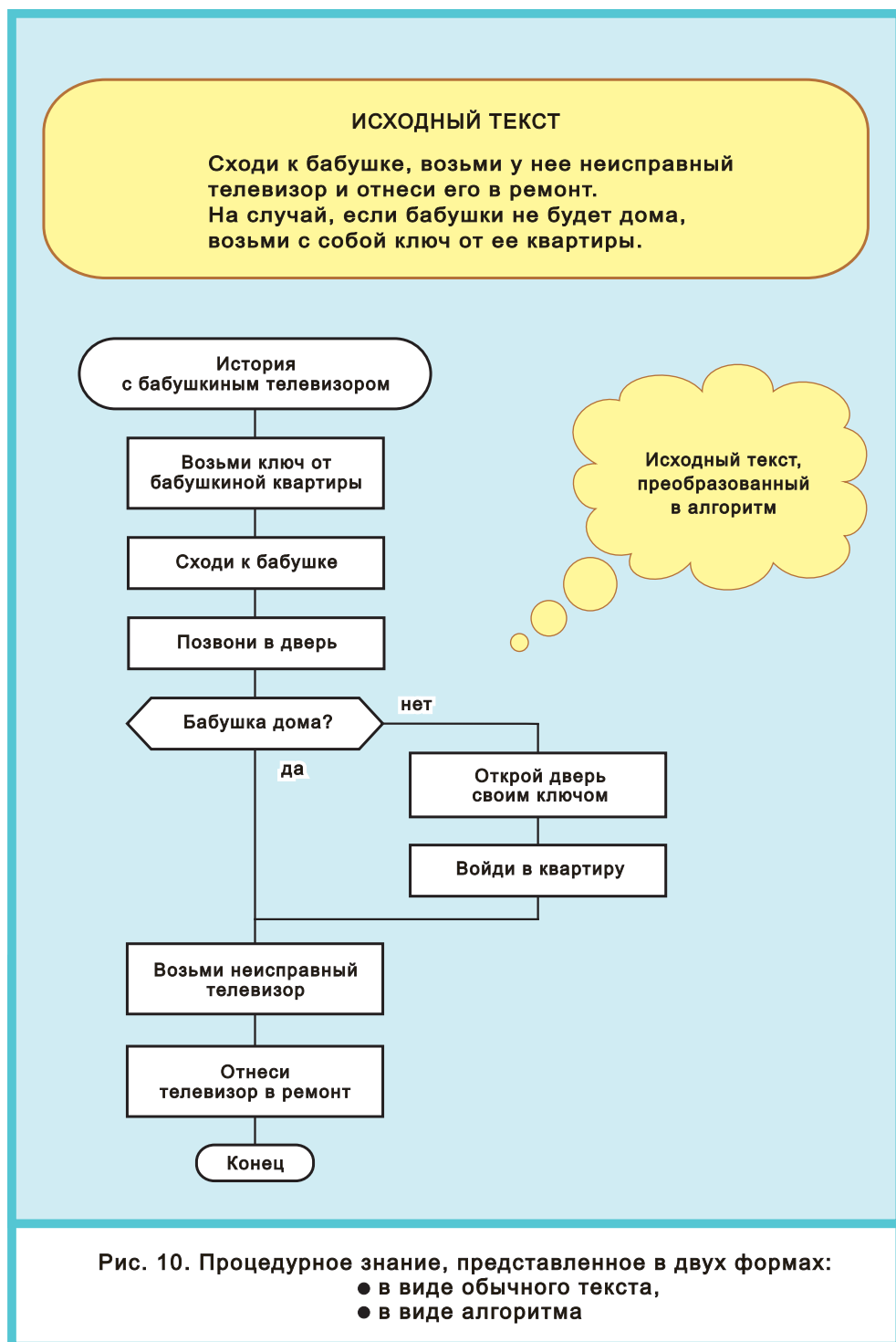


Рис. 10. Процедурное знание, представленное в двух формах:

- в виде обычного текста,
- в виде алгоритма

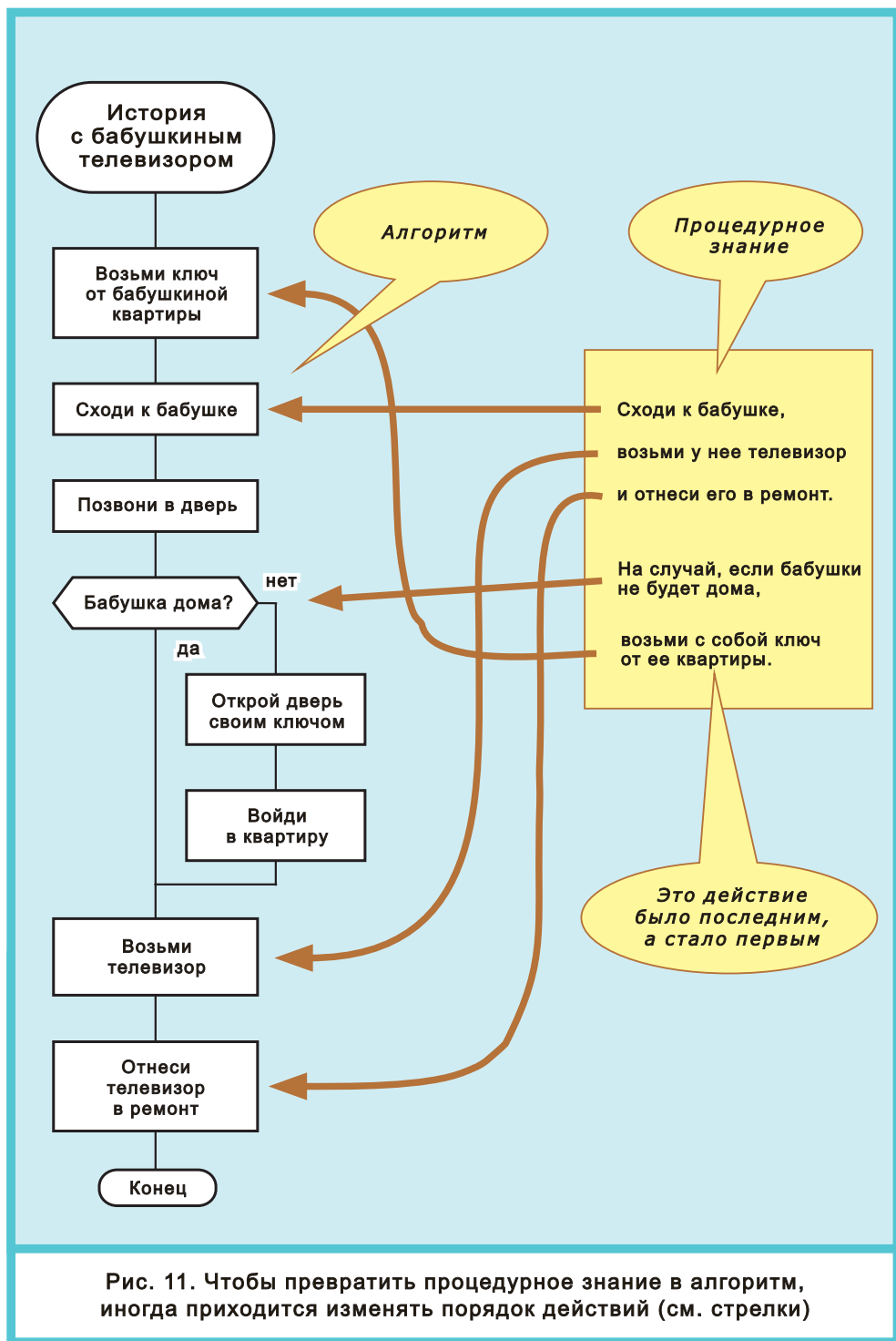


Рис. 11. Чтобы превратить процедурное знание в алгоритм, иногда приходится изменять порядок действий (см. стрелки)

Такие алгоритмы можно условно назвать «бытовыми». Приведем пример. Перед нами стихотворение Генриха Гейне.

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

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

Это стихотворение легко превращается в алгоритм (рис. 12). Там же виден кусочек декларативного знания, которое записано в иконе «комментарий» – («Я догадаюсь сам»).

§4. ЧТО ТАКОЕ ПЕРЕКЛЮЧАТЕЛЬ?

Схема на рис. 13 позволяет сделать в алгоритме развилку на три направления. Для этого используются две иконы «вопрос». Однако задачу можно решить и по-другому – с помощью переключателя (рис. 14).

Переключатель – разветвление алгоритма на несколько тропинок, которые потом сливаются в одну. Переключатель – сложная структура. Она строится из простых кирпичиков. Такими кирпичиками служат иконы «выбор» и «вариант». Зачем они нужны?

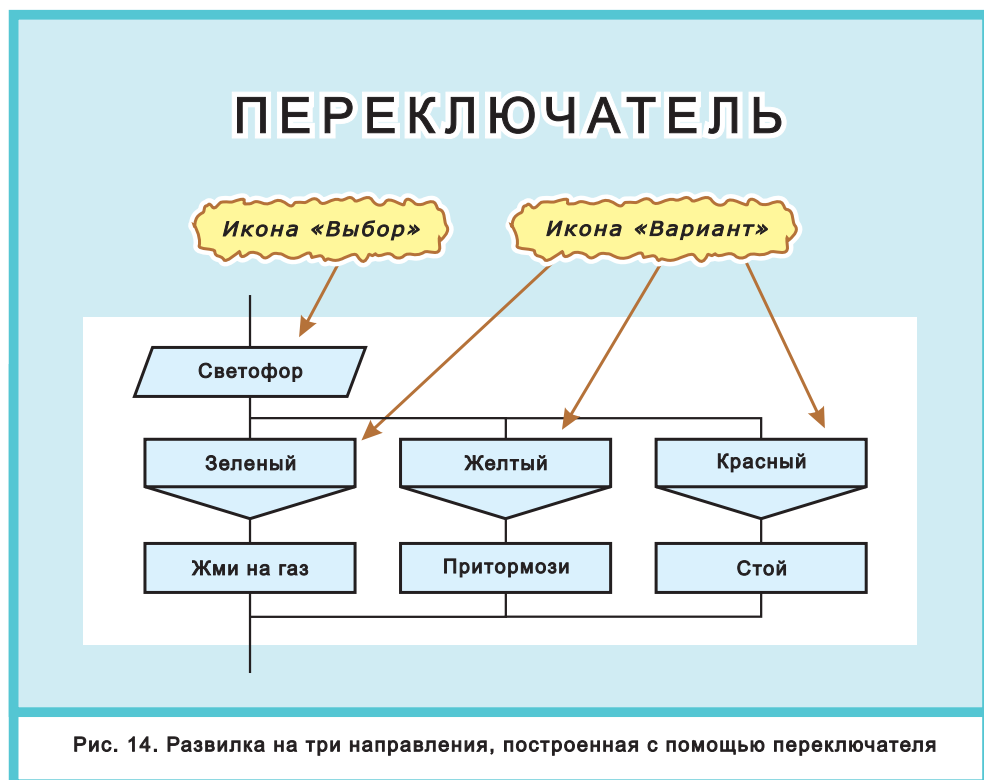
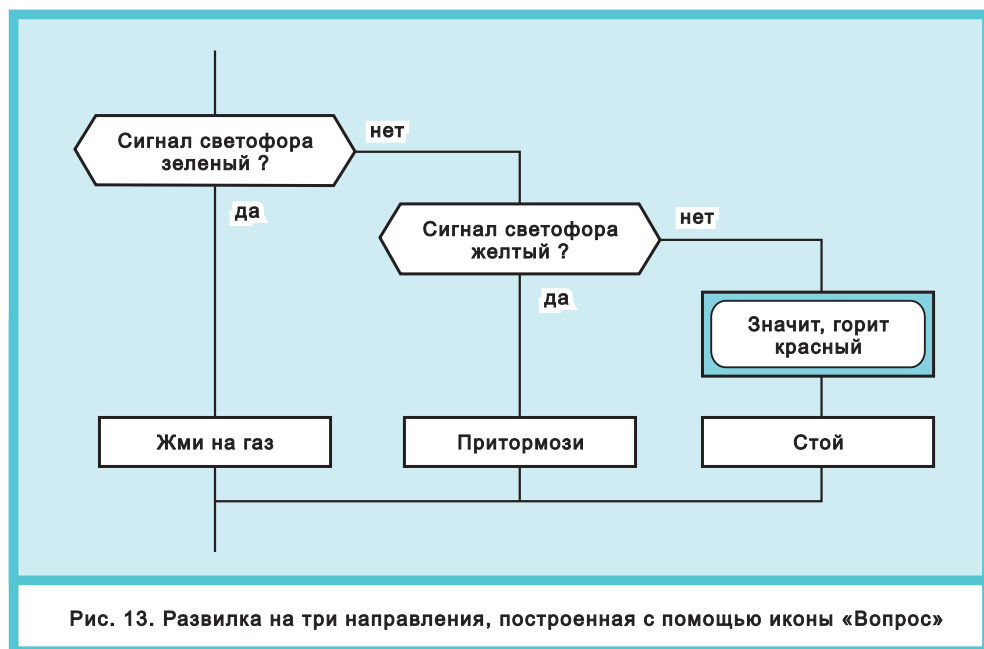
Икона «выбор» содержит вопрос, имеющий несколько ответов. Каждый ответ пишут в отдельной рамочке – иконе «вариант».

Пример 1. На рис. 12 в иконе «вариант» записан вопрос: «Какую дочь Дункан выдает замуж?». На этот вопрос есть два ответа, записанные в иконах «вариант»:

- чернуюбровую;
- с русою косой.



Рис. 12. Стихотворение Гейне, преобразованное в алгоритм



Пример 2. На рис. 14 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?». На этот вопрос есть три ответа:

- зеленый,
- желтый,
- красный.

Попробуем описать, как работает переключатель. Описание всегда начинается со слова «если». Вот примеры.

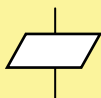
- Если Дункан выдает замуж чернобровую дочь – лети во весь опор назад.
- Если Дункан выдает замуж дочь с русою косой – ступай на рынок, купи веревку, вернись шагом и молчи (рис. 12).
- Если светофор зеленый – жми на газ.
- Если светофор желтый – притормози.
- Если светофор красный – стой (рис. 14).

Дополнительный пример показан на рис. 15.

Что такое
переключатель

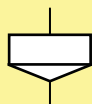
- Это часть алгоритма, имеющая один вход и один выход, внутри которой алгоритм разветвляется на несколько дорожек
- Переключатель строится с помощью икон «выбор» и «вариант»

Что такое
«выбор»



- Икона, которую рисуют в начале переключателя
- В ней пишут вопрос, имеющий два и более ответов

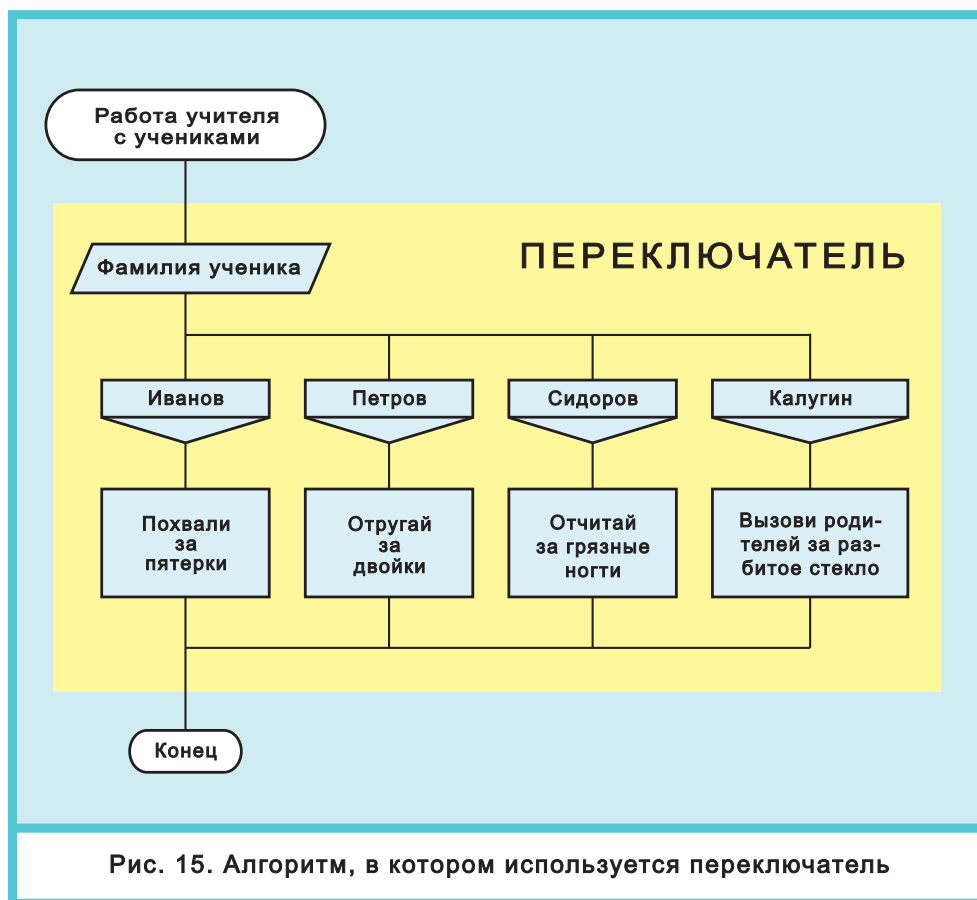
Что такое
«вариант»



Икона переключателя, в которой пишут ответ на вопрос

§5. БЛОК-СХЕМЫ АЛГОРИТМОВ

Блок-схема – чертеж алгоритма, описывающий последовательность действий с помощью графических фигур и соединительных линий. Это поучительная часть истории алгоритмов.



На рис. 16 показаны наиболее употребительные фигуры, используемые при вычерчивании блок-схем. Они соответствуют международному стандарту ISO 5807–85 и отечественному ГОСТ 19.701-90.

Хотя стандарты на блок-схемы считаются действующими, фактически они давно устарели.

С появлением дракон-схем блок-схемы полностью потеряли свое значение, так как они во всех отношениях уступают дракон-схемам. (Подробнее см. главу 15).

§6. АНАЛОГИ ДРАКОН-СХЕМ

Аналогами дракон-схем являются блок-схемы алгоритмов, диаграммы Насси-Шнейдермана, псевдокод (язык описания алгоритмов) и др. Другим аналогом дракон-схем служат диаграммы поведения (*behaviour diagrams*)

Фигура	Название фигуры	Пояснение
	Терминатор	Начало или конец алгоритма
	Процесс	Действие или последовательность действий, обработка данных любого вида
	Решение	Блок проверки условия, имеющий один вход и несколько альтернативных выходов, только один из которых может быть активирован после выполнения условия
	Предопределенный процесс	Процесс, состоящий из одной или нескольких операций, которые определены в другом месте (в подпрограмме, в модуле)
	Ввод или вывод данных	
	Начало цикла	Условие для инициализации, приращения, завершения помещаются внутри верхней или нижней фигур в зависимости от расположения операции, проверяющей условие
	Конец цикла	
	Подготовка	Модификация команды или группы команд с целью воздействия на некоторую последующую функцию
	Соединитель	Показывает выход в другую часть алгоритма или вход из другой части алгоритма. Обозначает обрыв линии и продолжение ее в другом месте
	Комментарий	Описательные комментарии или пояснительные записи в целях объяснения или примечания

Рис. 16. Блок-схемы алгоритмов и программ
(по ГОСТ 19.701—90)

языка UML, в частности, диаграмма деятельности (*activity diagram*), диаграмма состояний (*UML state machine diagram*) и некоторые диаграммы взаимодействия (*interaction diagrams*), например, диаграмма синхронизации (*timing diagram*).

§7. ВЫВОДЫ

1. Знания, выраженные с помощью любого письменного языка, можно разбить на две части:
 - процедурные знания;
 - декларативные знания.
2. Правильно записанные процедурные знания всегда можно превратить в алгоритм.
3. Учащиеся должны уметь:
 - писать, читать и понимать алгоритмы;
 - распознавать алгоритмы, которые встречаются в окружающей жизни, и записывать их.
4. Блок-схемы алгоритмов безнадежно устарели. Пользоваться ими недопустимо. Вместо них следует использовать дракон-схемы.

Часть II

АЛГОРИТМИЧЕСКИЙ ЯЗЫК ДРАКОН И УДОБНЫЕ ЧЕРТЕЖИ АЛГОРИТМОВ (ДРАКОН-СХЕМЫ)

ИКОНЫ И МАКРОИКОНЫ ЯЗЫКА ДРАКОН

§1. ПРЕИМУЩЕСТВА ДРАКОН-СХЕМ

Чем же отличаются дракон-схемы от блок-схем?

Хотя блок-схемы порою действительно улучшают понятность алгоритмов, 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 отличаются безупречной ясностью, прозрачностью, четкостью. Они обеспечивают быстроту и легкость понимания по принципу: «Посмотрел – и сразу понял!».

Благодаря использованию формальных и неформальных приемов дракон-схемы дают возможность изобразить любой, сколь угодно сложный алгоритм в наглядной и доходчивой форме.

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

И последнее. Блок-схемы не обеспечивают автоматическое преобразование алгоритма в машинный код. Дракон-схемы, напротив, отличаются математической строгостью. Они пригодны для автоматического получения кода и исполнения его на компьютере.

§2. ИКОНЫ

Чтобы графический язык стал гибким и выразительным, нужно тщательно спроектировать алфавит языка, сделать его стандартным и эргономичным.

Графоэлементы (графические буквы) языка ДРАКОН называются *иконами* (рис. 17). В языке ДРАКОН 26 икон.

Столь богатый алфавит обладает большой выразительной силой. Он позволяет изобразить алгоритмы любой сложности, включая параллельные процессы и процессы реального времени. И обеспечить максимальную наглядность и понятность полученной картинки.
























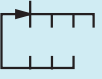




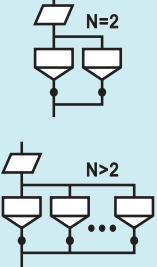






	Икона	Название иконы		Икона	Название иконы
И1		Заголовок	И14		Вывод
И2		Конец	И15		Ввод
И3		Действие	И16		Пауза
И4		Вопрос	И17		Период
И5		Выбор	И18		Пуск таймера
И6		Вариант	И19		Синхронизатор (по таймеру)
И7		Имя ветки	И20		Параллельный процесс
И8		Адрес	И21		Комментарий
И9		Вставка	И22		Правый комментарий
И10		Полка	И23		Левый комментарий
И11		Формальные параметры	И24		Петля цикла
И12		Начало цикла для	И25		Петля силуэта
И13		Конец цикла для	И26		Соединитель

Рис. 17. Иконы языка ДРАКОН

§3. МАКРОИКОНЫ

Подобно тому, как буквы объединяются в слова, иконы объединяются в составные иконы – *макроиконы* (рис. 18). Макроиконы – это составные алгоритмические структуры.

	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


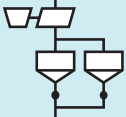

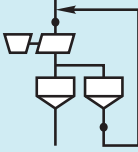







	Макроикона	Название макроиконы
10		Развилка по таймеру
11		Переключатель по таймеру
12		Обычный цикл по таймеру
13		Переключающий цикл по таймеру
14		Цикл ДЛЯ по таймеру
15		Цикл ЖДАТЬ по таймеру
16		Вставка по таймеру
17		Вывод по таймеру
18		Ввод по таймеру
19		Пуск таймера по таймеру
20		Параллельный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

Рис. 18. Макроиконы языка ДРАКОН

Соединяя иконы и макроиконы по определенным правилам, можно строить разнообразные алгоритмы. Примеры алгоритмов в изобилии представлены на страницах книги.

§4. ШАМПУР-БЛОК

Что такое
шампур-блок

- Это часть дракон-схемы, имеющая один вход сверху и один выход снизу
- Вход и выход шампур-блока всегда расположены на одной вертикали

Примерами шампур-блоков являются иконы И3, И5–И10, И12–И16, И18, И20, И21 (рис. 17) и макроиконы 2–20 (рис. 18).

§5. КАКИЕ ИКОНЫ И МАКРОИКОНЫ НАМ УЖЕ ЗНАКОМЫ?

Взглянем на рис. 17. Попытаемся найти иконы, которые нам уже встречались.

В главах 1 и 2 мы видели иконы И1 (заголовок), И2 (конец), И3 (действие), И4 (вопрос), И5 (выбор), И6 (вариант), И21 (комментарий).

Глядя на рис. 18, мы можем опознать две макроиконы: развилку и переключатель.

Остальные иконы и макроиконы мы пока не знаем. Они будут описаны в следующих главах.

§6. ВЫВОДЫ

1. Графический алфавит языка ДРАКОН содержит 26 икон.
2. Иконы объединяются в составные иконы – макроиконы.
3. В языке ДРАКОН имеется 20 макроикон.
4. При построении дракон-схем используются иконы и макроиконы.

АЛГОРИТМИЧЕСКАЯ СТРУКТУРА «СИЛУЭТ»

§1. ОСНОВНАЯ СТРУКТУРА ЯЗЫКА ДРАКОН

Конструкция «силуэт» – чрезвычайно мощный инструмент. Она обеспечивает большие удобства для пользователя. Данная конструкция не имеет аналогов в других алгоритмических языках.

Основным элементом силуэта является «ветка». С нее-то мы и начнем.

§2. 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. 19 и 20. Легко видеть, что рис. 19 не позволяет ответить на вопрос судьи. Зато рис. 20, наоборот, содержит нужную информацию. Более того, алгоритм на рис. 20 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*.

Разберем еще один пример. На рис. 21 представлен алгоритм «Поход за грибами», содержащий довольно большую последовательность действий. Иной раз такая последовательность может оказаться чрезмерно длинной и утомительной для чтения. Можно ли облегчить восприятие и анализ подобных «длиннющих» алгоритмов? Можно ли сделать «договязый» алгоритм обозримым и удобным для быстрого понимания?

Да, можно. Чтобы облегчить работу читателя и сделать алгоритм эргономичным, надо заблаговременно разрезать «длинную кишку» и разбить ее на смысловые части. Сделать это нетрудно. «Поход за грибами» (рис. 21) – это алгоритмический рассказ, в котором можно выделить три крупных куса, три самостоятельных темы:

ПРИМИТИВ

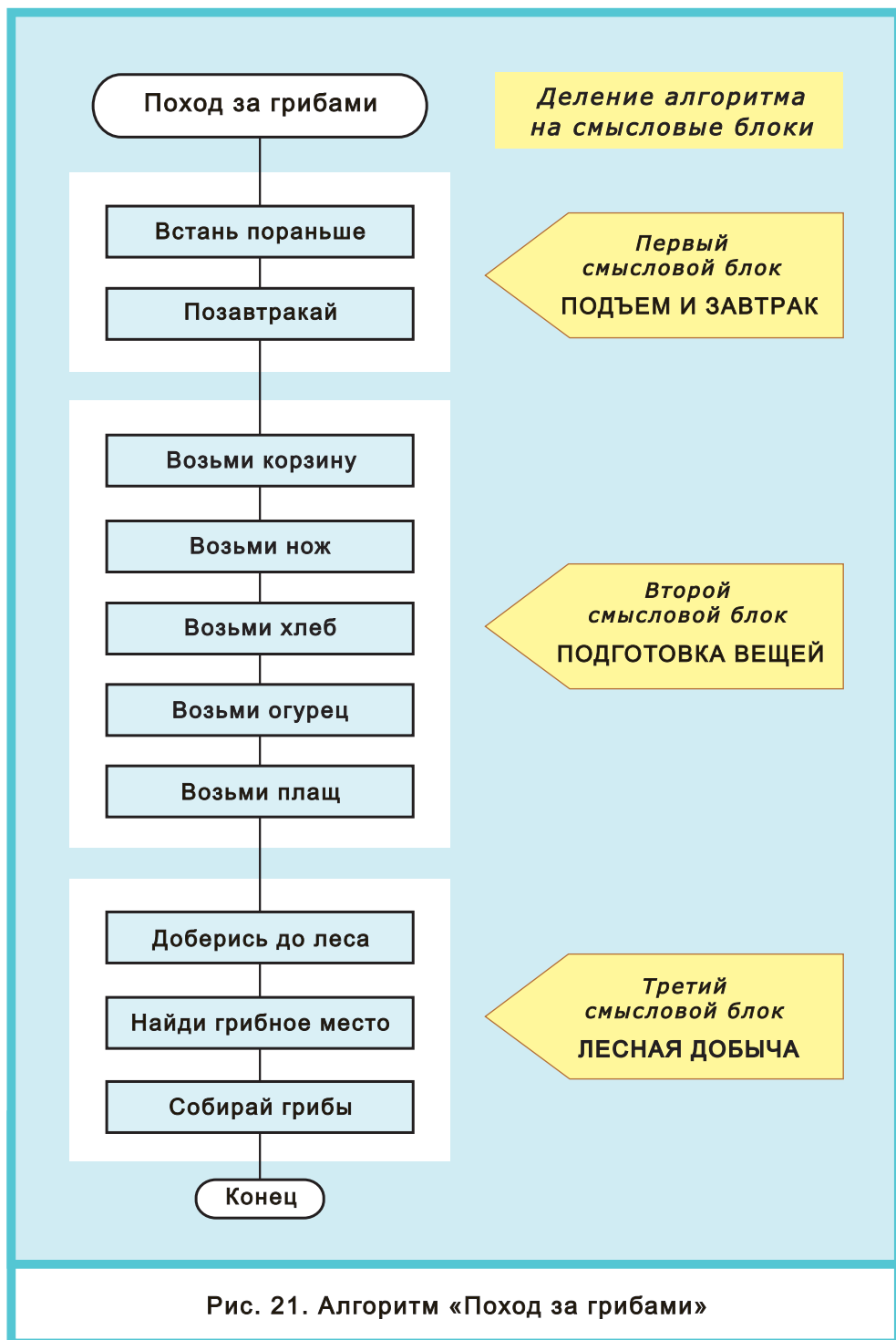


Рис. 19. Эта схема называется «примитив». В примитиве ветки отсутствуют

СИЛУЭТ





Рис. 20. Эта схема называется «силуэт». Силуэт делится на смысловые части (ветки), которые помогают понять структуру алгоритма



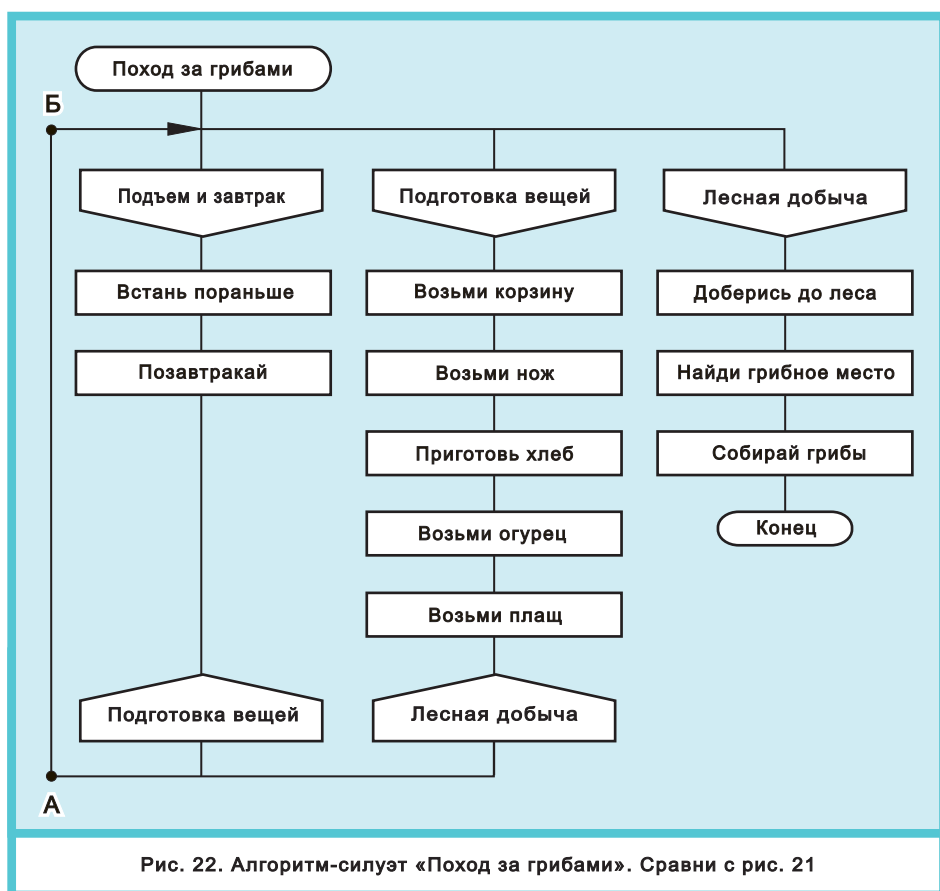
- Подъем и завтрак.
- Подготовка вещей.
- Лесная добыча.

Каждую тему можно нарисовать в виде ветки. Результат изображен на рис. 22. Названия новых икон приведены 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. 22?

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



Более сложный пример показан на рис. 25. Здесь мы видим четыре ветки:

- Подготовка к ловле.
- Ожидание клева.
- Рыбацкая работа.
- Обратная дорога.

Зачем нужна ветка? Чтобы помочь алгоритмисту разбить алгоритм на смысловые части. И, что очень важно, дать частям удобные названия, которые пишут в иконе «имя ветки».

Разделение проблемы на N смысловых частей реализуется путем разбиения алгоритма на N веток.

Ветка – составной оператор языка ДРАКОН, который не имеет аналогов в известных языках.

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)

Икона «имя ветки» очень похожа на икону «вариант». С непривычки их можно перепутать. Чтобы избежать досадных ошибок, сравните иконы на рис. 23 и 24.

Чем отличается икона «имя ветки» от иконы «вариант»?

Икона «имя ветки»

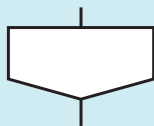


Рис. 23. У иконы «имя ветки» одна горизонтальная линия

Икона «вариант»

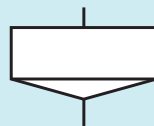


Рис. 24. У иконы «вариант» две горизонтальные линии

§3. ВХОД И ВЫХОДЫ ВЕТКИ

Ветка имеет один вход. И один или несколько выходов. Входом служит икона «имя ветки», содержащая название ветки.

Визуальный оператор «имя ветки» не выполняет никаких действий. Это всего лишь метка, объявляющая название смысловой части алгоритма.

Исполнение дракон-схемы всегда начинается с крайней левой ветки (рис. 20, 22, 25).

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».

§4. ЧТО ТАКОЕ ШАПКА?

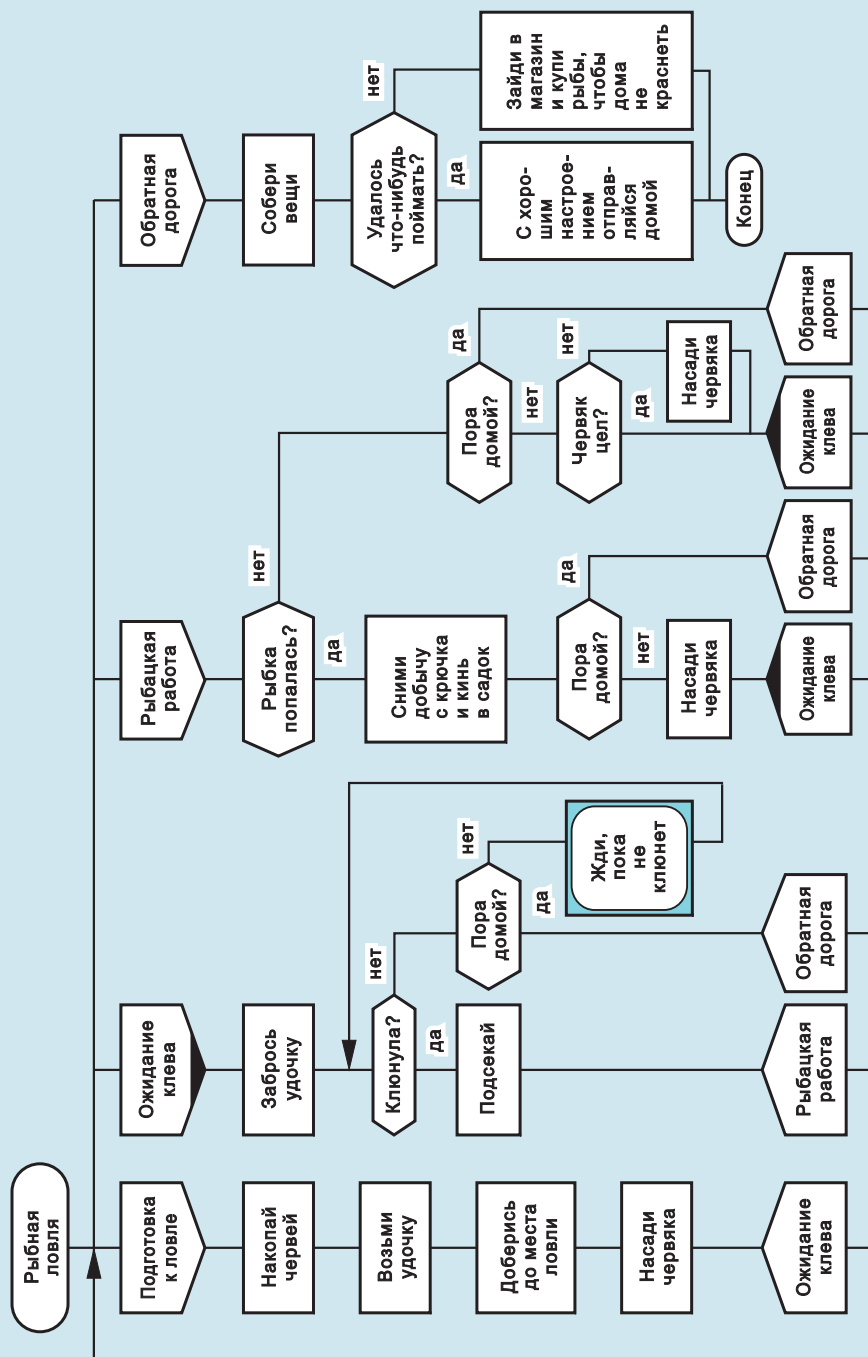
Многие алгоритмы чрезвычайно сложны. Чтобы разобраться в тонкостях такого алгоритма, нужно прилагать большие усилия и тратить много времени.

Подобную практику следует признать порочной. Алгоритмы можно и нужно сделать легкими для восприятия (эргономичными). Для этого надо давать читателю маленькие, но умные подсказки, проглотив которые, он мог бы легко сориентироваться в задаче и быстро понять материал. Одной из таких подсказок служит *шапка*. Название объясняется тем, что шапка находится сверху, «на голове» у алгоритма.

Что такое
шапка

Это верхняя часть дракон-схемы (рис. 25, 25а), которая включает заголовок алгоритма и комплект икон «имя ветки»

Назначение шапки – помочь читателю мгновенно (за несколько секунд или минут) сориентироваться в задаче и, расчленив ее на части, увидеть структуру алгоритма (рис. 25а). Причем увидеть не в фигуральном смысле слова, не с помощью воображения, не духовным оком, а своими двумя глазами – на бумаге или экране.





§5. ТРИ «ЦАРСКИХ» ВОПРОСА

Сталкиваясь с новой незнакомой задачей, 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 подсказку – ответ на все царские вопросы.

Вот ответы для рис. 22.

- Как называется задача? (*Читаем заголовок алгоритма*).
Поход за грибами.
- 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. Подготовка вещей.
 3. Лесная добыча.

§6. ТРЕХЭТАПНЫЙ МЕТОД ИЗУЧЕНИЯ АЛГОРИТМА

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

Благодаря этому шапка моментально привлекает к себе внимание читателя без всяких усилий с его стороны. Поэтому человеку не прихо-

дится рыскать глазами по темным закоулкам алгоритма, пытаясь выудить нужную информацию.

Таким образом, ДРАКОН предоставляет читателю эффективный трехэтапный метод познания незнакомой или забытой проблемы.

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.

§7. КАК РАБОТАЕТ СИЛУЭТ?

Правило. Выполнить алгоритм – значит пройти путь от начала до конца алгоритма.

Применим это правило к рис. 22. Будем считать, что существует воображаемый бегунок, который при работе алгоритма пробегает алгоритмическую дорожку от иконы «заголовок» до иконы «конец».

Как работает алгоритм-силуэт? Выехав из иконы «заголовок», бегунок мчится вниз по крайней левой ветке. Он движется через станции (рис. 22):

- Подъем и завтрак.
- Встань пораньше.
- Позавтракай.
- Подготовка 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 говорит: следующая станция называется «Подготовка вещей».

Из рис. 22 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. 22). Потом движется вверх к точке *Б*. Затем едет направо по стрелке и въезжает в верхнюю икону «Подготовка вещей».

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

§8. В ЧЕМ СЕКРЕТ ИКОНЫ «АДРЕС»?

Сейчас мы поступим, как чеховский злоумышленник – будем разбирать рельсы. Имеются в виду линии, окаймляющие алгоритм на рис. 22. Сотрем

линию *АБ*. Уберем также горизонтальные линии (шины), проходящие через точки *А* и *Б*. Результат представлен на рис. 26.

Как будет работать силуэт 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 так. Ведь в иконе «адрес» записан адрес следующей станции («Подготовка 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).

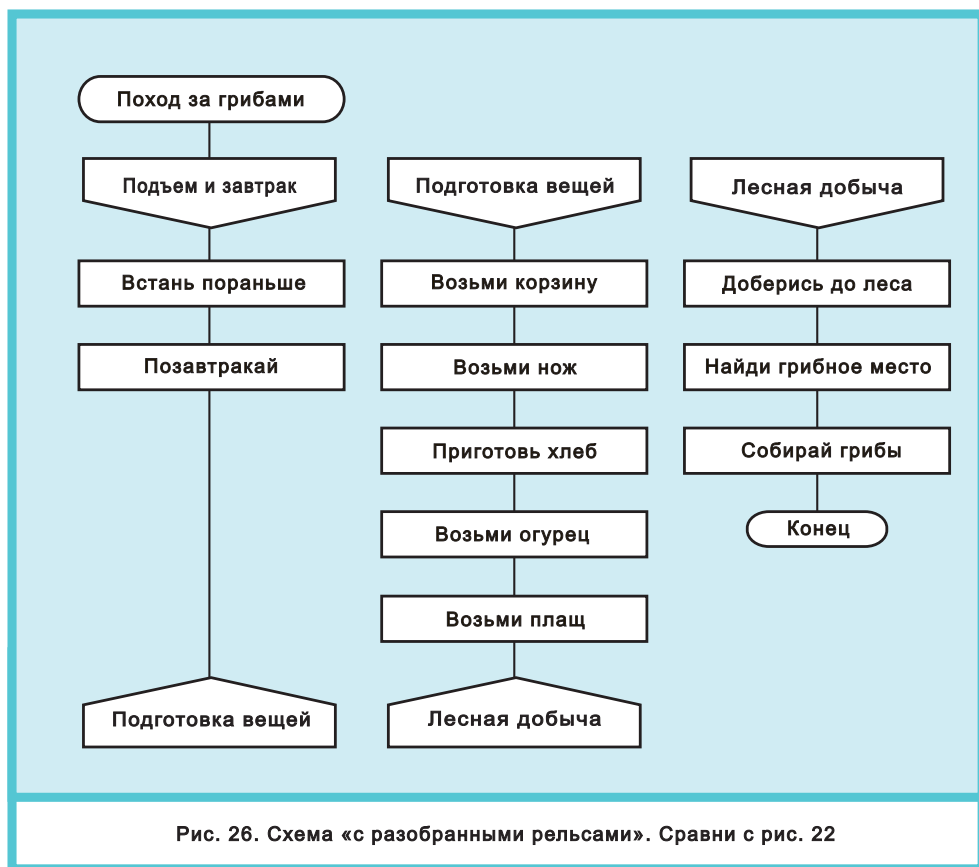


Рис. 26. Схема «с разобранными рельсами». Сравни с рис. 22

Что такое
икона «Адрес»



Это команда, передающая
управление в начало ветки А

Переходя от рис. 22 к рис. 26, мы для упрощения стёрли несколько линий. Теперь мы убедились, что для работы алгоритма они совершенно не нужны. 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 сбивает с толку читателя (рис. 26).

§9. 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. Очень важно правильно расположить ветки в пространстве. Но как это сделать?

Давайте мысленно перетасуем ветки на как колоду карт (рис. 26). И расположим их на чертеже в произвольном порядке. Легко сообразить, что подобное «перепутывание» веток никак не отразится на работе алгоритма. Ведь 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 тонкость. Перестановка веток не отражается на правильности алгоритма. Однако алгоритм должен быть не только правильным, но и понятным, эргономичным. Хаотичное расположение веток затрудняет понимание. А это 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 правило: «Чем правее, тем позже». Оно означает: чем правее находится ветка, тем позже она работает.

Отсюда вытекает эргономическая

Рекомендация. Чтобы схема была наглядной и удобной для восприятия, ветки должны быть упорядочены слева направо согласно принципу: «Более правая ветка работает позже, чем любая ветка, расположенная левее нее».

Алгоритм, нарисованный согласно правилу «чем правее – тем позже», считается хорошим, эргономичным (рис. 22). Схемы, где это правило нарушается, объявляются плохими. Их использование запрещено.

В разрешенных (эргономичных) алгоритмах имеет место следующий порядок работы (рис. 22, 25):

- 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. 25 веточный цикл помечен черными треугольниками.

§10. ТРИ ТИПА ВЕТОК

Различают три типа веток: одноадресная, многоадресная и безадресная.

Одноадресная ветка содержит только одну икону адрес. Например, на рис. 22 имеются две одноадресные ветки.

Многоадресная ветка содержит две или более икон адрес.

Рассмотрим рис. 25. Первая ветка одноадресная. Вторая ветка имеет два адреса:

- Рыбацкая работа.
- Обратная дорога.

Третья ветка содержит четыре адреса (четыре иконы адрес):

- Ожидание клева.
- Обратная дорога.
- Ожидание клева.
- Обратная дорога.

Таким образом, силуэт «Рыбная ловля» имеет две многоадресных ветки: двухадресную и четырехадресную.

Рассмотрим еще один случай. *Безадресная ветка* – ветка, содержащая икону конец и не содержащая икону адрес. Последняя ветка силуэта почти всегда безадресная (рис. 20, 22, 25). Исключением являются бесконечные алгоритмы, в которых отсутствует икона конец (рис. 140, 148).

§11. ПЕРЕСЕЧЕНИЯ ЛИНИЙ ЗАПРЕЩЕНЫ

Традиционные блок-схемы чаще всего имеют неформальную и неудобную для человека форму, похожую на спагетти. Изображенные на них

хитросплетения блоков, соединенные хаосом линий, больше напоминают «кучу мусора», нежели регулярную структуру.

Язык ДРАКОН выгодно отличается тем, что его графический узор имеет строгое математическое и эргономическое обоснование. Он 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 пересечений:

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

Как известно, соединитель используется для обрыва линии и продолжения ее в другом месте для избежания излишних пересечений.

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

§12. ВНЕШНИЕ СОЕДИНИТЕЛИ

Следует различать внутренние и внешние соединители. Первые были описаны в §11.

Внешние
соединители

Это иконы, обозначающие переход
горизонтальных шин силуэта
с листа на лист

Зачем нужен такой переход? Чем сложнее алгоритм, тем больше веток в силуэте. Увеличение числа веток приводит к выходу за пределы бумажного листа.

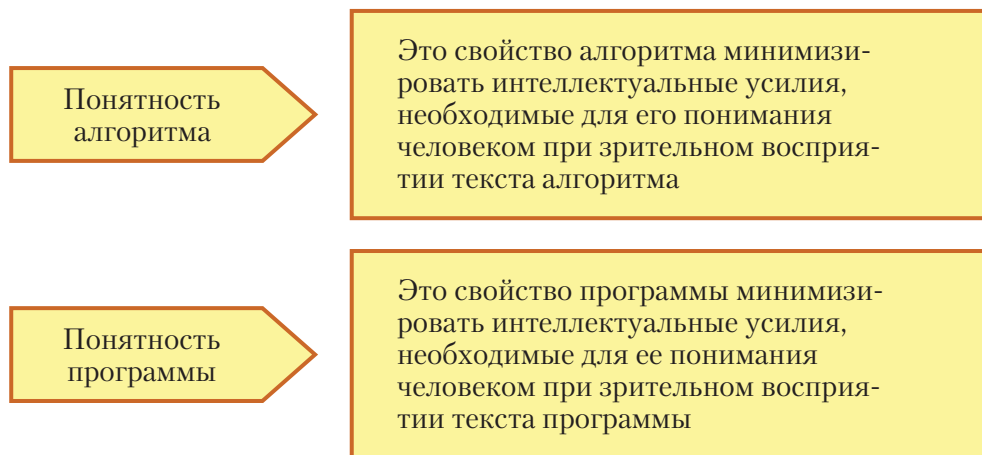
Каждый силуэт имеет две горизонтальные шины – верхнюю и нижнюю (см. рис. 22 и 25). Если алгоритм размещается на двух, трех или четырёх листах большого формата (например, А1), шины вытягиваются по горизонтали. И переходят с листа на лист. В местах перехода устанавливаются внешние соединители.

Такие соединители подробно описаны в главе 30.

§13. ЧТО ТАКОЕ ПОНЯТНОСТЬ АЛГОРИТМА?

Многие алгоритмисты жалуются, что свой собственный алгоритм они с трудом понимают через полгода, а то и через месяц. А если речь идет о чужом алгоритме? Тогда становится совсем тяжело. Нередко бывает легче написать свой алгоритм, нежели разобраться в том, что делает чужой.

Поэтому среди требований, предъявляемых к современным алгоритмическим языкам, на первое место все чаще выходит *понятность алгоритмов*.



§14. ЭРГОНОМИЧНЫЙ АЛГОРИТМ

Эргономичный алгоритм – алгоритм, удовлетворяющий критерию сверхвысокого понимания. 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 проверке за столом, тем больше вероятность, что вновь созданный алгоритм окажется правильным, безошибочным, надежным. Кроме того, эргономичные алгоритмы удобны для изучения, их проще объяснить другому человеку.

Все в алгоритме понятно и ясно,
Если он сделан эргономично.
Эргономично – это прекрасно!
Эргономично – значит отлично!

§15. ВЫВОДЫ

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

1. В иконе «заголовок» запрещается писать слово «начало». Вместо этого следует указать понятное и точное название алгоритма.
2. Разбейте сложный алгоритм на части, каждую часть изобразите в виде ветки. Дайте частям доходчивые и четкие названия и запишите их в иконах «имя ветки».
3. Вход в ветку возможен только через ее начало.
4. В иконе «адрес» разрешается писать имя одной из веток. Другие надписи запрещены.
5. Ветки следует располагать в пространстве согласно правилу: чем правее, тем позже. Наличие веточного цикла модифицирует это правило.
6. В иконе «конец» следует писать слово «Конец».
7. Соединительные линии могут идти либо горизонтально, либо вертикально. Наклонные линии не допускаются.
8. Пересечения линий запрещены.
9. Обрывы линий запрещены.
10. Внутренние соединители запрещены.

АЛГОРИТМИЧЕСКАЯ СТРУКТУРА «ПРИМИТИВ»

§1. ЧТО ТАКОЕ ПРИМИТИВ?

В языке ДРАКОН есть два логически законченных алгоритмических чертежа.

- силуэт (о котором шла речь выше);
- примитив.

Примитив

Это составная алгоритмическая структура, у которой иконы «заголовок» и «конец» лежат на одной вертикали

Примеры примитивов показаны на рис. 1, 3, 4–8, 10–12, 15, 19, 21.

§2. ЧТО ТАКОЕ ШАМПУР?

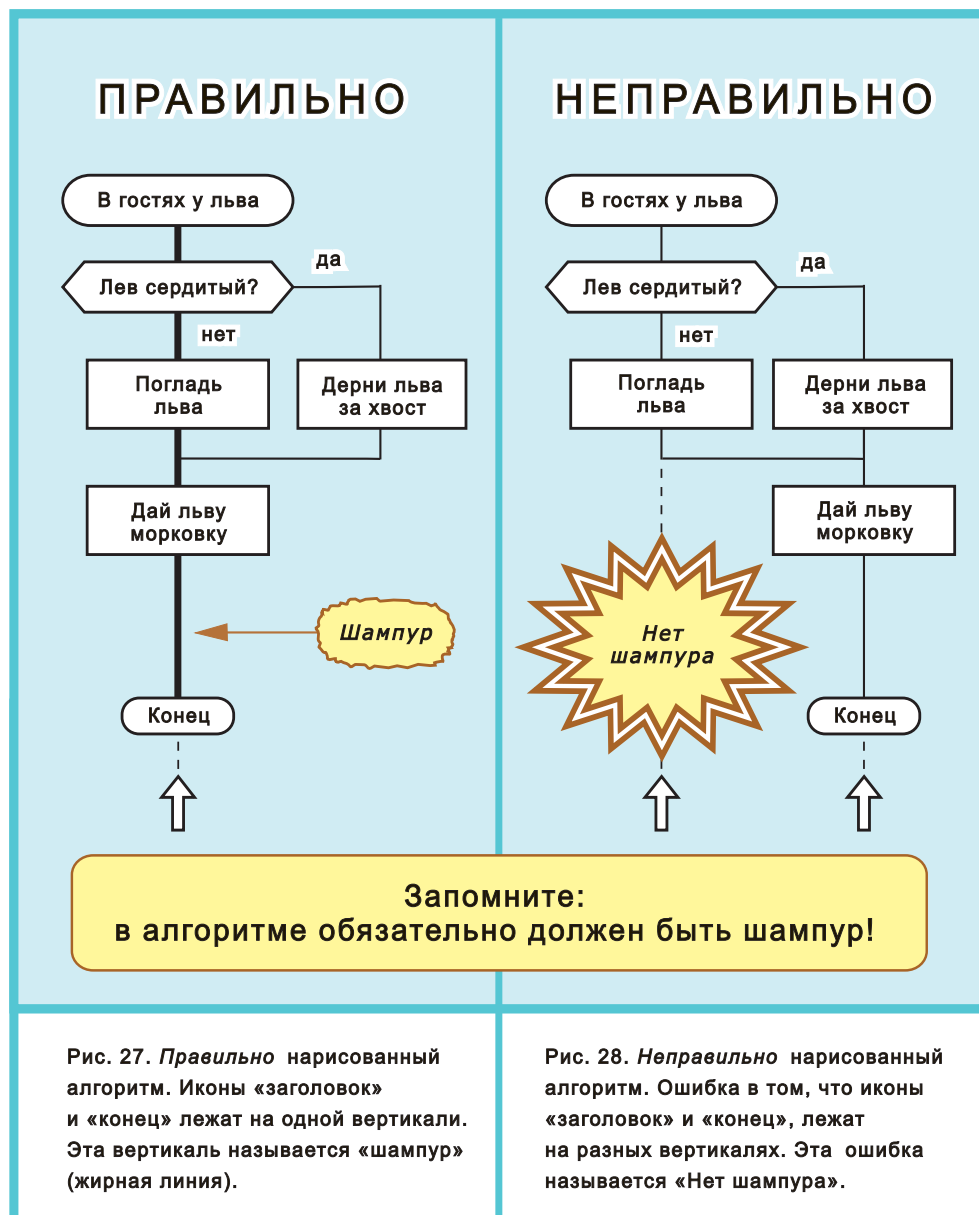
Шампур

- Это вертикальная линия, соединяющая иконы «заголовок» и «конец»
- Между ними на этой же линии помещается одна или несколько других икон

Образно говоря, шампур пронизывает иконы примитива (возможно, не все) подобно тому, как настоящий шампур пронизывает кусочки шашлыка (рис. 27).

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

Если это правило выполняется, дракон-схема становится упорядоченной, эргономичной, легкой для чтения (рис. 27). И наоборот, нарушение данного правила делает схему корявой, изломанной, неудобной для глаза (рис. 28).



§3. ЧТО ТАКОЕ ГЛАВНЫЙ МАРШРУТ?

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

На рис. 1 показан неразветвленный алгоритм. В нем всего один маршрут. В разветвленном алгоритме на рис. 4 таких маршрутов четыре. Если тропинок несколько, среди них можно выделить главный и побочные маршруты.

Главный маршрут
алгоритма

Это путь от начала до конца
алгоритма, который ведет к
наибольшему успеху

Например, на рис. 4 главный маршрут проходит по левой вертикали. Этот маршрут позволяет сыну выполнить поручение матери наилучшим образом и в полном объеме – купить один батон белого и один черного.

Остальные три маршрута на рис. 4 ведут либо к частичному успеху (удалось купить только белый хлеб или только черный), либо к полной неудаче (никакого хлеба нет).

§4. ПРАВИЛО ГЛАВНОГО МАРШРУТА

Рассмотрим задачу. В запутанном лабиринте, соединяющем начало и конец сложного алгоритма, нужно выделить один-единственный маршрут – царскую дорогу, «путеводную нить». С ней можно зрительно сравнивать все прочие маршруты, чтобы легко сориентироваться в задаче и не заблудиться в путанице развилок.

Путеводная нить должна быть визуалью легко различима. 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. 30), 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. 29).

ПРАВИЛЬНО

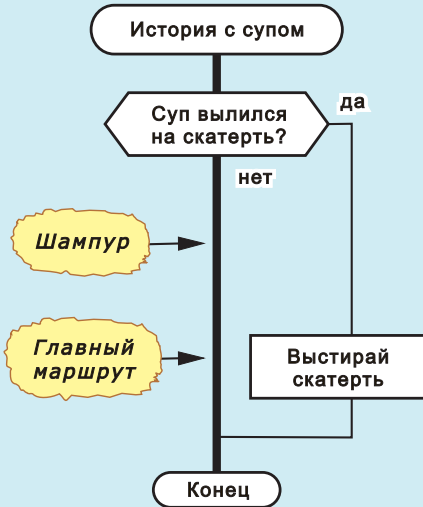


Рис. 29. Хорошая (эргономичная) схема. Главный маршрут идет по шампуру

НЕПРАВИЛЬНО



Рис. 30. Плохая схема. Нарушено правило «Главный маршрут должен идти по шампуру»

§5. ПОБОЧНЫЕ МАРШРУТЫ НЕЛЬЗЯ РИСОВАТЬ КАК ПОПАЛО

Что такое
побочный маршрут

Это 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, Господи! 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. 31 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.

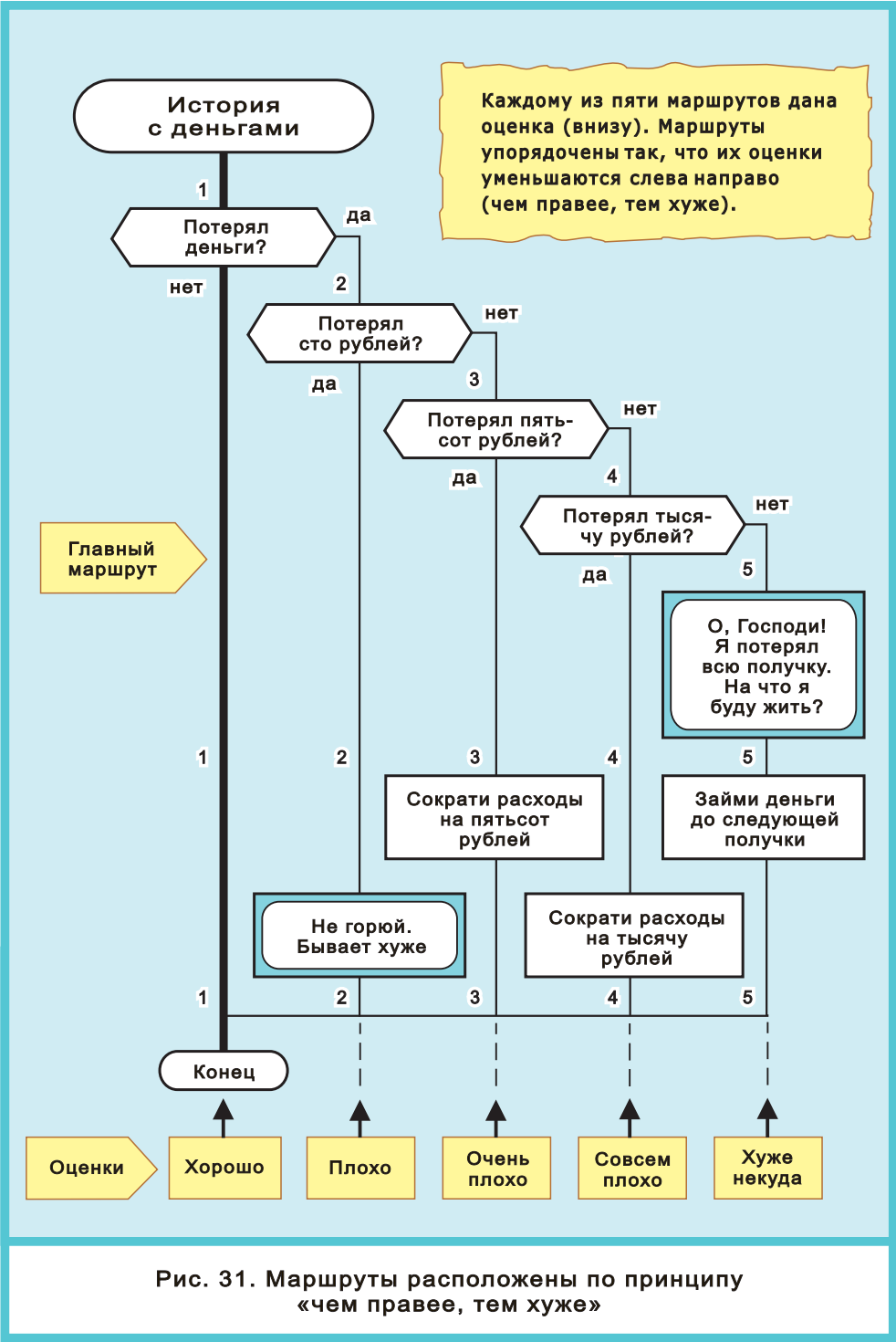
Язык ДРАКОН был разработан, в частности, потому, что традиционные блок-схемы алгоритмов, с эргономической точки зрения, не выдерживают критики. Они напоминают непроходимые джунгли, в которых легко запутаться и почти ничего нельзя понять.

ДРАКОН выгодно отличается тем, что его графический узор подчиняется жестким и тщательно продуманным правилам, которые дисциплинируют мышление, облегчают умственный труд.

Чтобы убедиться в этом, взглянем еще разок на рис. 31 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 такую схему, человек не станет плутать в потемках, так как при ее составлении использовано мудрое

Правило. Все хорошее – слева, все плохое – справа.

Про такой алгоритм можно сказать: «Все ясно, как на ладони!».



§6. ЧТО ДЕЛАТЬ, ЕСЛИ ПРИНЦИП «ЧЕМ ПРАВЕЕ, ТЕМ ХУЖЕ» НЕ РАБОТАЕТ?

Из физики известно: если начальная скорость ракеты меньше 7,8 километров в секунду, она непременно упадет на Землю. Если же разогнать ее чуть сильнее, но не больше 11,2 километров в секунду, она станет спутником Земли. При дальнейшем увеличении скорости ракета станет спутником Солнца.

А если скорость превысит 16,4 километра в секунду, ракета навсегда простится с Солнечной системой и умчится в безбрежные просторы космоса.

Алгоритм этой задачи показан на рис. 32.

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 на рис. 32 оно не имеет смысла. Поэтому 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.

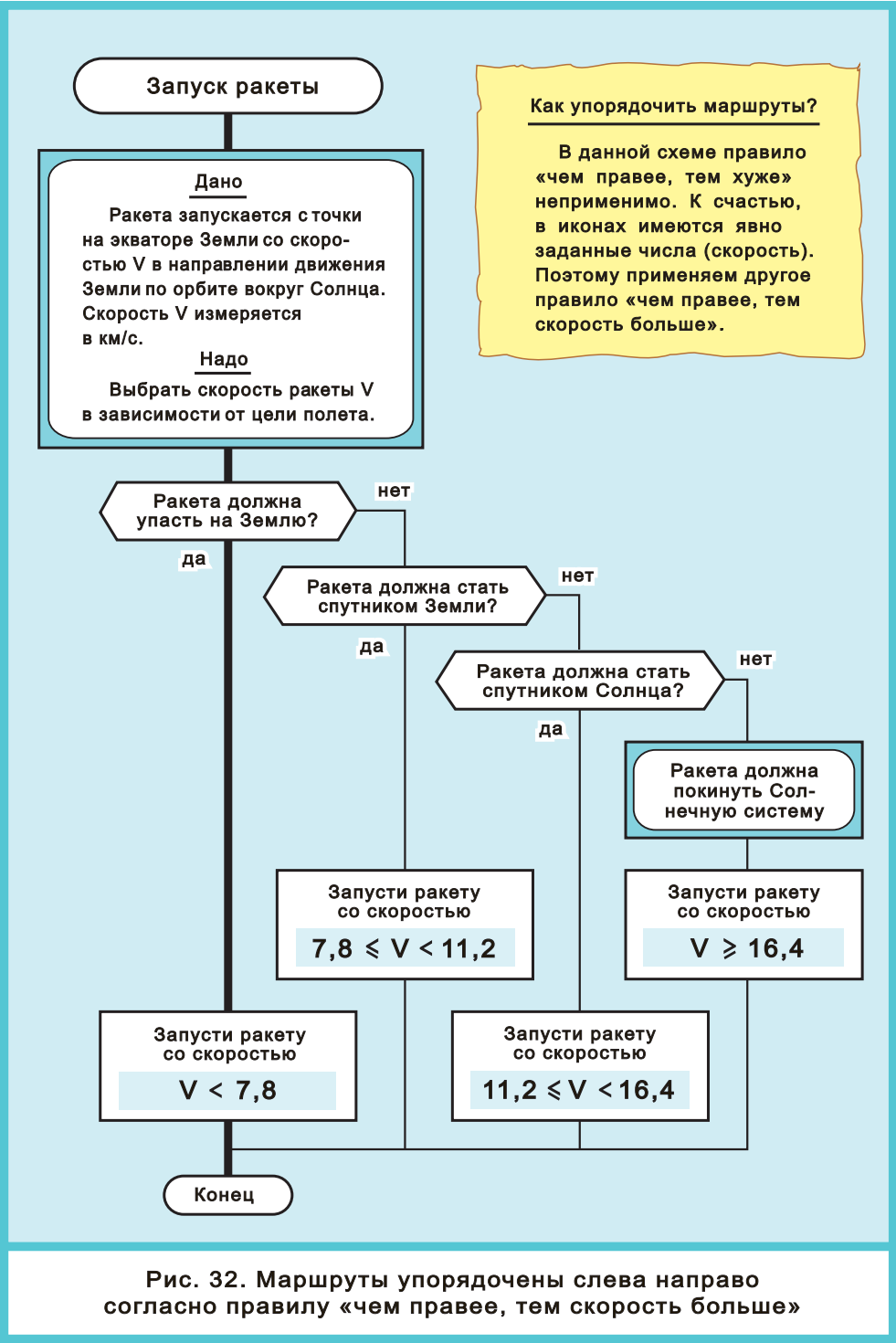
§7. КАК РИСОВАТЬ ПОБОЧНЫЕ МАРШРУТЫ?

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. 33). Но у всех правил есть общая черта. В схеме должен быть не хаос, а порядок. Здесь действует

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



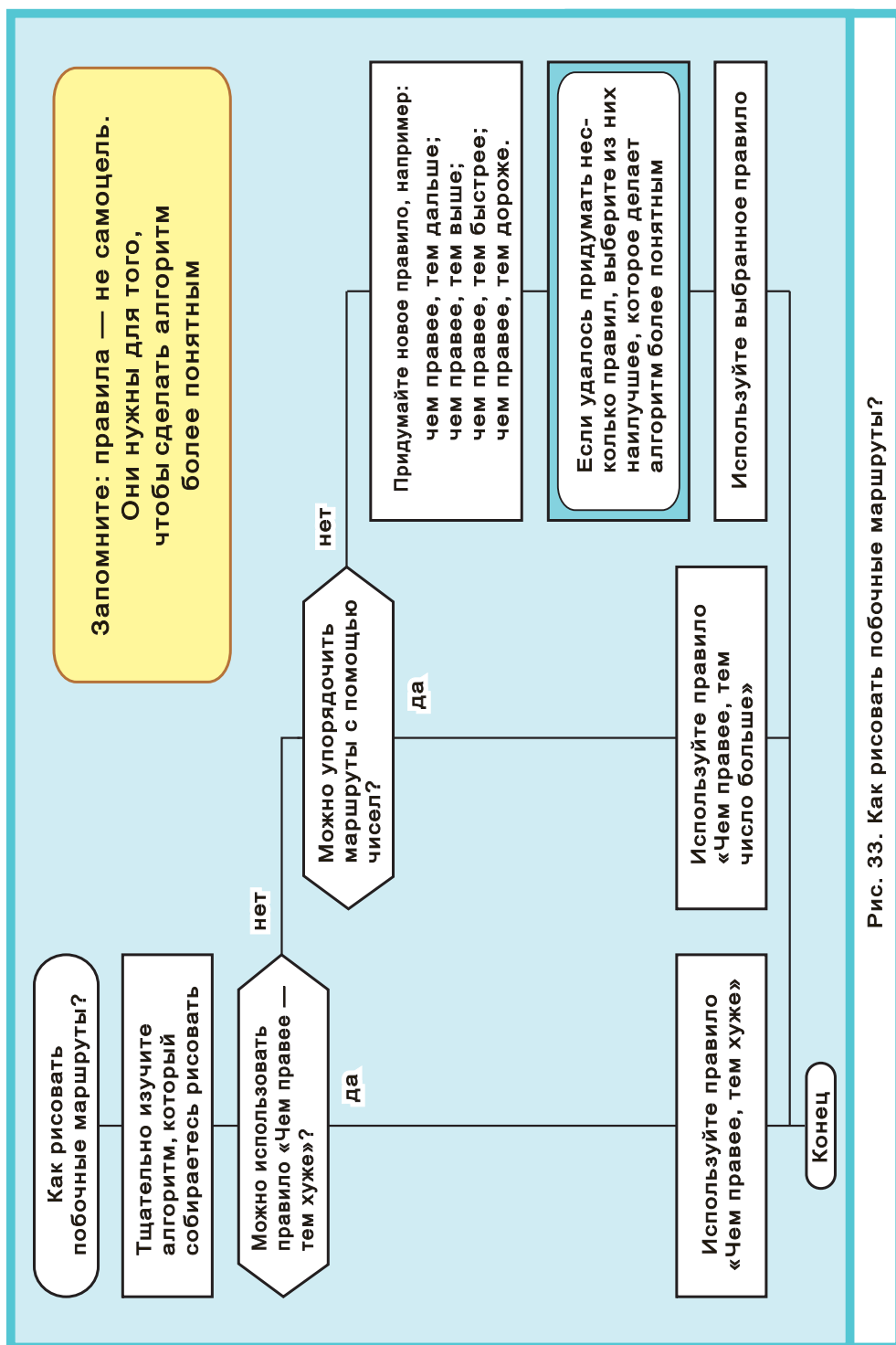
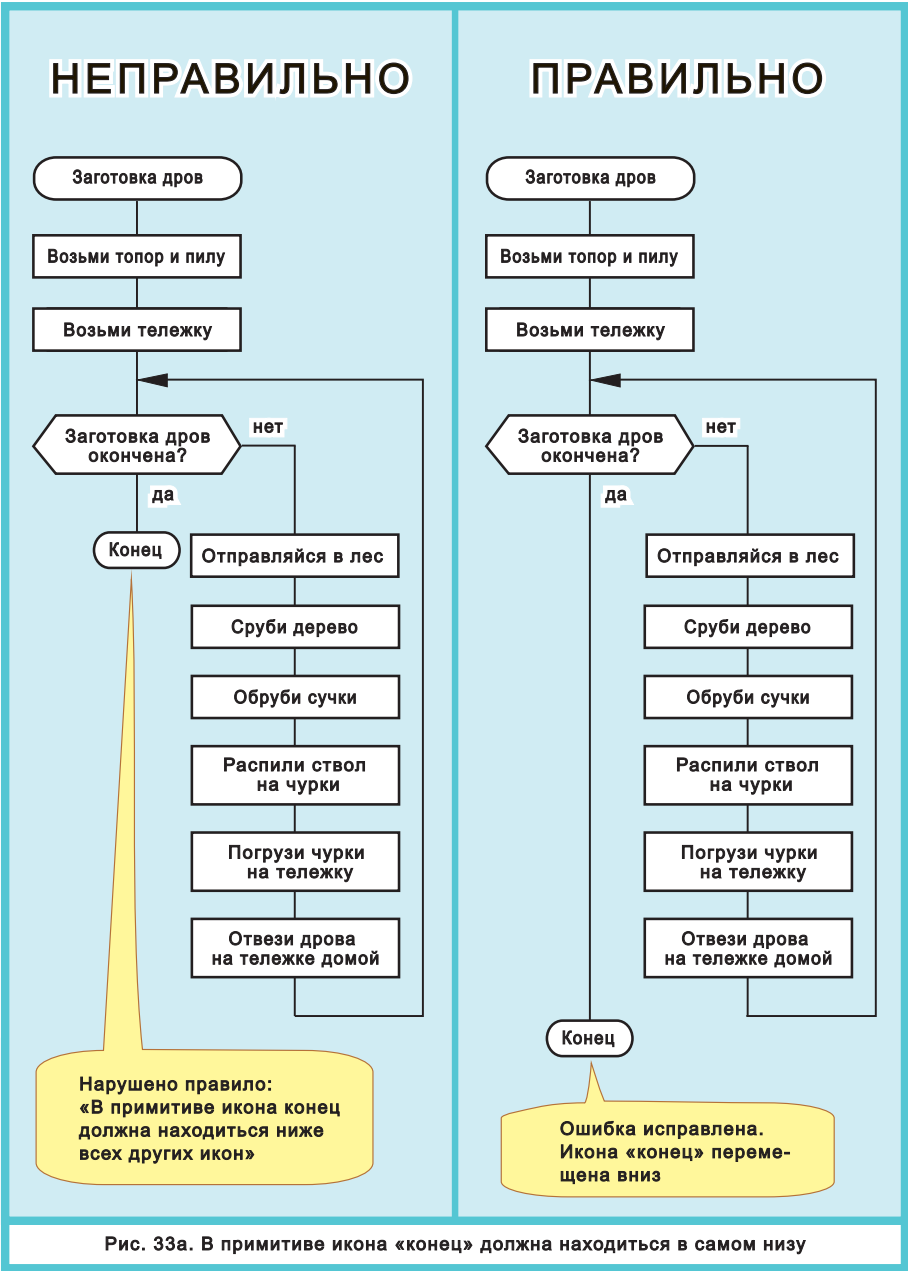


Рис. 33. Как рисовать побочные маршруты?

§8. ГДЕ ДОЛЖНА НАХОДИТЬСЯ ИКОНА «КОНЕЦ»?

В примитиве икона конец должна размещаться ниже всех других икон (рис. 33а).



§9. ВЫВОДЫ

1. В языке ДРАКОН используются две основные алгоритмические структуры:
 - силуэт;
 - примитив.
2. Следует различать:
 - главный маршрут;
 - побочные маршруты.
3. Главный маршрут идет по шампуру.
4. Побочные маршруты находятся справа от главного. Они располагаются слева направо по принципу: «чем правее, тем хуже».
5. Если указанный принцип нельзя применить, побочные маршруты должны подчиняться правилу: «смещение вправо от главного маршрута должно быть не произвольным, а упорядоченным».

СРАВНИМ СИЛУЭТ И ПРИМИТИВ

§1. ЧТО ЛУЧШЕ: СИЛУЭТ ИЛИ ПРИМИТИВ?

Напомним определения понятий, о которых читатель уже мог догадаться, глядя на рис. 19 и 20.

Силуэт – дракон-схема, разделенная на ветки.

Примитив – дракон-схема, не имеющая веток.

Сравним между собой силуэт и примитив. Какая схема лучше, легче для понимания?

Чтобы уяснить суть дела, возьмем один и тот же алгоритм «Поездка на автобусе». Изобразим его двумя способами: в виде силуэта (рис. 34) и в виде примитива (рис. 35).

Легко сообразить, что силуэт обладает серьезными преимуществами. В самом деле, примитив на рис. 35 не позволяет увидеть деление задачи на смысловые части. Иное дело рис. 34. 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. 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 десятка икон. В остальных случаях выгоднее использовать силуэт.

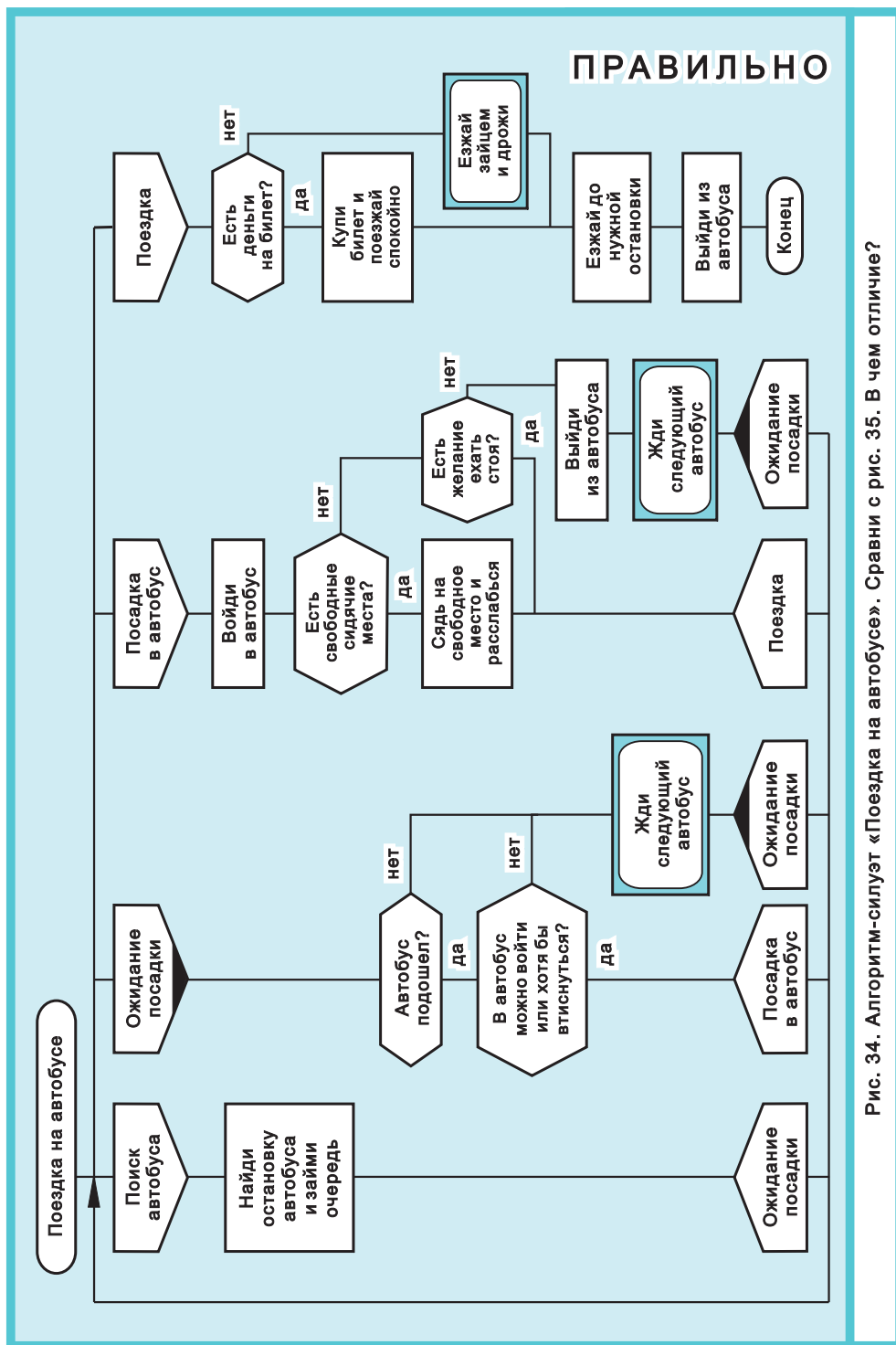
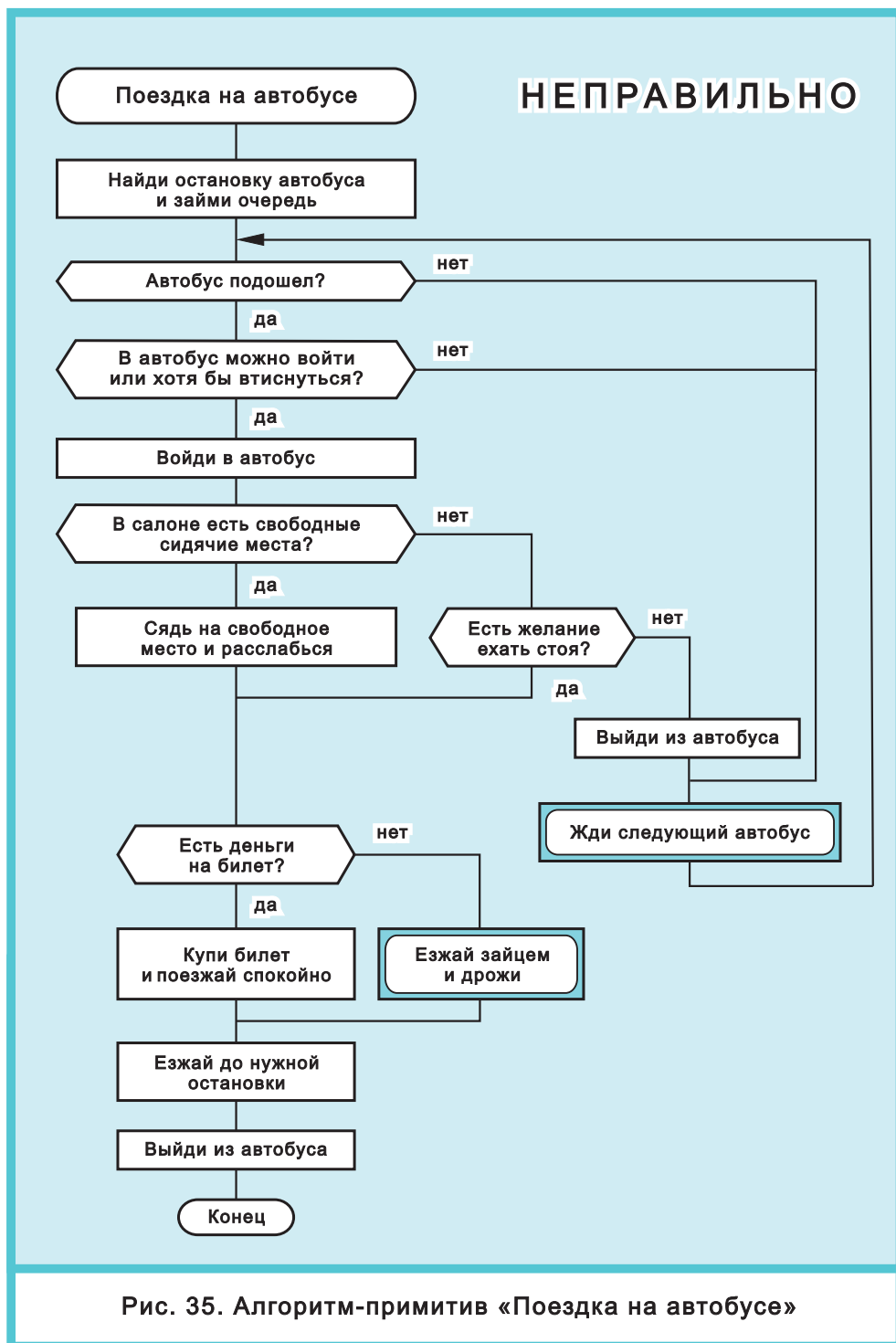


Рис. 34. Алгоритм-силуэт «Поездка на автобусе». Сравни с рис. 35. В чем отличие?



Нарушение этого правила обычно чревато неприятностями, ибо мешает читателю выявить сущность решаемой проблемы.

Например, алгоритм на рис. 35 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. 35 не позволяет читателю мгновенно (за несколько секунд) распознать зрительно-смысловую структуру алгоритма.

В самом деле, из скольких частей состоит решаемая задача? Глядя на рис. 35, ответить на этот вопрос довольно трудно. А быстро ответить – невозможно.

Положение в корне меняется, когда мы смотрим на рис. 34, 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. 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 алгоритма (ветки) четко выделены, образуя легко узнаваемый, стабильный, предсказуемый и целостный зрительный образ.

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

Окончательные рекомендации изложены в главе 30.

§2. ГЛАВНЫЙ МАРШРУТ СИЛУЭТА

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

Что такое
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. 34 четыре ветки. Следовательно, этот силуэт имеет четыре шампура.

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 действует

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

Например, на рис. 34 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. В зависимости от ответа (да или нет) может быть либо частичный успех (вы будете ехать, но не сидя, а стоя), либо неудача, поскольку вы выходите из автобуса несолоно хлебавши.

Главный маршрут
силуэта

Это последовательное соединение
главных маршрутов поочередно
работающих веток (веточные циклы
для простоты не рассматриваются)

Таким образом, ДРАКОН позволяет читателю быстро 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.

§3. ШАМПУР ВЕТКИ

Мы уже знаем, что каждая ветка имеет шампур. Рассмотрим тему подробнее.

Чтобы ветка была красивой (эргономичной), следует применить

***Правило.** Левый маршрут каждой ветки должен идти по прямой вертикальной линии – шампуру ветки.*

Например, шампур первой ветки на рис. 25 найти очень легко. Он идет через иконы:

- Подготовка к ловле.
- Накопай червей.
- Возьми удочку.

- Доберись до места ловли.
- Насади червяка.
- Ожидание клева.

Найти шампур второй ветки несколько труднее, так как ветка разветвленная – в ней три маршрута. Надо отыскать самый левый из них и убедиться, что он идет по вертикали. На рис. 25 видно, что шампур второй ветки бежит через иконы:

- Ожидание клева.
- Забрось удочку.
- Клюнула? Да.
- Подсекай.
- Рыбацкая работа.

Запомните

- В примитиве только один шампур. А в силуэте их несколько.
- Каждая ветка имеет свой шампур.
- В силуэте число шампуров равно числу веток (см. примеры на рис. 22, 25, 34).

§4. В КАКОМ ПОРЯДКЕ РАСПОЛАГАЮТСЯ ИКОНЫ «АДРЕС» В ДВУХАДРЕСНОЙ ВЕТКЕ?

Рассмотрим двухадресную ветку, то есть ветку, имеющую две иконы адрес. Какую из них следует рисовать слева, а какую – справа?

На этот счет (при отсутствии веточного цикла) имеется рекомендация:

- Слева (на шампуре ветки) рисуйте икону адрес, которая указывает на ветку, которая должна выполняться **РАНЬШЕ**.
- Справа рисуйте икону адрес, указывающую на ветку, которая будет выполняться **ПОЗЖЕ**.

Можно сказать короче. «Порядок расположения икон адрес должен соответствовать порядку выполнения веток».

§5. ПОРЯДОК ИКОН «АДРЕС» И ПОРЯДОК ВЕТОК ПРИ НАЛИЧИИ ВЕТОЧНОГО ЦИКЛА

Дракон-схема должна быть упорядоченной. В ней должен господствовать порядок. Порядок хорош тем, что устраняет путаницу. И создает комфортные условия для умственной работы.

Обратимся к рис. 35а. Ветка «Наклейка кафеля» трехадресная. В ней три иконы адрес:

- наклейка кафеля;
- доставка плиток;
- уборка.

Обратите внимание на расположение этих икон по горизонтали. Примечательно, что оно полностью совпадает с расположением трех веток (2-й, 3-й и 4-й).

Желательно, чтобы порядок икон адрес совпадал с расположением веток. К сожалению, это не всегда возможно.

Тем не менее, мы не должны упускать даже малейшую возможность навести порядок. Эту мысль подытоживает

***Рекомендация.** Желательно, чтобы порядок расположения икон адрес в многоадресной ветке совпадал с расположением веток, на которые ссылаются эти адреса.*

Вот пример. Предположим, что на рис. 35а в ветке «Наклейка кафеля» мы, ради эксперимента, поменяли местами иконы «вопрос»:

- все плитки приклеены;
- запас плиток кончился.

Это приведет к «плохому» расположению адресов:

- наклейка кафеля;
- уборка;
- доставка плиток.

И наоборот, вернув иконы «вопрос» на место, мы добьемся желаемого результата. Порядок икон «адрес» будет совпадать с расположением веток.

§6. КАК ОПИСАТЬ СИЛУЭТ С ПОМОЩЬЮ ТЕКСТОВОГО ЯЗЫКА (ПСЕВДОКОДА)?

Дорогой читатель! Это трудный параграф. При желании его вполне можно опустить.

Алгоритмы можно описывать по-разному. Например, с помощью текста (рис. 36) или с помощью графического языка ДРАКОН. Язык ДРАКОН имеет два лица: текстовое и визуальное.

В этой книге описан графический (визуальный) ДРАКОН. Текстовый язык не рассматривается. Но есть исключение. Если книга попадет программисту, у него наверняка возникнет вопрос. Как преобразовать графический язык ДРАКОН в текстовую форму?

В этом параграфе дан ответ, предназначенный для программистов.

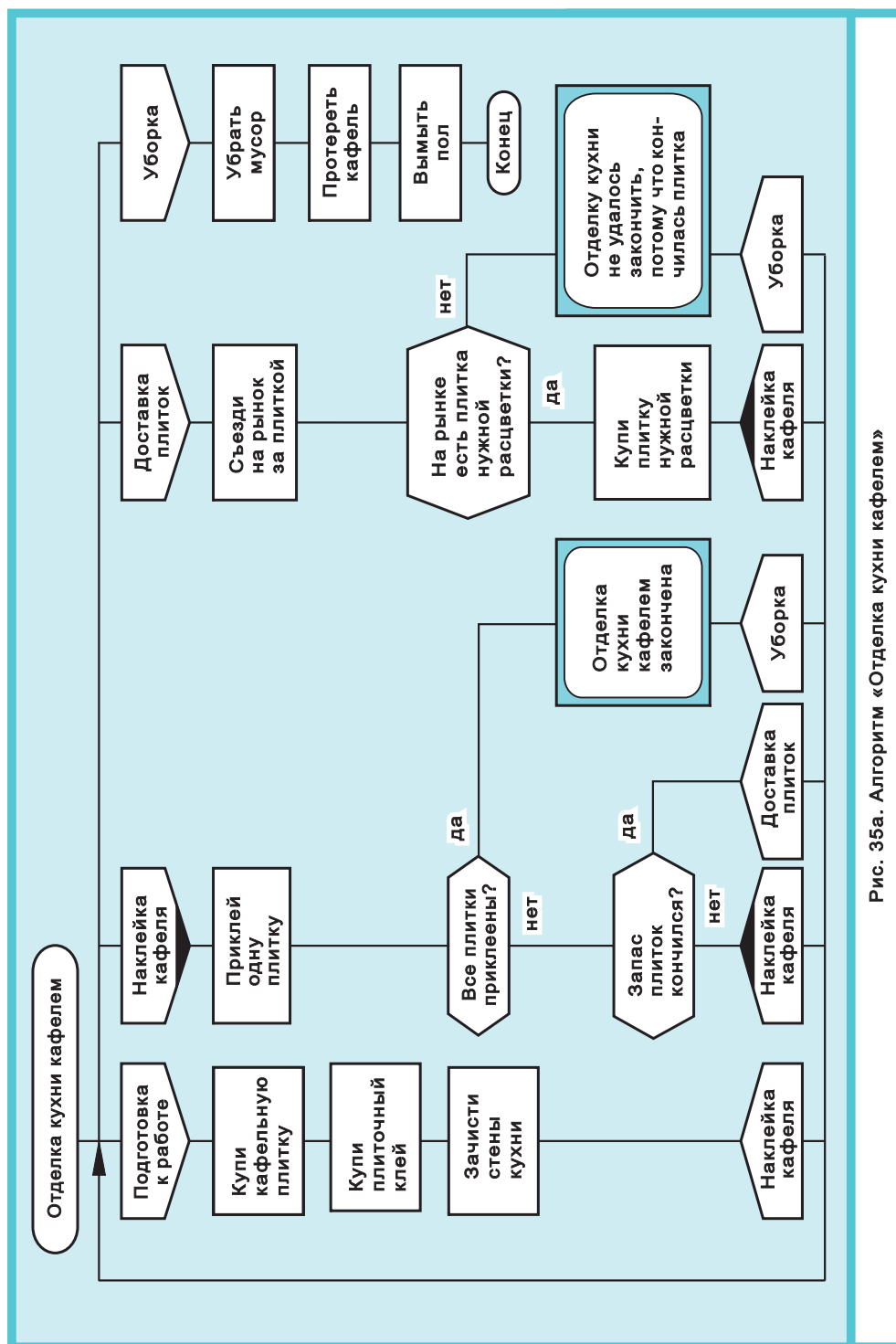


Рис. 35а. Алгоритм «Отделка кухни кафелем»

На рис. 36 видно, что для описания веток в текстовый язык (псевдокод) 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 записано справа от оператора АДРЕС.

Сравнивая две формы языка ДРАКОН (графическую и текстовую), можно заметить, что соответствующие алгоритмы (рис. 34 и 36) эквивалентны¹. Однако графический язык, несомненно, более нагляден и доходчив.

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

Повторим еще раз: в данной книге текстовая форма языка ДРАКОН не используется.

§7. ГРАФИЧЕСКИЙ И ТЕКСТОВЫЙ СИНТАКСИС ДРАКОНА

ДРАКОН – графический (визуальный) язык, в котором используются два типа элементов:

- графические фигуры (иконы),
- текстовые надписи, расположенные внутри или снаружи икон (*текстоэлементы*).

Следовательно, синтаксис ДРАКОНа распадается на две части.

Графический (визуальный) синтаксис охватывает алфавит икон, правила их размещения в поле чертежа и правила связи икон с помощью соединительных линий. Графический синтаксис описан в главе 33.

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

Оператор
языка ДРАКОН

Это икона или комбинация икон,
взятые вместе с текстовыми надписями

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 или хотя бы втиснуться? = ДА
                АДРЕС Посадка в автобус
        M1: ИНАЧЕ
            КОММЕНТАРИЙ Жди прихода следующего автобуса
            АДРЕС Ожидание посадки
            КОНЕЦ ЕСЛИ
        ИНАЧЕ
            ПЕРЕХОД НА M1
        КОНЕЦ ЕСЛИ
КОНЕЦ ВЕТКИ
ВЕТКА Посадка в автобус
    ВЫПОЛНИТЬ Войди в автобус
    ЕСЛИ В салоне есть свободные сидячие места? = ДА
        ВЫПОЛНИТЬ Сядь на свободное место и расслабься
        АДРЕС Поездка
    ИНАЧЕ
        ЕСЛИ Есть желание ехать стоя? = ДА
            АДРЕС Поездка
        ИНАЧЕ
            ВЫПОЛНИТЬ Выйди из автобуса
            КОММЕНТАРИЙ Жди прихода следующего автобуса
            АДРЕС Ожидание посадки
        КОНЕЦ ЕСЛИ
    КОНЕЦ ЕСЛИ
КОНЕЦ ВЕТКИ
ВЕТКА Поездка
    ЕСЛИ Есть деньги на билет? = ДА
        ВЫПОЛНИТЬ Купи билет и поезжай спокойно
    ИНАЧЕ
        КОММЕНТАРИЙ Езжай зайцем и дрожи
    КОНЕЦ ЕСЛИ
    ВЫПОЛНИТЬ Езжай до конечной остановки
    ВЫПОЛНИТЬ Выйди из автобуса
КОНЕЦ

```

Рис. 36. Текстовая запись алгоритма, соответствующая дракон-схеме на рис. 34

Одновременное использование графики и текста говорит о том, что ДРАКОН адресуется не только к словесно-логическому мышлению автора и читателя алгоритма. Сверх того, он активизирует интуитивное, образное, правополушарное мышление, стимулируя его не написанным, а именно нарисованным алгоритмом, то есть алгоритмом-картинкой.

§8. ВЫВОДЫ

1. Силуэт – дракон-схема, разделенная на ветки.
2. Примитив – дракон-схема, не имеющая веток.
3. Шампур ветки – это вертикаль, соединяющая икону «имя ветки» с иконой «адрес», а если у ветки несколько выходов – с левым из них.
4. Главный маршрут ветки должен идти по шампуру ветки.
5. Побочные маршруты ветки следует упорядочить слева направо по какому-либо критерию.
6. Чем правее (чем дальше от шампура данной ветки) расположена очередная вертикаль, тем менее успешные действия она выполняет.
7. Главный маршрут силуэта – последовательное соединение главных маршрутов поочередно работающих веток.
8. Левый маршрут каждой ветки должен идти по прямой вертикальной линии – шампуру ветки.
9. В примитиве только один шампур. А в силуэте их несколько.
10. Каждая ветка имеет свой шампур.
11. В силуэте число шампуров равно числу веток.
12. Синтаксис языка ДРАКОН распадается на две части.
13. Графический синтаксис охватывает алфавит икон, правила их размещения в поле чертежа и правила связи икон с помощью соединительных линий.
14. Текстовый синтаксис задает алфавит символов, правила их комбинирования и привязку к иконам.

КАК УЛУЧШИТЬ ПОНЯТНОСТЬ АЛГОРИТМОВ

§1. ВВЕДЕНИЕ

Наша цель – сделать алгоритмы понятными, приятными для глаза, эргономичными. Для этого надо действовать не вслепую, а использовать специальные методы. В этой главе будут изложены математические методы, позволяющие приблизиться к намеченной цели.

Глава делится на три части. Сначала мы рассмотрим понятия «развилка», «плечо развилки», «формула маршрута». После этого расскажем о математике. А затем на многочисленных примерах покажем, что математические приемы действительно позволяют улучшить понятность дракон-схем.

§2. ЧТО ТАКОЕ РАЗВИЛКА?

Развилка

- Это часть дракон-схемы, внутри которой маршрут сначала раздваивается, а затем соединяется в точке слияния.
- Развилка имеет один вход сверху и один выход снизу, лежащие на одной вертикали (рис. 37).

В состав развилки входят четыре элемента:

- икона «вопрос»,
- левое плечо,
- правое плечо,
- точка слияния.

§3. ЧТО ТАКОЕ ПЛЕЧО РАЗВИЛКИ?

Суть дела ясна из примера на рис. 37.

Левое плечо – это маршрут от нижнего выхода иконы «вопрос» (точка А) до точки слияния Д. Оно содержит ответ «да», три иконы и соединительные линии.

Правое плечо – это путь от правого выхода до точки слияния. На рис. 37 оно содержит ответ «нет» и линию БВГД.

Чем отличается икона «вопрос» от развилки? Ответ дан на рис. 38.

Правило. У развилки два плеча, левое и правое.

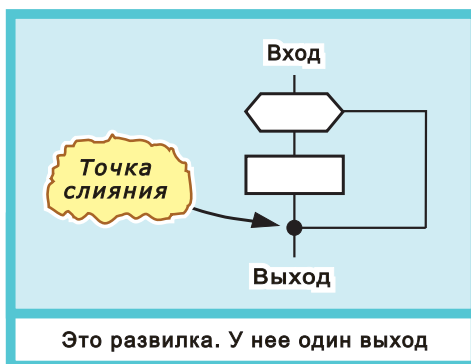
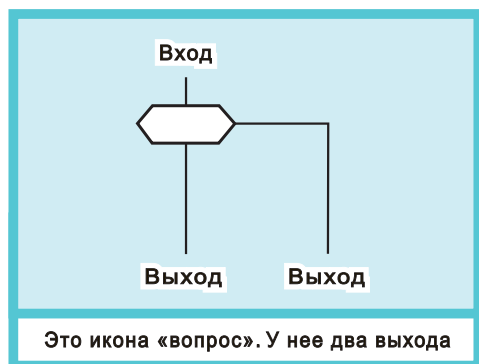
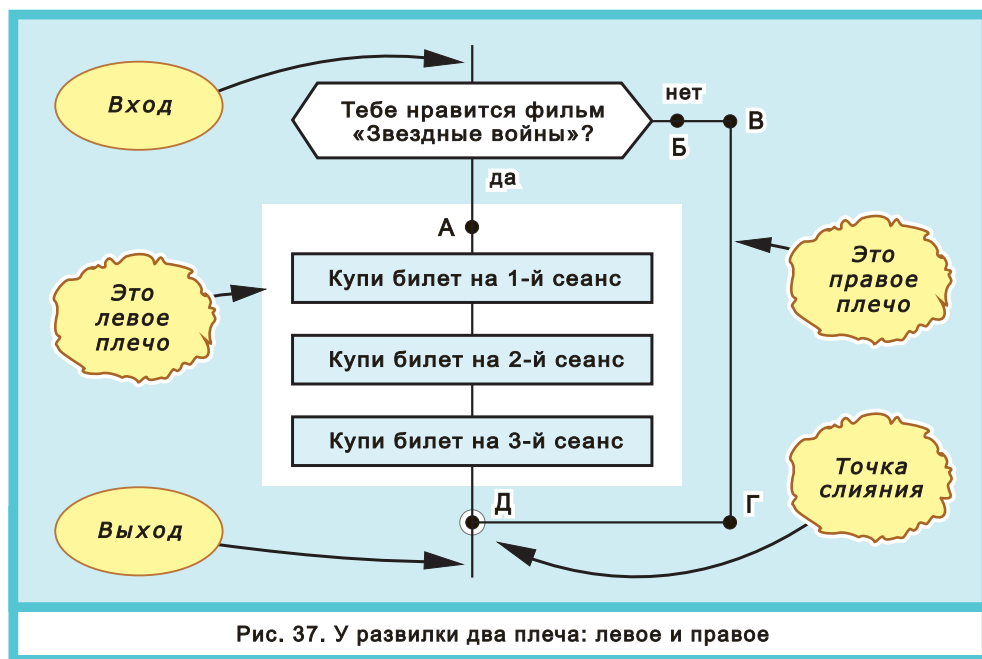


Рис. 38. Чем различаются развилка и икона «вопрос»?

§4. МАРШРУТЫ И ФОРМУЛЫ МАРШРУТОВ

Что такое маршрут

Это графический путь от начала до конца алгоритма, проходящий через иконы и соединительные линии.

На рис. 39 представлена схема «Охота на мамонта». Заменяем текст внутри икон буквами. Вместо «Охота на мамонта» запишем букву А, вместо «Поймай мамонта» – букву Б и т. д. В результате получим буквенную дракон-схему на рис. 39 (справа). Буквенные схемы удобно использовать для описания маршрутов.

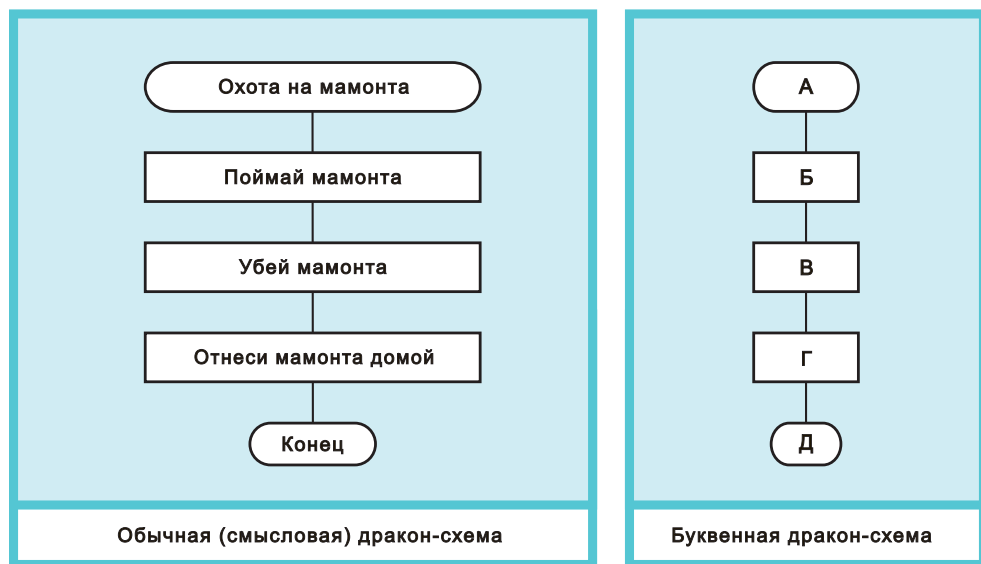


Рис. 39. Как преобразовать обычную дракон-схему в буквенную?

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. 39 (справа) описывается формулой

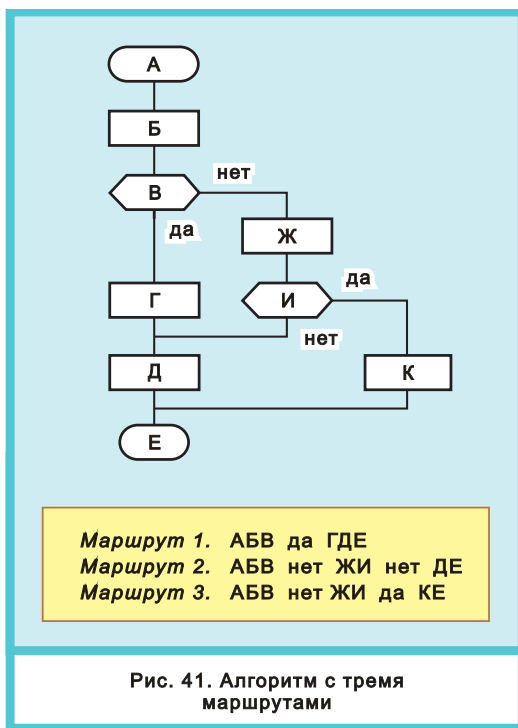
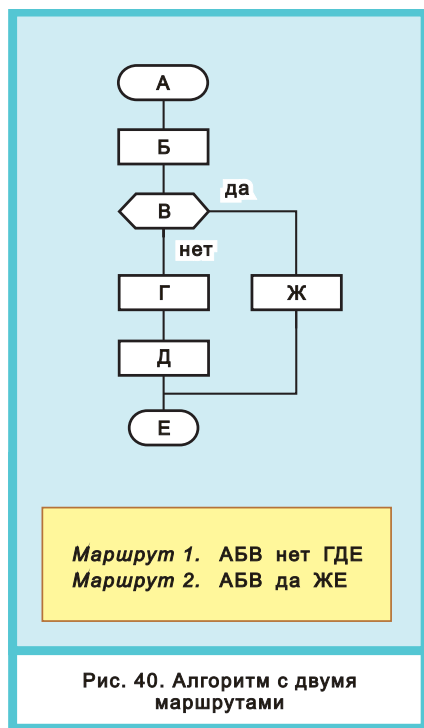
АБВГД

§5. МАРШРУТЫ РАЗВЕТВЛЕННОГО АЛГОРИТМА

Разветвленный алгоритм имеет несколько (два и более) маршрутов. У каждого маршрута своя, отличная от других формула. В формулах раз-

ветвленных алгоритмов наряду с буквами, обозначающими иконы, используются слова «да» и «нет», отделяемые пробелами.

Примеры формул показаны на рис. 40, 41.



§6. НАБОР МАРШРУТОВ

Что такое
набор маршрутов

- Это множество маршрутов данного алгоритма.
- Набор маршрутов и набор формул — одно и то же.

В разных алгоритмах разное число маршрутов. На рис. 40 два маршрута, на рис. 41 — три.

Чтобы написать набор маршрутов данного алгоритма, необходимо:

- преобразовать алгоритм в буквенную форму;
- для каждого маршрута буквенного алгоритма написать формулу маршрута.

Совокупность таких формул — это и есть набор маршрутов данного алгоритма.

§7. РАВНОСИЛЬНЫЕ АЛГОРИТМЫ

Равносильные
алгоритмы

Это алгоритмы, имеющие
одинаковый набор маршрутов
(одинаковый набор формул)

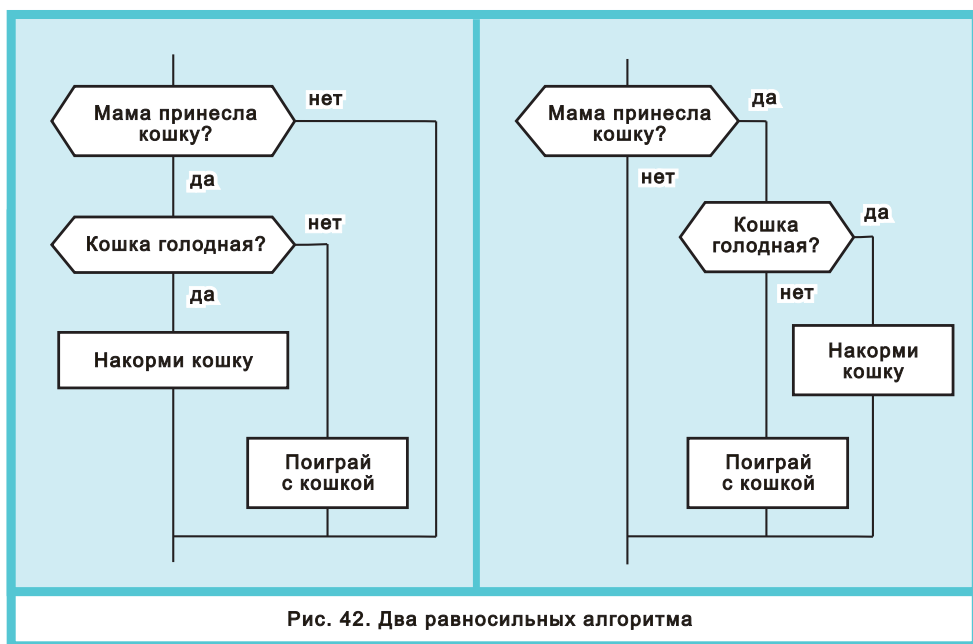
Как доказать, что алгоритмы X и Y равносильны?

Для этого нужно:

- определить набор маршрутов алгоритма X ;
- определить набор маршрутов алгоритма Y ;
- убедиться, что два набора формул совпадают.

Рассмотрим пример на рис. 42 и 43.

На рис. 42 видно, что два изображенных на нем алгоритма имеют один и тот же смысл, так как они выполняют одни и те же действия.



Докажем, что эти алгоритмы равносильны. Для этого преобразуем их в буквенную форму (рис. 43). Затем для каждого алгоритма составим набор маршрутов.

Для левого алгоритма эта работа выполняется так.

Сначала пишем формулу маршрута, идущего по шампуню:

Маршрут 1. А да В да Г

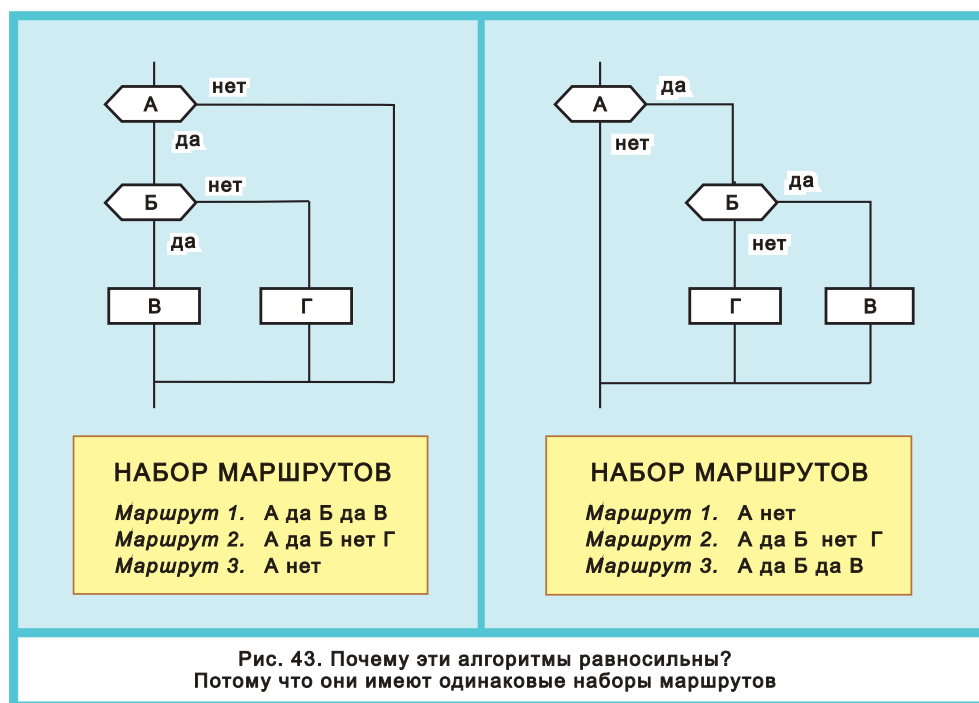
Затем, двигаясь вправо от шампура, поочередно описываем маршруты:

Маршрут 2. А да В нет Г

Маршрут 3. А нет

После этого повторяем эти действия для правого алгоритма.

В заключение сравниваем наборы маршрутов левого и правого алгоритмов на рис. 43. И убеждаемся, что они совпадают (с точностью до порядка записи формул). Отсюда вытекает, что дракон-схемы на рис. 42 равносильны, что и требовалось доказать.



§8. РОКИРОВКА

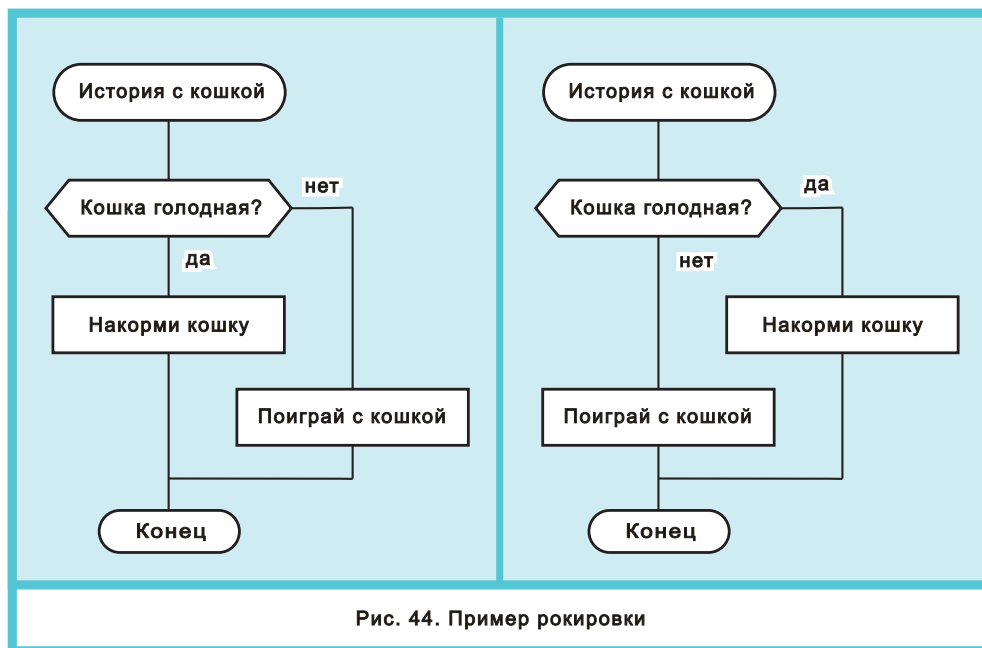
Что такое
рокировка

- Это преобразование алгоритма, при котором левое и правое плечи развилки меняются местами.
- При этом слова «да» и «нет» также меняются местами

Пример рокировки показан на рис. 44.

Формальное преобразование алгоритма X в алгоритм Y называется равносильным, если алгоритмы X и Y равносильны. Сказанное означает, что рокировка является равносильным преобразованием алгоритмов.

Что такое рокировка?
Это плеч перестановка!



§9. ТЕОРЕМА РОКИРОВКИ

Теорема. Если к визуальному алгоритму X применить операцию «рокировка», получим визуальный алгоритм Y , эквивалентный алгоритму X .

Это значит, что рокировка – эквивалентное преобразование алгоритма¹. Можно сказать и по другому. При рокировке смысл алгоритма не меняется.

Визуальная формула рокировки показана на рис. 45.

¹ Два алгоритма называются *эквивалентными*, если они дают одинаковые результаты для одних и тех же исходных данных. Пример: два алгоритма на рис. 44 эквивалентны.

РОКИРОВКА (визуальная формула)

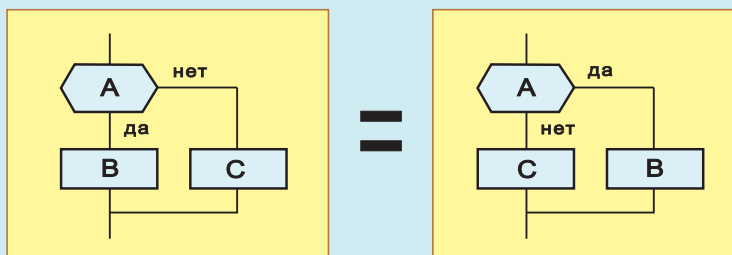


Рис. 45. При рокировке плечи развилки меняются местами

§10. РОКИРОВКА МОЖЕТ УЛУЧШИТЬ ЭРГОНОМИЧНОСТЬ АЛГОРИТМОВ

Полученный результат чрезвычайно важен. Ведь *рокировка позволяет улучшить наглядность и эргономичность алгоритмов*. Давайте посмотрим, как это делается.

На рис. 46 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.

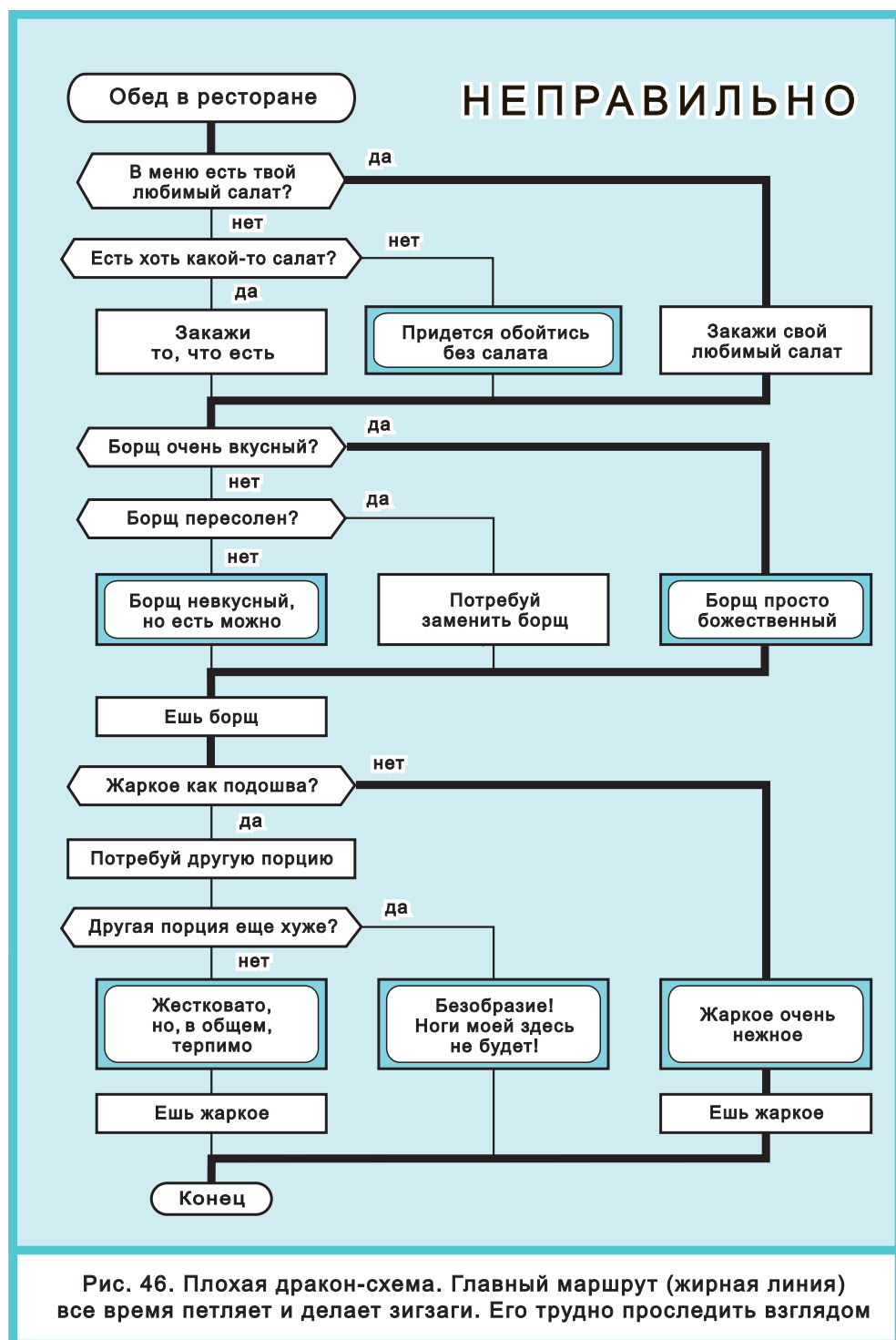
Первый раз делаем рокировку в развилке «В меню есть ваш любимый салат?». Это позволяет пустить главный маршрут по шампуру на верхнем участке. Однако внизу главный маршрут по-прежнему совершает неоправданные зигзаги.

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?».

В результате трех рокировок неэргономичная схема на рис. 46 превратилась в хорошую (эргономичную) схему на рис. 47.

Подведем итоги. Выпрямляя главный маршрут, мы делаем алгоритм более наглядным и легким для понимания. Главный маршрут – путеводная нить алгоритма, позволяющая быстрее уяснить суть дела и не заблудиться в хороводе развилочек.



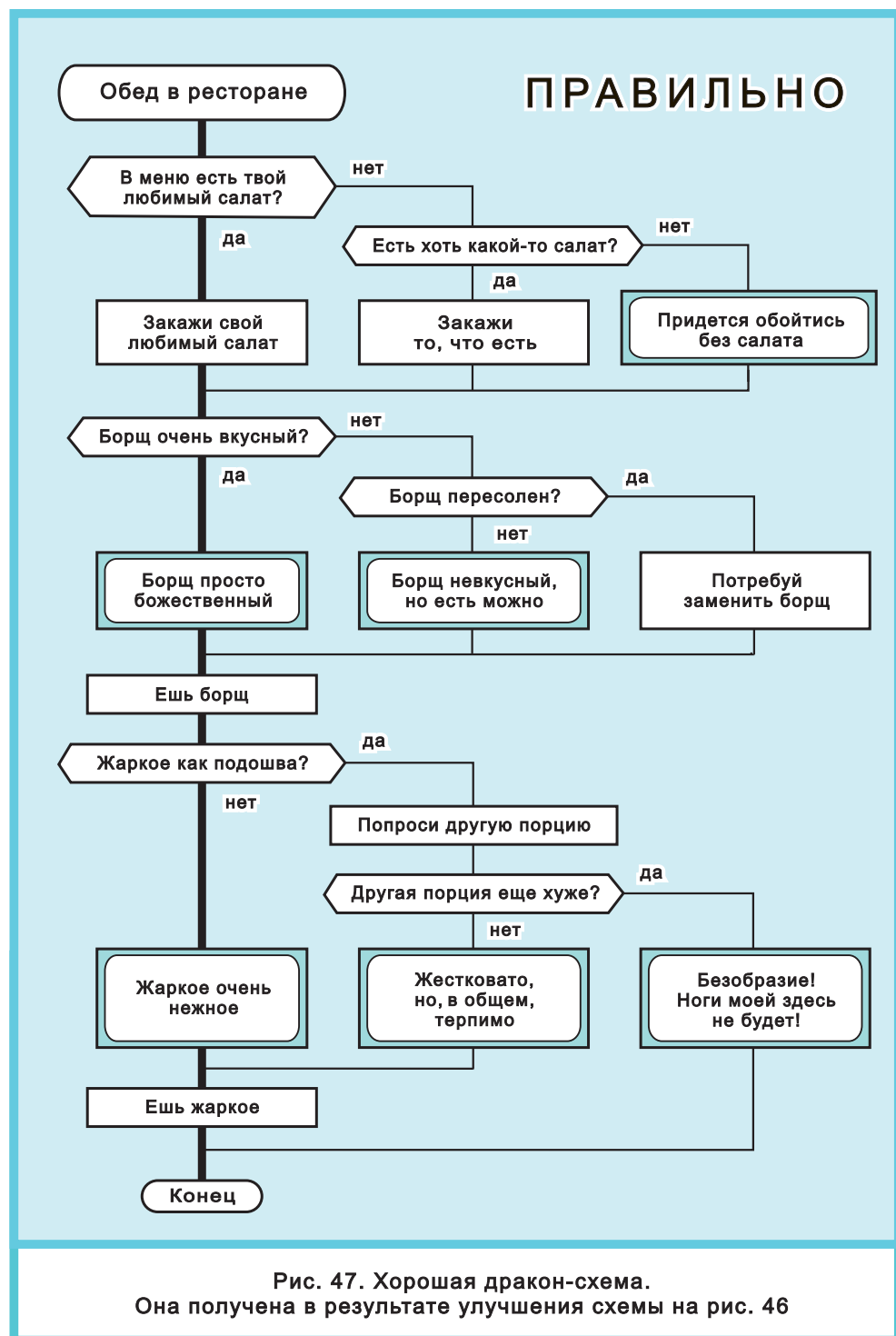


Рис. 47. Хорошая дракон-схема.
Она получена в результате улучшения схемы на рис. 46

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. При рокировке смысл алгоритма не меняется.

Закон рокировки придает нашим *эргономическим* действиям (позволяющим выпрямить «кривой» главный маршрут) *математическую* строгость и точность.

§11. ЕЩЕ ОДИН ПРИМЕР РОКИРОВКИ

На рис. 48–51 представлены четыре схемы, на которых описана история с хрустальной вазой. 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 «Хрустальная ваза свалилась на пол?» главный маршрут идет через «нет». Потому что, когда вещи падают, это плохо. А когда не падают – хорошо. Значит, две схемы – на рис. 48 и 49 – нарисованы неверно.

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

Проверим правило побочных маршрутов. На рис. 48–51 их два. Один описывает ситуацию, когда ваза упала, но уцелела. Во втором случае она разбилась. Первой ситуации выставим оценку «плохо», второй – «очень плохо».

Отсюда делаем вывод, что на рис. 50 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. 51). 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. 48 в хорошую на рис. 51, достаточно сделать всего две рокировки в обеих развилках.

Мы еще раз убедились, что преобразование «рокировка» позволяет улучшить эргономичность алгоритмов.

Сделаем оговорку. 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 буквенным схемам на рис. 40, 41, 43 бессмысленно, так как в данном случае рокировка не влияет на эргономичность.

НЕПРАВИЛЬНО НАРИСОВАННАЯ СХЕМА

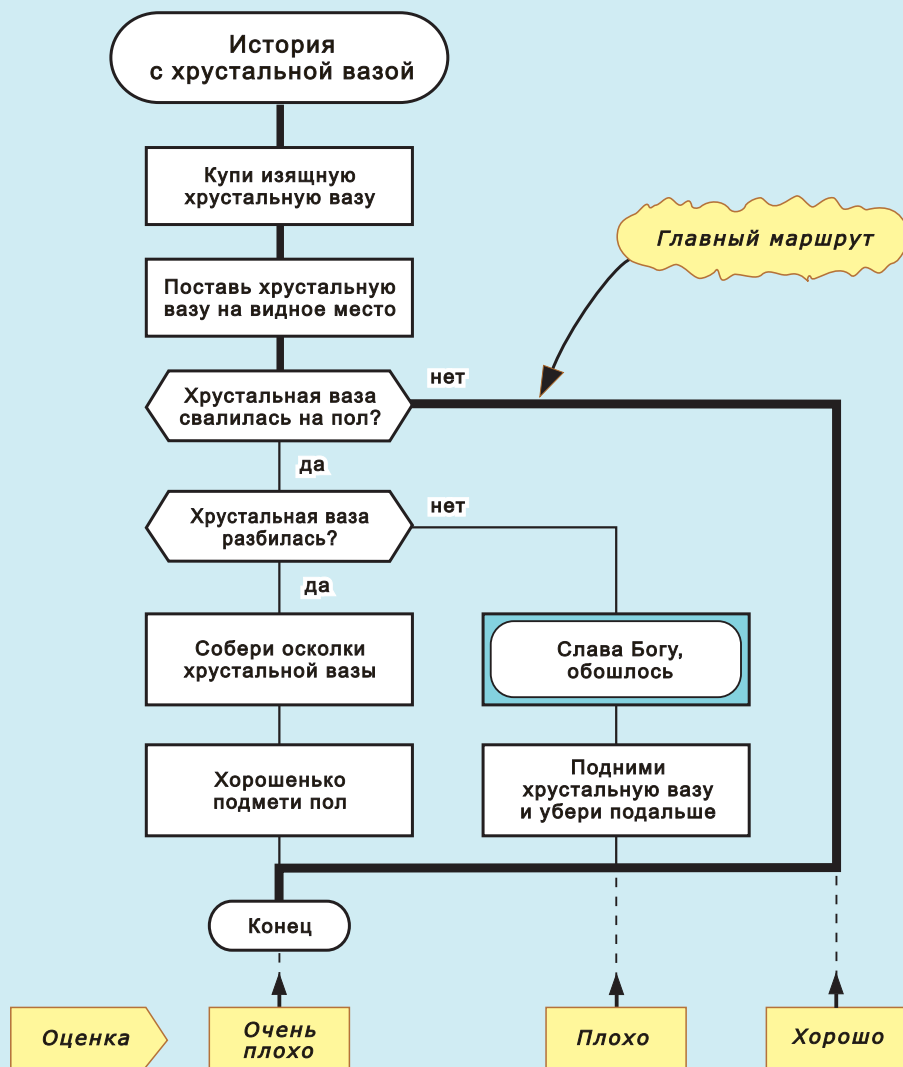


Рис. 48. Очень плохая дракон-схема. В ней две ошибки:

- главный маршрут не идет по шампуру
- нарушено правило «Чем правее, тем хуже»

ЕЩЕ ОДНА НЕПРАВИЛЬНО НАРИСОВАННАЯ СХЕМА

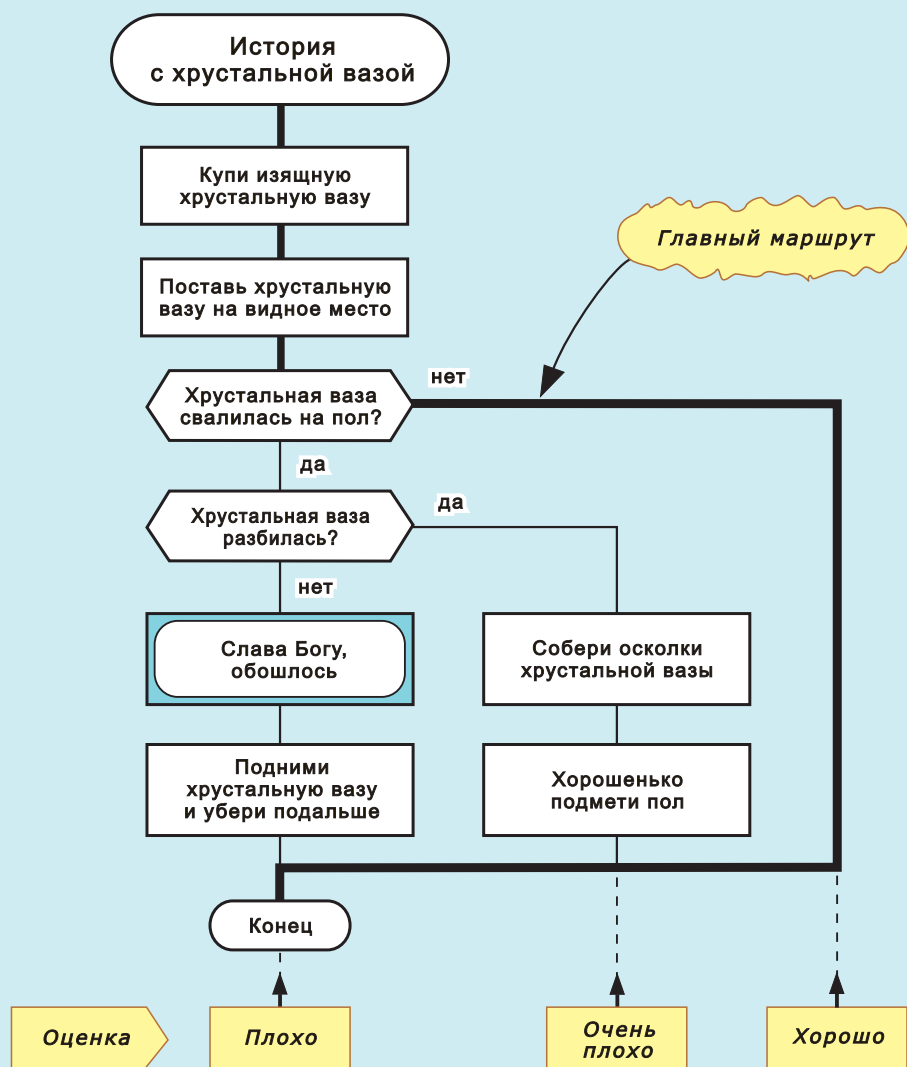


Рис. 49. Плохая дракон-схема.
Ошибка в том, что главный маршрут (жирная линия)
не идет по шампуру

ТРЕТЬЯ НЕПРАВИЛЬНО НАРИСОВАННАЯ СХЕМА

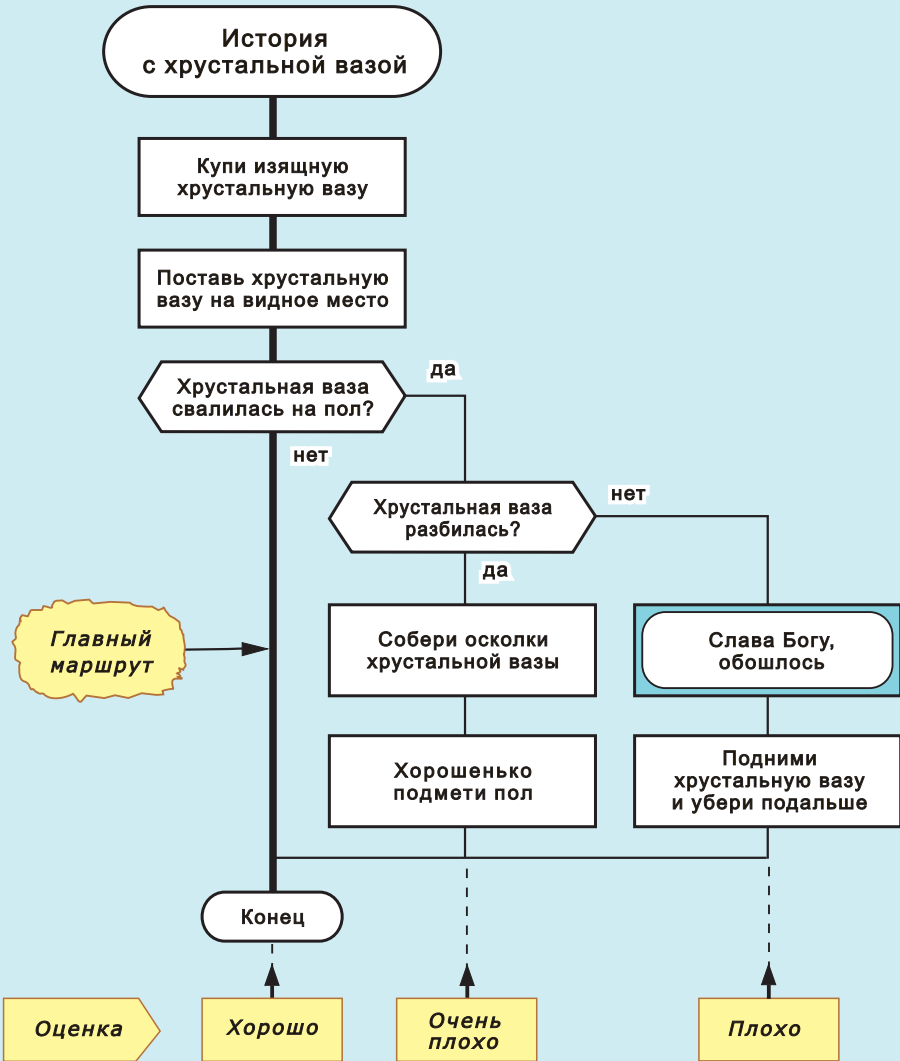


Рис. 50. Плохая дракон-схема.
Нарушено правило «Чем правее, тем хуже»

А ВОТ ЭТО ПРАВИЛЬНО НАРИСОВАННАЯ СХЕМА

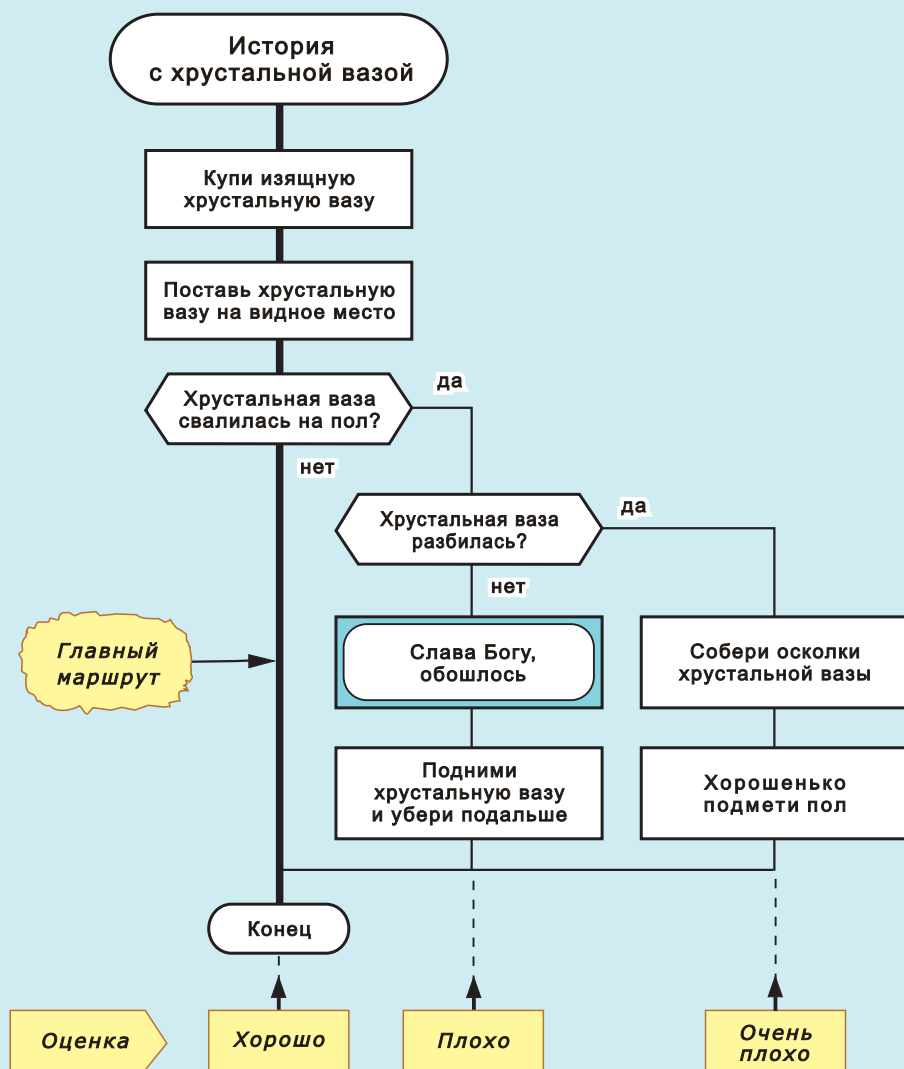


Рис. 51. Хорошая (эргономичная) дракон-схема:

- главный маршрут идет по шампуру
- соблюдается правило «Чем правее, тем хуже»

§12. ВЕРТИКАЛЬНОЕ ОБЪЕДИНЕНИЕ

Иногда бывает, что в дракон-схеме иконы повторяются. Например, на рис. 52 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. 52). В этом случае действует

Правило. Повторы запрещены.

Устранение повторов производится с помощью вертикальной линии (рис. 53) и называется *вертикальным объединением*.

При этом несколько икон объединяются в одну. Это делается так:

- выделяются две или более одинаковых икон, выходы которых присоединены к горизонтальной линии;
- входы упомянутых икон объединяются вертикальной линией,
- удаляются все одинаковые иконы, кроме крайней левой.

Теорема. Если к визуальному алгоритму X применить операцию «вертикальное объединение», получим визуальный алгоритм Y , эквивалентный алгоритму X .

§13. ГОРИЗОНТАЛЬНОЕ ОБЪЕДИНЕНИЕ

Рассмотрим задачу на рис. 54, где также имеются повторы, подлежащие устранению. Сначала устраним повторение иконы «Съешь кашу» и получим схему на рис. 55. В данном случае для исключения повторов используется не вертикальная, а горизонтальная линия (рис. 55). Соответствующая операция называется *горизонтальное объединение*.

При этом несколько икон также объединяются в одну. Для этого нужно:

- выделить две или более одинаковых икон, выходы которых присоединены к горизонтальной линии;
- объединить входы упомянутых икон горизонтальной линией,
- удалить все одинаковые иконы, кроме крайней левой.

Затем исключим повторение развилки «Есть можно?» Окончательный результат показан на рис. 56. Легко заметить, что схема на рис. 56 более удобна и занимает меньше места, чем исходная схема на рис. 54. Самое главное, она стала проще и намного понятнее.

Теорема. Если к визуальному алгоритму X применить операцию «горизонтальное объединение», получим визуальный алгоритм Y , эквивалентный алгоритму X .

НЕПРАВИЛЬНО

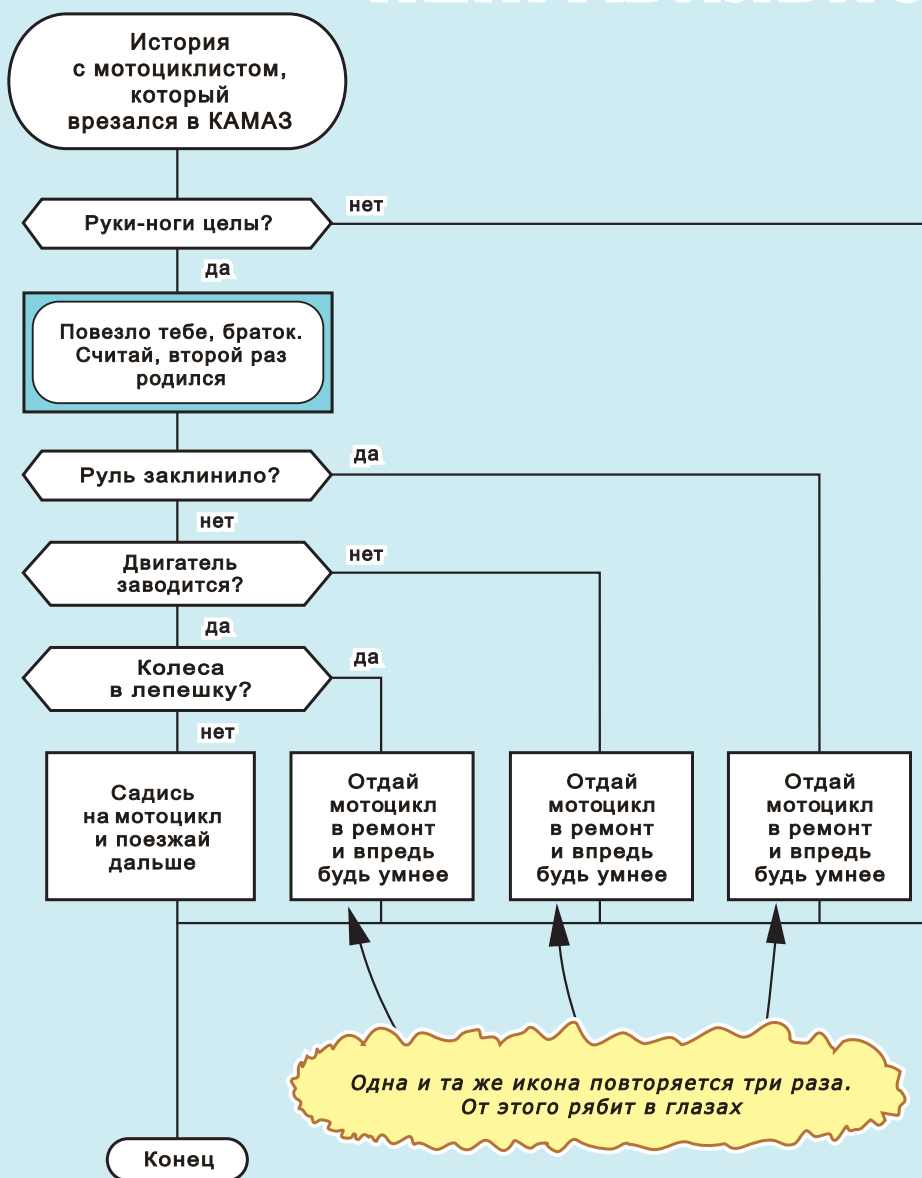


Рис. 52. Плохая дракон-схема.
Нарушено правило «Повторы запрещены».
К счастью, повторы можно убрать (см. рис. 53)

ПРАВИЛЬНО

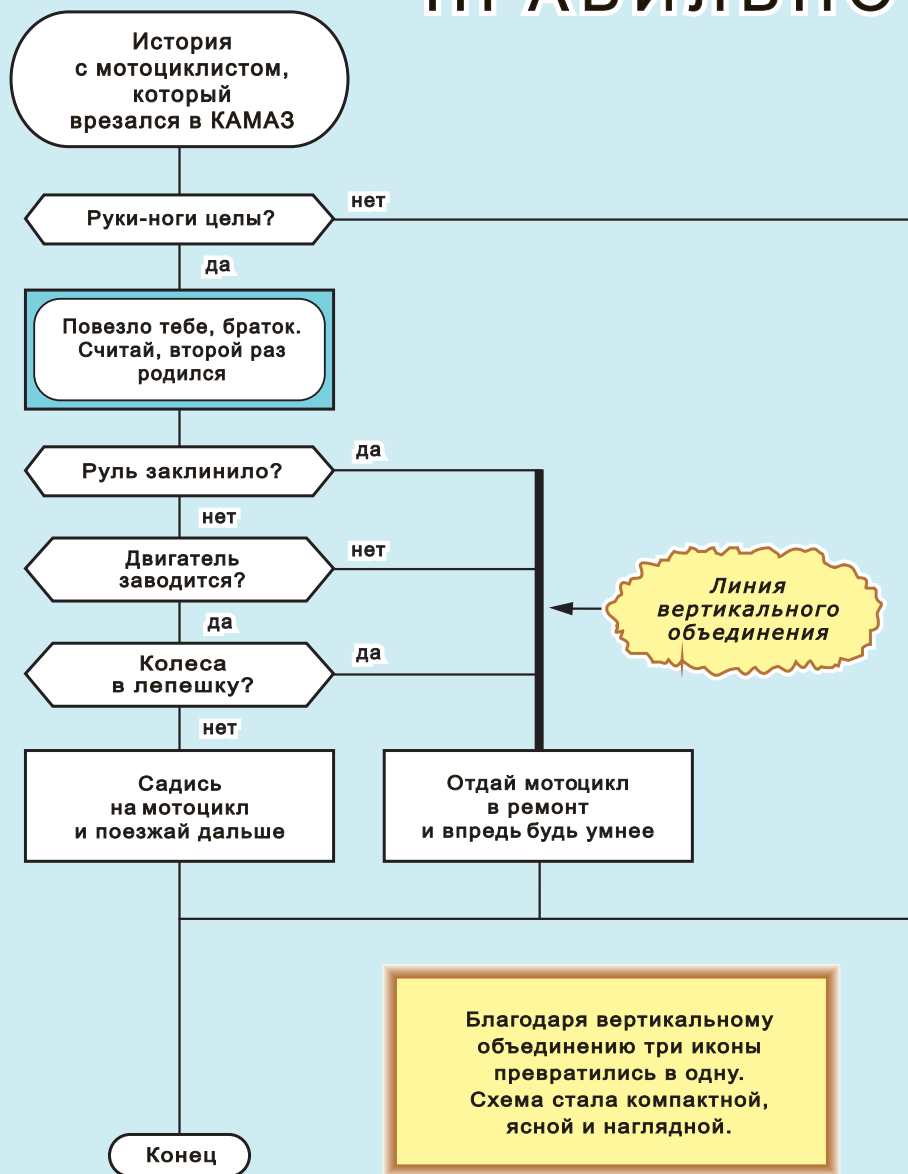
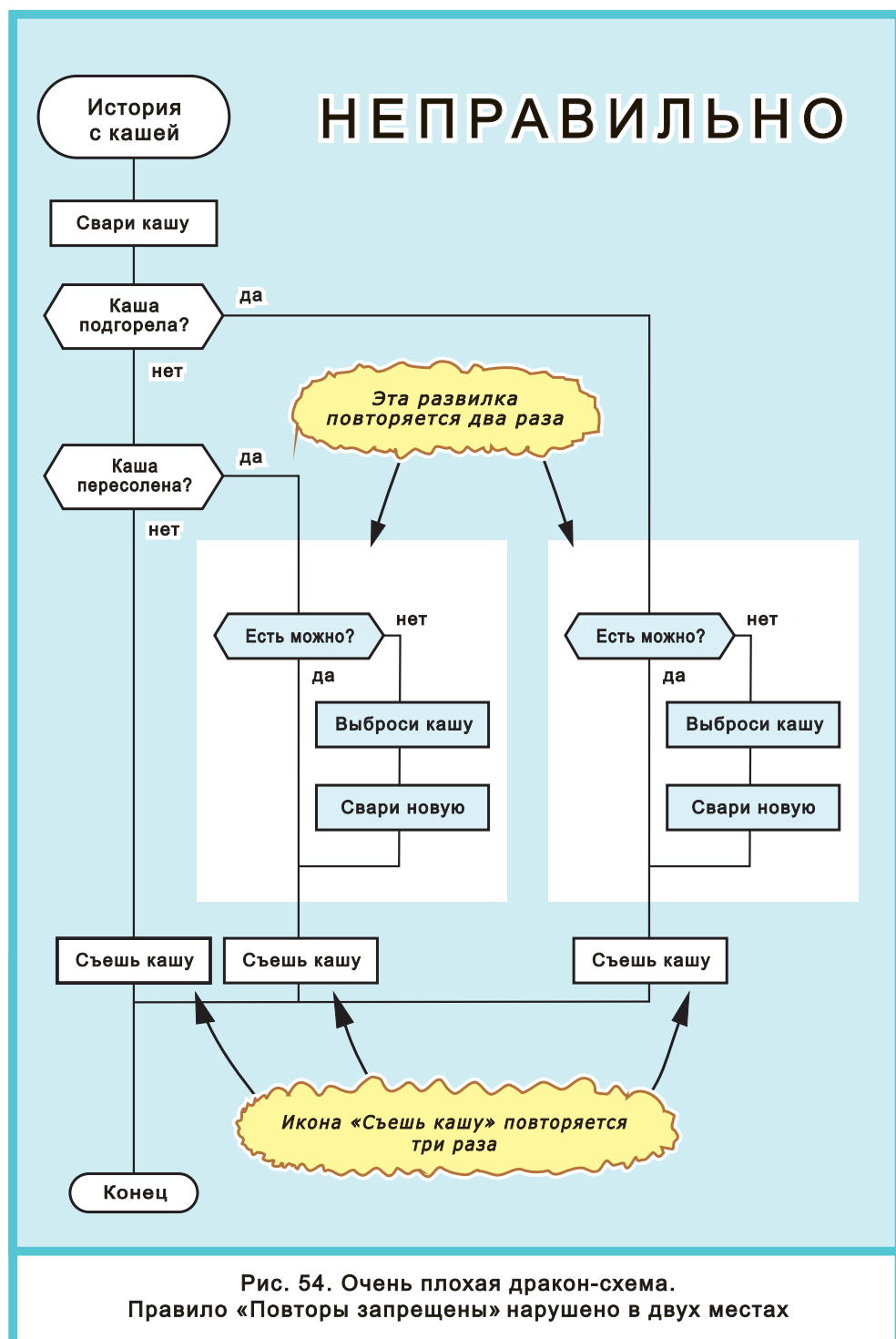
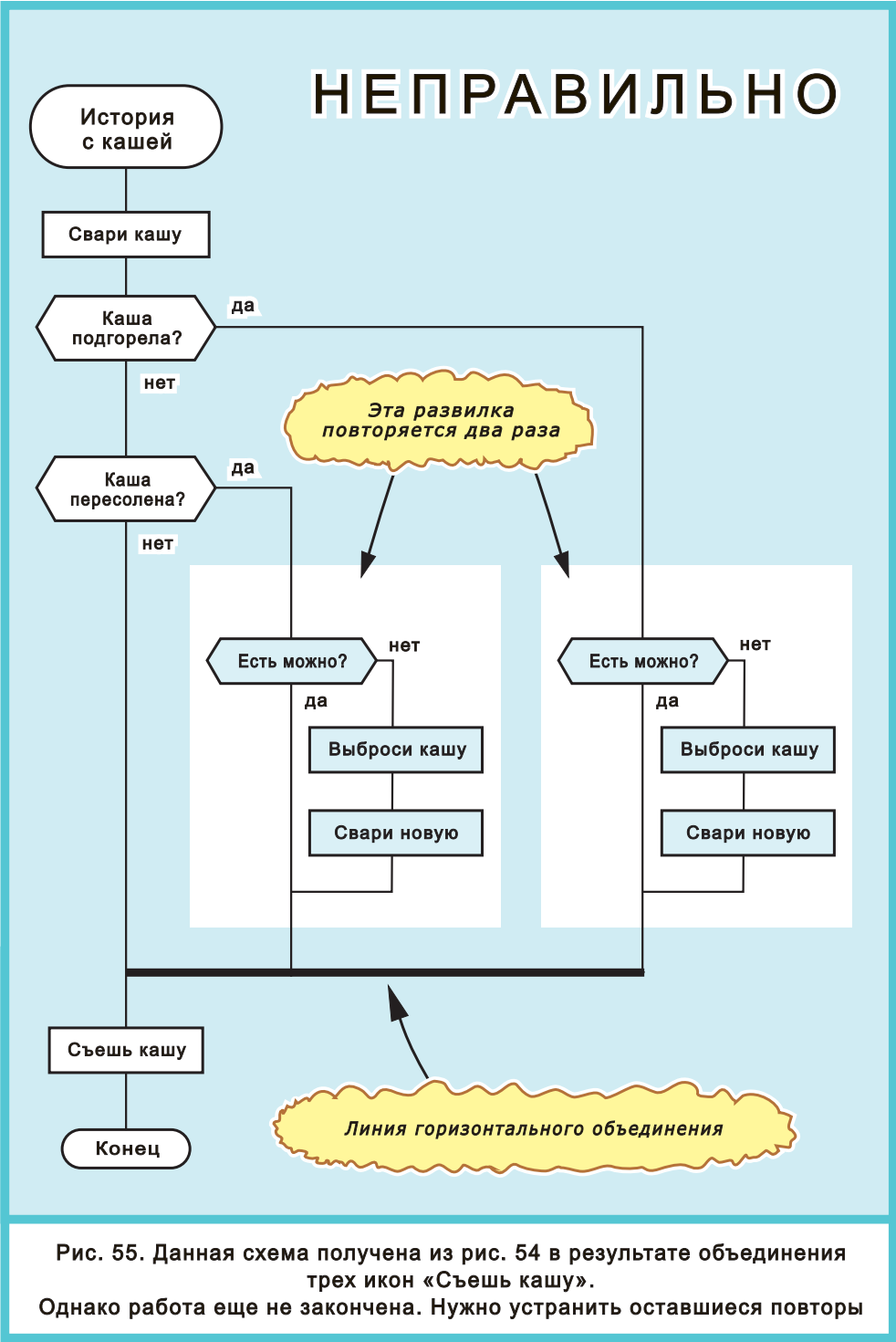
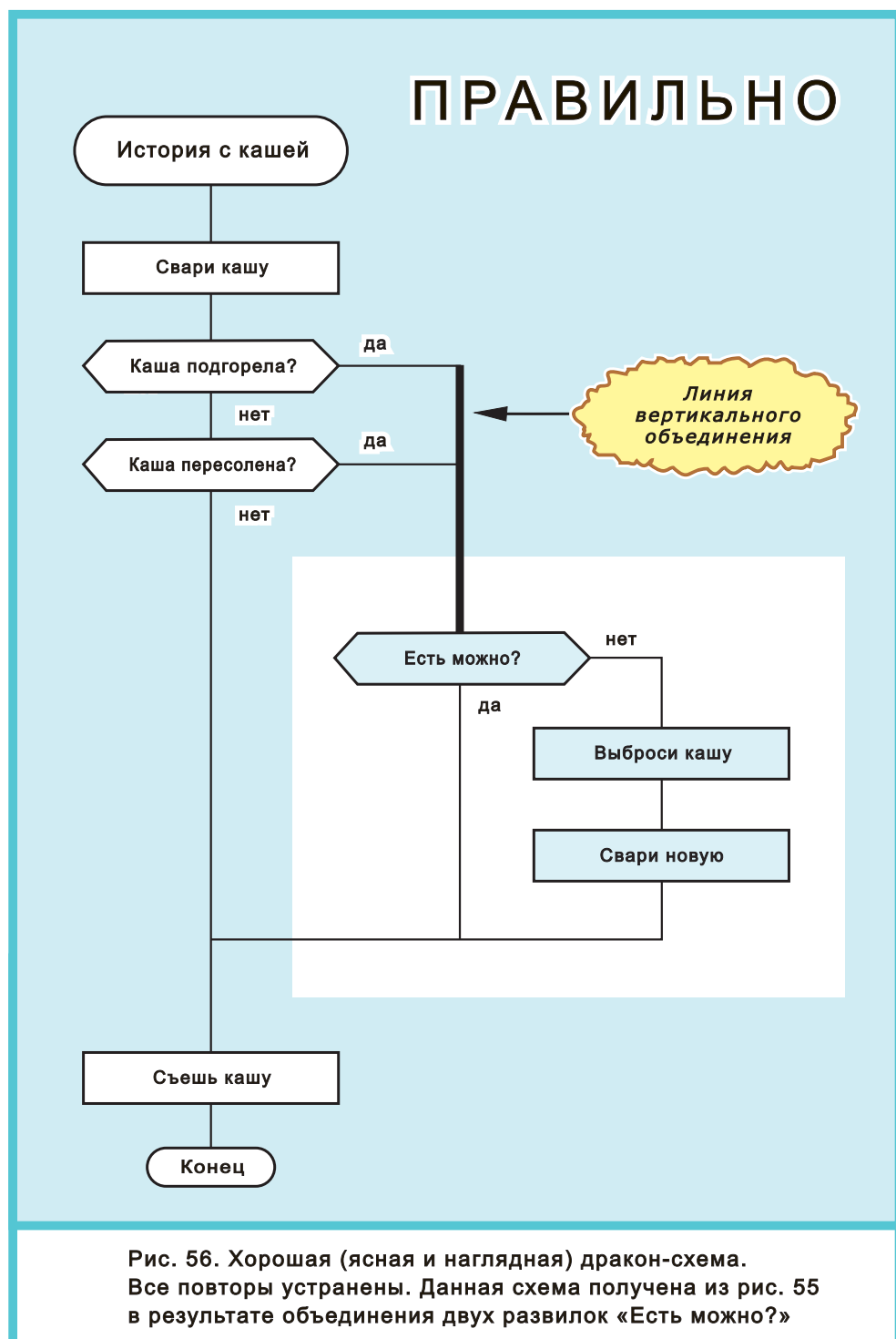


Рис. 53. Хорошая схема. Повторы устранены. Данная схема получена из рис. 52 с помощью операции «Вертикальное объединение»







§14. ВИЗУАЛЬНЫЕ ФОРМУЛЫ ОБЪЕДИНЕНИЯ

Формулы объединения показаны на рис. 57 и 58. На этих формулах для примера расставлены слова «да» и «нет». Следует иметь в виду, что это всего лишь один из возможных примеров расстановки. Возле каждой иконы «вопрос» указанные слова могут быть записаны любым из двух способов:

- «да» внизу, «нет» вправо;
- «нет» внизу, «да» вправо.

На рис. 59 и 60 показан еще один пример вертикального объединения.

Согласно традиционному подходу схема на рис. 60 считается запрещенной. Но с точки зрения графического языка ДРАКОН дело обстоит иначе: схема на рис. 60 разрешена к применению.

Совместное использование вертикального и горизонтального объединения в некоторых случаях позволяет существенно упростить дракон-схемы (см. пример на рис. 61 и 62).

§15. КАК УСТРАНИТЬ ПЕРЕСЕЧЕНИЯ ЛИНИЙ?

На рис. 63 в точке *X* линии пересекаются. Это очень плохо! Ведь пересечение – визуальная помеха. Она затрудняет восприятие и анализ алгоритмов. Обилие пересечений сбивает читателя с толку. Практика показывает: пересечение линий – это источник опасности, который может привести к ошибке.

Поэтому, как мы уже знаем, в языке ДРАКОН действует

Правило. Пересечения линий запрещены.

Существуют специальные приемы позволяющие устранить пересечения. Самый простой из них показан на рис. 63 и 64.

§16. ДОКАЗАТЕЛЬСТВО РАВНОСИЛЬНОСТИ АЛГОРИТМОВ

Покажем, что схемы на рис. 63 (где есть пересечение) и 64 (где нет пересечений) равносильны.

На рис. 63 имеются три маршрута:

Номер маршрута	Формула маршрута
<i>Маршрут 1</i>	А да Б нет В
<i>Маршрут 2</i>	А нет Г
<i>Маршрут 3</i>	А да Б да Д

ВЕРТИКАЛЬНОЕ ОБЪЕДИНЕНИЕ (визуальная формула)

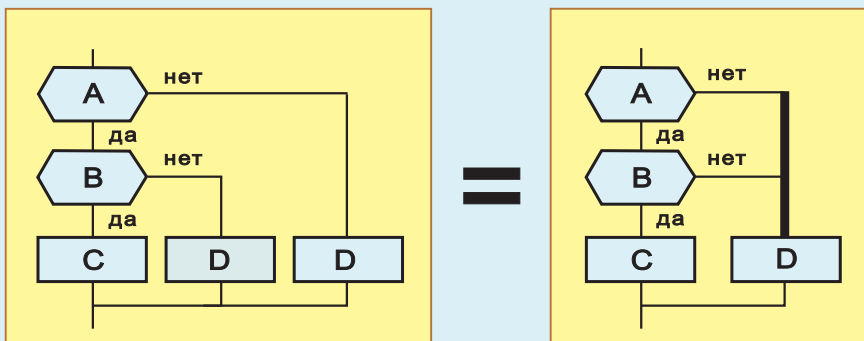


Рис. 57. Преобразование дракон-схемы «Вертикальное объединение»

ГОРИЗОНТАЛЬНОЕ ОБЪЕДИНЕНИЕ (визуальная формула)

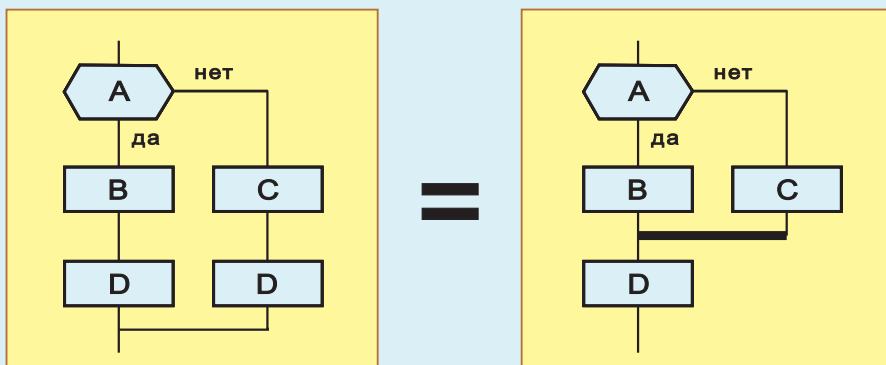
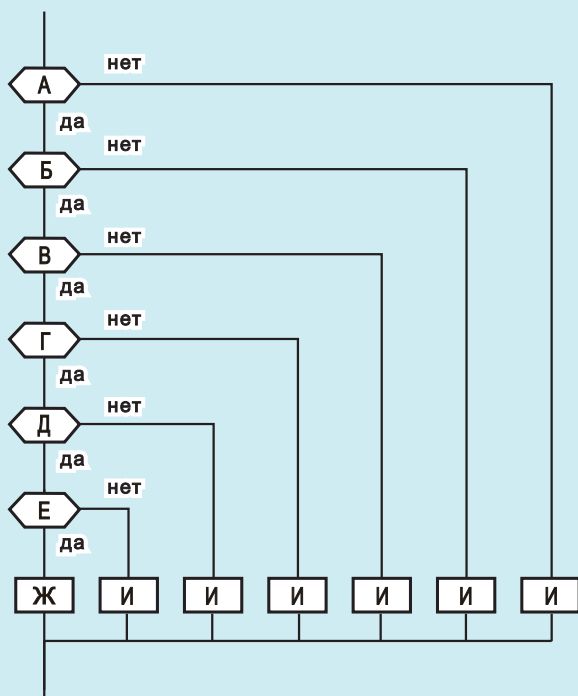
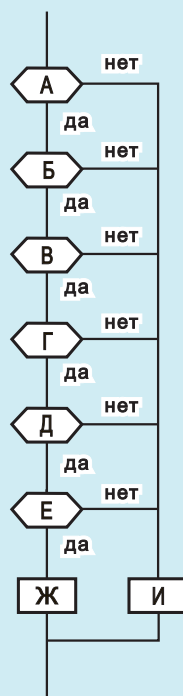


Рис. 58. Преобразование дракон-схемы «Горизонтальное объединение»

НЕПРАВИЛЬНО



ПРАВИЛЬНО

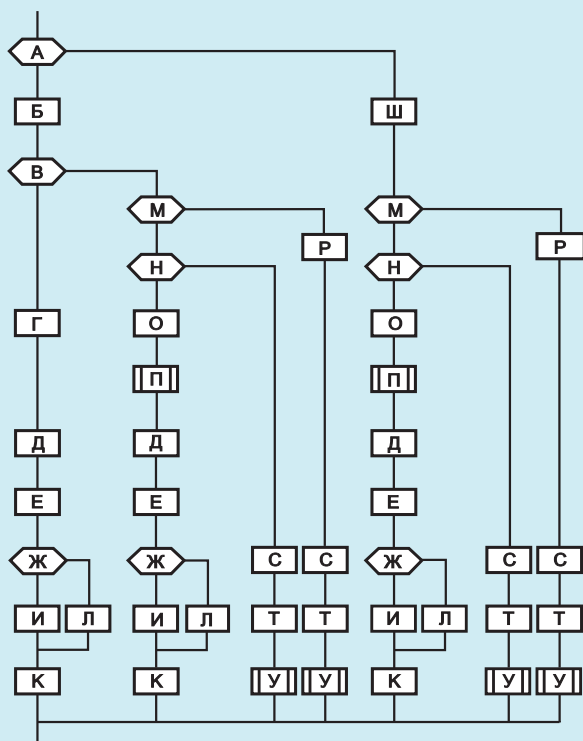


**Равносильное преобразование
«вертикальное объединение»
улучшает эргономичность дракон-схемы**

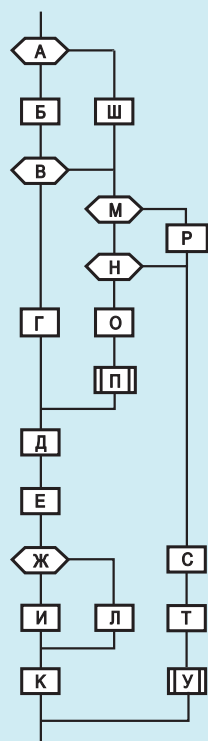
Рис. 59. Плохая схема.
В ней слишком много вертикалей (семь)
и одинаковых икон (шесть).
Кроме того, есть пять лишних изломов,
а пять горизонталей неоправданно длинные

Рис. 60. Хорошая схема.
Число вертикалей,
икон и изломов
удалось значительно
сократить

НЕПРАВИЛЬНО



ПРАВИЛЬНО



Равносильные преобразования «вертикальное объединение» и «горизонтальное объединение» позволяют преобразовать неэргономичный алгоритм (слева) в эквивалентный ему эргономичный алгоритм (справа)

Рис. 61. Плохая схема.
В ней слишком много икон и соединительных линий, без которых вполне можно обойтись

Рис. 62. Хорошая схема.
Число икон и соединительных линий удалось значительно сократить

Легко убедиться, что схема на рис. 64 имеет точно такие же маршруты. Совпадение маршрутов говорит о том, что на обеих схемах представлен один и тот же *набор маршрутов*. Значит, схемы на рис. 63 и 64 равносильны.

§17. КАКАЯ ОШИБКА ПОДСТЕРЕГАЕТ НАС ПРИ ОБЪЕДИНЕНИИ?

На рис. 65 (слева) икона *E* повторяется три раза. На первый взгляд кажется, что две иконы *E* можно убрать с помощью операции «горизонтальное объединение». Действуя подобным образом, получим результат на рис. 65 (в центре).

Мы допустили грубую ошибку! Вспомним, что при рисовании дракон-схем *пересечения запрещены*. А у нас получилось, что две линии пересекаются в точке *X*. Отсюда следует: разрешается объединять не любые одинаковые иконы, а только *соседние*.

Обратите внимание: на рис. 65 (слева) из трех одинаковых икон *E* только две правые являются соседними. А третья (крайняя левая) отделена от них иконой *Ж*. Поэтому она не может участвовать в объединении. Правильный ответ показан справа на рис. 65.

§18. ЧТО ДЕЛАТЬ, ЕСЛИ ЭРГНОМИЧЕСКИЕ 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. Правило маршрутов имеет более высокий приоритет, нежели стремление уменьшить число вертикалей.

В качестве иллюстрации сравним равносильные схемы на рис. 66 и 67. По критерию «минимизация числа вертикалей» выигрывает схема на рис. 66. У нее на одну вертикаль меньше. А как насчет порядка в маршрутах? Попробуем разобраться.

На рис. 66 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.

НЕПРАВИЛЬНО

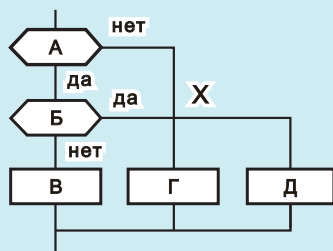


Рис. 63. Плохая схема.
Есть пересечение в точке X

ПРАВИЛЬНО

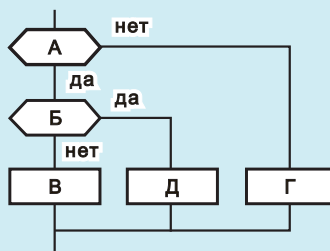
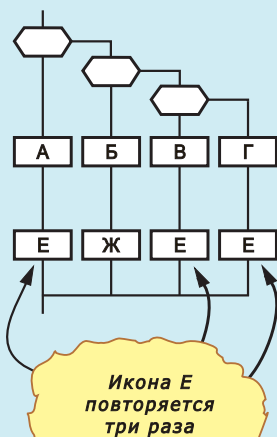


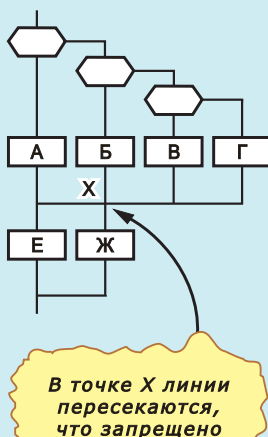
Рис. 64. Хорошая схема.
Пересечение устранено

НЕПРАВИЛЬНО



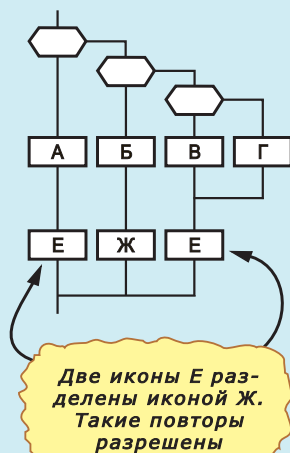
Плохая схема.
Нарушено правило
«Повторы запрещены»

НЕПРАВИЛЬНО



Плохая схема.
Одно лечим, другое
калечим. При устраи-
нении повторов
нарушено правило
«Пересечения зап-
рещены»

ПРАВИЛЬНО



Хорошая схема.
Все ошибки
устранены

Рис. 65. Устраняя повторы, следите, чтобы не появились пересечения

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 схему на рис. 66 нельзя признать эргономичной.

Чтобы исправить недостаток, необходимо выполнить:

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

В итоге получим безукоризненно четкую схему на рис. 67. 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 совсем плохи – пришлось лечь в больницу.

Как уже упоминалось, схема на рис. 67 тоже не без греха – в ней на одну вертикаль больше, чем на рис. 66. Тем не менее, мы признаем ее наилучшей, поскольку выполняется более приоритетное правило маршрутов.

§19. КАК УБРАТЬ «НЕУСТРАНИМОЕ» ПЕРЕСЕЧЕНИЕ ЛИНИЙ?

Мы знаем, что некоторые пересечения удалить очень легко. Иногда для этого достаточно слегка подвинуть линии (рис. 63 и 64).

Но так бывает не всегда. На рис. 68 (слева) показан «трудный орешек».

Как же быть в этом случае? Как устранить досадную помеху? Ответ дает

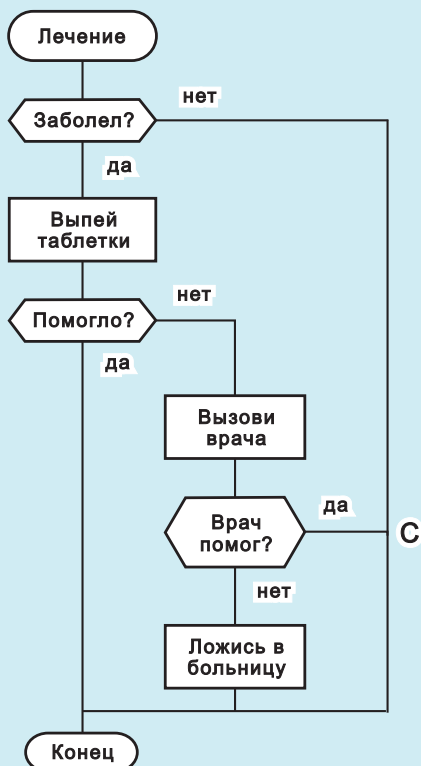
Правило. Чтобы убрать неустрашимое (без введения дополнительных переменных) пересечение линий в блок-схеме, надо превратить блок-схему в дракон-схему силуэт.

На рис. 68 (справа) изображен силуэт, который, как нетрудно убедиться, эквивалентен примитиву (слева) и вместе с тем не содержит ни одного пересечения.²

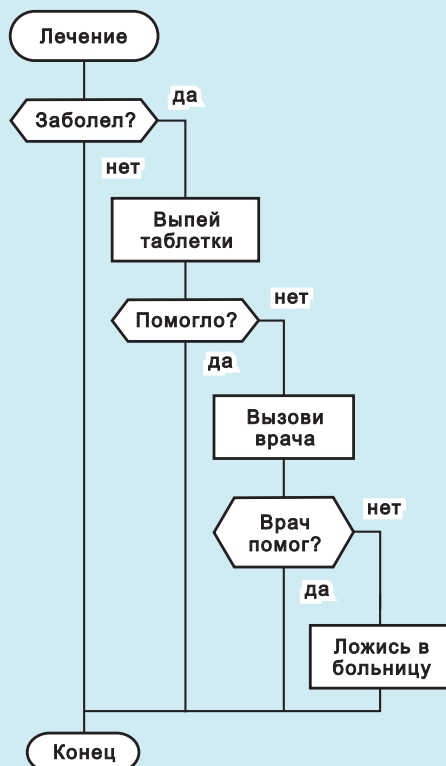
Таким образом, язык ДРАКОН позволяет устранить любое пересечение соединительных линий, используя строгие математические методы. Отсутствие визуальных помех (пересечений) создает дополнительные удобства для читателя, делает дракон-схему прозрачной, облегчает понимание.

² Используемый нами прием опирается на классический метод Ашкрофта-Манна [1].

НЕПРАВИЛЬНО



ПРАВИЛЬНО



Вопрос. Что важнее: уменьшить число вертикалей или навести в схеме уют и порядок?

Ответ. Уют и порядок намного важнее. Поэтому в первую голову надо упорядочить маршруты.

Рис. 66. Плохая схема.

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

Рис. 67. Хорошая схема.

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

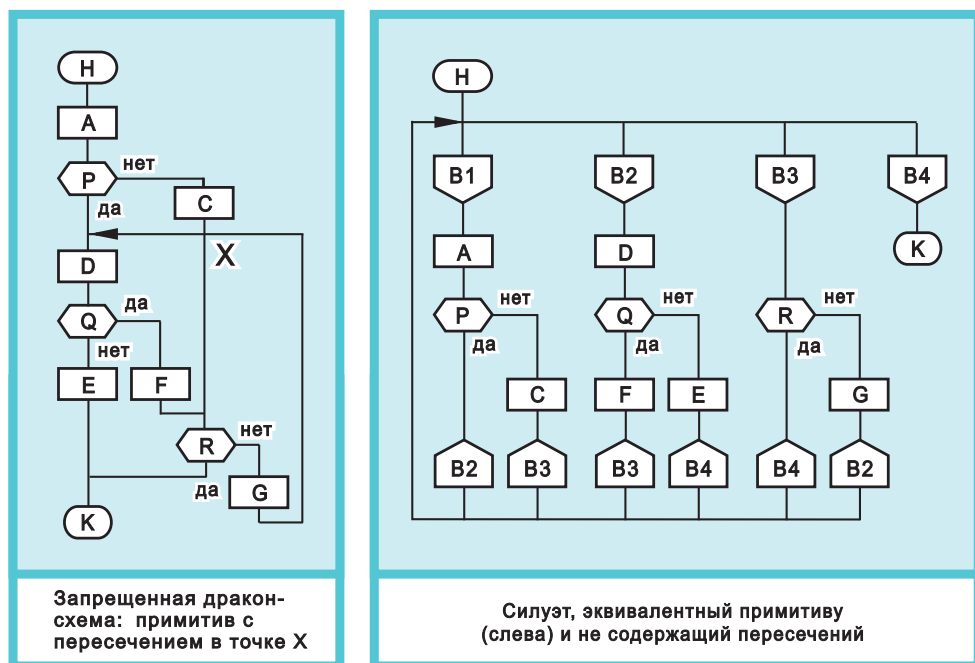


Рис. 68. Преобразование примитива в эквивалентный ему силуэт позволяет устранить любые пересечения линий

§20. ИКОНА «ВСТАВКА» КАК ЭРГОНОМИЧЕСКИЙ ПРИЕМ

Желательно (но вовсе не обязательно), чтобы дракон-схема целиком размещалась на одном листе бумаги. А если она слишком большая и вылезает за рамки прокрустово ложа?

Тогда можно взять бумагу побольше, 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 куски. Эти приемы позволяют разместить дракон-схему на листах желаемого размера. Рассмотрим проблему на «мини-атюрном» примере.

Предположим, линейный алгоритм состоит из 12 икон, а бумажный лист небольшой, так что на нем можно разместить по вертикали не более 8 икон. Как быть?

На рис. 69 и 70 показаны два варианта решения проблемы. Алгоритм на рис. 69 не годится, так как на участке *AB* бегунок движется вверх. Этот вариант придется отбросить, потому что нарушено

Правило. Движение вверх запрещено.

Преодолеть затруднение можно двумя способами:

- применить конструкцию «силуэт»;
- разделить алгоритм на части.

Первый способ мы уже знаем (рис. 68), поэтому рассмотрим второй. Для этого удалим из алгоритма несколько связанных по смыслу икон. А вместо них нарисуем икону-заместитель, которая называется *вставка* (рис. 70). 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* (рис. 70).

Обратите внимание: икона-вставка и процедура имеют одинаковое название «Собери рыбацкое снаряжение».

Мы использовали прием «Разрежь великана». В результате длинный исходный алгоритм (рис. 69) распался на два коротких (рис. 70).

Икона-вставка – это команда «Передай управление в процедуру» (рис. 71). 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».

§21. ЧТО ТАКОЕ ПОДСТАНОВКА?

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

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

Шаг 2. Вместо него подставляем икону вставка с именем *X*.

Шаг 3. К удаленному фрагменту добавляем икону заголовок с тем же именем *X* и икону конец. В результате получаем процедуру.

Два алгоритма на рис. 69 и 70 неравносильны, их формулы не совпадают. В самом деле, на рис. 69 мы видим 12 икон, а на рис. 70 – 15 икон (см. также рис. 71). 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). Таким образом, операция «подстановка» представляет собой эквивалентное (но не равносильное) преобразование алгоритмов.

§22. ЭПОХА ПОНЯТНЫХ АЛГОРИТМОВ

Мы рассмотрели ряд операций, позволяющих выполнить *эквивалентное* преобразование алгоритмов (при котором смысл алгоритма не меняется).

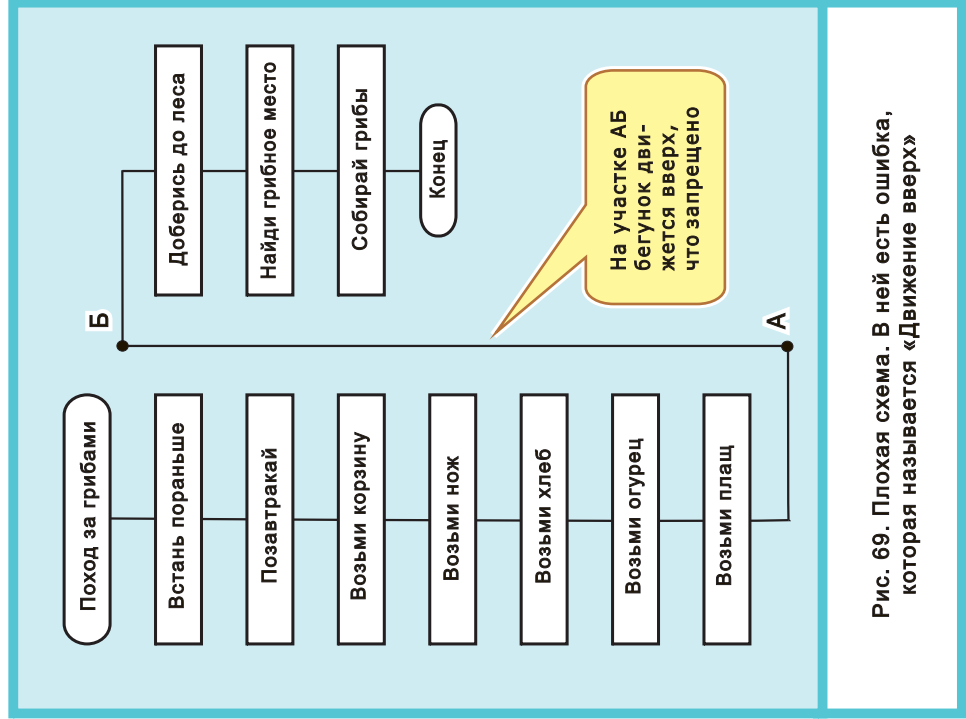


Рис. 69. Плохая схема. В ней есть ошибка, которая называется «Движение вверх»

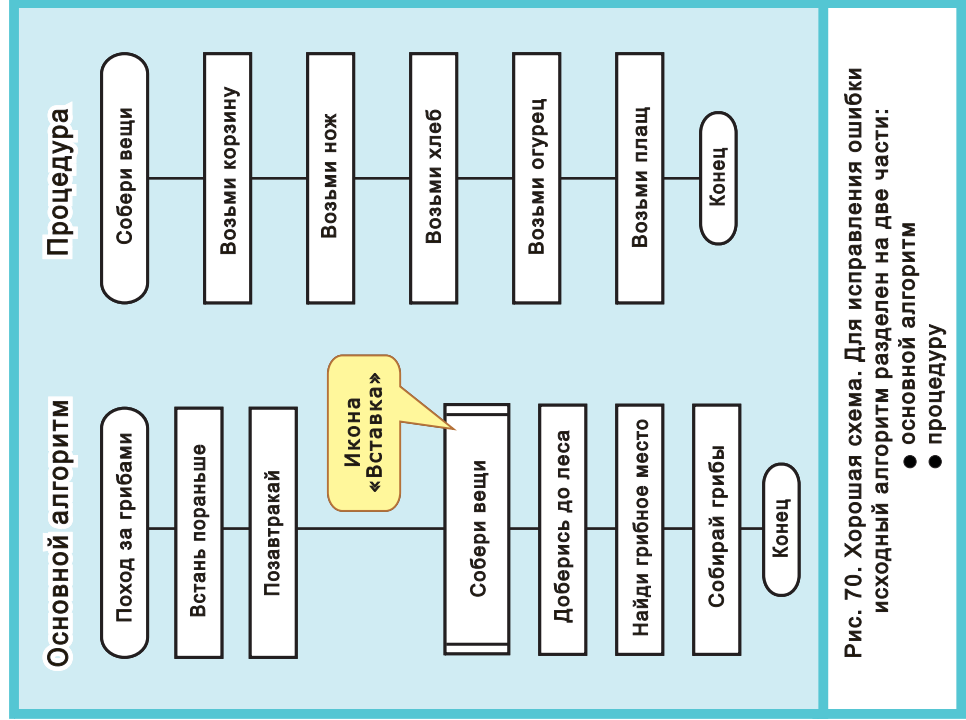


Рис. 70. Хорошая схема. Для исправления ошибки исходный алгоритм разделен на две части:
• основной алгоритм
• процедуру

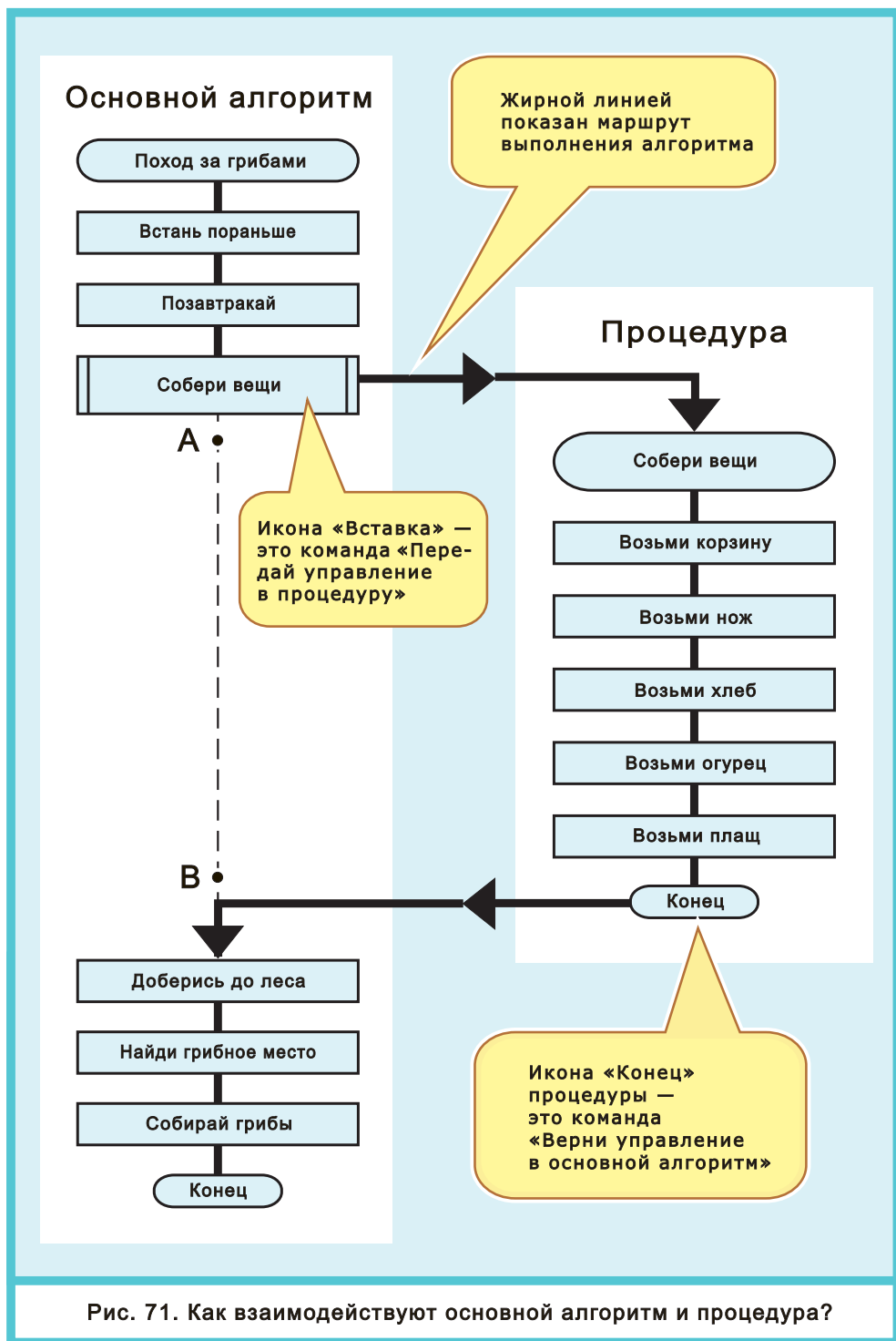


Рис. 71. Как взаимодействуют основной алгоритм и процедура?

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 понятность алгоритмов. Отсюда следует важный

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

Попытаемся заглянуть в будущее. Можно предположить, что 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 дальновидных специалистов.

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

§23. ВЫВОДЫ

1. *Развилка* – часть дракон-схемы, внутри которой маршрут сначала раздваивается, а затем соединяется в точке слияния.
2. У развилки два плеча, левое и правое.
3. *Маршрут* – это графический путь от начала до конца алгоритма, проходящий через иконы и соединительные линии.
4. Маршрут можно описать с помощью *формулы*, которая представляет собой последовательность букв, обозначающих иконы.
5. *Набор маршрутов* – это множество маршрутов данного алгоритма. Набор маршрутов и набор формул – одно и то же.
6. *Равносильные алгоритмы* – это алгоритмы, имеющие одинаковый набор маршрутов (одинаковый набор формул).
7. *Рокировка* – преобразование алгоритма, при котором левое и правое плечи развилки меняются местами.
8. Формальное преобразование алгоритма X в алгоритм Y называется равносильным, если алгоритмы X и Y равносильны.
9. Рокировка является равносильным преобразованием алгоритмов.

10. Если к визуальному алгоритму X применить операцию «рокировка», получим визуальный алгоритм Y , эквивалентный алгоритму X .
11. Рокировка позволяет улучшить наглядность и эргономичность алгоритмов.
12. *Вертикальное объединение* – операция, при которой несколько одинаковых икон объединяются в одну с помощью вертикальной линии.
13. Вертикальное объединение является равносильным преобразованием алгоритмов.
14. *Горизонтальное объединение* – операция, при которой несколько одинаковых икон объединяются в одну с помощью горизонтальной линии.
15. Горизонтальное объединение является равносильным преобразованием алгоритмов.
16. Если к визуальному алгоритму X применить операцию «вертикальное объединение» (или «горизонтальное объединение»), получим визуальный алгоритм Y , эквивалентный алгоритму X .
17. Вертикальное объединение и горизонтальное объединение позволяют улучшить наглядность и эргономичность алгоритмов.
18. Чтобы убрать неустранимое (без введения дополнительных переменных) пересечение линий в блок-схеме, надо превратить блок-схему в дракон-схему силуэт.
19. Описанные в данной главе математические методы позволяют значительно увеличить понятность алгоритмов для человеческого зрительного восприятия и человеческого мышления.

ПРОСТЫЕ ЦИКЛИЧНЫЕ АЛГОРИТМЫ

§1. КАКИЕ БЫВАЮТ АЛГОРИТМЫ?

Различают три типа алгоритмов: линейные, разветвленные и циклические.

Линейные алгоритмы – алгоритмы, в которых нет ни разветвлений, ни повторяющихся (циклических) участков. Примеры показаны на рис. 1, 5–8, 19–22, 39, 69, 70.

Разветвленные алгоритмы – алгоритмы, в которых есть разветвления, но нет повторяющихся (циклических) участков. Примеры представлены на рис. 4, 9, 11–15, 27, 29–33, 46–56, 59–62, 67.

Циклические алгоритмы (сокращенно *циклы*) – алгоритмы, в которых есть повторяющиеся (циклические) участки. Нам уже встречались такие алгоритмы на рис. 25, 33а–35. Пришла пора рассмотреть данную тему более детально.

§2. ОБЗОР ЦИКЛОВ ЯЗЫКА ДРАКОН

В языке ДРАКОН имеется следующий ассортимент циклов:

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

Первые четыре цикла рассмотрены в этой и следующей главе, цикл ЖДАТЬ – в главе 13.

§3. ОБЫЧНЫЙ ЦИКЛ

Термин «обычный цикл» обозначает три типа циклов:

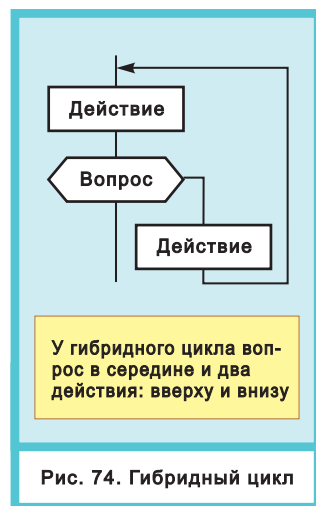
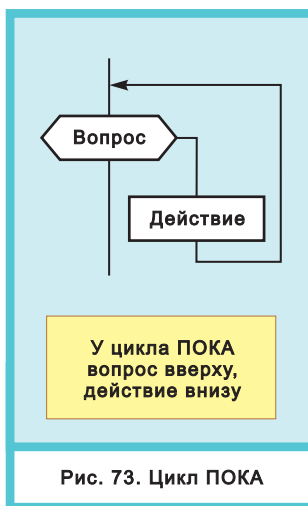
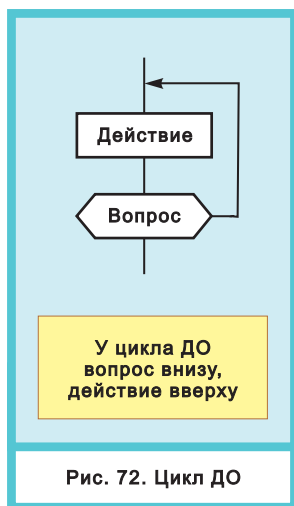
- цикл ДО (*do-while*),

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

Отличить их очень легко. У цикла ДО вопрос рисуют внизу, а действие вверх (рис. 72). У цикла ПОКА – все наоборот (рис. 73). Гибридный цикл – это «помесь» цикла ДО и цикла ПОКА (рис. 74).

Вопрос вверх – наверняка
Перед нами цикл ПОКА.
У цикла ДО, наоборот,
Вопрос внизу сидит, как крот.

Взглянем на рис. 18, макроикона 4. В этой макроиконе две точки ввода. Если ввести икону «действие» в верхнюю точку ввода, получим цикл ДО (рис. 72). Если – в правую, получим цикл ПОКА (рис. 73). Если заполнить обе точки, образуется гибридный цикл (рис. 74).



§4. ЦИКЛИЧНЫЕ АЛГОРИТМЫ

Циклические алгоритмы – вещь довольно сложная. Поэтому мы будем рассказывать о них 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 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.

§5. РАССКАЗ О ЗМЕЕ ГОРЫНЫЧЕ (ЦИКЛ «ДО»)

Предположим, у Змея Горыныча три головы. Чтобы победить Змея, Илья Муромец должен отсечь злодею все головы. То есть три раза исполнить команду «Отруби голову» (рис. 75). Если у Змея пять голов, эту команду придется повторить пять раз (рис. 76).

А если у Змея сто или тысяча голов? Тогда – если действовать в лоб – придется написать подряд сто или тысячу команд. Ясно, что это нелепый путь. Чтобы изложить алгоритм подобным образом, никакой бумаги не хватит!

Чтобы не писать слишком много одинаковых команд, нужно использовать цикл. *Цикл* – это повторяющееся исполнение одних и тех же команд. Фокус в том, что команда в алгоритме записывается *один* раз, а исполняется *много* раз – столько, сколько нужно.

Как работает цикл? Предположим, у Змея только одна голова, которую мы победили с помощью команды 1 (рис. 77). В этом случае на вопрос 2: «У Змея Горыныча остались живые головы?» – отвечаем «нет», и алгоритм заканчивается.

Если же у Змея есть уцелевшие головы, отвечаем «да» и, проходя через стрелку, снова попадаем на вход иконы 1. После чего рубим следующую голову. Еще раз задаем вопрос 2. Если у Змея по-прежнему остались головы, отвечаем «да» и вновь возвращаемся на вход иконы 1.

Таким образом, исполнение команд 1 и 2 может продолжаться и сто, и двести, и тысячу раз – до тех пор, пока наш алгоритм (управляющий «роботом» Ильей Муромцем) не отсечет у Змея все головы до последней – см. также рис. 78.

Циклический
алгоритм

Это алгоритм, в котором команды записываются один раз, а выполняются многократно

§6. ЦИКЛ СО СДОБНЫМИ ПЛЮШКАМИ (ЦИКЛ «ПОКА»)

Однажды Карлсон, который живет на крыше, нашел кошелек и открыл его (рис. 79). 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 горю плюшек и наелся до отвала.

Таким образом, цикл на рис. 79 может:

- ни разу не выполняться;

Змей Горыныч и циклические алгоритмы

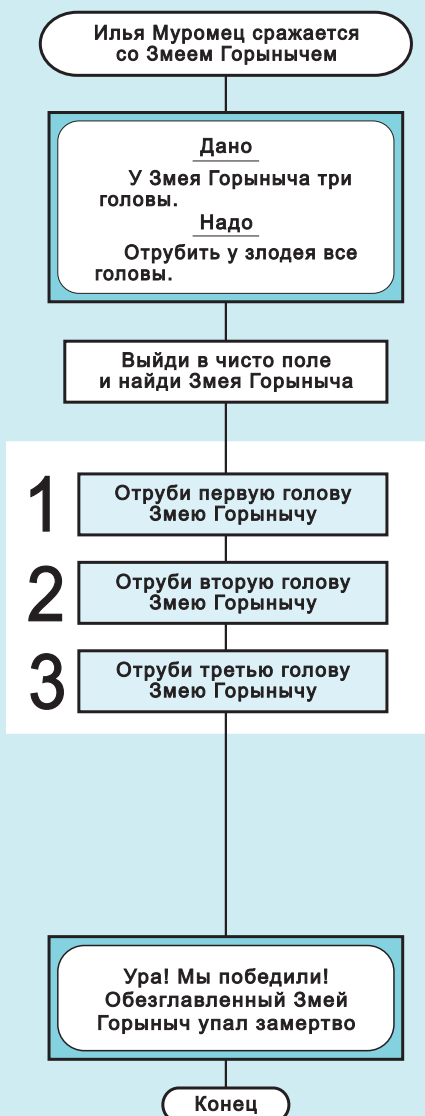


Рис. 75. Плохой (очень длинный) алгоритм. Чтобы отрубить три головы, приходится писать три команды «Отруби голову»

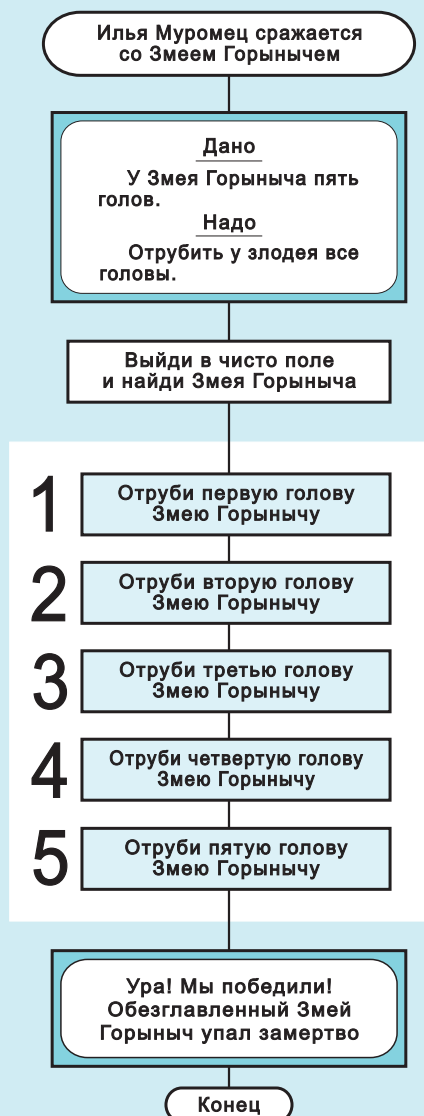


Рис. 76. Плохой (очень длинный) алгоритм. Чтобы отрубить пять голов, надо написать пять команд «Отруби голову». А если у Змея сто голов?

Змей Горыныч и циклические алгоритмы



Это и есть ЦИКЛ,
которого боится
Змей Горыныч

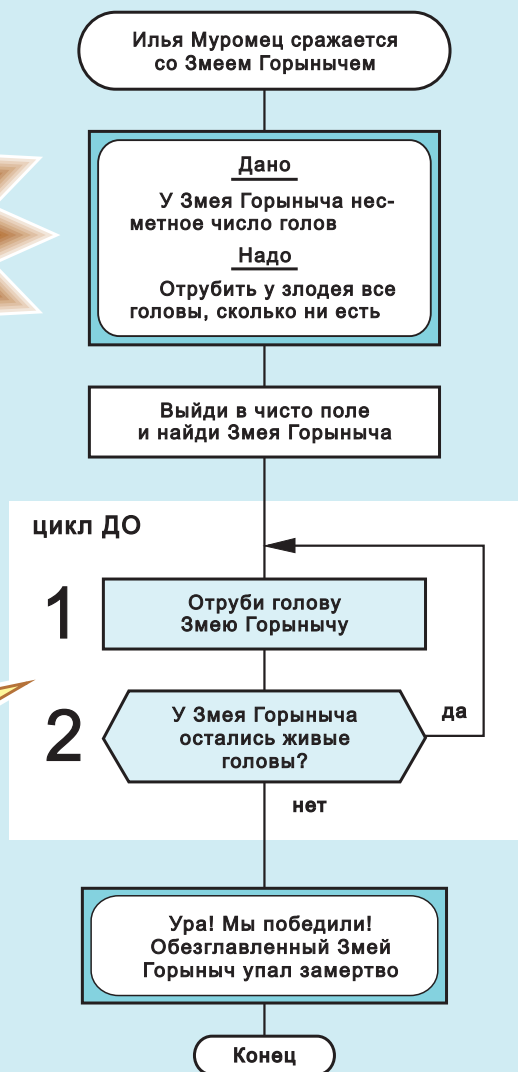


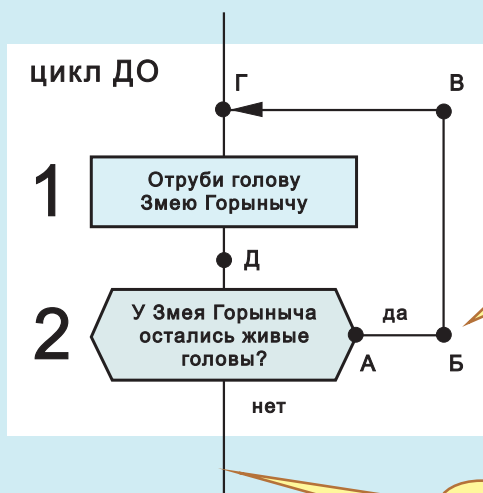
Рис. 77. Хороший (короткий) алгоритм.

А почему он короткий? Потому что в нем есть ЦИКЛ.

Циклический алгоритм позволяет отрубить любое число голов.

При этом нужна всего одна команда «Отруби голову».

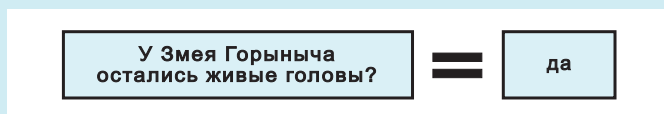
Змей Горыныч и циклические алгоритмы



Если у Змея есть живые головы, бегунок выезжает из иконы 2 направо через «да». Затем он едет по круговой дороге АБВГДА. Он может кружить по ней много раз.

Когда все головы упадут на землю, бегунок поедет вниз через «нет»

УСЛОВИЕ ПРОДОЛЖЕНИЯ ЦИКЛА



УСЛОВИЕ ОКОНЧАНИЯ ЦИКЛА

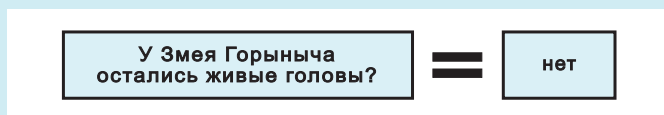


Рис. 78. Цикл — это замечательно!
Команды 1 и 2 повторяются много раз,
если выполняется УСЛОВИЕ ПРОДОЛЖЕНИЯ ЦИКЛА

- выполняться один раз;
- выполняться много раз (два и более).

Такой цикл имеет специальное название – «цикл ПОКА».

Рассмотрим подробнее. Цикл на рис. 79 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.

§7. ОСОБЕННОСТЬ ЦИКЛА «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. 77 сначала выполняется действие «Отруби голову Змею Горынычу». И только после этого возникает вопрос: «У Змея Горыныча остались живые головы?».

Такая же картина на рис. 80. В начале цикла выполняются два действия:

- Покрась одну доску.
- Шагни вправо на ширину доски.

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

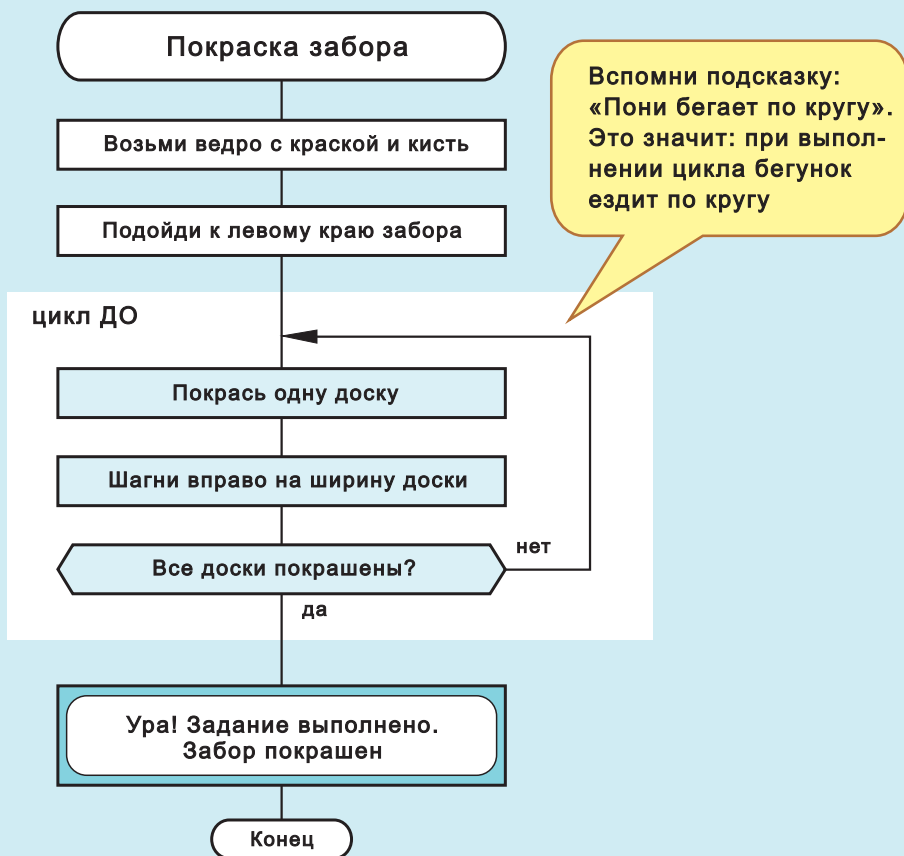
Когда все доски будут покрашены, бегунок выходит из иконы «вопрос» через «да». И алгоритм заканчивает работу.

§8. СРАВНЕНИЕ ЦИКЛОВ «ПОКА» И «ДО»

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» (рис. 79).

Рассмотрим противоположный пример, связанный с циклом ДО (рис. 77). Невозможно представить, чтобы на битву с Ильей Муромцем изначально прибыл Змей-инвалид, у которого нет ни одной головы. Поэтому можно быть уверенным, что до начала сражения, по крайней





УСЛОВИЕ ПРОДОЛЖЕНИЯ ЦИКЛА

Все доски покрашены?

=

нет

УСЛОВИЕ ОКОНЧАНИЯ ЦИКЛА

Все доски покрашены?

=

да

Рис. 80. Пример цикла ДО

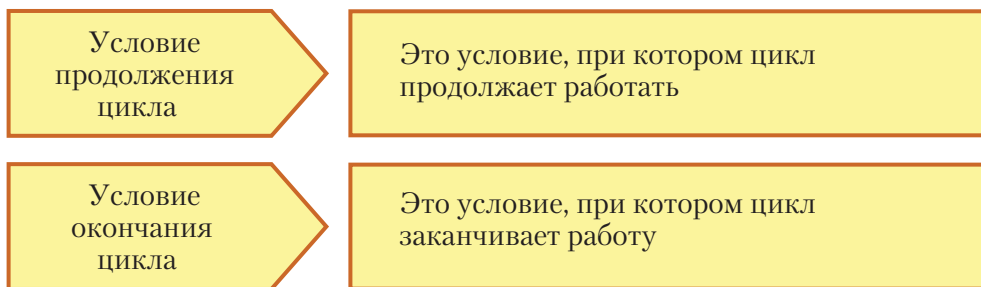
мере, одна голова у Змея непременно есть. Следовательно, Илья Муромец обязательно отсечет Змею хотя бы одну голову. Этот пример подтверждает, что в цикле ДО действие выполняется, как минимум, один раз.

В цикле ДО действие обязательно выполняется хотя бы один раз. А в цикле ПОКА при некоторых условиях действие может ни разу не выполняться.

§9. УСЛОВИЕ ПРОДОЛЖЕНИЯ И ОКОНЧАНИЯ ЦИКЛА

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

На рис. 79 (внизу) показаны два важных условия.



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.

Пример 1. На рис. 79 условие продолжения цикла имеет вид:

В кошельке есть денежки? = да

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

В кошельке есть денежки? = нет

Пример 2. На рис. 80 условие продолжения цикла имеет вид:

Все доски покрашены? = нет

Когда все доски будут покрашены, логическая переменная изменит свое значение с «нет» на «да». В этот момент условие продолжения цикла исчезнет. И появится условие окончания цикла:

Все доски покрашены? = да

Сравнивая примеры 1 и 2 (рис. 79 и 80), можно сформулировать

Правило. В правой части условия (подчиняясь логике задачи) алгоритмист по своему усмотрению может ставить как «да», так и «нет».

§10. ВЫВОДЫ

1. Различают три типа алгоритмов: линейные, разветвленные и циклические.
2. В языке ДРАКОН имеются следующие циклы:
 - обычный цикл;
 - веточный цикл;
 - цикл ДЛЯ;
 - переключающий цикл;
 - цикл ЖДАТЬ.
3. Термин «обычный цикл» обозначает три типа циклов:
 - цикл ДО;
 - цикл ПОКА;
 - гибридный цикл.
4. Существующие циклы, используемые во всем мире, имеют недостаток. Они накладывают на творческую мысль алгоритмиста ограничения. Язык ДРАКОН позволяет снять многие из этих ограничений. В результате алгоритмическая мысль становится более естественной и плодотворной.
5. *Условие продолжения цикла* – условие, при котором цикл продолжает работать.
6. *Условие окончания цикла* – условие, при котором цикл заканчивает работу.
7. У обычного цикла условие продолжения цикла соответствует правому выходу иконы «вопрос».
8. У обычного цикла условие окончания цикла соответствует нижнему выходу иконы «вопрос».
9. Условие окончания цикла может помечаться как словом «нет», так и словом «да». То же самое относится и к условию продолжения цикла.

ОСОБЕННОСТИ ЦИКЛИЧНЫХ АЛГОРИТМОВ

§1. ВВЕДЕНИЕ

В этой главе освещаются следующие вопросы:

- досрочный выход из цикла;
- веточный цикл;
- цикл ДЛЯ;
- переключающий цикл.

§2. ВХОДЫ И ВЫХОДЫ ЦИКЛИЧНОГО АЛГОРИТМА

Цикл (циклический алгоритм) имеет только один вход. А выходов может быть несколько. Один выход цикла называется основным, остальные – досрочными.

Начнем с изучения досрочных выходов.

§3. 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. 79?

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. 79 нельзя выйти раньше времени. Вспомним – в кошельке пятьсот монет. Значит каждая команда цикла

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

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. 81).

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. 81 имеет не один, а два выхода: *основной* и *досрочный*. Через первый мы выходим, когда в кошельке кончились деньги. Через второй – когда Карлсон наелся.

§4. ДОСРОЧНЫЙ ВЫХОД ИЗ ЦИКЛА «ДО»

Папино слово – закон! А папа сказал: сегодня нужно покрасить забор. На рис. 80 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. 82. Алгоритм на рис. 82 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. 83.

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. 84. 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.

Повторенье – мать ученья. Давайте еще раз повторим

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

§5. ОСОБЕННОСТИ ОБЫЧНОГО ЦИКЛА

Рассмотрим циклы ДО и ПОКА в общем виде.

Анализируя рисунки 72–84, можно сделать следующие замечания.

- Визуальный 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).

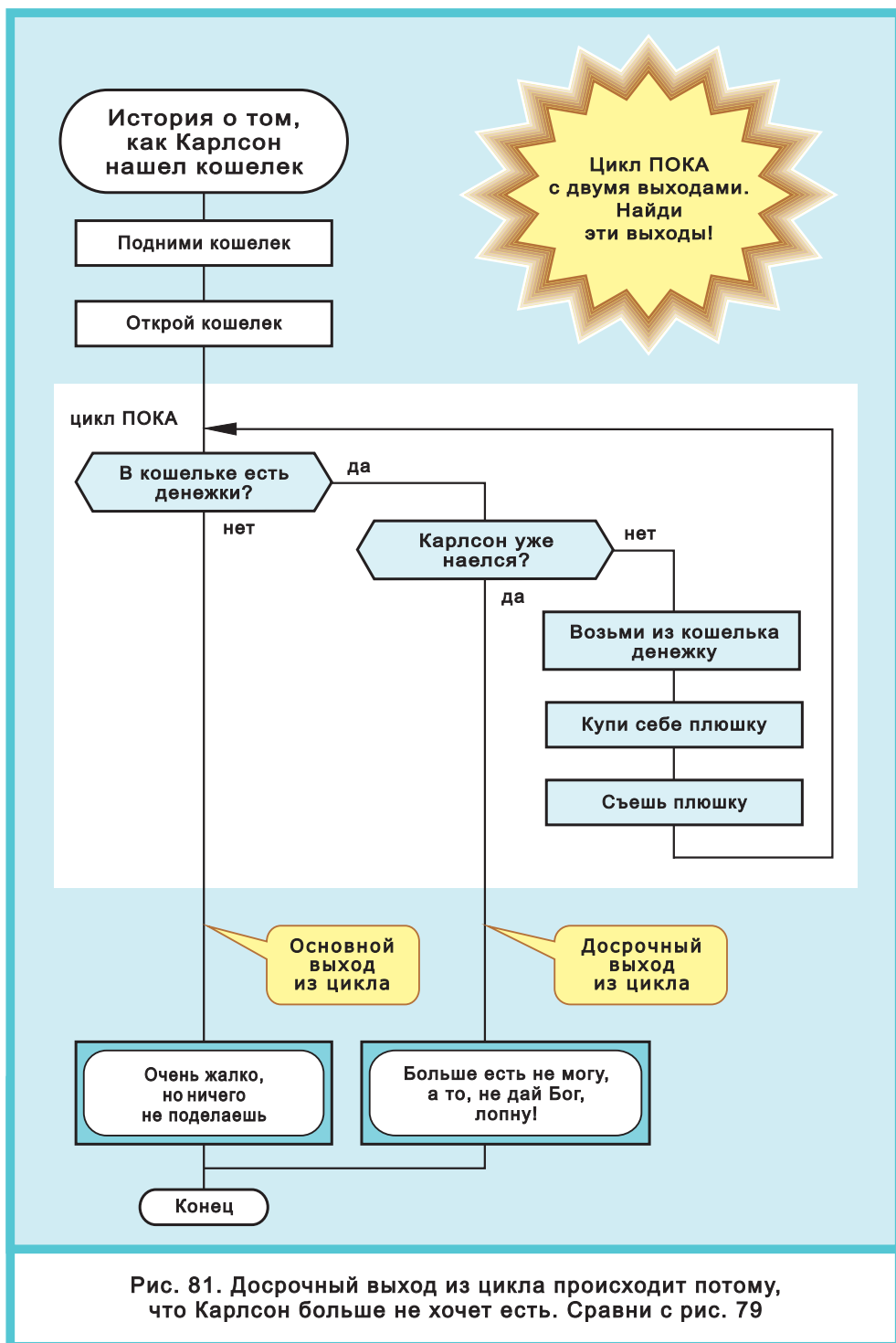


Рис. 81. Досрочный выход из цикла происходит потому, что Карлсон больше не хочет есть. Сравни с рис. 79

Досрочный выход из цикла

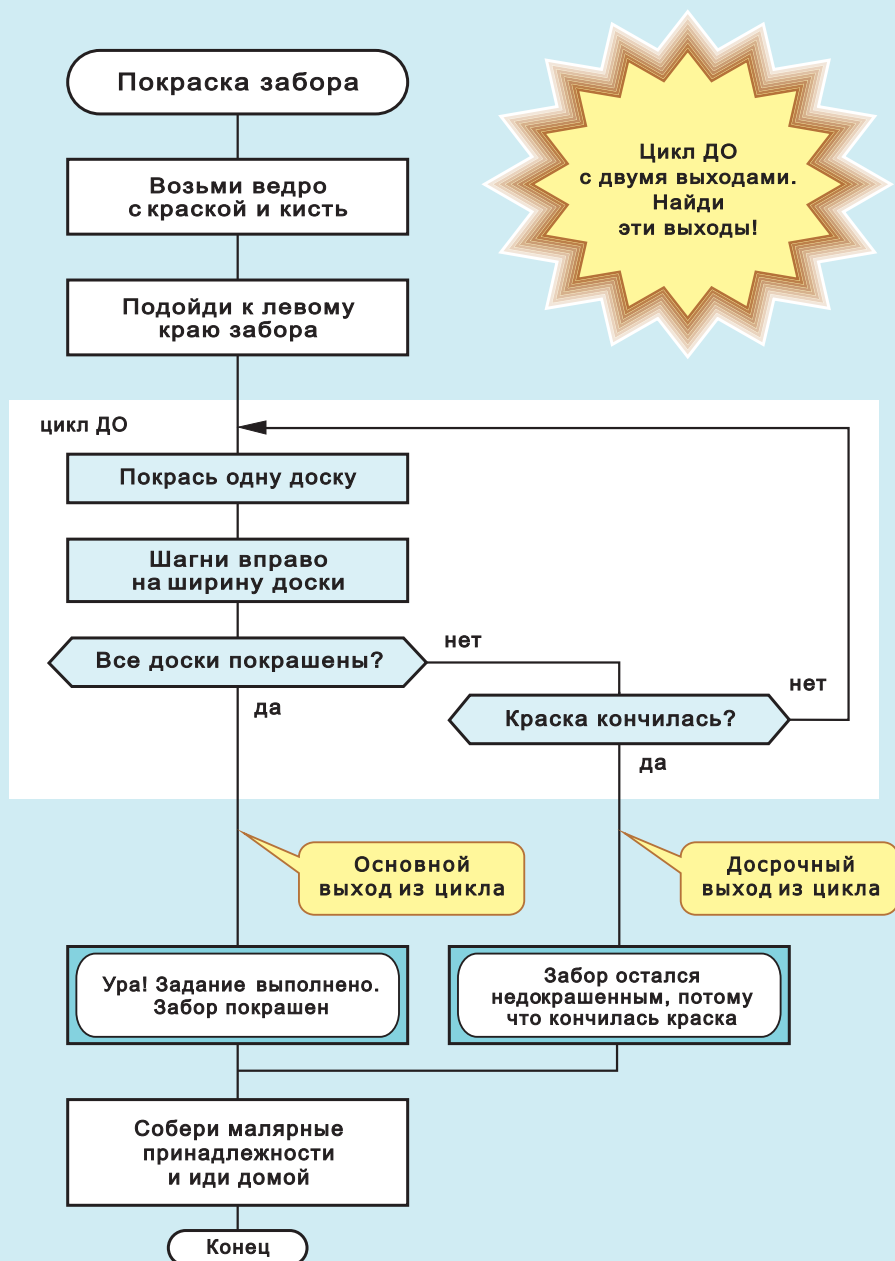


Рис. 82. Досрочный выход из цикла, потому что кончилась краска

Досрочный выход из цикла

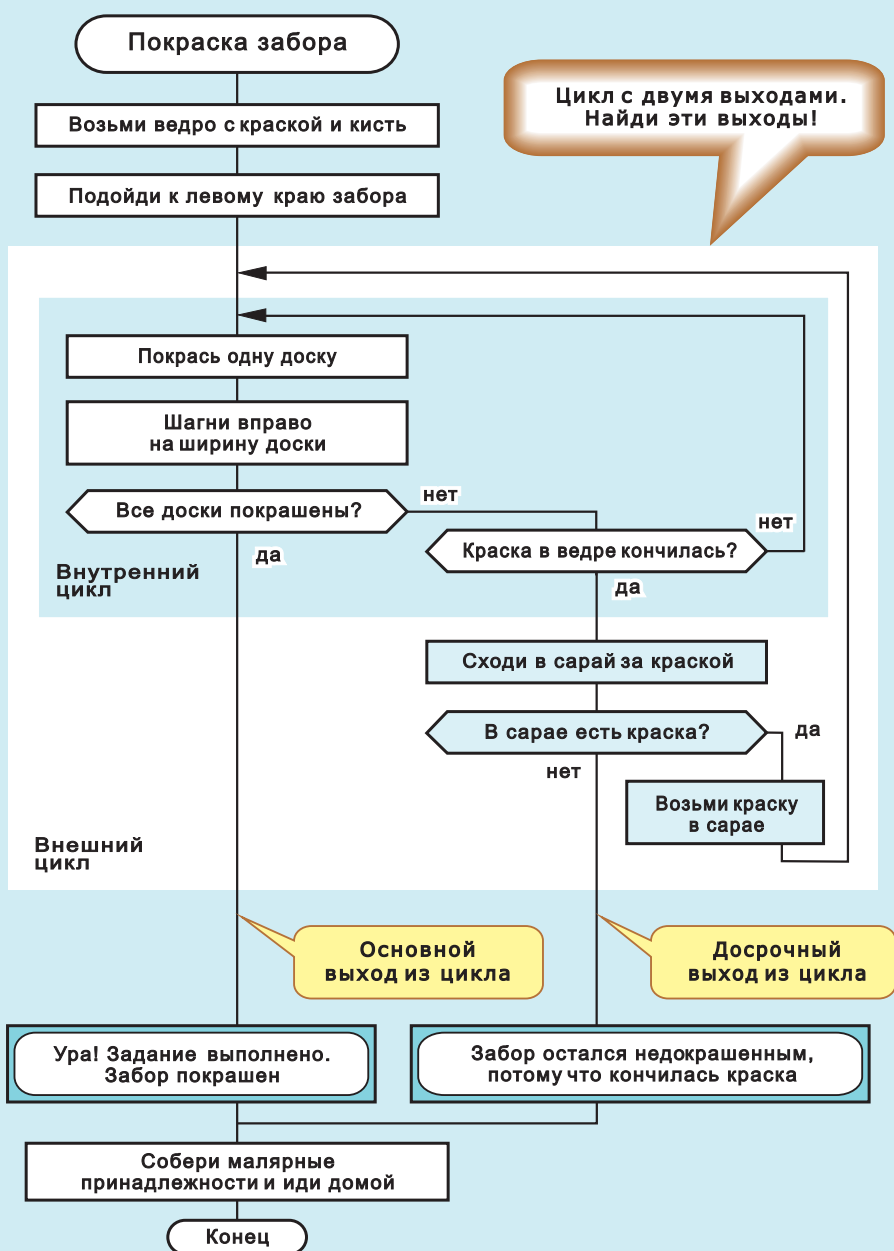


Рис. 83. Досрочный выход из цикла, потому что краска в ведре кончилась. И в сарае краски тоже нет

Досрочный выход из цикла

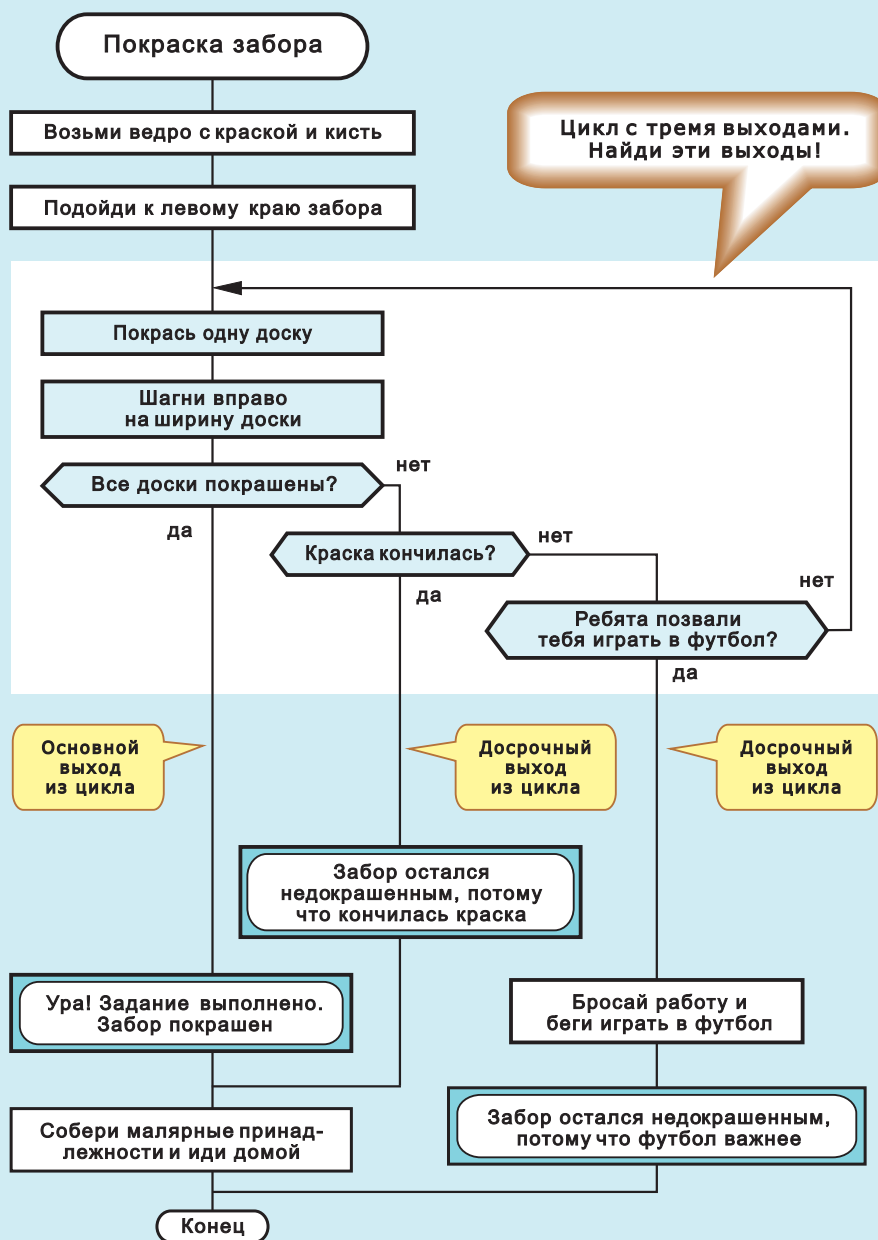


Рис. 84. Два досрочных выхода: (1) потому, что краска кончилась, (2) потому что ребята позвали играть в футбол

- Если цикл имеет более одного выхода, основной выход, как правило, размещается на главной вертикали. Дополнительные (досрочные) – справа от нее.
- Шампур цикла проходит через икону «вопрос».
- Тело цикла ПОКА находится справа от шампура.
- Тело цикла ДО находится на шампуре.
- 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. 79).

§6. 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 или нескольких веток
- Чтобы построить веточный цикл, нужно написать в иконе «адрес» название данной ветки (или более левой ветки)

На рис. 80 изображен циклический алгоритм «Покраска забора». Можно ли нарисовать его в виде силуэта? Да, можно. Результат показан на рис. 85.

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. 85).

Другой пример веточного цикла показан на рис. 86.

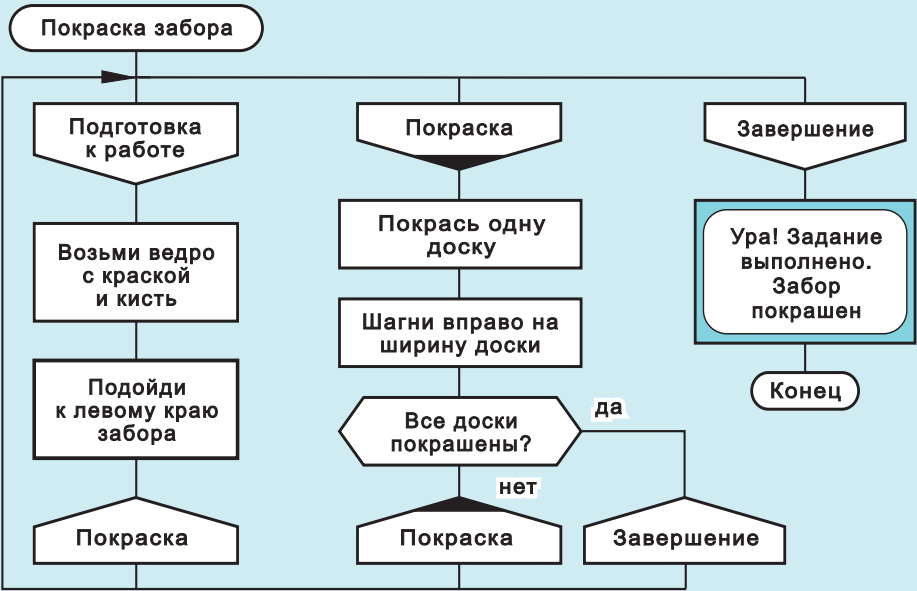


Рис. 85. Силуэт с веточным циклом. Сравни с рис. 80

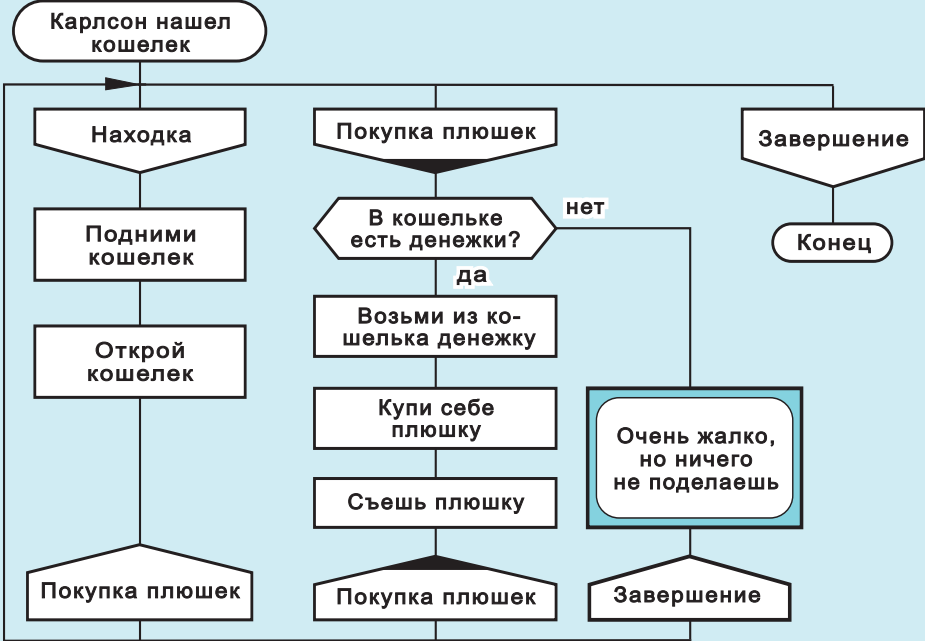


Рис. 86. Силуэт с веточным циклом. Сравни с рис. 79

§7. СЛОЖНЫЕ ВЕТОЧНЫЕ ЦИКЛЫ

Как и любой цикл, веточный цикл может иметь основной и досрочный выходы (рис. 87 и 88).

Веточный цикл можно использовать в сочетании с циклами ДО и ПОКА. Например, на рис. 89 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. 85–89)?

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

§8. ЦИКЛ ДЛЯ

На рис. 90 и 91 показаны два варианта решения простой математической задачи. В первом случае используется цикл ДО, во втором – цикл ДЛЯ.

Цикл ДЛЯ – составной графический оператор (рис. 18, макроикона 6). Он содержит иконы «начало цикла ДЛЯ» и «конец цикла ДЛЯ» (рис. 17, иконы И12 и И13), между которыми располагаются одна или несколько других икон.

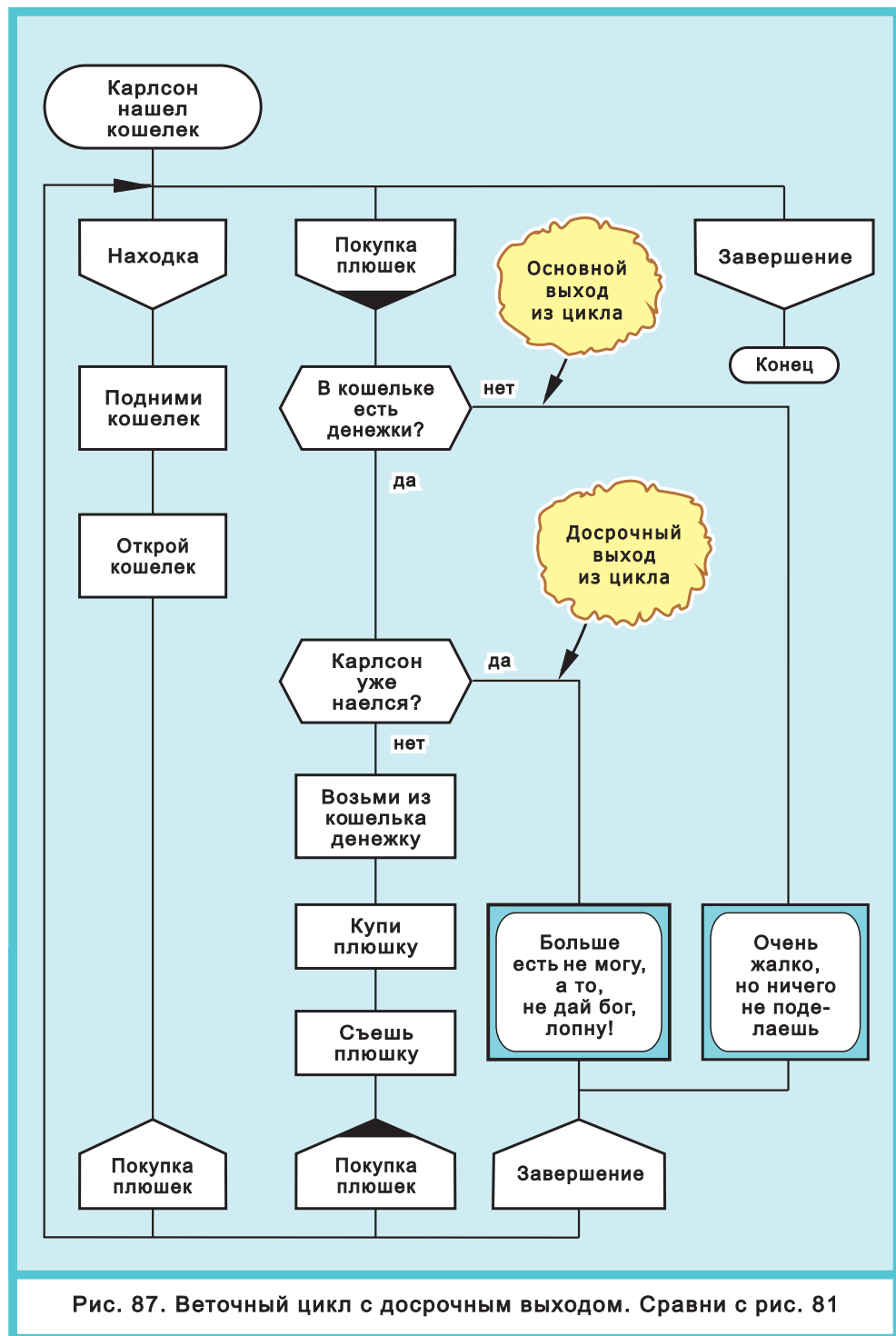
Внутри иконы «начало цикла ДЛЯ» указываются переменная цикла, ее начальное и конечное значения и шаг. Правила записи этих величин определяется выбранным вариантом текстового синтаксиса. На рис. 91 изображен вариант, по умолчанию принимающий, что шаг равен 1.

§9. ПЕРЕКЛЮЧАЮЩИЙ ЦИКЛ

Переключатель позволяет создать особый тип цикла – *переключающий цикл* (рис. 18, макроикона 5). Для этого нужно мысленно оторвать выход правой ветви переключателя, загнуть его вверх и присоединить стрелку в нужное место (рис. 92).

На рис. 93 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.



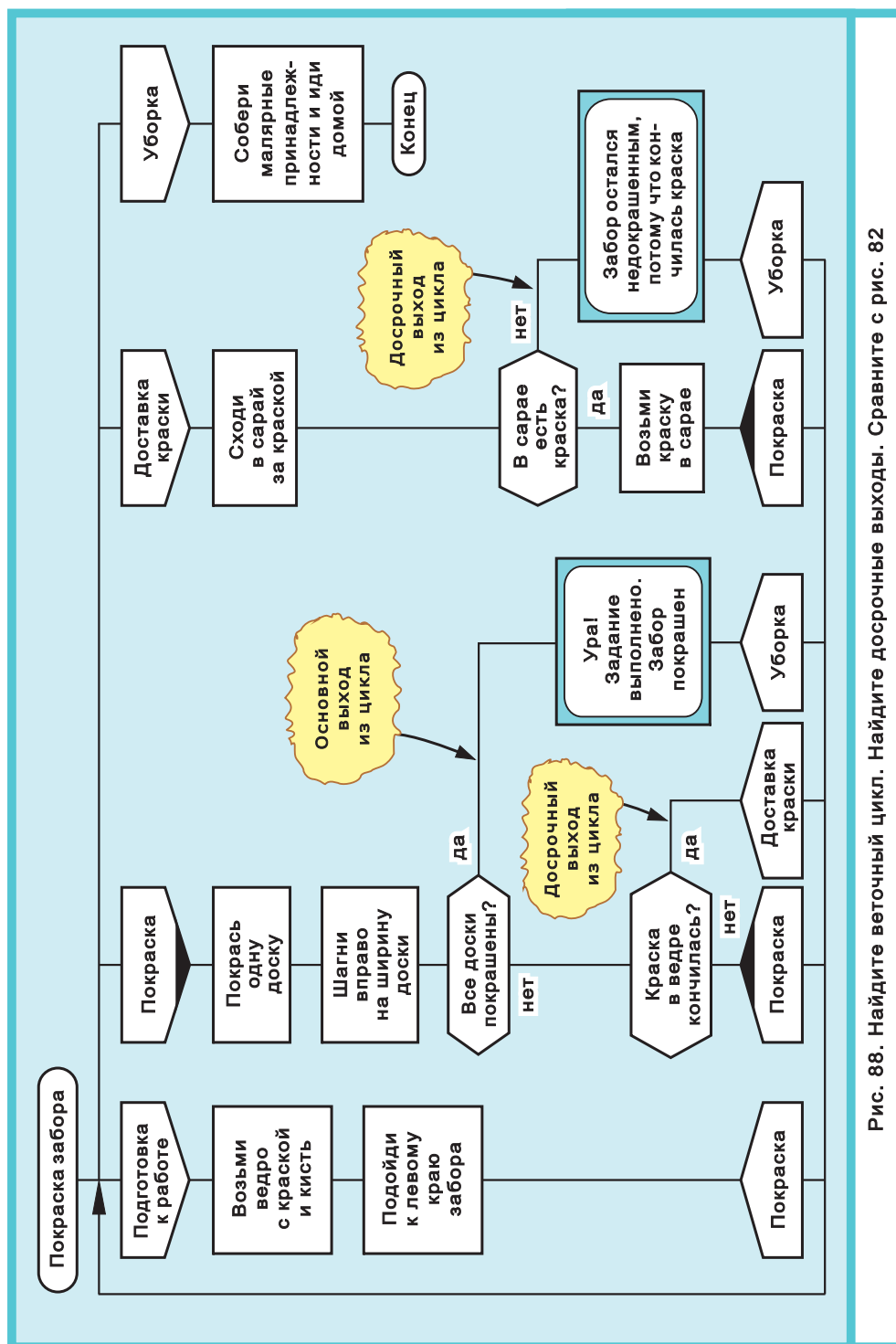
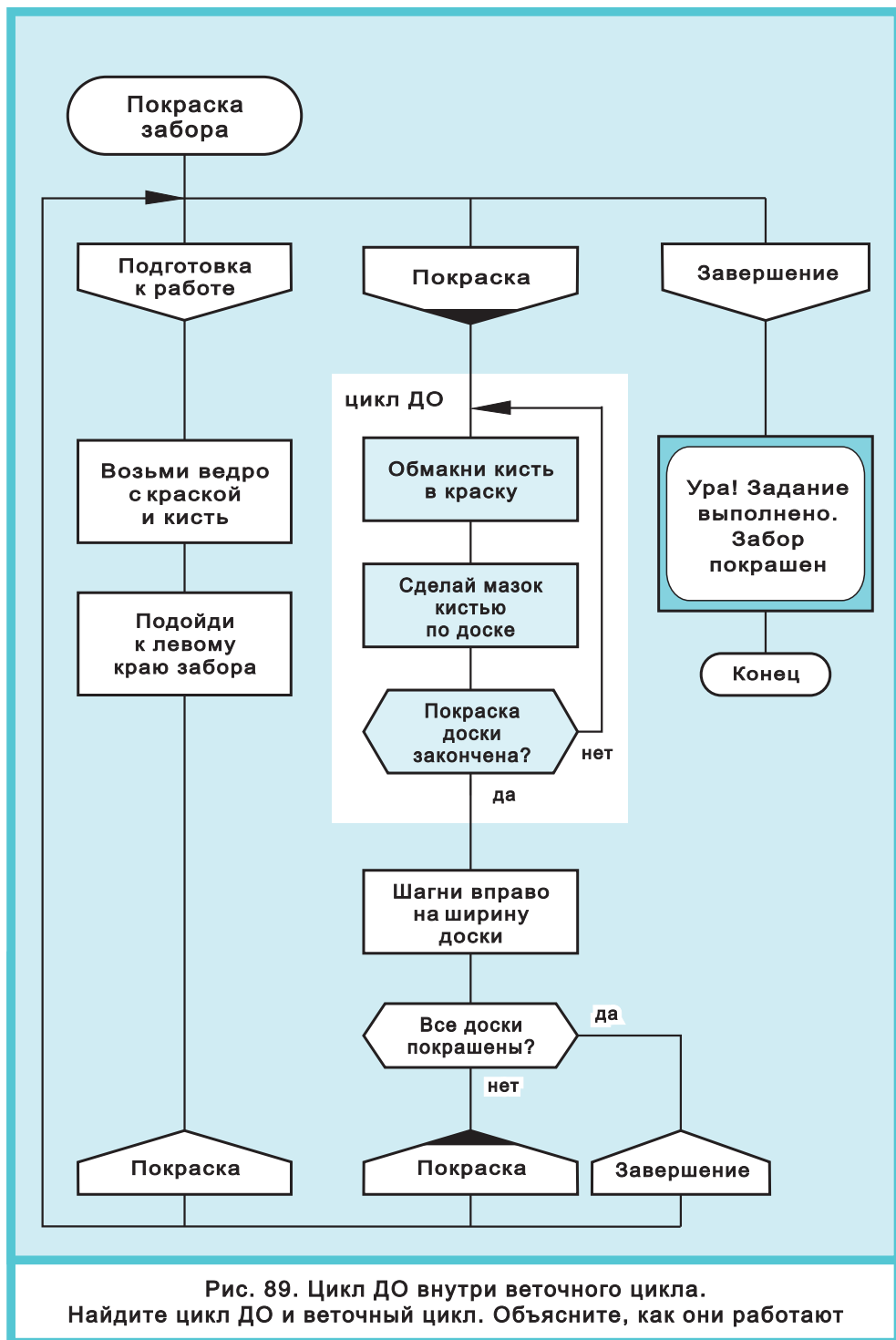
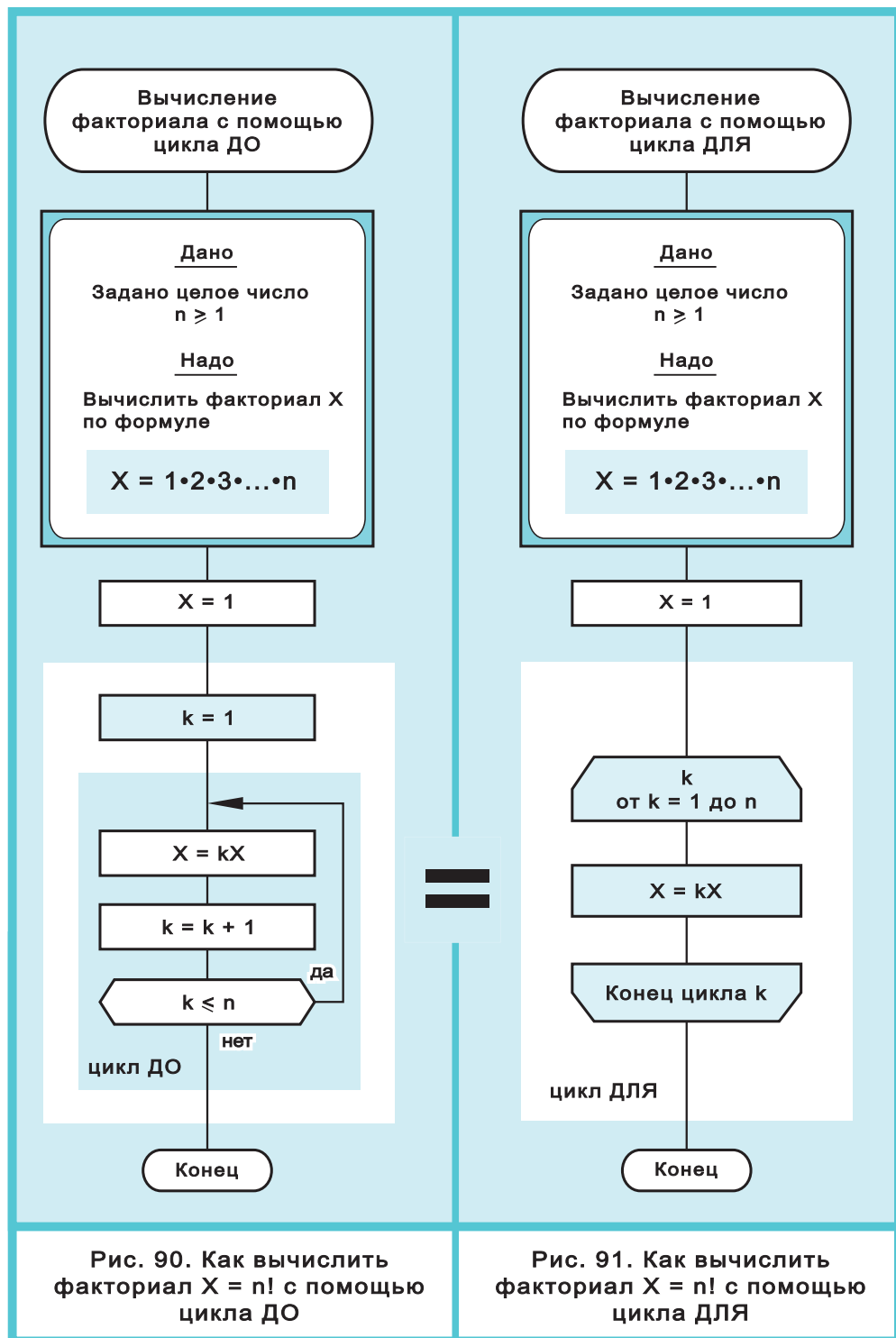
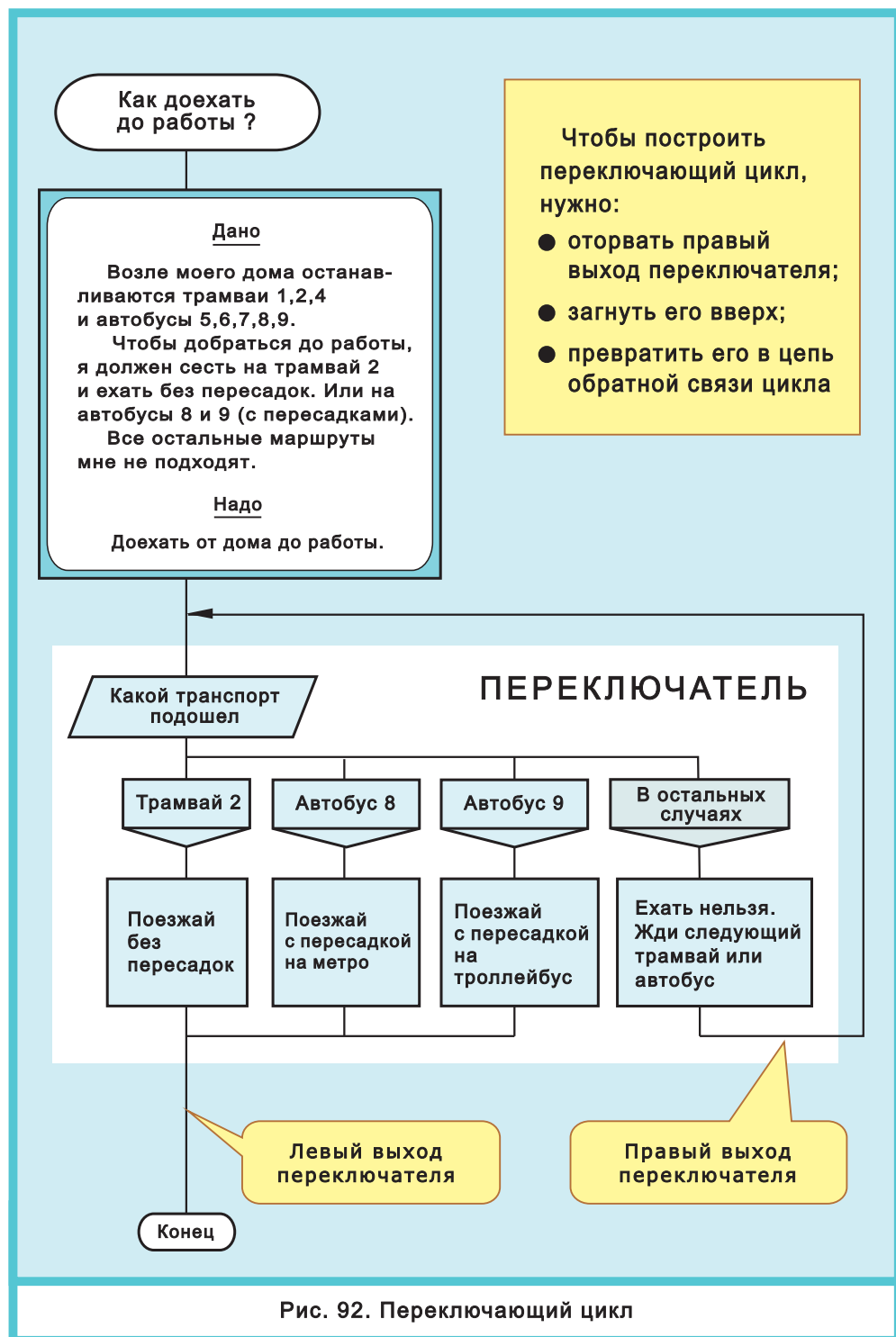


Рис. 88. Найдите веточный цикл. Найдите досрочные выходы. Сравните с рис. 82







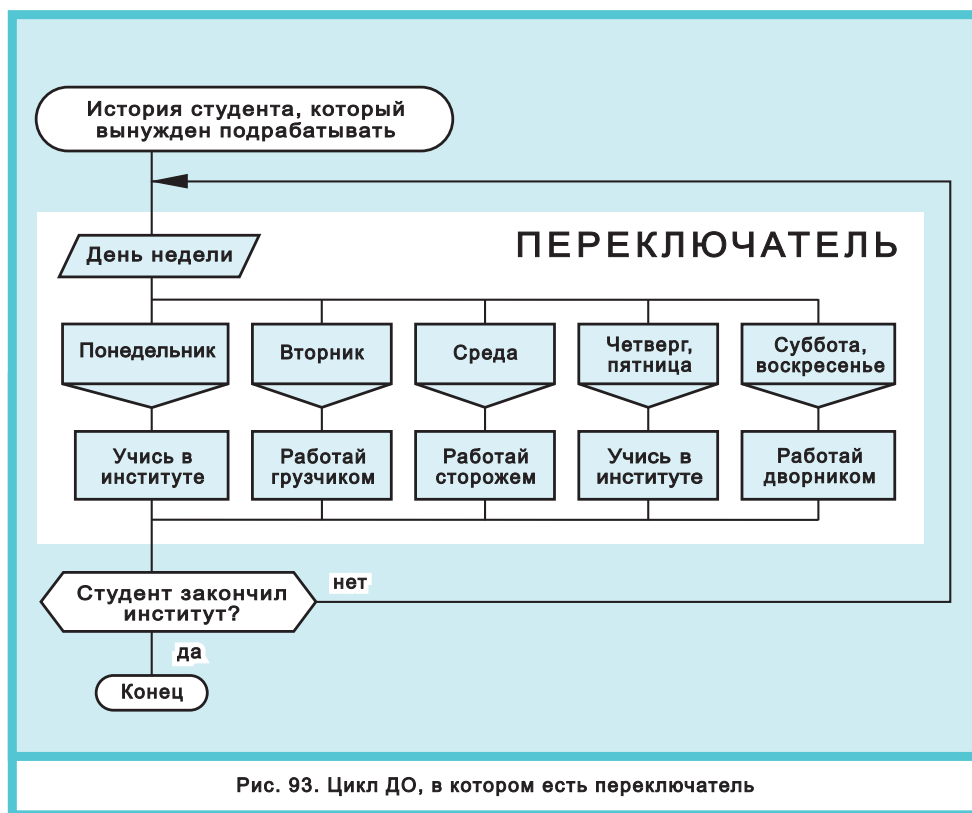


Рис. 93. Цикл ДО, в котором есть переключатель

§10. КОГДА В ИКОНЕ «ИМЯ ВЕТКИ» ПИШУТ СЛОВО «ЗАВЕРШЕНИЕ» ИЛИ «ВЫХОД»?

Рассмотрим случай, когда в последней ветке силуэта находятся только две иконы: «имя ветки» и «конец». В этом случае в иконе «имя ветки» пишут слово Завершение (рис. 86, 87). Или Выход.

Если в последней ветке (кроме икон «имя ветки» и «конец») нарисованы одна или несколько икон «комментарий», в иконе «имя ветки» также пишут Завершение (рис. 85, 89). Или Выход.

Правило. При описанных условиях запрещается заменять слово Завершение (или Выход) каким-либо другим словом.

Если же в последней ветке (кроме икон имя ветки и конец) находится икона, отличная от комментария, вместо слова Завершение (или Выход) пишут слово, обозначающее содержание ветки. Например, на рис. 88 использовано слово Уборка.

§11. ВЫВОДЫ

1. Обычный цикл имеет один вход. И один или несколько выходов.
2. Первый выход цикла называется основным, остальные – досрочными.
3. Цикл с одним выходом представляет собой шампур-блок (вход и выход находятся на одной вертикали).
4. Если цикл имеет более одного выхода, основной выход, как правило, размещается на главной вертикали. Дополнительные (досрочные) – справа от нее.
5. Шампур цикла проходит через икону «вопрос».
6. Тело цикла ПОКА находится справа от шампура.
7. Тело цикла ДО находится на шампуре.
8. Петля цикла находится правее главной вертикали и закручена против часовой стрелки.
9. Икона «вопрос» задает *условие цикла*, которое распадается на две части: условие продолжения цикла и условие окончания цикла
10. *Веточный цикл* – это повторное исполнение одной или нескольких веток.
11. Чтобы построить веточный цикл, нужно написать в иконе «адрес» название данной ветки (или более левой ветки).
12. Веточный цикл используется только в силуэте.
13. Как и любой цикл, веточный цикл может иметь основной и досрочный выходы.
14. В веточном цикле в иконах «имя ветки» и «адрес» используются маленькие черные треугольники (флажки).
15. Флажки привлекают внимание и позволяют легко заметить веточный цикл даже при беглом взгляде.
16. Если в силуэте есть два веточных цикла, то, чтобы их различить, используют два разных маркера (см. ниже рис. 183 и 184).
17. Если в последней ветке силуэта находятся только иконы: «имя ветки», «конец» и, возможно, «комментарий», в иконе «имя ветки» пишут слово Завершение. Или Выход.

СЛОЖНЫЕ ЦИКЛИЧНЫЕ АЛГОРИТМЫ. СТРУКТУРА «ЦИКЛ В ЦИКЛЕ»

§1. ВВЕДЕНИЕ

Русская матрешка – игрушка с секретом. Откроешь одну – в ней прячется другая. Откроешь вторую – а в ней еще одна.

Между прочим, цикл тоже можно спрятать внутри другого цикла. Получается структура «цикл в цикле», очень похожая на матрешку в матрешке.

§2. ЦИКЛ «ДО» ВНУТРИ ЦИКЛА «ДО»

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

На рис. 80 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. 94). *(Рис. 94 находится слева от рис. 95).*

Что же мы сделали? Команду «Покрась одну доску» мы превратили в циклический алгоритм на рис. 94. Передвинем этот цикл вправо и вставим его в рис. 95. Мы получили конструкцию «цикл в цикле».

Этот алгоритм работает так. Предположим, забор красит Том Сойер. Сначала Том выполняет две команды (рис. 95):

- 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. 95).

§3. ЦИКЛ «ДО» ВНУТРИ ЦИКЛА «ПОКА»

В алгоритме на рис. 79 заменим икону «Съешь плюшку» на цикл ДО, состоящий из икон «Откуси кусок плюшки» и «Плюшка съедена?» (рис. 96). 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. 96).

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 $50 \times 3 = 150$ «откусываний»).

§4. ЦИКЛ «ПОКА» ВНУТРИ ЦИКЛА «ДО»

Рассмотрим алгоритм на рис. 97. Предположим, старый пасечник Микола решил собрать мед во всех ульях своей пасеки. Сначала Микола выполняет две команды (рис. 97):

Задание

1. Открой рис. 80.
2. Найди команду «Покрась одну доску».
3. Вместо этой команды подставь цикл ДО.
4. В результате рис. 94 (см. внизу) превратится в рис. 95 (см. справа).



Рис. 94. Цикл «Покраска доски»

Задание

1. Найди цикл «Покраска доски» на рис. 94 и 95.
2. Убедись, что цикл ДО (2) находится внутри цикла ДО (1).

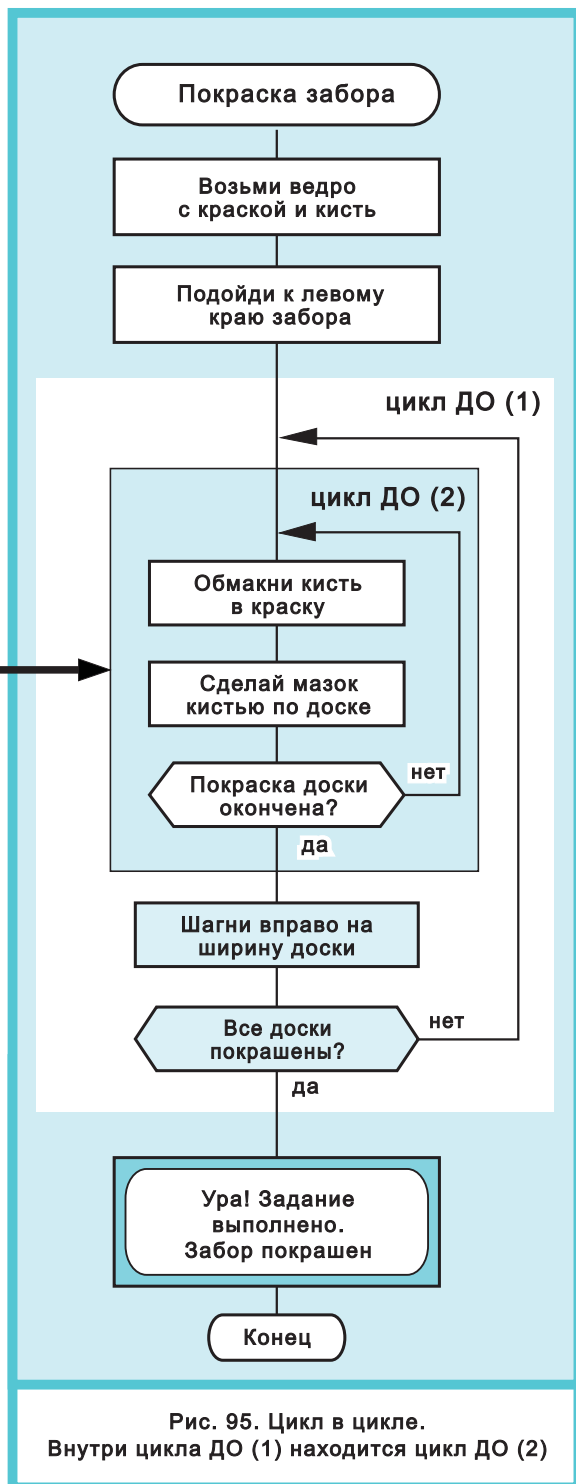


Рис. 95. Цикл в цикле.
Внутри цикла ДО (1) находится цикл ДО (2)

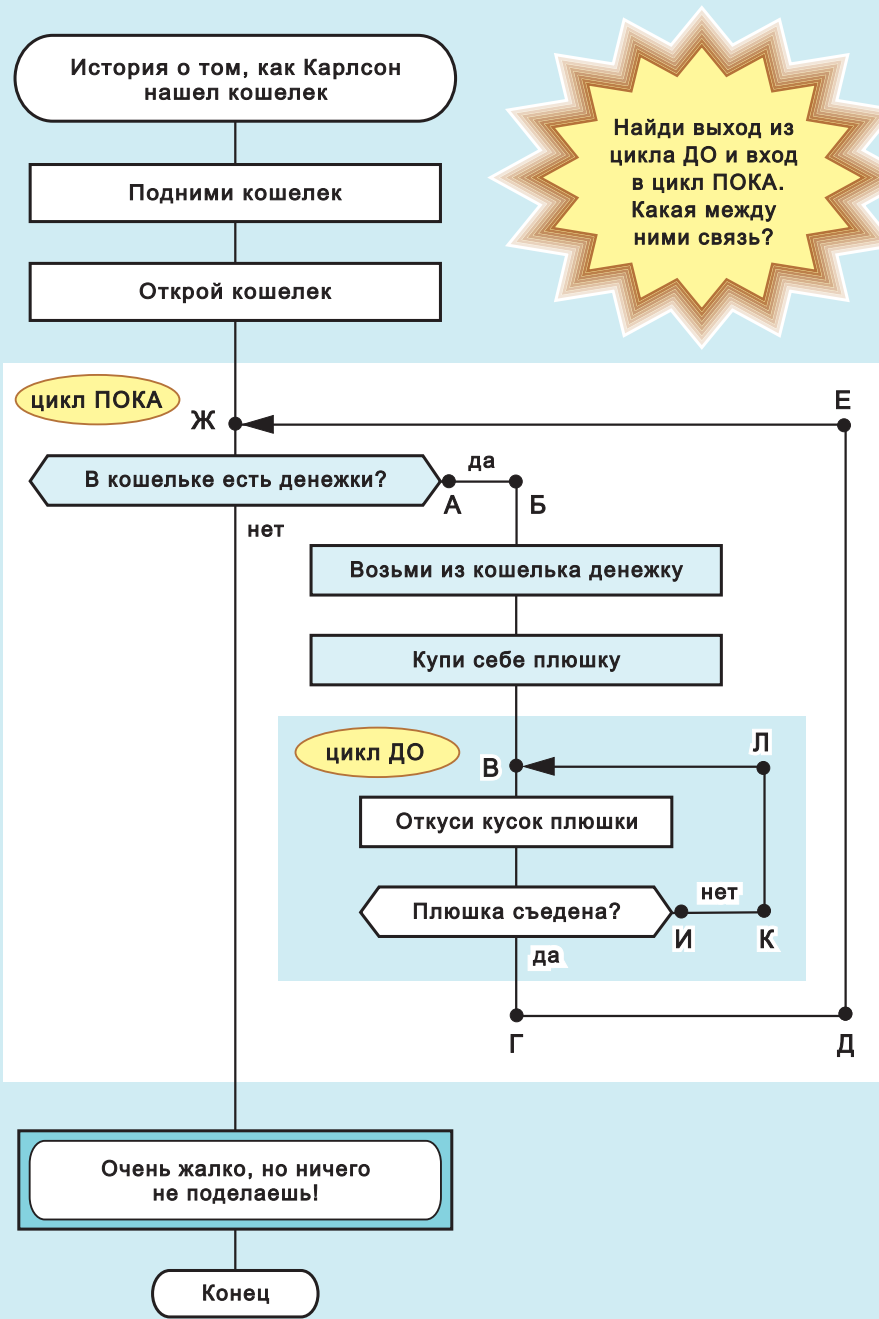


Рис. 96. Цикл в цикле. Внутри цикла ПОКА находится цикл ДО. Сравни с рис. 79

- Надень защитную маску против пчел.
- Зайди на пасеку.

Дальше нужно взять мед в самом первом улье. Для этого Микола забирает мед из первого улья, делая это поочередно, порцию за порцией. При этом он исполняет команды, содержащиеся в цикле ПОКА (2):

- Мед из данного улья собран полностью?
- Собери следующую порцию меда из данного улья.

При выполнении цикла ПОКА (2) бегунок кружит по внутренней петле. За это время Микола берет одну порцию меда, потом другую и т. д. Когда он покончит с первым ульем, будет выполнено условие окончания цикла ПОКА (2).

Мед из данного улья собран полностью? = да

После этого Микола выполняет команды:

- Переходи к следующему улью.
- Мед из всех ульев собран полностью?

Если нет (если еще остались полные ульи) Микола продолжает обходить пасеку, улей за ульем. Когда он покончит с последним ульем будет выполнено условие окончания цикла ДО (1):

Мед из всех ульев собран полностью? = да

Предположим, на пасеке 30 ульев, причем из каждого улья Микола достает 6 порций меда. Это значит, что каждая команда цикла ДО будет выполнена 6 раз, а каждая команда цикла ПОКА – 180 раз. (Чтобы обойти 30 ульев, взяв из каждого 6 порций, нужно собрать $30 \times 6 = 180$ порций меда).

§5. ЦИКЛ «ПОКА» ВНУТРИ ЦИКЛА «ПОКА»

Рассмотрим алгоритм на рис. 98. Это рассказ о том, как Ваня получал деньги и тратил их на еду. Сначала Ваня выполняет две команды (рис. 98):

- Зайди в банк.
- Проверь свой счет.

Прежде всего, Ваня интересуется: «На счете есть деньги?». Если нет, то история заканчивается. Потому как на нет и суда нет.

Если же деньги есть, выполняется условие продолжения цикла ПОКА (1):

На счете есть деньги? = да

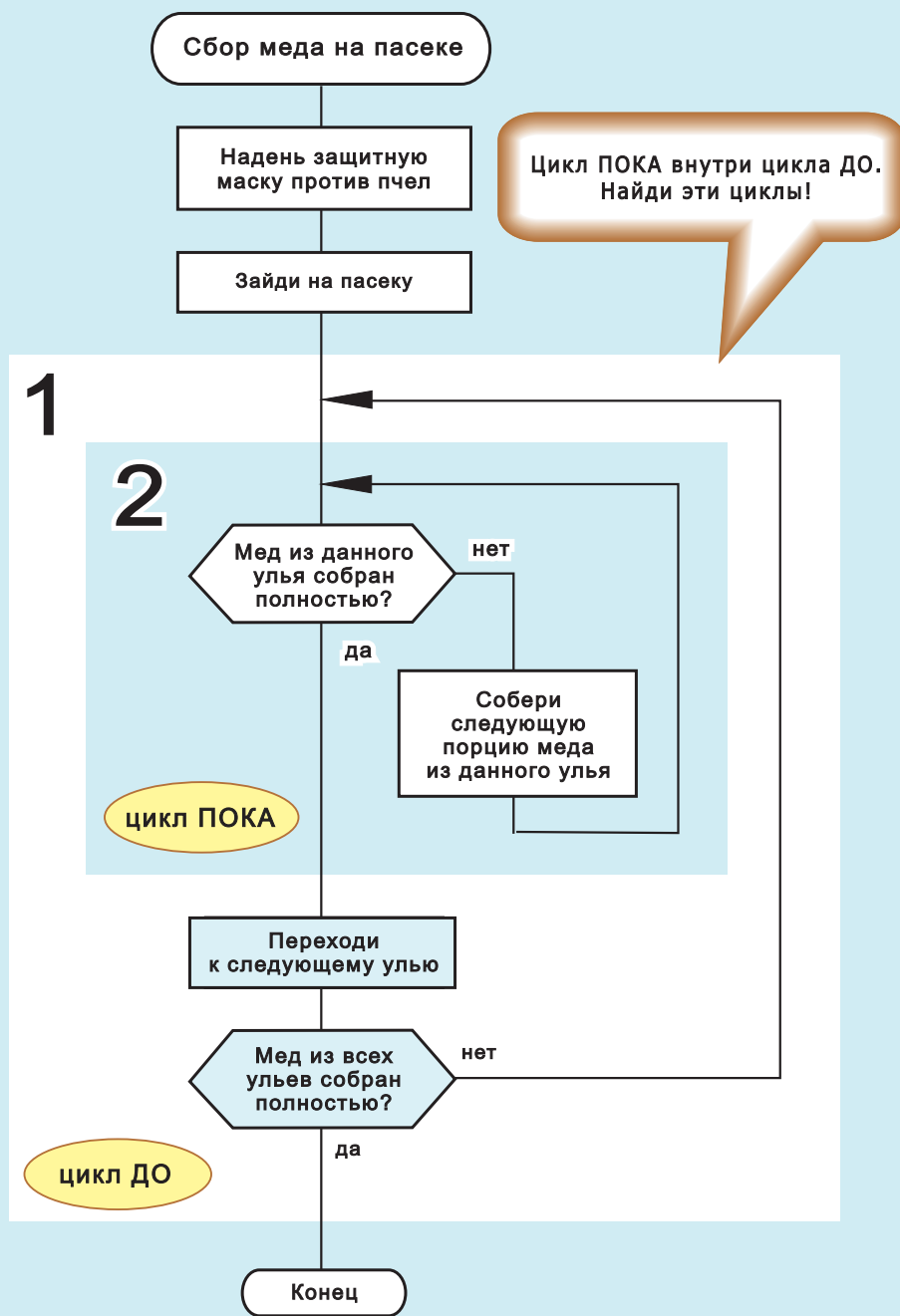


Рис. 97. Сбор меда на пасеке. Цикл ПОКА (2) внутри цикла ДО (1)

В этом случае Ваня выполняет команды:

- Сними со счета часть денег.
- Купи себе еду на неделю.

Затем начинает работу цикл ПОКА (2). Если еда есть, выполняется условие продолжения цикла ПОКА (2):

На сегодня еда есть? = да

Поэтому Ваня выполняет команды:

- Позавтракай.
- Пообедай.
- Поужинай.

Когда наступит следующий день, на вопрос «На сегодня еда есть?» следует ответ «да».

При выполнении цикла ПОКА (2) бегунок кружит по внутренней петле семь раз (то есть в течение недели). Когда неделя кончится, еда тоже кончится (поскольку еда была куплена ровно на неделю). Бегунок выходит из иконы «вопрос» через нет. И попадает на вход внешнего цикла ПОКА (1).

Перед Ваней вновь возникает вопрос «На счете есть деньги?». И так далее.

§6. КАК ВСТАВИТЬ ЦИКЛ «ДО» В ЦИКЛ «ПОКА» В ОБЩЕМ ВИДЕ

На рис. 99 показано построение абстрактной (буквенной) конструкции «цикл ДО в цикле ПОКА».

Слева на рис. 99 изображен буквенный цикл ПОКА. Найдите икону С и превратите ее в цикл ДО. Для этого внутрь иконы С вставьте цикл ДО (в центре). Затем переместите цикл ДО внутрь цикла ПОКА. Результат показан на рис. 99 (справа).

Таким образом, три схемы на рис. 99 отвечают на вопрос: как вставить цикл ДО внутрь цикла ПОКА.

§7. СТРУКТУРА «ЦИКЛ В ЦИКЛЕ»

На рис. 100 представлены четыре варианта структуры «цикл в цикле».

- цикл ДО (2) внутри цикла ДО (1);
- цикл ДО (2) внутри цикла ПОКА (1);
- цикл ПОКА (2) внутри цикла ДО (1);
- цикл ПОКА (2) внутри цикла ПОКА (1).

Внешний цикл обозначен цифрой 1, внутренний – цифрой 2. 1-й вариант показан на рис. 95 (Покраска забора).

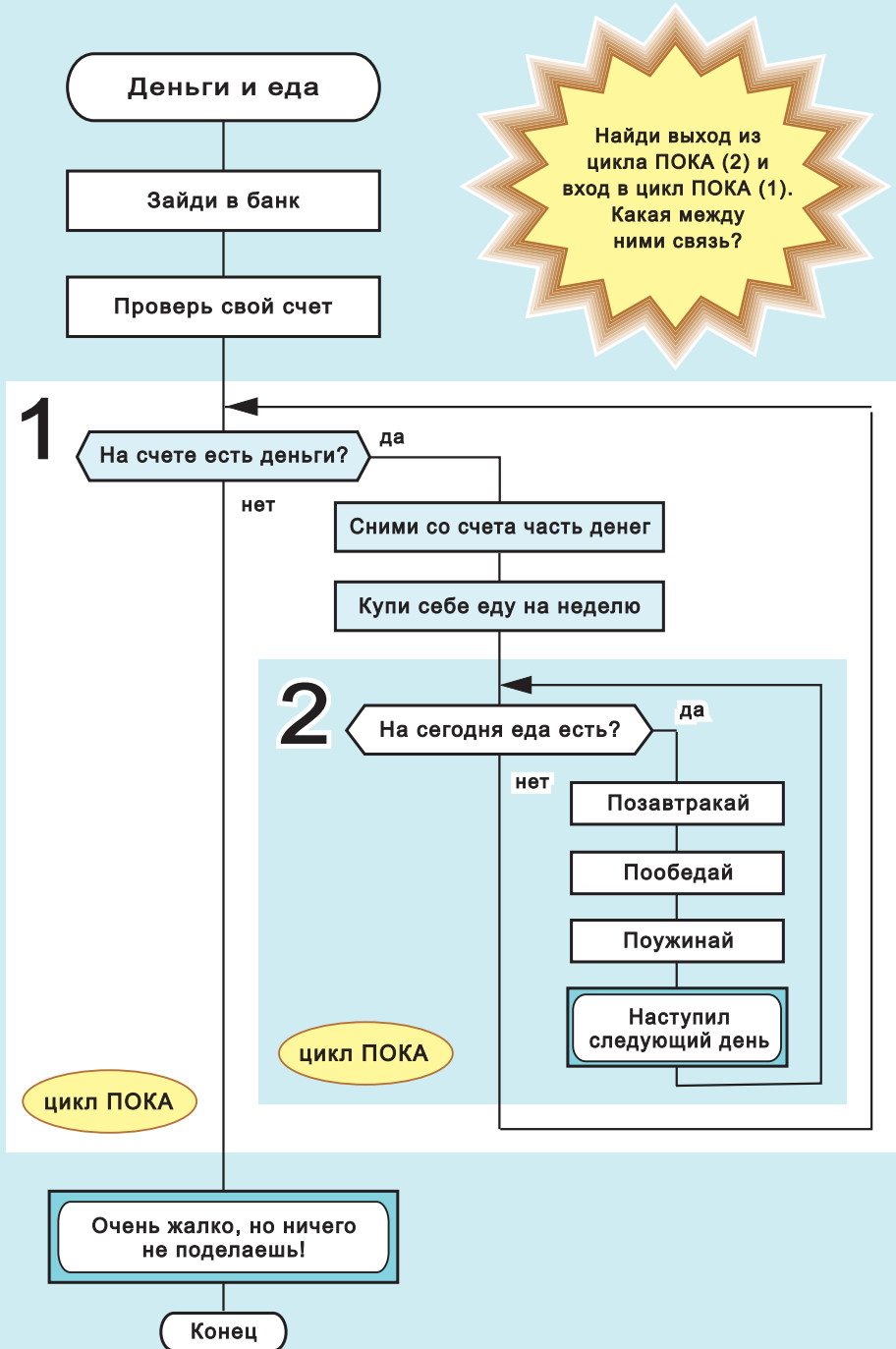


Рис. 98. Деньги и еда. Цикл ПОКА (2) внутри цикла ПОКА (1)

2-й вариант – на рис. 96 (Карлсон нашел кошелек).

3-й вариант – на рис. 97 (Сбор меда на пасеке).

4-й вариант – на рис. 98 (Деньги и еда).

Рис. 100 очень важен – он задает правила графического синтаксиса, используемые при создании конструкции «цикл в цикле».

Обратите внимание. Эти правила не надо учить и запоминать. Почему? Потому что правила «помнит» и выполняет компьютерная программа, которая называется «конструктор алгоритмов». Об этом подробно рассказано в главе 32.

Памятка

В любых дракон-схемах (в том числе, в циклах) запрещается использовать пересечение линий. Этот запрет выполняется автоматически – с помощью «конструктора алгоритмов». Рисунки в этой книге подтверждают это правило. Пересечения линий нигде и никогда не используются.

§8. ТРОЙНАЯ СТРУКТУРА «ЦИКЛ В ЦИКЛЕ»

На рис. 101 показаны три цикла ДО, вставленные друг в друга. При этом используются следующие обозначения:

1 – внешний цикл ДО.

2 – средний цикл ДО. Он находится внутри внешнего цикла ДО.

3 – внутренний цикл ДО. Он находится внутри среднего цикла ДО.

На рис. 102 представлены три цикла ПОКА, помещенные друг в друга. Приведем обозначения:

1 – внешний цикл ПОКА.

2 – средний цикл ПОКА. Он находится внутри внешнего цикла ПОКА.

3 – внутренний цикл ПОКА. Он находится внутри среднего цикла ПОКА.

§9. ТРОЙНАЯ СТРУКТУРА «ЦИКЛ В ЦИКЛЕ». ПОКАЗАНЫ ДОСРОЧНЫЕ ВЫХОДЫ И ВЕТОЧНЫЙ ЦИКЛ

В заключение рассмотрим задачу повышенной трудности. На дракон-схеме необходимо показать три основных и три досрочных выхода, а именно:

- досрочный выход из цикла ПОКА (1);
- основной выход из цикла ПОКА (1);
- досрочный выход из цикла ПОКА (2);
- основной выход из цикла ПОКА (2);
- досрочный выход из цикла ПОКА (3);
- основной выход из цикла ПОКА (3);

ЦИКЛ В ЦИКЛЕ

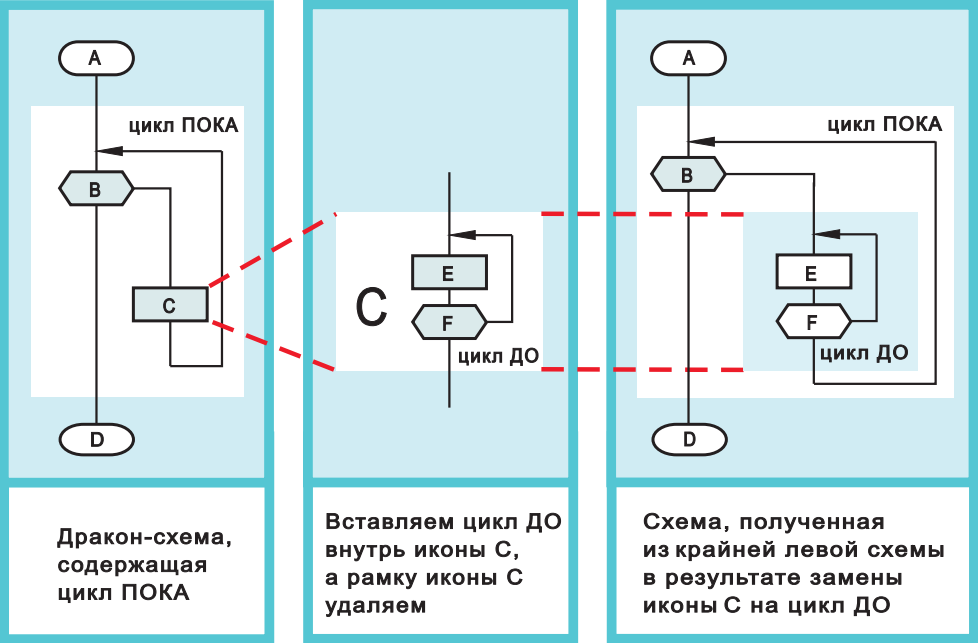


Рис. 99. Как построить цикл в цикле

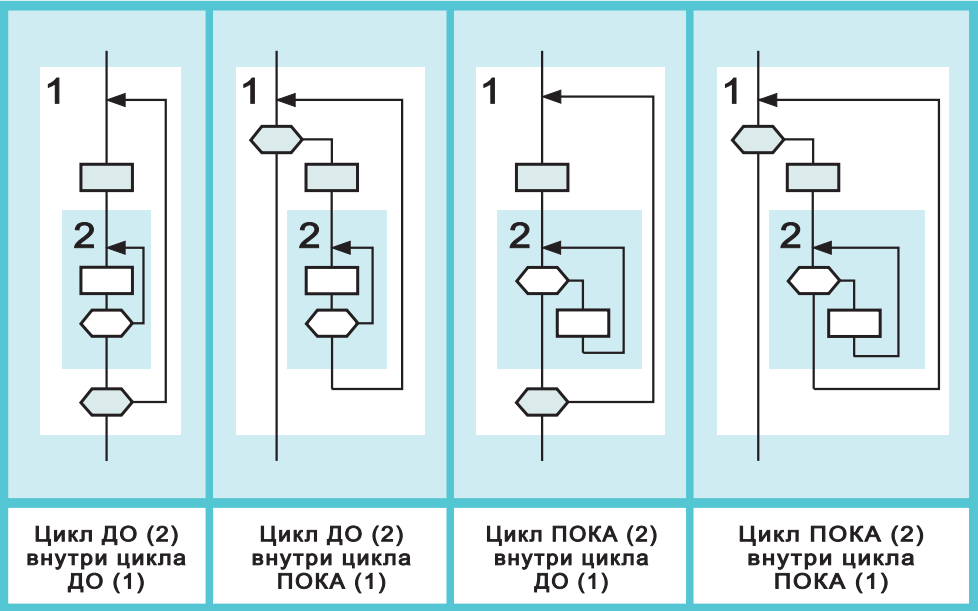
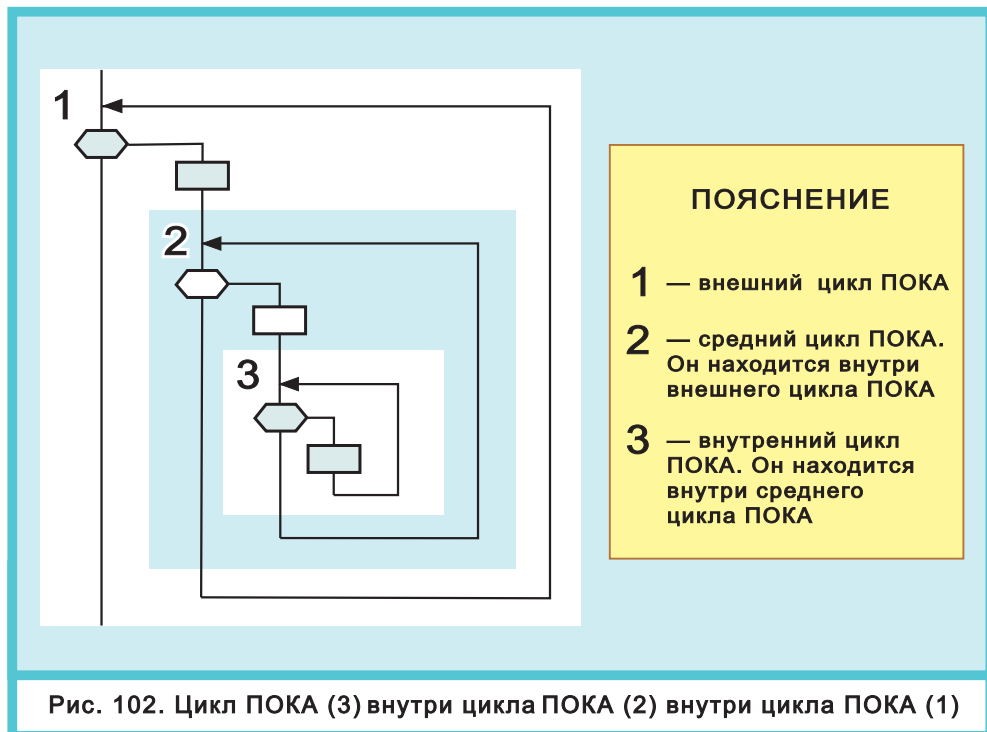
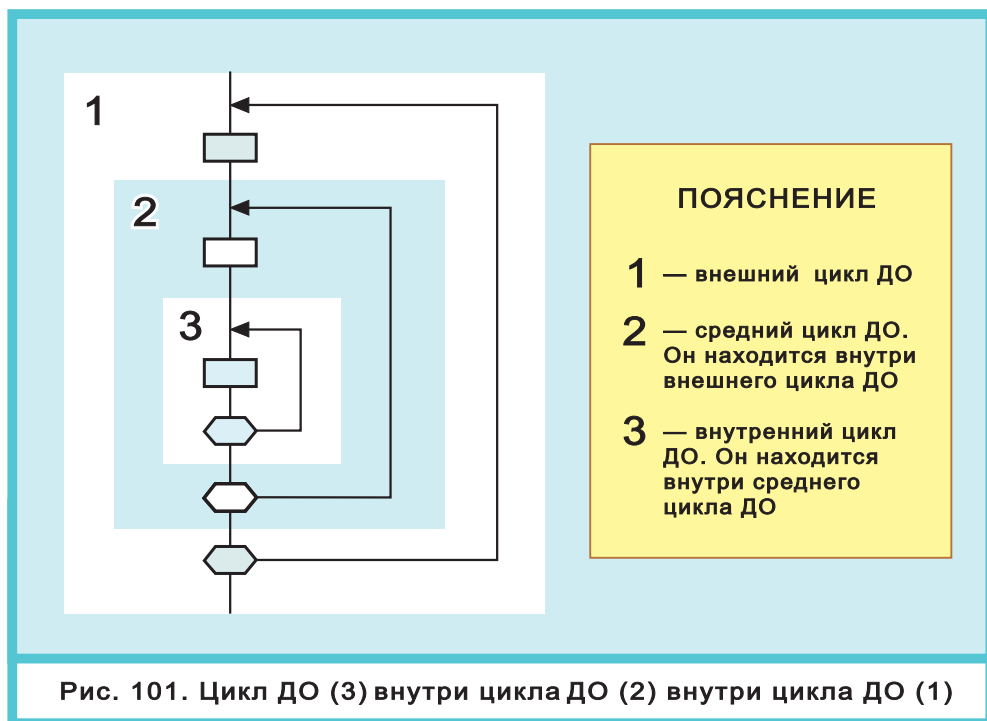


Рис. 100. Четыре варианта структуры «цикл в цикле»



ПОЯСНЕНИЕ

- 1** — внешний (веточный) цикл ПОКА. Бегунок делает петли между иконой «имя ветки» А и иконой «адрес» А. Показаны основной и досрочный выход из этого цикла
- 2** — средний цикл ПОКА. Стрелкой показана петля среднего цикла. Показаны основной и досрочный выход из этого цикла
- 3** — внутренний цикл ПОКА. Стрелкой показана петля внутреннего цикла. Показаны основной и досрочный выход из этого цикла

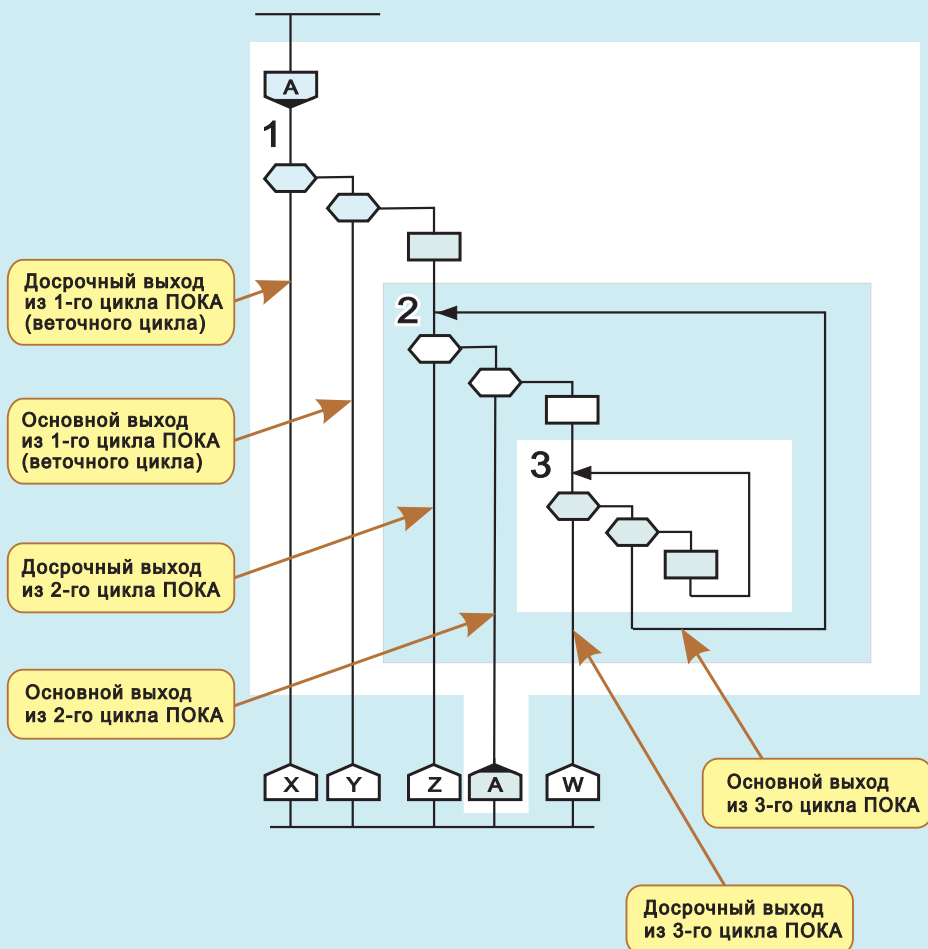


Рис. 103. Цикл ПОКА (3) внутри цикла ПОКА (2) внутри цикла ПОКА (1). Основные и досрочные выходы

Поставленная задача решена на рис. 103. Там изображена тройная структура «цикл в цикле» для цикла ПОКА. Эту конструкцию можно назвать так: «Цикл ПОКА (3) внутри цикла ПОКА (2) внутри цикла ПОКА (1).

Для каждого из циклов 1 (внешний), 2 (средний) и 3 (внутренний) показаны основной и досрочный выходы. Внешний цикл построен с использованием веточного цикла (рис. 103).

§10. ВЫВОДЫ

1. В главе рассмотрены четыре варианта структуры «цикл в цикле»:
 - цикл ДО внутри цикла ДО;
 - цикл ДО внутри цикла ПОКА;
 - цикл ПОКА внутри цикла ДО;
 - цикл ПОКА внутри цикла ПОКА.
2. Рис. 100 задает правила графического синтаксиса, используемые при создании конструкции «цикл в цикле».
3. Эти правила не надо учить и запоминать. Потому что правила «помнит» и выполняет программа «конструктор алгоритмов».

ЛОГИЧЕСКИЕ ФОРМУЛЫ, ИСПОЛЬЗУЕМЫЕ В АЛГОРИТМАХ

§1. ВИЗУАЛИЗАЦИЯ ЛОГИЧЕСКОЙ 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 рассказ. Будем считать, что покупка щенка возможна в том и только в том случае, когда выполняются три условия (рис. 104):

- у покупателя есть деньги (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. 104).

§2. ЖЕЛАТЕЛЬНО ИЗБЕГАТЬ СЛОЖНЫХ ВЫРАЖЕНИЙ

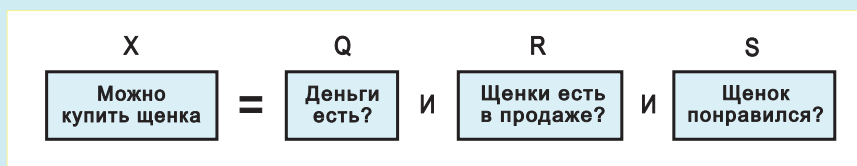
В традиционных алгоритмических языках значениями логических переменных считаются пары (ИСТИНА, ЛОЖЬ) или (1, 0). С эргономической точки зрения, такой подход нельзя признать удачным.

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

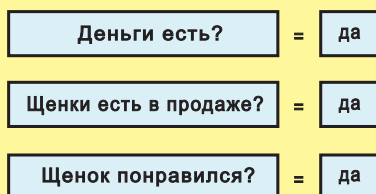
Ненужную сложность следует решительно устранить.

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».

ПРИМЕР ЛОГИЧЕСКОЙ ФУНКЦИИ «И»

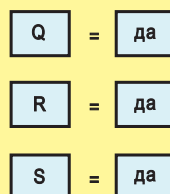


Логическая функция
«Можно купить щенка»
принимает значение «да»,
если одновременно
выполняются три условия:



Во всех остальных случаях
логическая функция
«Можно купить щенка»
принимает значение «нет»

Логическая функция
 $X = Q \text{ и } R \text{ и } S$
принимает значение «да»,
если одновременно
выполняются три условия:



Во всех остальных случаях
логическая функция
 $X = Q \text{ и } R \text{ и } S$
принимает значение «нет»

КАКИЕ СЛОВА ЭРГОНОМИЧНЕЕ: «ДА, НЕТ» ИЛИ «ИСТИНА, ЛОЖЬ»?

Большинство людей, отвечая на вопрос «Деньги есть?», говорят «да» или «нет». И это правильно.

Однако программисты, следуя правилам математической логики, поступают по-другому. Отвечая на тот же самый вопрос, они говорят «Истина» или «Ложь». В данном случае термины «Истина», «Ложь» слишком трудны и запутывают дело.

В языке Дракон этот недостаток устранен. И принято правило:
ЛОГИЧЕСКИЕ ПЕРЕМЕННЫЕ ПРИНИМАЮТ ЗНАЧЕНИЯ «ДА» И «НЕТ»

Рис. 104. Логическая функция «Можно купить щенка» является функцией трех логических переменных, связанных операций «И»

Логическая
переменная
величина

Это переменная величина,
которая принимает два значения:
«да» и «нет»

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

Правило. В языке ДРАКОН логические термины ИСТИНА, ЛОЖЬ, 1, 0 считаются нежелательными и, как правило, не используются.

§3. ЛОГИЧЕСКАЯ ФУНКЦИЯ «И»

Логическая
функция И

- Это функция, которая принимает значение «да», если все логические переменные имеют значение «да».
- В остальных случаях функция принимает значение «нет» (рис. 104).

§4. АЛГОРИТМЫ, ИСПОЛЪЗУЮЩИЕ ФУНКЦИЮ «И»

На рис. 105 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. 106.

§5. ДВА СПОСОБА ЗАПИСИ ФУНКЦИИ «И»

На языке ДРАКОН существуют два способа записи функции И: текстовый и визуальный (графический).

В первом случае используют одну икону «вопрос», внутри которой пишут логическое выражение, состоящее из логических переменных, соединенных знаками логической операции И (рис. 106, 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. 106, справа).

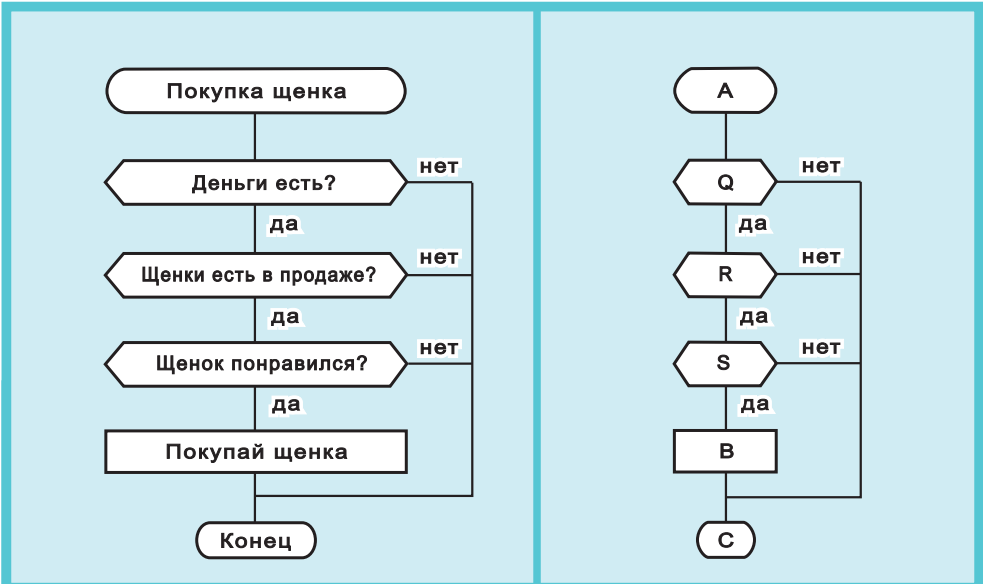
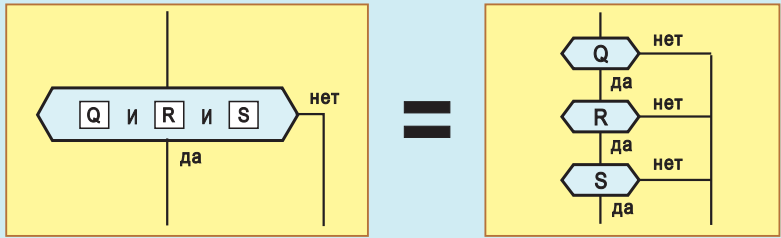


Рис. 105. Примеры алгоритмов с операцией «И»

ВИЗУАЛЬНАЯ ФОРМУЛА «И»



Алгоритм «И».
Нерекомендуемая схема

Логическое выражение построено с помощью одной иконы «вопрос», внутри которой записано составное условие Q и R и S

Алгоритм «И».
Рекомендуемая схема

Логическое выражение построено с помощью трех икон «вопрос», в каждой из которых записано простое условие

Рис. 106. Рисуйте дракон-схему «И», как показано справа. Избегайте нерекомендуемых схем

§6. КАКОЙ СПОСОБ ЛУЧШЕ: ТЕКСТОВЫЙ ИЛИ ВИЗУАЛЬНЫЙ?

Визуальная формула на рис. 106 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 выбирают визуальный способ как более легкий. Однако подготовленные специалисты, знакомые с основами математической логики, иногда предпочитают текстовый метод. Таким людям можно посоветовать освоить оба метода.

§7. СРАВНЕНИЕ МАТЕМАТИЧЕСКОЙ ФОРМУЛЫ И ДРАКОН-СХЕМЫ

На рис. 107 приведена математическая формула «И» и равносильная ей дракон-схема «И». Какую из них следует предпочесть? Какая является более понятной? 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 (рис. 108):

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

Два объекта на рис. 107 (текстовый слева и графический справа) математически эквивалентны. Это значит, что дракон-схема является *математической формулой*.

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

И последнее. Анализируя формулу слева на рис. 107, приходится вникать в сложные подробности, например:

$$X = [Q \& R \& S = \text{да}] = [(Q = \text{да}) \& (R = \text{да}) \& (S = \text{да})]$$

Дракон-схема (рис. 107, справа) хороша тем, что полностью избавляет читателя от подобной ненужной работы.

На рис. 107 и 108 вместо знака «И» использован знак «&» (конъюнкция). Оба знака имеют одинаковый смысл и обозначают одно и то же. Далее мы будем использовать оба знака.

МАТЕМАТИЧЕСКАЯ ФОРМУЛА

Если
($Q \& R \& S = \text{да}$)
то В

Условный оператор
Если X то В
с логическим выражением
 $X = (Q \& R \& S = \text{да})$

ДРАКОН-СХЕМА «И»

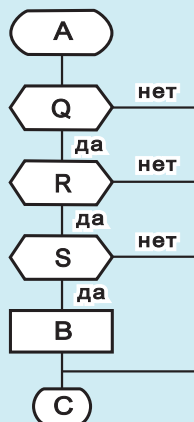
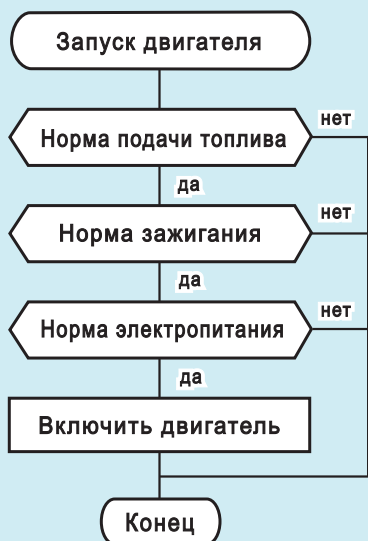


Рис. 107. Математическая формула и дракон-схема «И»



Пример производственного
алгоритма, в котором
используется логическая
функция «И»

Если
(Норма подачи топлива = да) &
(Норма зажигания = да) &
(Норма электропитания = да)
То
Включить двигатель

Рис. 108. Пример производственного алгоритма с операцией «И»

§8. ЛОГИЧЕСКАЯ СВЯЗКА «И»

Знак «И» имеет еще одно название – *логическая связка И*.

Почему связка? Потому что знак «И» связывает между собой логические переменные.

Например, на рис. 104 описана логическая функция $X = Q$ и R и S . В этой формуле логическая связка «И» повторяется два раза. Она связывает между собой три логические переменные Q , R , S .

§9. ВИЗУАЛИЗАЦИЯ ЛОГИЧЕСКОЙ ФУНКЦИИ «ИЛИ»

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

На этот счет может быть много 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. 109):

- 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. 109).

§10. 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. 109).

§11. АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ФУНКЦИЮ «ИЛИ»

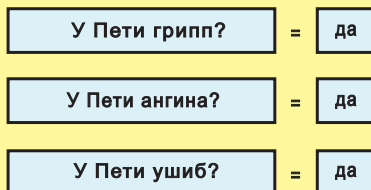
На рис. 110 приведены два примера алгоритмов. Слева рассказ о Петиних недугах. Справа тот же самый алгоритм представлен в абстрактной математической форме.

Попытайтесь в правом алгоритме найти и выделить логическую функцию «ИЛИ». Полученный результат пригодится при анализе рис. 111.

ПРИМЕР ЛОГИЧЕСКОЙ ФУНКЦИИ «ИЛИ»



Логическая функция «Петя заболел» принимает значение «да», если выполняется хотя бы одно из трех условий:



Во всех остальных случаях логическая функция «Петя заболел» принимает значение «нет»

Логическая функция $X = L \text{ или } M \text{ или } N$ принимает значение «да», если выполняется хотя бы одно из трех условий:



Во всех остальных случаях логическая функция $X = L \text{ или } M \text{ или } N$ принимает значение «нет»

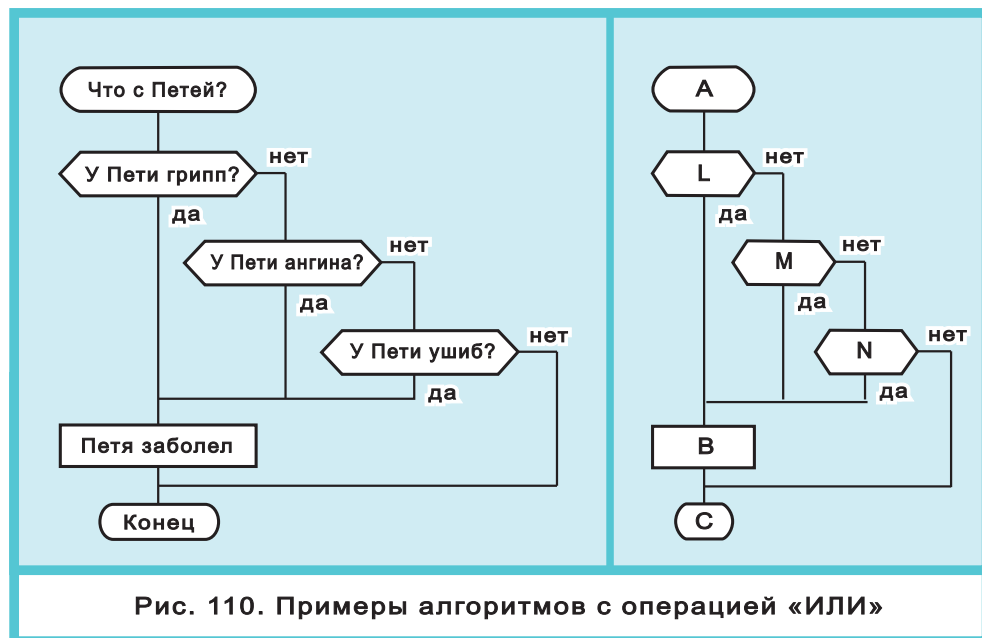
КАКИЕ СЛОВА ЭРГОНОМИЧНЕЕ: «ДА, НЕТ» ИЛИ «ИСТИНА, ЛОЖЬ»?

Большинство людей, отвечая на вопрос «Петя заболел?», отвечают «да» или «нет». И это правильно.

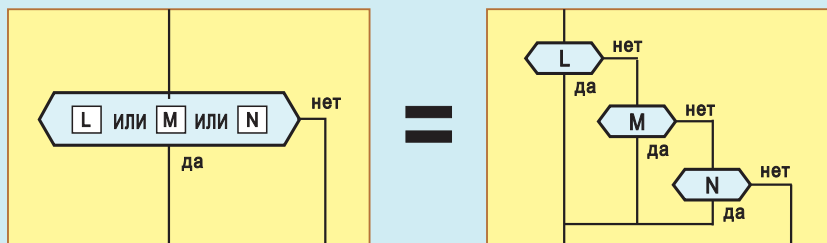
Однако программисты поступают по-другому. Отвечая на тот же самый вопрос, они говорят «Истина» или «Ложь». Это плохо. Потому что в данном случае термины «Истина» и «Ложь» слишком трудны и запутывают дело.

В языке Дракон этот недостаток устранен. И принято правило:
ЛОГИЧЕСКИЕ ПЕРЕМЕННЫЕ ПРИНИМАЮТ ЗНАЧЕНИЯ «ДА» И «НЕТ»

Рис. 109. Логическая функция «Петя заболел» является функцией трех логических переменных, связанных операцией «ИЛИ»



ВИЗУАЛЬНАЯ ФОРМУЛА «ИЛИ»



Алгоритм «ИЛИ». Нерекомендуемая схема

Логическое выражение построено с помощью одной иконы «вопрос», внутри которой записано составное условие «L или M или N»

Алгоритм «ИЛИ». Рекомендуемая схема

Логическое выражение построено с помощью трех икон «вопрос», в каждой из которых записано простое условие

Рис. 111. Рисуйте дракон-схему «ИЛИ», как показано справа. Избегайте нерекомендуемых схем

§12. ДВА СПОСОБА ЗАПИСИ ФУНКЦИИ «ИЛИ»

На языке ДРАКОН существуют два способа записи функции ИЛИ: текстовый и визуальный (графический).

В первом случае используют одну икону «вопрос», внутри которой пишут логическое выражение, состоящее из логических переменных, соединенных знаками логической операции ИЛИ (рис. 111, слева).

В другом случае лесенкой рисуют N икон «вопрос», где N – число логических переменных. Причем в каждой иконе записывают одну логическую переменную (рис. 111, справа).

§13. КАКОЙ СПОСОБ ЛУЧШЕ: ТЕКСТОВЫЙ ИЛИ ВИЗУАЛЬНЫЙ?

Визуальная формула на рис. 111 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.

§14. СРАВНЕНИЕ МАТЕМАТИЧЕСКОЙ ФОРМУЛЫ И ДРАКОН-СХЕМЫ

На рис. 112 приведена математическая формула и равносильная ей дракон-схема.

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

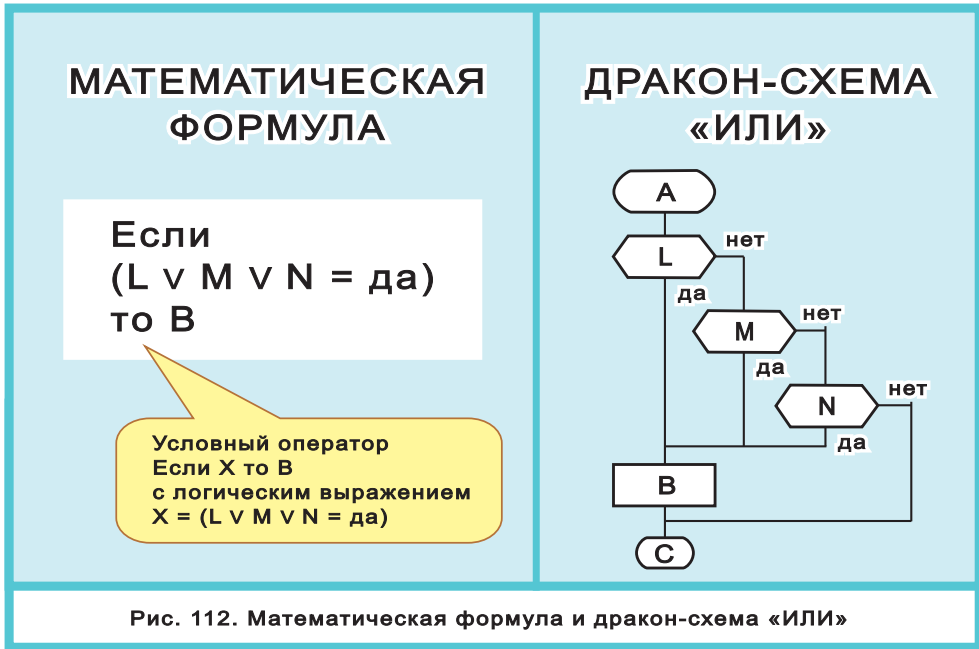
Справа – «демократическая» графическая формула, записанная на языке ДРАКОН. Она понятна значительно более широкому кругу работников. Как показывает практика, правая формула доступна даже тем людям, которые испытывают непреодолимые затруднения при работе со сложной левой формулой.

Анализируя формулу слева на рис. 112, 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. 112, справа) хороша тем, что избавляет читателя от подобной ненужной работы.

В §14 и на рис. 112 вместо знака «ИЛИ» использован знак « \vee » (дизъюнкция). Оба знака имеют одинаковый смысл и обозначают одно и то же. Далее мы будем использовать оба знака.



§15. ЛОГИЧЕСКАЯ СВЯЗКА «ИЛИ»

Знак «ИЛИ» имеет еще одно название – *логическая связка ИЛИ*.

Почему связка? Потому что знак «ИЛИ» связывает между собой логические переменные.

Например, на рис.109 описана логическая функция $X = L$ или M или N . В этой формуле логическая связка «ИЛИ» повторяется два раза. Она связывает между собой три логические переменные L, M, N .

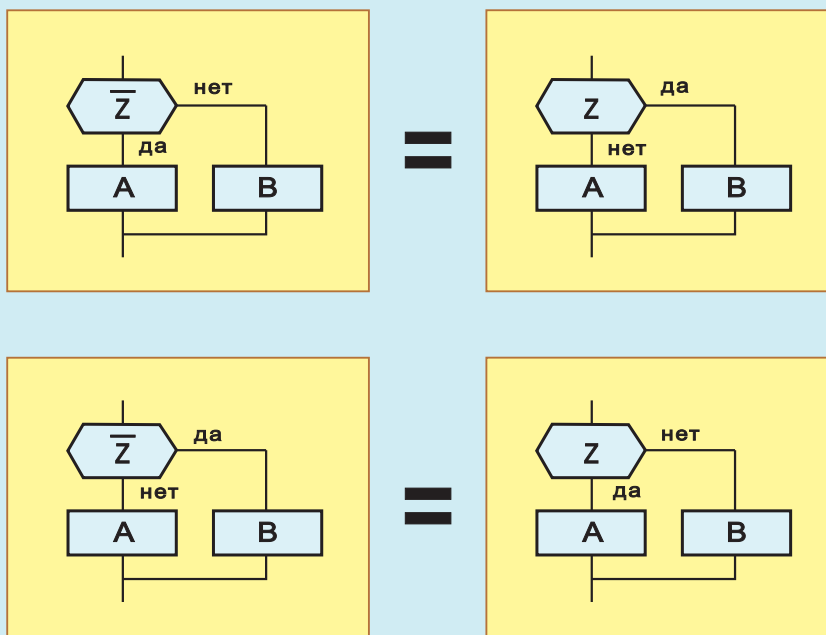
§16. ЛОГИЧЕСКАЯ ФУНКЦИЯ «НЕ»

Логическая
функция НЕ

- Это функция $W = \overline{Z}$, где логические переменные Z и W принимают инверсные значения, то есть удовлетворяют условиям:
- если $Z = \text{да}$, то $W = \text{нет}$;
- если $Z = \text{нет}$, то $W = \text{да}$.

Иллюстрация приведена на рис. 113.

ВИЗУАЛЬНЫЕ ФОРМУЛЫ



Схемы слева содержат двойное отрицание.

С точки зрения эргономики, двойное отрицание нежелательно, так как оно увеличивает вероятность ошибок.

Схемы справа позволяют убрать двойное отрицание

Алгоритм «НЕ». Нерекомендуемая схема

Логическое выражение «НЕ» построено с помощью знака логического отрицания – верхней черты

Алгоритм «НЕ». Рекомендуемая схема

Чтобы убрать знак логического отрицания (верхнюю черту), поменяйте местами надписи «да» и «нет»

Рис. 113. Визуальные формулы, позволяющие освободиться от знака логического отрицания (верхней черты).

§17. ЛОГИЧЕСКОЕ ОТРИЦАНИЕ ЖЕЛАТЕЛЬНО УСТРАНИТЬ ИЗ ДРАКОН-СХЕМ

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

«Если это возможно, избегайте отрицаний в булевых [логических] выражениях. Представляется, что их понимание представляет трудность для многих программистов» [1].

Поясним. В похожие ловушки часто попадают не только программисты. Подобные трудности испытывают и многие другие люди.

К счастью, от этой неприятности можно избавиться. Ниже будет показано, что логическое отрицание (и кое-что еще) можно безболезненно изъять из графических логических выражений.

§18. КАК УБРАТЬ ЗНАКИ «И», «ИЛИ», «НЕ» ИЗ ИКОНЫ ВОПРОС?

Эргономика позволяет сделать алгоритмы (дракон-схемы) более легкими, удобными для понимания. Глядя на эргономичную дракон-схему, человек может сказать: «Посмотрел – и сразу понял!». «Взглянул – и мигом во всем разобрался!».

Многие люди испытывают трудности, когда видят, что внутри икон вопрос записаны сложные логические формулы, содержащие знаки И, ИЛИ, НЕ. Таким людям можно помочь, устранив эти «неприятные» знаки.

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

Доказательство представлено на рис. 114. Этот рисунок содержит *практические* приемы, которые нужно запомнить. Ниже даны пояснения в виде трех вопросов и трех ответов.

Вопрос 1. Как удалить из иконы вопрос логическое отрицание НЕ (верхнюю черту) – см. рис. 114, слева вверху.

Ответ 1. Удалите из иконы вопрос верхнюю черту и поменяйте местами слова «да» и «нет» (рис. 114, справа вверху).

Вопрос 2. Как удалить из иконы вопрос логический знак И (рис. 114, слева, чуть ниже).

Ответ 2. Вместо одной иконы вопрос со знаком И нарисуйте две иконы вопрос:

- не содержащие знак И;
- расположенные на одной вертикали;

- помеченные словом «да» у нижнего выхода (рис. 114, справа, чуть ниже).

Вопрос 3. Как удалить из иконы вопрос логический знак ИЛИ (рис. 114, слева, внизу).

Ответ 3. Вместо одной иконы вопрос со знаком ИЛИ нарисуйте две иконы вопрос:

- не содержащие знак ИЛИ;
- расположенные лесенкой;
- помеченные словом «да» у нижнего выхода (рис. 114, справа внизу).

Логические знаки И, ИЛИ, НЕ всегда можно убрать из дракон-схем.

§19. ЛОГИЧЕСКИЙ ФРАГМЕНТ ДРАКОН-СХЕМЫ

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

Примеры логических фрагментов показаны на рис. 106, 111, 114–116.

§20. СТАНДАРТНАЯ И НЕСТАНДАРТНАЯ ДРАКОН-СХЕМА «И» ДЛЯ ДВУХ ЛОГИЧЕСКИХ ПЕРЕМЕННЫХ

Стандартная дракон-схема «И» для двух логических переменных – это схема, которая содержит две иконы вопрос:

- расположенные на одной вертикали;
- помеченные словом «да» у нижнего выхода;
- в каждой иконе вопрос содержится одна переменная.

Стандартная схема «И» представлена на рис. 115, слева.

Нестандартная дракон-схема «И» для двух логических переменных – это схема, которая получена с помощью рокировки схемы «И» (рис. 115, справа).

Стандартная и нестандартная схемы «И» равносильны. Они описывают один и тот же алгоритм.

§21. ОТЛИЧИЯ МЕЖДУ СТАНДАРТНОЙ И НЕСТАНДАРТНОЙ СХЕМАМИ «И»

Укажем отличия между схемами.

- В стандартной схеме «И» обе иконы «вопрос» лежат на шампуре (рис. 115, слева).
- В нестандартной схеме «И» иконы «вопрос» расположены лесенкой (рис. 115, справа).

- В стандартной схеме нижние выходы икон «вопрос» помечены словом «да» (рис. 115, слева).
- В нестандартной – словом «нет» (рис. 115, справа).

- В стандартной схеме левый выход считается главным, правый – инверсным.
- В нестандартной схеме все наоборот. Левый выход инверсный, а правый – главный.

- В стандартной схеме главный выход лежит на шампуре и обозначается буквой X . Инверсный выход находится справа от шампура и обозначается буквой \bar{X} (рис. 115, слева).
- В нестандартной схеме, наоборот, главный выход расположен справа от шампура. Инверсный – лежит на шампуре (рис. 115, справа).

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

§22. СТАНДАРТНАЯ И НЕСТАНДАРТНАЯ ДРАКОН-СХЕМА «ИЛИ» ДЛЯ ДВУХ ЛОГИЧЕСКИХ ПЕРЕМЕННЫХ

Стандартная дракон-схема «ИЛИ» для двух логических переменных – это схема, которая содержит две иконы вопрос:

- расположенные лесенкой;
- помеченные словом «да» у нижнего выхода;
- в каждой иконе вопрос содержится одна переменная.

Стандартная схема «ИЛИ» представлена на рис. 116, слева

Нестандартная дракон-схема «ИЛИ» для двух логических переменных – это схема, которая получена с помощью рокировки схемы «ИЛИ» (рис. 116, справа).

Стандартная и нестандартная схемы «ИЛИ» равносильны. Они описывают один и тот же алгоритм.

§23. ОТЛИЧИЯ МЕЖДУ СТАНДАРТНОЙ И НЕСТАНДАРТНОЙ СХЕМАМИ «ИЛИ»

Укажем отличия между схемами «ИЛИ».

- В стандартной схеме «ИЛИ» две иконы «вопрос» расположены лесенкой (рис. 116, слева).
- В нестандартной схеме «ИЛИ» обе иконы «вопрос» лежат на шампуре (рис. 116, справа).

- В стандартной схеме нижние выходы икон «вопрос» помечены словом «да» (рис. 116, слева).
- В нестандартной – словом «нет» (рис. 116, справа).

- В стандартной схеме левый выход считается главным, правый – инверсным.
- В нестандартной схеме все наоборот. Левый выход инверсный, а правый – главный.

- В стандартной схеме главный выход лежит на шампуре и обозначается буквой X . Инверсный выход находится справа от шампура и обозначается буквой \bar{X} (рис. 116, слева).
- В нестандартной схеме, наоборот, главный выход расположен справа от шампура. Инверсный – лежит на шампуре (рис. 116, справа).

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

§24. ОБОБЩЕНИЕ

Обобщим материал, представленный в §21 и §23.

Обратите внимание: отличия показаны в желтых рамках. В зеленых рамках отличий НЕТ. Отсюда вытекает правило:

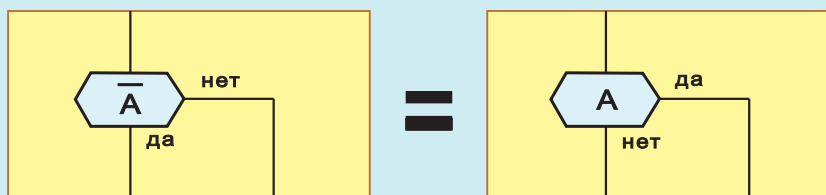
Главный выход X для стандартных схем И и ИЛИ находится на шампуре.

Инверсный выход \bar{X} для нестандартных схем И и ИЛИ находится на шампуре.

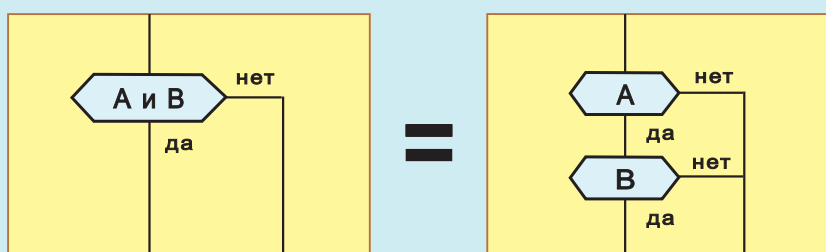
В заключение приведем теорему, которая справедлива для любого логического фрагмента.

Теорема. Если главный выход логического фрагмента есть результат вычисления логической функции \underline{X} , то инверсный выход вычисляет логическое отрицание \bar{X} .

КАК УДАЛИТЬ ЛОГИЧЕСКОЕ ОТРИЦАНИЕ



КАК УДАЛИТЬ ЛОГИЧЕСКУЮ СВЯЗКУ «И»



КАК УДАЛИТЬ ЛОГИЧЕСКУЮ СВЯЗКУ «ИЛИ»

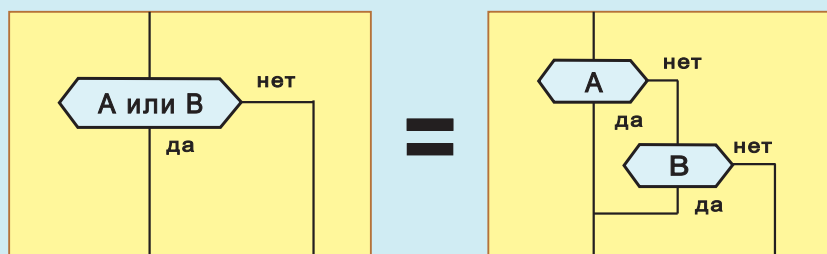
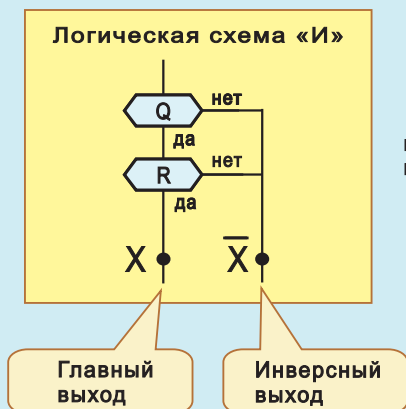


Рис. 114. Как удалить из иконы «вопрос» знаки НЕ, И, ИЛИ?

СТАНДАРТНАЯ СХЕМА «И»



НЕСТАНДАРТНАЯ СХЕМА «И»

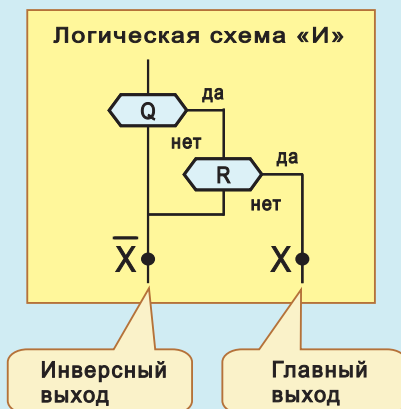
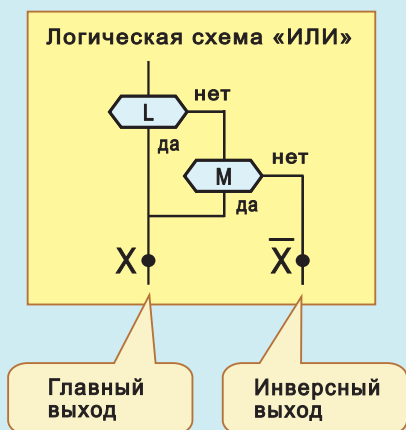


Рис. 115. Стандартный и нестандартный вид логической схемы «И»

СТАНДАРТНАЯ СХЕМА «ИЛИ»



НЕСТАНДАРТНАЯ СХЕМА «ИЛИ»

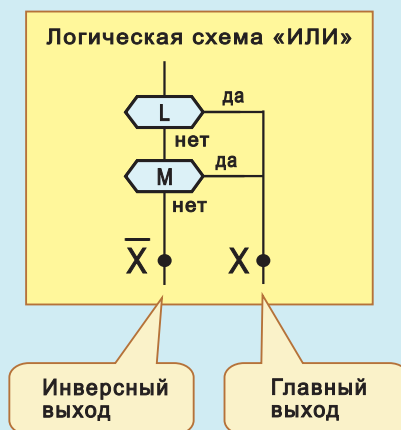


Рис. 116. Стандартный и нестандартный вид логической схемы «ИЛИ»

§25. КАНОНИЧЕСКАЯ ДРАКОН-СХЕМА

Каноническая
дракон-схема

Это эргономичная дракон-схема, выполняющая логические операции, которая:

- способна описать *любую* логическую функцию;
- не содержит логических знаков И, ИЛИ, НЕ;
- не содержит повторяющихся частей, которые можно удалить с помощью равносильных преобразований.

§26. ЧЕМ РАЗЛИЧАЮТСЯ КАНОНИЧЕСКАЯ И СТАНДАРТНАЯ ДРАКОН-СХЕМА?

Стандартная схема – жестко определенное понятие. Ее графическая форма и все ее элементы заданы. Они заданы логической формулой. Например, формула $X = Q$ и R однозначно задает стандартную схему «И» для двух переменных Q, R , показанную на рис. 115, слева.

Каноническая схема – понятие гибкое. Эта схема может менять свою графическую форму и входящие в нее элементы.

В основе любой схемы (и канонической, и стандартной) лежит логическая формула. Стандартная схема может описывать ОДНУ-ЕДИНСТВЕННУЮ формулу. В отличие от нее, каноническая схема может описывать не одну-единственную, а ЛЮБУЮ логическую формулу.

В самом деле, напишите любую логическую формулу. Для выбранной вами формулы можно начертить соответствующую ей каноническую схему.

§27. ПРИВЕДЕНИЕ ДРАКОН-СХЕМЫ К КАНОНИЧЕСКОМУ ВИДУ

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

Рассмотрим приведение к каноническому виду для четырех частных случаев (рис. 117–120).

В первом примере используются три преобразования (рис. 117):

- удаление логического отрицания;
- рокировка;
- удаление логической связки И.

На рис. 117 видно, что в данном случае каноническая схема совпадает со стандартной схемой «И» для двух переменных.

Во втором примере используются преобразования (рис. 118):

- удаление логического отрицания;
- рокировка;
- удаление логической связки ИЛИ.

Этот пример демонстрирует, что каноническая схема совпадает со стандартной схемой «ИЛИ» для двух переменных.

Всегда ли так бывает? Всегда ли каноническая схема совпадает со стандартной? Нет, не всегда. Чтобы убедиться в этом, рассмотрим еще два примера.

Во третьем примере используются четыре преобразования (рис. 119):

- удаление логического отрицания;
- рокировка;
- удаление логической связки И;
- удаление логического отрицания.

Анализ рисунка 119 свидетельствует, что в данном случае каноническая схема отличается от стандартной. В самом деле, в стандартной схеме «И» нижний выход иконы *B* помечен словом «да». А в канонической – указанный выход помечен словом «нет». Следовательно, каноническая и стандартная схемы – разные вещи.

Во четвертом примере используются три преобразования (рис. 120):

- удаление логической связки «И»;
- удаление логического отрицания;
- рокировка.

Рис. 120 показывает, что и в этом случае каноническая схема отличается от стандартной. В самом деле, в стандартной схеме «ИЛИ» нижний выход иконы *A* помечен словом «да». А в канонической – указанный выход помечен словом «нет». Следовательно, каноническая и стандартная схемы отличаются друг от друга.

На рис. 121 представлены упражнения, которые помогут читателю закрепить материал.

§28. СТАНДАРТНЫЕ И НЕСТАНДАРТНЫЕ ДРАКОН-СХЕМЫ «И», «ИЛИ» ДЛЯ ТРЕХ ЛОГИЧЕСКИХ ПЕРЕМЕННЫХ

Выше, на рис. 115 и 116 мы изучили стандартные и нестандартные дракон-схемы для двух логических переменных. Для схемы И переменные были обозначены буквами *Q* и *R*. Для схемы ИЛИ – буквами *L* и *M*.

Теперь рассмотрим ту же задачу для трех логических переменных (рис. 122). Для схемы И переменные обозначены буквами *Q*, *R*, *S*. Для схемы ИЛИ – буквами *L*, *M*, *N*.

Сравните рис. 122 с рис. 115 и 116. Выявите сходство и отличия.

Нарисуйте аналогичные рисунки для четырех логических переменных.

ПРИВЕДЕНИЕ ДРАКОН-СХЕМЫ К КАНОНИЧЕСКОМУ ВИДУ

(Пример 1)

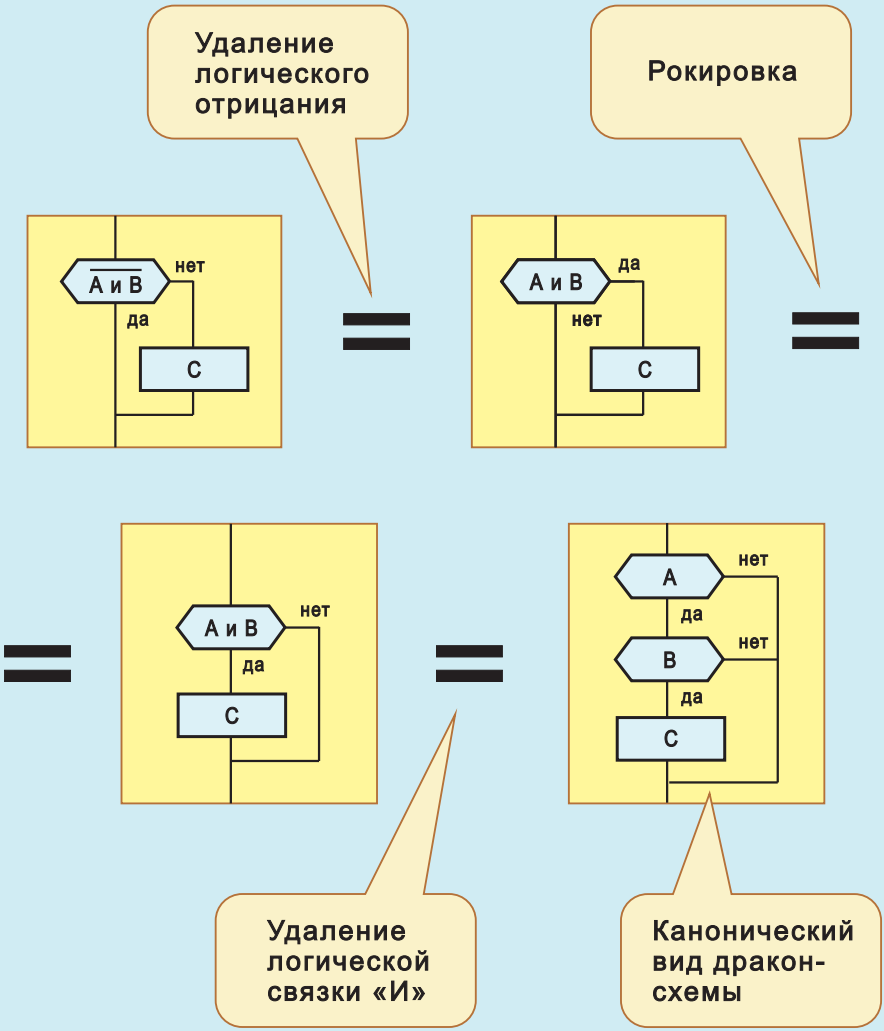


Рис. 117. Цепочка равносильных преобразований позволяет привести дракон-схему к каноническому виду. (Пример 1)

ПРИВЕДЕНИЕ ДРАКОН-СХЕМЫ К КАНОНИЧЕСКОМУ ВИДУ

(Пример 2)

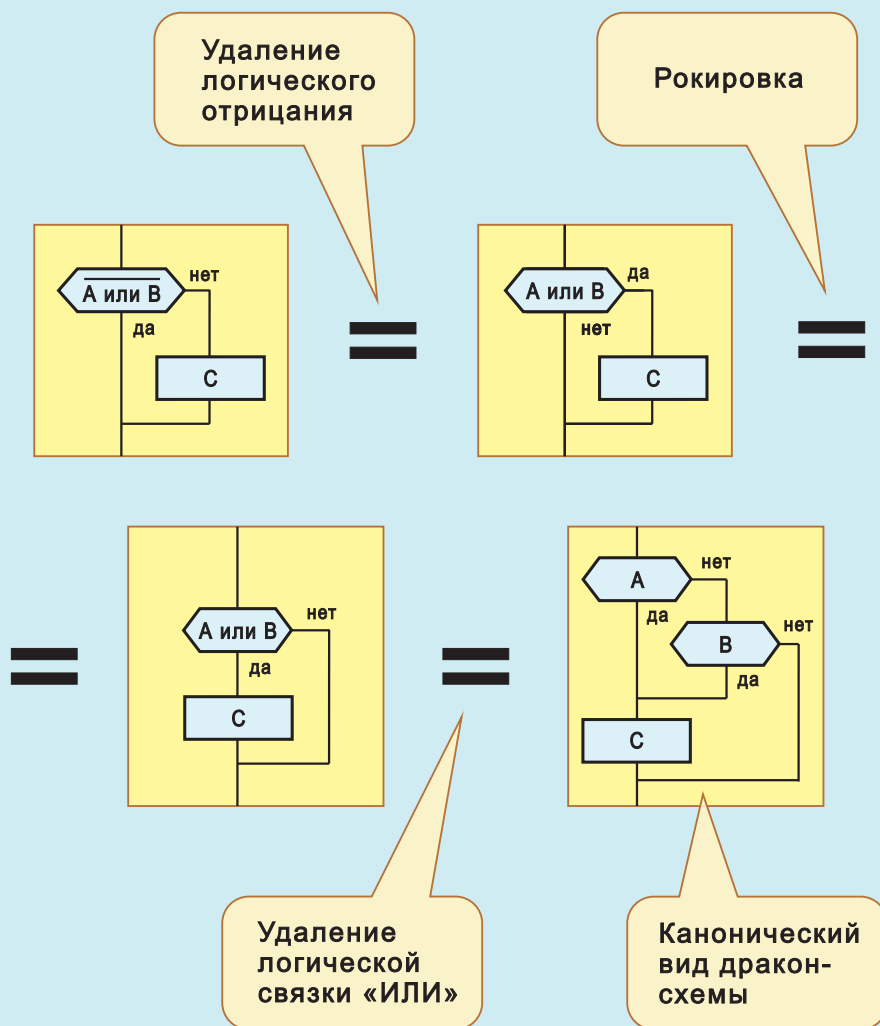


Рис. 118. Цепочка равносильных преобразований позволяет привести дракон-схему к каноническому виду. (Пример 2)

ПРИВЕДЕНИЕ ДРАКОН-СХЕМЫ К КАНОНИЧЕСКОМУ ВИДУ

(Пример 3)

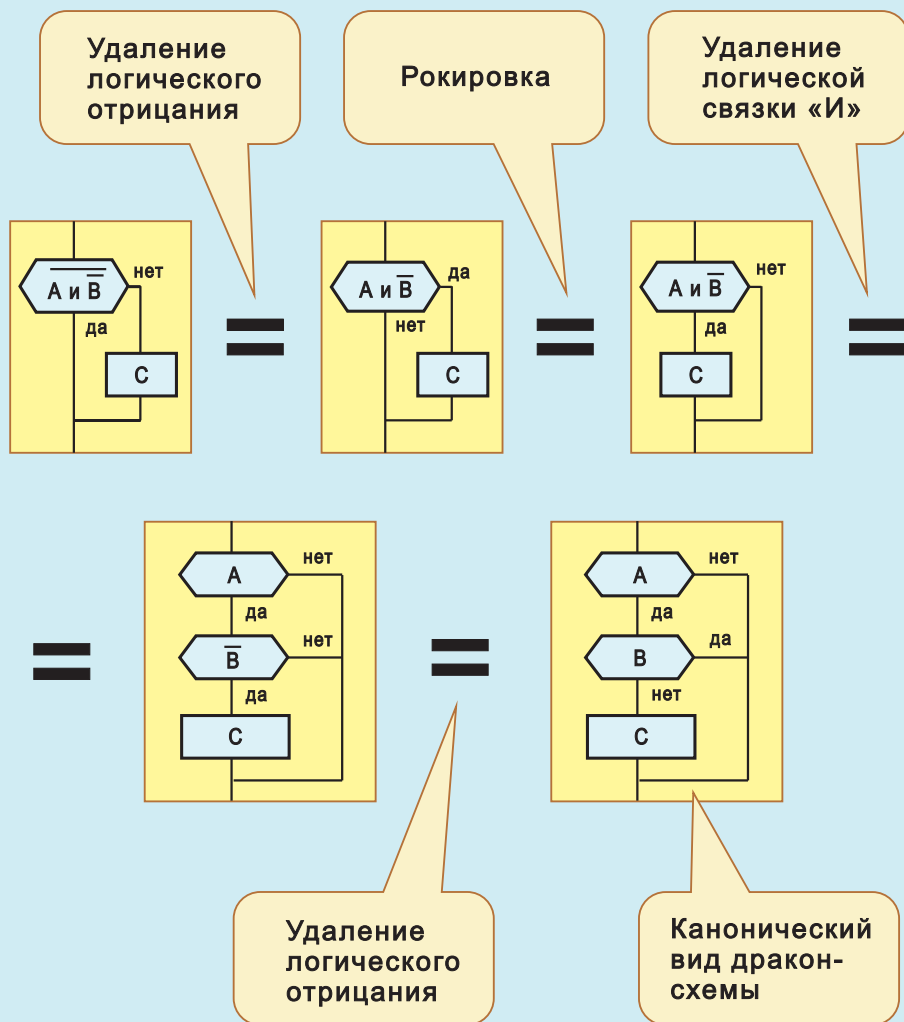


Рис. 119. Цепочка равносильных преобразований позволяет привести дракон-схему к каноническому виду. (Пример 3)

ПРИВЕДЕНИЕ ДРАКОН-СХЕМЫ К КАНОНИЧЕСКОМУ ВИДУ

(Пример 4)

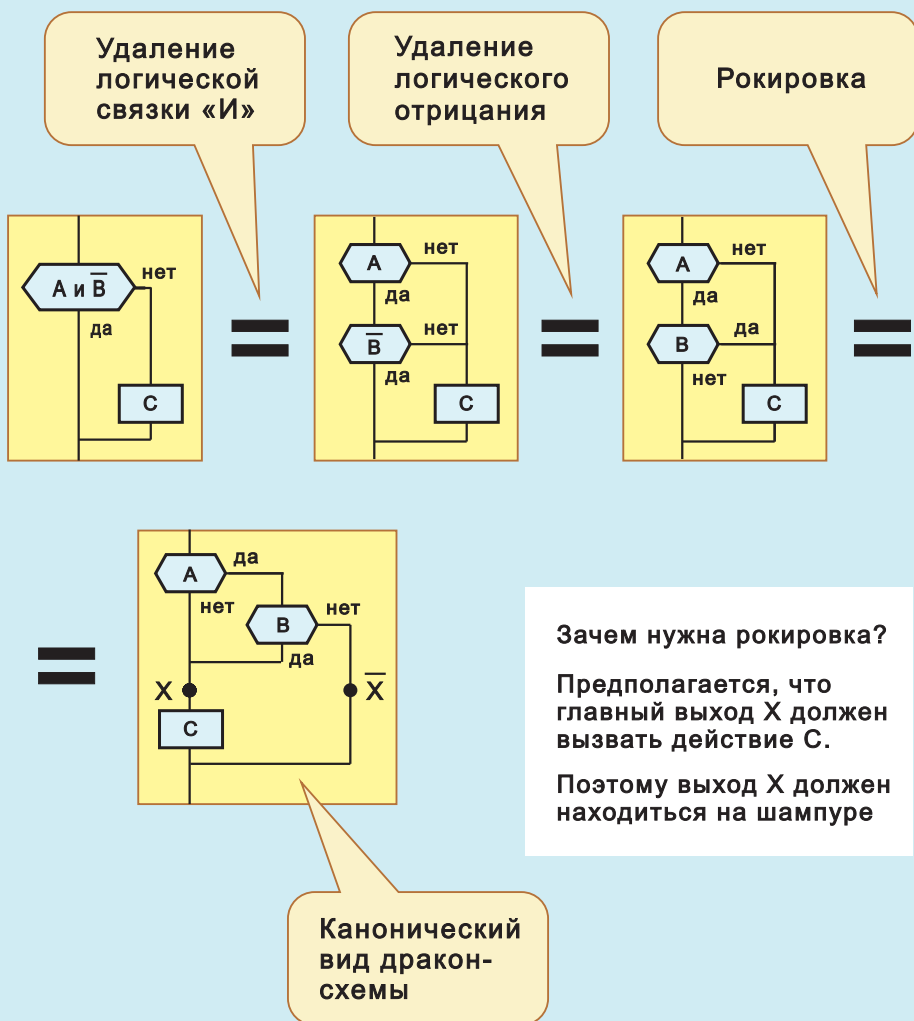


Рис. 120. Цепочка равносильных преобразований позволяет привести дракон-схему к каноническому виду. (Пример 4)

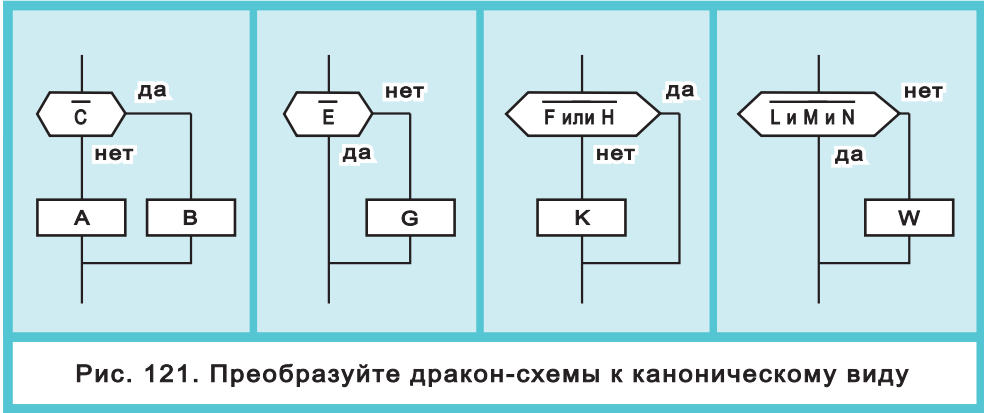


Рис. 121. Преобразуйте дракон-схемы к каноническому виду

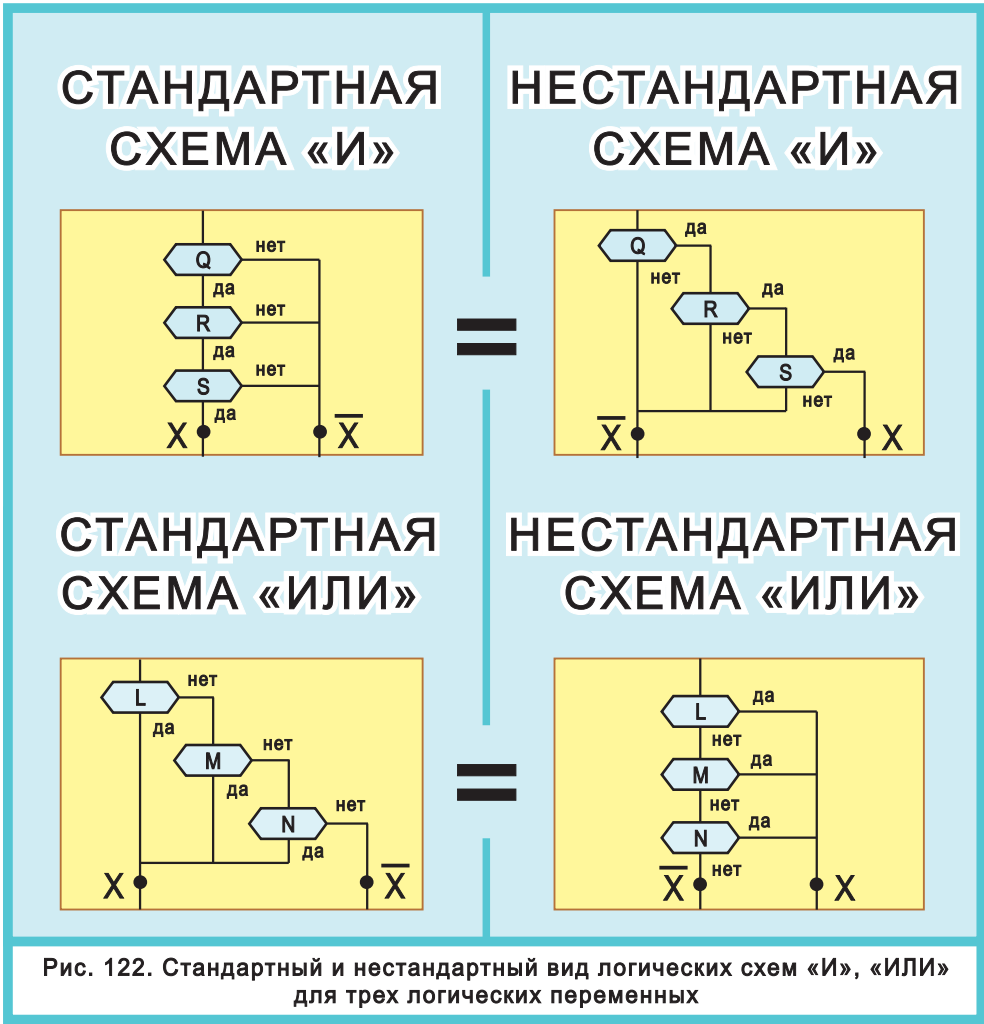


Рис. 122. Стандартный и нестандартный вид логических схем «И», «ИЛИ» для трех логических переменных

§29. ВИЗУАЛИЗАЦИЯ СЛОЖНЫХ ЛОГИЧЕСКИХ ФУНКЦИЙ

Рассмотрим функцию

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

На рис. 123 показан визуальный способ записи этой функции. На рисунке видно, что формула (1) разбивается на три части:

- A и \bar{B} и C ;
- D и E и \bar{F}
- операция «ИЛИ».

Функция A и \bar{B} и C изображается с помощью трех икон A , \bar{B} , C , расположенных на одной вертикали. Аналогично рисуют функцию D и E и \bar{F} . Связка ИЛИ реализуется с помощью линий, объединяющих нижние выходы икон C и \bar{F} в точке K (рис. 123).

В формуле (1) некоторые члены записаны без логического отрицания (A , B , D , E), другие – с отрицанием (\bar{B} , \bar{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. 123).

Другие примеры алгоритмов, вычисляющих сложные логические функции, представлены на рис. 124, 125.

§30. КОГДА ЛУЧШЕ ИСПОЛЬЗОВАТЬ СТАНДАРТНУЮ СХЕМУ, А КОГДА – НЕСТАНДАРТНУЮ?

На рис. 126 представлен алгоритм проверки приборов – но не весь, а только его правая часть. Левая часть алгоритма опущена, о чем говорят две линии обрыва.

Ветка «Контроль приборов» двухадресная. В ней две иконы адрес. Слева нарисована икона адрес «завершение», справа – «контроль электросети».

Здесь есть ошибка. Иконы адрес перепутаны местами. Слева должна стоять икона адрес «контроль электросети», справа – «завершение»

В чем причина ошибки? Как ее исправить?

Сначала сделаем пояснение. В ветке «Контроль приборов» проверяются три прибора: синий, зеленый и желтый. Если испортился (отказал) хотя бы один прибор из трех, это значит, что хотя бы из одной иконы вопрос выходим через «да». В результате происходит отбой системы, и алгоритм заканчивает работу (рис. 126).

Если же все три прибора исправны (нет ни одного отказа), то из всех икон вопрос выходим через «нет». Затем запускается ветка «Контроль электросети». Производится проверка трех сетей (синей, зеленой и желтой). После этого алгоритм завершается.

Теперь расскажем об ошибке.

Выше (глава 6, §4) приведена рекомендация для двухадресных веток. Напомним ее.

ПРИМЕР СЛОЖНОЙ ЛОГИЧЕСКОЙ ФУНКЦИИ

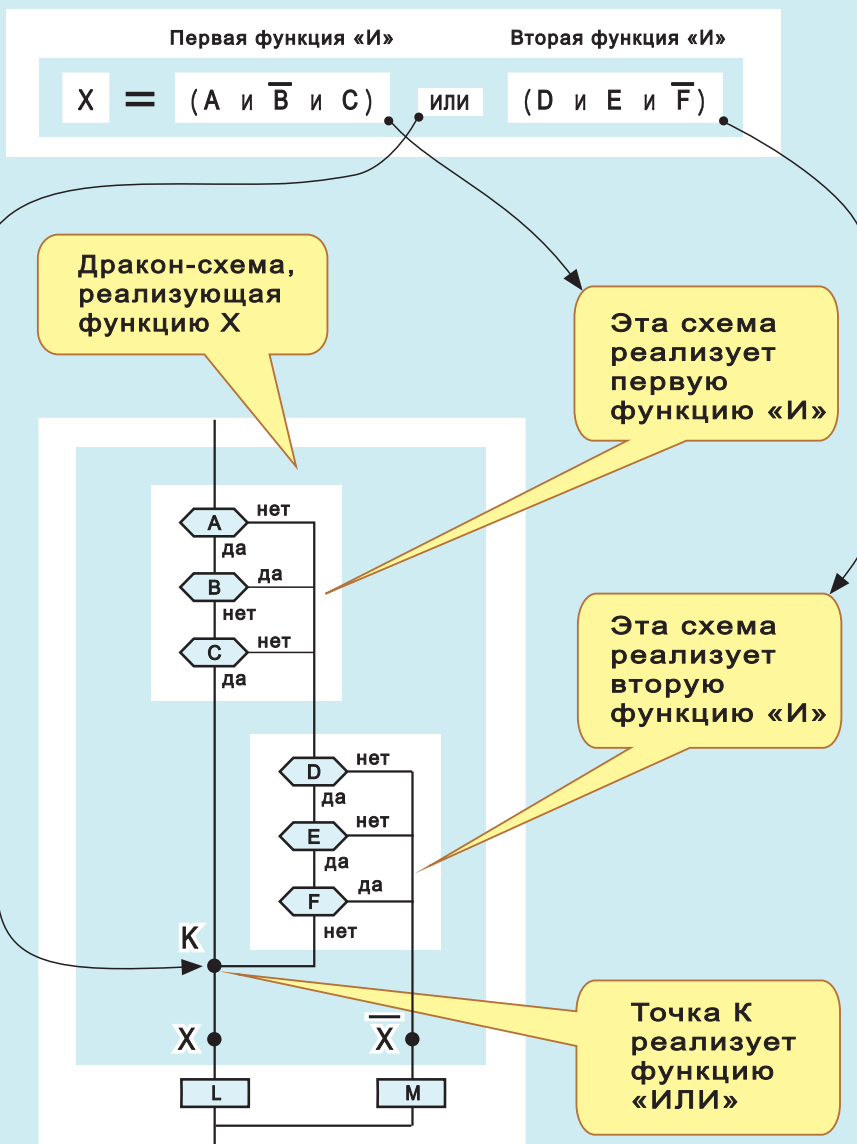


Рис. 123. Как нарисовать дракон-схему для сложной логической функции?

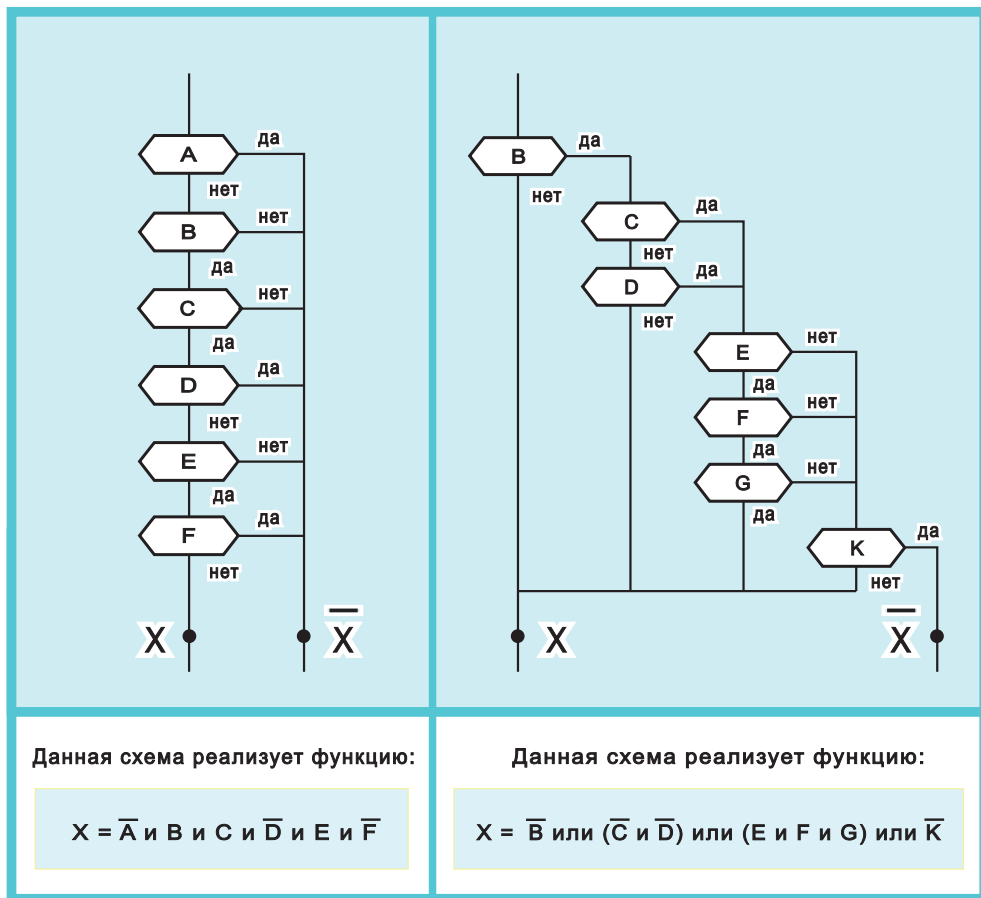
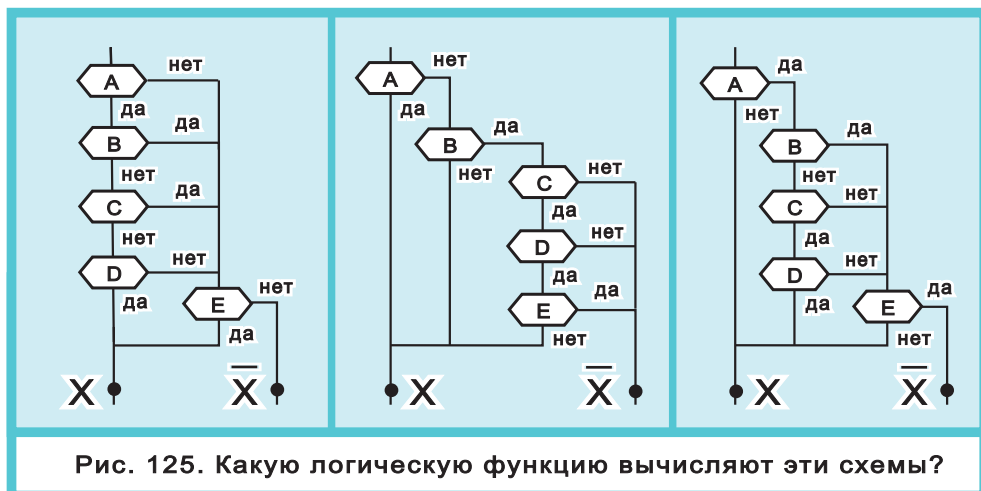


Рис. 124. Преобразование дракон-схемы в логическую функцию



НЕПРАВИЛЬНО

1. В ветке «Контроль приборов» неверно расположены иконы адрес. Нарушено правило «Порядок адресов должен соответствовать порядку веток».
2. Чтобы исправить ошибку, надо сделать рокировку в белом фрагменте. Стандартную схему ИЛИ надо заменить на нестандартную. При этом две иконы адрес («Завершение» и «Контроль электросети») поменяются местами

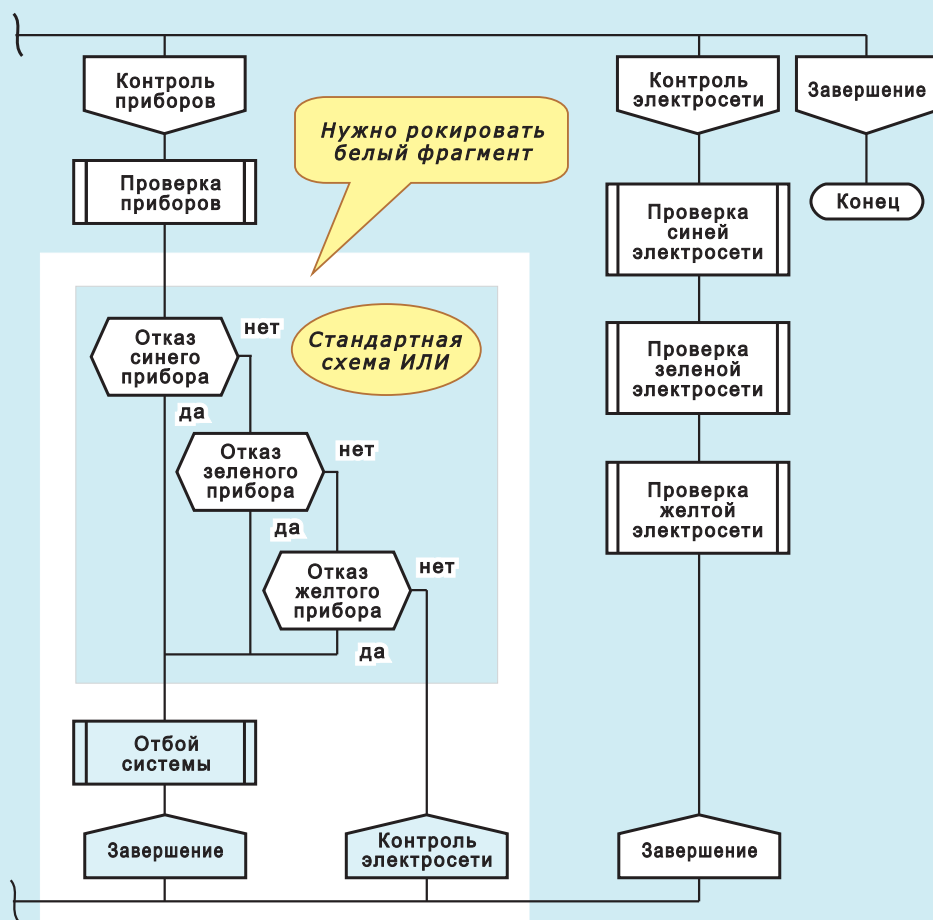


Рис. 126. Нарушено правило «Порядок адресов должен соответствовать порядку веток». Чтобы исправить ошибку, в левой ветке надо сделать рокировку. См. рис. 127

ПРАВИЛЬНО

1. В ветке «Контроль приборов» иконы адрес расположены правильно. Порядок расположения икон адрес соответствует порядку выполнения веток.
2. На рис. 126 сделана рокировка в белом фрагменте. В результате две иконы адрес на рис. 127 расположились так: «Контроль электросети» и «Завершение». В таком же порядке выполняются и две последние ветки

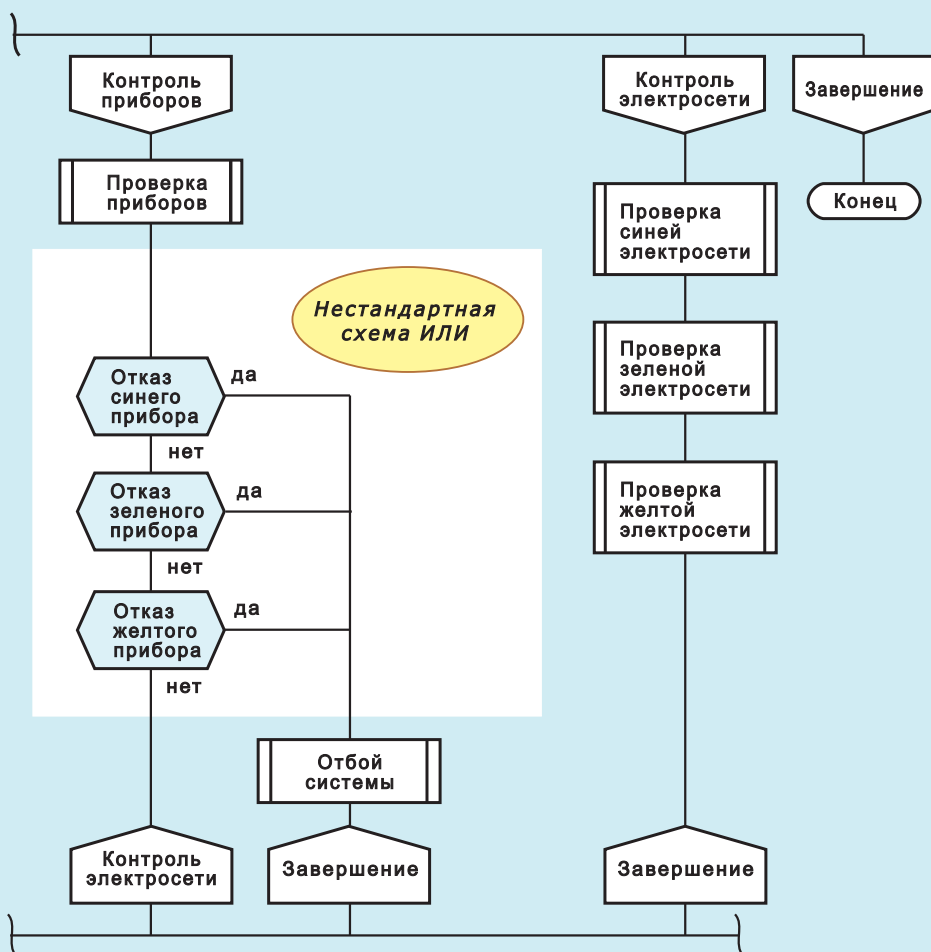


Рис. 127. Правильная схема. Все ошибки исправлены. Выполнено правило «Порядок адресов должен соответствовать порядку веток». Стандартная схема ИЛИ (на рис. 126) преобразована в нестандартную схему ИЛИ

СЛЕВА (на шампуре ветки) рисуйте икону адрес, которая указывает на ветку, которая должна выполняться РАНЬШЕ.

СПРАВА рисуйте икону адрес, указывающую на ветку, которая будет выполняться ПОЗЖЕ.

Сформулируем эту мысль иначе.

Порядок пространственного расположения адресов (слева направо) должен соответствовать порядку выполнения веток во времени (слева направо).

Ошибка состоит в том, что на рис. 126 это правило нарушено. Чтобы устранить недостаток, надо сделать рокировку в белом фрагменте.

Теперь мы можем ответить на вопрос, поставленный в заголовке параграфа. Особенность данной рокировки в том, что стандартная схема ИЛИ превращается в нестандартную. После рокировки две иконы адрес на рис. 127 расположились так, как нужно: «контроль электросети» и «завершение».

Они передают управление на две ветки:

- контроль электросети (эта ветка работает раньше);
- завершение (эта ветка работает позже).

Таким образом, замена стандартной схемы ИЛИ на нестандартную (в данном случае) дает полезный эффект, а именно, позволяет выполнить правило «Порядок расположения адресов должен соответствовать порядку выполнения веток».

§31. 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 знаки И, ИЛИ, НЕ из условных операторов.
3. Визуализация логических формул во многих практически важных случаях заметно облегчает их понимание и уменьшает вероятность ошибок.

ЧТО ТАКОЕ ЭРГОНОМИЧНЫЙ ТЕКСТ?

§1. МОЖНО ЛИ СДЕЛАТЬ ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ ЭРГОНОМИЧНЫМИ?

Одна из основных целей языка ДРАКОН – улучшение понятности алгоритмов. До сих пор мы решали эту задачу методом визуализации, превращая часть текста в эргономичный графический образ.

А как должна выглядеть другая часть текста – та, что не подлежит визуализации? Имеется в виду текст, который записывается внутри икон. Можно ли этот текст сделать более ясным и доходчивым? Более эргономичным?

Поставленный вопрос слишком обширен и сложен. Поэтому сузим тему и ограничимся частной задачей. Как следует записывать идентификаторы логических переменных и логические выражения, чтобы сделать их более понятными?

§2. ПРИМЕР ДЛЯ ИССЛЕДОВАНИЯ ЭРГОНОМИЧНОСТИ ЛОГИЧЕСКИХ ВЫРАЖЕНИЙ

Чтобы прояснить вопрос, желательно иметь под рукой какой-нибудь пример, на котором мы будем «проигрывать» различные методы улучшения эргономичности.

Предположим, нужно создать алгоритм, управляющий автомобилем-роботом, проезжающим через перекресток со светофором в условиях реального дорожного движения. Примем соглашение, что робот движется только по прямой, и выберем самый простой алгоритм управления (рис. 128).

Логический признак, разрешающий (или запрещающий) роботу ехать вперед, имеет идентификатор «Можно.ехать.через.перекресток». Будем считать, что данный признак принимает значение «да» в трех случаях:

- горит зеленый сигнал светофора (и нет помех для движения);
- желтый сигнал загорелся, когда робот уже выехал на перекресток (и нет помех для движения);
- светофор сломался – нет ни зеленого, ни желтого, ни красного сигнала (и нет помех для движения).

В остальных случаях признак имеет значение «нет», которое запрещает роботу движение через перекресток.

§3. МОДЕЛЬ ДВИЖЕНИЯ РОБОТА

Пусть $Y = \text{Можно.ехать.через.перекресток}$

Выделим на рис. 128 фрагмент, который вычисляет логическую функцию Y . То есть формирует признак «Можно.ехать.через.перекресток».

Эта часть изображена в белом прямоугольнике на рис. 129.

Можно доказать, что указанный фрагмент всегда можно заменить на одну икону «вопрос» согласно формуле на рис. 130. Эта формула позволяет преобразовать алгоритм на рис. 129 в алгоритм на рис. 131.

Алгоритм на рис. 131 имеет две особенности:

- он полностью соответствует алгоритму на рис. 128 и 129;
- он представляет названный алгоритм в сжатой форме.

Фактически рис. 131 – это наиболее простая модель движения робота.

§4. ОБОЗНАЧЕНИЯ ЛОГИЧЕСКИХ ПЕРЕМЕННЫХ

Введем обозначения, показанные на рис. 132, которым соответствуют очевидные равенства:

$Y = \text{Можно.ехать.через.перекресток}$ (1)

$A = \text{Зеленый.сигнал.светофора}$ (2)

$B = \text{Желтый.сигнал.светофора}$ (3)

$C = \text{Красный.сигнал.светофора}$ (4)

$D = \text{Робот.выехал.на.перекресток}$ (5)

$E = \text{Помехи.для.движения}$ (6)

Если принять указанные условия и обозначения, логическая функция Y задается формулой

$$Y = (A \ \& \ \neg E) \vee (B \ \& \ D \ \& \ \neg E) \vee (\neg A \ \& \ \neg B \ \& \ C \ \& \ \neg E) \quad (7)$$

В §4 и на рис. 132 для операции «НЕ» вместо верхней черты « \neg » использован знак « \neg ». Оба знака имеют одинаковый смысл и обозначают одно и то же. Далее мы будем использовать оба знака.

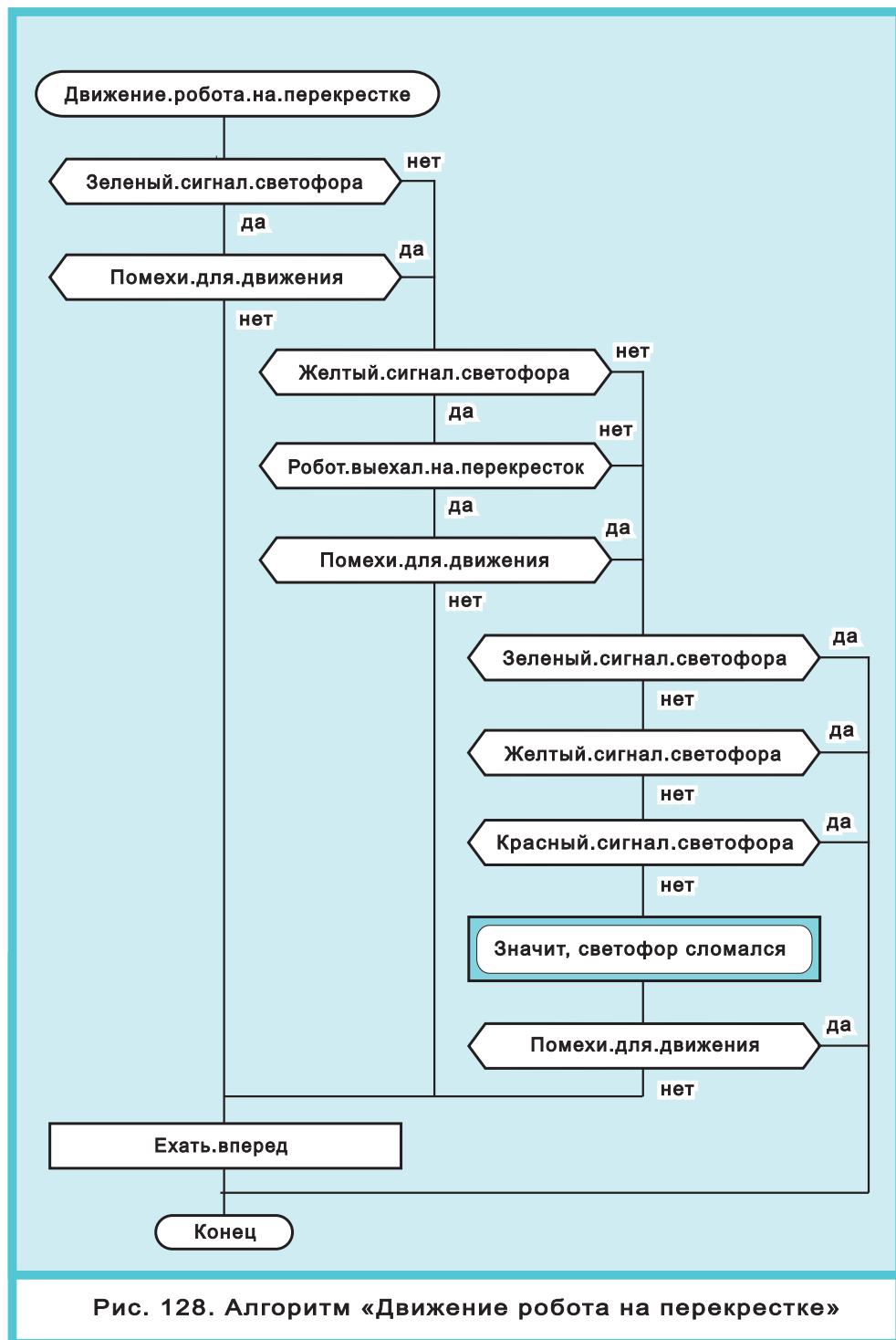


Рис. 128. Алгоритм «Движение робота на перекрестке»

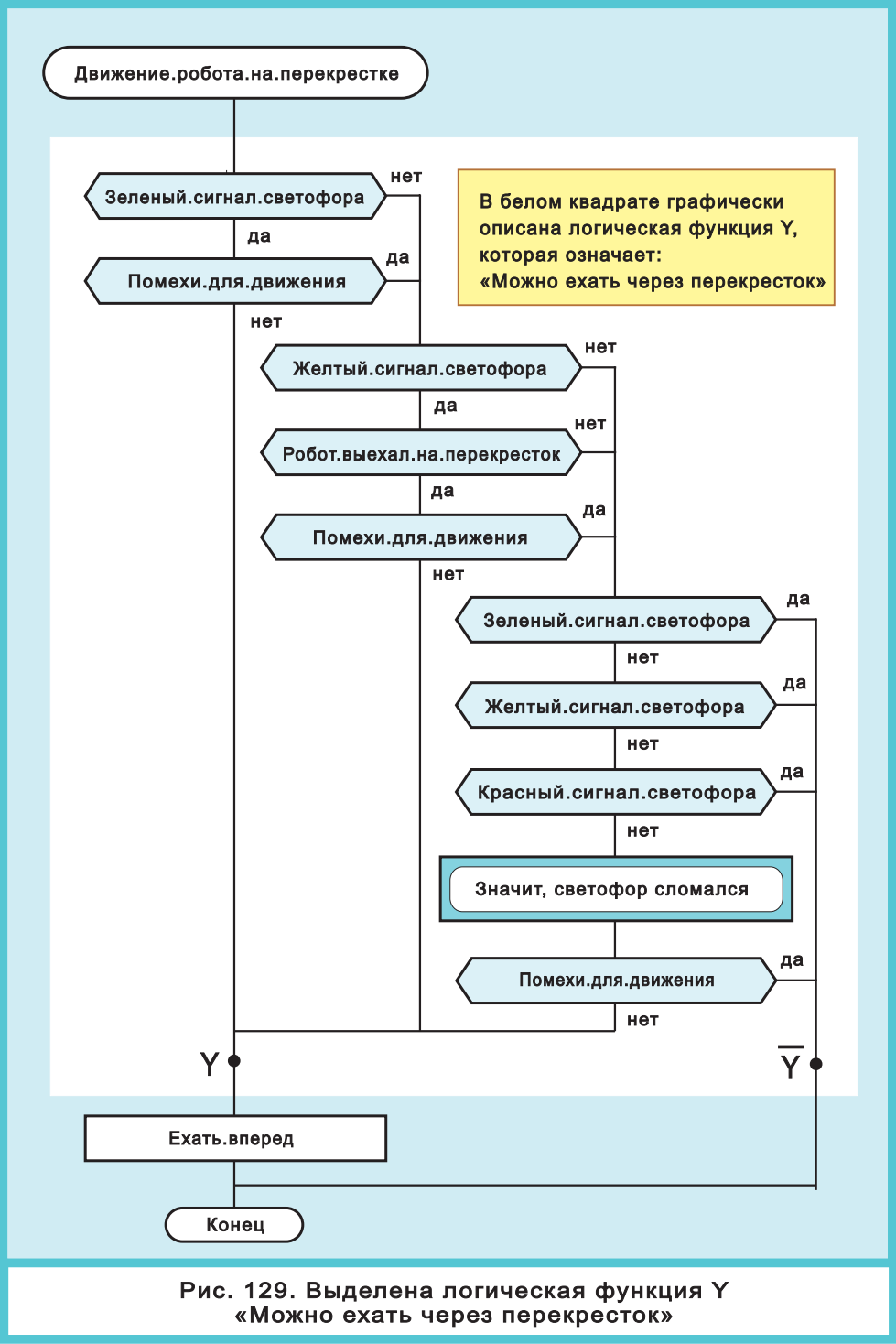


Рис. 129. Выделена логическая функция Y «Можно ехать через перекресток»

ВИЗУАЛЬНАЯ ФОРМУЛА (пример)

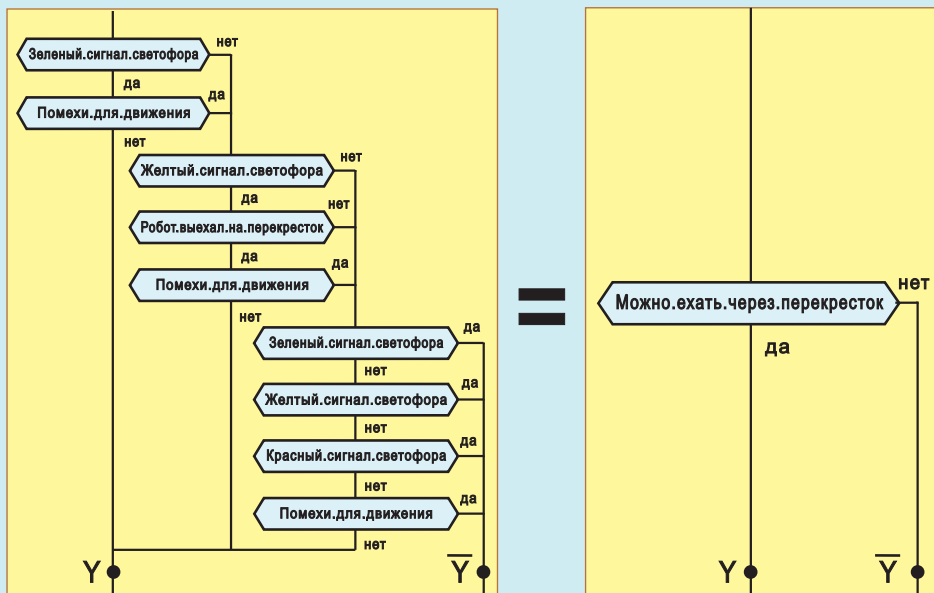


Рис. 130. Логическая функция $Y = \text{Можно.ехать.через.перекресток}$ показана дважды: в развернутом и сжатом виде

1. Логическая функция $Y = \text{Можно.ехать.через.перекресток}$ показана на рис. 130 (слева) в виде графической формулы.

2. На рис. 130 (справа) функция Y записана в виде идентификатора внутри иконы вопрос.

3. На рис. 131 (справа) эта икона является частью алгоритма. Если $\text{Можно.ехать.через.перекресток} = \text{да}$, выполняется команда «Ехать.вперед».

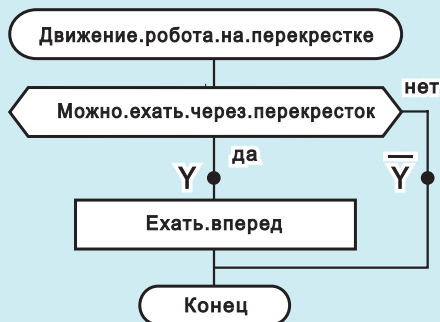


Рис. 131. Логическая функция Y «спрятана» в иконе вопрос

§5. ПЛАН ИЗУЧЕНИЯ ПРОБЛЕМЫ

Пример, представленный на рис. 128–132, позволяет приступить к делу. Ниже мы рассмотрим несколько вариантов записи логических выражений и сравним их между собой с точки зрения эргономики.

Будем считать, что движением робота управляет бортовой компьютер. Глазами робота являются пять датчиков, формирующих логические сигналы A, B, C, D, E . Эти сигналы сообщают компьютеру информацию, необходимую для управления движением робота через перекресток.

§6. ЛОГИЧЕСКОЕ ВЫРАЖЕНИЕ С АБСТРАКТНЫМИ ИДЕНТИФИКАТОРАМИ

Произведем эквивалентное преобразование алгоритма на рис. 131. Учитывая (1), заменим идентификатор «Можно.ехать.через.перекресток» буквой Y . Затем вместо Y подставим логическое выражение из формулы (7). В итоге получим алгоритм на рис. 133.

Некоторые математики, скорее всего, похвалят этот алгоритм. Они, возможно, скажут, что с математической точки зрения выражение в иконе «вопрос» является компактным, лаконичным, изящным и обозримым.

К сожалению, подобная позиция не учитывает эргономических соображений и является устаревшей.

Недостаток формулы (7) и алгоритма на рис. 133 состоит в том, что идентификаторы A, B, C, D, E не смысловые, а абстрактные. Они оставляют наши знания о предметной области за пределами алгоритмического текста.

Чем это плохо? Люди, которые прекрасно знают прикладную задачу, но не знают или забыли обозначения (1)–(6), например заказчики, постановщики задач, не могут понять эти обозначения. Они воспринимают идентификаторы A, B, C, D, E и составленные из них формулы как бессмысленный набор символов. Следовательно, эти люди лишаются возможности принять участие в проверке правильности алгоритмов. И вносят свой вклад в устранение ошибок.

Чтобы обнаружить ошибку в логическом выражении, необходимо хорошо понимать его смысл. Чтобы уяснить суть логического выражения на рис. 133, человек вынужден помнить не только смысловые понятия, но и абстрактные идентификаторы. Он должен твердо знать соответствие между ними. Это создает двойную нагрузку на память человека-алгоритмиста. Из-за этого возникают дополнительные и ничем не оправданные трудности при поиске и выявлении ошибок.

Для большинства специалистов, знающих прикладную задачу, логическое выражение на рис. 133 не дает никакой подсказки о смысле (семантике) логических переменных. Это логическое выражение фактически представляет собой загадочный набор иероглифов. Оно служит типичным примером эргономической неряшливости традиционных методов математического описания прикладных задач.

Вывод. В прикладных задачах использование абстрактных идентификаторов в логических выражениях эргономически недопустимо.

§7. ЛОГИЧЕСКОЕ ВЫРАЖЕНИЕ С КОРОТКИМИ СМЫСЛОВЫМИ ИДЕНТИФИКАТОРАМИ

Абстрактные идентификаторы использовались на первом этапе развития программирования. Сегодня в прикладных программах они встречаются гораздо реже, уступив место так называемым мнемоническим именам.

Мнемоническое имя – это короткий смысловой идентификатор, который в большинстве случаев имеет длину не более 6 или 8 символов.

Преобладание 8-символьных идентификаторов характерно для второго этапа развития языков программирования. Вот типичная рекомендация того периода:

«Не оправдано применение имен, подобных X или I , тогда как имена MAX или $NEXT$ передают смысл гораздо точнее» [1].

Сходную мысль высказывает Дени Ван Тассел:

«Имена переменных должны быть выбраны так, чтобы наилучшим образом определять те величины, которые они представляют. Например, в операторе $X = Y + Z$ имена переменных выбраны неудачно, поскольку совсем не использована мнемоника. Такая запись оператора:

$$PRICE = COST + PROFIT$$

намного лучше» [2].

Последуем совету. Продолжая наш пример с автомобилем-роботом, заменим абстрактные идентификаторы на мнемонические имена согласно таблице.

Абстрактный идентификатор	Мнемоническое имя
Y	Можнех
A	Зелсиг
B	Желсиг
C	Красиг
D	Робнапер
E	Помдвиж

ЛОГИЧЕСКАЯ ФУНКЦИЯ Y

ОПИСЫВАЕТСЯ ФОРМУЛОЙ

$$Y = (A \& \neg E) \vee (B \& D \& \neg E) \vee (\neg A \& \neg B \& \neg C \& \neg E)$$

где Y, A, B, C, D, E – логические переменные,
 Y – означает «Можно.ехать.через.перекресток»
 A – означает «Зеленый.сигнал.светофора»
 B – означает «Желтый.сигнал.светофора»
 C – означает «Красный.сигнал.светофора»
 D – означает «Робот.выехал.на.перекресток»
 E – означает «Помехи.для.движения»
 & – логическая операция «И»
 V – логическая операция «ИЛИ»
 ¬ – логическая операция «НЕ»

Рис. 132. Обозначения логических переменных

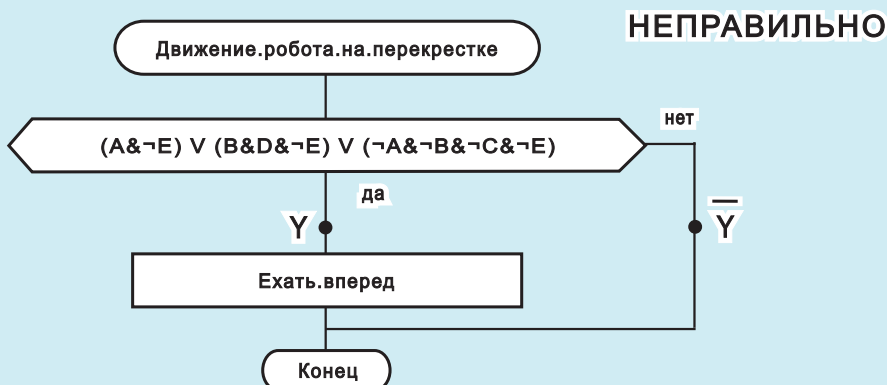


Рис. 133. Эргономически неудачная схема с абстрактными идентификаторами

На рис. 134 показан алгоритм, полученный в результате такой замены. Можно ли назвать логическое выражение на рис. 134 эргономичным?

Очевидно, что мнемонические имена лучше абстрактных. Они были придуманы с благородной целью – облегчить запоминание понятий, чтобы формальное имя создавало намек на содержательную сторону дела.

Увы! Говорить намеками – вовсе не значит говорить понятно. Причина неудачи в том, что длина идентификатора 8 символов слишком мала. Она явно недостаточна для хорошего, ясного и доходчивого описания сложных понятий. Поэтому при создании 8-символьных идентификаторов приходится экономить каждый символ и часто использовать невразумительные слова-обрубки, такие, как Зелсиг (зеленый сигнал светофора), Красиг (красный сигнал светофора) и т. д.

В самом деле, глядя на идентификатор «Робнапер» (рис. 134) мало кто догадается, что речь идет о признаке «Робот.выехал.на.перекресток».

Сравнивая логические выражения на рис. 133 и 134, можно сказать, что в последнем случае понятность алгоритма, если и увеличилась, то незначительно. Таким образом, 8-символьные смысловые идентификаторы не могут обеспечить требуемое улучшение эргономических характеристик логических выражений.

§8. ЛОГИЧЕСКОЕ ВЫРАЖЕНИЕ С ДЛИННЫМИ СМЫСЛОВЫМИ ИДЕНТИФИКАТОРАМИ

Для третьего этапа развития алгоритмических языков характерен переход к длинным смысловым идентификаторам, содержащим до 32 символов.

Увеличение длины идентификатора до 32 символов позволяет получить два важных эргономических преимущества:

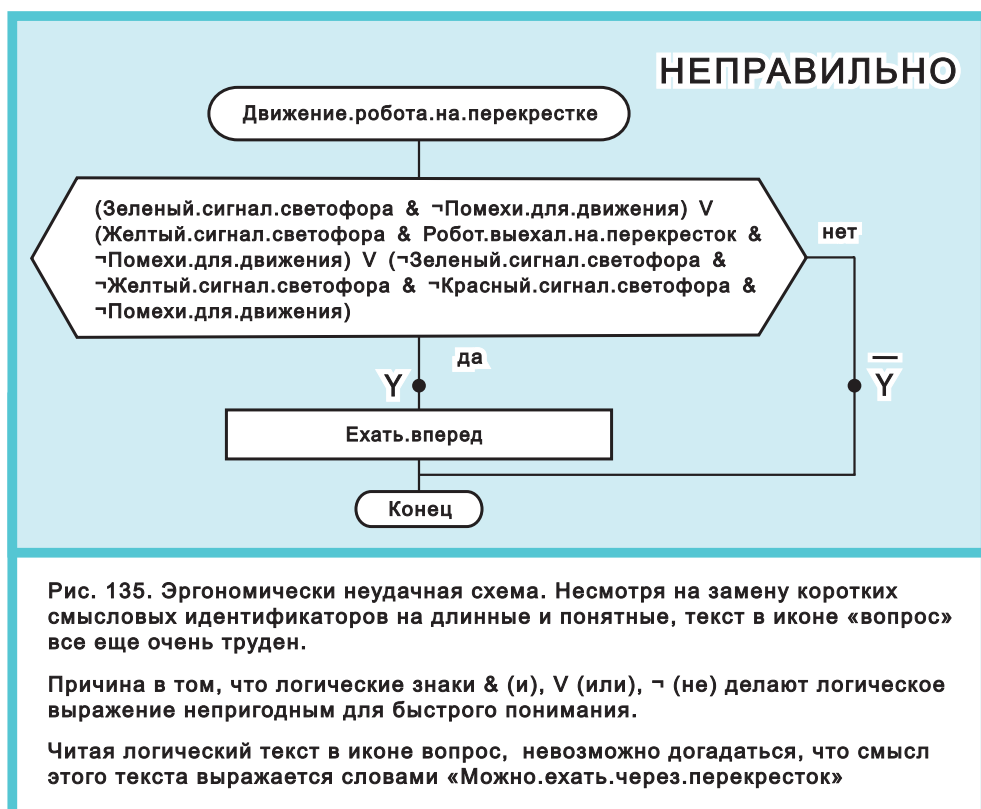
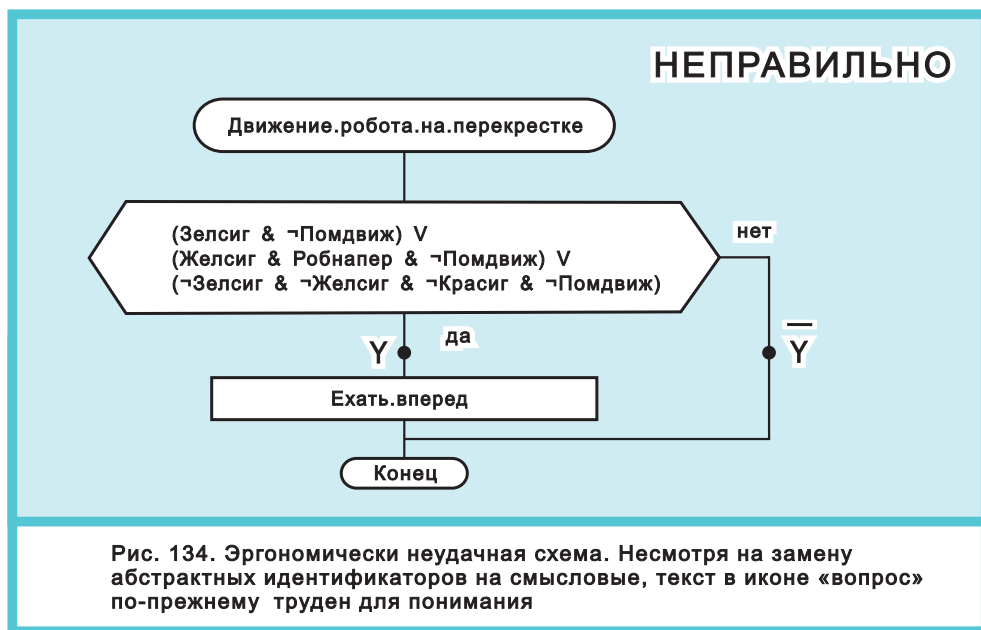
- Во многих (хотя и не во всех) случаях появляется возможность отказаться от сокращений и использовать полные слова;
- Для разграничения слов, входящих в состав идентификатора, можно ввести эргономичные разделители, например точку или нижнюю черту.

Пример логического выражения, в котором используются идентификаторы с точками-разделителями и полными (несокращенными) словами, представлен на рис. 135. Легко видеть, что такая запись более понятна, чем предыдущие примеры. Тем не менее, есть одно «но».

§9. ГЛАВНЫЙ ВОПРОС УСЛОВНОГО ОПЕРАТОРА

На рис. 131 задан вопрос «Можно.ехать.через.перекресток?», который объясняет принцип разветвления алгоритма. При ответе «да» выполняется команда «Ехать.вперед». При ответе «нет» действия отсутствуют.

Данный вопрос играет ключевую роль. Если его удалить, алгоритм становится непонятным. В связи с этим целесообразно ввести новое понятие, выявляющее суть проблемы.



Главный вопрос условного оператора – это ясный и понятный текст, записанный в иконе «вопрос» с помощью одного идентификатора.

А теперь взглянем на рис. 133–135. Нетрудно заметить, что в иконе «вопрос» записана масса любопытных подробностей, однако интересующий нас вопрос отсутствует. Он бесследно исчез.

Логические выражения на рис. 133–135 имеют важный недостаток.

В них нет главного вопроса. Нет ключа, объясняющего существование алгоритма!

Налицо парадокс. Логические выражения не дают явной информации о том, *на какой именно вопрос* мы отвечаем «да» или «нет». Более того, они не позволяют читателю легко и быстро восстановить формулировку главного вопроса. И даже не стимулируют у него стремления к получению подобной информации.

Не будет преувеличением сказать, что перечисленные логические выражения затуманивают суть дела. Потому что отсутствие главного вопроса ничем нельзя компенсировать. В итоге алгоритмы на рис. 133–135 оказываются непонятными, эргономически неприемлемыми.

Вывод 1. Использование эргономически правильных длинных смысловых идентификаторов является необходимым, но не достаточным условием для построения эргономичного логического текста.

Вывод 2. Вторым условием является использование главного вопроса условного оператора.

§10. ВИЗУАЛЬНЫЕ ПОМЕХИ ЗАСОРЯЮТ ЗРИТЕЛЬНУЮ СЦЕНУ НЕНУЖНОЙ ИНФОРМАЦИЕЙ

В традиционных языках для значений логических переменных используют слова *TRUE* и *FALSE*, *ИСТИНА* и *ЛОЖЬ*, 1 и 0. Однако логико-эргономические исследования показывают, что указанные обозначения являются избыточными. Они могут быть безболезненно и с пользой для дела исключены из алгоритмического текста.

Стремление «уничтожить» лишние обозначения объясняется эргономическими причинами. Известно, что все ненужные записи являются визуальными помехами, которые засоряют текст алгоритма и путают читателя.

Язык ДРАКОН позволяет решить задачу двумя способами:

- с помощью рамочного логического выражения;
- с помощью визуального логического выражения.

§11. УСТРАНЕНИЕ ВИЗУАЛЬНЫХ ПОМЕХ С ПОМОЩЬЮ РАМОЧНОГО ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ

В общем случае *идентификатор главного вопроса* может оказаться неопределенным. Чтобы устранить этот недочет, надо заранее присвоить ему нужное значение. Для этого можно использовать, например, процедуру под названием «Формирование.признака» (рис. 136).

Рассмотрим случай, когда процедура строится с помощью рамочного логического выражения (рис. 137).

В этом случае применяют икону «действие», внутри которой записывают оператор присваивания. Графический оператор на рис. 137 означает, что идентификатору «Можно.ехать.через.перекресток» присваивается некоторое значение. Какое именно?

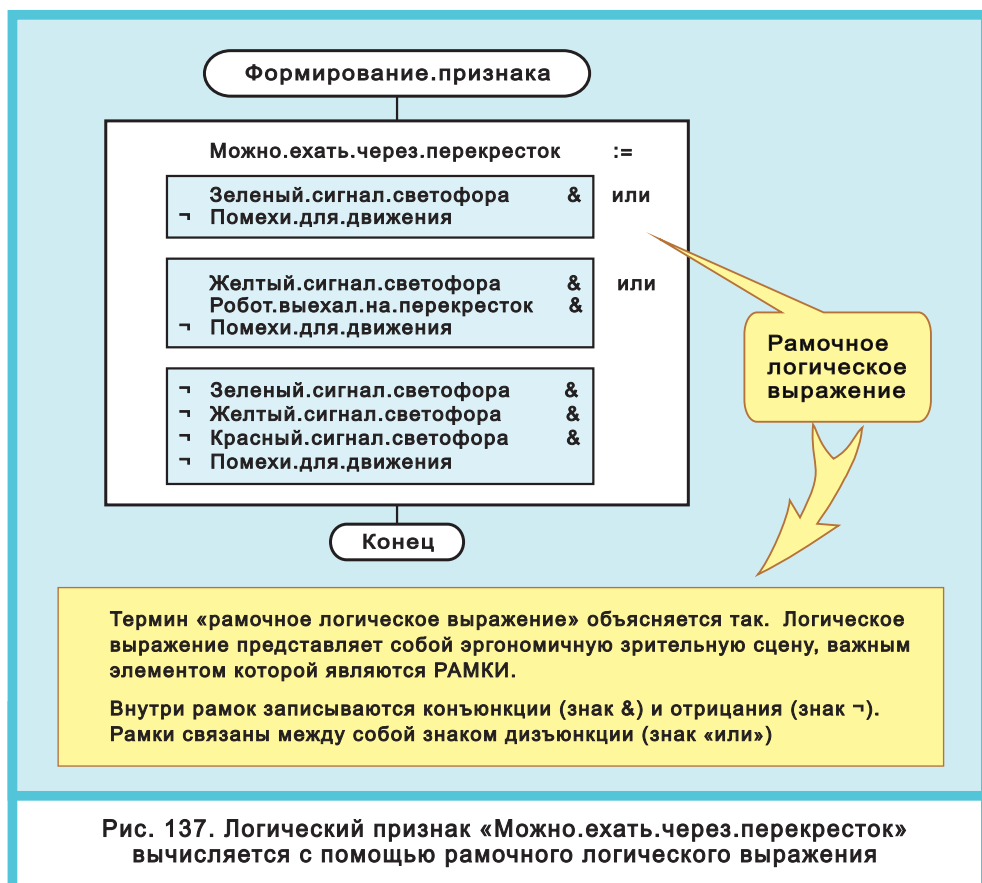
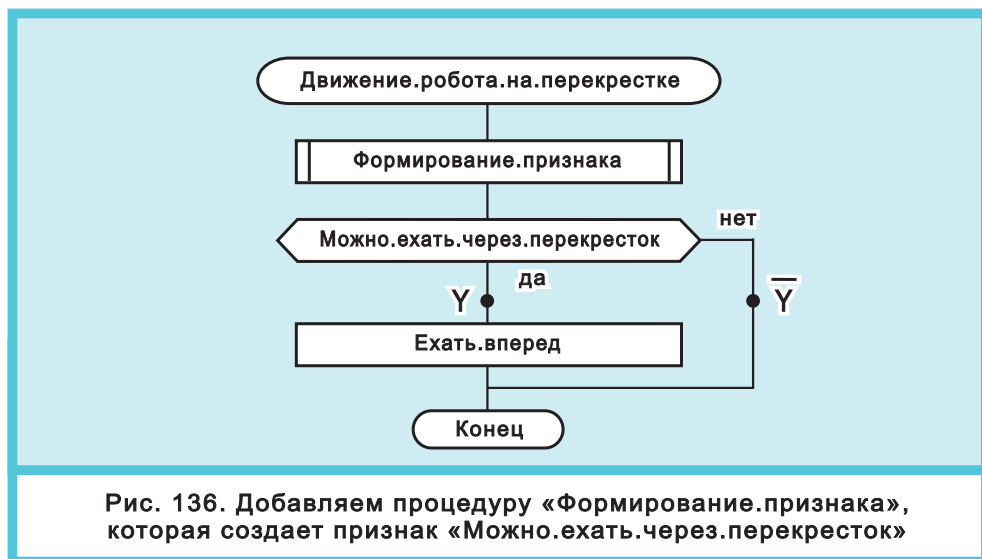
Ответ таков. Нужно вычислить *рамочное логическое выражение*, записанное в трех рамках, соединенных знаками ИЛИ. Результатом вычисления будет «1» или «0».

Таким образом, цель достигнута, хотя обозначения «1» и «0» в тексте алгоритма отсутствуют (рис. 137).

§12. ПРАВИЛА ЗАПИСИ РАМОЧНЫХ ЛОГИЧЕСКИХ ВЫРАЖЕНИЙ

Увеличение длины идентификаторов приводит к тому, что традиционная горизонтальная запись логических выражений становится невозможной. В связи с этим применяется вертикальная запись, пример которой показан на рис. 137. Вертикальный логический текст на языке ДРАКОН пишут в соответствии со следующими правилами.

- В иконе «действие» размещают один оператор присваивания.
- В верхней строке пишут идентификатор логической переменной и знак присваивания.
- Ниже пишут логическое выражение, причем каждая конъюнкция заключается в прямоугольную рамку.
- Для операций И, ИЛИ, НЕ используют обозначения &, ИЛИ, 7 соответственно.
- Используют идентификаторы длиной до 32 символов.
- Первые символы всех идентификаторов располагают на одной вертикали.
- Знак логического отрицания 7 пишут слева от идентификатора внутри рамки.
- Все знаки отрицания (если они есть) помещают на одной вертикали.



- Знаки конъюнкции & записывают справа от идентификатора внутри рамки.
- Все знаки & пишут на одной вертикали.
- Вертикальные линии рамок располагают на одной вертикали.
- Знаки присваивания := и знаки ИЛИ помещают на одной вертикали.

§13. ЗРИТЕЛЬНАЯ СЦЕНА РАМОЧНОГО ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ

Взглянем на икону «действие» как на зрительную сцену (рис. 137). В ней можно увидеть четыре столбца, разделенные воображаемыми вертикальными линиями. Назовем столбцы *зрительными зонами*.

Каждая зона имеет строго определенное назначение.

- В 1-й зоне пишут знаки логического отрицания ¬ (если они есть).
- Во 2-й зоне пишут идентификаторы.
- В 3-й зоне пишут знаки конъюнкции &.
- В 4-й зоне пишут знак присваивания := и знаки ИЛИ.

В итоге зрительная сцена приобретает четкую регулярную структуру. Она разделена на 4 зоны, в каждой из которых должна находиться заранее определенная информация.

Благодаря горизонтальным рамкам и вертикальным зрительным зонам зрительная сцена становится не хаотичной, а предсказуемой, упорядоченной и удобной для чтения.

§14. УСТРАНЕНИЕ ВИЗУАЛЬНЫХ ПОМЕХ С ПОМОЩЬЮ ВИЗУАЛЬНОГО ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ

Рассмотрим случай, когда процедура «Формирование.признака» строится с помощью визуального логического выражения (рис. 138).

В этом случае применяют икону «полка» (рис. 17, икона И10).

На верхнем этаже полки пишут зарезервированное предложение «Установить признак» или «Снять признак». На нижнем этаже указывают идентификатор признака «Можно.ехать.через.перекресток» (рис. 138).

Рассмотрим еще два примера оператора «полка»:



Левый оператор говорит, что логической переменной «Норма.насоса» присваивается значение 1. Правый оператор означает, что переменной «Норма.насоса» присваивается значение 0.

Легко видеть, что в этих операторах (как и в операторах на рис. 138) используется та же хитрость, что и на рис. 137. Логической переменной присваивается значение 1 или 0, хотя обозначения 1 и 0 в тексте алгоритма нигде не встречаются!

§15. ОБСУЖДЕНИЕ

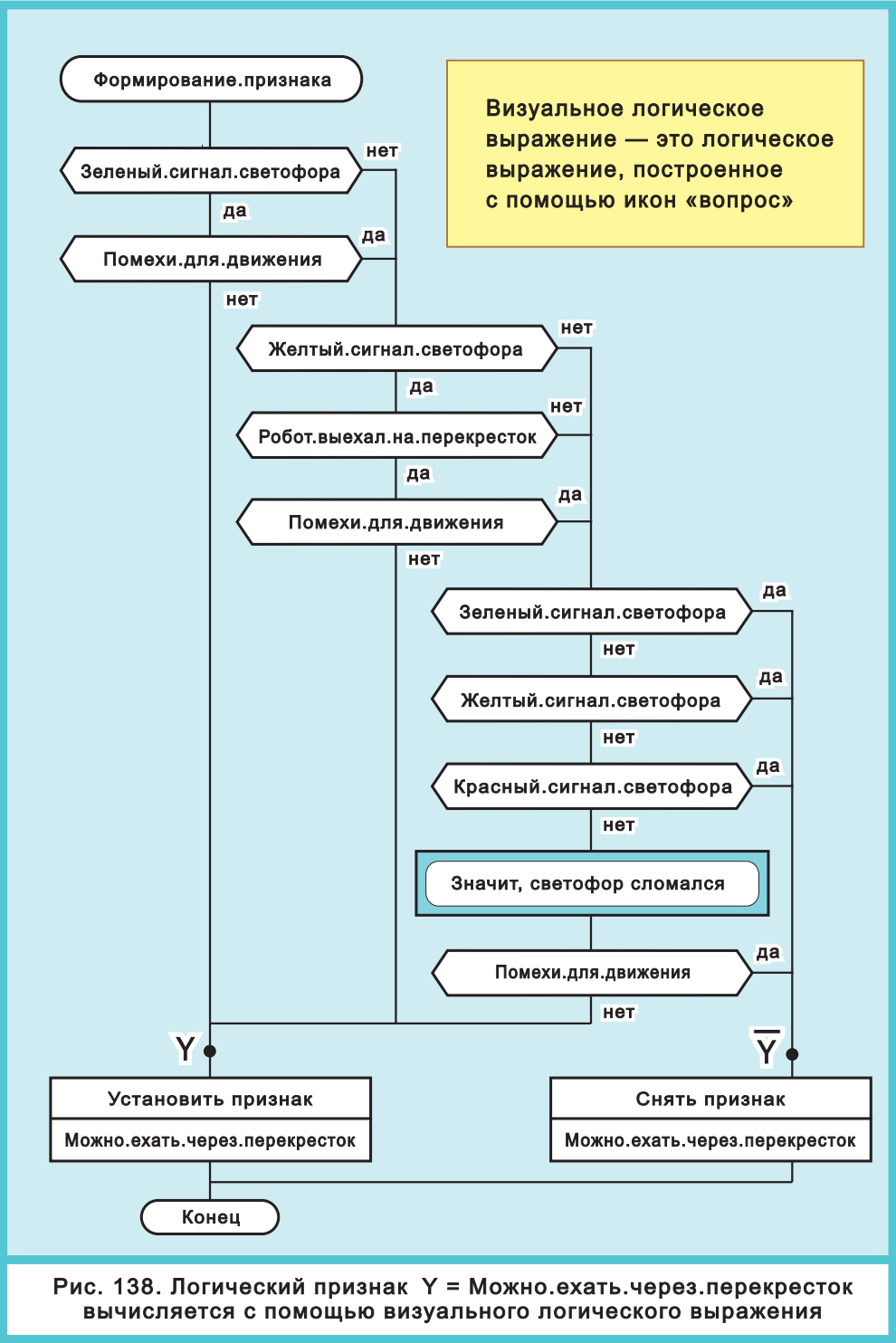
Ранее мы пришли к выводу, что алгоритм на рис. 135 является эргономически неудачным. Каким образом можно его исправить? Вопрос отнюдь не прост. По-видимому, в разных ситуациях он может приводить к разным ответам. В связи с этим изложенные ниже соображения и советы имеют не обязательный, а всего лишь рекомендательный характер. Их нужно рассматривать как один из возможных способов решения проблемы.

- В иконе «вопрос» не следует записывать логическое выражение, в особенности сложное. Вместо него рекомендуется поместить единственный идентификатор, содержащий ясную и четкую словесную формулировку главного вопроса.
- В общем случае указанный идентификатор может оказаться неопределенным. Чтобы исключить эту неприятность, необходимо заблаговременно присвоить ему нужное значение. Для этого существуют два метода: рамочный и визуальный.
- В *рамочном методе* используется рамочное логическое выражение, записанное в иконе «действие». Производится вычисление рамочного выражения. Результат присваивается идентификатору главного вопроса (рис. 137).
- В *визуальном методе* применяется визуальное логическое выражение и два оператора «Установить признак» и «Снять признак», записанные в иконах «полка» (рис. 138).

В отличие от рамочного при визуальном методе вычисление логического выражения как таковое отсутствует. Визуальное выражение разветвляет процесс и приводит его в одну из двух точек (Y или \bar{Y} на рис. 138). В первой точке выполняется оператор «Установить признак», во второй – «Снять признак». Алгоритмы на рис. 137 и 138 эквивалентны.

Инструментальные программы языка ДРАКОН должны обеспечить автоматический перевод рамочного алгоритма (рис. 137) в визуальный (рис. 138) и наоборот. Предоставление такой услуги пользователю создает для него дополнительный комфорт.

Пользователь получает возможность сравнить две формы представления логических знаний и выбрать ту, которая ему больше по душе.



Поскольку вкусы автора алгоритма и его читателей могут отличаться, каждый из них может получить листинг (чертеж) алгоритма в том виде, который лично ему больше нравится. В итоге каждый реализует свое право на индивидуальное предпочтение той или иной формы представления знаний.

§16. ЕЩЕ РАЗ О ВИЗУАЛЬНЫХ ПОМЕХАХ

Специальные обозначения для значений логических переменных как принадлежность алгоритмического языка – это анахронизм, который следует исключить из языка как совершенно не нужное и даже вредное «архитектурное излишество».

Чтобы оправдать этот вывод, сравним два выражения в условном операторе (10) и (11):

(Норма 1 = 1) & (Норма 2 = 1) & (Авария = 0), то... (10)

Если Норма 1 & Норма 2 & Авария, то.... (11)

Формула (10) читается так:

Если признак «Норма 1» равен единице и признак «Норма 2» равен единице и признак «Авария» равен нулю, то... (10a)

Формула (11) читается так:

Если есть признак «Норма 1» и есть признак «Норма 2» и нет признака «Авария», то... (11a)

Фраза (11a) намного понятнее. По своему лексическому строю, эта фраза соответствует обычным речевым оборотам, которыми пользуются специалисты предметной области, не являющиеся программистами. Она точно отражает суть дела и доступна всем работникам.

В отличие от нее фраза (10a) содержит искусственные и нарочитые вкрапления «равен единице» и «равен нулю», появление которых неоправданно удлинняет текст и затрудняет восприятие. В итоге предложение становится непонятным для всех, кроме программистов.

§17. ОСОБЕННОСТИ РАБОТЫ С ДЛИННЫМИ (ЭРГОНОМИЧНЫМИ) ИДЕНТИФИКАТОРАМИ

Предположим, что желательная длина формального смыслового идентификатора составляет примерно 32 символа.

Желательно, чтобы конкретные идентификаторы в зависимости от сложности понятия имели длину не менее 25 и не более 32 символов.

Чтобы исключить ошибки при ручном вводе столь длинных идентификаторов в компьютер, целесообразно ввести запрет повторного ввода. Это значит, что идентификатор вводится в систему только один раз и запоминается в базе данных.

При необходимости повторного ввода осуществляется копирование из базы данных. Такой способ требует наличия специальных инструментальных средств, но гарантирует идентичность всех копий одного и того же идентификатора.

Предположение об оптимальности 32-символьных идентификаторов согласуется с анализом истории развития алгоритмических языков, который обнаруживает отчетливую тенденцию:

- переход от абстрактных кодов и имен к 6- или 8-символьным мнемоническим именам;
- затем – переход к 32-символьным смысловым идентификаторам.

Вместе с тем многие специалисты, следуя устоявшимся привычкам, «застряли» на этапе 8-символьных имен. Суть в том, что опыт использования новых возможностей, связанных с появлением 32-символьных имен пока еще относительно невелик.

Между тем, эргономические перспективы, открывающиеся с увеличением длины имен до 32 символов, обещают существенно изменить наши прежние представления и привычки. Благодаря этому замечательному нововведению язык формальных идентификаторов по своей доходчивости значительно приближается к естественному человеческому языку, что отчетливо видно на рис. 137 и 138.

§18. ТЕОРЕМА

Материалы этой главы показывают, что справедлива

Теорема. Если некоторый фрагмент дракон-схемы имеет один вход, два выхода и содержит только иконы «вопрос» (две и более), причем внутри каждой иконы записан один идентификатор, этот фрагмент всегда можно заменить на одну икону вопрос, внутри которой записан один идентификатор.

Иллюстрацией являются рис. 129–131.

§19. ОПЕРАТОР «ПОЛКА»

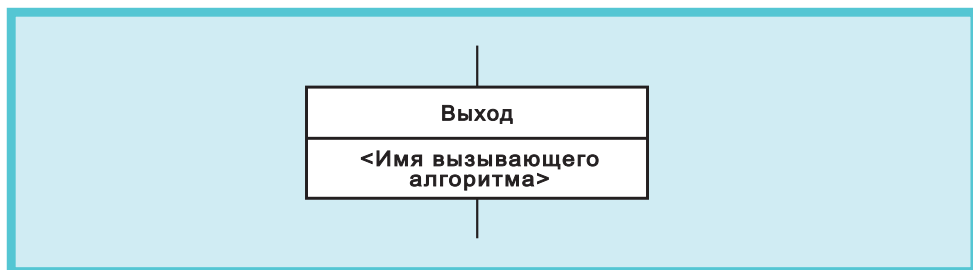
Оператор «полка» может выполнять разные функции.

Функция 1. На верхнем этаже полки пишут ключевое предложение «Установить признак» или «Снять признак». На нижнем этаже указывают идентификатор признака. Этот случай мы уже описали (см. примеры в §14 и рис. 138).

Функция 2. Предположим, нужно срочно выйти из алгоритма. Причем, не из данного алгоритма (из процедуры), а из вызывающего алгоритма. Или даже из алгоритма более высокого уровня.

Как это сделать?

Ответ дает икона полка. На верхнем этаже пишем ключевое слово «Выход». На нижнем – имя вызывающего (или более высокого алгоритма).



В результате управление передается на один или несколько уровней вверх.

Для решения задачи на дракон-схеме добавляются икона полки, икона адрес «завершение» и икона имя ветки «Завершение» (рис. 139).

Таким образом, оператор полки обеспечивает:

- прекращение работы данного алгоритма (процедуры);
- немедленный выход из вызывающего алгоритма или алгоритма более высокого уровня.

При этом надо четко различать:

- фактическую работу алгоритма;
- эргономичное изображение дракон-схемы.

Фактически полка с надписью «Выход» играет роль конца работы. Иными словами (см. рис. 139), маршрут, доходя до полки, ОБРЫВАЕТСЯ. Происходит выход из алгоритма, но не через икону конец, а через икону полки. Полка играет роль конца.

Но с *эргономической* точки зрения, такие «обрывы» нежелательны. Потому что у дракон-схемы будет нарисован не один конец, а несколько.

Фактически алгоритм может иметь несколько концов. Неприятность в том, что глядя на схему, человеку трудно воспринимать алгоритм, имеющий несколько концов. Такой алгоритм похож на елку, увешанную «концами», как елочными игрушками. Подобная зрительная сцена распыляет внимание и мешает сосредоточиться на главном.

Поэтому, с эргономической точки зрения, желательно создать видимость того, что у схемы всего один конец. Это нетрудно сделать. Рис. 139 наглядно показывает, что бегунок, двигаясь к концу алгоритма, пробегает через иконы:

- икона полка с надписью «Выход»;
- икона адрес «завершение»;
- икона имя ветки «завершение»;
- икона конец.

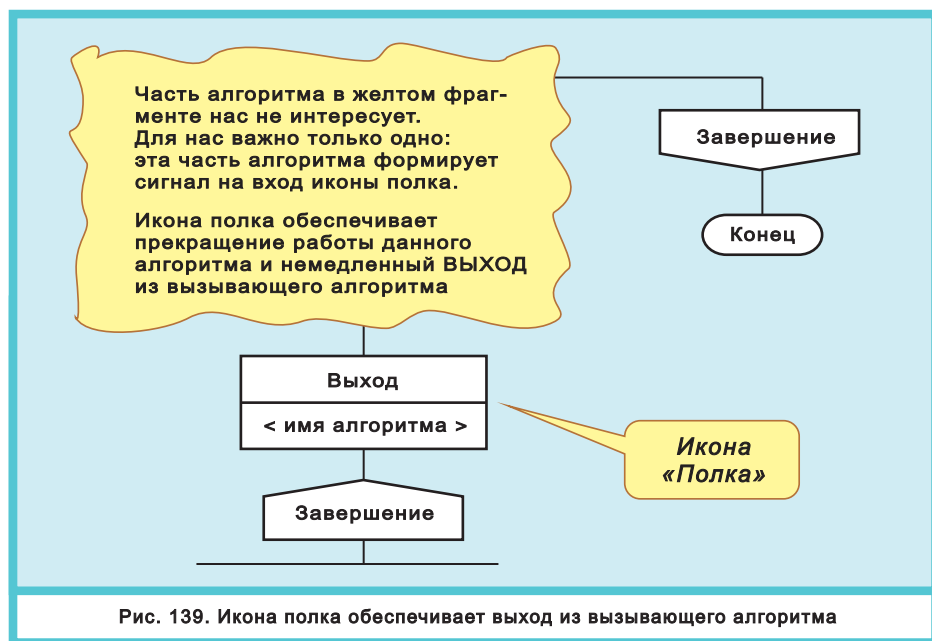
Возможно, читатель возразит. Дескать, такая схема не отвечает фактическому положению дел и дезориентирует.

С этим возражением нельзя согласиться. На рисунке 139 показано все, что нужно, для исчерпывающего понимания алгоритма. Кроме того, соблюдается эргономическое

Правило. Дракон-схема имеет только один конец.

Подытожим сказанное. Изложим функции полки в общем виде. На верхнем этаже полки пишут ключевое слово (нужно указать перечень ключевых слов и их семантику).

На нижнем этаже пишут значение, соответствующее ключевому слову.



§20. ВЫВОДЫ

1. В этой главе предложен двухэтапный метод эргономизации логических выражений.
2. На первом этапе производится разделение логических записей на две части, из которых одна подлежит визуализации, а другая сохраняется в текстовом виде.

3. На втором этапе производится эргономизация обеих частей: визуальной и текстовой.
4. Эргономизация текстовой части включает, в частности, следующие приемы:
 - оптимизацию длины и правил записи идентификаторов;
 - выбор альтернативы: логическое выражение или идентификатор главного вопроса;
 - исключение обозначений для значений логических переменных;
 - сравнительный анализ визуальной и рамочной форм записи и выбор одной из них.

АЛГОРИТМЫ РЕАЛЬНОГО ВРЕМЕНИ

§1. ВВЕДЕНИЕ

Можно ли в алгоритмах изображать время? Как это сделать? В данной главе изложен ответ на эти вопросы.

Давайте посмотрим, как работает светофор. Самый обычный светофор, который стоит на перекрестке и регулирует уличное движение. Алгоритм управления светофором показан на рис. 140.

Шапка алгоритма представлена на рис. 141. Как известно, шапка позволяет получить ответ на три «царских» (наиболее важных) вопроса:

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

Вот ответы для рис. 141.

- Как называется задача? (*Читаем заголовок алгоритма*).
Управление светофором.
- Из скольких частей она состоит? (*Считаем иконы «имя ветки»*).
Из трех.
- Как называется каждая часть? (*Читаем текст в иконах «имя ветки»*).
 1. Управление зеленым светом.
 2. Управление красным светом.
 3. Ночной режим.

На рис. 140 первая ветка начинается с команды ВКЛЮЧИТЬ ЗЕЛЕНЫЙ (имеется в виду «Включить зеленый сигнал светофора»).

Вторая команда – икона «пауза». Она изображается перевернутой трапецией. Команда «пауза» отсчитывает время, записанное внутри иконы. Когда отсчет времени закончен, запускается следующая (после паузы) команда.

Рассмотрим три команды, записанные в первой ветке:

- Включить зеленый.
- Пауза 2 минуты.
- Выключить зеленый.

Что означают эти команды? Они говорят, что зеленый свет будет гореть ровно 2 минуты, а затем погаснет.

Рассмотрим следующие три команды:

- Включить желтый.
- Пауза 10 секунд.
- Выключить желтый.

Смысл этой тройки команд очевиден. Желтый свет будет гореть 10 секунд, затем погаснет.

Вторая ветка называется «Управление красным светом».

Прочитаем три команды во второй ветке:

- Включить красный.
- Пауза 2 минуты.
- Выключить красный.

Это значит: красный свет будет гореть 2 минуты, а потом погаснет.

Читаем следующие команды:

- Включить желтый.
- Пауза 10 секунд.
- Выключить желтый.

Смысл ясен: желтый свет будет гореть 10 секунд, после чего погаснет.

Обобщим сказанное и опишем последовательность смены сигналов светофора:

- зеленый горит 2 минуты;
- желтый горит 10 секунд;
- красный горит 2 минуты;
- желтый горит 10 секунд.

После этого последовательность все время повторяется.

На рис. 140 в иконах «пауза» использована избыточная запись:

- Пауза 2 минуты.
- Пауза 10 секунд.

Это можно и нужно записать короче. Слово «Пауза» опускают. Вместо «10 секунд» пишут «10с». И т. д.

Алгоритм на рис. 140 работает в двух режимах, которые постоянно чередуются:

- дневной режим;
- ночной режим.

Ночной режим описан в третьей ветке. Ночью красный и зеленый сигналы светофора отключаются. Вместо них всю ночь мерцает желтый мигающий.

§2. БЕСКОНЕЧНЫЕ АЛГОРИТМЫ

Алгоритмы, которые мы рассматривали во всех предыдущих главах, обязательно имели конец. Проще говоря, они непременно заканчивались, то есть прекращали работу с помощью иконы «конец».

Однако на рис. 140 икона «конец» отсутствует. Это значит, что в данном случае мы имеем дело с бесконечным алгоритмом.

Бесконечный
алгоритм

- Это алгоритм, который работает все время, круглосуточно.
- В таком алгоритме специально предусмотрен бесконечный цикл. Выход из бесконечного цикла не предусмотрен.

Разумеется, прекращение работы алгоритма возможно. Оно происходит в результате внешней причины. Например, при выключении питания, при поломке оборудования и т. д.

Тем не менее, мы воочию убедились, что алгоритм, управляющий светофором, не имеет конца. Он работает круглосуточно. Потому что дорожное движение – бесконечный процесс. Который, как и сама жизнь, никогда не останавливается.

§3. СПИСОК ОПЕРАТОРОВ РЕАЛЬНОГО ВРЕМЕНИ

В языке ДРАКОН имеется пять икон реального времени (рис. 17, 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. 18 макроикона 7). Икона «синхронизатор» служит для образования тринадцати составных операторов (рис. 18, макроиконы 8–20).

Назначение операторов поясним, как всегда, на примерах.

§4. ОПЕРАТОРЫ ВВОДА-ВЫВОДА

В языке ДРАКОН предусмотрены два визуальных оператора ввода-вывода: «вывод» (рис. 17, икона И14) и «ввод» (рис. 17, икона И15). Они не от-

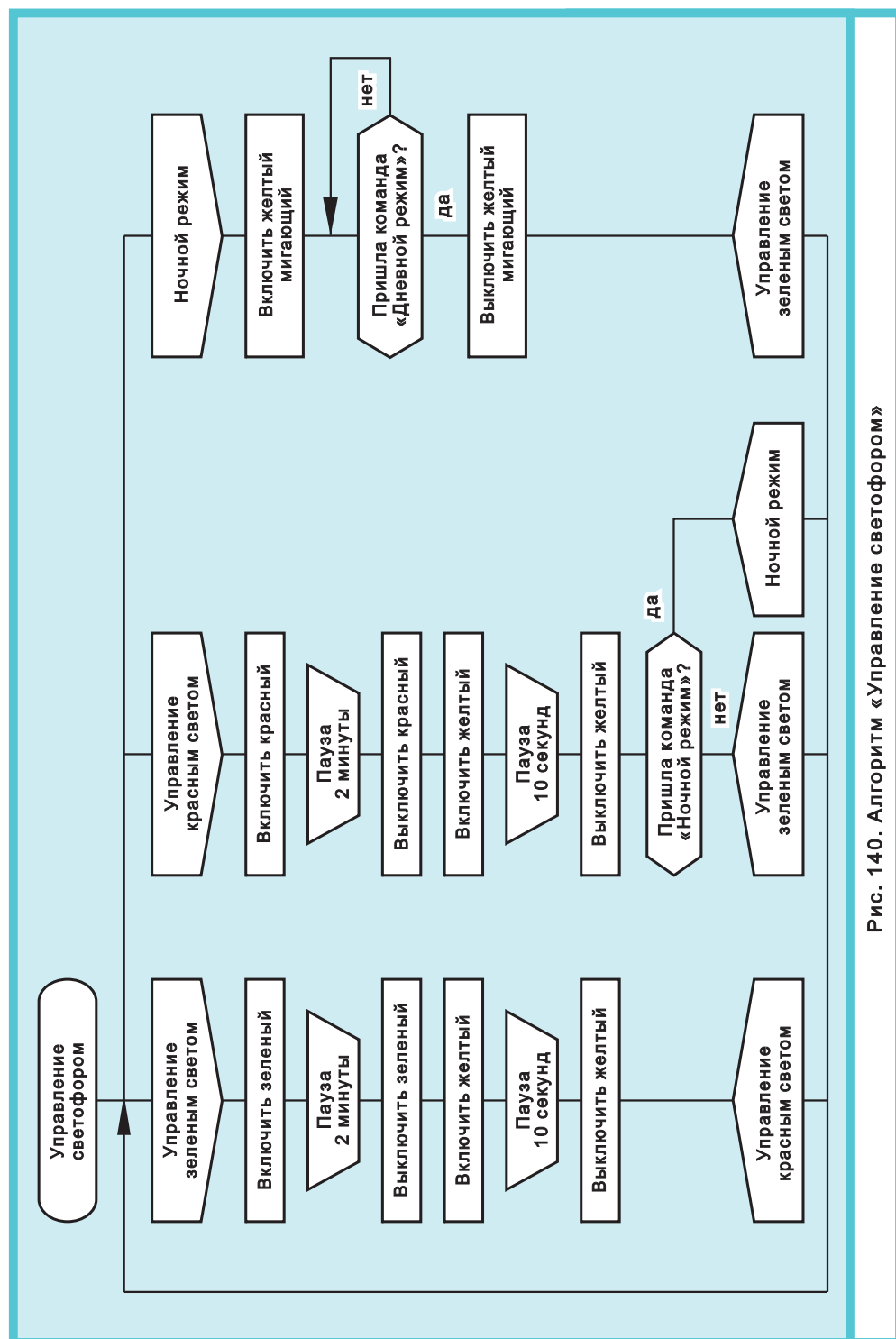


Рис. 140. Алгоритм «Управление светофором»

носятся к операторам реального времени и рассматриваются здесь только потому, что встречаются в ближайшем примере.

На рис. 17 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. 142, 143).

§5. 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 реальным временем. Например, при заправке топливом космических 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.

Соответствующий алгоритм представлен на рис. 142. Задержка выдачи команд реализуется с помощью иконы «пауза». Внутри последней указывается время необходимой задержки. Например, 2мин (2 минуты), 45с (45 секунд) и т. д.

Если говорить более точно, верхний оператор «пауза» на рис. 142 работает так. После выдачи команды ОТКРЫТЬ.ТРУБОПРОВОД в управляющем компьютере запускается программный счетчик времени на 2 минуты. По истечении этого времени компьютер выдает в линию связи команды ВКЛЮЧИТЬ.НАСОС и ОТКРЫТЬ.ЗАСЛОНКУ.

§6. 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

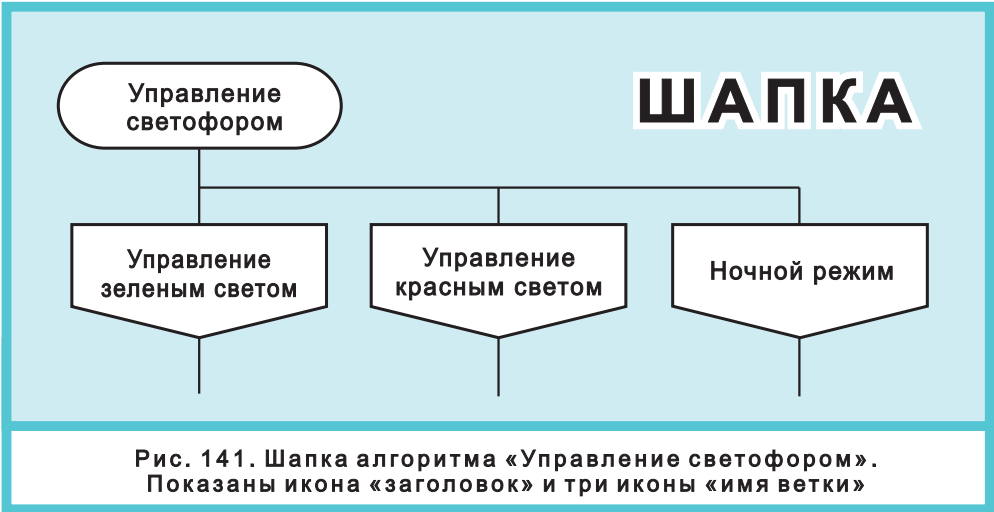


Рис. 142. Пример использования оператора «пауза»

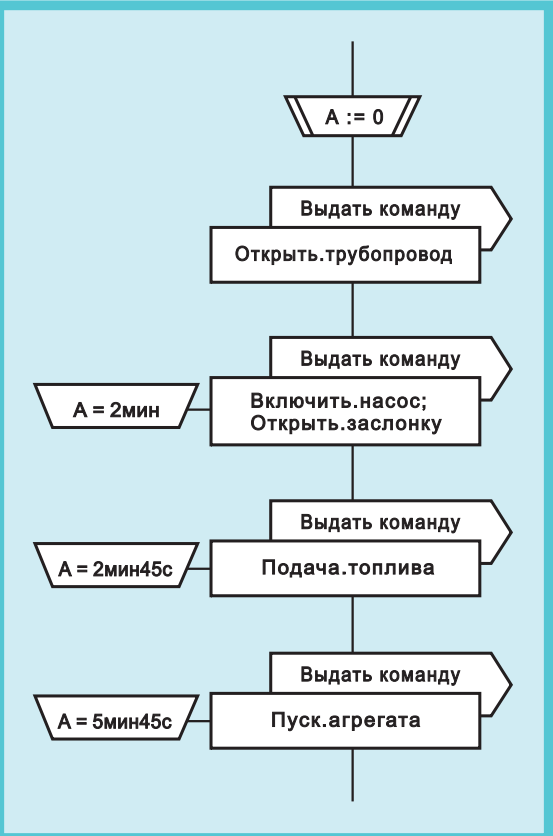


Рис. 143. Пример использования операторов «пуск таймера» и «синхронизатор»

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», то есть обнулить и запустить 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.

Описанный алгоритм изображен на рис. 143. В нем используются операторы «пуск таймера» и «синхронизатор», совместная работа которых обеспечивает нужный эффект.

Оператор «пуск таймера» порождает, обнуляет и запускает таймер и присваивает ему имя *A*.

Оператор «синхронизатор» задерживает выполнение размещенного справа от него визуального оператора до наступления момента, указанного в иконе «синхронизатор».

Например, синхронизатор $A = 2 \text{ мин } 45 \text{ с}$ на рис. 143 задерживает выдачу команды ПОДАЧА.ТОПЛИВА до момента, когда таймер *A* отсчитает 2 минуты 45 секунд.

Сравнивая алгоритмы на рис. 142 и 143, можно заметить, что они почти эквивалентны. Почему почти?

Чтобы разобраться, рассмотрим идеальный случай. Предположим, что время, необходимое для выдачи одной команды равно нулю. Имеется в виду, что перечисленные ниже команды выдаются за время, равное нулю:

- ОТКРЫТЬ.ТРУБОПРОВОД;
- ВКЛЮЧИТЬ.НАСОС;
- ОТКРЫТЬ.ЗАСЛОНКУ;
- ПОДАЧА.ТОПЛИВА;
- ПУСК.АГРЕГАТА.

В этом случае оба алгоритма 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 алгоритмы работают по-разному.

Практика показывает, что в некоторых ситуациях предпочтительным является *принцип паузы* (когда используется икона «пауза»). А в других – *принцип таймера* (когда используются иконы «пуск таймера» и «синхронизатор»).

Оба инструмента оказываются в равной степени необходимыми и полезными.

§7. АЛГОРИТМ РЕАЛЬНОГО ВРЕМЕНИ

На рис. 144 представлен более сложный алгоритм, в котором применяются операторы «пауза», «пуск таймера» и «синхронизатор».

В средней ветке изображена икона «пауза» с надписью 2мин48с. Это означает, что после завершения процедуры ВОЛШЕБНЫЙ РЕМОНТ ТАРЕЛКИ 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 надписью $A = 0$. Данный оператор порождает, обнуляет и запускает таймер A .

В той же ветке установлены три иконы «синхронизатор по таймеру» с надписями $A = 3\text{мин}$, $A = 5\text{мин}$ и $A = 8\text{мин}$. 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.

На рис. 144 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.

§8. ЦИКЛ ЖДАТЬ

Предположим, нужно в течение 3 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. 144 используются два оператора:

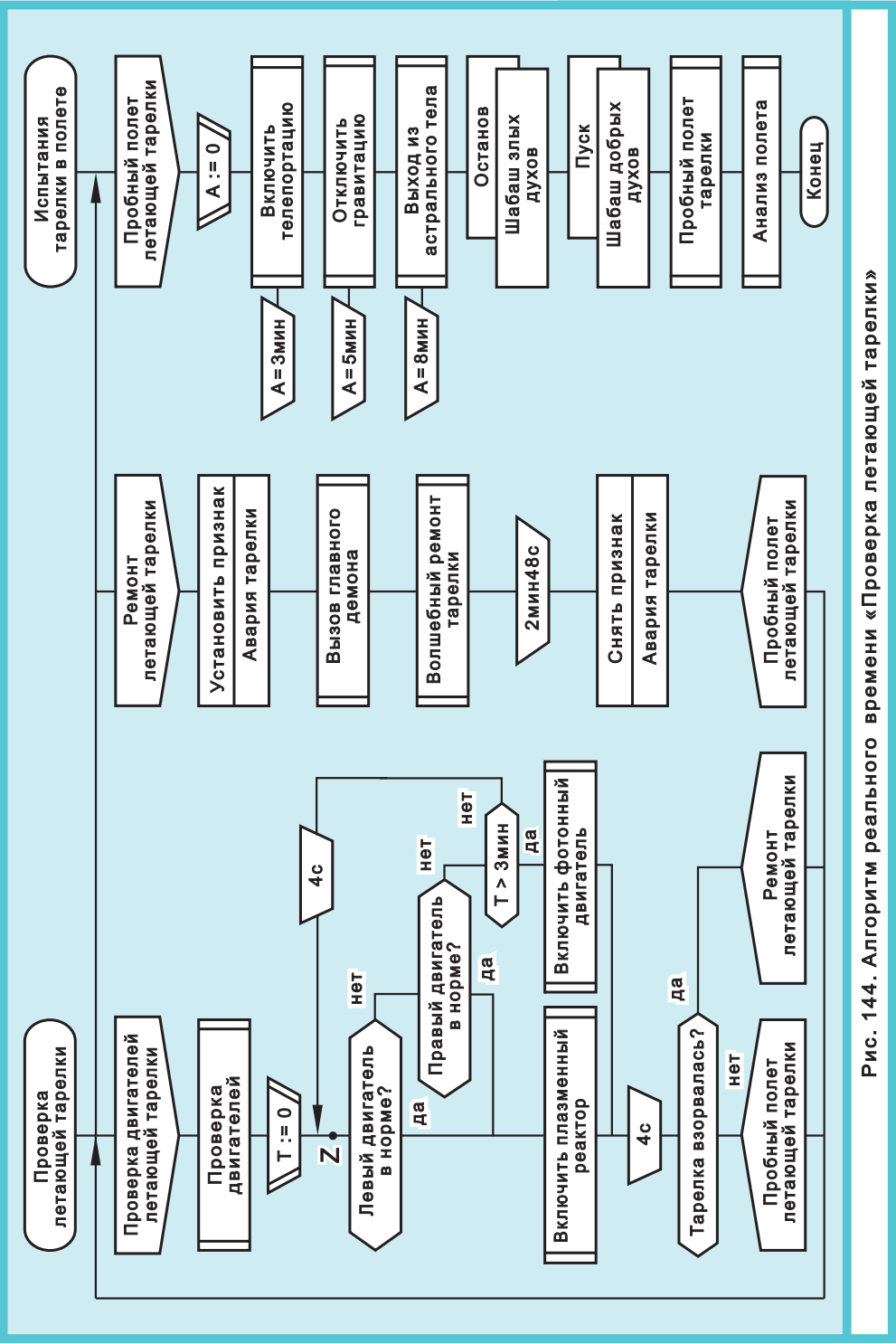


Рис. 144. Алгоритм реального времени «Проверка летающей тарелки»

- 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?
- $T > 3\text{мин.}$

Последний оператор проверяет: значение таймера T больше трех минут?

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

На рис. 144 видно, что цикл ЖДАТЬ закончится в момент обнаружения одного из ожидаемых признаков, а если они не появятся, – через 3 минуты.

§9. ЦИКЛ «ЖДАТЬ» В ОБЩЕМ ВИДЕ

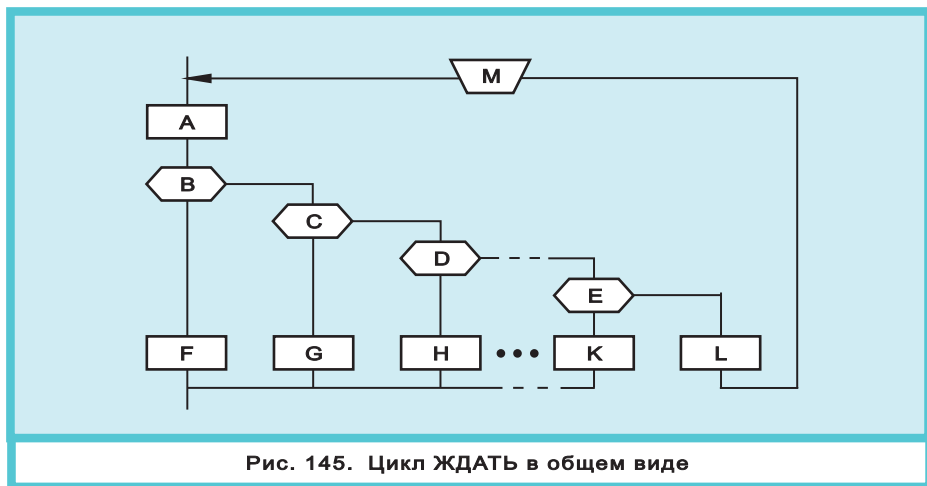
В общем виде цикл ЖДАТЬ показан на рис. 145. Он позволяет организовать режим ожидания признаков B, C, D, \dots, E .

- Если первым появится признак B , выполняется действие F .
- Если B отсутствует и первым придет признак C , реализуется действие G .
- Если B и C отсутствуют и первым будет D , выполняется H . И так далее.
- Операторы A и L обычно не используются.

- Задача ожидания нескольких признаков (когда система должна по-разному реагировать на каждый признак) является типичной при разработке систем реального времени.
- Цикл ЖДАТЬ – удобное средство для решения подобных задач.

§10. ОПЕРАТОР «ПЕРИОД»

Сравнивая макроиконки 4 и 7 на рис. 18 (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 алгоритм реального времени организует ожидание как дискретный процесс и запускает цикл ЖДАТЬ периодически. Отсюда вытекает, что период – важная характеристика цикла ЖДАТЬ.

Как работает оператор «период»? Поясним на примере. На рис. 144 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.

§11. 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. 146) или параллельно (рис. 147).

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

Например, на рис. 144 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. 146.

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. 147).

Чтобы организовать параллельную работу, нужно в дракон-схеме основного алгоритма *A* нарисовать икону «параллельный процесс» (рис. 17, 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. 144. 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. 144).

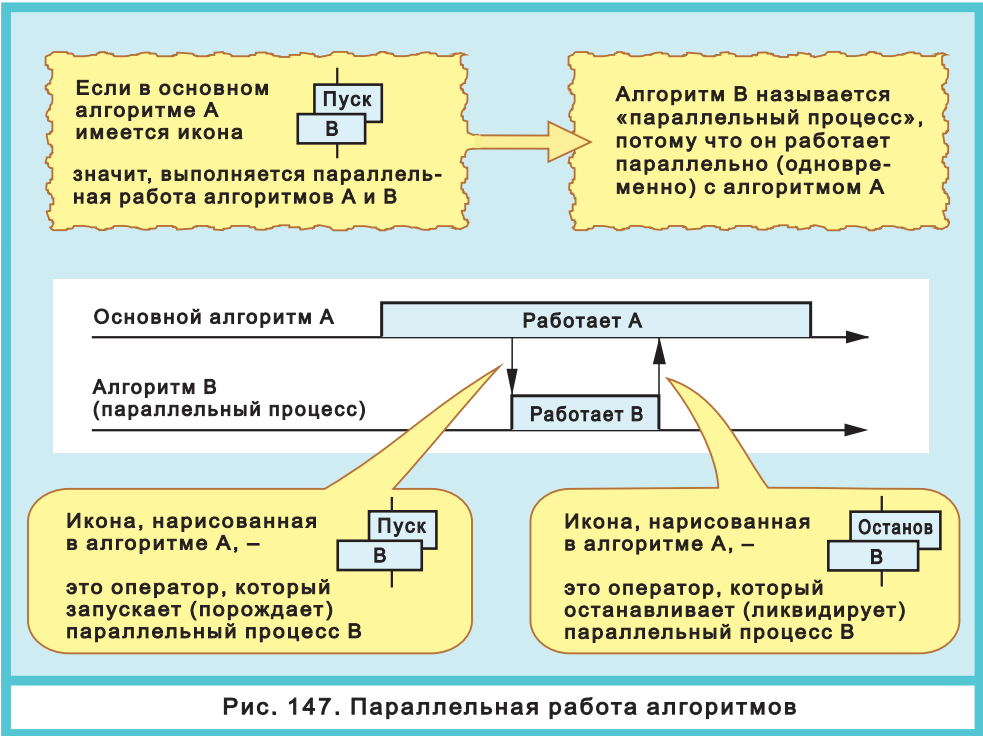
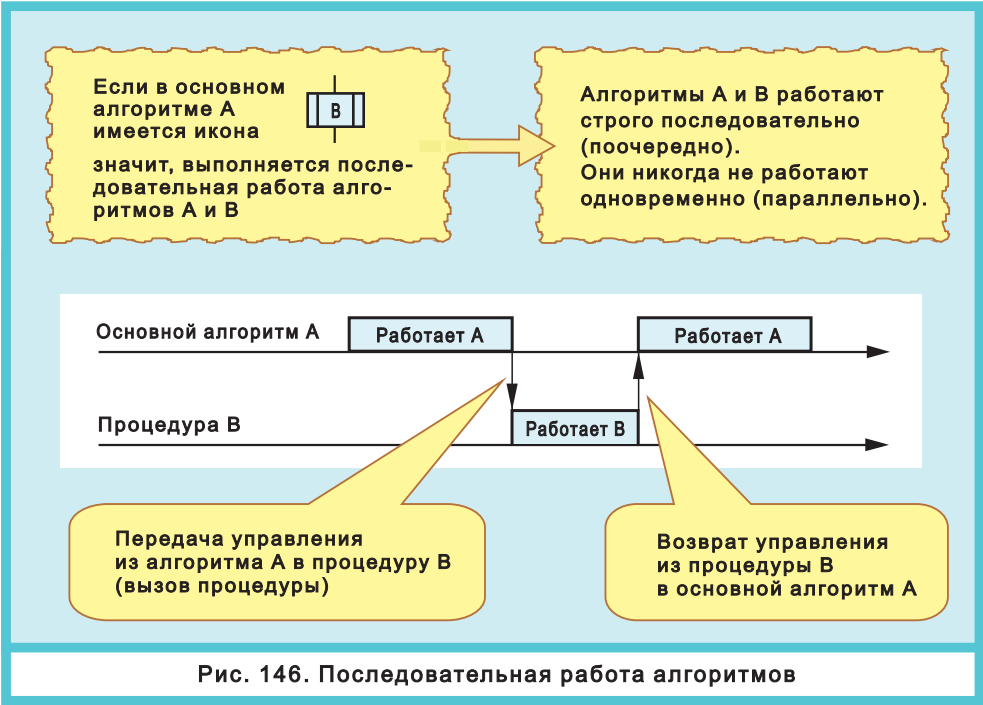
Указанная команда ликвидирует параллельный процесс **ШАБАШ ЗЛЫХ ДУХОВ**. В этот момент одновременная работа заканчивается.

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

§12.ОСОБЕННОСТИ ОПЕРАТОРОВ РЕАЛЬНОГО ВРЕМЕНИ

Операторы реального времени – это формальные операторы языка **ДРАКОН**. Однако их можно использовать и 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. 140 и 148. При этом в цикле **ЖДАТЬ** икону «период» обычно опускают, чтобы не загромождать рисунок (см. последнюю ветку на рис. 140). Однако если длительность периода нужна для понимания, икону «период» можно сохранить (рис. 148).



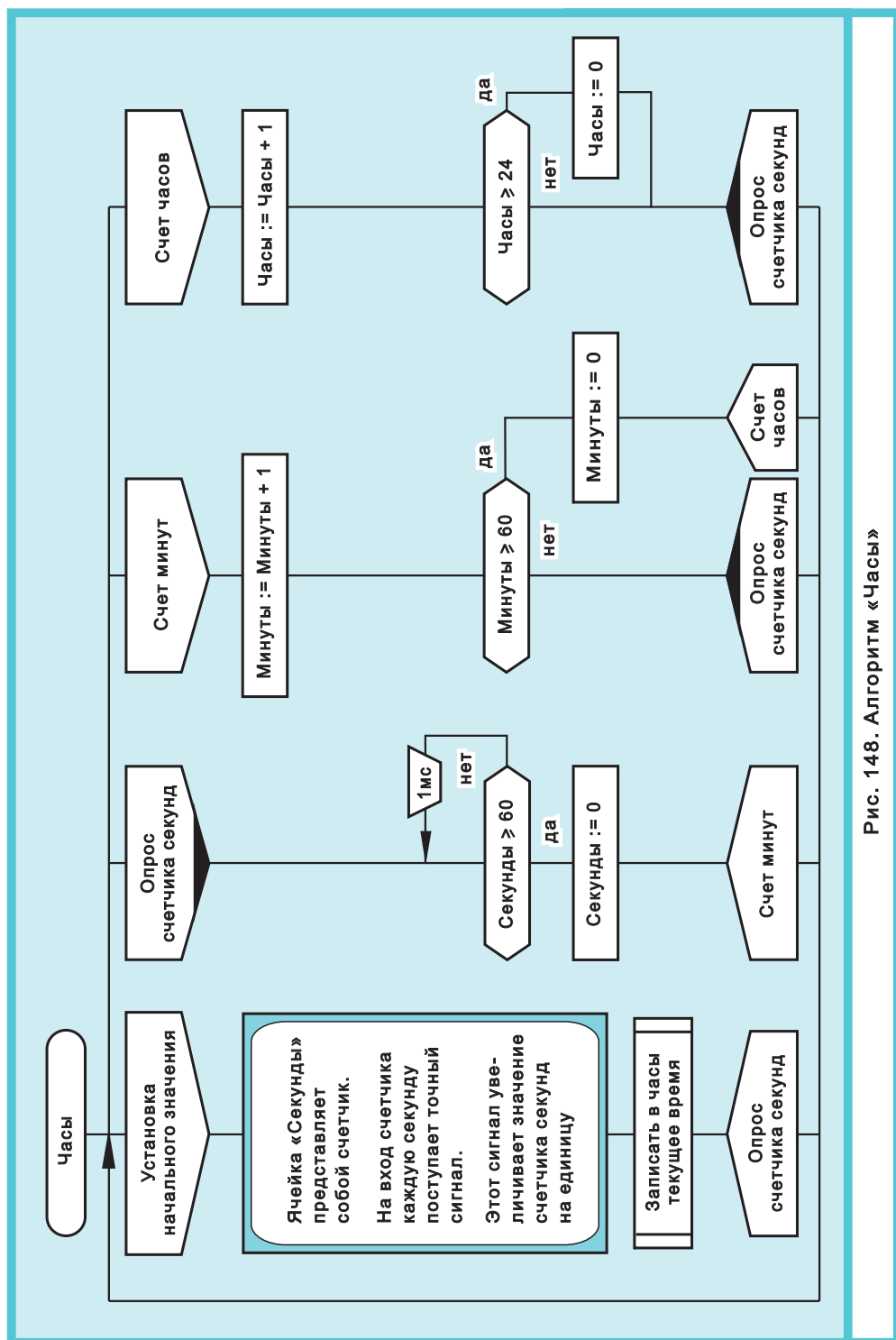


Рис. 148. Алгоритм «Часы»

§13. ОПЕРАТОР «ПРИСВОИТЬ»

Рассмотрим выражение $X = X + 1$. Что означает знак $=$ в этом выражении?

Предположим, что знак $=$ означает равно. Мы сразу придем к противоречию.

Действительно, если вычесть X из обеих частей уравнения, получим

$$0 = 1$$

что неверно.

Дело в том, что в алгоритмах в выражении $X = X + 1$ знак $=$ означает не «равно», а «присвоить».

Оператор «присвоить» $X = X + 1$ означает, что переменная X в левой части выражения увеличивает свое значение на 1.

Во избежание путаницы для оператора «присвоить» вместо знака $=$ используют знак «:=» (двоеточие и равно).

Говорят, что выражение $X := A$ присваивает переменной X в левой части то значение, которое записано в правой части (A). Например, $X := 157$ означает: переменной X присвоить значение 157.

В этой главе мы уже несколько раз использовали оператор «присвоить». Например, на рис. 143 таймеру A присвоено значение 0 ($A := 0$).

На рис. 144 обнулены два таймера $T := 0$ и $A := 0$.

На рис. 148 имеются пять операций присваивания:

- Секунды := 0
- Минуты := 0
- Минуты := Минуты + 1
- Часы := 0
- Часы := Часы + 1

§14. НЕСКОЛЬКО ВХОДОВ В ДРАКОН-АЛГОРИТМ

Дракон-алгоритм может иметь более одного входа. Чтобы организовать дополнительный вход, нужно поместить икону «заголовок» над иконой «имя ветки», как показано на рис. 144 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 алгоритм.

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

§15. КРАТКОЕ СООБЩЕНИЕ ДЛЯ ПРОГРАММИСТОВ

На рис. 140, 142–145 показаны операторы реального времени: «пауза», «пуск таймера», «синхронизатор», «период», а также цикл ЖДАТЬ. Эти операторы нарисованы внутри алгоритмов. Поэтому может создаться впечатление

чатление, что они реализуются этими алгоритмами (то есть прикладными программами реального времени). Но это не так.

На самом деле перечисленные операторы реализуются совместно:

- прикладной программой реального времени;
- дракон-диспетчером, входящим в состав операционной системы реального времени.

Когда в прикладной программе встречается оператор «пауза», происходят события, не показанные на наших рисунках. А именно, выход из прикладной программы и передача управления в дракон-диспетчер (с одновременной передачей параметра, записанного в иконе «пауза»).

Получив параметр, диспетчер отсчитывает время, указанное в паузе. Когда время истечет, диспетчер возвращает управление в прикладную программу – в точку, расположенную после иконы «пауза».

Иными словами, всякий раз, когда на рисунке алгоритма изображена пауза происходят два события:

- выход из прикладной программы (в начале паузы);
- вход в прикладную программу (в конце паузы).

Рассмотрим еще один пример – оператор «период». Длительность периода отсчитывает не прикладная программа на рис. 144, 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. 144). Если все три условия дают ответ «нет», оператор «период» возвращает управление в дракон-диспетчер. Таким образом, функционирование цикла ЖДАТЬ обеспечивается совместными усилиями прикладной программы и дракон-диспетчера.

Этот вывод относится ко всем операторам реального времени.

На рисунках показаны алгоритмы, которые имеют одно начало (один вход) и один конец (один выход). В действительности программы реального времени имеют много входов и много выходов. Дополнительные входы и выходы появляются всякий раз, когда в алгоритм добавляется оператор пауза или период. Но эти дополнительные входы и выходы на рисунках не показаны. Они не показаны из эргономических соображений – чтобы не загромождать рисунок.

§16. 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 алгоритмов и программ для управляющих систем.
3. Важную роль играет эргономичность операторов реального времени. Как и другие операторы языка ДРАКОН, они имеют визуальный характер. Это 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.

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ

§1. ВВЕДЕНИЕ

Параллельные процессы играют важную роль в технике и многих других областях.

Определение параллельного процесса дано на рис. 147. Икона «параллельный процесс» показана на рис. 17, фигура И20. Краткое описание процесса дано в §11 главы 13 (см. также рис. 144).

В данной главе следующие выражения рассматриваются как синонимы:

- параллельный процесс;
- процесс;
- параллельный алгоритм.

§2. ПАРАЛЛЕЛЬНЫЕ ПРОЦЕССЫ В АЛГОРИТМЕ «ПРОВЕРКА АГРЕГАТА И РАКЕТЫ»

На рис. 149 изображены 15 параллельных процессов:

- вызывающий алгоритм ПРОВЕРКА АГРЕГАТА И РАКЕТЫ;
- 14 вызываемых алгоритмов, каждый из которых обозначен иконой «параллельный процесс» (7 алгоритмов в первой ветке и 7 – во второй).

Все вызываемые процессы запускаются сигналом ПУСК. Момент запуска точно определен оператором синхронизатор. Например, процесс КОНТРОЛЬ ПРИБОРОВ запускается в момент 103с (103 секунды).

Параллельные процессы могут заканчиваться двумя способами:

- по команде «Останов» (см. пример на рис. 144, правая ветка);
- без использования команды «Останов», то есть естественным путем, когда каждый процесс решит свою задачу и достигнет конца.

На рис. 149 показан случай, когда все вызываемые процессы заканчиваются естественным путем. Поэтому команда **ОСТАНОВ** не используется.

§3. ВРЕМЕННАЯ ДИАГРАММА ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

На рис. 150 показана временная диаграмма, иллюстрирующая алгоритм на рис. 149. В верхней строке темным цветом выделен вызывающий алгоритм. Он имеет самую большую длительность. Ниже расположены вызываемые процессы.

В самом верху указано время запуска всех процессов по таймеру. Процессы имеют разную длительность, потому что каждый процесс выполняет задачу за разное время.

§4. ПАРАЛЛЕЛЬНЫЕ ПРОЦЕССЫ В АЛГОРИТМЕ «ПРОВЕРКА ВОЗДУШНОГО СНАЙПЕРА»

На рис. 149 показан упрощенный случай. Одна и та же операция повторяется 14 раз. 14 синхронизаторов задают 14 моментов времени, определяющих запуск 14 параллельных процессов.

На рис. 151 показан более сложный случай. Наряду с таймером, синхронизатором и процессами применяются следующие иконы: вывод, вставка, вопрос и полка.

В первой ветке имеются 4 синхронизатора. Два из них запускают процессы **ЗАПРАВКА УСКОРИТЕЛЯ** и **АННИГИЛЯЦИЯ КВАРКОВ**. Третий включает процедуру **ЗАЩИТА НЕБЕСНОГО ЭКРАНА**. Четвертый выдает команду **НЕЙТРОННЫЙ ЗАЛП**.

Используются не только синхронизаторы, но и две паузы длительностью 5 секунд каждая. Первая пауза задерживает пуск процесса **ЗАЩИТА АТОМНЫХ ЯДЕР**. Вторая задерживает выдачу команды **БЛОКИРОВКА ШИФРА**.

Во второй ветке выполняются операции:

- через синхронизатор (момент 319 секунд по таймеру) запускается процесс **РАСКРУТКА ЭЛЕКТРОНОВ**;
- устанавливаются два признака **НЕТ НОРМЫ** и **ВКЛЮЧЕН РЕВЕРС**;
- применяются две иконы вопрос: **ПЕРЕХОД НА МАЛУЮ ТЯГУ?** и **КОНТРОЛЬ УРОВНЯ?**
- выдается команда **ВКЛЮЧИТЬ КАРУСЕЛЬ**;
- запускается процесс **ДЕРЖАТЬ УРОВЕНЬ**;
- и т.д.

В алгоритме на рис. 151 используются пять вызываемых параллельных процессов.

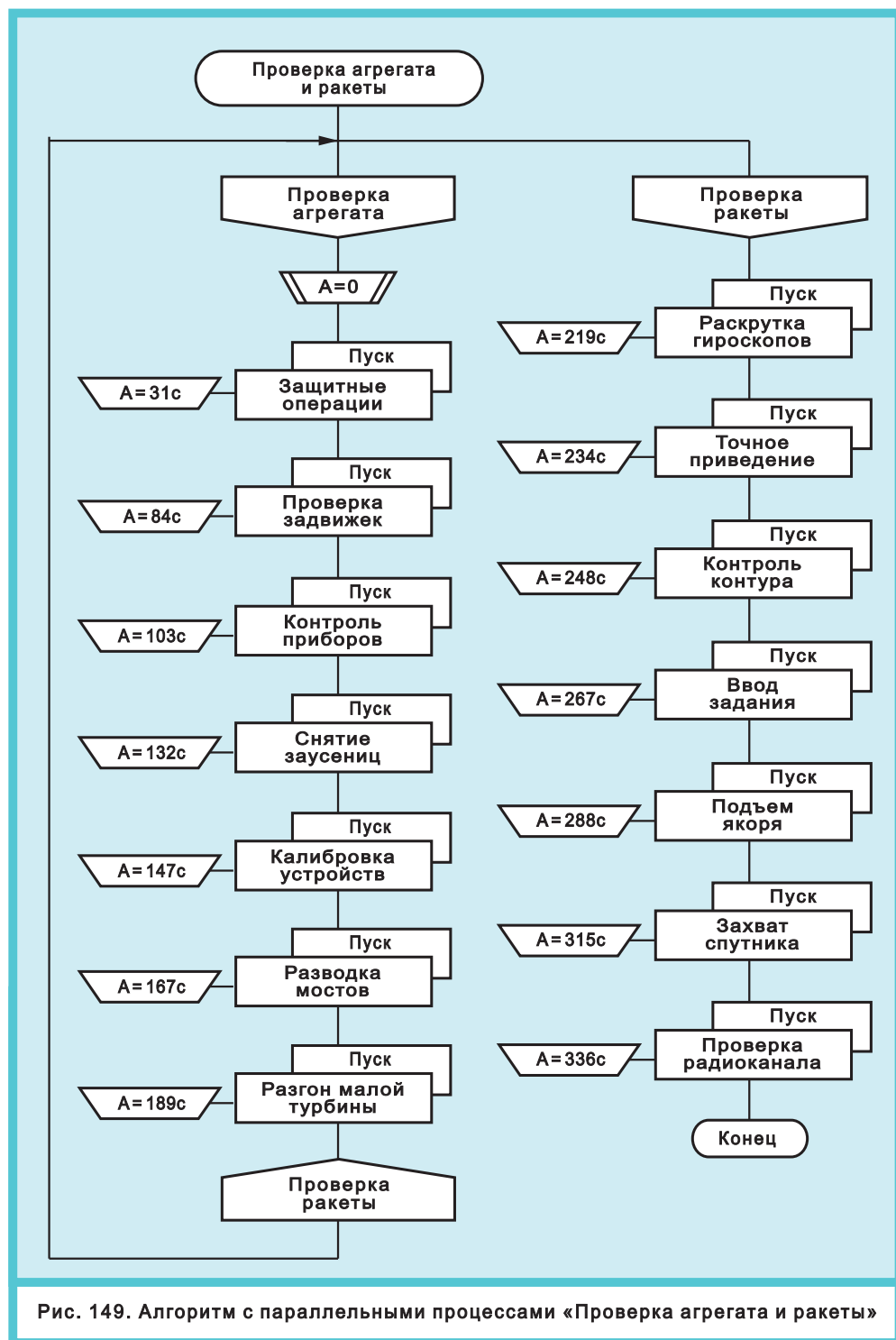


Рис. 149. Алгоритм с параллельными процессами «Проверка агрегата и ракеты»

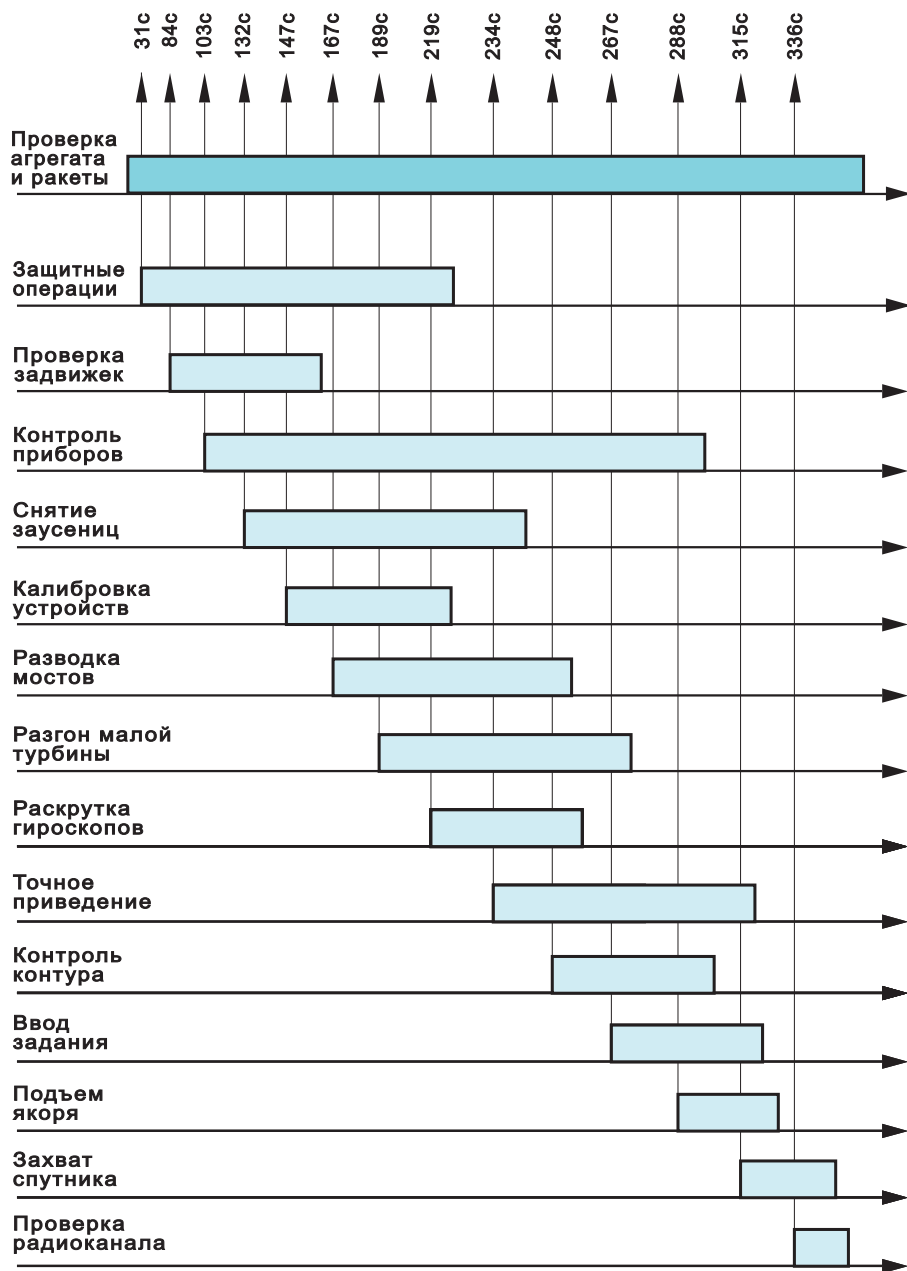


Рис. 150. Циклограмма параллельных процессов, запускаемых из алгоритма на рис. 149

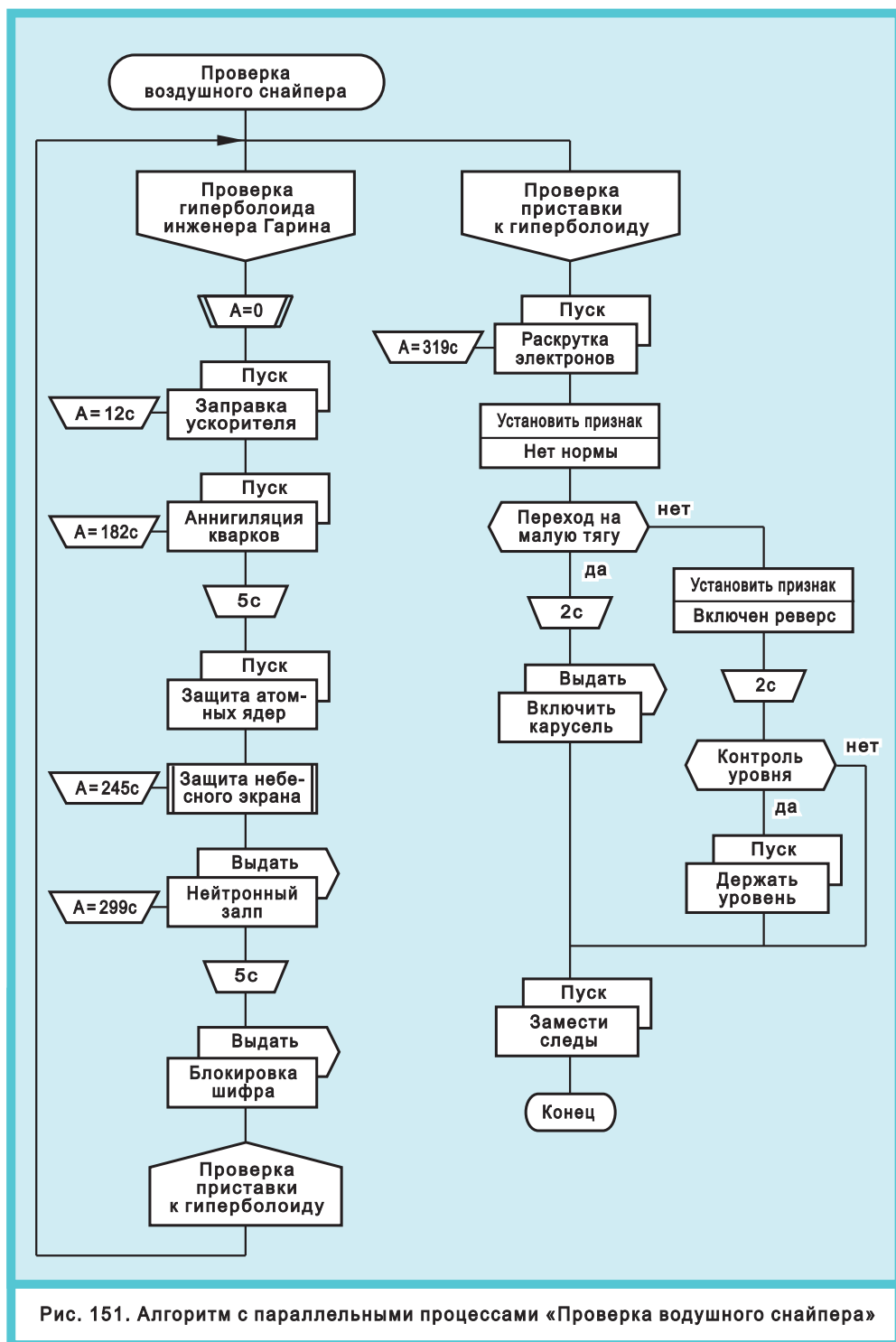


Рис. 151. Алгоритм с параллельными процессами «Проверка воздушного снайпера»

В первой ветке (через синхронизаторы) запускаются два процесса.
Во второй ветке применяются три процесса. Один запускается через синхронизатор. Второй – через иконы пауза и вопрос. Третий – через более сложную схему.

§5. КОМАНДЫ УПРАВЛЕНИЯ ПАРАЛЛЕЛЬНЫМИ ПРОЦЕССАМИ

Команды управления пишут на верхнем этаже иконы «параллельный процесс». Ниже представлен перечень команд управления:

Пуск	Осуществляет пуск параллельного процесса
Останов	Осуществляет останов параллельного процесса
Стоп	Осуществляет приостановку параллельного процесса
Рестарт	Осуществляет повторный пуск приостановленного параллельного процесса

§6. АЛЬТЕРНАТИВНЫЙ СПОСОБ ИЗОБРАЖЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

На рис. 152 показан алгоритм «Последний этап строительства дома». Можно сказать, что этот алгоритм нарисован «на другом языке». Вводятся новые обозначения и правила.

1. Жирная горизонтальная линия обозначает начало нескольких параллельных процессов.

2. Треугольник означает, что происходит прием сигналов от N параллельных процессов и их обработка.

3. Сигнал на выходе треугольника появится только тогда, когда:

- на входы треугольника поступят (не обязательно одновременно) N процессов;

- каждый из N процессов не только начнется, но и закончится. Иначе говоря, выходной сигнал треугольника возникнет в тот момент, когда завершится последний из N процессов;

4. Выходной сигнал треугольника разрешает выполнение действия, находящегося после треугольника.

Завершение всех процессов на входе треугольника позволяет выполнить действие на его выходе.

§7. ЖИРНЫЕ ЛИНИИ НА РИС. 152

На рис. 152 изображены три жирные параллельные линии. Каждая символизирует начало параллельных процессов.

Верхняя жирная линия обозначает начало четырех процессов:

- Подводка электролинии.
- Закупка электропроводов.
- Монтаж электрощита.
- Закупка водопроводных труб.

Средняя жирная линия указывает на начало следующих процессов:

- Прокладка электропроводки.
- Устройство крыши.
- Установка окон.
- Прокладка водопровода.

Нижняя жирная линия «начинает» процессы:

- Установка электроламп.
- Отделочные работы.

§8. ТРЕУГОЛЬНИКИ НА РИС. 152

На рис. 152 показаны четыре треугольника:

Левый верхний треугольник выполняет две функции:

- контролирует три процесса и дожидается, когда кончится последний из них;
- разрешает начать действие «Прокладка электропроводки».

Можно сказать иначе. Прокладка электропроводки начнется только после того, как закончатся процессы:

- Подводка электролинии.
- Закупка электропроводов.
- Монтаж электрощита.

Правый верхний треугольник выполняет две функции:

- контролирует два процесса и дожидается, когда кончится последний из них;
- разрешает начать действие «Прокладка водопровода».

Другими словами, «Прокладка водопровода» начнется лишь тогда, когда закончатся процессы:

- Монтаж электрощита.
- Закупка водопроводных труб.

Оставшиеся два треугольника работают аналогично.

§9. ВРЕМЕННАЯ ДИАГРАММА ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ ПРИ СТРОИТЕЛЬСТВЕ ДОМА

Циклограмма процесса «Последний этап строительства дома» показана на рис. 153.

Вспомним, что на вход левого верхнего треугольника (рис. 152) поступают сигналы трех процессов:

- Подводка электролинии.
- Закупка электропроводов.
- Монтаж электрощита.

На циклограмме изображен частный случай, когда названные три процесса:

- не пересекаются;
- расположены в таком порядке:
1) Закупка...2) Подводка... 3) Монтаж...

В этом случае окончание процесса «Монтаж электрощита» разрешает начать процесс «Прокладка электропроводки», как показано на рис. 153.

§10. ОБРАБОТКА ИНФОРМАЦИИ, ВЫПОЛНЯЕМАЯ СХЕМОЙ «ТРЕУГОЛЬНИК»

Уже говорилось, что любой треугольник на рис. 152 выполняет операции по обработке информации. О каких операциях идет речь?

Выделим на рис. 152 фрагмент, содержащий левый верхний треугольник и присоединенные к нему процессы. Поместим этот фрагмент на рис. 154 и исследуем его.

Алгоритм, реализуемый треугольником, показан на рис. 155. Он состоит из двух веток. Первая ветка проверяет, что все три входных процесса НАЧАЛИ работать. Вторая – что все входные процессы КОНЧИЛИ работу.

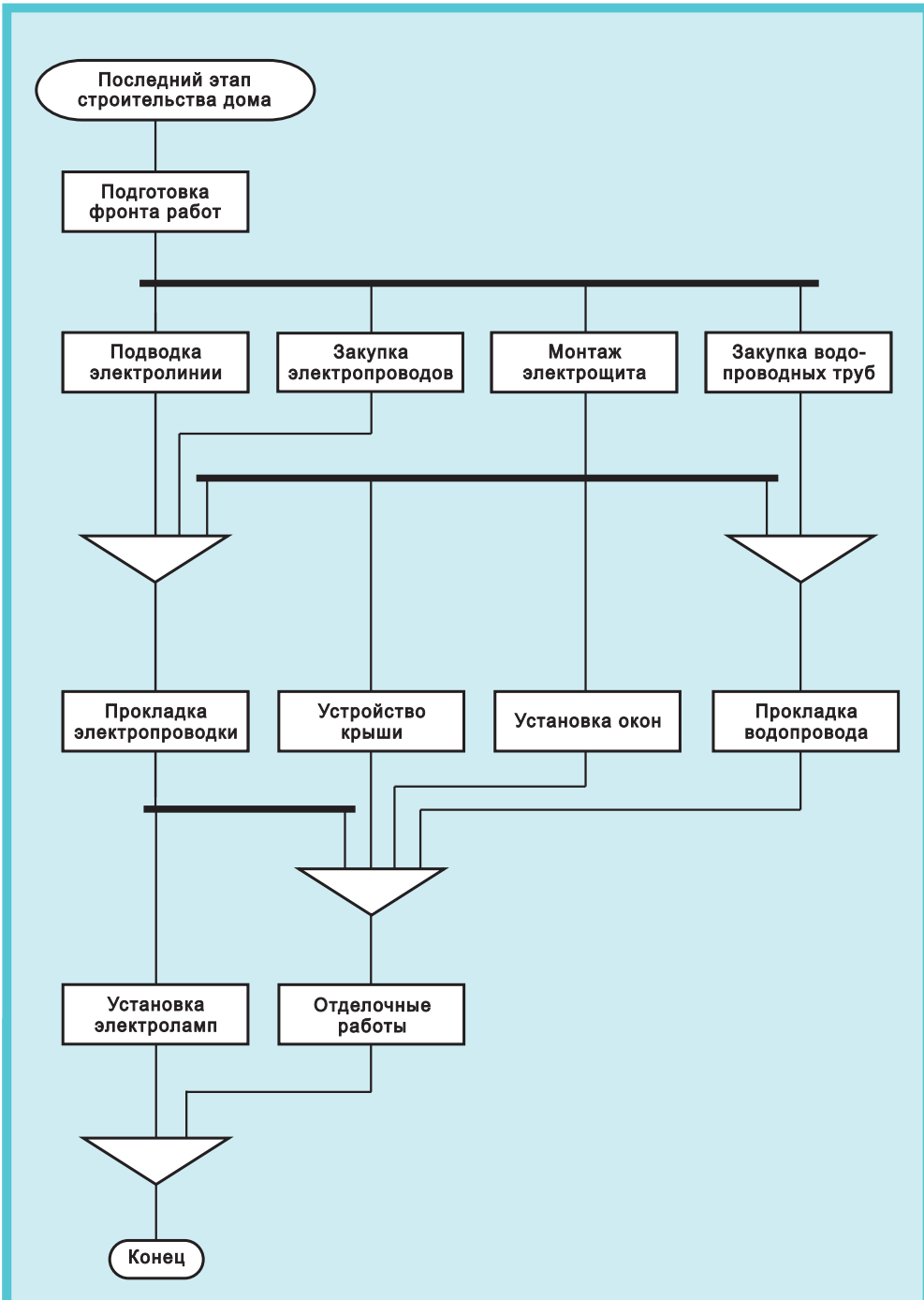
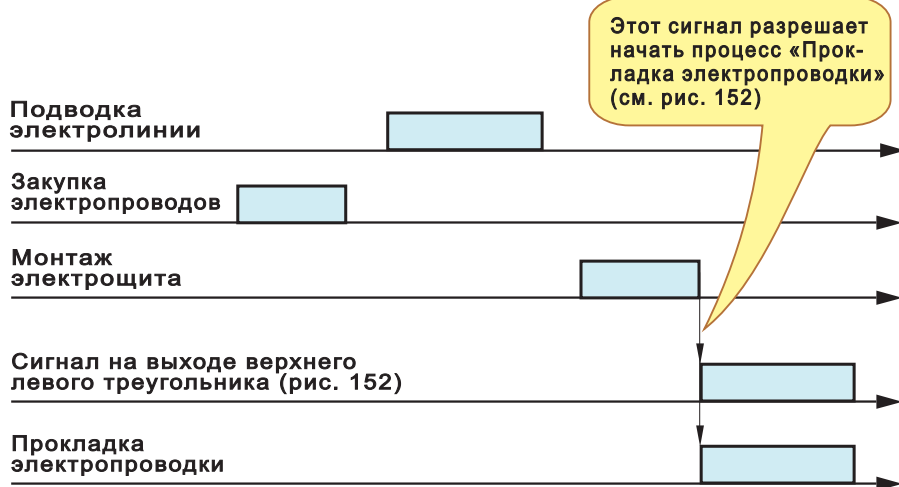


Рис. 152. Параллельные процессы на последнем этапе строительства дома



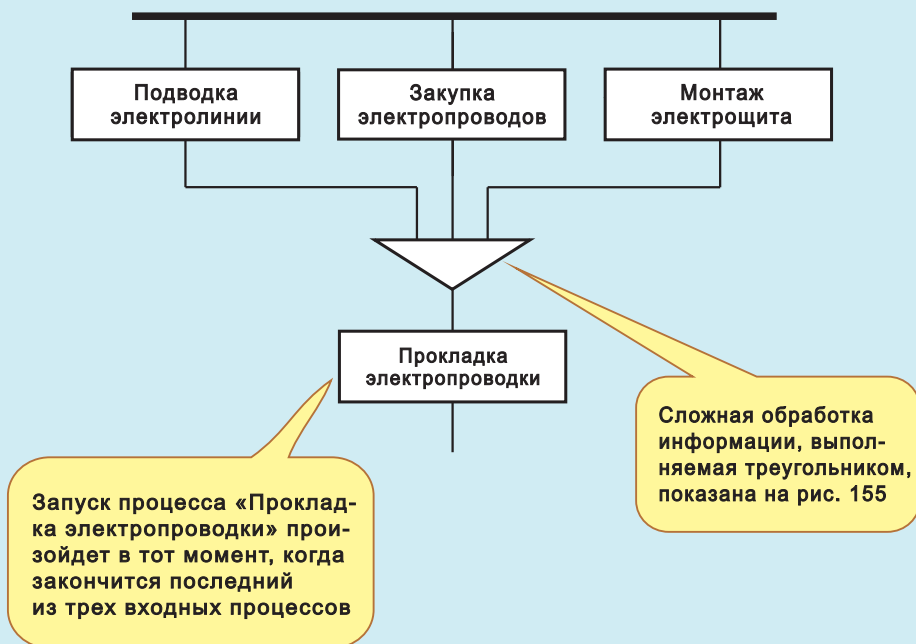
ПОЯСНЕНИЕ

- Три параллельных процесса (Подводка электролинии, Закупка электропроводов, Монтаж электрошита) могут начинаться и заканчиваться в любой момент времени
- Эти процессы могут не совпадать во времени или полностью или частично перекрываться
- Каждый из этих трех процессов обязательно должен начаться и закончиться
- Критическим моментом времени является окончание последнего из трех параллельных процессов
- На данном рисунке для примера показан случай, когда последним заканчивается процесс Монтаж электрошита
- В момент окончания процесса Монтаж электрошита на выходе треугольника появляется сигнал
- Этот сигнал разрешает начать процесс «Прокладка электропроводки»

Рис. 153. Циклограмма параллельных процессов на стройке дома. Показан момент, когда заканчивается последний из трех процессов и появляется разрешение начать процесс «Прокладка электропроводки» (см. рис. 152)

ФРАГМЕНТ

ПОСЛЕДНЕГО ЭТАПА СТРОИТЕЛЬСТВА ДОМА



ПОЯСНЕНИЕ

- Три параллельных процесса (Подводка электролинии, Закупка электропроводов, Монтаж электрощита) подают сигналы на вход треугольника
- Сигналы сообщают о моментах начала и конца каждого из трех параллельных процессов
- Треугольник производит обработку этих сигналов
- В результате обработки на выходе треугольника появляется сигнал, запускающий процесс «Прокладка электропроводки»

Рис. 154. Треугольник выполняет сложную обработку информации и запускает процесс «Прокладка электропроводки»

Рассмотрим первую ветку. Три процесса могут образовать 6 комбинаций (перестановок). Перечислим все 6 комбинаций.

1) Подводка линии	2) Закупка проводов	3) Монтаж щита
1) Подводка линии	3) Монтаж щита	2) Закупка проводов
2) Закупка проводов	1) Подводка линии	3) Монтаж щита
2) Закупка проводов	3) Монтаж щита	1) Подводка линии
3) Монтаж щита	1) Подводка линии	2) Закупка проводов
3) Монтаж щита	2) Закупка проводов	1) Подводка линии

Первая ветка разветвляется на 6 маршрутов. Каждый маршрут описывает одну из 6 комбинаций. Вторая ветка имеет точно такую же структуру.

В начале первой ветки имеется цикл ЖДАТЬ. В исходном положении все три процесса не работают. Цикл ЖДАТЬ поочередно опрашивает эти процессы. И определяет, какой процесс ПЕРВЫМ начнет работать (рис. 155).

Предположим, что первым включился в работу процесс «Монтаж электрощита». Из иконы вопрос «Монтаж...» выходим через «да» и попадаем в следующий цикл ЖДАТЬ. Этот цикл поочередно опрашивает ДВА процесса («Закупка проводов», «Подводка линии»). И определяет, какой из них ПЕРВЫМ начнет работать (рис. 155).

Предположим, первым начал работу процесс «Закупка проводов». Из иконы вопрос «Закупка ...» выходим через «да» и попадаем в следующий цикл ЖДАТЬ. Этот цикл периодически опрашивает ОДИН процесс («Подводка линии»). И «терпеливо» ждет, когда он вступит в работу (рис. 155).

Когда ожидание увенчается успехом, из иконы вопрос «Подводка линии» выходим через «да». Это означает, что мы прошли первую ветку по маршруту:

Монтаж щита	Закупка проводов	Подводка линии
-------------	------------------	----------------

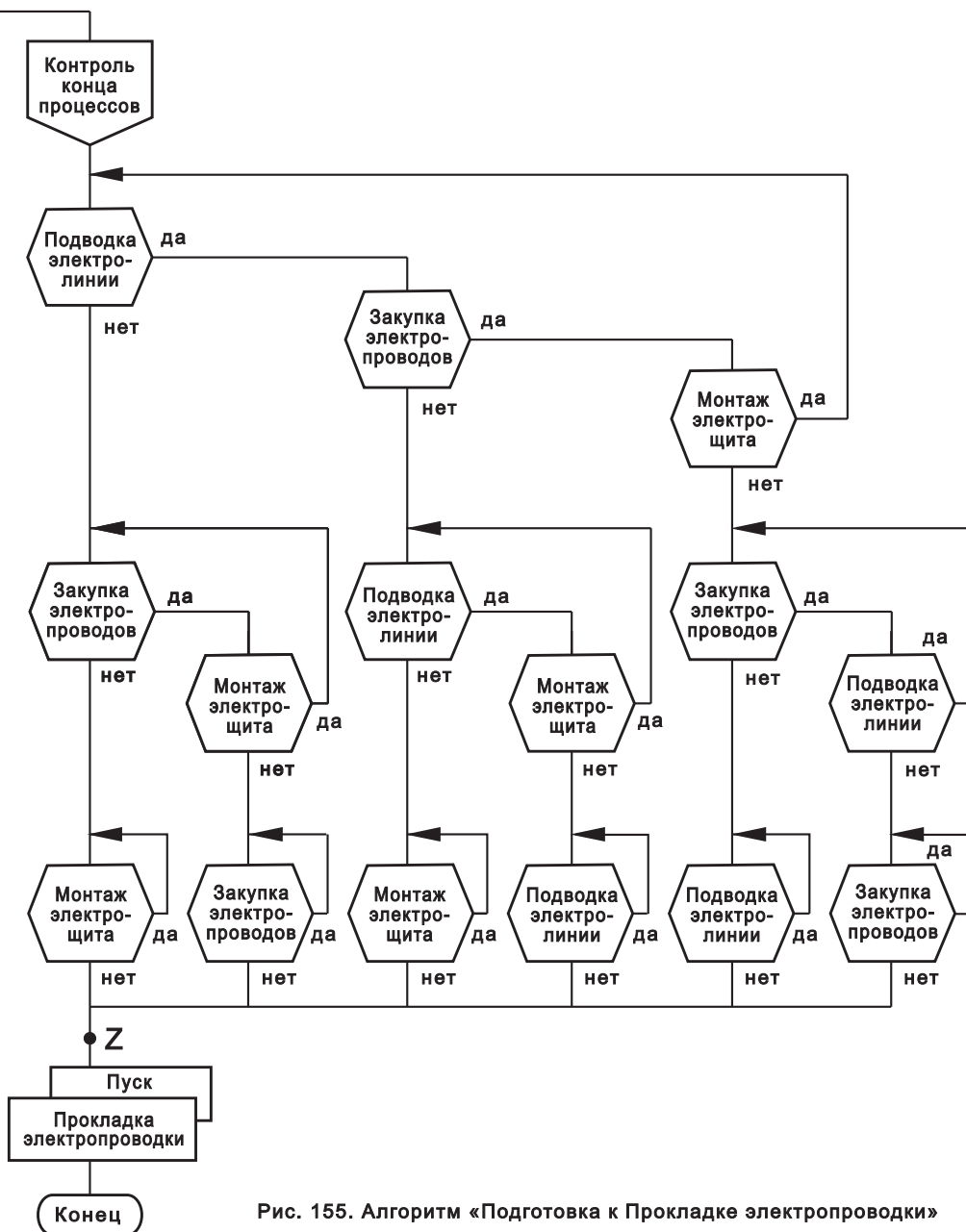
В итоге мы приходим в икону адрес, которая отправляет нас в начало второй ветки.

§11. МОМЕНТ ПЕРЕХОДА ИЗ ПЕРВОЙ ВЕТКИ ВО ВТОРУЮ НА РИС. 155

Вспомним логику работы алгоритма на рис. 155.

Первая ветка проверяет, что все три входных процесса НАЧАЛИ работать. Вторая – что все входные процессы КОНЧИЛИ работу.

Мы мысленно остановились в тот момент, когда первая ветка алгоритма завершила работу.



Зададим вопрос. В каком состоянии в этот момент находятся интересные нас процессы?

1. Мы знаем, что все три процесса НАЧАЛИ работать.
2. Мы знаем, что процесс «Подводка линии» продолжает работать.
3. Нам не известно состояние процессов «Монтаж щита» и «Закупка проводов». Возможно, они продолжают работать. Однако возможно, что они уже КОНЧИЛИ работу.

При разработке алгоритма второй ветки следует учесть худший случай. Худшим является случай, когда все три процесса продолжают работать (то есть ни один из них не кончил работу). Алгоритм второй ветки разработан с учетом такой возможности.

§12. ОБРАБОТКА ИНФОРМАЦИИ ВО ВТОРОЙ ВЕТКЕ НА РИС. 155

Задача второй ветки – определить, что все три процесса ЗАКОНЧИЛИ работу.

Вторая ветка работает, как первая. Только надписи «да» и «нет» всюду меняются местами.

В начале второй ветки имеется цикл ЖДАТЬ. В исходном положении, по крайней мере, один процесс работает. Однако, может быть и так, что все три процесса продолжают работать. Цикл ЖДАТЬ поочередно опрашивает три процесса и определяет, какой процесс первым кончил работать (рис. 155).

Предположим, что первым кончил работу процесс «Закупка проводов». Из иконы вопрос «Закупка ...» выходим через «нет» и попадаем в следующий цикл ЖДАТЬ. Этот цикл поочередно опрашивает ДВА процесса («Подводка линии», «Монтаж щита»). И определяет, какой из них первым кончит работать (рис. 155).

Дальше вторая ветка работает аналогично первой. Когда вторая ветка определит, что три процесса закончились (см. точку Z на рис. 155), производится Пуск процесса «Прокладка электропроводки».

На этом алгоритм завершается.

В схеме на рис. 155 используются 20 циклов ЖДАТЬ. Для простоты икона «период» не показана.

§13. РАЗДЕЛЕНИЕ И СЛИЯНИЕ ПРОЦЕССОВ

Пункт разделения
(concurrent fork)

- Обозначает разделение одного процесса на несколько параллельных процессов
- Обозначает разделение одного маршрута на несколько параллельных маршрутов

На рис. 152 пункт разделения обозначается жирной линией.

Пункт слияния
(concurrent join)

- Обозначает слияние параллельных процессов в один процесс
- Обозначает слияние нескольких параллельных маршрутов в один маршрут

На рис. 152 пункт слияния обозначен треугольником.

Таким образом, на рис. 152 пункт разделения и пункт слияния имеют *разные* обозначения.

Зачем нужны разные обозначения? Чтобы подчеркнуть, что в пункте слияния (в треугольнике) производится сложная обработка информации.

Всегда ли нужно это подчеркивать? Нет, не всегда.

Бывают случаи (и их немало), когда акцент на обработке информации в треугольнике является неуместным.

В такой ситуации целесообразно убрать треугольник. И ввести единое обозначение (*жирная горизонтальная линия*) и для пункта разделения, и для пункта слияния.

Пример показан на рис. 155а.

Если удалить треугольники на рис. 152, получим рис. 155б.

Нетрудно заметить, что удаление треугольников упрощает схему и устраняет лишние изломы линий. Поэтому рис. 155б является более простым и удобным, чем рис. 152.

§14. НЕДОСТАТОК РИСУНКА 155а

На рис. 155а имеются две совершенно одинаковые жирные линии: верхняя и нижняя. Это обстоятельство скрывает от читателя тот факт, что эти линии выполняют принципиально разные функции:

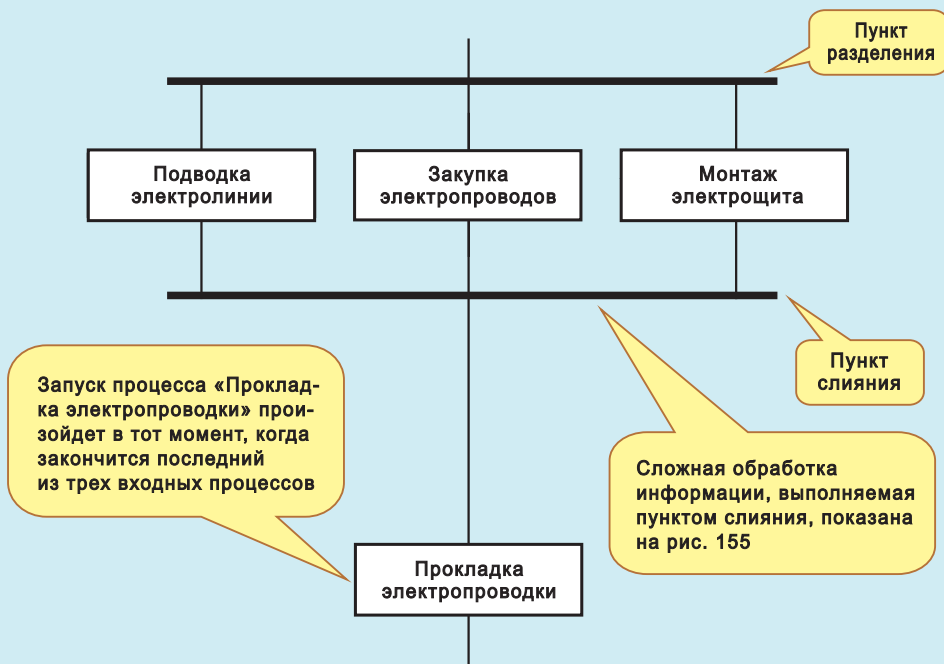
- верхняя линия выполняет простейшую функцию, символизируя начало параллельных процессов;
- нижняя линия, напротив, выполняет очень сложную функцию. Она реализует алгоритм обработки информации, показанный на рис. 155.

Если необходимо сделать акцент на обработке информации, надо пункт слияния изображать в виде треугольника, как на рис. 154.

Если же такой акцент не нужен, треугольник следует убрать. И рис. 154 заменить на рис. 155а.

ФРАГМЕНТ

ПОСЛЕДНЕГО ЭТАПА СТРОИТЕЛЬСТВА ДОМА



ПОЯСНЕНИЕ

- Три параллельных процесса (Подводка электролинии, Закупка электропроводов, Монтаж электрошита) подают сигналы на вход пункта слияния
- Сигналы сообщают о моментах начала и конца каждого из трех параллельных процессов
- Пункт слияния производит обработку этих сигналов
- В результате обработки на выходе пункта слияния появляется сигнал, запускающий процесс «Прокладка электропроводки»

Рис. 155а. Пункт слияния выполняет сложную обработку информации и запускает процесс «Прокладка электропроводки»

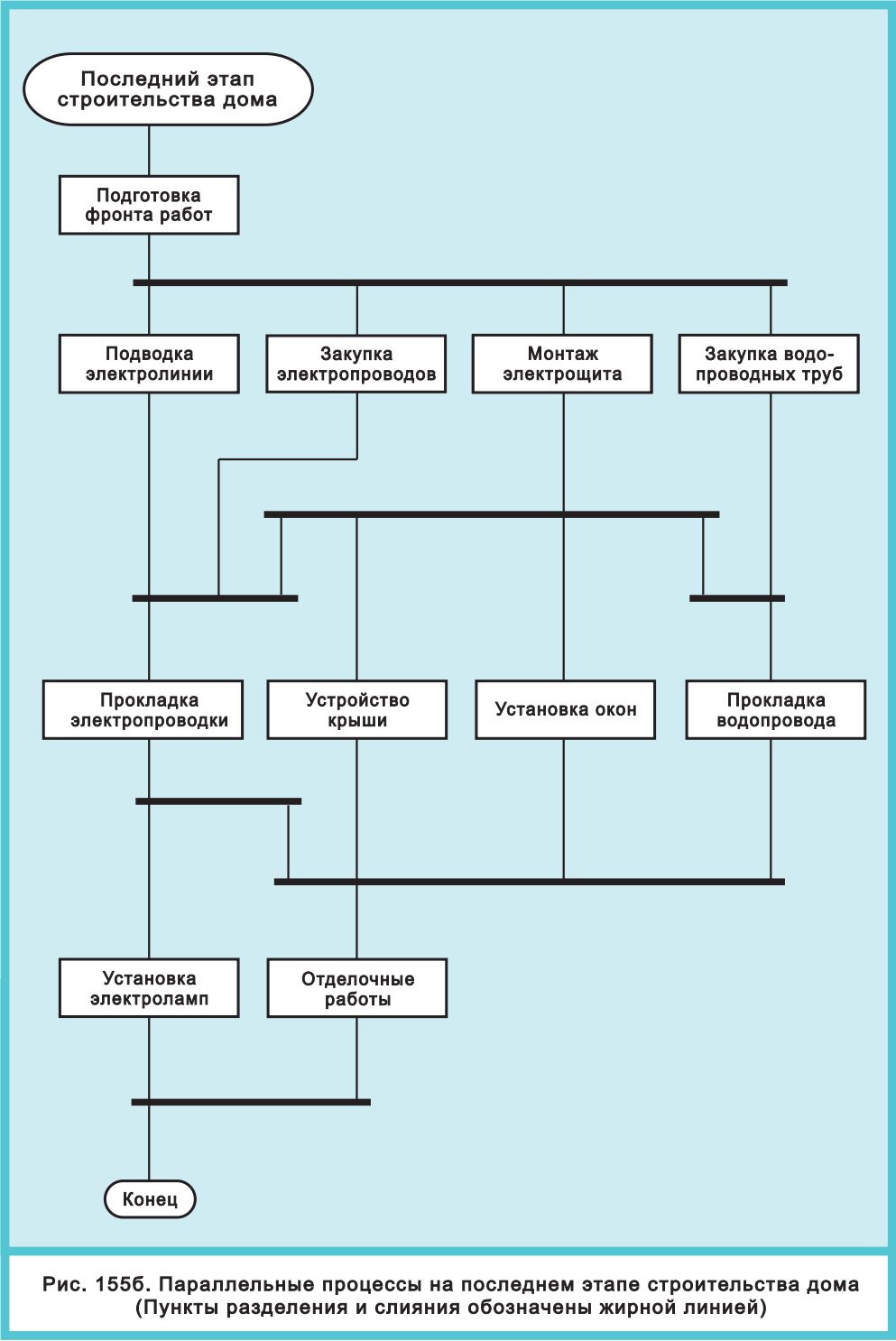


Рис. 1556. Параллельные процессы на последнем этапе строительства дома (Пункты разделения и слияния обозначены жирной линией)

§15. СРАВНЕНИЕ ЯЗЫКОВ

В данной книге все алгоритмы нарисованы на языке ДРАКОН. Но в этой главе (§§6–9, 13, 14) мы отошли от этого принципа. При описании строительства дома использован другой язык (назовем его «язык Z»). Последний значительно удобнее для изображения параллельных процессов типа «строительство дома».

В §§10–12 мы снова вернулись к ДРАКОНу. Потому что язык Z принципиально не может описать обработку информации в схеме треугольника. А язык ДРАКОН решает эту задачу точно и подробно.

Подведем итоги. Для разных задач нужны разные языки. В тех случаях, когда детали не нужны, когда требуется укрупненное и наглядное описание параллельных процессов, в некоторых случаях (но, разумеется, не всегда) язык Z оказывается более удобным.

Язык ДРАКОН можно расширить, включив в его состав элементы языка Z.

§16. ВЫВОДЫ

1. На языке ДРАКОН параллельный процесс запускается командой Пуск.
2. На языке ДРАКОН параллельные процессы могут заканчиваться двумя способами:
 - командой Останов;
 - без использования команды Останов, когда каждый процесс выполнит свою задачу и достигнет конца.
3. На языке ДРАКОН предусмотрены четыре команды управления параллельными процессами: Пуск, Останов, Стоп, Рестарт.
4. Рассмотрен альтернативный способ изображения параллельных процессов с помощью языка Z.
5. Язык Z – это условное название икон, обозначающих пункты разделения параллельных процессов (*concurrent fork*) и пункты слияния параллельных процессов (*concurrent join*).
6. Язык Z имеет две иконы: жирную линию и треугольник.
7. На языке Z начало нескольких параллельных процессов обозначается жирной линией.
8. На языке Z конец параллельных процессов обозначается:
 - либо треугольником (если нужно подчеркнуть, что в треугольнике выполняется сложная обработка информации);
 - либо жирной линией (если акцент на обработке информации не нужен).

9. Выходной сигнал треугольника возникает в тот момент, когда завершится последний из N параллельных процессов.
10. Завершение всех процессов на входе треугольника позволяет выполнить действие на его выходе.
11. Икона «треугольник» выполняет обработку информации по сложному алгоритму. Этот алгоритм нельзя описать на языке Z , но можно описать на языке ДРАКОН.
12. Для разных задач нужны разные языки. В тех случаях, когда детали не нужны, когда требуется укрупненное и наглядное описание параллельных процессов, в некоторых случаях язык Z оказывается более удобным.
13. Язык ДРАКОН целесообразно расширить и включить в его состав иконы языка Z .

ДРАКОН-СХЕМЫ И БЛОК-СХЕМЫ

§1. КРАСОТА 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* (эргономичным) 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. Тем 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 обду-

мывания неизбежно замедляется, что снижает производительность умственного труда.

ДРАКОН – графический язык, язык зрительных образов. С учетом сказанного можно уточнить: ДРАКОН – язык красивых (эргономичных) 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].

§2. ДРАКОН-СХЕМА ДОЛЖНА БЫТЬ ЛАКОНИЧНОЙ

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

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

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].

§3. СЛЕДУЕТ ИЗБЕГАТЬ НЕОПРАВДАНЫХ ИЗЛОМОВ СОЕДИНИТЕЛЬНЫХ ЛИНИЙ

Существуют «вредные» мелочи, которые затрудняют понимание дракон-схемы. Одна из них – 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. 156 и 157. На рис. 156 показана обычная блок-схема, заимствованная из технической литературы [3]. На рис. 157 изображена эквивалентная ей дракон-схема.

Эти рисунки позволяют выявить различия между неприглядной блок-схемой и красивой дракон-схемой. С точки зрения правил, рождающих алгоритмическую красоту, блок-схема на рис. 156 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 правила (правило лаконичности и правило минимизации изломов). И убедились, что в дракон-схеме они строго соблюдаются, а в блок-схеме грубо нарушены.

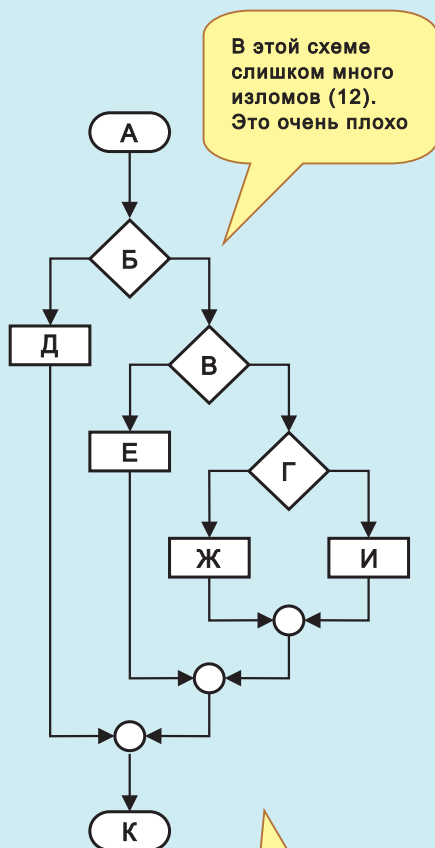
§4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ ДВУХ СХЕМ

Обратимся снова к рис. 156 и 157. Мы сделали лишь первый шаг к устранению графических недочетов. На рис. 156 осталось еще немало «грязи», которую необходимо отмыть.

Итак, продолжим наш критический анализ. Блок-схема на рис. 156 имеет следующие недостатки.

- Для обозначения развилки используется ромб, который занимает слишком много места. Ромб 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

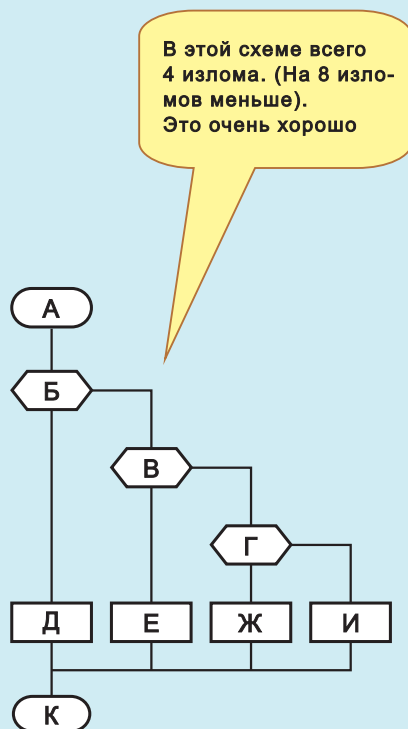
НЕПРАВИЛЬНО



Нарушено правило лаконичности. 3 кружка и 14 стрелок совершенно не нужны. Они являются паразитными элементами, «визуальными помехами», которые отвлекают внимание от главного.

Рис. 156. Плохая схема. Недостатки: слишком много изломов; имеются паразитные элементы

ПРАВИЛЬНО



Паразитные кружки и стрелки полностью устранены. Схема стала компактной, ясной и удобной. Правило лаконичности соблюдается.

Рис. 157. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

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 более наглядной и компактной.

Да, конечно, каждое из этих улучшений является незначительным 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.

§5. КРИТИКА ТРАДИЦИОННЫХ БЛОК-СХЕМ

Цивилизация не может жить без чертежей, не может жить и без алгоритмов. Блок-схемы алгоритмов очень важны для понимания тайн современного производства и управления. Однако многие преподаватели и разработчики алгоритмов создают алгоритмических «уродцев», на которые неприятно смотреть. По их мнению, красота и алгоритм несовместимы.

К счастью, в последнее время появились алгоритмические «мятежники», которые называют традиционные блок-схемы алгоритмов [4] «мусорными блок-схемами». Потому что изображенные на этих схемах хитросплетения блоков, соединенные хаосом рваных линий, больше напоминают кучу мусора, нежели регулярную структуру.

Образчик подобного мусора представлен на рис. 158 [5]. Этот «мусорный» алгоритм мы подвергли «косметической операции» в «салоне алгоритмической красоты». И превратили в изящную дракон-схему (рис. 159). Сравним что было и что стало.

Схема на рис. 158 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 входа (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 выделения главного маршрута.

Блок-схема на рис. 158, как и предыдущая (рис. 156), по всем параметрам проигрывает дракон-схеме. Мы убедились, что алгоритмическая красота достигается благодаря совокупному действию многих правил, каждое из которых, взятое по отдельности, выглядит скромным и будничным.

§6. РАЗРЫВ ШАМПУРА – СЕРЬЕЗНАЯ ОШИБКА

Разорванный шампур резко искажает зрительный образ дракон-схемы и может повлечь за собой большие неприятности. Между тем еще совсем недавно блок-схема цикла с разорванным шампуром не только не осуждалась, а наоборот, была рекомендована стандартом *ANSI* (Американский национальный институт стандартов).

На рис. 160 представлена схема, взятая из источника [6], где она характеризуется как «стандартная блок-схема *ANSI*».

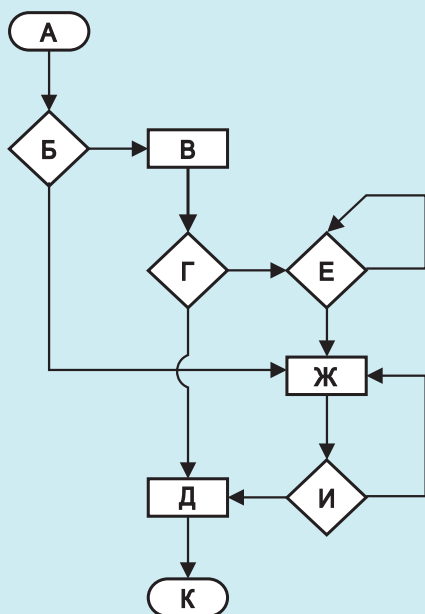
Сравнение этой схемы с эквивалентной дракон-схемой на рис. 161 позволяет выявить многочисленные дефекты:

- 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.

НЕПРАВИЛЬНО

В этой схеме нет шампура.
Это плохо. Схема без шампура,
как всадник без головы.

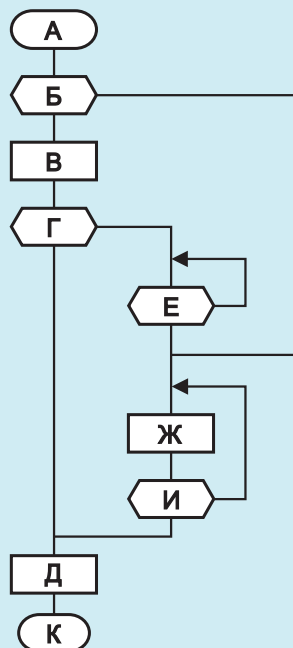


В этой схеме много
эргономических ошибок.
Она похожа на запутанный
клубок, в котором трудно
разобраться

Рис. 158. Плохая схема. Такие схемы часто рисуют многие уважаемые ученые, забывающие об эргономике

ПРАВИЛЬНО

В этой схеме
есть шампур.
Это хорошо.

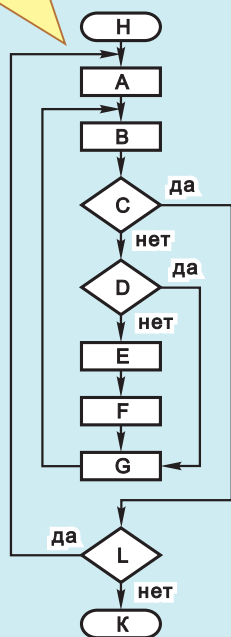


Все ошибки исправлены.
Шампур — путеводная нить
для понимания схемы

Рис. 159. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

НЕПРАВИЛЬНО

В этой схеме слишком много стрелок (12). Это плохо. Лишние стрелки являются визуальными помехами.



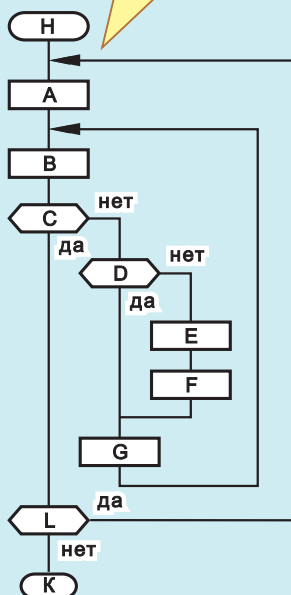
В схеме царит беспорядок.

1. Ошибка «Разрыв шампура».
2. Ошибка: икона G имеет два входа (один сбоку) и выход слева.
3. Ошибка: петли цикла закручены по часовой стрелке.

Рис. 160. Плохая схема. В ней много эргономических ошибок, что затрудняет понимание алгоритма

ПРАВИЛЬНО

В этой схеме всего 2 стрелки. (На 10 стрелок меньше). Это очень хорошо.



Беспорядок устранен.

В схеме, как и положено, есть шампур. Входы и выходы икон нарисованы не хаотично, а по правилам. В результате схема стала ясной и наглядной.

Рис. 161. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

§7. ГРАФИЧЕСКИЙ АНАЛИЗ ЦИКЛА «ПОКА»

На рис. 162–165 изображены блок-схема и дракон-схема цикла ПОКА. Сравним их между собой [7, 8].

На блок-схеме (рис. 162) можно указать следующие недочеты:

- Между иконами *E* и *K* имеет место разрыв шампура (нарушено правило, согласно которому один из путей, идущих от входа к выходу, должен проходить по главной вертикали).
- Ниже иконы *E* через разрыв шампура проходят две нежелательные горизонтальные линии
- Две петли обратной связи циклов ПОКА закручены по часовой стрелке (в дракон-схеме они закручены против часовой стрелки).
- Используются неудобные ромбы (в дракон-схеме их заменяют эргономичные иконы «вопрос»).
- Используются 8 стрелок, из которых 6 – паразитные (в дракон-схеме всего 2 стрелки).
- Имеется два избыточных излома линии (в блок-схеме 10 изломов, в дракон-схеме только 8).
- В блок-схемах отсутствует графическая стандартизация циклов. В дракон-схемах стандартизация циклов строго соблюдается. Об этом можно судить по рис. 163 и 165. Голубая заливка окружает внутренний цикл ПОКА. Белая заливка окружает внешний цикл ПОКА.

Сравнивая рис. 162–165, легко убедиться, что дракон-схемы во всех отношениях превосходят блок-схемы.

§8. ШАМПУР КАК «КОМПОЗИЦИОННЫЙ ЦЕНТР» ЗРИТЕЛЬНОЙ КАРТИНЫ АЛГОРИТМА

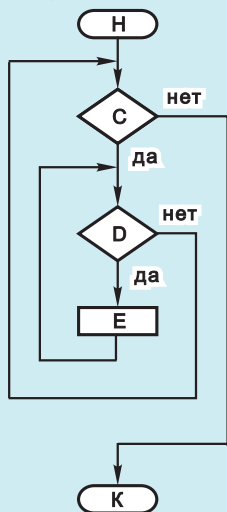
В живописи есть понятие *композиционный центр картины*. Так называют основное место картины, центр внимания, смысловое ядро произведения, которое невольно притягивает к себе взгляд зрителя.

Вернемся к зрительному образу дракон-схемы алгоритма. Аналогом композиционного центра можно считать шампур примитива (и шампуры веток силуэта).

Роль шампура в зрительно-смысловой структуре алгоритма очень велика. Он выполняет роль *главной вертикали* зрительной картины алгоритма. Шампур должен быстро и естественно привлечь к себе внимание читателя, правильно сориентировать его в структуре алгоритма. И тем самым облегчить восприятие и понимание существа дела.

НЕПРАВИЛЬНО

В этой схеме слишком много стрелок (8) и изломов (10). Это плохо. Лишние стрелки и изломы являются визуальными помехами.

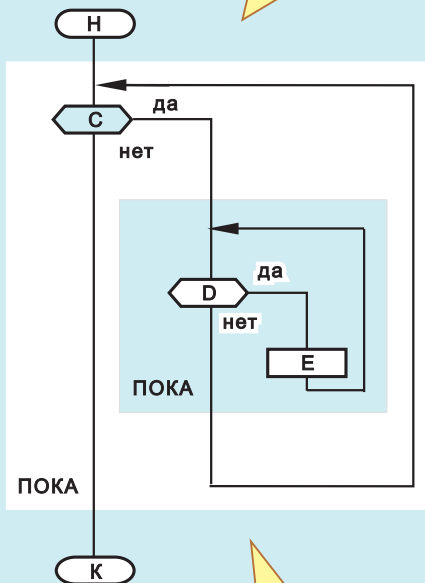


В схеме много ошибок.
1. Ошибка «Разрыв шампура».
2. Ниже иконы Е через разрыв шампура проходят две горизонтальные линии.
3. Ошибка: в двух циклах ПОКА петли цикла закручены по часовой стрелке.

Рис. 162. Плохая схема. В ней много эргономических ошибок, что затрудняет понимание алгоритма

ПРАВИЛЬНО

В этой схеме всего 2 стрелки. (На 6 стрелок меньше). Каждая стрелка обозначает цикл. Две стрелки обозначают две петли цикла. В этой схеме всего 8 изломов. (На 2 излома меньше).

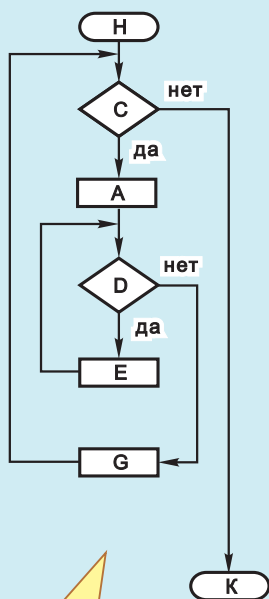


Ошибки устранены.
В схеме, как и положено, есть шампур.
Голубая заливка окружает внутренний цикл ПОКА.
Белая заливка окружает внешний цикл ПОКА.
В результате схема стала ясной и наглядной.

Рис. 163. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

НЕПРАВИЛЬНО

В этой схеме слишком много стрелок (8). Лишние стрелки являются визуальными помехами, которые отвлекают внимание от главного.

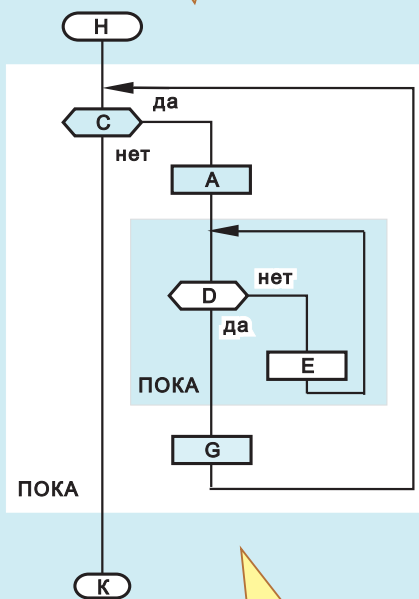


В схеме есть ошибки.

1. Ошибка «Нет шампура».
2. Ошибка: икона Е имеет выход слева.
3. Ошибка: икона G имеет вход справа и выход слева.
4. Ошибка: в двух циклах ПОКА петли цикла закручены по часовой стрелке.

ПРАВИЛЬНО

В этой схеме всего 2 стрелки. (На 6 стрелок меньше). Каждая стрелка обозначает цикл. Две стрелки обозначают две петли цикла



Ошибки устранены.

В схеме, как и положено, есть шампур. Голубая заливка окружает внутренний цикл ПОКА. Белая заливка окружает внешний цикл ПОКА. В результате схема стала ясной и наглядной.

Рис. 164. Плохая схема. В ней много эргономических ошибок, что затрудняет понимание алгоритма

Рис. 165. Хорошая (эргономичная) схема. Она нарисована по правилам языка ДРАКОН

Отсутствие шампура делает зрительный образ алгоритма «бесформенным», лишенным композиционного центра. В этом случае читатель лишается необходимых зрительных ориентиров, что заметно затрудняет чтение алгоритма.

Введение шампура в структуру алгоритма можно рассматривать как полезное средство, заметно облегчающее труд алгоритмистов.

§9. ВЫВОДЫ

1. Дракон-схемы имеют два принципиальных отличия от блок-схем. Дракон-схемы подчиняются:
 - строгим математическим правилам;
 - когнитивно-эргономическим правилам.
2. Красота алгоритма – это красота его зрительного образа, в частности, красота дракон-схемы.
3. Алгоритм можно назвать *красивым* (эргономичным) в том случае, если процесс зрительного восприятия и понимания алгоритма протекает с максимальной скоростью, наименьшими усилиями и максимальным эстетическим наслаждением.
4. Красота способствует более быстрому и глубокому пониманию алгоритмов. Чем больше красоты, тем легче мы постигаем суть дела. Тем быстрее и качественнее протекает важный производственный процесс, играющий немалую роль в мировой экономике, – процесс массовой разработки алгоритмов и программ.
5. Зрительный образ алгоритма должен быть лаконичным. Все ненужные, лишние детали должны быть отсечены.
6. Дракон-схема должна содержать лишь те элементы, которые необходимы для сообщения читателю существенной информации, точного понимания ее смысла и стимулирования правильных решений и разумных действий.
7. Чтобы дракон-схема была удобной для чтения, количество изломов соединительных линий должно быть минимальным.
8. В зрительно-смысловой структуре алгоритма шампур играет важную роль. Шампур должен быстро и естественно привлечь к себе внимание читателя, правильно сориентировать его в структуре алгоритма.
9. При отсутствии шампура уничтожается основа для выделения главного маршрута.
10. Разрыв или отсутствие шампура делает зрительный образ алгоритма «бесформенным», лишенным композиционного центра. В этом случае читатель лишается необходимых зрительных ориентиров, что затрудняет чтение алгоритма.

11. Алгоритмическая красота дракон-схем достигается благодаря совокупному действию многих правил, каждое из которых, взятое по отдельности, выглядит скромным и будничным.
12. В данной главе указаны эргономические недостатки блок-схем и описаны парные им достоинства дракон-схем.
13. Наиболее важным дефектом блок-схем является недостаток выразительных средств. Алгоритмическая структура «силуэт», являющаяся основным и наиболее мощным инструментом языка ДРАКОН, принципиально не может быть выражена в виде блок-схем.
14. Изложенные в данной книге теоретические обоснования, сравнительный анализ дракон-схем и блок-схем, а также многочисленные примеры убедительно доказывают, что блок-схемы не имеют будущего. Блок-схемы безнадежно устарели. Пользоваться ими недопустимо. Вместо них следует использовать дракон-схемы.

КОРОТКО О ПРОГРАММИРОВАНИИ

§1. ВВЕДЕНИЕ

Книга посвящена алгоритмам. Вопросы программирования в ней не рассматриваются. Данная глава является исключением. Цель главы – пояснить связь между языком Дракон и программированием.

Язык Дракон можно присоединить к некоторым языкам программирования и получить так называемые *гибридные языки*:

язык Дракон + язык Си	= гибридный язык Дракон-Си
язык Дракон + язык Дельфи	= гибридный язык Дракон-Дельфи
язык Дракон + язык Си#	= гибридный язык Дракон-Си#
язык Дракон + язык Джава	= гибридный язык Дракон-Джава
язык Дракон + язык Питон	= гибридный язык Дракон-Питон
язык Дракон + язык Перл	= гибридный язык Дракон-Перл
язык Дракон + язык Руби	= гибридный язык Дракон-Руби
язык Дракон + язык Ада	= гибридный язык Дракон-Ада
язык Дракон + язык Оберон	= гибридный язык Дракон-Оберон

и др.

В качестве примера рассмотрим гибридный язык Дракон-Си. И превратим программы на языке Си в программы на гибридном языке Дракон-Си. В этом случае Си называется *целевым языком*.

Пояснения даны на рис. 166 и 167.

§2. КАК ИЗОБРАЗИТЬ ОПЕРАТОР ЯЗЫКА СИ «IF–ELSE» НА ЯЗЫКЕ ДРАКОН-СИ

В первом примере на рис. 166 рассмотрен оператор *if–else*. В средней графе показана программа на языке Си, в которой используется этот оператор.

А в правой графе изображена математически эквивалентная ей программа на языке Дракон-Си.

В чем сходство этих программ?

Некоторые выражения из программы на языке Си без изменения перекочевали в дракон-программу. Вот пример:

● $a \geq 0$ (a больше нуля)

В правой графе это выражение помещено в икону вопрос, которая снабжена выходами «да» и «нет».

Еще пример:

● $x = f1;$

● $y = f2;$

В правой графе эти два оператора присваивания помещены в левую икону действие, присоединенную к выходу «да». (Заметим, что на языке Си присваивание обозначается знаком $=$).

Еще один пример:

● $x = r1;$

● $y = r2;$

В правой графе эти операторы помещены в правую икону действие, присоединенную к выходу «нет».

На этом сходство программ кончается. Укажем отличия.

В си-программе мы видим два ключевых слова *if*, *else*, четыре фигурных скобки и две круглые скобки.

В дракон-программе они исчезают и превращаются в чертеж. Благодаря этому схема приобретает важное качество – наглядность.

§3. КАК ИЗОБРАЗИТЬ ОПЕРАТОР ЯЗЫКА СИ «IF–ELSEIF–ELSE» НА ЯЗЫКЕ ДРАКОН-СИ?

Во 2-м примере на рис. 166 рассмотрен оператор *if–elseif–else*. Рядом показаны эквивалентные программы на языках Си и Дракон-Си. В обеих программах названный оператор записан в текстовой и графической форме соответственно.

В чем сходство двух программ? Ниже указаны выражения, записанные в си-программе, которые без изменения переносятся в дракон-программу:

● $x < b$

● $x = b$

● $x < c$

● $x = c$

● $x = d$

Эти выражения на схеме помещаются в две иконы вопрос и три иконы действие.

Перечислим отличия. В си-программе имеются три ключевых слова *if*, *elseif*, *else*, две круглые скобки и три точки с запятой. В дракон-программе они отсутствуют, превращаясь в чертеж.

Чертеж отчетливо показывает пространственную структуру программы. Благодаря этому зрительное восприятие структуры заметно улучшается.

§4. КАК ИЗОБРАЗИТЬ КЛЮЧЕВЫЕ СЛОВА «SWITCH, CASE, BREAK, DEFAULT» НА ЯЗЫКЕ ДРАКОН-СИ?

В 3-м примере на рис. 166 представлены ключевые слова *switch*, *case*, *break*, *default*.

Ниже представлен список выражений, которые не изменяются. В неизменном виде они переходят из си-программы в дракон-программу (в иконы выбор, вариант и действие):

- *n*
- *1*
- *2*
- *default*
- *x=a*
- *x=b*
- *x=c*

В си-программе видим большое количество избыточных слов и текстовых символов:

- *switch* (два раза),
- *case 1*,
- *case 2*,
- *break*,
- две фигурные скобки,
- две круглые скобки,
- две косые скобки,
- три двоеточия,
- пять точек с запятой,
- две звездочки.

Нужны ли все эти символы? Нет, не нужны!

В дракон-программе эти знаки исчезают. Они превращаются в приятный для глаза графический образ, пригодный для быстрого симультанного (панорамного) восприятия.

§5. КАК ИЗОБРАЗИТЬ ЦИКЛ «WHILE» НА ЯЗЫКЕ ДРАКОН-СИ?

В 4-м примере на рис. 166 рассмотрен цикл *while* (ПОКА). В средней графе показан цикл *while* на языке Си. В правой графе – эквивалентный ему графический цикл ПОКА на языке Дракон-Си.

Перечислим выражения, которые без изменений перемещаются в дракон-программу (в иконы вопрос и действие):

- $n++ < 50$
- $x = z + n;$
- $x = z - n;$

В си-программе имеются избыточные элементы:

- *while* (два раза);
- две круглые скобки;
- две фигурные скобки;
- две косые скобки;
- две звездочки.

В дракон-программе эти «паразитные» знаки отсутствуют, превращаясь в чертеж. Чертеж наглядно показывает маршруты и петлю обратной связи цикла.

§6. КАК ИЗОБРАЗИТЬ ЦИКЛ «DO-WHILE» НА ЯЗЫКЕ ДРАКОН-СИ?

В 5-м примере на рис. 167 рассмотрен цикл *do-while* (ДО). В средней графе показан цикл *do-while* на языке Си. В правой графе – эквивалентный ему графический цикл ДО на языке Дракон-Си.

Укажем выражения, которые без изменения переносятся в дракон-программу (в иконы действие и вопрос):

- $y = p - a;$
- $y = p + a;$
- $n-- > b$

Как и в предыдущих случаях, в си-программе имеются избыточные элементы:

- *do* (два раза);
- *while* (два раза);
- две круглые скобки;
- две фигурные скобки;
- две косые скобки;
- две звездочки.

В дракон-программе эти знаки исчезают, потому что на чертеже они не нужны. Чертеж гораздо лучше, чем текст, показывает структуру цикла.

Примеры 6 и 7 на рис. 167 можно анализировать по аналогии.

§7. КАК ПОСТРОИТЬ ГИБРИДНЫЙ ЯЗЫК ДРАКОН-СИ?

Чтобы построить язык Дракон-Си, надо по определенным правилам соединить графический синтаксис Дракона с текстовым синтаксисом языка Си.

Из последнего следует удалить все элементы, функции которых реализуются визуальными операторами Дракона.

Любой гибридный язык (например, Дракон-Си) почти полностью сохраняет концепцию, структуру, типы данных и другие особенности целевого языка (например, Си). При этом в строго определенном числе случаев текстовая нотация целевого языка заменяется на графическую. Такой прием позволяет существенно улучшить эргономический облик целевого языка.

§8. ПРЕОБРАЗОВАНИЕ ИЗ ГРАФИКИ В ИСХОДНЫЙ ТЕКСТ И ОБЪЕКТНЫЙ КОД

Предположим, нужно построить систему визуального программирования на гибридном языке Дракон-Си. Задачу можно решить, например, с помощью трех программ:

- дракон-конструктор,
- дракон-транслятор,
- компилятор языка Си.

Пользователь с помощью дракон-конструктора рисует на экране компьютерную программу на языке Дракон-Си (рис. 166, 167, правая графа).

Дракон-транслятор преобразует выходные коды конструктора в исходный текст языка Си (рис. 166, 167, средняя графа).

Стандартный компилятор Си превращает исходный текст Си в объектный код.

§9. ОСОБЕННОСТИ ЯЗЫКА ДРАКОН-СИ

Чтобы лучше уяснить особенности языка Дракон-Си, произведем мысленно обратное преобразование. Как видно из рис. 166 и 167, при преобразовании текстовой программы в визуальную исходный текст Си-программы разбивается на две части.

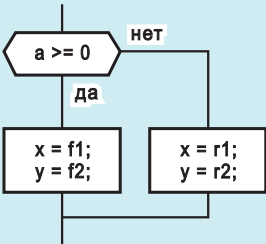
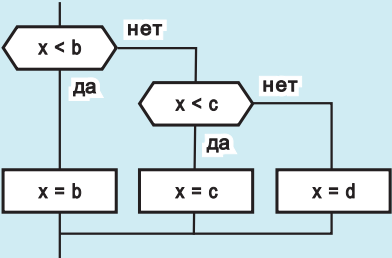
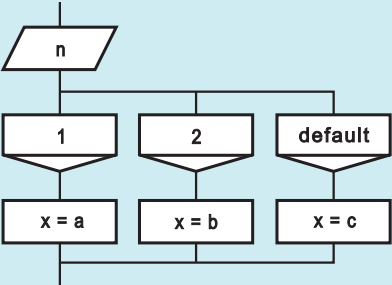
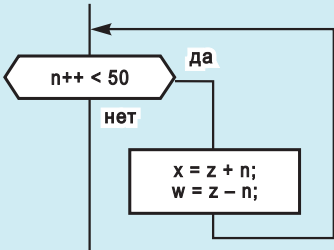
Операторы языка Си	Программы на языке Си	Программы на языке Дракон-Си
<div data-bbox="211 349 283 419">1</div> <div data-bbox="199 446 296 472">if-else</div>	<pre data-bbox="425 291 555 538"> if (a >= 0) { x = f1; y = f2; } else { x = r1; y = r2; } </pre>	
<div data-bbox="211 640 283 710">2</div> <div data-bbox="199 737 296 816">if- elseif- else</div>	<pre data-bbox="425 649 577 799"> if (x < b) x = b; elseif (x < c) x = c; else x = d; </pre>	
<div data-bbox="211 949 283 1019">3</div> <div data-bbox="199 1046 296 1160">switch, case, break, default</div>	<pre data-bbox="425 919 568 1192"> switch (n) { case 1: x = a; break; case 2: x = b; break; default: x = c; } /* switch */ </pre>	
<div data-bbox="211 1301 283 1372">4</div> <div data-bbox="211 1407 283 1434">while</div>	<pre data-bbox="400 1310 593 1442"> while (n++ < 50) { x = z + n; w = z - n; } /* while */ </pre>	

Рис. 166. Примеры программ на языке Си и эквивалентные им программы на языке Дракон-Си

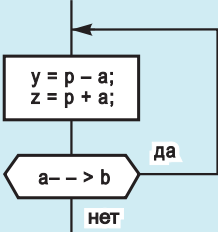
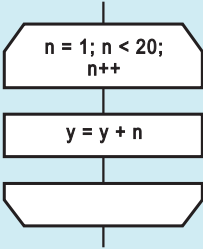
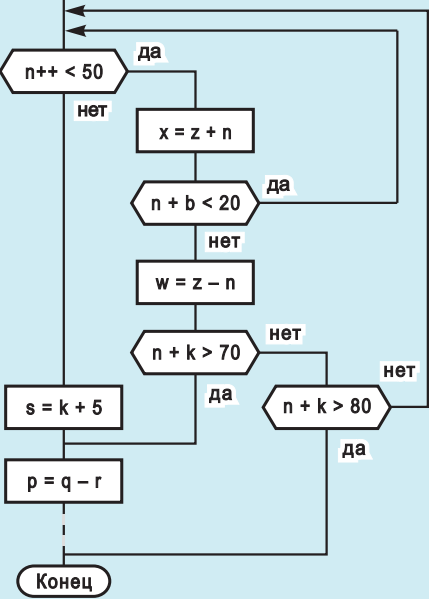
Операторы языка Си	Программы на языке Си	Программы на языке Дракон-Си
<div>5</div> <div>do-while</div>	<pre>do { y = p - a; z = p + a; } while (a-- > b); /* do while */</pre>	
<div>6</div> <div>for</div>	<pre>for (n=1; n<20; n++) y = y + n;</pre>	
<div>7</div> <div>while, continue, goto, return</div>	<pre>while (n++ < 50) { x = z + n; if (n + b < 20) continue; w = z - n; if (n + k > 70) go to m1; if (n + k > 80) return; } /* while */ s = k + 5; m1: p = q - r; } /* end */</pre>	

Рис. 167. Примеры программ на языке Си и эквивалентные им программы на языке Дракон-Си (продолжение)

Операторы присваивания, условные выражения и т. д. почти без изменения переносятся в визуальную программу и размещаются внутри ее икон.

Остальные текстоэлементы языка Си (которые можно называть удаляемыми или «паразитными») становятся ненужными. Они превращаются в графические линии и ключевые слова «да» и «нет» (*yes* и *no*).

Рис. 166 и 167 показывают, что список удаляемых (паразитных) элементов языка Си оказывается внушительным. Он включает все ключевые слова в примерах 1–7 кроме *default*, все фигурные, круглые и косые скобки, двоеточия, метки, комментарии в примерах 3–5, и, кроме того, точки с запятой в примерах 2, 3, 7 и отчасти 6.

§10. ПРОГРАММА ОБРАБОТКИ МАССИВОВ НА ЯЗЫКЕ ДРАКОН

На рис. 168 и 169 даны примеры программ, в которых имеются операции с массивами.

Для описания данных используется икона «полка». На верхнем этаже полки пишут ключевое слово «данные». На нижнем – собственно данные.

МАССИВ ВЕЩ Вес.кролика [100]

Эта запись означает, что задан одномерный массив с именем «Вес.кролика», содержащий 100 элементов, каждый из которых является вещественным числом (ВЕЩ).

Основным элементом обеих программ служит цикл ДЛЯ. В иконе «начало цикла ДЛЯ» в верхней строке пишут слово «Цикл» и после пробела односимвольное имя (алиас), обозначающее переменную цикла (буква k на рис. 168, 169).

В нижней строке указывают диапазон ее изменения, например,

от $k = 1$ до 100

В иконе «конец цикла ДЛЯ» делают запись

Конец цикла k

Или в общем виде

Конец цикла <переменная цикла>

Смысл операторов, организующих обработку массивов, пояснен в комментариях на рис. 168 и 169.

§11. ПЕРЕМЕННАЯ ЦИКЛА В ДРАКОН-ПРОГРАММЕ

Что означает переменная цикла k на рис. 168 и 169? Каков ее физический смысл в данной программе?

Многие программисты забывают (или не считают нужным) ответить на этот вопрос. В результате смысл буквы k остается неясным, а сама программа нередко становится непонятной.

Чтобы этого не случилось, можно применить эргономический прием.

В иконе «начало цикла ДЛЯ» в средней строке следует написать формализованный комментарий, например

$k \equiv \text{Номер.кроличьей.клетки}$

Знак тождественного равенства \equiv показывает, что после него следует имя-комментарий, то есть комментарий, который пишется по правилам записи идентификаторов. Эргономическое преимущество формализованного комментария включает два момента.

- Он позволяет устранить традиционную «забывчивость» программистов и доходчиво объяснить читателю смысл абстрактного идентификатора. Дескать, k – это номер кроличьей клетки.
- Объяснение размещается на поле чертежа именно там, где нужно (в иконе «начало цикла ДЛЯ»). Это значит, что читатель получает ответ моментально – в ту самую секунду, когда он впервые увидел алиас k и в его голове забрезжил вопрос: а что такое k ?

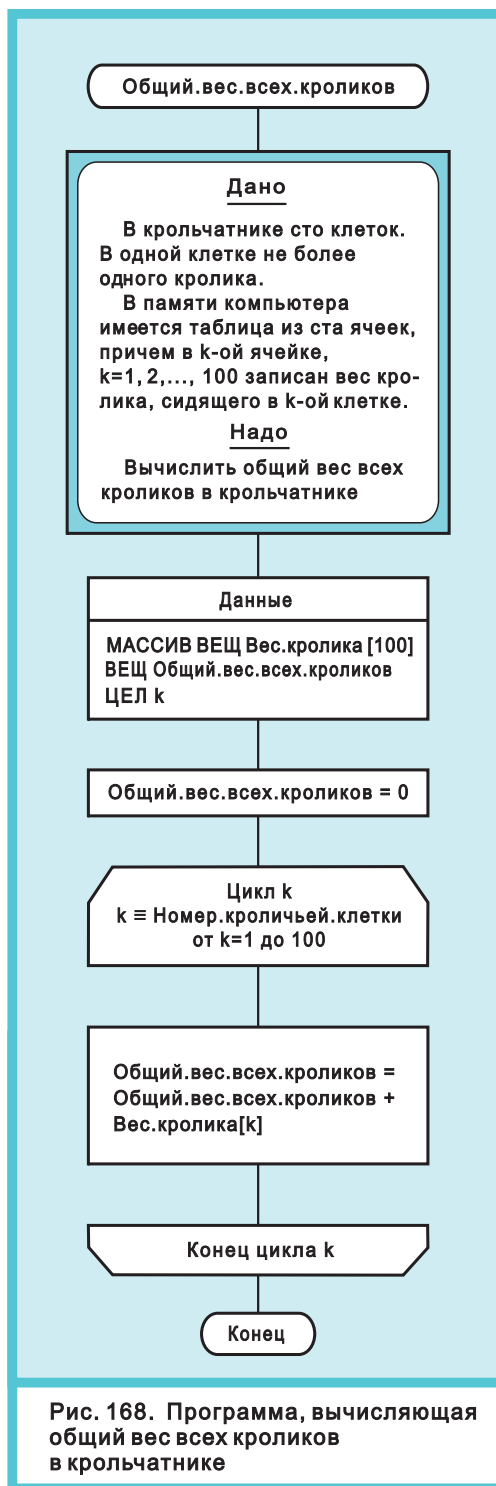
В итоге непонятный символ k превращается в понятный текст «Номер.кроличьей.клетки».

§12. СЕМЕЙСТВО ДРАКОН-ЯЗЫКОВ

Дракон – не один язык, а целое семейство, которое может включать практически неограниченное число дракон-языков. Все гибридные языки дракон-семейства имеют одинаковый графический синтаксис. Это обеспечивает зрительное сходство дракон-схем различных гибридных языков. Каждый гибридный язык семейства отличается тем, что имеет свой собственный текстовый синтаксис.

Строгое разграничение графического и текстового синтаксиса позволяет в максимальной степени расширить сферу применения гибридных языков, обеспечивая гибкость и универсальность выразительных средств языков семейства.

При этом единообразие правил графического синтаксиса гибридных языков обеспечивает их концептуальное единство. А разнообразие текстовых правил (то есть возможность выбора любого текстового синтаксиса) определяет гибкость языков и легкую настройку на различные предметные и иные области. См. также главу 6, §7.



§13. КАК ПОСТРОИТЬ ГИБРИДНЫЙ ЯЗЫК ПРОГРАММИРОВАНИЯ В ОБЩЕМ СЛУЧАЕ?

Чтобы построить гибридный язык, нужно выполнить 5 шагов.

Шаг 1. Выбрать целевой язык (например, язык Си).

Шаг 2. Использовать графический синтаксис языка Дракон в качестве графического синтаксиса гибридного языка Дракон-Си.

Шаг 3. Использовать синтаксис целевого языка (синтаксис языка Си) в качестве текстового синтаксиса гибридного языка Дракон-Си.

Шаг 4. Удалить из текстового синтаксиса гибридного языка Дракон-Си все элементы, которые заменяются управляющей графикой ДРАКОНа.

Шаг 5. Создать транслятор из дракон-схемы в исходный код языка Си.

Примечание. Язык Си выбран для примера. Вместо него можно подставить любой целевой язык.

При использовании гибридных языков исходным текстом программы считается дракон-схема и только она.

Пример. Предположим, пользователь работает в связке *Дракон-конструктор – Транслятор Дракон-Си – Keil*. Понятно, что исходником служит дракон-схема. При отладке программы не следует вносить исправления в промежуточные текстовые Си-файлы. Все исправления нужно вносить в исходный код, то есть в дракон-схему.

Петр Приклонский сообщает:

«Я уже больше года работаю на связке *Дракон-конструктор – DrakonToC – Keil*. И ни в коем случае не позволяю себе править промежуточные текстовые Си-файлы. Исходник – это дракон-схема!» [1]

Подробнее см. [2].

§14. ДВА ЭТАПА РАЗВИТИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

С точки зрения человеческого фактора, в истории развития языков программирования можно выделить два этапа.

На первом этапе появились языки высокого уровня, которые (по сравнению с ассемблером) сделали исходный текст программы более понятным и удобным для человека. И значительно увеличили производительность труда программистов.

На втором этапе (который, по-видимому, только начинается) некоторые языки высокого уровня смогут работать в сочетании с языком Дракон, образуя гибридные языки. При этом функция исходного кода программы переходит к дракон-схемам.

Это позволит отказаться от текстовых управляющих структур, используемых в языках высокого уровня, и заменить их на управляющую графику Дракона.

Что это даст? Исходный код программы станет еще более понятным и удобным для человека. И, следовательно, еще больше увеличится производительность труда программистов.

§15. ПЛАН РАЗВИТИЯ И ЧАСТИЧНОЙ УНИФИКАЦИИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Опыт использования языка Дракон и гибридных языков позволяет предложить план развития и частичной унификации языков высокого уровня из трех пунктов.

1. Использовать графический синтаксис языка Дракон в качестве стандарта, позволяющего осуществить частичную унификацию языков высокого уровня.
2. Текстовый синтаксис следует заимствовать из целевого языка. При этом следует удалить все элементы текстового синтаксиса, которые заменяются управляющей графикой Дракона.
3. Преобразовать языки высокого уровня в гибридные языки.

Как показывают первые опыты подобной работы, переход от языков высокого уровня к гибридным языкам программирования свидетельствует о заметном повышении производительности труда программистов.

§16. ВЫВОДЫ

1. Язык Дракон можно присоединить к некоторым языкам программирования и получить так называемые гибридные языки:

язык Дракон + язык Си	= гибридный язык Дракон-Си
язык Дракон + язык Дельфи	= гибридный язык Дракон-Дельфи
язык Дракон + язык Джава	= гибридный язык Дракон-Джава
язык Дракон + язык Си#	= гибридный язык Дракон-Си#
язык Дракон + язык Питон	= гибридный язык Дракон-Питон
язык Дракон + язык Перл	= гибридный язык Дракон-Перл
язык Дракон + язык Руби	= гибридный язык Дракон-Руби
язык Дракон + язык Ада	= гибридный язык Дракон-Ада
язык Дракон + язык Tcl	= гибридный язык Дракон-Tcl
язык Дракон + язык Оберон	= гибридный язык Дракон-Оберон
язык Дракон + язык Ассемблер	= гибридный язык Дракон-Ассемблер
и т. д.	

2. Гибридный язык почти полностью сохраняет концепцию, структуру, типы данных и другие особенности целевого языка. В строго определенном числе случаев текстовая нотация целевого языка заменяется на графическую нотацию Дракона. Такой прием позволяет улучшить эргономический облик гибридного языка и повысить производительность труда программистов.

Часть III

**АЛГОРИТМЫ
ПРАКТИЧЕСКОЙ ЖИЗНИ
(ПРИМЕРЫ)**

АЛГОРИТМЫ В МЕДИЦИНЕ

§1. ЯЗЫК ДРАКОН ПОЗВОЛЯЕТ ПРЕДСТАВЛЯТЬ ПРОЦЕДУРНЫЕ МЕДИЦИНСКИЕ ЗНАНИЯ В УДОБНОЙ ФОРМЕ

В предыдущей части (главы 1–16) мы вкратце познакомились с языком ДРАКОН.

В этой части (главы 17–25) перейдем к практике. Мы рассмотрим большое число примеров из самых различных областей деятельности. И покажем, что ДРАКОН позволяет наглядно изображать алгоритмы, можно сказать, в любых ситуациях.

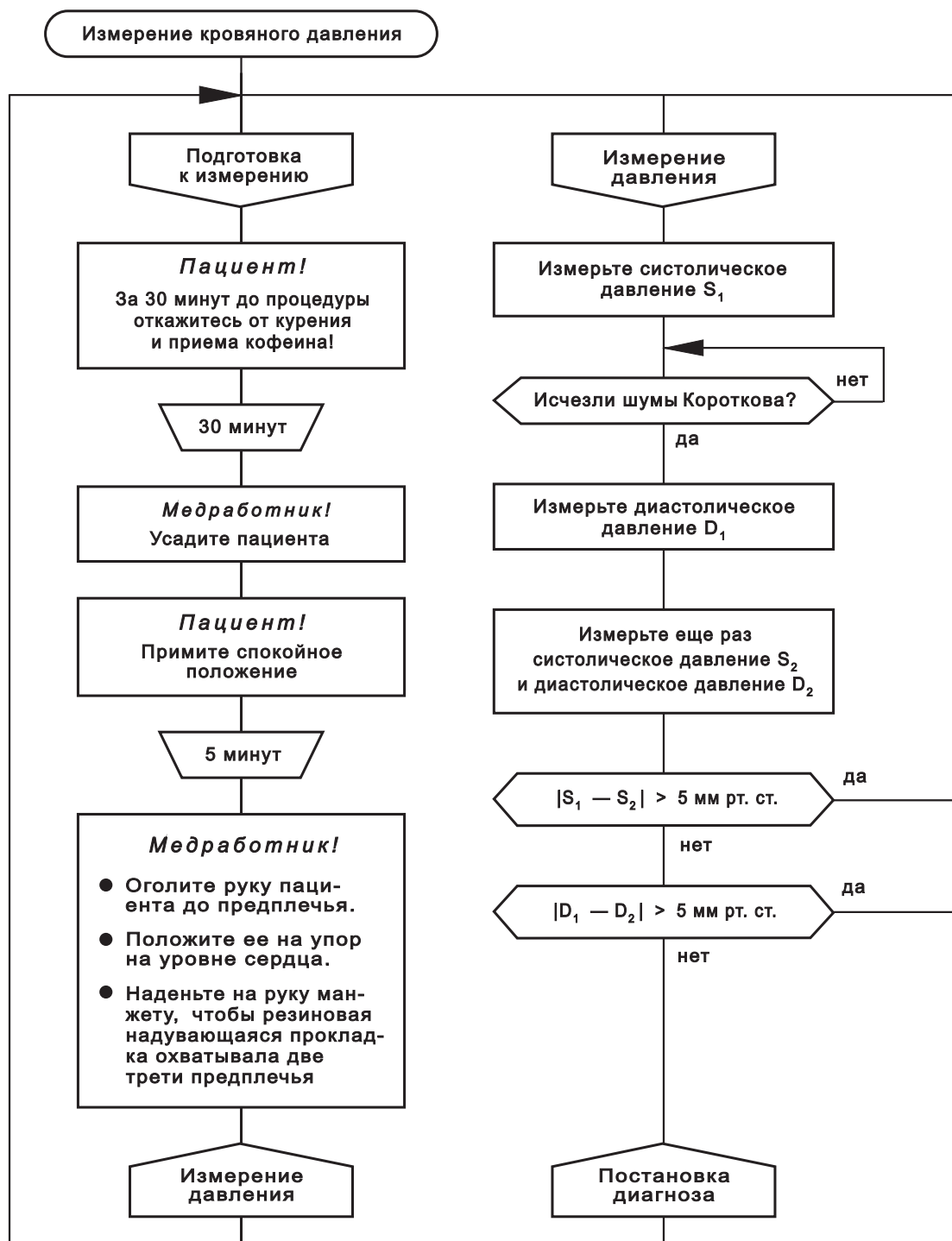
Что это дает? Поскольку процесс создания алгоритмов оказывается чрезвычайно легким, он становится доступным практически для любого человека.

С появлением ДРАКОНа специалисты приобретают новые возможности:

- они могут выражать свои знания на своем родном профессиональном языке, но в формализованном графическом виде;
- они могут рисовать алгоритмы с большой скоростью, которая раньше считалась невозможной;
- мысли одного специалиста благодаря ДРАКОНу становятся понятными другим специалистам.

В итоге люди получают мощное средство профессионального общения.

Начнем с медицины. И покажем, что ДРАКОН легко отображает процедурные медицинские задачи (медицинские алгоритмы).



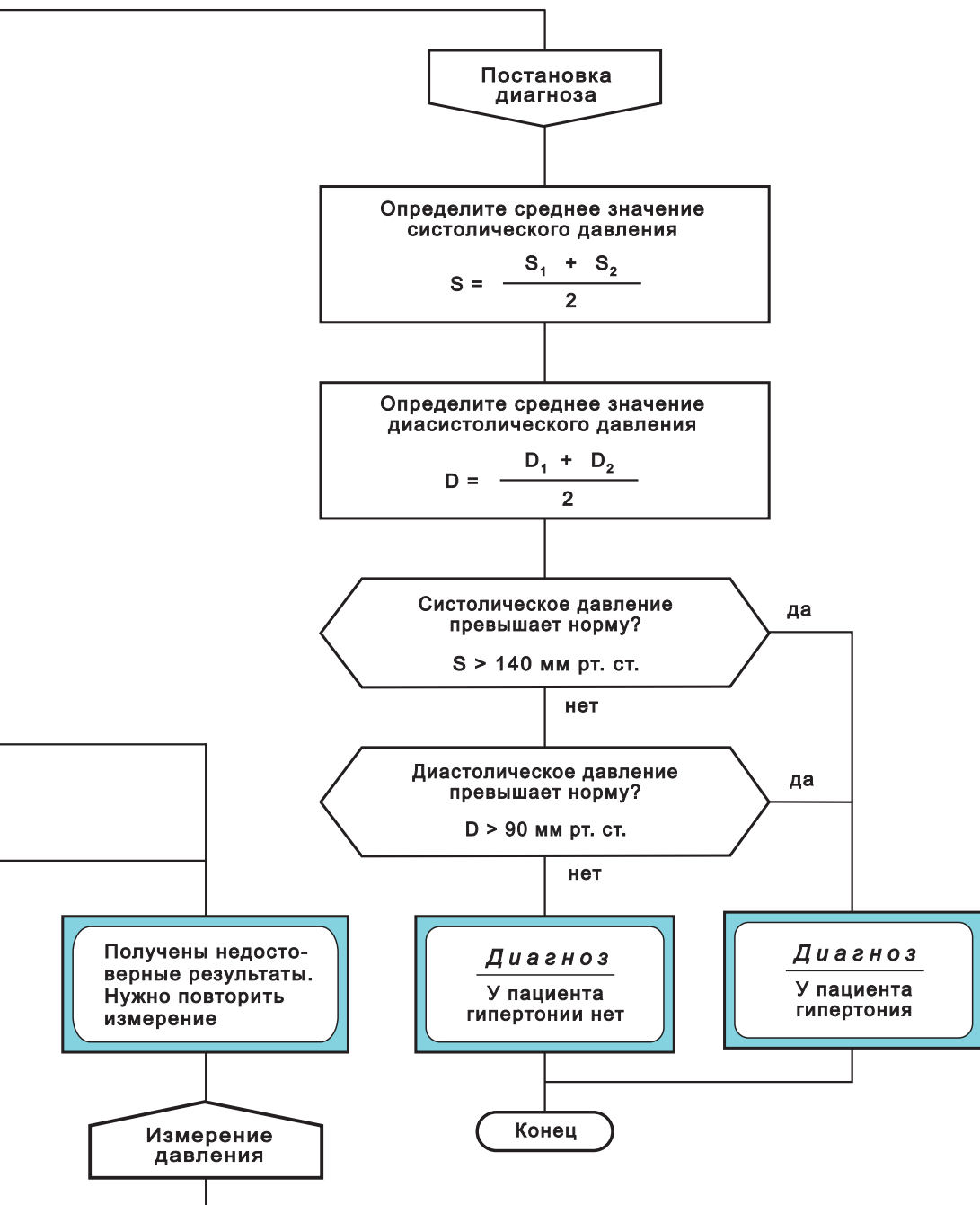
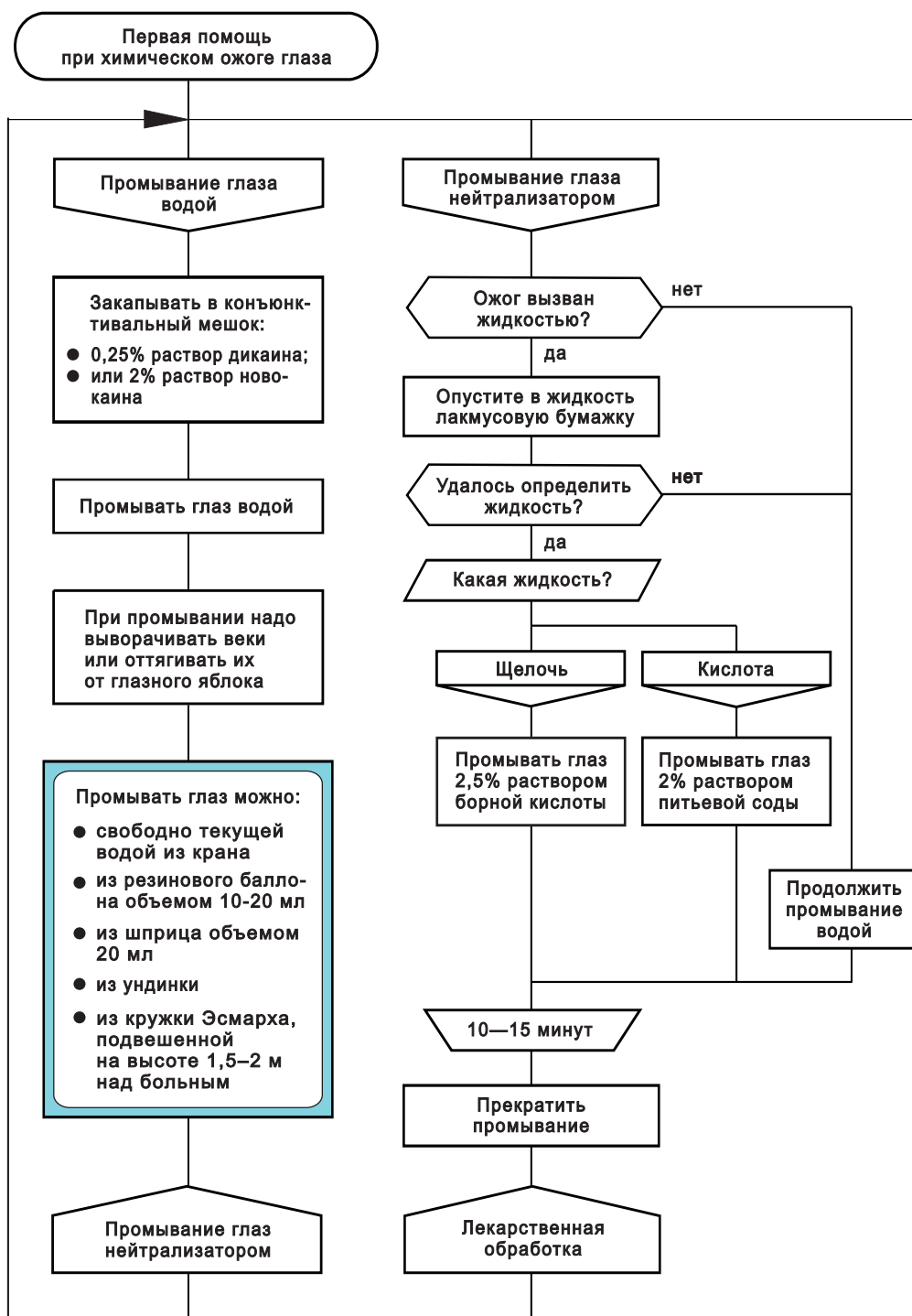


Рис. 170. Алгоритм «Измерение кровяного давления»



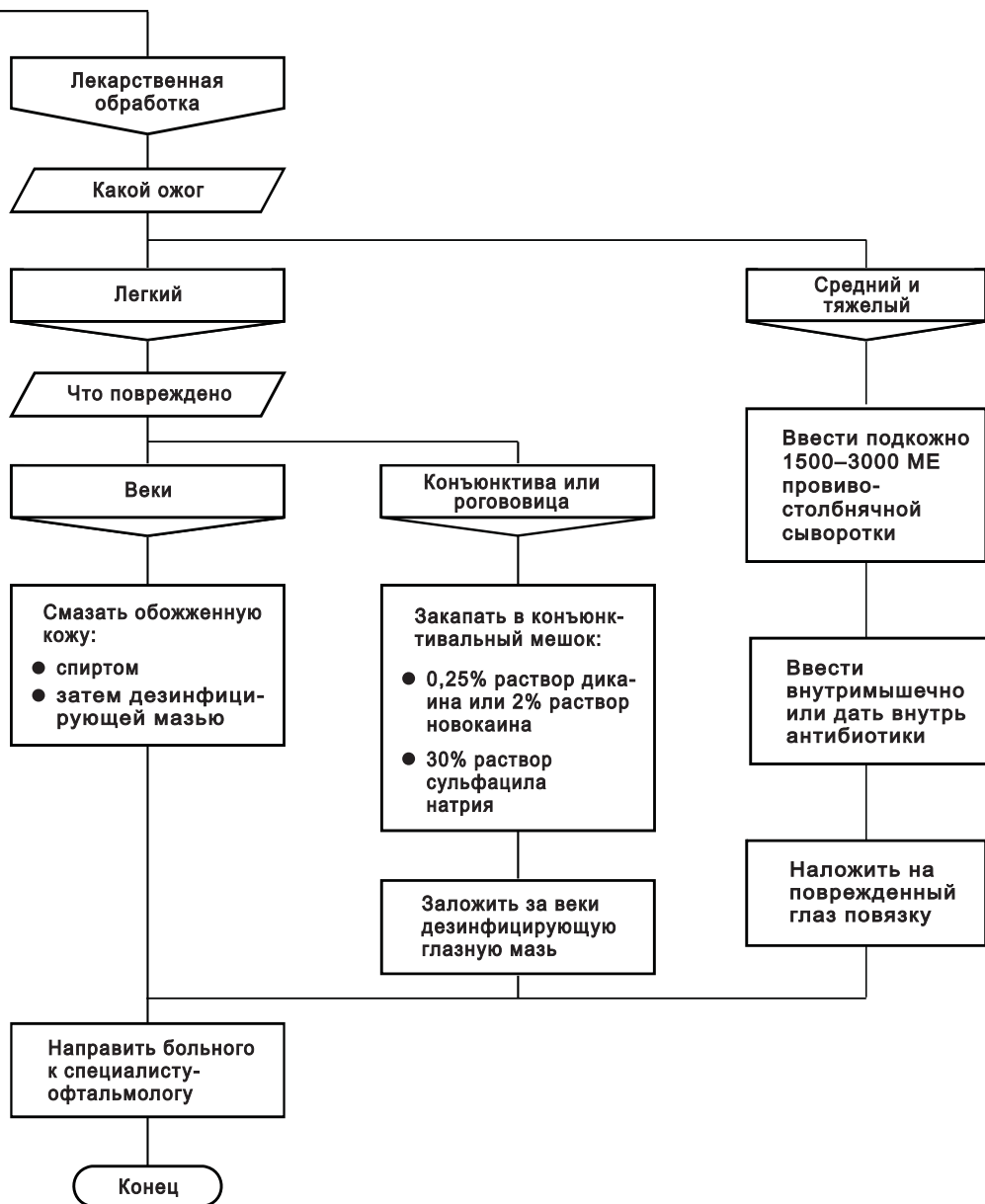


Рис. 171. Первая помощь при химическом ожоге глаза

§2. ИЗМЕРЕНИЕ КРОВЯНОГО ДАВЛЕНИЯ

Медики редко произносят слово «алгоритм». А жаль! – ведь алгоритмы составляют значительную часть медицинских знаний.

На рис. 170 представлен знакомый почти каждому пример – измерение кровяного давления.

Пояснение

Артериальное давление – это давление внутри кровеносных сосудов (артерий), обеспечивающих продвижение крови по кровеносной системе. Оно измеряется в миллиметрах ртутного столба.

Артериальное давление считается нормальным, если верхнее (систолическое) давление равно 120 мм. ртутного столба. А нижнее (диастолическое) давление равно 80 мм. ртутного столба.

Если давление систематически повышается, это плохо для здоровья. Давление считается повышенным, когда верхнее давление превышает 140 мм. рт. столба. Или когда нижнее давление превышает 90 мм. рт. столба.

Алгоритм на рис. 170 получен путем точного воспроизведения инструкции для врачей, подготовленной комитетом США по профилактической медицине [1].

Обычной практикой является однократное измерение артериального давления. Но американские медики рекомендуют измерять давление не один, а два раза.

Если разница между двумя измерениями меньше 5 мм рт. столба, результат считается достоверным. Если же разница превышает указанную величину, врачи рекомендуют аннулировать результат, как недостоверный. И повторить измерение еще раз.

Медицинский алгоритм на рис. 170 в точности отражает американские рекомендации.

§3. ПЕРВАЯ ПОМОЩЬ ПРИ ХИМИЧЕСКОМ ОЖОГЕ ГЛАЗА

Ожог – это повреждение тканей, вызванное тепловым, химическим, электрическим или радиационным воздействием.

Химические ожоги органов зрения вызваны прямым действием химических веществ: кислот, щелочей и других химических агентов (клеи, красители и пр.). В общем случае ожоги глаз могут быть вызваны как твердыми веществами, так и жидкостями.

На рис. 171 изображена первая помощь при химическом ожоге глаз. Дракон-схема составлена на основании «Практического руководства для врачей общей (семейной) практики» [2].

Деление алгоритма на три ветки показывает, что первая помощь при химическом ожоге состоит из трех этапов:

- промывание глаз водой;
- промывание глаз нейтрализатором;
- лекарственная обработка.

Содержание каждой ветки позволяет дать целостную картину проблемы. И, сверх того, указать точную последовательность действий, выполняемых при оказании первой помощи пострадавшему глазу.

§4. АЛГОРИТМЫ В МЕДИЦИНЕ

Процесс обследования, диагностики и лечения часто представляет собой некоторую последовательность действий. Следовательно, его можно рассматривать как алгоритм.

Алгоритмические описания часто встречаются во многих медицинских руководствах. Сюда относятся описания иммунологических методов, клиническая лабораторная диагностика, микробиологические инструкции по идентификации микроорганизмов и многое другое.

К сожалению, форма представления этих алгоритмов далека от совершенства. Этот недостаток вносит серьезные затруднения в процесс распространения медицинских знаний. И неблагоприятно отражается на разработке новых методов лечения.

Можно предположить, что учебные альбомы и компьютерные библиотеки медицинских алгоритмов (дракон-схем) могли бы принести ощутимую пользу в медицинских научных исследованиях. А также в системе медицинского образования и во врачебной практике.

Кроме того, дракон-схемы могут существенно облегчить взаимопонимание медиков между собой и со специалистами смежных профессий. В частности, с медицинскими программистами.

§5. ВЫВОДЫ

1. Важным недостатком современной медицины является отсутствие эффективных способов описания *процедурных* медицинских знаний. *Процедурные* знания — это знания о точной последовательности процессов и действий, например:
 - процессов, протекающих в организме человека,
 - действий, выполняемых медицинским персоналом.
2. В медицинской литературе доминирует текст и некачественные рисунки. Правила создания подобных рисунков достались врачам в наследство от предыдущих эпох, что существенно тормозит фиксацию, понимание и передачу медицинских знаний.

3. Чтобы сделать новый мощный прорыв в развитии медицины, необходимы не только новые медицинские знания, но и новые способы описания знаний.
4. Визуализация медицинских знаний и разработка новых методов описания знаний представляют собой самостоятельное направление научных исследований в области медицины.
5. Язык ДРАКОН – новое средство, пригодное для описания структуры медицинской деятельности и медицинских алгоритмов. Оно позволяет выявить и обнажить логические инварианты деятельности, сделать их явными, зримыми, доступными для всех врачей и студентов-медиков.
6. Язык ДРАКОН кардинальным образом облегчает труд формализации процедурных медицинских знаний и повышает его производительность.
7. Широкомасштабное внедрение языка ДРАКОН для описания процедурных медицинских знаний позволит сделать эти знания намного более ясными, понятными, доходчивыми. Не исключено, что продуманное использование языка ДРАКОН в перспективе позволит сократить сроки медицинского образования. И получить другие важные преимущества.

АЛГОРИТМЫ В ПРОМЫШЛЕННОСТИ

§1. ДВА ТИПА АЛГОРИТМОВ

Современная промышленность – царство алгоритмов. Эти алгоритмы можно разделить на две части:

- выполняемые автоматически с помощью компьютеров,
- выполняемые вручную или с помощью различных механизмов. (Имеются в виду алгоритмы, описывающие структуру человеческой деятельности и технологических процессов).

§2. АЛГОРИТМ И ДЕЯТЕЛЬНОСТЬ. В ЧЕМ СХОДСТВО И В ЧЕМ РАЗЛИЧИЕ?

Математические алгоритмы и человеческая деятельность не разделены китайской стеной. У них есть нечто общее. Но что именно?

Не претендуя на строгость (в данном случае она не нужна), можно сделать два замечания.

Алгоритм

Это последовательность информационных действий, ведущая к поставленной цели

Деятельность

Это последовательность информационных и физических действий, ведущая к поставленной цели

Таким образом, отличие состоит в том, что в алгоритме физические действия являются запрещенными, а в деятельности – разрешенными. Примерами физических действий служат: транспортировка груза, нагрев детали, пуск ракеты, зашивание раны и т. д.

§3. КЛАССИЧЕСКИЕ АЛГОРИТМЫ

Сначала рассмотрим классические (математические) алгоритмы, затем – неклассические алгоритмы, описывающие структуру деятельности.

На рис. 90 и 91 изображены математические алгоритмы. Они вычисляют факториал. На рис. 90 факториал вычисляется с помощью цикла ДО. На рис. 91 – с помощью цикла ДЛЯ.

На рис. 140 представлен алгоритм «Управление светофором». В данном случае результатом алгоритма являются не числа, а процесс управления уличным движением. Если светофор установлен на бойком месте, он должен работать круглосуточно, не останавливаясь ни на секунду. Поэтому управляющий светофором компьютер и «живущий» в нем алгоритм работают непрерывно, безостановочно.

На рис. 148 показан еще один алгоритм, который «замурован» в крохотном компьютере, спрятанном в электронных ручных часах. Он вычисляет время в часах, минутах и секундах. Этот алгоритм постоянно обновляет результаты вычислений, чтобы часы не отстали от жизни.

§4. НЕКЛАССИЧЕСКИЕ АЛГОРИТМЫ

Перейдем к анализу неклассических алгоритмов. На рис. 172 изображена проверка самолета. При такой проверке часть операций выполняется автоматически, часть – вручную.

Является ли проверка самолета алгоритмом? С классической точки зрения – нет. Потому что кое-что делается вручную. Примером таких операций на рис. 172 являются ремонтные работы:

- ремонт двигателей;
- ремонт крыльев;
- ремонт шасси
- ремонт топливной системы;
- ремонт бортовых систем;
- ремонт системы пожаротушения;
- ремонт электрооборудования.

- Возникает противоречие. С одной стороны, необходимо иметь *целостную* картину проверки самолета, включающую все операции: и автоматические, и ручные.
- С другой стороны, программисты обычно исключают из рассмотрения ручные операции. Потому что компьютер не участвует в их реализации.

В итоге алгоритмисты и программисты делают себя «полуслепыми», отказываясь анализировать ручные операции, без которых работа промышленности становится невозможной.

Почему алгоритмисты допускают эту ошибку? Потому что они привыкли иметь дело только с классическими алгоритмами.

Как разрешить противоречие? Надо расширить понятие алгоритма, чтобы оно охватывало не только компьютерные операции, но и человеческую деятельность.

Еще один пример. На рис. 174 изображена технология изготовления сиропа и маринада. Здесь показаны не информационные, а физические действия. Их выполняет не компьютер, а различные технологические установки и агрегаты: мешкоопрокидыватель, вибросито, магнитная ловушка, бункер-накопитель и т. д.

Технология, показанная на рис. 174, спроектирована, реализована и эксплуатируется человеком. Следовательно, она является *человеческой деятельностью*. С точки зрения математики, схема на рис. 174 не является алгоритмом, ибо она не удовлетворяет классическому определению алгоритма [1]. Чтобы избежать путаницы, мы называем подобные схемы неклассическими алгоритмами.

§5. ИЗГОТОВЛЕНИЕ ФРУКТОВЫХ КОНСЕРВОВ

На рис. 174 представлен технологический процесс изготовления фруктовых консервов из косточковых плодов.

Эргономичная графика обладает серьезными преимуществами по сравнению с текстовым описанием технологических процессов.

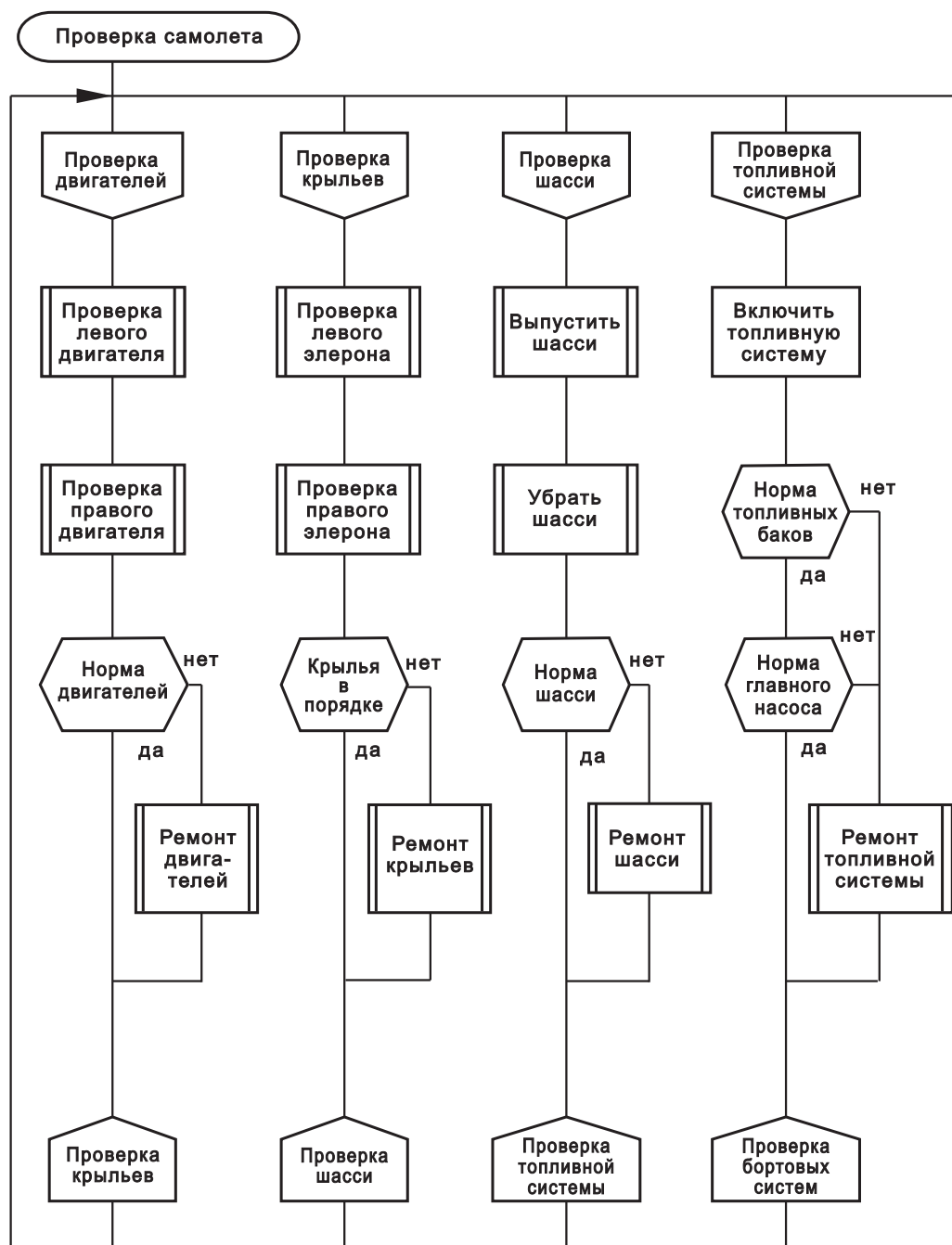
Она позволяет за короткое время (буквально за считанные минуты) составить подробное представление о технологических операциях и последовательности их выполнения.

Реальный технологический процесс может быть очень сложным. Обычно его описывают как головной процесс, содержащий большое число вставок. Например, в процессе на рис. 173 показана вставка «Изготовление сиропа и маринада», раскрытая на рис. 174.

§6. ДРАКОН И ТЕХНОЛОГИЧЕСКИЕ ПРОЦЕССЫ

Дракон-схемы технологических процессов могут найти применение в следующих случаях:

- создание наглядных плакатов, дающих целостное представление о процессе во всей его многосложности и используемых в качестве демонстрационных материалов. При этом в иконе «комментарий» могут помещаться чертежи, фотографии, схемы установок, станков, сетей трубопроводов и другого оборудования;
- выпуск технологической документации;
- проектирование и моделирование технологических процессов;
- создание визуальной базы данных о техпроцессах;



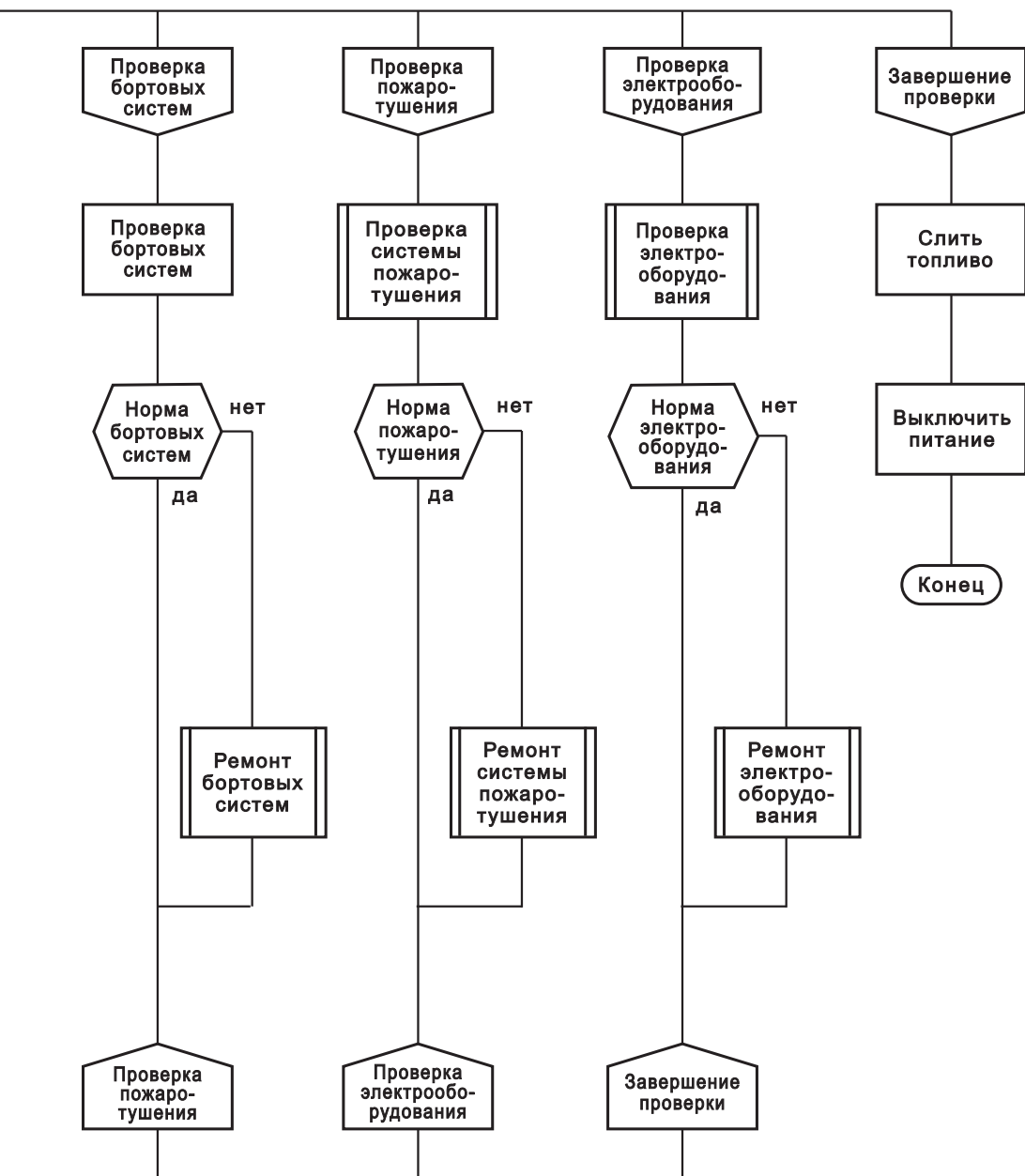


Рис. 172. Алгоритм «Проверка самолета»

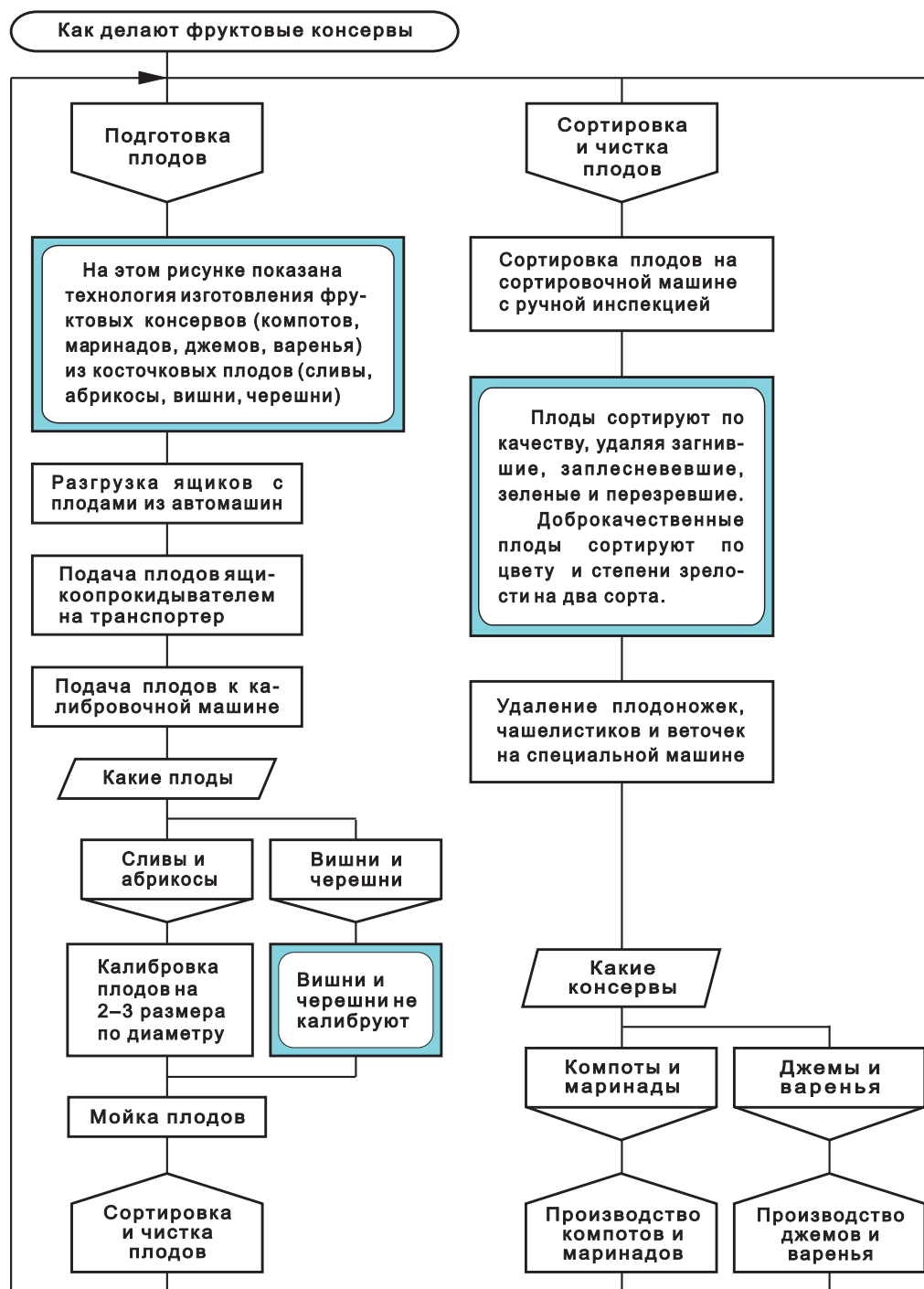
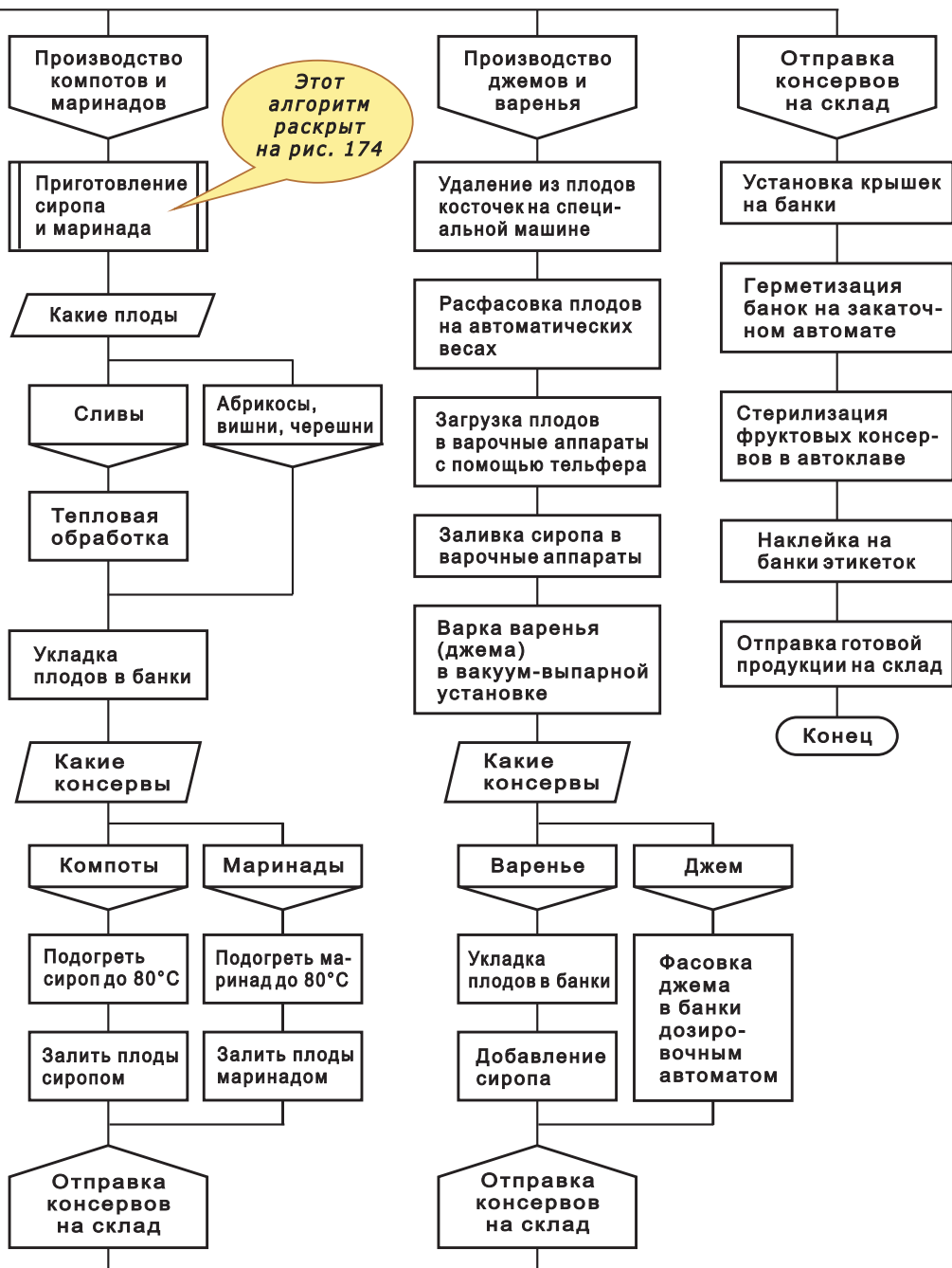


Рис. 173. Технология изготовления фруктовых консервов



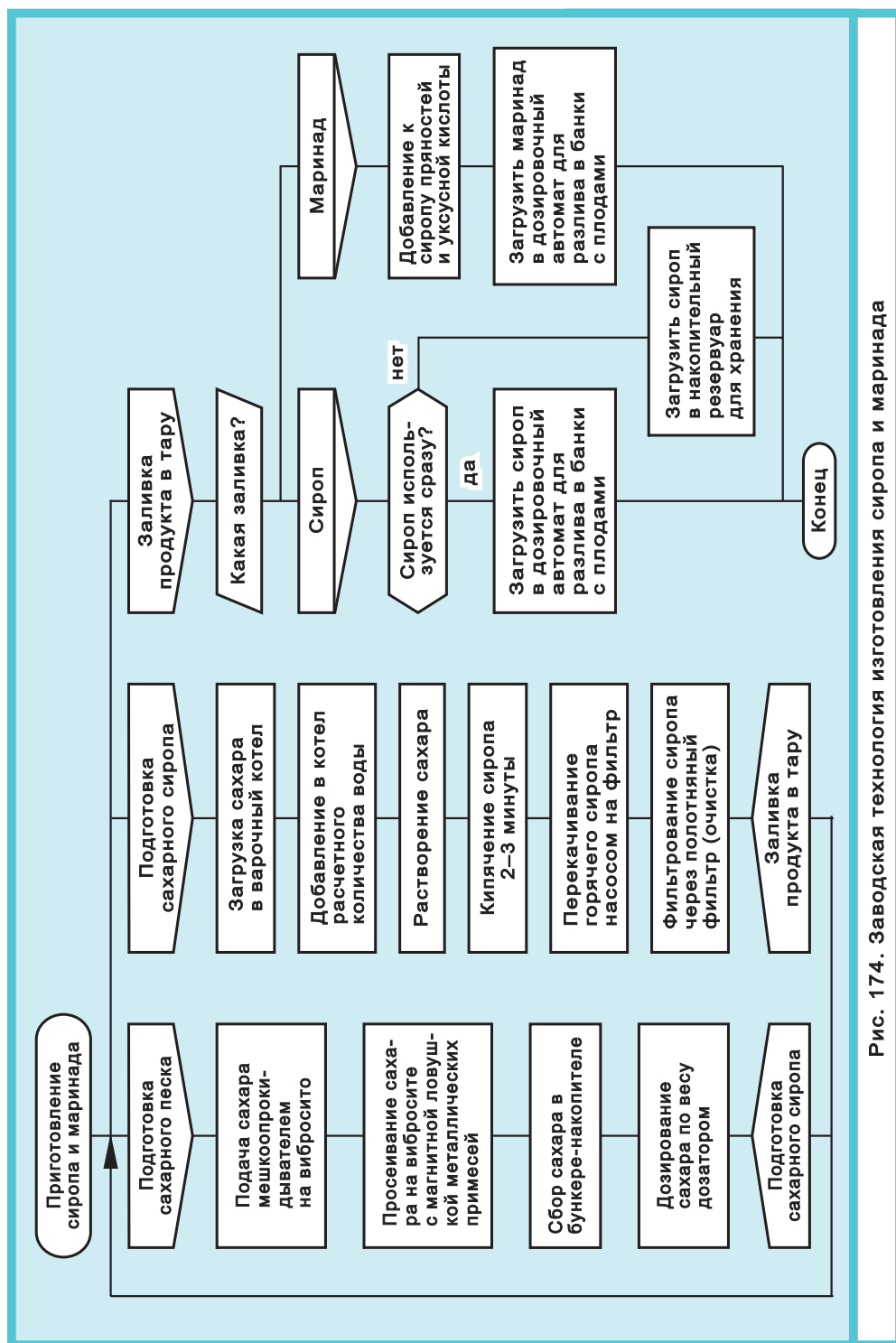


Рис. 174. Заводская технология изготовления сиропа и маринада

- создание экспертных систем для проектирования технологических процессов, а также тренажеров для эксплуатационников;
- изготовление альбомов и каталогов технологических процессов для обучения или рекламы. Можно рекомендовать формат бумажной страницы альбома А3, имея в виду, что оригинал-макеты альбомов готовятся на принтере формата А3.

§7. ВЫВОДЫ

1. Человеческие знания делятся на процедурные и декларативные.
2. Процедурные знания тесно связаны с человеческой деятельностью и трудовыми процессами. Они выявляют, закрепляют в сознании и объективируют структуру деятельности.
3. В настоящее время отсутствуют эффективные и эргономичные способы описания человеческой деятельности (работы). Язык ДРАКОН позволяет устранить этот пробел.
4. Процедурные знания охватывают классические и неклассические алгоритмы.
5. Язык ДРАКОН позволяет единообразно, стандартным способом описать оба типа алгоритмов (классические и неклассические).
6. При проектировании сложных промышленных объектов желательно и даже обязательно иметь *целостный* взгляд на проблему, показывающий все ее аспекты. На этом пути возникает трудность. В силу сложившихся привычек программисты обычно делят проблему на две части:
 - алгоритм, который можно превратить в компьютерную программу;
 - операции, выполняемые вручную или с помощью механизмов. Последние невозможно превратить в программу. Поэтому многие программисты считают, что эта часть дела их не касается.
7. В результате у проектировщиков сложных промышленных объектов нередко возникает не *целостный* взгляд на проблему, а *кусочно-рванный*, что порождает неоптимальные или ошибочные решения.
8. Одной из причин этого недостатка является отсутствие языковых средств, помогающих разработчикам вырабатывать целостное видение проблемы.
9. Язык ДРАКОН позволяет устранить этот недостаток. Он дает разработчикам надежные средства, позволяющие изображать не кусочно-рваную, а целостную картину любых промышленных (и не только промышленных) алгоритмических процессов.

АЛГОРИТМЫ В ТОРГОВЛЕ

§1. РАЗНООБРАЗИЕ ТОРГОВЫХ АЛГОРИТМОВ

В торговле используется огромное количество разнообразных алгоритмов. Примерами являются алгоритмы оптовой и розничной торговли для частных и бюджетных предприятий, бухгалтерские расчеты, управление складскими операциями. Сюда же можно отнести обработку платежных поручений, расчет зарплаты, управление персоналом и многое другое.

Все эти операции можно изобразить на языке ДРАКОН.

ДРАКОН представляет процедурные аспекты торговой деятельности в наиболее ясной и понятной форме. Понятной всем участникам торгового процесса – от рядового бухгалтера до руководителя торговой фирмы.

§2. ПРОДАЖА ДЕТСКИХ ИГРУШЕК

На рис. 175 представлен алгоритм «Продажа детских игрушек». Суть дела поясним на примере. Оптовый покупатель желает закупить 1000 детских игрушек, например, куклу Барби. Возможны три ситуации:

- на складе продавца 5000 кукол Барби. В этом случае заявка оптовика будет удовлетворена полностью;
- на складе, увы, всего лишь 90 кукол Барби. Тут возможны варианты. Оптовик либо берет то, что есть (покупает 90 кукол), либо отказывается от товара;
- на складе нет ни одной куклы. Стало быть, сделка сорвалась.

Силуэт состоит из четырех веток:

- оформление заказа;
- выставление счета;
- отмена заказа на игрушки;
- завершение.

Первая ветка (оформление заказа) полностью отражает описанные выше ситуации. Она имеет три иконы адрес:

- выставление счета;
- отмена заказа на игрушки;
- завершение.

Вспомним рекомендацию из §5 главы 6:

Желательно, чтобы порядок расположения икон «адрес» в многоадресной ветке совпадал с пространственным расположением веток, на которые ссылаются эти адреса.

На рис. 175 эта рекомендация выполнена. Действительно, порядок расположения икон адрес в первой ветке совпадает с расположением веток, на которые ссылаются эти адреса.

Вторая ветка (выставление счета) содержит действия:

- рассчитать, сколько стоит заказ;
- выставить счет покупателю;
- покупатель согласен оплатить счет?;
- и т. д.

Третья говорит об отмене заказа на игрушки.

§3. ПРОДАЖА АВИАБИЛЕТОВ

В следующем алгоритме речь идет о продаже авиабилетов (рис. 176).

Данный алгоритм отчасти похож на предыдущий. Отличие в том, что продажа игрушек описана упрощенно (популярно), а продажа авиабилетов изложена профессиональным языком. Эта разница хорошо видна в таблице.

Продажа игрушек	Продажа авиабилетов
Дать заявку на склад на нужное число игрушек	Дать заявку на нужное число билетов
Нужное число игрушек есть на складе?	Нужное число билетов зарезервировано?
На складе есть хоть сколько-то игрушек?	Число билетов > 0 ?
Сообщить клиенту, что игрушек очень мало	Сообщить клиенту о частичном выполнении заявки
Клиент согласен купить то, что есть?	Клиент согласен на частичное выполнение заявки?
Отменить заказ на игрушки	Отменить резервирование билетов

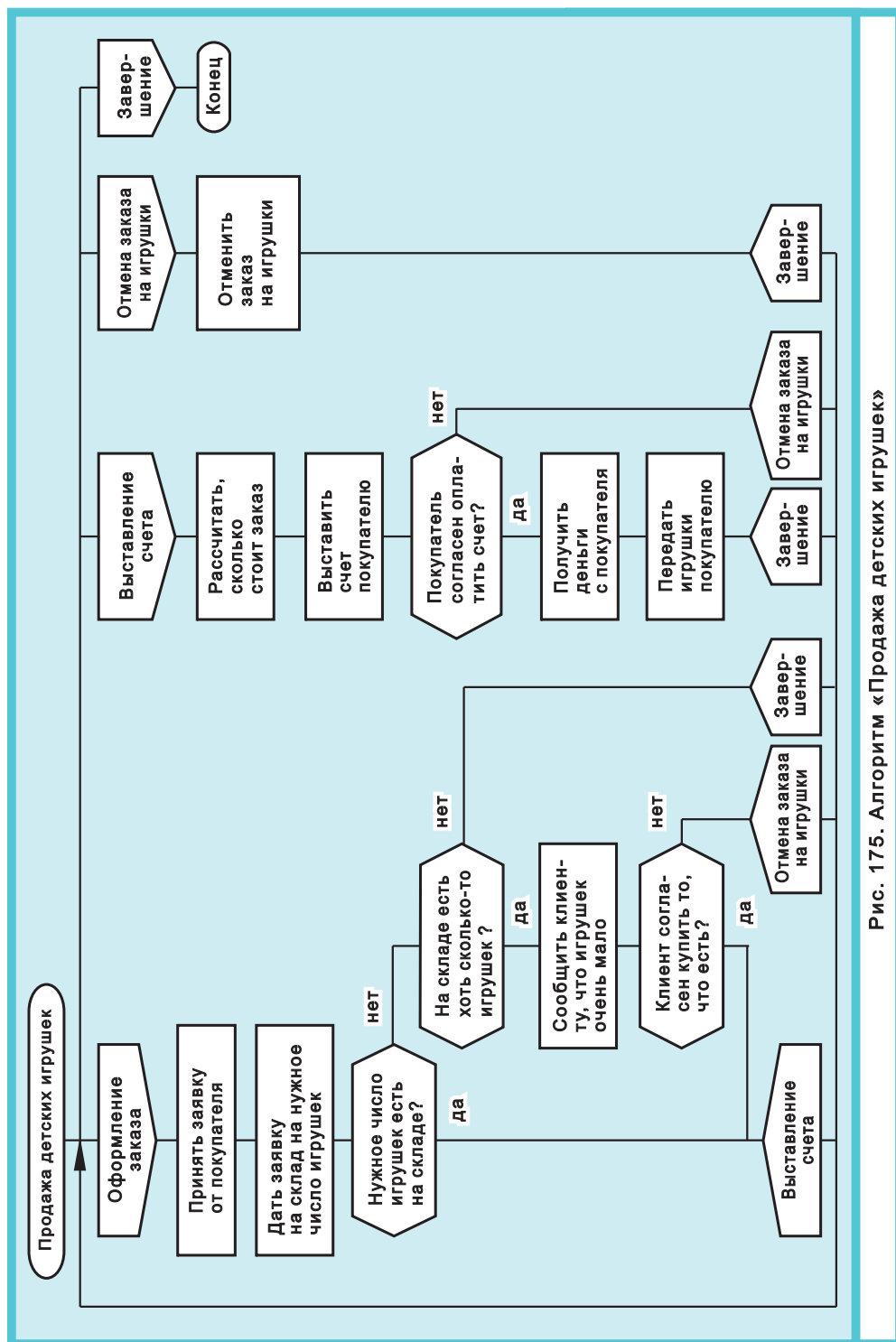


Рис. 175. Алгоритм «Продажа детских игрушек»

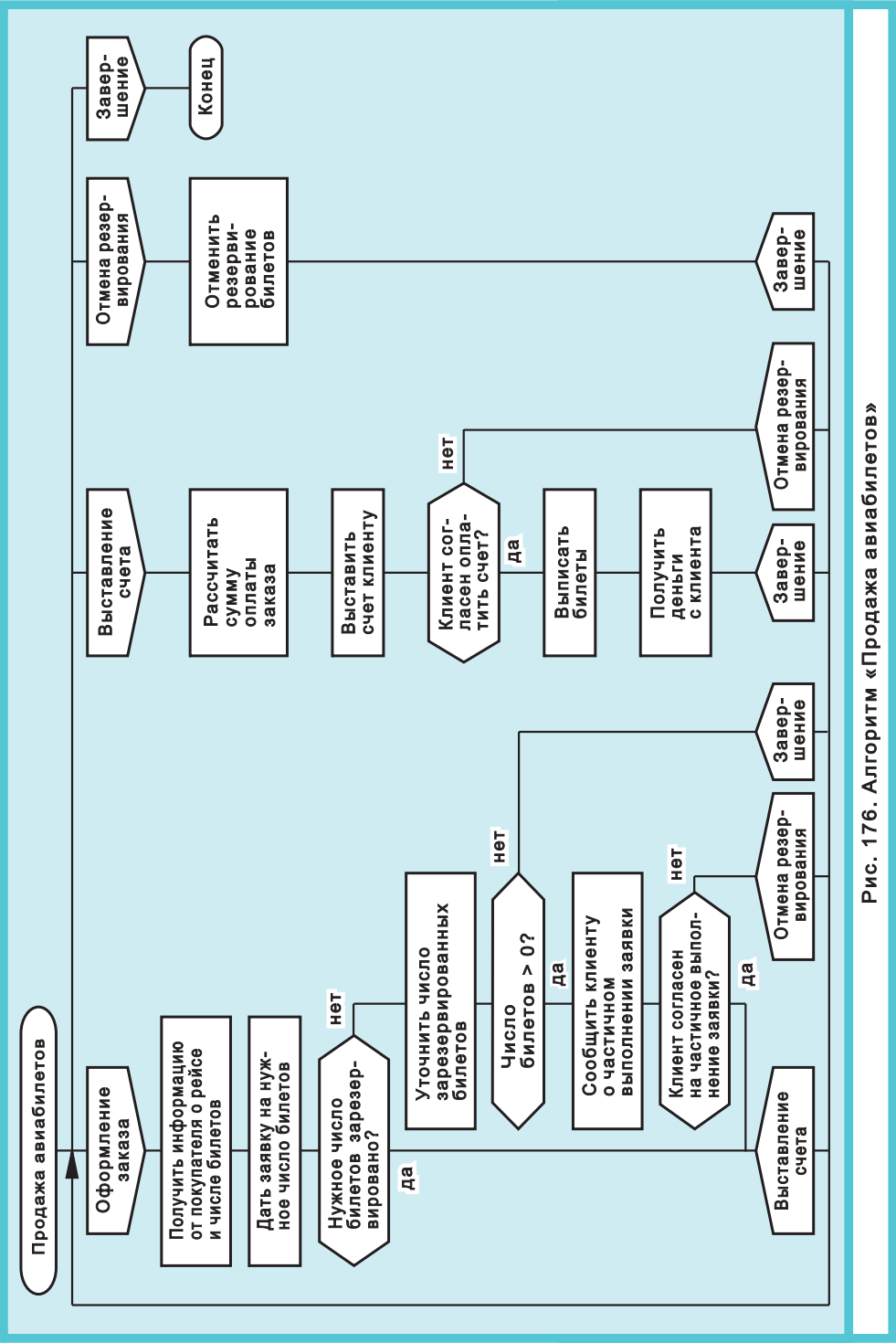
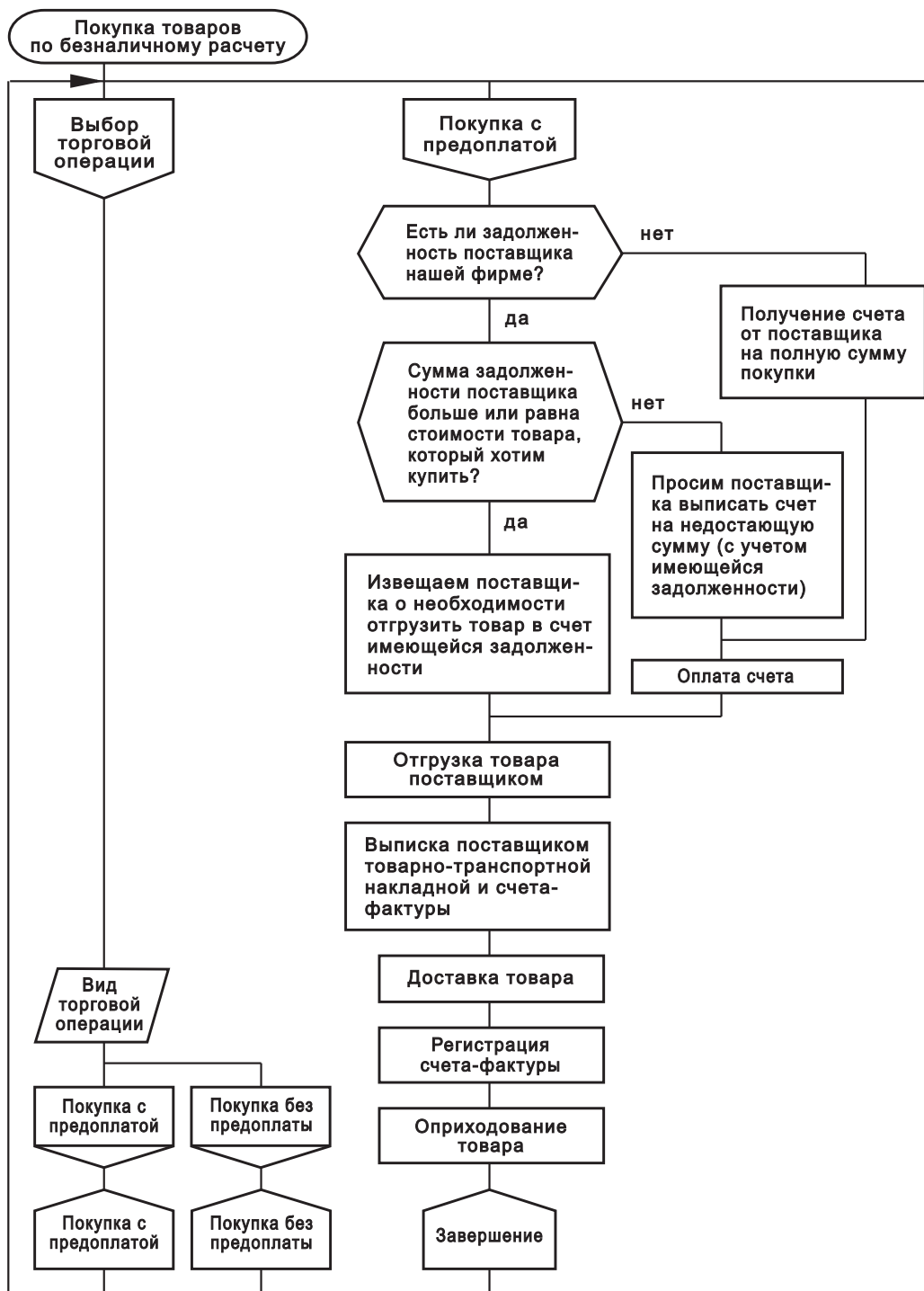


Рис. 176. Алгоритм «Продажа авиабилетов»



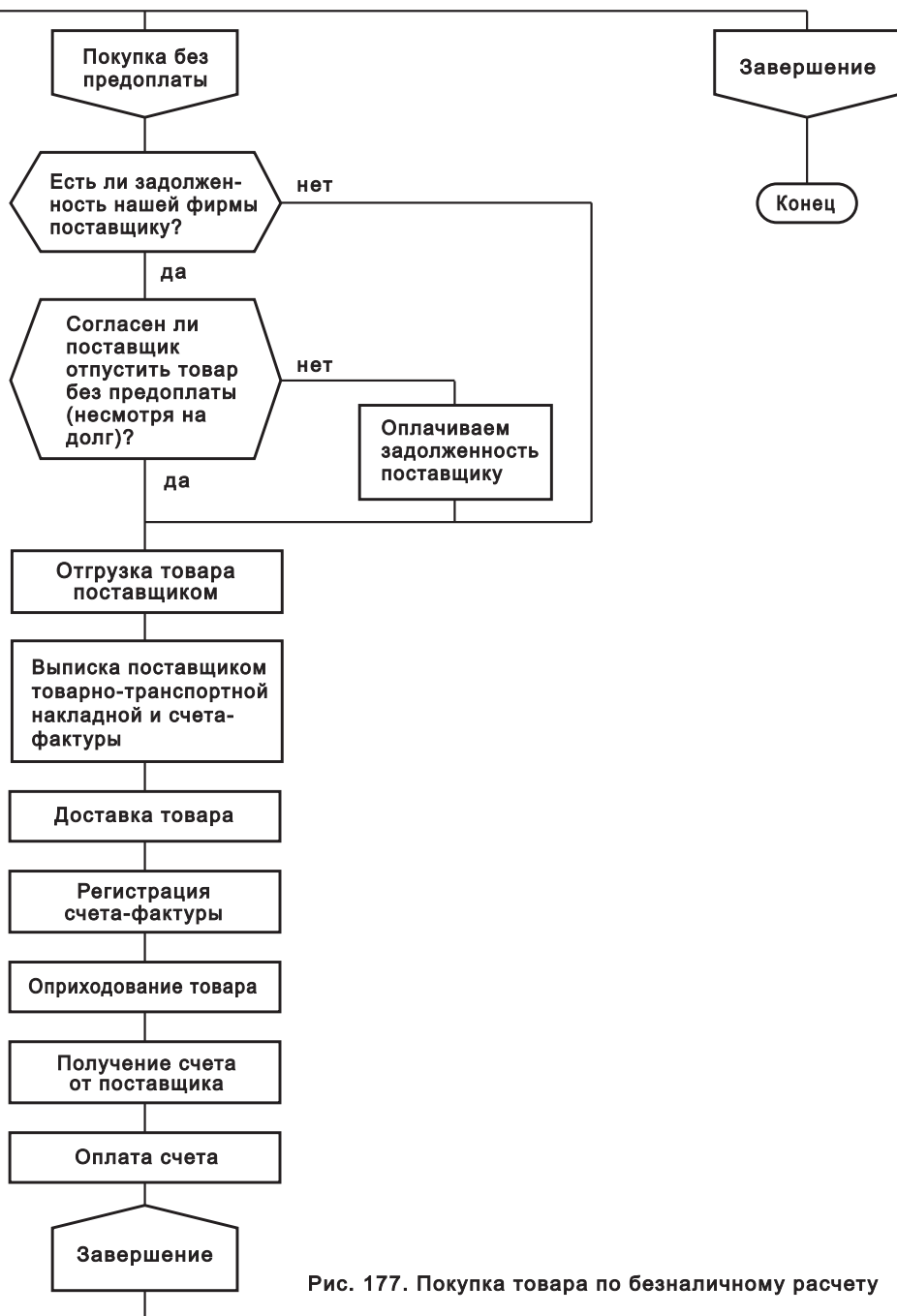


Рис. 177. Покупка товара по безналичному расчету

§4. ПОКУПКА ТОВАРОВ ПО БЕЗНАЛИЧНОМУ РАСЧЕТУ

Рассмотрим покупку товаров по безналичному расчету (рис. 177). Силуэт состоит из четырех веток :

- выбор торговой операции;
- покупка товаров с предоплатой;
- покупка товаров без предоплаты;
- завершение.

Первая ветка выбирает один из двух видов торговых операций. Вторая ветка описывает покупку товаров с предоплатой. Третья – покупку без предоплаты.

Мы показали наиболее простые примеры. В реальной торговой практике встречаются очень сложные и чрезвычайно разнообразные алгоритмы.

§5. ВЫВОДЫ

1. Торговля, как человеческая деятельность, требует высокого уровня взаимопонимания между заказчиками и разработчиками программных комплексов, обслуживающих торговые операции.
2. Между тем должное взаимопонимание является труднодостижимой целью.
3. Недостаточное взаимопонимание значительно снижает производительность труда и нередко влечет за собой упущенную выгоду.
4. Причина состоит в том, что сегодня отсутствуют наглядные языковые средства:
 - пригодные для эффективного описания алгоритмов, используемых в торговой деятельности,
 - способные обеспечить быстрое взаимопонимание между заказчиками и разработчиками программных комплексов торгового назначения.
5. Отсутствие удобного языка для описания торговых алгоритмов является болевой точкой торговли.
6. Нынешний «торговый язык» отстал от жизни и превратился в тормоз развития отрасли.
7. Язык ДРАКОН позволяет устранить недостаток и добиться значительного повышения производительности труда в этой области.

АЛГОРИТМЫ БУХГАЛТЕРСКОГО УЧЕТА¹

§1. ПРОБЛЕМА НЕПОНИМАНИЯ В СИСТЕМАХ БУХГАЛТЕРСКОГО УЧЕТА

В современные системы бухгалтерского учета закладывается все больше сложных правил анализа и обработки информации. Эти правила очень трудно объяснить бухгалтеру, потому что они представляют собой длинные цепочки анализа и принятия решения. При внедрении бухгалтерских программ очень часто возникает проблема НЕПОНИМАНИЯ.

Если бухгалтер не понимает, как программа формирует проводки, у него возникает недоверие. Недоверие порождает неумение и нежелание использовать программу.

§2. ВНЕДРЕНИЕ ПРОГРАММ БУХГАЛТЕРСКОГО УЧЕТА

Наша фирма «Инженеры информации» занимается внедрением программы 1С-Бухгалтерия. Программа поставляется в типовой конфигурации, которая содержит наиболее распространенные в бухгалтерском учете документы. Эти документы при заполнении формируют по заданным правилам бухгалтерские проводки. Проводки формируются в разных вариантах. Результат проводки зависит от остатков и оборотов бухгалтерских счетов в разные моменты времени.

Трудность в том, что схема учета, принятая на предприятии, часто отличается от схемы учета, заложенной в стандартной конфигурации.

Задача нашей фирмы – объяснить бухгалтеру схему учета, принятую в типовой конфигурации программы «1С-Бухгалтерия». Бухгалтер должен увидеть отличия своей схемы учета от типовой конфигурации. И принять решение о доработке типовой конфигурации.

¹ Эта глава написана А.Н. Шилиным.

§3. КАК МЫ ИСПОЛЬЗОВАЛИ ЯЗЫК ДРАКОН

Мы долго искали метод доступного и наглядного изображения правил, заложенных в программу бухгалтерского учета. Несколько лет назад нам встретила книга «В. Д. Паронджанов. Как улучшить работу ума. Новые средства для образного представления знаний, развития интеллекта и взаимопонимания. М.: Радио и связь, 1998. – 352 с.». В ней описан язык визуального отображения информации ДРАКОН. Он был разработан при создании вычислительной системы космического корабля «Буран» и сейчас применяется при проектировании ракетной техники.

- Методы и описания языка ДРАКОН напоминают блок-схемы. Но в нем есть то, что отсутствует в блок-схемах.
- В язык ДРАКОН заложена система правил, которые улучшают восприятие информации. Оформление блок-схем по этим правилам делает их понятным любому человеку.

Раньше (до знакомства с ДРАКОНОм) мы были вынуждены описывать работу документов в текстовом виде с использованием текстовых алгоритмических конструкций. См. пример на рис. 178.

§4. ДРАКОН-СХЕМА, ОПИСЫВАЮЩАЯ РАБОТУ ДОКУМЕНТА «ПРИХОДНЫЙ КАССОВЫЙ ОРДЕР»

ДРАКОН позволил перейти от текста к графике. Пример дракон-схемы показан на рис. 179. Легко видеть, что содержание текста (рис. 178) и схемы (рис. 179) полностью совпадает. Основные предложения в тексте и надписи на рисунке не отличаются. Исчезли лишь ключевые слова: Если, Тогда, Конец Если и т.д. Вместо них появились линии, отражающие ход управления программой.

Сравнивая две формы представления алгоритма, легко убедиться, что дракон-схема намного удобнее. Благодаря схеме изложение мысли приобрело наглядность и доходчивость, свойственные графике.

Отметим важную особенность. В документе на рис. 178 обработка счета 62 (взаиморасчеты с покупателями) слишком объемна. Если показать ее в тексте полностью, из-за большого объема будет потеряна ясность и наглядность.

Поэтому обработка счета 62 выделена в отдельную процедуру, которая на рис. 178 указана одной фразой «Обработка счета 62». В текстовой форме процедура приведена на рис. 180. Соответствующая ей дракон-схема показана на рис. 181.

§5. СРАВНЕНИЕ ТЕКСТА И ДРАКОН-СХЕМЫ

Ясно, что схема на рис. 181 намного наглядней, чем текст. Почему?

Правила языка ДРАКОН заставляют разбивать графическую информацию на отдельные обособленные блоки. Двумерное пространство графики позволяет наглядно расположить эти блоки и показать связь между ними.

- В тексте мы используем практически одно измерение – вертикаль. А в дракон-схеме информация эффективно разворачивается в двух измерениях – вертикальном и горизонтальном.
- Это очень важно, потому что система «глаз–мозг» лучше воспринимает панорамную информацию.

§6. ДРАКОН РЕШАЕТ ПРОБЛЕМУ ВЗАИМОПОНИМАНИЯ

Когда мы только начинали использовать в своей работе дракон-схемы, то опасались, что бухгалтеры нас не поймут, настраивались на долгие объяснения. К нашему удивлению, вопросов о внутреннем устройстве дракон-схем практически не было.

Многие бухгалтеры, глядя на дракон-схемы, сразу же выделяли существенные для себя проблемы. И тут же начинали объяснять, что их не устраивает. И какие изменения следует внести, чтобы учесть особенности конкретного производства. Очень быстро устанавливалась атмосфера взаимного понимания решаемой задачи.

- Любой специалист, занимающийся автоматизацией, знает, как трудно понять: что именно хочет заказчик.
- Применение языка ДРАКОН дало нам возможность преодолеть непонимание и недоверие бухгалтеров при внедрении программ «1С».

Кроме того, мы используем язык ДРАКОН для проектирования задач бухгалтерского учета. То есть, применяем дракон-схемы, записывая новую информацию при постановке задач «с нуля».

Подведем итоги. После нескольких лет проблем и трудностей, которые мы испытывали при обучении бухгалтеров и внедрении бухгалтерских программ, нам удалось, наконец, найти нужную методику работы. Теперь в наших руках есть набор правил и процедур языка ДРАКОН, с помощью которых можно ясно и полно описать нашу работу.

Самое главное, мы добились, что бухгалтер стал понимать суть работы программы «1С». Язык ДРАКОН решил для нас с заказчиком важнейшую проблему взаимопонимания.

Пример текстовой записи работы документа «Приходный кассовый ордер»

Работа с документом «Приходный Кассовый Ордер»

Какой счет записан в поле Корреспондирующий счет

Если счет 46 (Реализация) Тогда

 Если в документе указан Налог С Продаж Тогда

 Проводка Дебет(46) Кредит(68,30)

 Конец Если

 Проводка Дебет(50) Кредит(46)

Если счет 62 (Покупатели) Тогда

 Обработка счета 62 (Покупатели)

Если счет 64 (Предоплата) Тогда

 Если в документе указан Налог С Продаж Тогда

 Если местные законы требуют платить с авансов Налог С Продаж Тогда

 Начисляем Налог С Продаж с аванса

 Проводка Дебет(64) Кредит(68,30)

 Конец Если

 Конец Если

 Выделяем НДС с аванса

 Проводка Дебет(64) Кредит(68,2)

 Проводка Дебет(50) Кредит(64)

Если другой счет (не 46, 62, 64) Тогда

 Проводка Дебет(50) Кредит(Корреспондирующий счет)

Конец Если

Конец работы с документом

Примечание. Нормативные документы часто меняются. В настоящее время налог с продаж отменен. Но мы не стали исправлять документ, сохранив его как историческую реликвию

Рис. 178. Текстовая запись (псевдокод) обработки документа
«Приходный кассовый ордер»

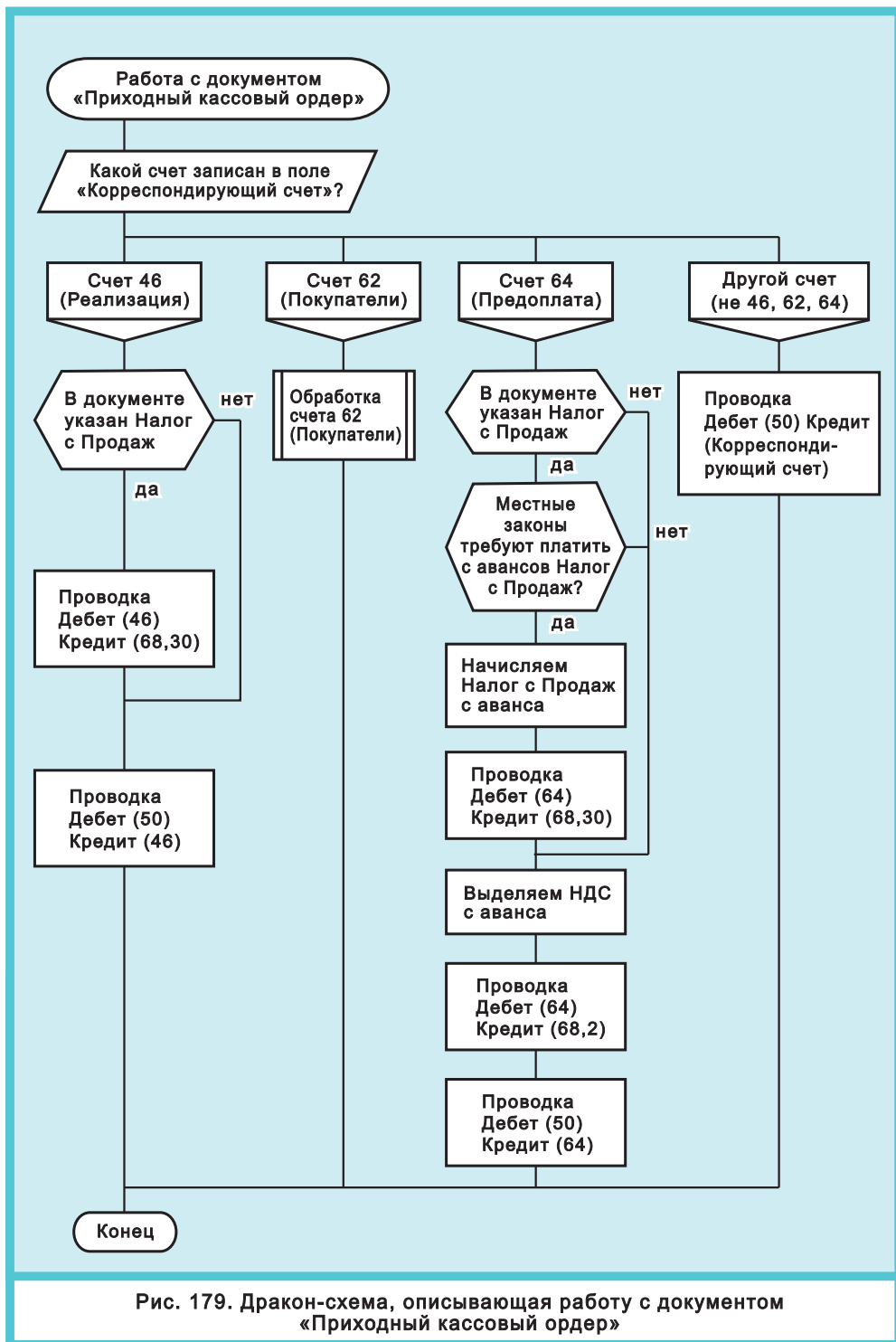


Рис. 179. Дракон-схема, описывающая работу с документом «Приходный кассовый ордер»

Пример текстовой записи обработки Счета 62 при поступлении денег в кассу

Обработка счета 62

Проверить задолженность Покупателя

Если Есть задолженность у Покупателя Тогда

Если Покупатель платит больше чем должен нам Тогда

Предупредить бухгалтера об этом

Конец Если

Если Метод определения выручки По Оплате тогда

Если есть неоплаченная отгрузка Тогда

Вычислить какая часть долга Покупателя гасится суммой Оплаты

Пропорционально уменьшить сумму неоплаченной отгрузки

Проводка Дебет (ПС) (ПС — забалансовый счет).

Конец Если

Конец Если

Конец Если

Если в документе указан Налог С Продаж Тогда

Начисляем неоплаченный при отгрузке Налог С Продаж.

Проводка Дебет(76,4) Кредит(68,30)

Конец Если

Если Метод определения выручки По Оплате Тогда

Начисляем неоплаченный при отгрузке НДС.

Проводка Дебет(76,4) Кредит(68,2)

Если в документе указан НГСМ Тогда

Начисляем неоплаченный при отгрузке НГСМ.

Проводка Дебет(76,4) Кредит(67,2)

Конец Если

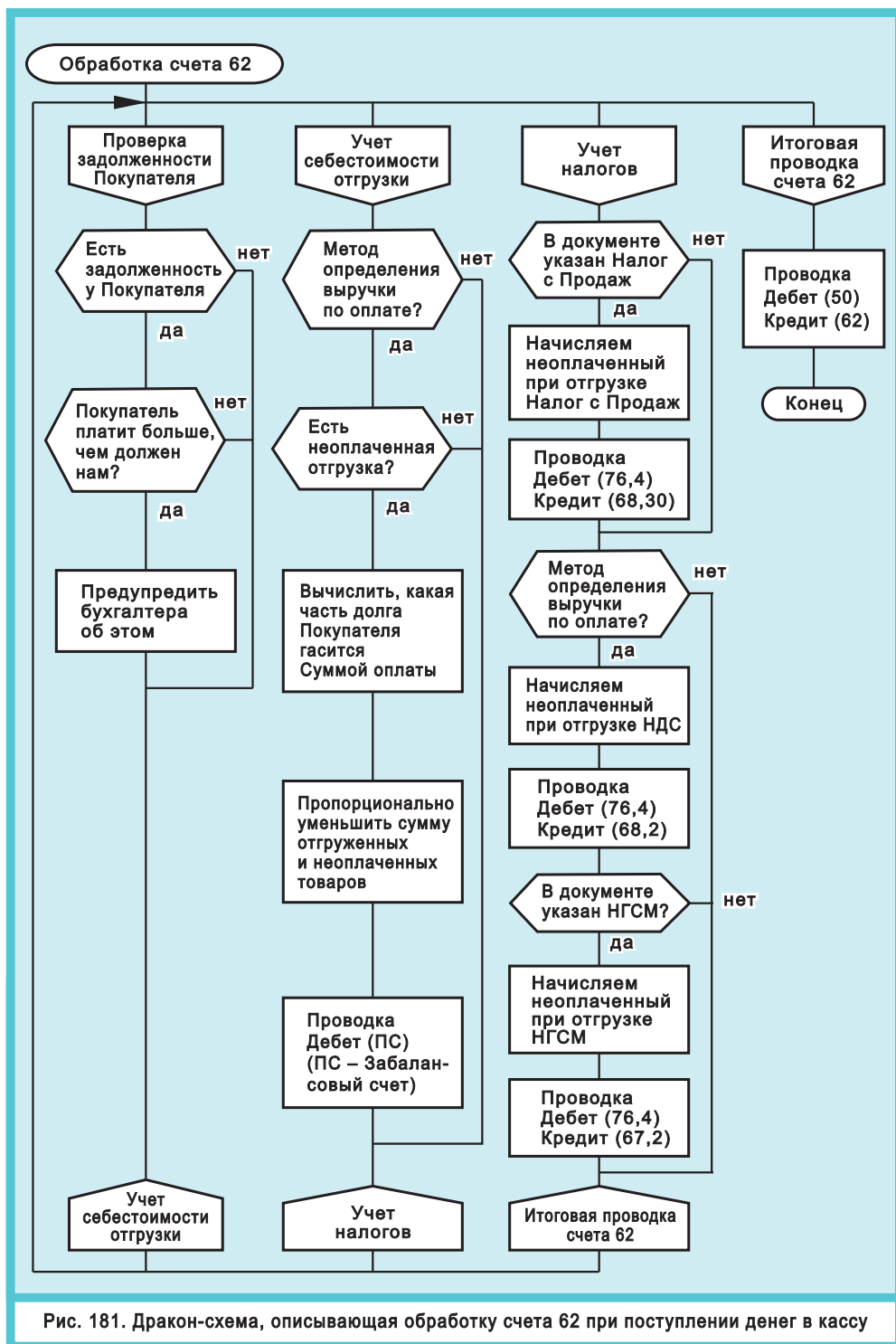
Конец Если

Проводка Дебет(50) Кредит(62)

Конец обработки счета 62

Примечание. Нормативные документы часто меняются. В настоящее время налог с продаж отменен. Но мы не стали исправлять документ, сохранив его как историческую реликвию

Рис. 180. Текстовая запись (псевдокод) обработки Счета 62



§7. ВЫВОДЫ

1. В современных системах бухгалтерского учета применяются сложные правила анализа и обработки информации.
2. Эти правила зачастую трудно объяснить бухгалтеру, потому что они представляют собой длинные цепочки анализа и принятия решения. При внедрении бухгалтерских программ нередко возникает проблема *непонимания*.
3. Если бухгалтер не понимает, как программа формирует проводки, у него возникает недоверие. Недоверие порождает неумение и нежелание использовать программу.
4. Язык ДРАКОН позволяет решить данную проблему.
5. В язык заложена система правил, которые улучшают восприятие информации. Оформление схем по этим правилам делает их понятным любому человеку.
6. Благодаря дракон-схемам изложение мысли приобрело наглядность и доходчивость, свойственные графике.
7. В тексте мы используем практически одно измерение – вертикаль. А в дракон-схеме информация эффективно разворачивается в двух измерениях – вертикальном и горизонтальном. Это важно, потому что система «глаз–мозг» лучше воспринимает панорамную информацию.

АЛГОРИТМЫ В АТОМНОЙ ЭНЕРГЕТИКЕ

§1. АЛГОРИТМЫ ДЛЯ ОПИСАНИЯ СЛОЖНЫХ ПРОЦЕССОВ

За последние сто лет сложность труда скачкообразно изменилась. Труд стал во много раз сложнее, разнообразнее, многограннее.

Чтобы подготовка к сложному труду была эффективной, во многих (хотя и не во всех) случаях необходимо иметь *описание предстоящей работы*. Такое описание должно быть очень хорошим. Это значит – доступным, удобным, легким для понимания.

Существуют ли такие описания? Нет, не существуют!
Сделаем оговорку. Описания, конечно, есть, и их довольно много. Но они, как правило, имеют низкое качество. Имеющиеся описания изложены трудным, непонятным (текстовым) языком.
Они непригодны для эффективного обучения, так как требуют слишком большого времени для понимания. Это значит, что производительность понимания слишком мала.

До последнего времени этой проблеме (проблеме хороших описаний) не уделялось должного внимания. Следствием подобного пренебрежения являются значительные экономические и интеллектуальные потери.

В связи с этим можно сделать несколько замечаний.

- Существующие способы описания сложных процессов отстали от жизни и не удовлетворяют современным требованиям.
- Во многих случаях сложные процессы можно рассматривать как алгоритмы.
- Чтобы организовать производство хороших описаний для сложных процессов, нужно иметь эргономичный алгоритмический язык. Язык, доступный для широкого круга специалистов, не знакомых с алгоритмами.

- В качестве такого языка целесообразно использовать язык ДРАКОН.
- Широкое применение дракон-схем для описания структуры сложной деятельности и наукоемких процессов позволит заметно повысить продуктивность умственного труда.

§2. ЧРЕЗВЫЧАЙНО СЛОЖНЫЕ АЛГОРИТМЫ

Иногда высказывают мнение, что ДРАКОН хорошо описывает простые задачи и непригоден для изображения сложных проблем. Это неверно. ДРАКОН специально сконструирован, чтобы облегчить описание и решение широкого спектра задач, включая самые сложные. Более того, чем сложнее проблема, тем больше выигрыш от использования языка ДРАКОН.

Одной из подобных задач является проектирование ядерного реактора. Ясно, что это грандиозная, запредельная по сложности проблема.

Если изобразить сложную проблему на языке ДРАКОН, наблюдается следующий неожиданный эффект. Хорошо знакомая задача на глазах преобразается и предстает перед нашим изумленным взором в совершенно новом свете. Она резко упрощается и становится ясной, четкой, обозримой.

Конечно, все это не надо понимать слишком буквально. Сложность есть сложность, и полностью избавиться от нее невозможно. Тем не менее, одна из основных идей когнитивной эргономики в том и заключается, чтобы – в пределах возможного – устранить неоправданную сложность. И представить громоздкую и трудную задачу в максимально удобной и наглядной форме. По принципу: «Посмотрел – и сразу понял!». «Взглянул – и сразу стало ясно!».

§3. ДРАКОН И СЛОЖНОСТЬ

Язык ДРАКОН способен упростить и сделать наглядной сложнейшую задачу проектирования ядерного реактора.

Прежде всего, необходимо получить целостный взгляд на проблему. Для этого служит специальная фигура – *шапка*. Она задает смысловой костяк алгоритма.

Изюминка в том, что шапка делает человеку ценные подсказки, облегчающие работу ума. Как известно, шапка (рис. 182) позволяет получить ответ на три «царских» (наиболее важных) вопроса:

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



Вот ответы для рис. 183.

- Как называется задача? (*Читаем заголовок алгоритма*).
Проектирование ядерного реактора.
- Из скольких частей она состоит? (*Считаем иконы «имя ветки»*).
Из шести.
- Как называется каждая часть? (*Читаем текст в иконах «имя ветки»*).
 - выбор схемно-конструктивных решений;
 - расчетный анализ стационарных режимов;
 - расчетный анализ систем безопасности;
 - расчетный анализ аварий;
 - расчетный анализ элементов реактора;
 - расчет радиационной защиты.

Сколько времени нужно, чтобы впитать эту информацию? Минуту? Две? Три? Пять?

Шапка дает общее представление, без деталей. Благодаря шапке читатель получает ориентировку о проблеме практически мгновенно. Причем, что очень важно, время ориентировки почти не зависит от сложности задачи и составляет считанные минуты.

Из шапки мы узнали заголовки веток. После этого, как всегда, приступаем к двум стандартным операциям:

- изучению содержания веток,
- анализу их взаимодействия.

Чтобы выполнить эти операции и тщательно изучить алгоритм на рис. 183, читателю вряд ли потребуется больше, чем пятнадцать–двадцать минут. (Речь, разумеется, идет о специалистах или студентах, а не о человеке с улицы).

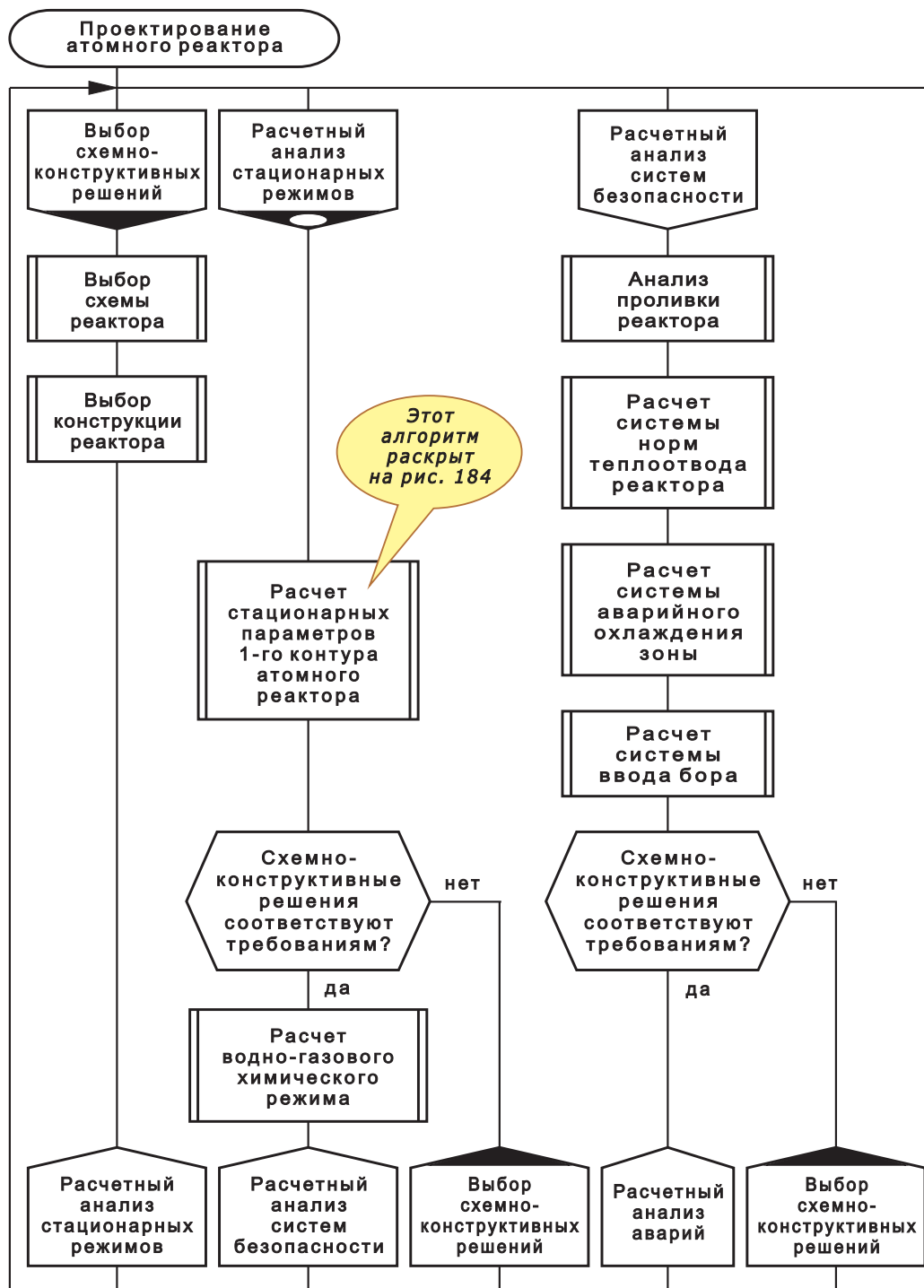
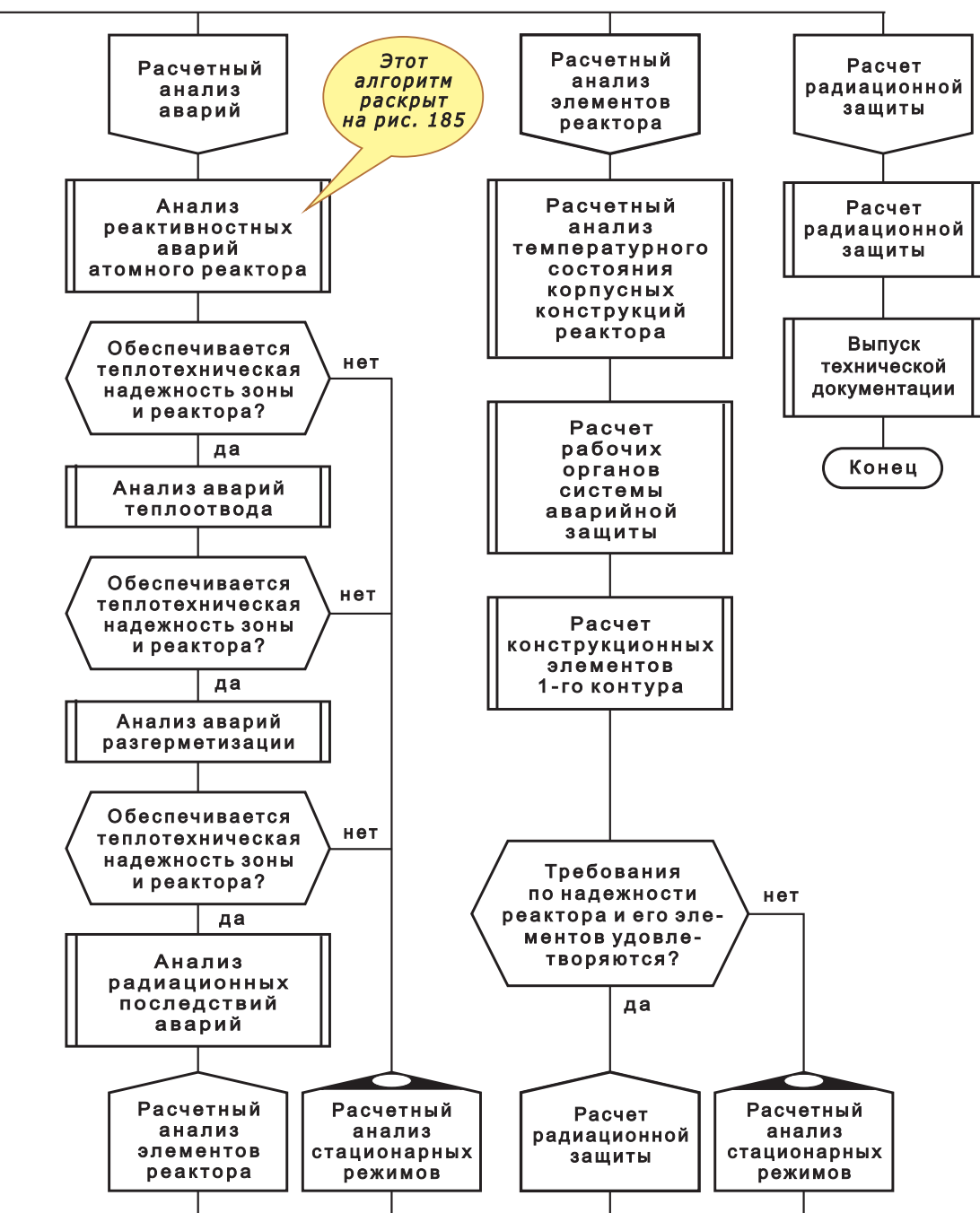


Рис. 183. Очень сложный алгоритм «Проектирование атомного реактора»



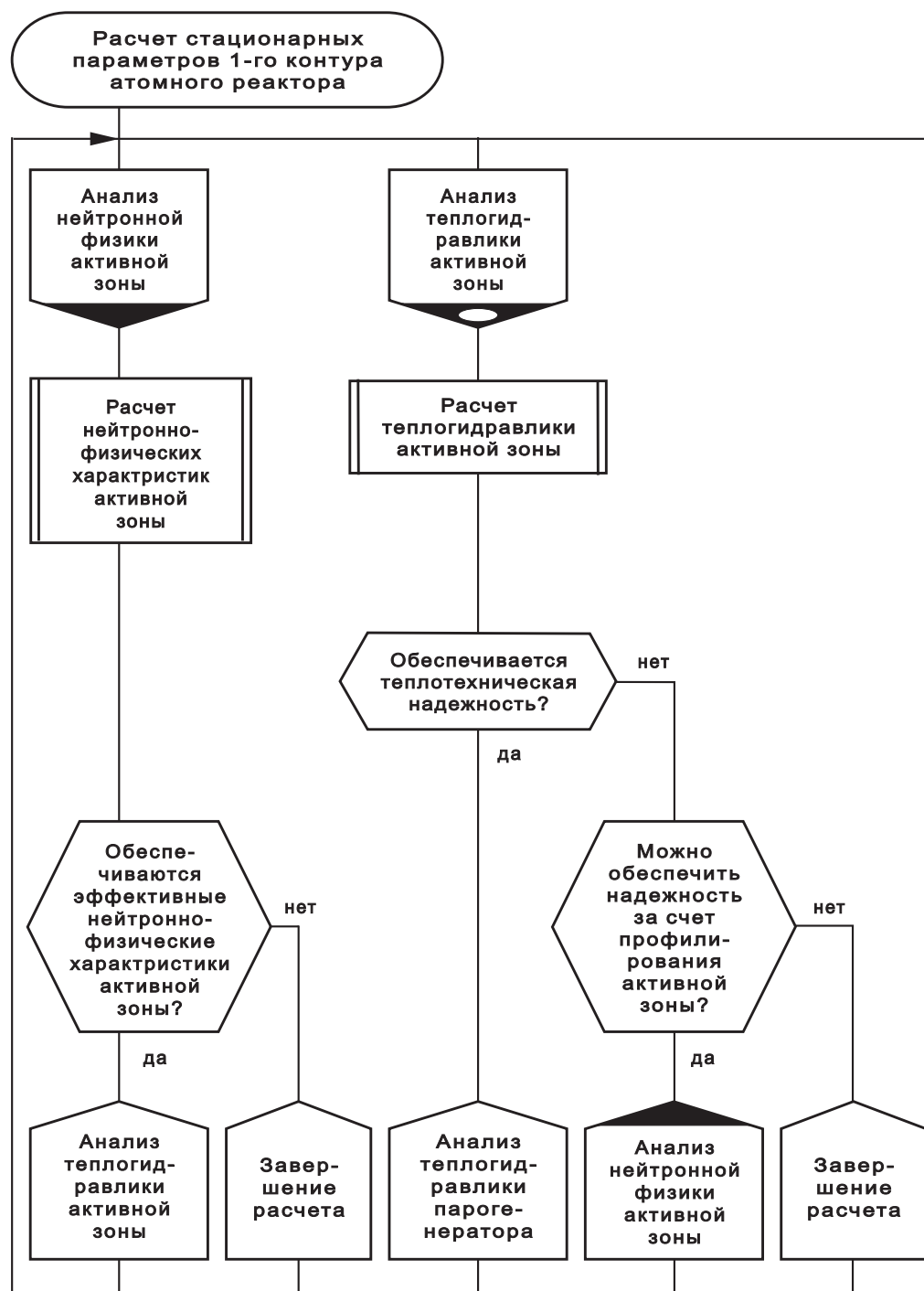
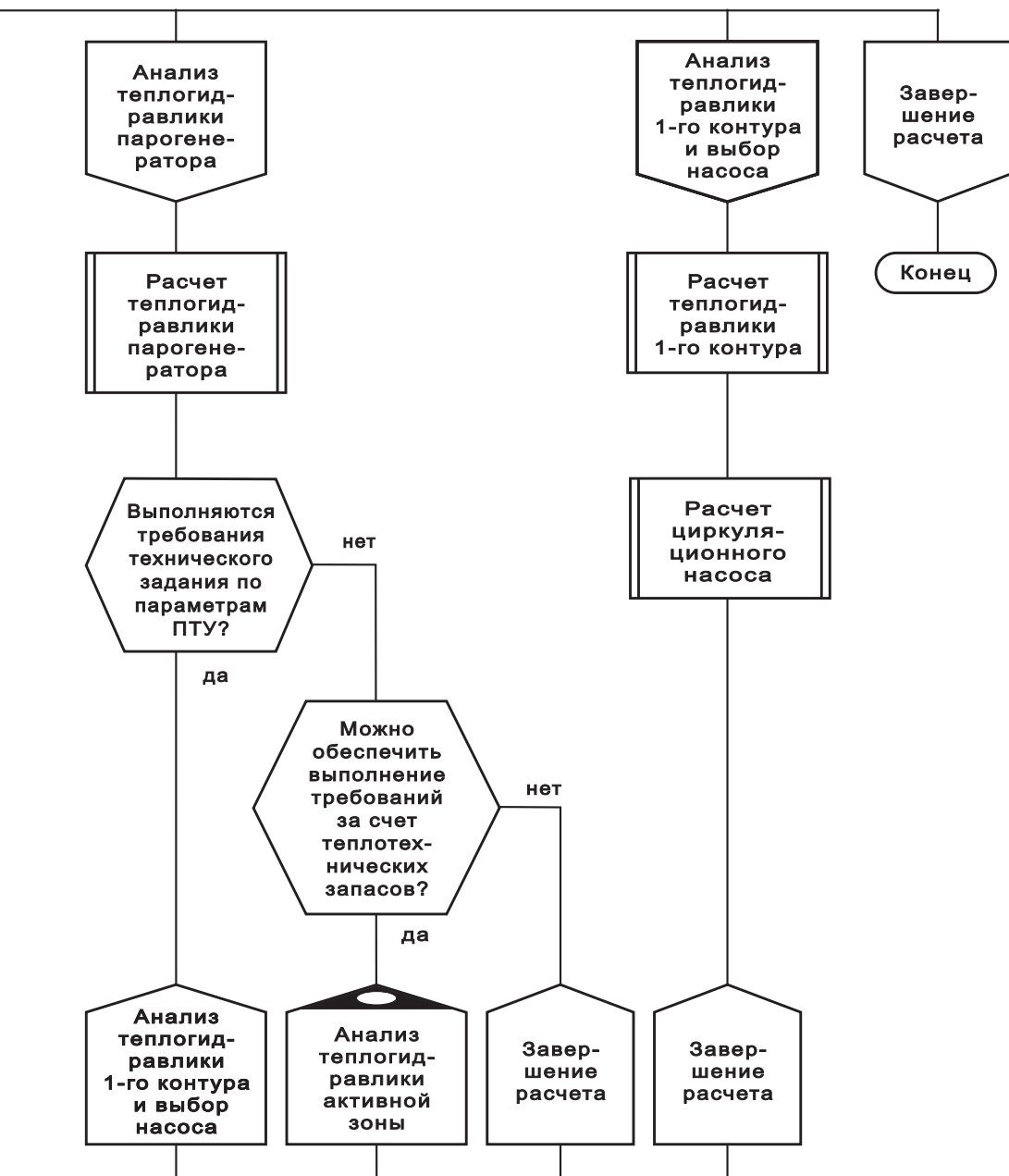


Рис. 184. Алгоритм «Расчет стационарных параметров 1-го контура атомного реактора»



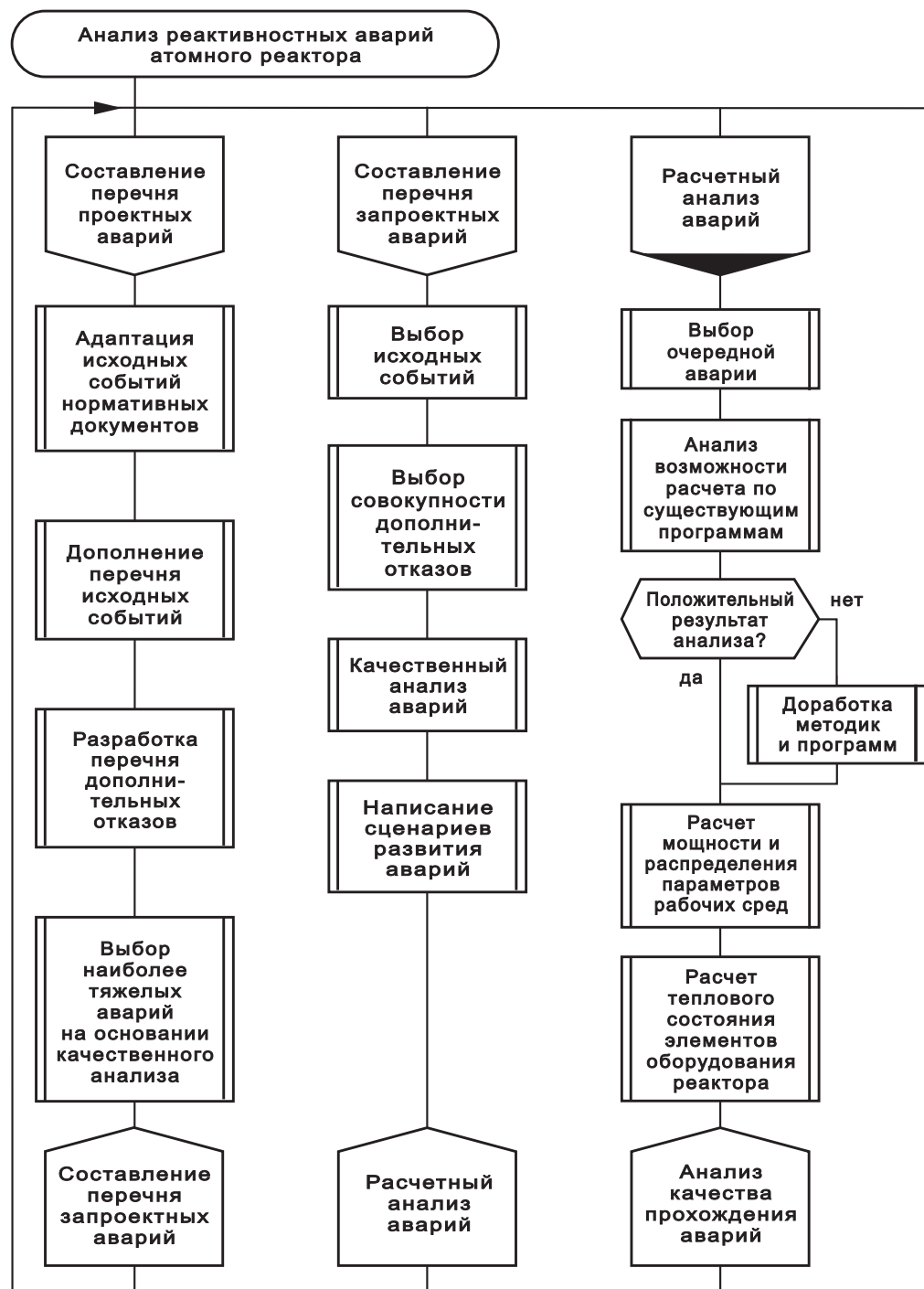
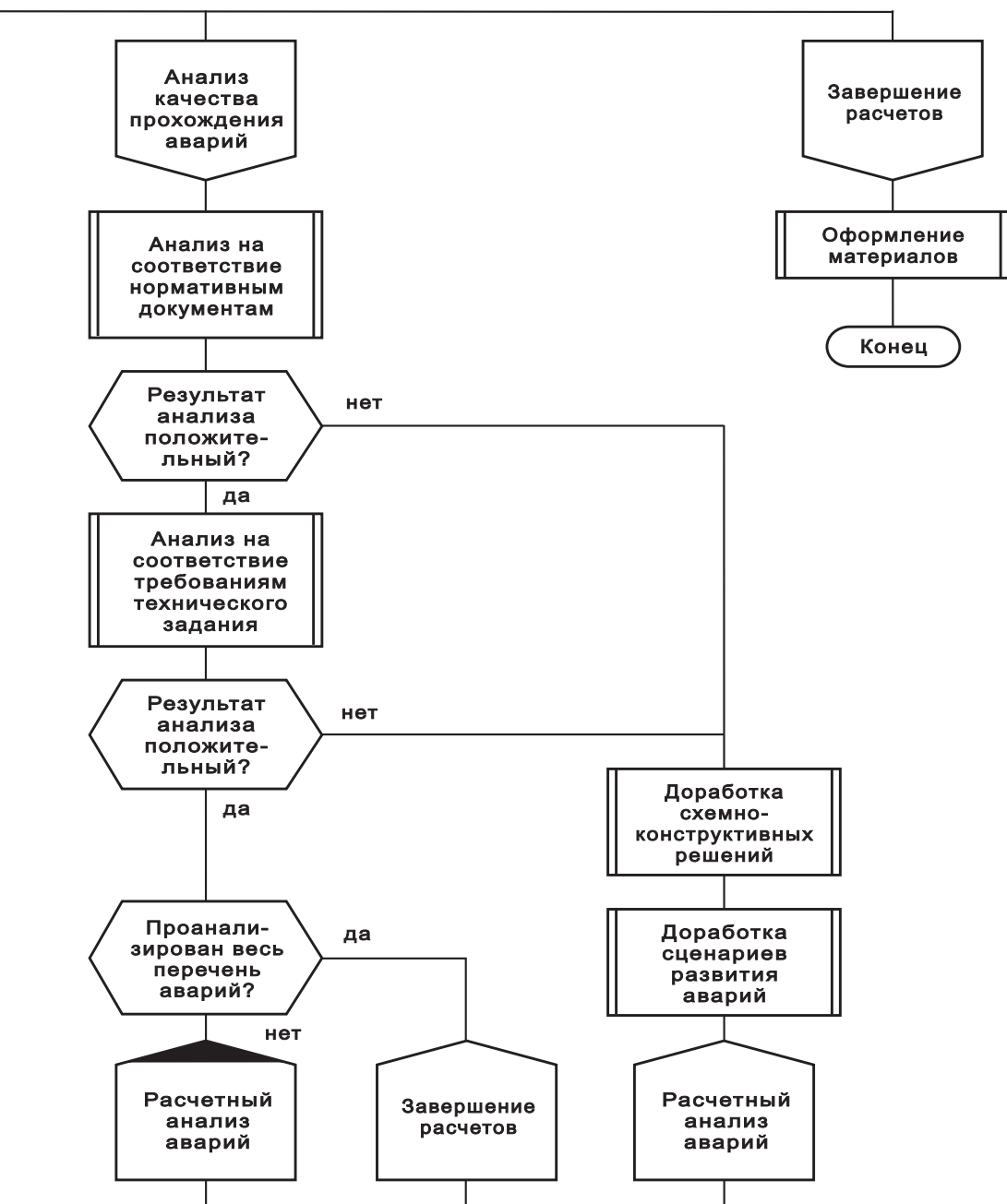


Рис. 185. Алгоритм «Анализ реактивных аварий атомного реактора»



§4. ВАЖНАЯ РОЛЬ ВСТАВОК

Алгоритм на рис. 183 содержит большое число вставок (процедур). Например, во второй ветке есть вставка «Расчет стационарных параметров 1-го контура ядерного реактора». А в четвертой – вставка «Расчет реактивностных аварий ядерного реактора». Эти вставки раскрыты на рис. 184 и 185.

Рис. 183–185 убедительно демонстрируют, что любой, сколь угодно сложный алгоритм можно изобразить с помощью простого и единообразного приема – *декомпозиции*.

Декомпозиция алгоритма – разбиение алгоритма на несколько уровней иерархии (которые похожи на этажи в многоэтажной пирамиде).

Поскольку мы ведем речь об эргономичных (наглядных) алгоритмах, уместно использовать термин *эргономичная декомпозиция* алгоритма.

Верхний уровень иерархии (верхний этаж) показан на рис. 183. Его можно рассматривать как вершину гигантской пирамиды, откуда открывается взгляд на проблему с высоты птичьего полета. Там же перечисляются все алгоритмы второго уровня, которые в нашей воображаемой пирамиде расположены на один шаг ближе к земле.

Рассматривая алгоритмы второго уровня (примеры которых показаны на рис. 184 и 185), легко заметить, что в них указываются алгоритмы третьего уровня, которые находятся еще ближе «к земле». То есть дают более детальное знакомство с проблемой.

Постепенно спускаясь с вершины пирамиды к ее основанию, мы наблюдаем последовательное разбиение (декомпозицию) сложной проблемы на все более мелкие и подробные детали. В конечном итоге (когда мы спустимся «на уровень земли») эти детали дадут исчерпывающее и полное описание нашего алгоритма. При необходимости его можно дополнить справочником, содержащим определения всех использованных понятий.

Важным достоинством является тот факт, что язык ДРАКОН не зависит от уровня иерархии. Он используется и на самом верху, и у основания пирамиды. Благодаря этому достигается резкое упрощение алгоритмов любой сложности. В итоге особо сложная («уму непостижимая») проблема превращается в относительно простую, ясную и наглядную.

До сих пор практически отсутствовали *эффективные эргономичные* изобразительные средства, позволяющие одновременно решать две задачи: когнитивную формализацию и визуализацию больших алгоритмов. По этой причине целостный взгляд на большой алгоритм, как на детерминированный многоступенчатый процесс, имеющий начало и конец, по сути дела был недоступен широкому кругу специалистов и учащихся.

Раньше целостный взгляд на проблему оставался достоянием узкой группы элитных специалистов, которые «все держат в голове». Из-за этого остальным участникам сложного проекта вынужденно отводилась незавидная роль винтиков творческого коллектива, которые должны знать свой «шесток». И которым «не положено» иметь целостное панорамное видение процесса во всей его многосложности.

Язык ДРАКОН позволяет сделать важный шаг к устранению этого недостатка. Он дает возможность более эффективно организовать совместную работу участников сложного проекта. И более разумно использовать интеллектуальные ресурсы их коллективного мозга.

Хорошая (эргономичная) графика упорядочивает сложный текст и структурирует его. Она разбивает огромную словесную глыбу на маленькие порции, которые можно проглотить легко и с удовольствием. В результате читатель получает долгожданную помощь, так как алгоритм становится более легким для понимания.

§5. ВЫВОДЫ

1. Чтобы обеспечить эффективное выполнение сложной наукоемкой работы, необходимо иметь ее описание.
2. Существующие способы описания особо сложных трудовых процессов отстали от жизни и не удовлетворяют современным требованиям.
3. Чтобы организовать производство хороших описаний для особо сложных трудовых процессов, нужно иметь специальный алгоритмический язык. Язык, доступный для широкого круга специалистов, не знакомых с алгоритмами.
4. Можно предположить, что широкое применение дракон-схем для описания структуры особо сложной деятельности и наукоемких трудовых процессов позволит значительно улучшить работу ума и повысить эффективность развития общества.

В этой главе описан метод *эргономичной декомпозиции*, позволяющий строить сложные алгоритмы. Наряду с ним можно использовать еще один метод – метод *многостраничных силуэтов*, который будет описан в главе 30.

АЛГОРИТМЫ В БИОЛОГИИ

§1. РУКОТВОРНЫЕ И НЕРУКОТВОРНЫЕ АЛГОРИТМЫ

До сих пор мы рассуждали об алгоритмах, которые придуманы *человеком*. Но существуют и иные процессы, к созданию которых человек не имеет никакого отношения.

Имеются в виду *нерукотворные* алгоритмы, созданные природой в ходе естественного развития. В результате миллиардов лет эволюции на Земле образовалась жизнь. И связанные с ней *естественные алгоритмы*.

Планета Земля – царство жизни. В каждом живом организме происходят процессы невообразимой сложности, необходимые для поддержания жизни.

Рассматривая жизнь с точки зрения молекулярной биологии, генетики и других биологических наук, мы сталкиваемся со сложнейшими процессами, которые можно и нужно назвать алгоритмами.

Алгоритмы существуют всюду, где есть жизнь. Они ухитряются жить в каждом живом существе, в каждой живой клетке – от холерного вибриона до незабудки и слона.

Какова роль человека в естественных алгоритмических процессах? Она очевидна. Человек не проектирует, не творит и не создает эти алгоритмы. Он всего лишь *изучает* их.

Человечество заинтересовано в том, чтобы биологические науки продвигались вперед быстрыми темпами. От чего зависит скорость познания естественных алгоритмов? В частности, от формы представления алгоритмических процессов.

К сожалению, представители биологических наук не владеют эффективными способами записи естественных алгоритмов. Нынешний биологический язык отстал от жизни и превратился в тормоз развития науки.

§2. ВИЗУАЛИЗАЦИЯ БИОЛОГИЧЕСКИХ АЛГОРИТМОВ

Чем глубже человеческий разум проникает в тайны живой материи, тем яснее становится, что живые существа во многих случаях ведут себя как информационные биомашин, перерабатывающие информацию с помощью биоалгоритмов.

Опыт показывает, что биологические алгоритмы очень похожи на самые обычные алгоритмы, с которыми мы постоянно сталкиваемся в технике. А раз так, язык ДРАКОН может стать удобным средством для выражения и накопления знаний об алгоритмических процессах, протекающих в живых организмах.

В качестве примера рассмотрим алгоритм работы человеческого сердца.

Отрывок из школьного учебника

«Наше сердце постоянно в работе... Оно работает непрерывно 70–80 лет и более. В чем секрет его неутомимости?...

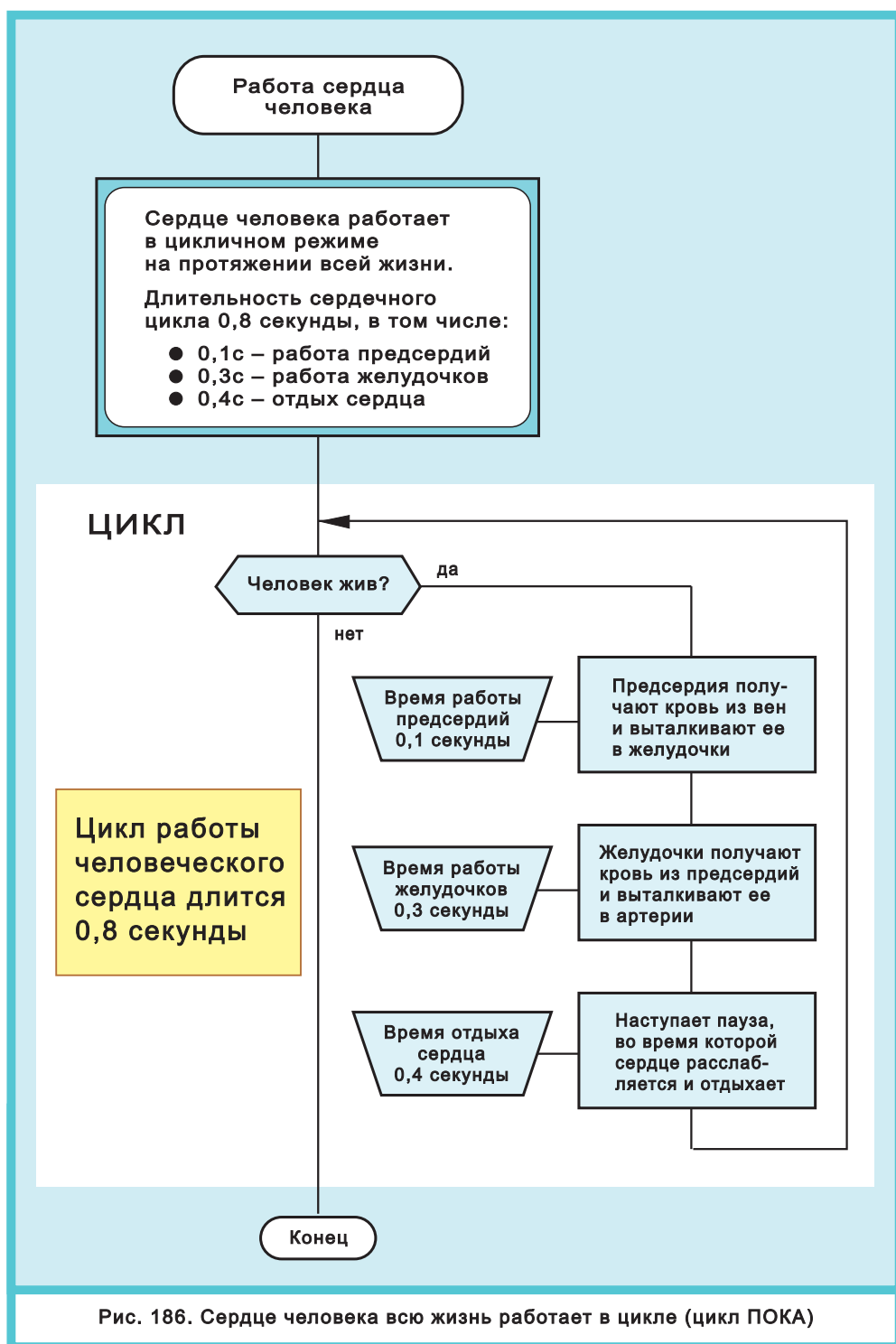
Во многом это объясняется особенностями работы сердца. Оно последовательно сокращается и расслабляется с короткими промежутками для отдыха. В одном сердечном цикле можно выделить три фазы.

Во время первой фазы, которая у взрослого человека длится 0,1 с, сокращаются предсердия, а желудочки находятся в расслабленном состоянии. За ней следует вторая фаза (она более продолжительная – 0,3 с): желудочки сокращаются, а предсердия расслаблены. После этого наступает третья, заключительная фаза – *пауза*, во время которой происходит общее расслабление сердца. Ее продолжительность 0,4 с. Весь сердечный цикл занимает 0,8 с.

Вы видите, что в течение одного сердечного цикла предсердия тратят на работу 12,5% времени сердечного цикла, а желудочки 37,5%. Остальное время, а это 50%, сердце отдыхает.

В этом секрет долголетия сердца, удивительной его работоспособности. Небольшие промежутки отдыха, следующие за каждым сокращением, дают возможность сердечной мышце отдохнуть и восстановить силы» [1].

Работа сердца очень сложна. Поэтому мы выбрали лишь малую часть работы нашего главного насоса. И изобразили ее в виде упрощенного алгоритма (рис. 186).



§3. КАК РАЗМНОЖАЮТСЯ ЖАБЫ

Рассмотрим еще один пример. Для начала приведем цитату из известной биологической книги, описывающую алгоритм размножения жаб.

«Рассмотрим изменения, происходящие в организме жабы в сезон размножения. Глаза жабы воспринимают свет и передают эту информацию в мозг, который определяет, что продолжительность светового дня увеличивается.

Гипоталамус направляет в гипофиз соответствующие рилизинг-факторы. Гипофиз начинает выделять в кровь различные гормоны...

Когда семенники и яичники «обнаруживают» их присутствие в крови, они начинают увеличиваться в размерах, продуцировать гаметы, а также выделять собственные гормоны и среди них – половые: тестостерон и эстроген. Реагируя на половые гормоны, мозг посылает нервные импульсы к мышцам – животное начинает поиск места для размножения и брачного партнера.

Так, благодаря сложному взаимодействию органов чувств, нервов, мозга, мышц и эндокринных желез животное адекватно реагирует на смену сезона – наступление весны» [2].

Описание этого алгоритма на языке ДРАКОН показано на рис. 187. Чтобы не утомлять читателя громоздкими биологическими терминами, мы упростили алгоритм. А в комментариях дали необходимые пояснения.

Следует подчеркнуть, что реальные биологические алгоритмы исключительно сложны. Традиционная для биологической литературы текстовая форма представления алгоритмических знаний вносит неоправданные трудности для читателей и является устаревшей.

§4. КИШЕЧНАЯ ПАЛОЧКА И ФАГ ЛЯМБДА

Рассмотрим более сложный пример биологического алгоритма.

Бактериофаг (сокращенно фаг) – это вирус, который пожирает бактерии. Таких вирусов очень много. Один из них, *фаг λ* (фаг лямбда), при некоторых условиях уничтожает бактерии, которые живут в кишечнике. Эти бактерии называются «кишечная палочка».

Фаг и бактерия похожи на карлика и великана. Фаг крохотный, а бактерия громадная. И, тем не менее, победа всегда достается фагу.

Фаг похож на клизму с острым концом. Этим кинжалом он легко протыкает стенку бедной бактерии. В шаре «клизмы» находится главное оружие фага – его генетический код. Через роковой прокол, этот

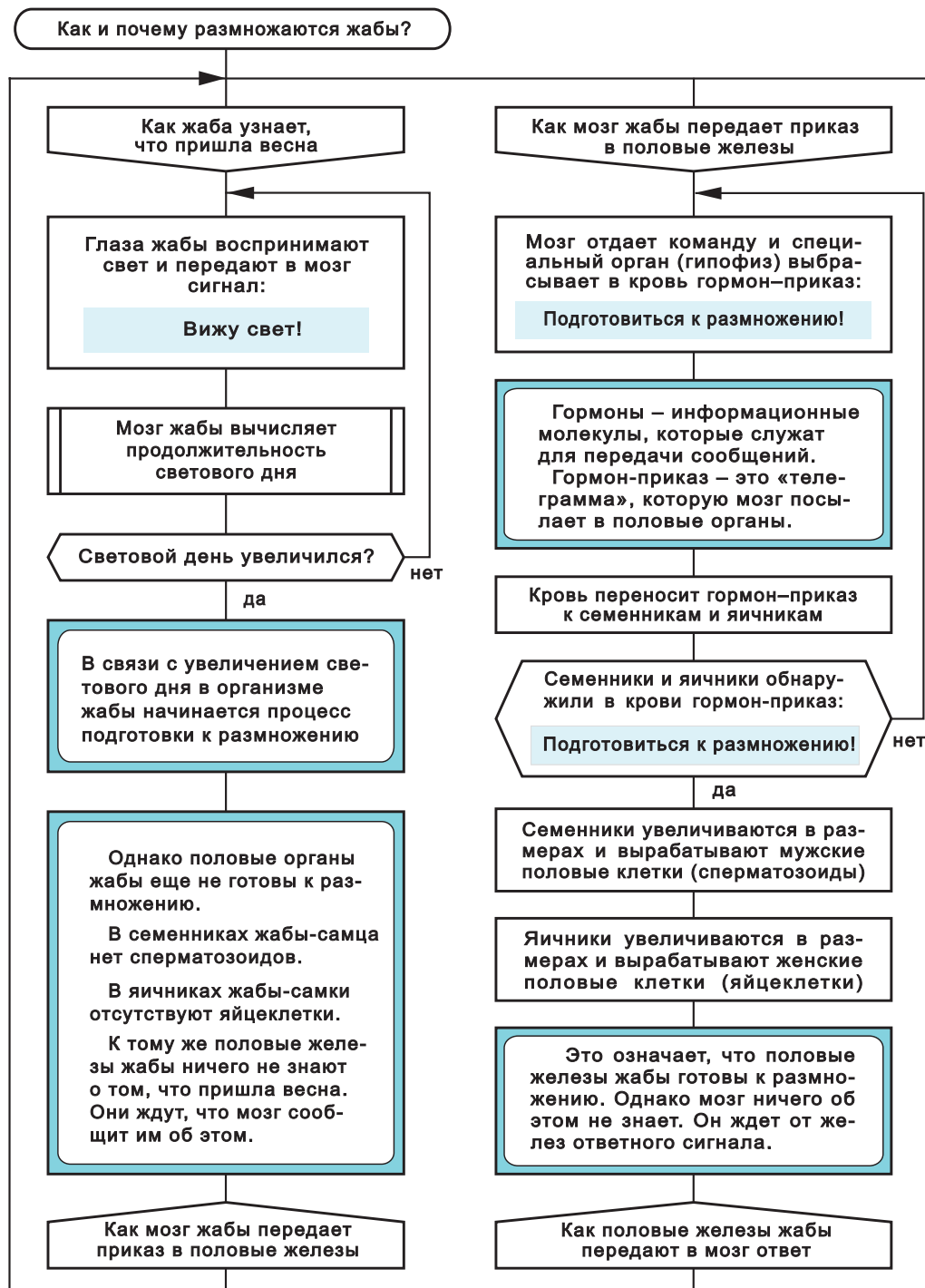
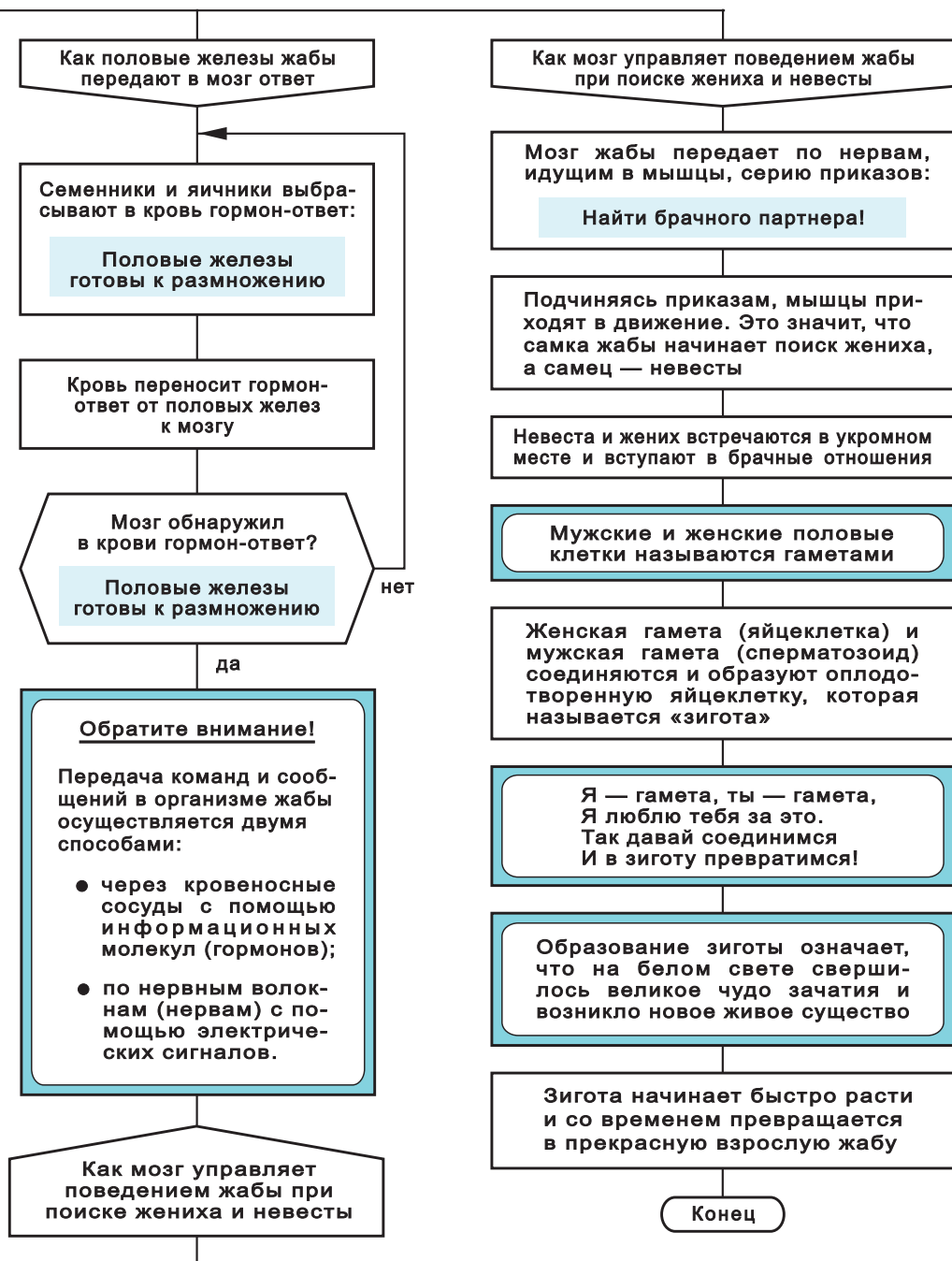


Рис. 187. Как размножаются жабы? Использование языка ДРАКОН для описания биологических процессов



код, подобно диверсанту со взрывчаткой, проникает внутрь бактерии и «взрывает» ее.

Рассмотрим эту историю подробнее.

§5. ДВА СПОСОБА ВЗАИМОДЕЙСТВИЯ ФАГА И БАКТЕРИИ

В начале фаг прокалывает оболочку бактерии (рис. 188). Затем генетический код фага через образовавшуюся дырку проскальзывает внутрь бактерии (рис. 189).

Дальнейшие события в бактериальной клетке могут развиваться двумя путями.

§6. РАЗМНОЖЕНИЕ ФАГОВ И ГИБЕЛЬ БАКТЕРИИ

В первом случае внутри клетки происходит интенсивное размножение фагов. Этот процесс идет очень быстро.

Примерно через 45 минут происходит гибель бактерии. Внутри бактерии скапливается огромное число фаговых частиц. Эти частицы в буквальном смысле разрывают бактериальную клетку.

Из лопнувшей бактерии высвобождаются около 100 фагов (рис. 190). Алгоритм пожирания бактерии фагами показан на рис. 192 [3].

§7. ГИБЕЛЬНОЕ ДЕЙСТВИЕ УЛЬТРАФИОЛЕТОВОГО ИЗЛУЧЕНИЯ

Во втором случае в дело вступают тормозящие вещества. Они запрещают размножение фагов. Вместо этого генетический код фага замирает и «прячется» внутри генетического кода бактерии.

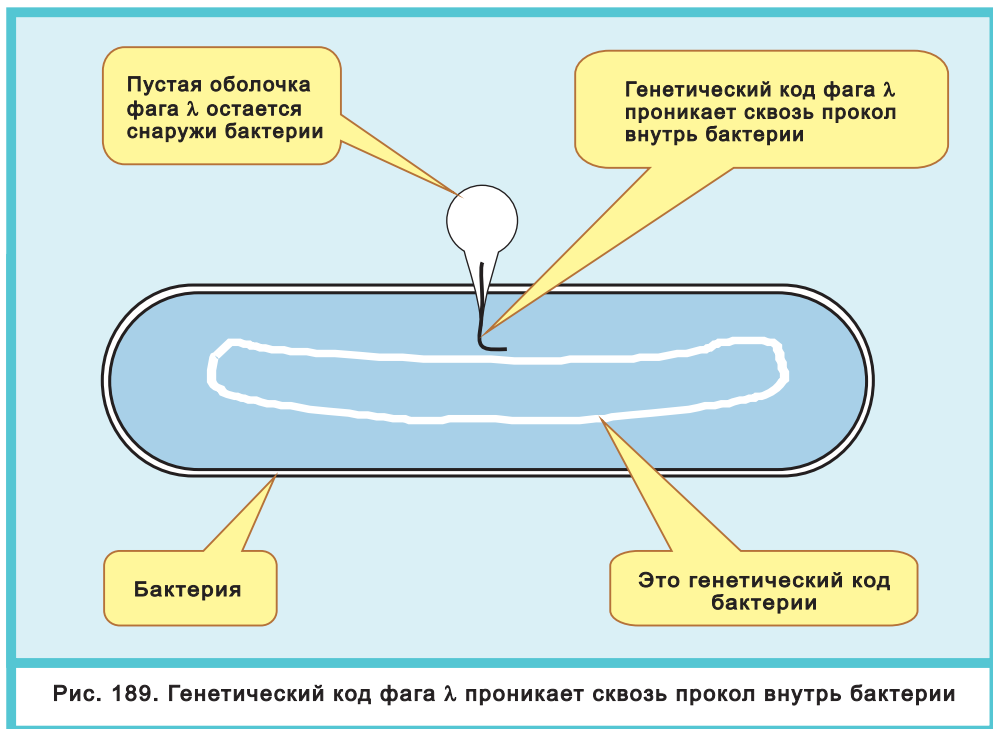
На рис. 191 показана ситуация, когда фаг становится пассивным. Он не размножается. Не причиняет бактерии никакого вреда. В этот период бактерия спокойно размножается. И спрятанный в ней фаг пассивно размножается вместе с ней.

Если же в этот момент на фаг и бактерию брызнет ультрафиолетовый свет, произойдет катастрофа. Ультрафиолет резко меняет свойства фага. Мирное сосуществование кончается. Фаг «просыпается» и начинает быстро размножаться внутри бактерии (рис. 193).

Через 45 минут бактерия разрывается и погибает. И новая сотня ужасных фагов вырывается наружу через стенки лопнувшей бактерии.

Таким образом, мы рассмотрели два алгоритма:

- фаг уничтожает бактерии (рис. 192);
- имеет место «мирное сосуществование» фага и бактерии, которое «трагически» заканчивается при попадании ультрафиолетового излучения (рис. 193).



РАЗМНОЖЕНИЕ ФАГА И ГИБЕЛЬ БАКТЕРИИ

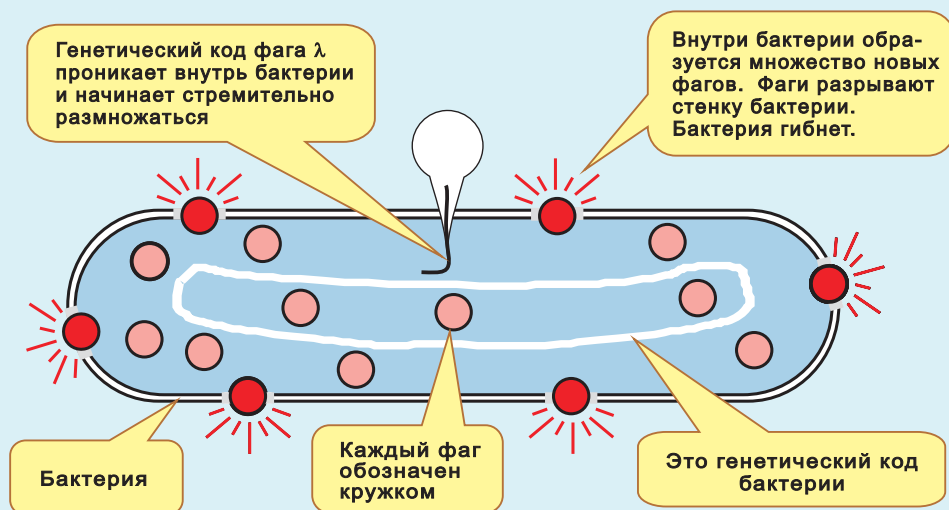


Рис. 190. Размножение фага внутри бактерии и гибель бактерии

ФАГИ НЕ РАЗМНОЖАЮТСЯ. ГЕНЕТИЧЕСКИЙ КОД ФАГА ВСТРОЕН В КОД БАКТЕРИИ

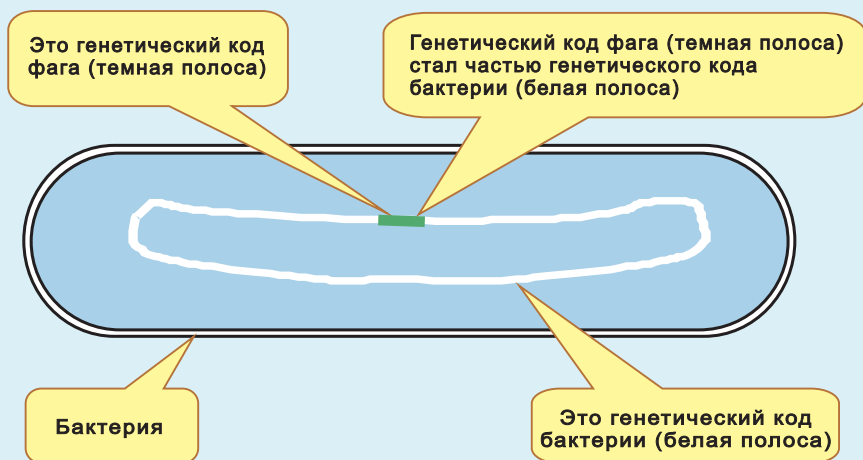


Рис. 191. Размножение фагов не происходит. Генетический код фага (темная полоса) пассивно покоеится внутри кода бактерии (белая полоса).

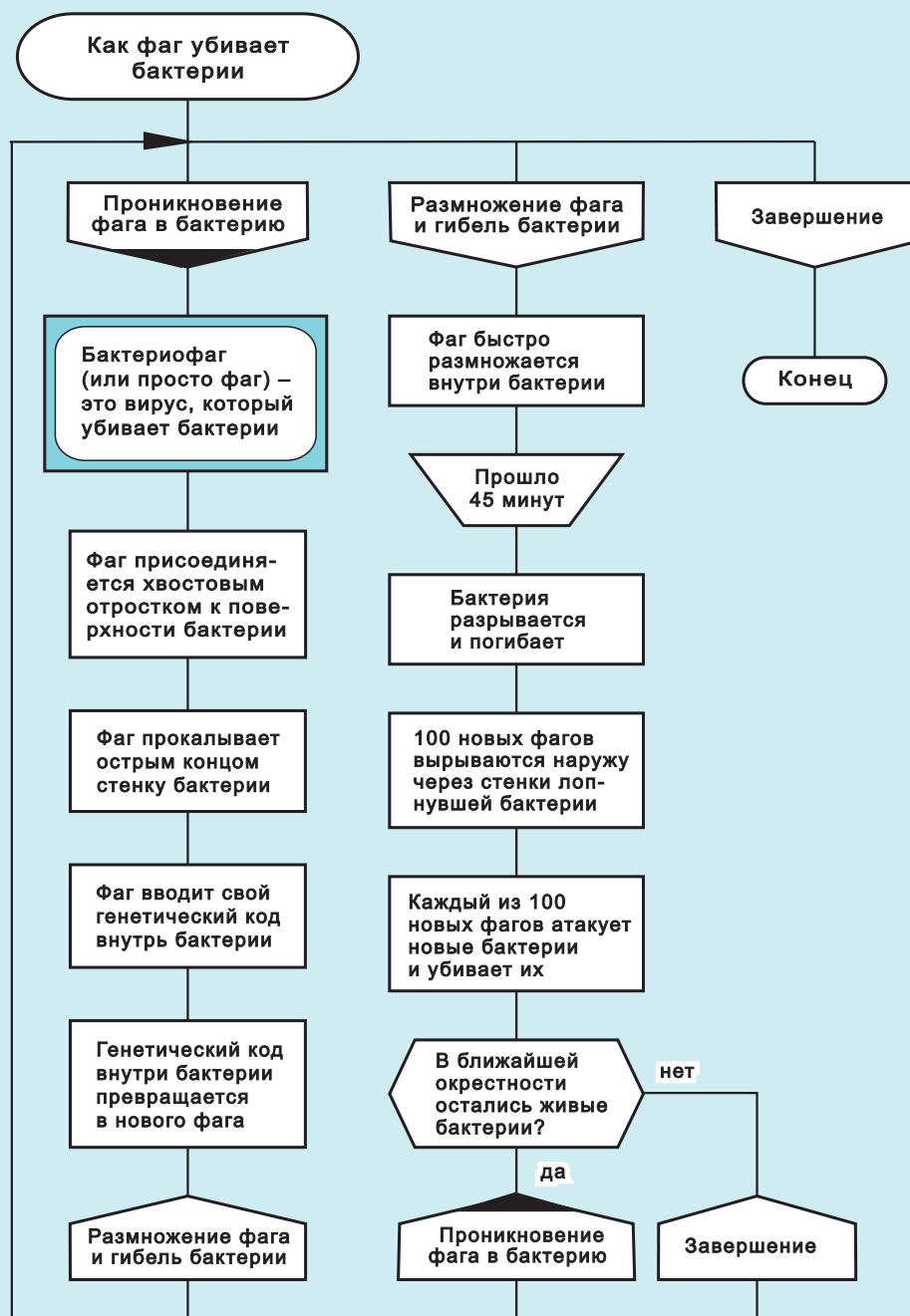
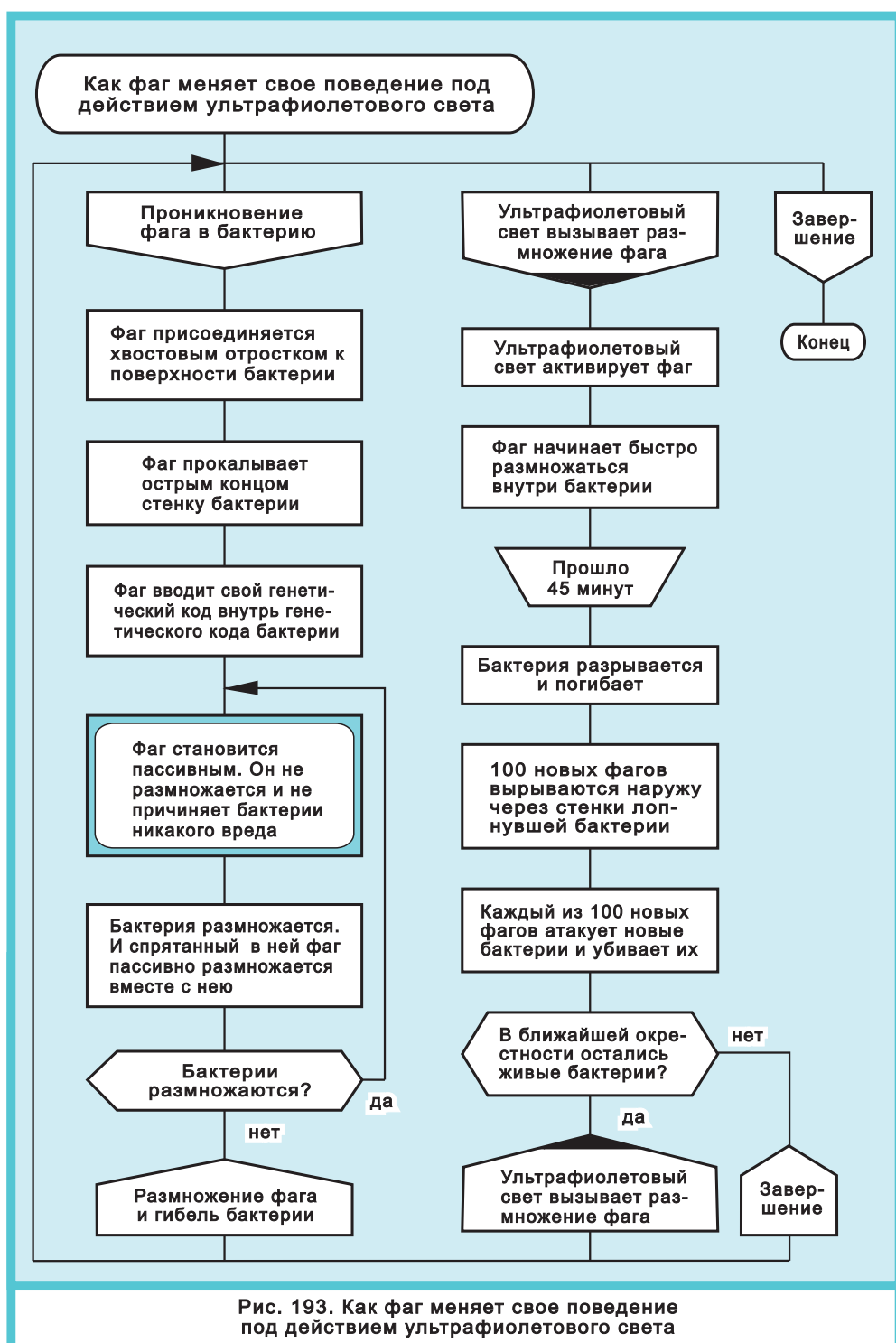


Рис. 192. Как фаг уничтожает бактерии



§8. НУЖНЫ НОВЫЕ СРЕДСТВА ДЛЯ ОПИСАНИЯ БИОЛОГИЧЕСКИХ АЛГОРИТМОВ

Чтобы биологическая наука интенсивно развивалась, она должна овладеть новым языком для описания биологических знаний. Этот язык должен быть очень хорошим. Он должен позволять создавать высококачественные биологические алгоритмы – доступные, удобные, легкие для понимания.

Существуют ли такие алгоритмы? Нет, не существуют! Это вносит значительные трудности в работу биологов.

Имеющиеся биоалгоритмы, как правило, имеют низкое качество. Они изложены трудным, запутанным языком. Они непригодны для эффективного обучения, так как требуют слишком большого времени для понимания. Чтобы поправить дело, нужен новый язык, способный внести в алгоритмы ясность.

§9. ЯЗЫК ДРАКОН И БИОЛОГИЯ

Язык ДРАКОН может оказать существенную помощь биологам. Создание бумажных альбомов и компьютерных библиотек биологических дракон-схем даст возможность улучшить форму представления биологических знаний. Сделать ее более строгой и наглядной. Выявить и устранить алгоритмические пробелы в знаниях. Укрепить позиции информационной биологии. Облегчить дальнейшее исследование таинственного механизма функционирования живых организмов.

§10. НЕОБЫЧНАЯ РОЛЬ СИНХРОНИЗАТОРА

Мы знаем, что икона «синхронизатор» представляет собой перевернутую трапецию с боковым отростком справа (рис. 17, пункт И19). На рис. 186 эта икона повторяется три раза. Но! На этом рисунке икона «синхронизатор» используется необычно, в несвойственной для себя роли. Поясним.

- Верхняя икона «синхронизатор» говорит, что длительность работы предсердий равна 0,1 секунды.
- Средняя икона говорит, что длительность работы желудочков равна 0,3 секунды.
- Нижняя икона – что длительность отдыха сердца равна 0,4 секунды.

Что здесь необычного? А вот что. Внутри иконы указана *длительность* процесса, изображенного справа от синхронизатора – в иконе действие.

Парадокс в том, что на рис. 186 три иконы «синхронизатор» не выполняют никаких функций. Эти три иконы можно безболезненно удалить из схемы (при условии, что длительность будет указана внутри трех икон действие).

Например, внутри верхней иконы действие можно написать: «Предсердия получают кровь из вен и выталкивают ее в желудочки. Длительность этого процесса равна 0,1 секунды». Беда в том, что высказывание получилась слишком длинным и неудобным для чтения. Чтобы облегчить труд читателя, длинную словесную глыбу надо разбить на два предложения. И поместить их в две разные иконы – синхронизатор и действие. Именно так и сделано на рис. 186.

Отсюда следует, что необычная роль иконы «синхронизатор» на рис. 186 состоит в следующем:

- Икона не выполняет никаких действий. Она служит только для того, чтобы разбить текст на мелкие порции и облегчить чтение дракон-схемы.
- Икона «синхронизатор» ничего не синхронизирует. Она выполняет чисто иллюстративную функцию – сообщает читателю *длительность* действия или процесса.

Еще один пример, когда синхронизатор выполняет иллюстративную функцию, приведен в следующей главе (рис. 195).

Таким образом, мы познакомились с *иллюстративной* функцией иконы синхронизатор.

В заключение напомним, что *основная* функция синхронизатора используется при алгоритмизации процессов реального времени. При решении основной задачи надо *указать точный момент времени* по таймеру, чтобы задержать процесс, изображенный справа от синхронизатора. Задержать до тех пор, пока таймер отсчитает время до момента, *указанного в иконе «синхронизатор»*. Основная функция подробно рассмотрена в главах 13 и 14.

§11. ВЫВОДЫ

1. Алгоритмы удобно разделить на две группы:
 - рукотворные (создаваемые человеком и выполняемые компьютером или вручную);
 - нерукотворные (биологические) алгоритмы.
2. Представители биологических наук не владеют эффективными способами записи биологических алгоритмов. Нынешний биологический язык отстал от жизни и превратился в тормоз развития науки.
3. Различные типы алгоритмов (рукотворные и нерукотворные) принято описывать по-разному, с помощью различных и часто неудобных средств.
4. Язык ДРАКОН устраняет этот разноречивый. Он позволяет описывать все алгоритмы *единообразно*, стандартным способом, с помощью одной и той же удобной системы чертежей (одной и той же эргономичной графической нотации).

АЛГОРИТМЫ В СЕЛЬСКОМ ХОЗЯЙСТВЕ

§1. ПОМИДОРЫ НА САДОВОМ УЧАСТКЕ

Выращивание помидоров многим кажется очень простым делом. Так ли это?

С позиций агрономической науки это, конечно, не так. Чтобы получить высокий урожай аппетитных плодов, необходимо тщательно соблюдать технологию посева и ухода за помидорами. А она отнюдь не проста.

Рассмотрим пример. На рис. 194 показана технология (алгоритм) выращивания помидоров на садовом участке.

Алгоритм содержит девять процедур. Три из них раскрыты на рис. 195–197.

На рис. 195 изображен алгоритм «Подготовка семян для проращивания».

На рис. 196 – алгоритм «Посев семян в ящики с грунтом».

На рис. 197 – алгоритм «Уход за сеянцами в ящике».

Алгоритмы, показанные на рис. 194–197, заимствованы из книги [1]. На обложке сказано: «Точное, последовательное и наглядное описание каждого шага, ведущего к обильному урожаю». В этой книге все технологические операции показаны в виде подробных дракон-схем.

§2. ДРАКОН-СХЕМЫ ПОЛЕЗНО ДОПОЛНИТЬ РИСУНКАМИ И КАРТИНКАМИ

Чтобы увеличить наглядность, в книге [1] использован полезный прием. Каждая дракон-схема окружена рисунками. Причем рисунки расположены очень удобно – они буквально прижаты к той дракон-иконе, содержание которой нужно объяснить.

Что это дает? Предположим, в иконе говорится о марлевом мешочке с семенами, к которому приделан ярлычок (рис. 195). Как только читатель пробежал эти слова, он тут же по соседству *видит* этот мешочек, из под резинки которого лихо торчит вверх бумажный ярлычок.

ОБЩИЙ ПЛАН ТЕХНОЛОГИИ ВЫРАЩИВАНИЯ ПОМИДОРОВ

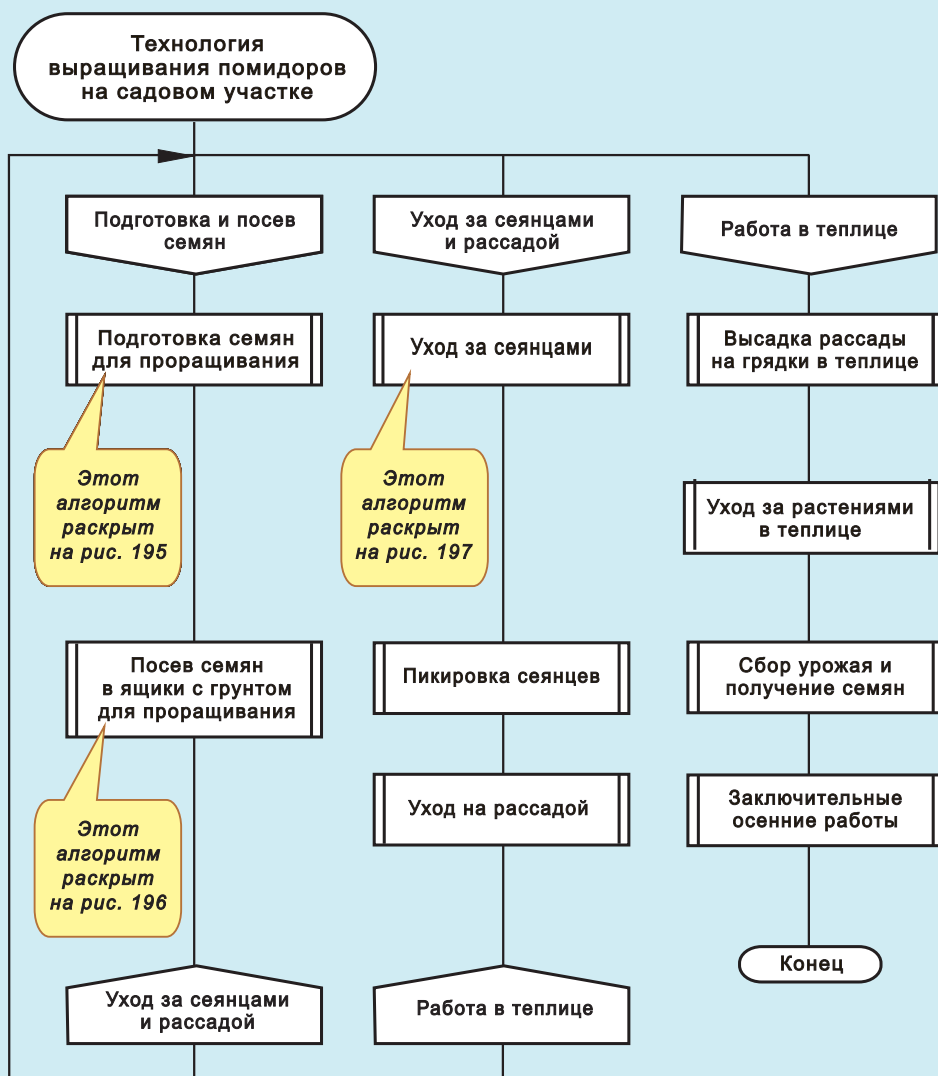


Рис. 194. Технология выращивания помидоров на садовом участке

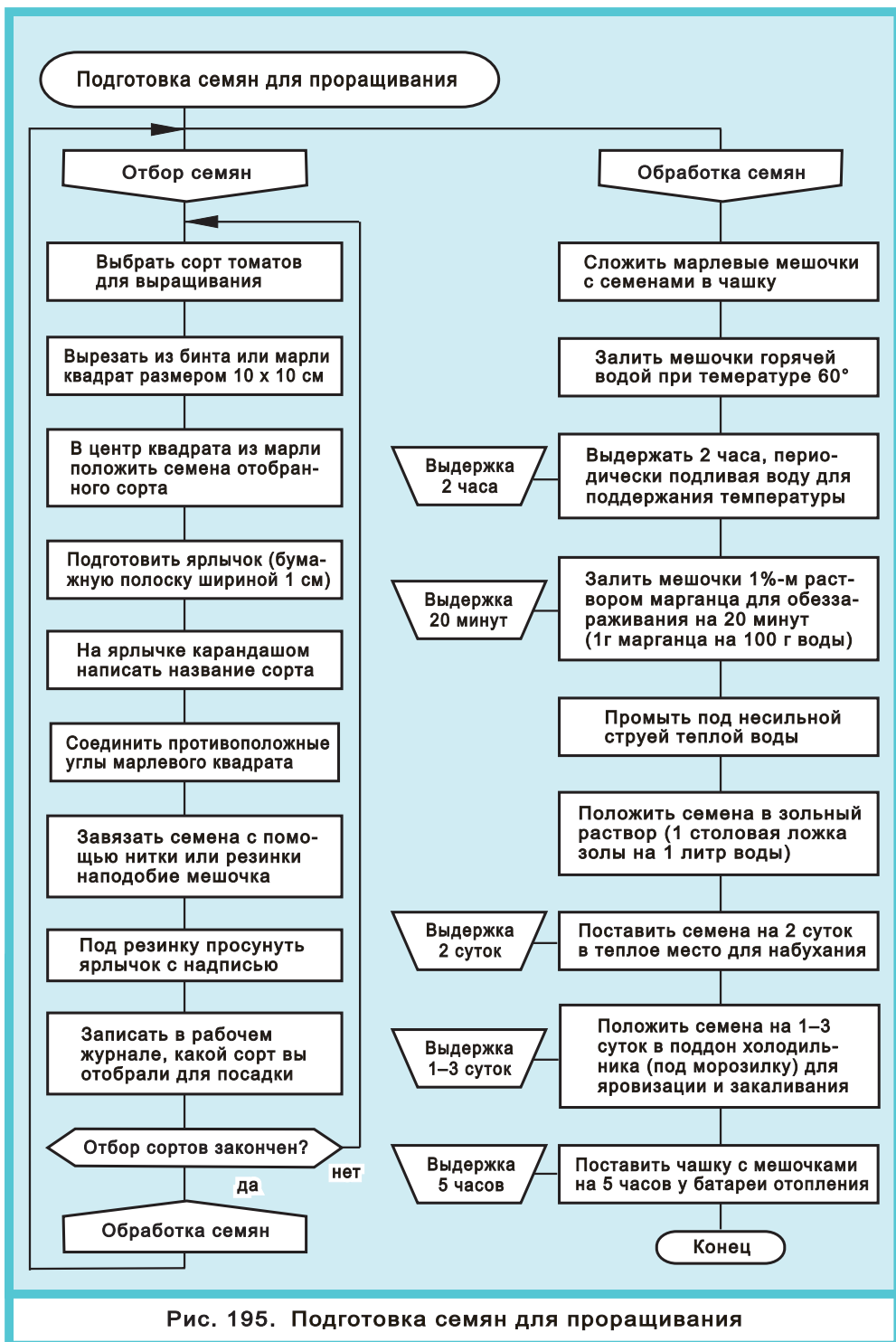


Рис. 195. Подготовка семян для проращивания

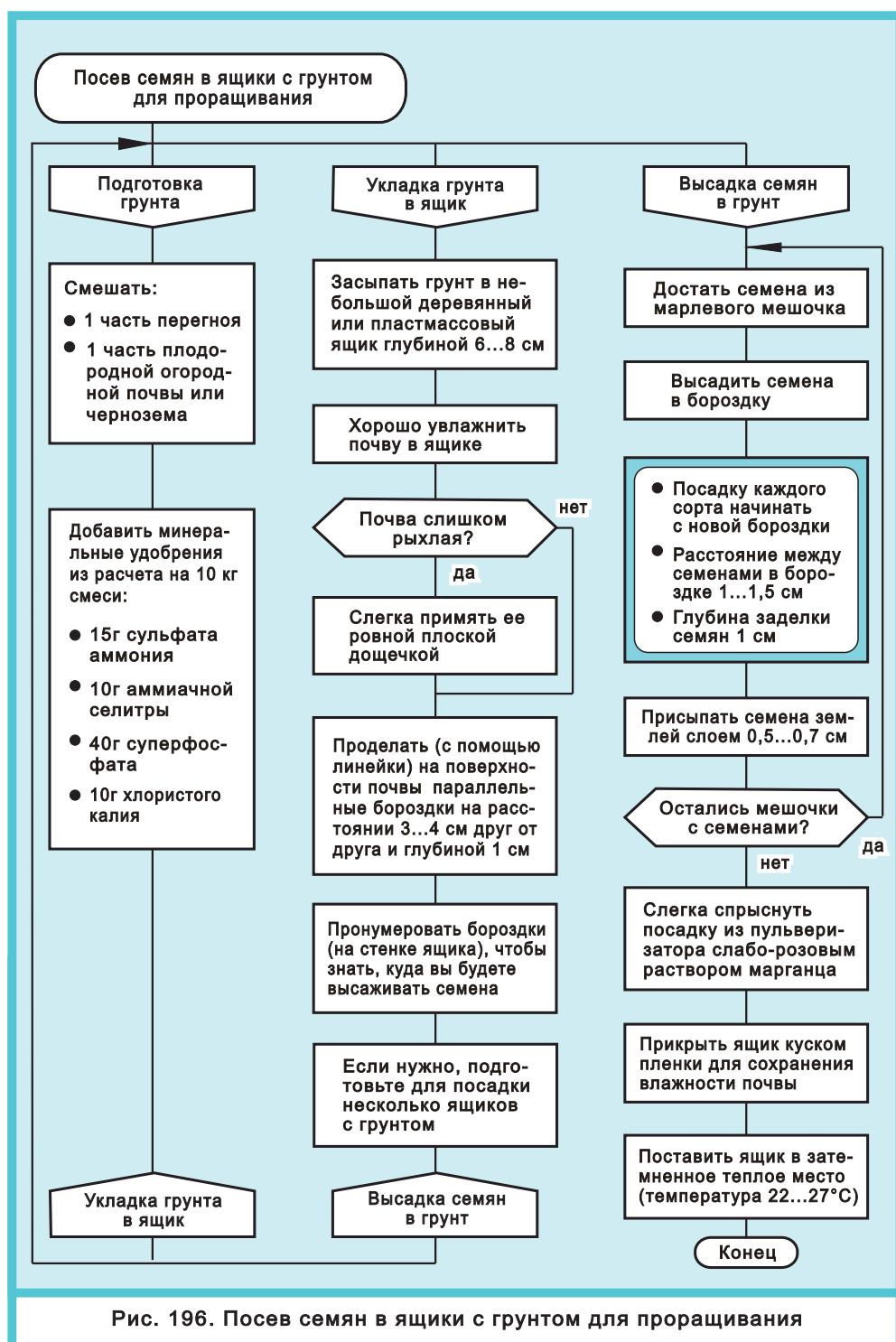


Рис. 196. Посев семян в ящики с грунтом для проращивания

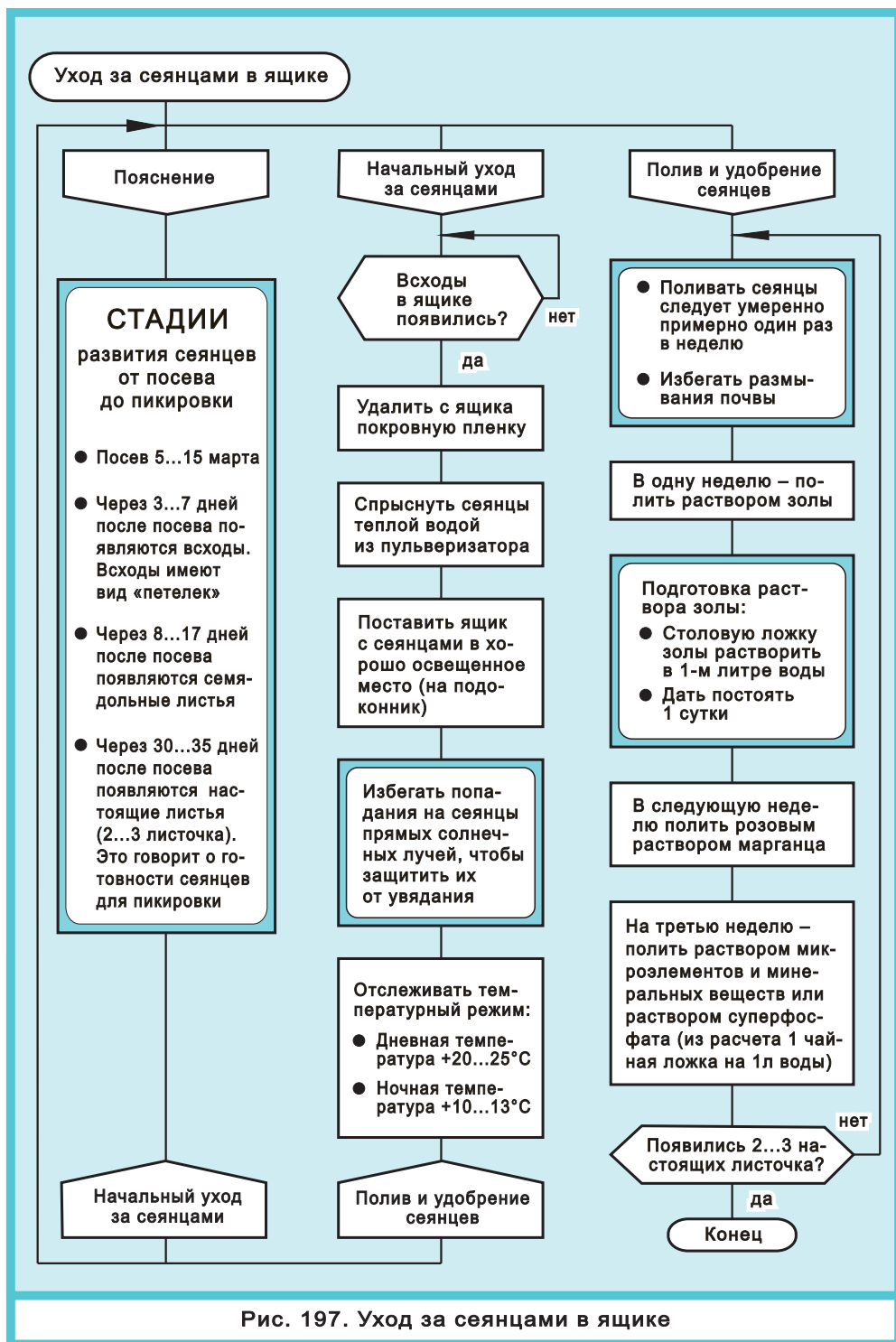


Рис. 197. Уход за сеянцами в ящике

Еще пример. На рис. 197 в первой ветке говорится, что «всходы имеют вид петелек». Рядом нарисованы эти самые петельки.

Дальше речь идет о семядольных и настоящих листьях. Что это за листья?

У читателей такой вопрос не возникает. В книге [1] прекрасно видны эти листья – на рисунках, дополняющих дракон-схему. Так что отличие между семядольными и настоящими листьями сразу бросается в глаза. Читатель не затрачивает никаких усилий, чтобы уяснить эти понятия.

§3. ГРАФИЧЕСКИЙ КОММЕНТАРИЙ

Дракон-схема – это графический образ. Естественно возникает мысль: если уж речь зашла о графике, надо использовать ее не частично, а в полной мере.

Такую задачу можно решить с помощью иконы «комментарий». В ДРАКОНе возможен не только традиционный (текстовый), но и графический комментарий. Из-за недостатка места, в этой главе не удалось разместить графический комментарий. Но это не беда. Нужный пример можно посмотреть в другом месте – см. главу 24, рис. 198. В той же главе показан и комментарий в виде математических формул (формульный комментарий).

§4. ВЫВОДЫ

1. Язык ДРАКОН способен изображать биологические алгоритмы. Это позволяет использовать его в сельском хозяйстве – для описания алгоритмов, которые используются в агрохимических и агрофизических науках, а также в почвоведении.
2. Другим важным приложением ДРАКОНа является применение его для наглядного описания сельскохозяйственных технологий.

АЛГОРИТМЫ В СРЕДНЕЙ ШКОЛЕ

§1. ДРАКОН ПОМОГАЕТ ИЗУЧАТЬ МАТЕМАТИКУ

На рис. 198 представлена школьная задача по геометрии. Эта задача структурирована по правилам языка ДРАКОН. В первой ветке дано условие задачи с чертежом. Во второй – вычисление площади круга и треугольника. В третьей – вычисление искомой площади S .

Чем отличается решение математической задачи с помощью дракон-схемы от традиционной записи? Выделим три важных отличия.

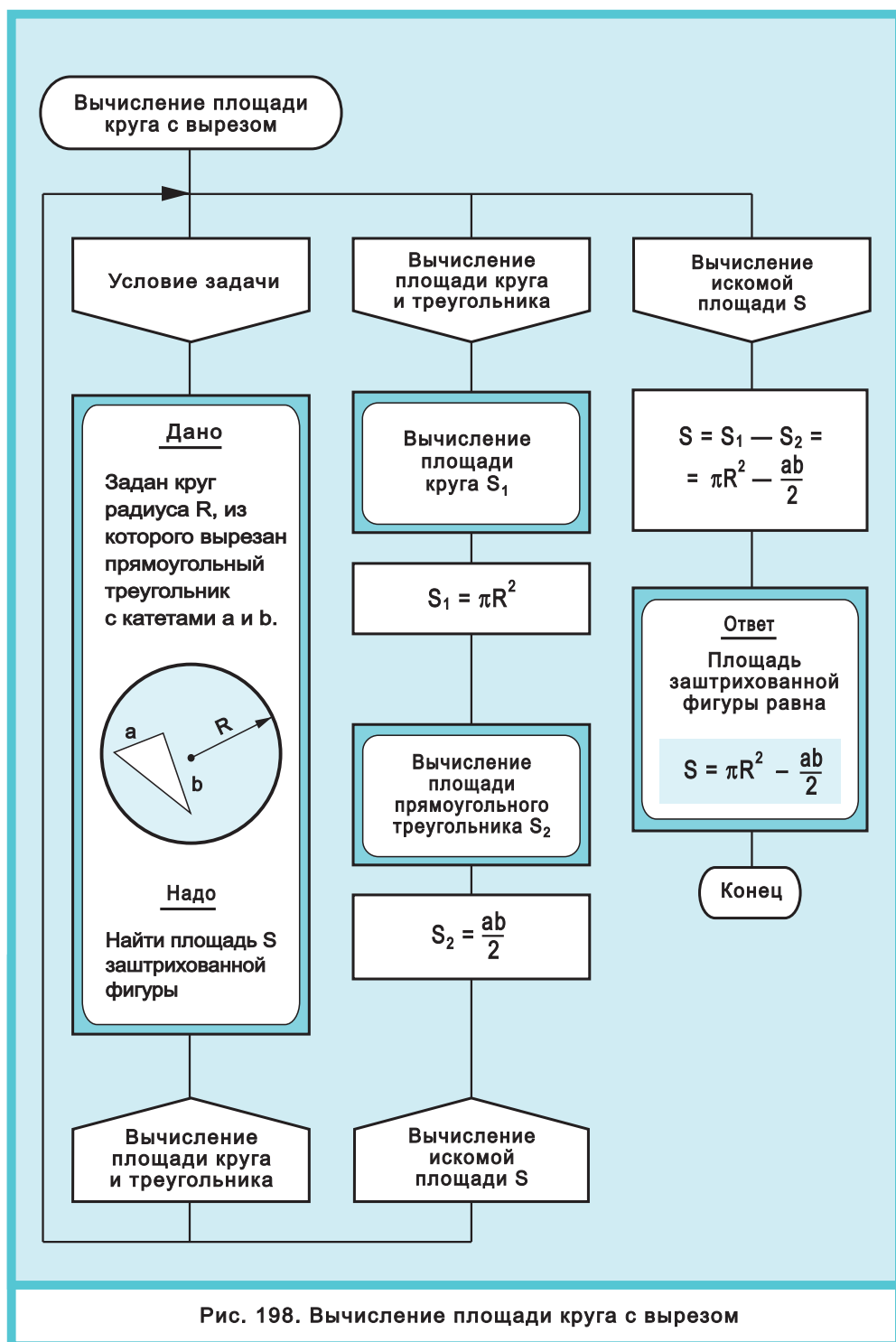
- Обеспечена simultaneity восприятия: на одной зрительной сцене находятся: 1) условие задачи; 2) решение; 3) ответ.
- Зрительная сцена имеет предельно четкую структуру. Она упорядочена и по вертикали, и по горизонтали.
- Все этапы решения и формулы имеют разъясняющие словесные заголовки. Последние записываются не где угодно, а в специальных рамочках, каждая из которых «знает свое место».

Рассмотрим еще одну задачу. На рис. 199 показана дракон-схема, описывающая решение квадратного уравнения. Она позволяет не только найти корни уравнения, но и проверить их с помощью формул Виета (см. ветку «Проверка корней»).

§2. СОВЕТЫ УЧИТЕЛЮ

Желательно, чтобы школьники научились:

- рассматривать математическое решение задачи как зрительную сцену;



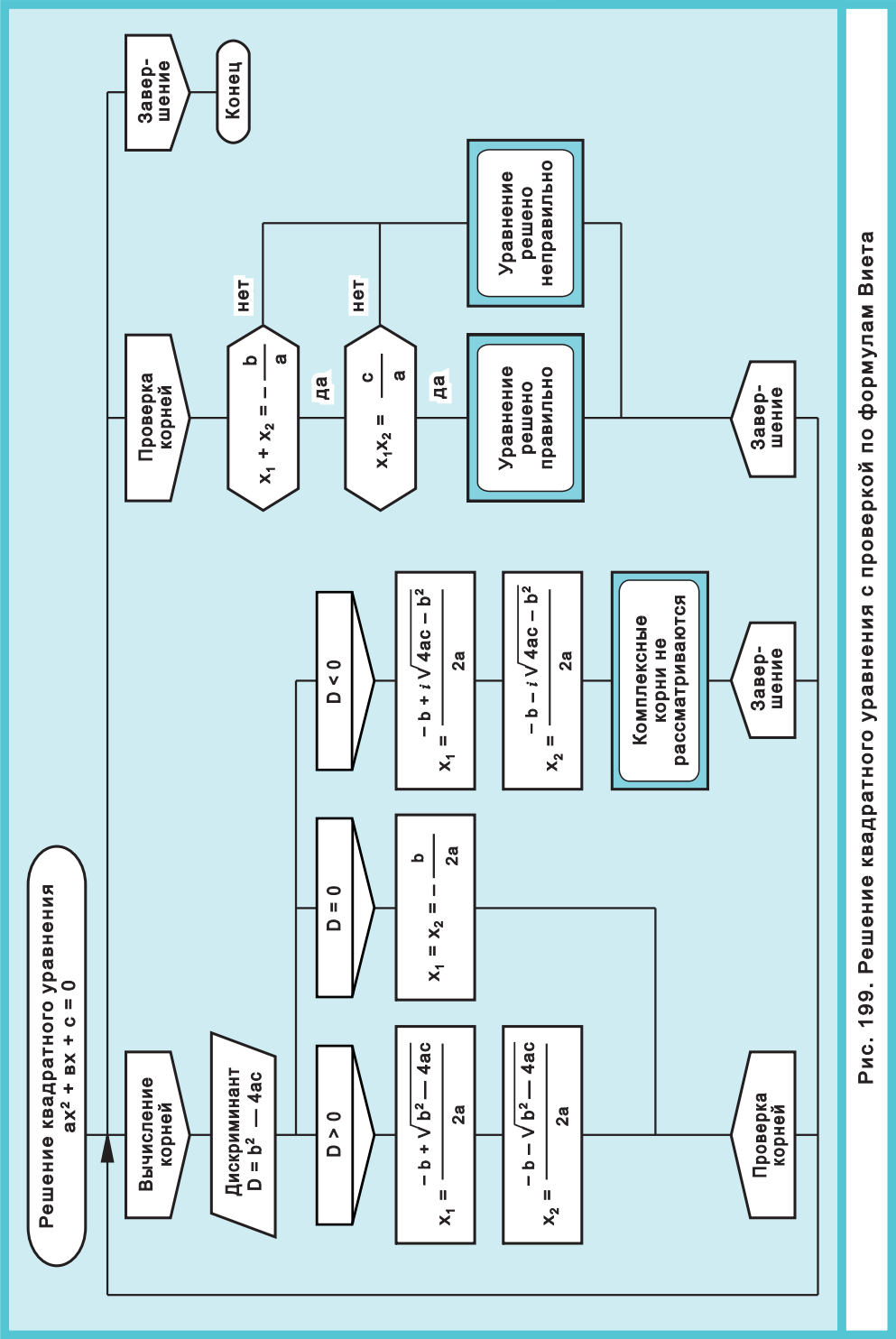


Рис. 199. Решение квадратного уравнения с проверкой по формулам Виета

- правильно строить математическую зрительную сцену на плоскости чертежа;
- изображать решение математических задач, выявляя их визуальную структуру;
- структурировать задачу в пространстве чертежа по правилам языка ДРАКОН;

Иначе говоря, необходимо изучить правила визуального мышления, которые лежат в основе языка ДРАКОН.

Желательно научить школьников не только разбивать ход решения задачи на отдельные этапы, но и придумывать для них краткие и точные заголовки. Ученик должен знать, что эти заголовки следует записывать в шапке дракон-схемы (в иконах «заголовок» и «имя ветки»).

§3. О ПОЛЬЗЕ ЭРГОНОМИЧНЫХ СТРЕЛОК В МАТЕМАТИКЕ

Рассмотрим алгоритм на рис. 200, который содержит «маленький секрет», связанный с числом 4. Действительно, в условии задачи мы видим 4 отдельных условия (см. икону «комментарий»). В ветке «Решение задачи» имеются 4 вертикали, помеченные буквами *A*, *B*, *C*, *D*. Таким образом, в данной задаче действует

Правило. Число условий равно числу вертикалей. Причем каждому условию соответствует своя вертикаль.

Копнув глубже, можно заметить, что в задаче выполняется закономерность:

- каждому условию соответствует свое решение,
- каждое решение изображается отдельной вертикалью.

В обычных условиях указанные закономерности невидимы – они скрыты от читателя. Человек может правильно решить задачу, даже не заметив, что у него под носом затаился «маленький секрет». Это значит, что он решил задачу поверхностно, не проникнув в ее тайные глубины.

Чтобы устранить недостаток, надо сделать читателю небольшую, но умную эргономическую подсказку. Такой подсказкой служат 4 широкие голубые горизонтальные стрелки. Каждая из них устанавливает зрительную связь между условием и соответствующей вертикалью. Эти стрелки сразу бросаются в глаза и помогают читателю разгадать «секрет».

Разумеется, можно обойтись и без стрелок. Но если мы хотим проявить заботу о читателе, облегчить его утомительный труд и оказать ему дополнительную помощь, то стрелки и многие другие эргономические мелочи, несомненно, будут полезны.

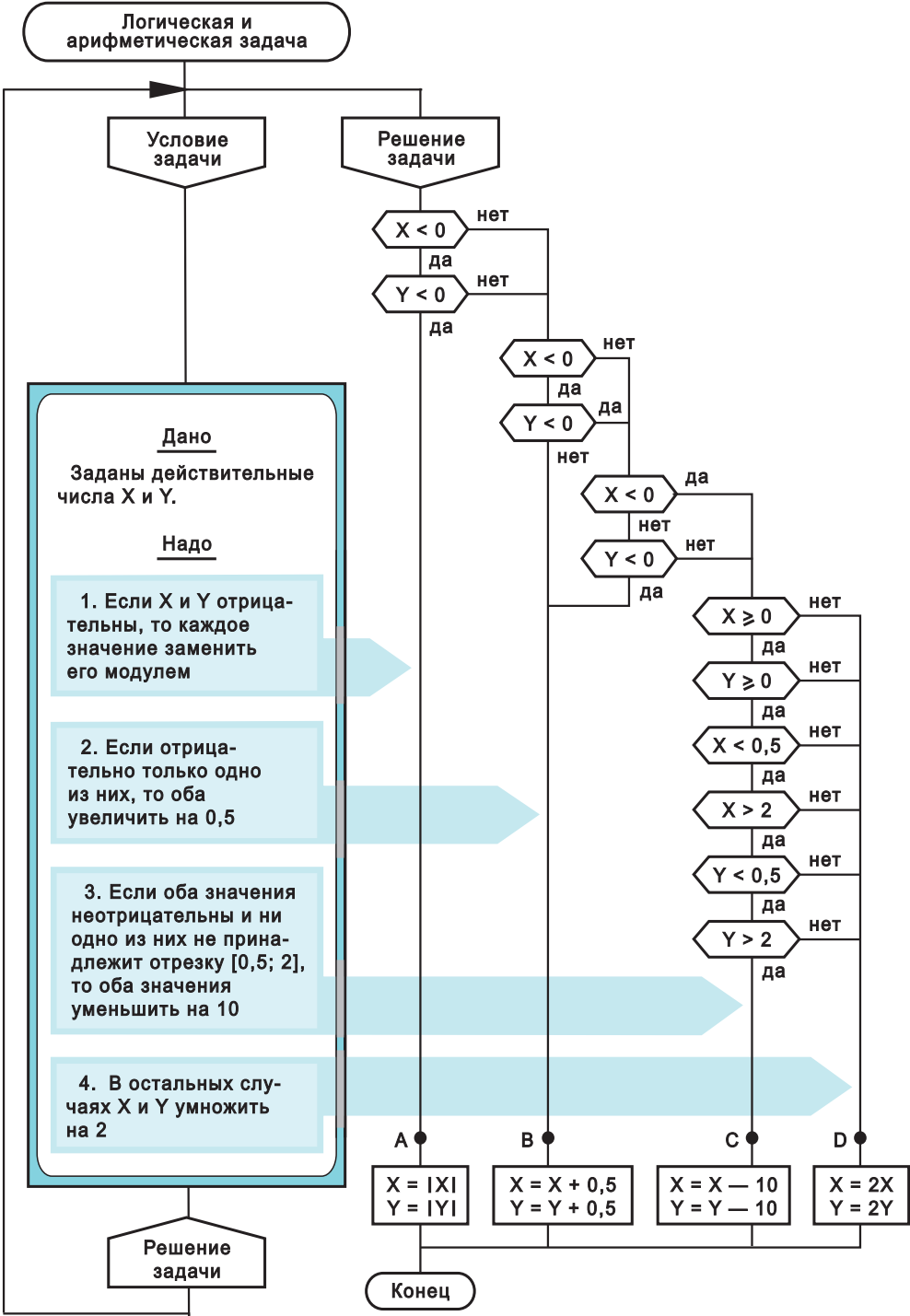


Рис. 200. Сложная логическая задача с простыми вычислениями

Эту мысль следует подчеркнуть особо. С логической точки зрения, указанные стрелки являются «архитектурным излишеством» и совершенно не нужны для решения задачи.

Но с эргономической точки зрения, дело обстоит иначе.

Цель эргономики – облегчение умственного труда, минимизация интеллектуальных усилий, достижение «легкости и глубины мышления». Именно это и достигается. Указывая на нужную вертикаль, стрелки помогают читателю проникнуть в глубинную суть задачи.

Каждая вертикаль имеет два «золотых» конца: верхний и нижний. Верхний конец содержит алгоритмическую запись *условия*, нижний – *решения*. Все это математическое богатство и открывается ученику благодаря стрелкам.

Если от правого конца стрелки читатель поднимет глаза вверх, он увидит алгоритмическую реализацию условия. Если же посмотрит вниз – увидит решение. Поскольку стрелок четыре, подобную операцию придется проделать четыре раза. Но после этих упражнений структура алгоритма станет абсолютно прозрачной, ясной и наглядной (рис. 200).

§4. ЕЩЕ ОДИН ЭРГОНОМИЧНЫЙ АЛГОРИТМ

Разберем задачу на рис. 201. Это «провокационная» задача, так как она имеет не один, а шесть ответов. Чтобы «раскопать» правильные ответы, нужно рассмотреть восемь (!) вариантов решения. И отбросить два из них как бессмысленные.

Типичная ошибка состоит в том, что учащиеся не видят эти восемь вариантов решения. И по этой причине упускают из виду один или несколько ответов.

На рис. 201 решение задачи изображено в виде чертежа (дракон-схемы). При этом используется ряд эргономических приемов, облегчающих понимание.

- Чертеж создает целостный графический образ решения, в котором иконы связаны между собой соединительными линиями.
- Зрительная сцена буквально заставляет учащихся учесть все варианты, так как пропущенный вариант порождает оборванную линию типа «висячий хвост». Последняя сразу же будет опознана как ошибка и исправлена.
- Схема упорядочена по вертикали таким образом, что «бегунок», двигаясь по линиям сверху вниз, перемещается от условия задачи к одному из ответов. Значит, зрительная сцена организована не хаотично, а по правилу «Вверху – *условие задачи*, внизу – *ответы*».

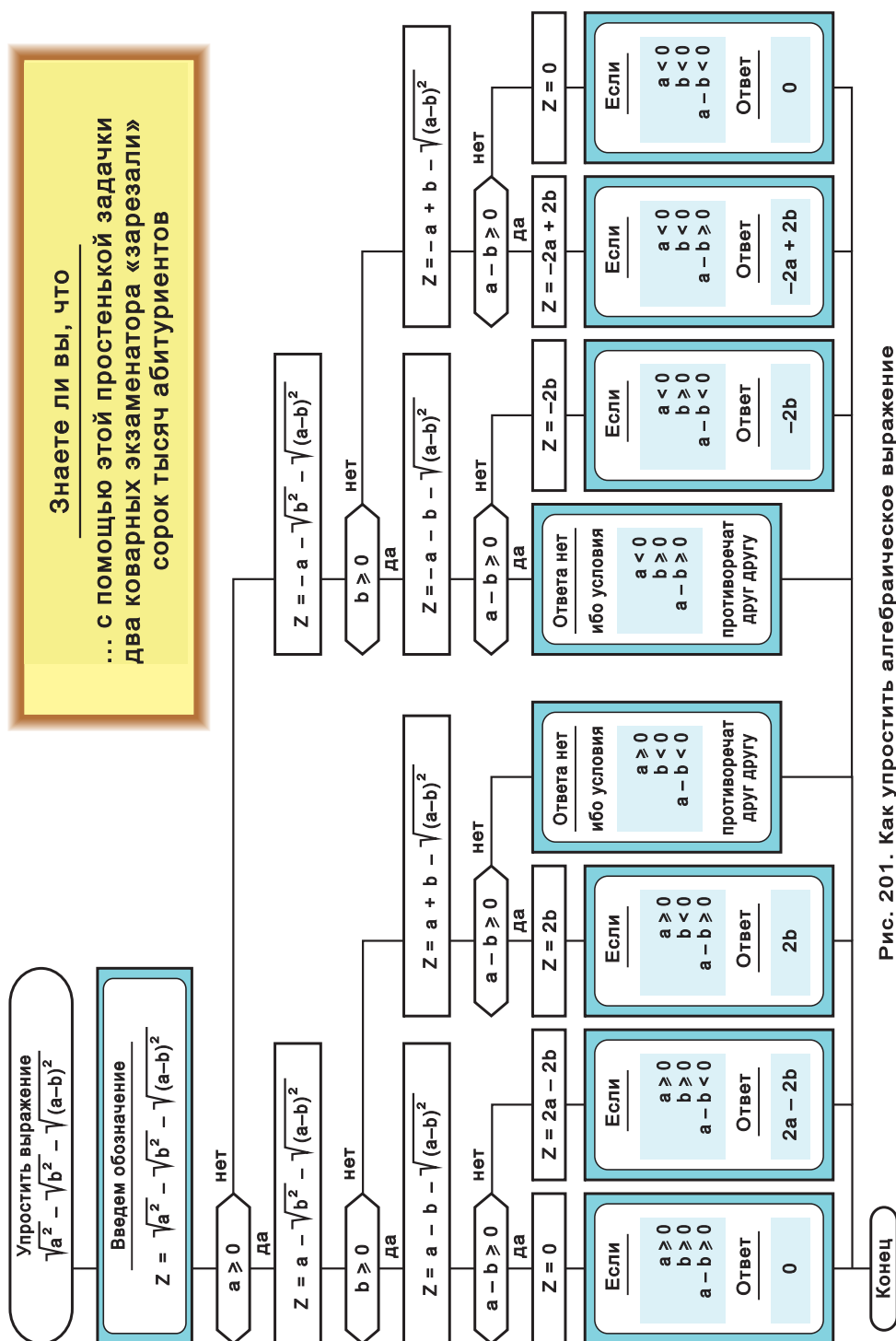


Рис. 201. Как упростить алгебраическое выражение

- Схема упорядочена и по горизонтали. На одной горизонтали расположены однородные иконы, внутри которых находятся близкие по смыслу текстовые надписи. Это дисциплинирует чертеж, делает его зрительно предсказуемым, облегчает чтение и понимание.
- Чтобы облегчить зрительное различение хороших и плохих вариантов, нижняя горизонталь (горизонталь ответов) расщеплена на два уровня, слегка смещенных друг относительно друга.
- На нижнем уровне размещены шесть хороших вариантов, содержащих правильные ответы. Чуть выше находятся два плохих варианта, которые не содержат ответов.

Перечисленные эргономические приемы помогают читателю упорядочить свои мысли, привести их в стройную систему. В итоге сложная задача, словно по волшебству, упрощается, становится наглядной и обозримой.

Мы убедились, что с помощью языка ДРАКОН можно упростить алгебраическое выражение.

§5. ДРАКОН ПОМОГАЕТ ИЗУЧАТЬ ХИМИЮ

На рис. 202 показана схема простейшего химического опыта – определение кислотности раствора. Если лакмусовая бумажка, погруженная в раствор, краснеет, значит, раствор кислый. Если синеет – щелочной. Если цвет бумажки не изменился – раствор нейтральный.



Рассмотрим сложный химический опыт (рис. 203). Учитель берет одно из шести химических удобрений и помещает его в колбу. Что в колбе – неизвестно. Это может быть аммиачная селитра, натриевая соль, сульфат аммония, суперфосфат, сильвинит или калийная соль.

Нужно выполнить эксперимент, позволяющий узнать, какое вещество находится в колбе.

Опыт делают в два этапа. Сначала идут подготовительные действия:

1. Положить удобрение в три сосуда.
2. В первый сосуд добавить серную кислоту H_2SO_4 .
3. Во второй сосуд добавить раствор хлорида бария BaCl .
4. В третий сосуд добавить щелочь.

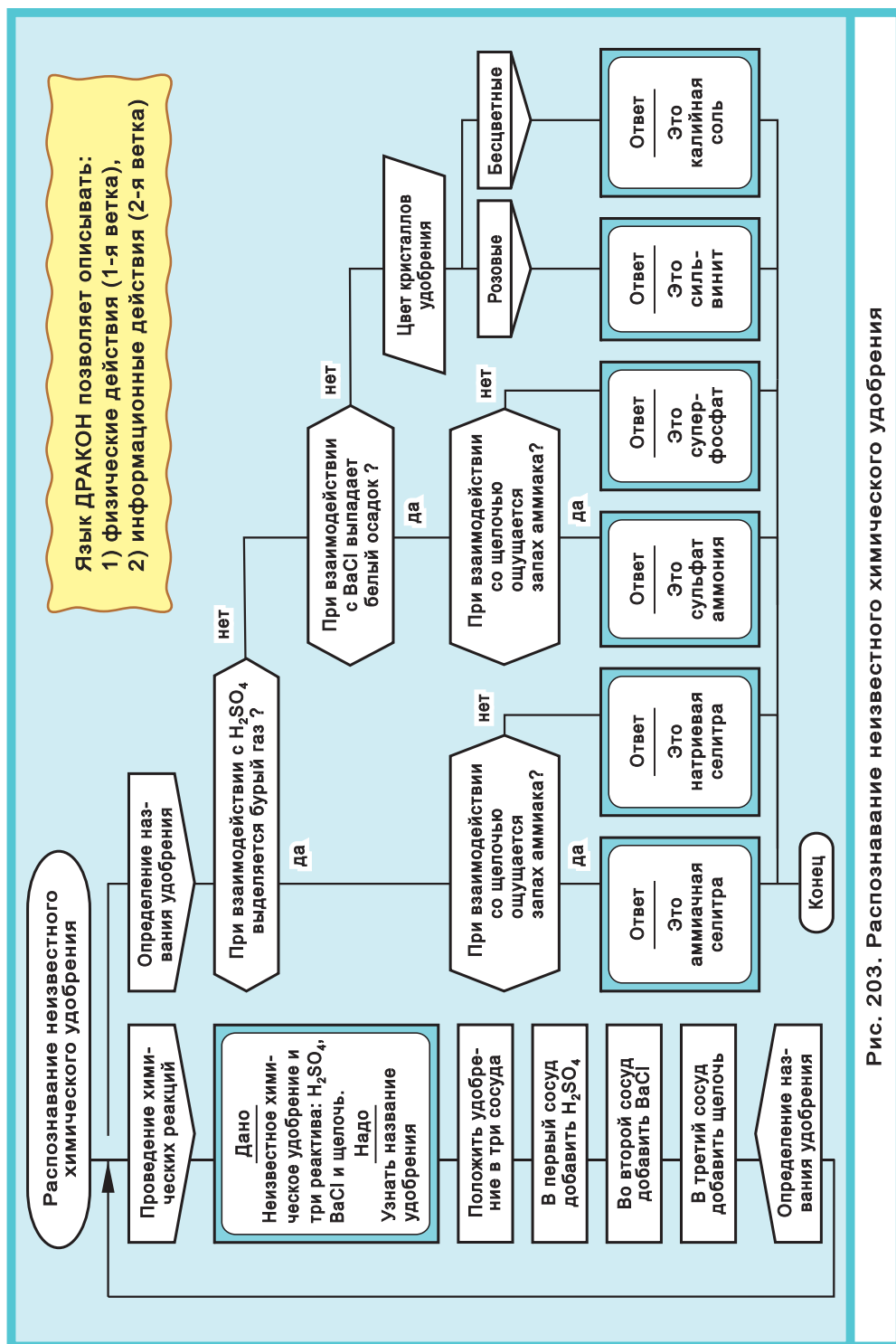
После этого анализируются результаты и определяется неизвестное вещество на основании таблицы 1.

Таблица 1

Название вещества	Внешний вид вещества	Что мы видим или ощущаем, если в данное вещество добавить:		
		H_2SO_4	BaCl	Щелочь
Аммиачная селитра	Белые кристаллы	Бурый газ	–	Запах аммиака
Натриевая селитра	Бесцветные кристаллы	Бурый газ	Помутнение	–
Сульфат аммония	Светло-серые кристаллы	–	Белый осадок	Запах аммиака
Суперфосфат	Светло-серый порошок	–	Белый осадок	–
Сильвинит	Розовые кристаллы	–	–	–
Калийная соль	Бесцветные кристаллы	–	–	–

Дракон-схема на рис. 203 работает так. Рабочая точка процесса движется от иконы «заголовок» к иконе «конец». По мере движения иконы и соединительные линии вспыхивают и горят на экране более ярким светом, выделяя пройденную часть пути.

Когда процесс дошел до иконы-вопрос «При взаимодействии с H_2SO_4 выделяется бурый газ?», данная икона начинает мерцать, привлекая к себе внимание и требуя ответа. Реагируя на это событие, ученик подводит курсор к нужному ответу (да или нет) и щелкает клавишей мыши.



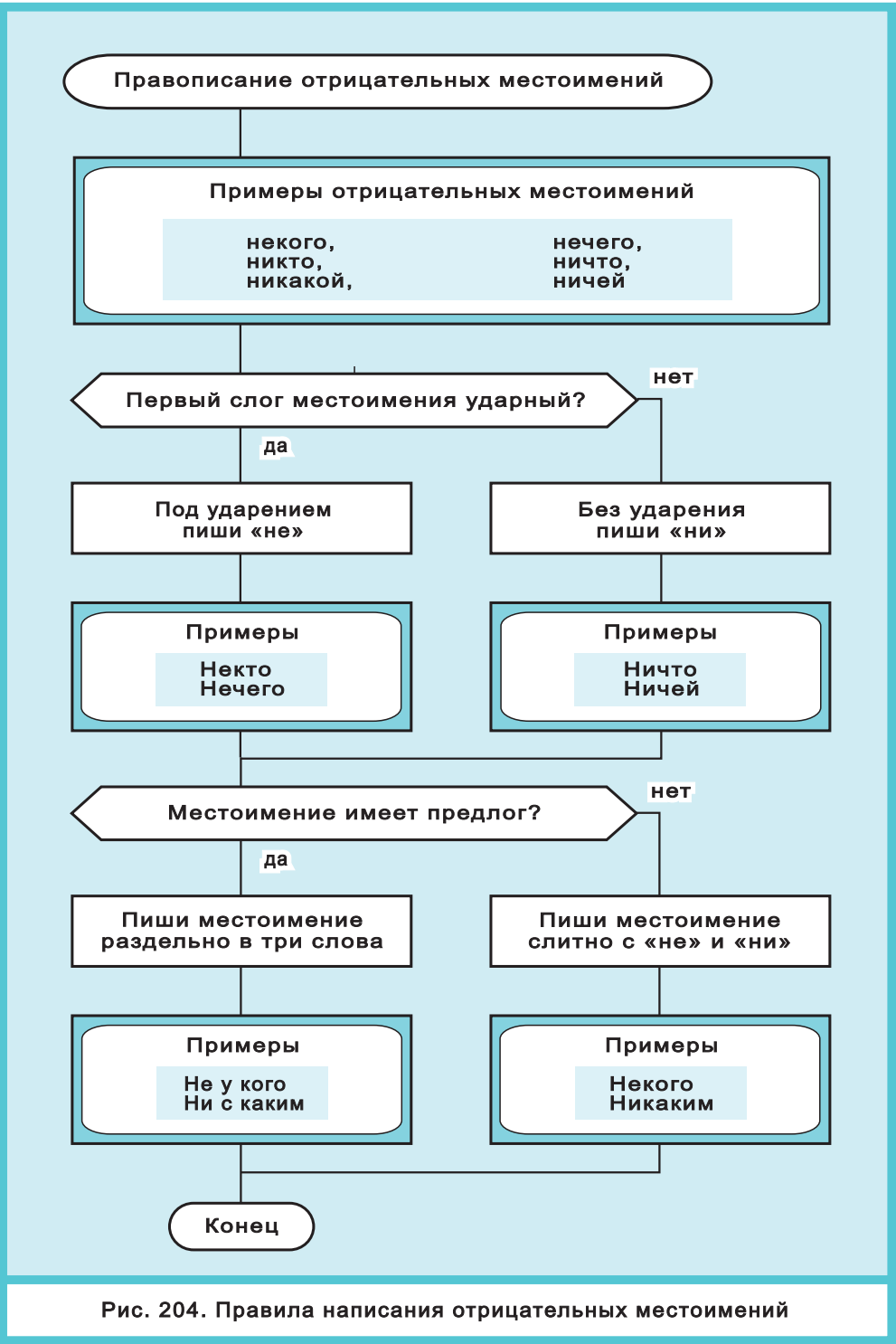


Рис. 204. Правила написания отрицательных местоимений

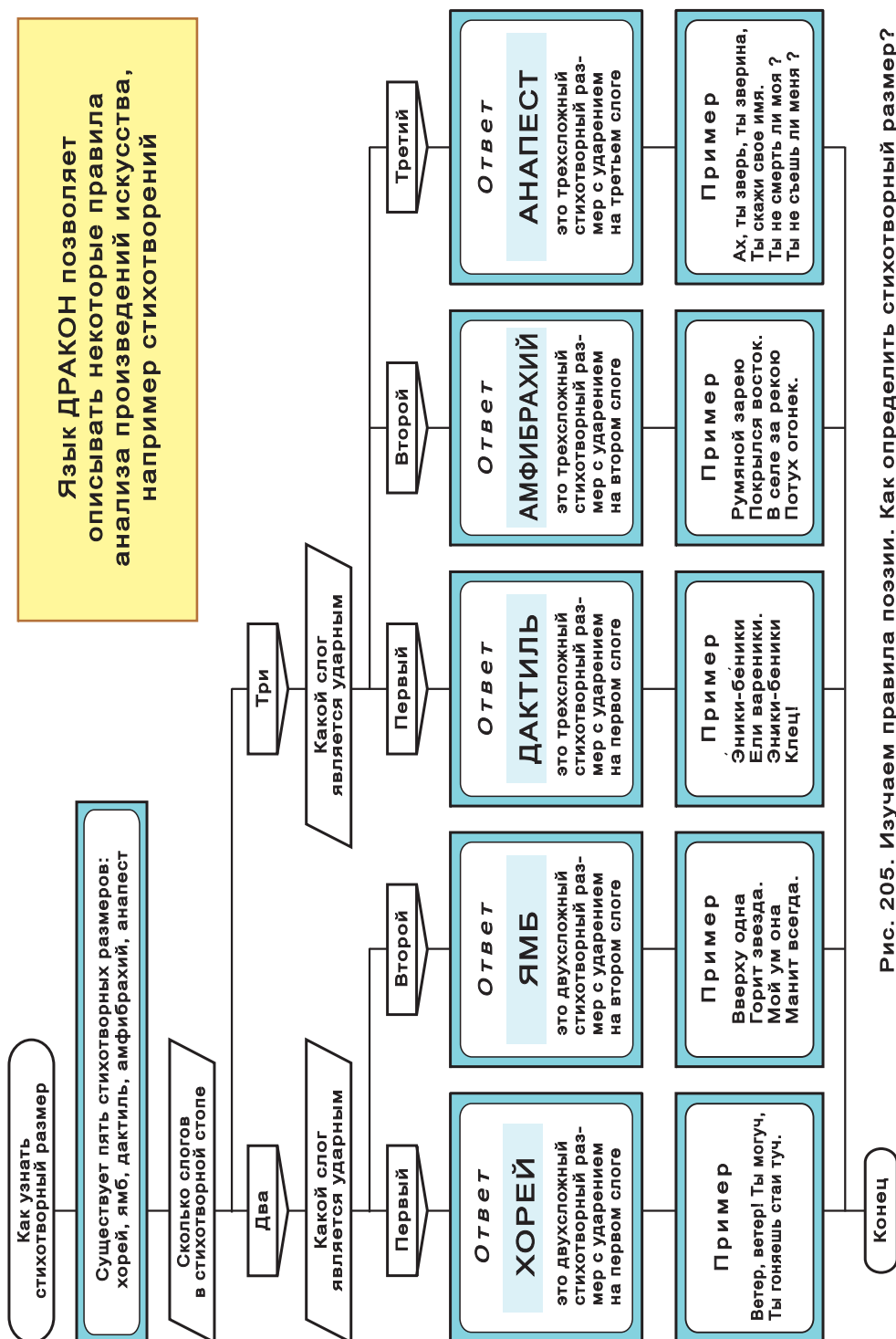


Рис. 205. Изучаем правила поэзии. Как определить стихотворный размер?

Икона перестает мерцать и (при ответе «да») загорается путь, ведущий к иконе «При взаимодействии со щелочью ощущается запах аммиака?», которая начинает мерцать. Далее события повторяются, пока на экране не загорится искомое название удобрения.

С точки зрения процесса познания, проблема распознавания удобрения состоит из трех частей:

- постановка задачи;
- описание действий с исследуемым веществом и реактивами;
- логический анализ результатов опыта.

В первой ветке даны постановка задачи (икона «комментарий») и описание последовательности ручных манипуляций (четыре иконы «действие»). Во второй ветке демонстрируется логический анализ и получение ответа.

Важно, что все три части проблемы предъявляются зрителю в одном визуальном поле. Благодаря этому обеспечивается симультанность восприятия и улучшение работы ума.

§6. ДРАКОН ПОМОГАЕТ ИЗУЧАТЬ ГУМАНИТАРНЫЕ ПРЕДМЕТЫ

Приведем еще пару примеров, подтверждающих достоинства языка ДРАКОН. Эти примеры демонстрируют возможность его применения в различных сферах человеческой деятельности.

Рис. 204 иллюстрирует использование языка для изображения грамматических правил. На рис. 205 показан пример формализации простейших правил анализа стихотворений.

§7. ВЫВОДЫ

1. Участь в школе, ученик получает знания из различных областей знания. Или, как говорят, он изучает несколько предметов.
2. Между этими предметами существуют связи, но ученик, как правило, их не замечает. Он не видит *межпредметные связи*.
3. Чтобы устранить недостаток, надо специально объяснить школьникам, в чем именно заключаются межпредметные связи.
4. Одним из эффективных «мостиков» между предметами являются алгоритмы и язык ДРАКОН.
5. В разных предметах для решения многих, хотя и не всех задач используются алгоритмы.
6. Алгоритмы и дракон-схемы представляют собой удобное средство, облегчающее для школьника выявление и уяснение сути межпредметных связей.
7. Благодаря алгоритмам межпредметные связи становятся зримыми и понятными для учащихся.

АЛГОРИТМЫ ГОСУДАРСТВЕННОГО И МУНИЦИПАЛЬНОГО УПРАВЛЕНИЯ

§1. НЕДОСТАТКИ АЛГОРИТМОВ УПРАВЛЕНИЯ

Журден, герой Мольера, очень удивился, когда узнал, что всю жизнь говорил прозой. Многие управленцы высшего, среднего и низшего звена чем-то похожи на Журдена. Они принимают решения по той или иной технологии, но, как правило, не обращают на это внимания.

Говоря подробнее, можно отметить три недостатка.

- Чиновники-управленцы далеко не всегда осознают, что «говорят прозой», то есть используют определенную технологию управления, точнее говоря, *алгоритмы управления*.
- Во многих случаях они не умеют выделять в этой технологии стандартные и нестандартные участки.
- Они не рассматривают процессы выработки управленческих решений как алгоритмические процессы. И не умеют описывать эти процессы в алгоритмической форме.

Социально-гуманитарные знания (в том числе знания по государственному, муниципальному и корпоративному управлению) принято записывать в виде линейного текста, который слишком труден для понимания [1, 2]. Если время изучения текста ограничено (что бывает почти всегда, так как у делового человека каждая минута на счету), линейный текст провоцирует непонимание.

Мировой опыт разработки языков программирования показывает, что линейный текст (включая ступенчатый) не пригоден для записи понятных алгоритмов.

Традиционный способ записи алгоритмов неоправданно сложен и непригоден для массового использования в управленческих структурах. Попытки его внедрения бессмысленны, ибо требуют огромных трудозатрат, превышающих разумные пределы.

§2. АЛГОРИТМЫ УПРАВЛЕНЧЕСКИХ РЕШЕНИЙ

Укажем особенности предлагаемого подхода.

- Исключая из рассмотрения декларативные знания, ограничимся только процедурными знаниями. Для простоты будем считать, что процедурные знания и алгоритмы – одно и то же.
- Вместо существующих методов записи алгоритмов (которые слишком сложны и непригодны для целей государственного, муниципального и корпоративного управления) предлагается новый, значительно более легкий способ записи алгоритмов.
- Алгоритмы управления надо изображать не в виде линейного или ступенчатого текста, а в виде понятного графического чертежа.
- Предлагаемый чертеж (дракон-схема) похож на блок-схему алгоритма (ГОСТ 19.701–90), но выгодно отличается от нее более эффективной структуризацией, математической строгостью и повышенным комфортом для пользователей.

§3. ЯЗЫК ДРАКОН В УПРАВЛЕНИИ

Язык ДРАКОН значительно превосходит другие языки по критерию «понятность» (эргономичность) алгоритмов.

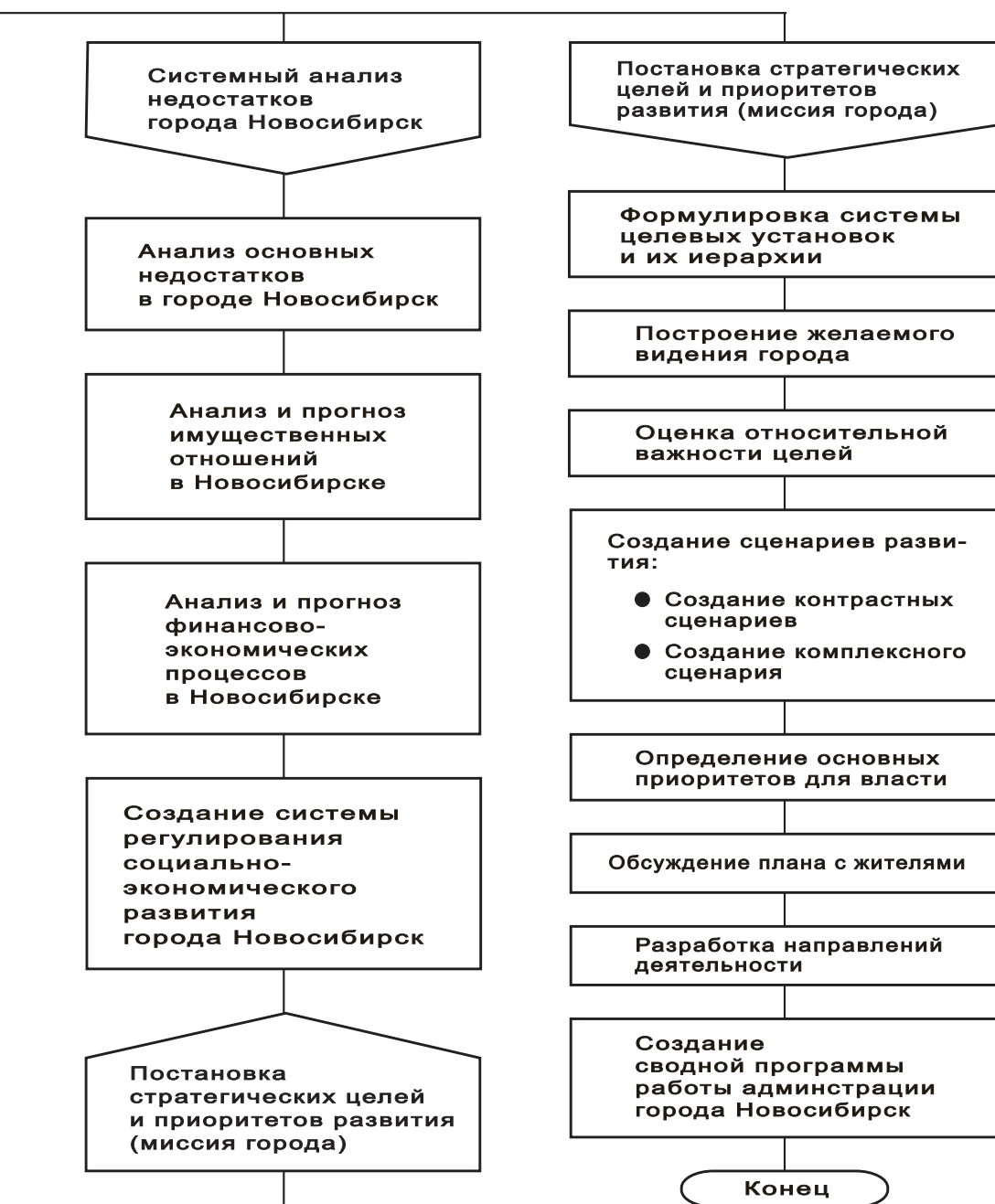
Рассмотрим пример, позволяющий дать зрительный образ алгоритма управления, представленного на языке ДРАКОН. На рис. 206 показан алгоритм «Создание концепции социально-экономического развития Новосибирска».

В данном случае чертеж алгоритма является математически строгим, а надписи в блоках даны на естественном языке. Это очень удобно при решении многих задач управления.

Большое число дракон-схем из области стратегического планирования представлены в работе [3].



Рис. 206. Создание концепции социально-экономического развития Новосибирска



§4. ВЫВОДЫ

1. Современные системы государственного, муниципального и корпоративного управления чрезвычайно сложны и трудны для понимания.
2. Непонимание приводит к принятию неоптимальных, неэффективных или ошибочных управленческих решений.
3. Управленцы далеко не всегда осознают, что используют ту или иную технологию управления, точнее говоря, *алгоритмы управления*.
4. Они не рассматривают процессы выработки управленческих решений как алгоритмические процессы. И не умеют описывать эти процессы в алгоритмической форме.
5. Причина в том, что существующие методы алгоритмизации слишком сложны. Они рассчитаны на математиков и программистов и недоступны для работников управления. При нынешнем состоянии дел управленцы лишены возможности грамотно использовать алгоритмы в своей практической работе.
6. Учебная и деловая литература по вопросам управления фактически не содержит развернутого описания алгоритмов управления. (А если и содержит, то понять эти алгоритмы за приемлемое время невозможно).
7. В связи с отсутствием необходимых руководств практическое знакомство с реальными алгоритмами управления осуществляется не во время учебы, а непосредственно на рабочем месте работников управления и продолжается много лет.
8. Язык ДРАКОН позволяет частично устранить этот недостаток. Представление алгоритмов управления в виде наглядных дракон-схем облегчает для работников управления решение многих управленческих задач.

Часть IV

**МАТЕМАТИЧЕСКИЕ
АЛГОРИТМЫ
(ПРИМЕРЫ)**

ПРОСТЫЕ МАТЕМАТИЧЕСКИЕ АЛГОРИТМЫ

§1. ВВЕДЕНИЕ

В предыдущей части (главы 17–25) мы рассмотрели примеры алгоритмов, взятых из практической жизни.

В этой части (главы 26–29) перейдем к математическим задачам.

Рассмотрев оба класса алгоритмов, можно убедиться, что язык ДРАКОН позволяет наглядно изображать алгоритмы практически в любых ситуациях.

§2. АЛГОРИТМ ЕВКЛИДА

Наибольшим общим делителем НОД двух целых чисел A и B называется их общий делитель d , который делится на любой другой общий делитель чисел A и B .

Например, для чисел 12 и 18 наибольший общий делитель равен 6. Он делится на все общие делители этих чисел: 1, 2, 3, 6.

На рис. 207 показан алгоритм вычисления НОД (алгоритм Евклида) двух натуральных чисел методом вычитания.

Алгоритм Евклида методом вычитания можно сформулировать так. Пусть даны натуральные числа A и B . Надо по очереди вычитать из большего числа меньшее. Наступит момент, когда вычитаемое станет равно разности. В результате получится НОД (рис. 207).

§3. СУММА ЦИФР ЦЕЛОГО ЧИСЛА

Задача. Дано целое число N . Найти сумму цифр этого числа.

Сумму цифр числа N обозначим через S . В начале алгоритма положим $S = 0$. Переменная $N_{ост}$ обозначает остаток от деления целого числа N на 10.

Алгоритм вычисления суммы цифр S показан на рис. 208.

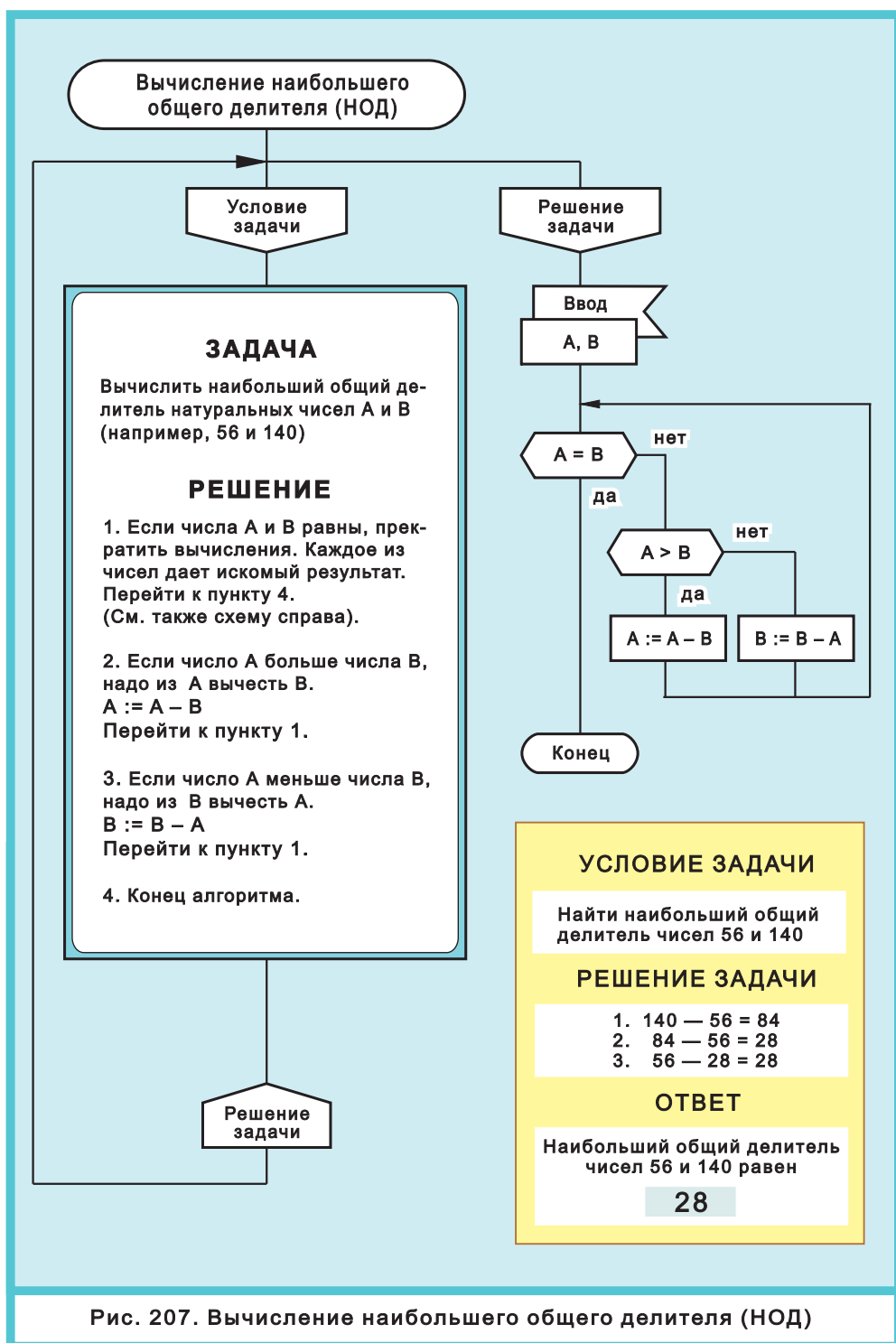


Рис. 207. Вычисление наибольшего общего делителя (НОД)

Пример. Трехзначное число N содержит три цифры XYZ .

$$X = N/100 - N_{\text{осм}1}/100$$

$$Y = N_{\text{осм}1}/10 - N_{\text{осм}2}/10$$

$$Z = N_{\text{осм}2}$$

Число N можно представить так $N = 100X + 10Y + Z$

При проверке алгоритма для $N = 759$ получим результат 21. Если $N = 2651$, сумма цифр равна 14 (рис. 208).

§4. ВЫЧИСЛИТЬ ДВА ЧИСЛА

Задача. Даны действительные числа M и N . Меньшее из этих чисел заменить их полусуммой. А большее – удвоенным произведением чисел M и N .

При сравнении двух чисел ($M > N$) и ($M < N$) надо учесть два случая:

- Если первое число (M) больше второго (N), найти произведение $2M \times N$.
- Если первое число (M) меньше второго (N), найти половину суммы M и N .

Следует помнить, что алгоритм (рис. 209) должен реализовать оба возможных варианта:

$$\begin{aligned} M &> N; \\ M &< N. \end{aligned}$$

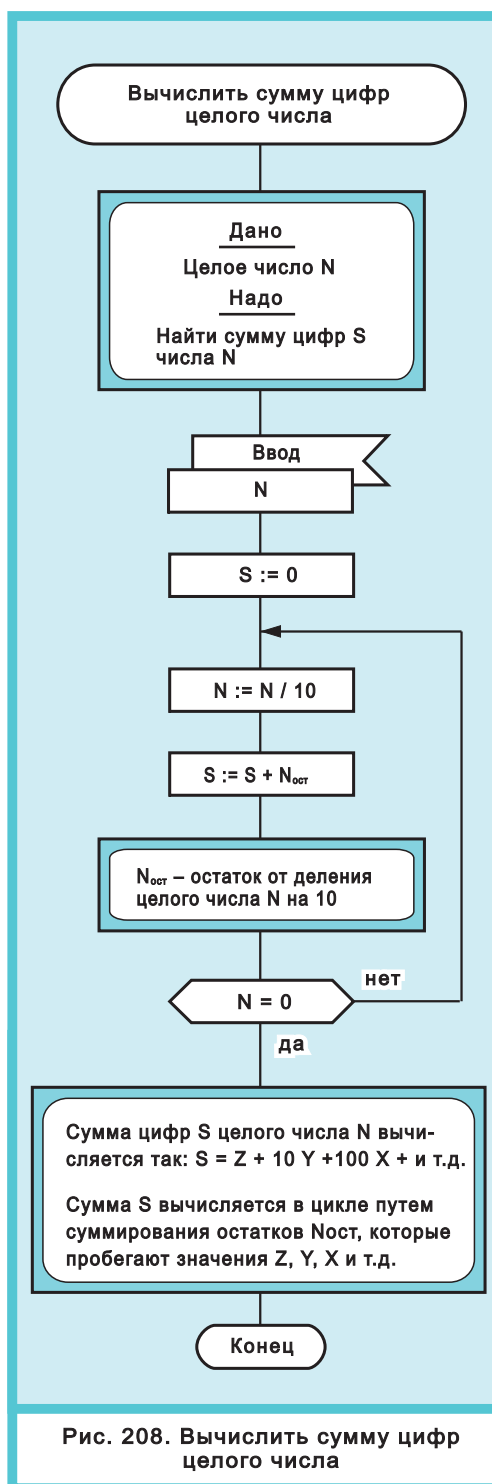
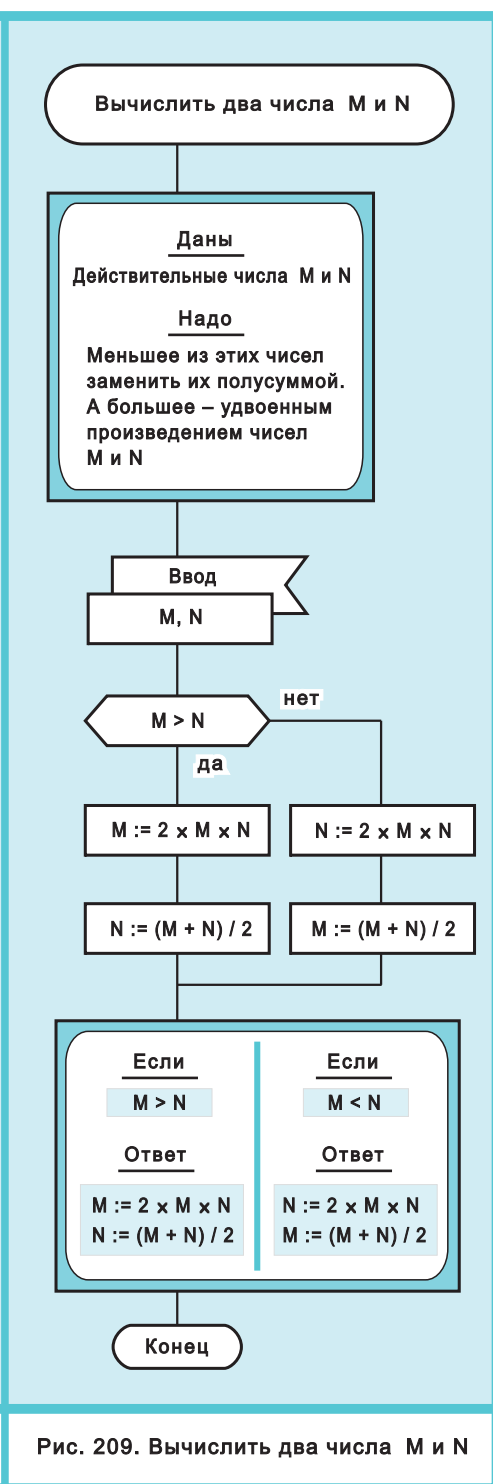
Пример. Если $M = 5$, $N = 6$, то в результате получим $M = 5,5$, $N = 60$ (рис. 209).

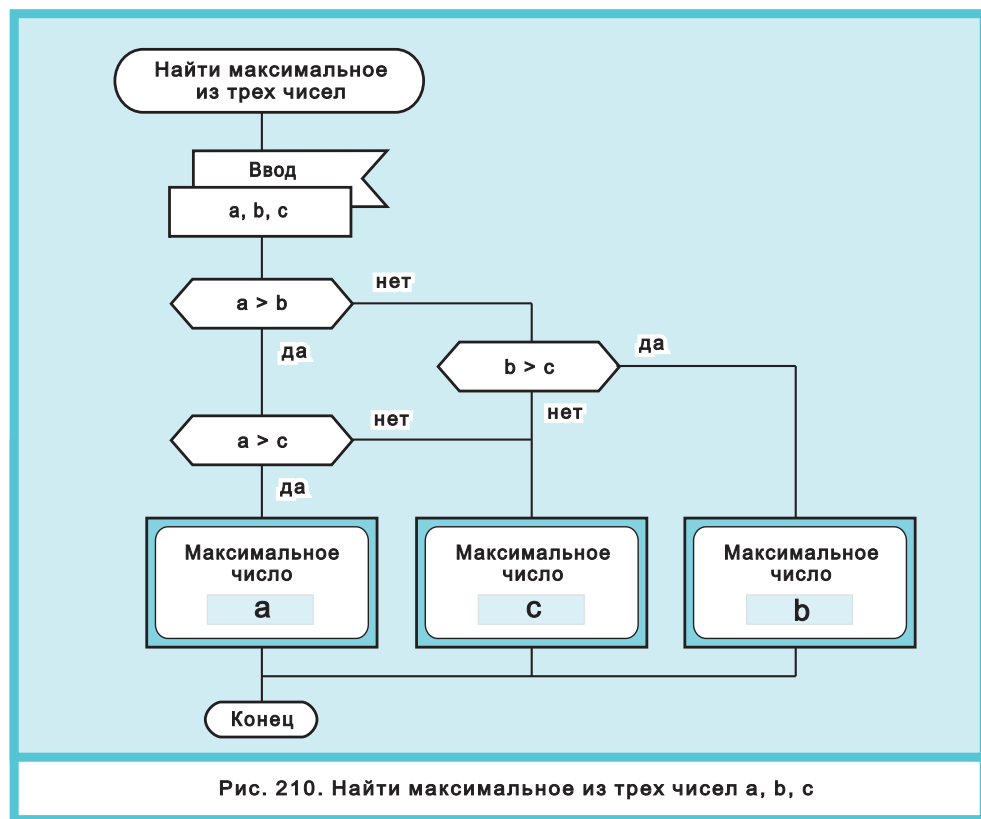
§5. НАЙТИ МАКСИМАЛЬНОЕ ИЗ ТРЕХ ЧИСЕЛ

Задача. Даны три действительных числа a , b , c . Найти максимальное из трех чисел.

Рассмотрим алгоритм на рис. 210.

- Если $a > b$ и $a > c$, максимальным числом будет a .
- Если $c > a > b$, максимальным числом будет c . Тот же результат получим, если $c > b > a$.
- Если $b > a$ и $b > c$, максимальным числом будет b (рис. 210).

Рис. 208. Вычислить сумму цифр
целого числаРис. 209. Вычислить два числа M и N



Пример. Найти максимальное из трех чисел: 14, -67, 45.
Ответ. 45

§6. ВЫВОДЫ

В данной главе показано использование языка ДРАКОН при построении следующих алгоритмов:

1. Вычислить наибольший общий делитель двух натуральных чисел (алгоритм Евклида).
2. Найти сумму цифр целого числа.
3. *Задача.* Меньшее из действительных чисел M и N заменить их полусуммой. А большее – удвоенным произведением чисел M и N .
4. Найти максимальное из трех действительных чисел.

АЛГОРИТМЫ С МАССИВАМИ

§1. ОБЪЕДИНЕНИЕ ДВУХ МАССИВОВ

Задача. Объединить два массива длины N в один чередованием элементов массива.

Даны два массива. Массив C имеет N элементов. Массив D также содержит N элементов.

$$C(N) = (c_1, c_2, \dots, c_N)$$

$$D(N) = (d_1, d_2, \dots, d_N)$$

При слиянии массивов C и D образуется массив E длиной $2N$ элементов.

$$E(N) = (e_1, e_2, \dots, e_{2N})$$

Соединим два массива в один по следующему правилу:

- Элементы нового массива E с нечетными номерами — это элементы из первого массива C .
- Элементы нового массива E с четными номерами — это элементы из второго массива D .

Алгоритм объединения массивов представлен на рис. 211. Схема расположения элементов всех трех массивов показана на рис. 213.

§2. ЭЛЕМЕНТЫ МАССИВА В ОБРАТНОМ ПОРЯДКЕ

Задача. Расположить элементы массива в обратном порядке.

Рассмотрим исходный массив A , в котором N элементов:

$$A(N) = (a_1, a_2, \dots, a_N)$$

Обратный массив, в котором элементы массива расположены в обратном порядке, обозначим через $A^*(N)$:

$$A^*(N) = (a_N, a_{N-1}, \dots, a_2, a_1)$$

Алгоритм, преобразующий исходный массив $A(N)$ в обратный массив $A^*(N)$, показан на рис. 212.

Пояснение к алгоритму

Предположим, что массив $A(N)$ содержит 10 элементов.

Шаг 1.

Выбираем переменную x и присваиваем ей значение a_{10} . ($x := a_{10}$).

Присваиваем числу a_{10} значение a_1 . ($a_{10} := a_1$).

Присваиваем числу a_1 значение x . ($a_1 := x$).

В результате этих действий мы поменяли местами элементы a_{10} и a_1 . То есть переместили десятый элемент массива a_{10} на место первого a_1 .

Шаг 2.

Присваиваем переменной x значение a_9 . ($x := a_9$).

Присваиваем числу a_9 значение a_2 . ($a_9 := a_2$).

Присваиваем числу a_2 значение x . ($a_2 := x$).

Мы поменяли местами элементы a_9 и a_2 . То есть переместили девятый элемент массива a_9 на место второго a_2 .

Шаг 3.

Присваиваем переменной x значение a_8 . ($x := a_8$).

Присваиваем числу a_8 значение a_3 . ($a_8 := a_3$).

Присваиваем числу a_3 значение x . ($a_3 := x$).

Мы поменяли местами элементы a_8 и a_3 . То есть переместили восьмой элемент массива a_8 на место третьего a_3 .

Шаг 4 и Шаг 5.

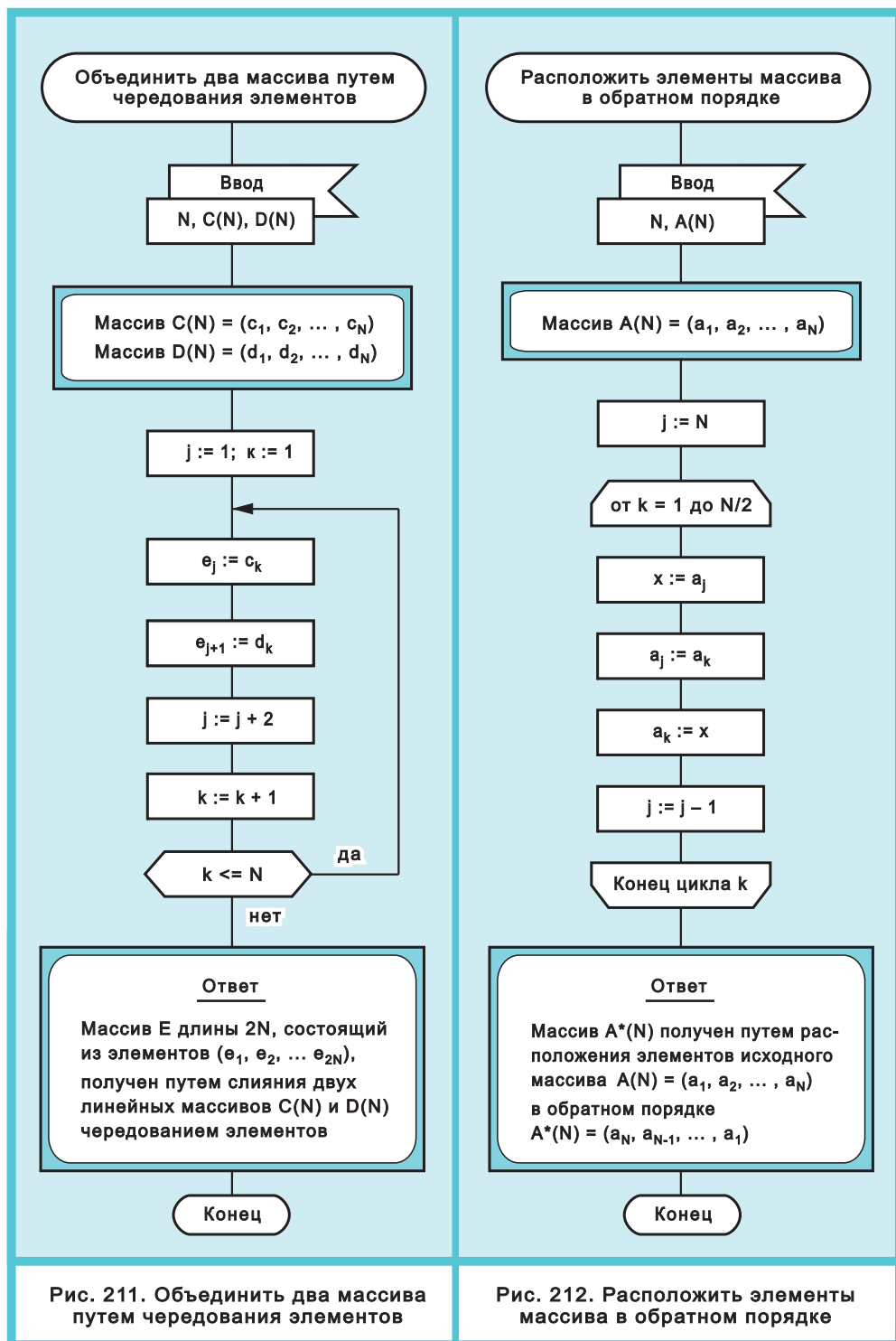
В последних двух шагах выполняем два действия:

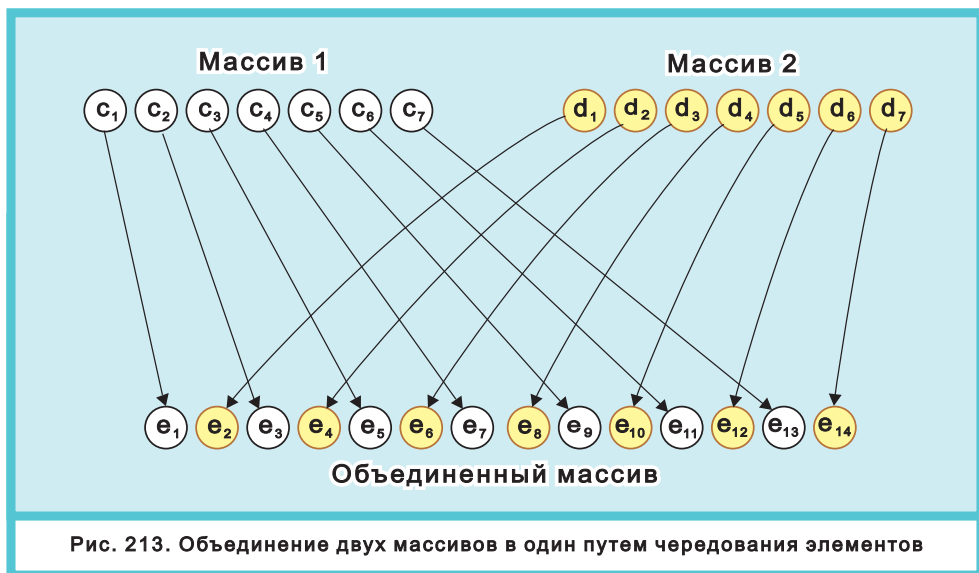
- меняем местами элементы a_7 и a_4 ;
- меняем местами элементы a_6 и a_5 .

Подведем итоги. Выполнив описанные шаги, мы превратили исходный массив $A(10)$ в обратный массив $A^*(10)$

$$A(10) = (a_1, a_2, \dots, a_{10})$$

$$A^*(10) = (a_{10}, a_9, \dots, a_1)$$





§3. ВЫВОДЫ

В данной главе показано использование языка ДРАКОН при построении следующих алгоритмов:

1. Объединить два массива длины N в один чередованием элементов массива.
2. Расположить элементы массива в обратном порядке.

АЛГОРИТМЫ ПОИСКА ДАННЫХ

§1. ПОИСК ЗАДАННОГО ЭЛЕМЕНТА

Задача. Задана последовательность $\{X_j\} = x_1, x_2, \dots, x_N$, где j изменяется от 1 до N . Длина последовательности N является фиксированной величиной. Все элементы имеют разные значения. Один из элементов может иметь значение P (эталон).

Выполнить поиск элемента, равного P , и определить его номер.

Результатом поиска может быть одно из двух. Либо поиск завершился успешно, и эталон P найден. Либо поиск оказался неудачным, и эталон P не найден.

Упрощенный алгоритм поиска выглядит так.

1. Сравнить очередной элемент с эталоном P .
2. Перейти к следующему элементу.
3. Если не все элементы просмотрены, повторить, начиная с пункта 1.

Условия окончания поиска таковы:

- Элемент найден, то есть $P = x_j$;
- Элемент не найден, хотя последовательность просмотрена от начала до конца.

Алгоритм поиска заданного элемента показан на рис. 214.

§2. ПОИСК МАКСИМАЛЬНОГО ЭЛЕМЕНТА

Задача. Задана последовательность $\{X_j\} = x_1, x_2, \dots, x_N$, где j изменяется от 1 до N . Длина последовательности N является фиксированной величиной. Все элементы имеют разные значения.

Найти максимальный элемент в последовательности и определить его номер.

Обозначим максимальный элемент через max , а его номер через N_{max} , где max – это переменная величина, которую будем называть эталоном.

Для начала присвоим переменной (эталону) значение первого элемента. Поиск производится путем последовательного сравнения всех элементов с эталоном max .

Сравним с эталоном второй элемент. Если он меньше эталона, надо изменить эталон. Для этого присвоим эталону значение второго элемента. И продолжим сравнение – теперь с третьим элементом. Таким образом можно сравнить все элементы последовательности с «эталоном».

План алгоритма состоит в следующем.

1. Сравнить эталон с очередным элементом последовательности.
2. Перейти к следующему элементу.
3. Если не все элементы последовательности просмотрены, повторить действия, начиная с пункта 1.

Начнем поиск с первого элемента. И примем его за эталонное значение $max = x_1$. Переменной N_{max} присвоим начальное значение, равное 1. Затем сравним с эталоном max второй элемент. И так далее.

Алгоритм поиска максимального элемента показан на рис. 215.

§3. ПОИСК ДЕЛЕНИЕМ ПОПОЛАМ (ДВОИЧНЫЙ ПОИСК)

Упрощенная задача. Задана упорядоченная последовательность целых чисел: 1, 2, 3, 4, 5, 6, 7, 8, 9. Надо найти число 4.

1. Выделяем средний элемент последовательности – число 5.
2. Выясняем, что 5 больше, чем 4.
3. Делим последовательность на две части:
 - 1, 2, 3, 4, 5,
 - 6, 7, 8, 9.
4. Искомое число 4 находится в первой части (1, 2, 3, 4, 5). Поэтому вторую часть (6, 7, 8, 9) отбрасываем.
5. В первой части (1, 2, 3, 4, 5) выделяем средний элемент – число 3.
6. Выясняем, что 3 меньше 4.
7. Делим последовательность на две части:
 - 1, 2, 3,
 - 4, 5.
8. Искомое число 4 находится во второй части (4, 5). Поэтому первую часть (1, 2, 3) отбрасываем.
9. Вторую часть (4, 5) делим пополам и находим искомое число 4.

В общем виде двоичный поиск состоит в следующем:

- Множество элементов делим пополам.
- Определяем, в какой из половин находится искомый элемент.

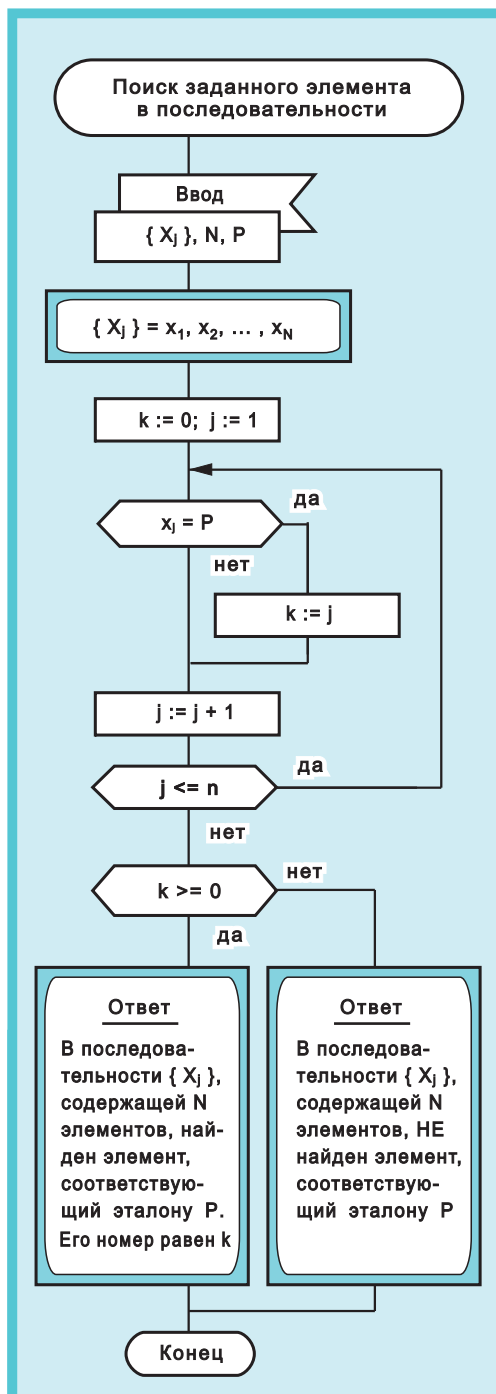


Рис. 214. Поиск заданного элемента в последовательности

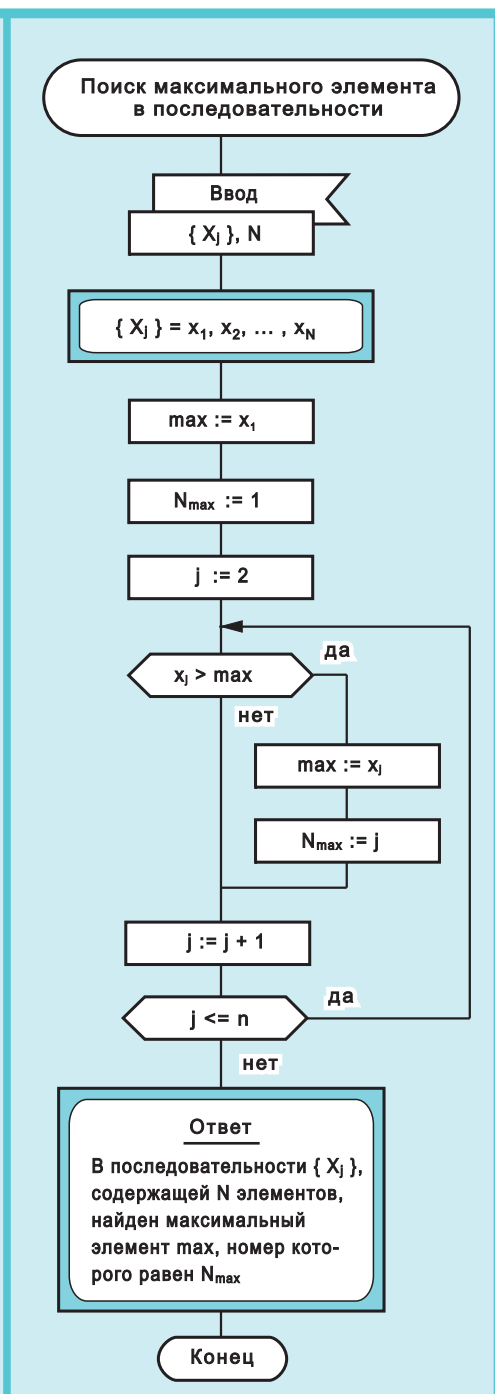


Рис. 215. Поиск максимального элемента в последовательности

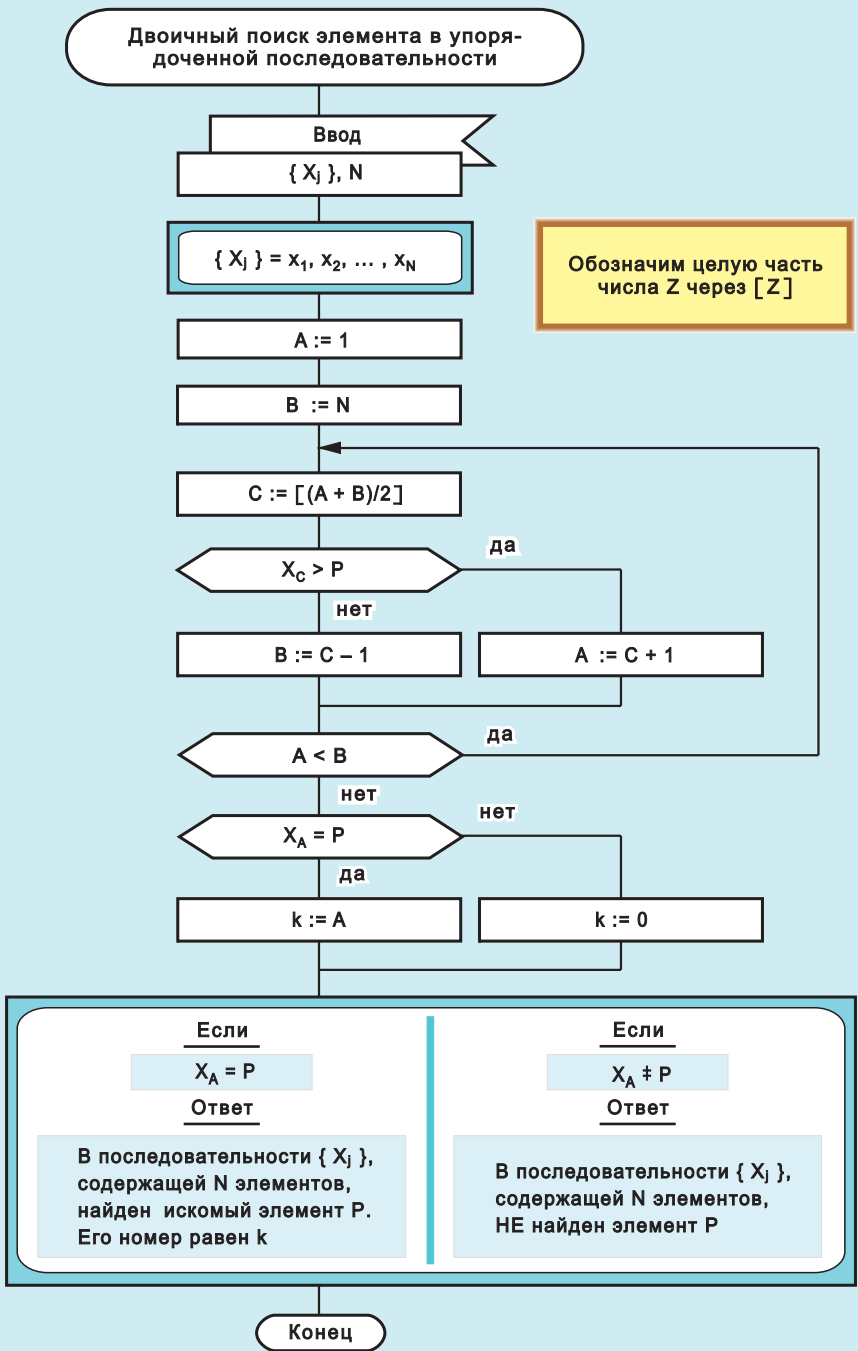


Рис. 216. Двоичный поиск элемента в упорядоченной последовательности

- Выбранную половину делим пополам и т. д.
- Процесс продолжается до тех пор, пока очередная половина не станет равной единственному элементу, которое будет искомым.
- В противном случае будет установлено, что последняя половина не содержит искомого элемента.

Алгоритм двоичного поиска (поиск делением пополам) – это алгоритм, в котором последовательно уменьшается интервал исследуемых данных в два раза.

Задача. Задана упорядоченная последовательность элементов. Порядковые номера элементов называются индексами. Наименьший индекс в последовательности равен A , наибольший – B . Если разделить интервал $[A, B]$ пополам, получится некоторый индекс C . Обозначим искомым элемент через переменную P .

Если $X_C > P$, то элемент P следует искать в интервале $[A, B]$, где $B := C - 1$. Если $X_C < P$, то элемент P надо искать в интервале $[A, B]$, где $A := C + 1$. Следует повторять эти действия до тех пор, пока A и B не совпадут.

Проверить условие $X_A = P$ и выбрать один из ответов:

1. Последовательность содержит искомым элемент P .
2. Последовательность не содержит элемент P .

План алгоритма имеет вид:

1. Вычислить C .
2. Сравнить X_C с переменной P .
3. Определить A и B .
4. Если A и B не совпадают, повторить с пункта 1.

Алгоритм двоичного поиска показан на рис. 216.

§4. ВЫВОДЫ

В данной главе показано использование языка ДРАКОН при построении следующих алгоритмов:

1. Выполнить поиск заданного элемента в последовательности и определить его номер.
2. Найти максимальный элемент в последовательности и определить его номер.
3. Найти заданный элемент в последовательности делением пополам (двоичный поиск).

РЕКУРСИВНЫЕ АЛГОРИТМЫ

§1. РЕКУРСИЯ

Рекурсия – это обращение функции к самой себе.

Мы рассмотрим классический пример рекурсии – вычисление факториала. Этот пример является классическим только из-за удобства для объяснения понятия рекурсии.

Однако следует помнить, что рекурсивное вычисление факториала не дает выигрыша по сравнению с обычным способом вычисления факториала с помощью циклов (см. рис. 90, 91).

§2. ОПРЕДЕЛЕНИЕ ФАКТОРИАЛА

Факториал числа n (обозначается $n!$, произносится *эн факториал*) – произведение всех натуральных чисел до n включительно:

$$n! = 1 * 2 * \dots * n$$

По определению полагают $0! = 1$. Факториал определён только для целых неотрицательных чисел.

§3. РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ ФАКТОРИАЛА

В §2 дано стандартное определение факториала – перемножением чисел от 1 до n в цикле.

Теперь дадим рекурсивное определение факториала:

$$n! = (n - 1)! * n, \quad \text{если } n > 1 \quad (1)$$

$$n! = 1, \quad \text{если } n \leq 1 \quad (2)$$

Формула (1) показывает, что мы определили факториал $n!$ через сам факториал $(n - 1)!$ Это и есть рекурсивное определение.

Формула (2) распадается на две части:

$$\begin{aligned} 0! &= 1, \\ 1! &= 1. \end{aligned}$$

§4. ГРАНИЧНОЕ УСЛОВИЕ

В рекурсивном определении должно присутствовать *граничное условие*, при котором рекурсия прекращается.

Граничным условием в данном случае является $n \leq 1$.

Пример. Вычислим значение $5!$, несколько раз применив правило (1).

$$5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1! = 5 * 4 * 3 * 2 * 1$$

Легко заметить, что при данном вычислении рекурсия обрывается в момент, когда $1! = 1$. Это и есть граничное условие $n \leq 1$ (см. формулу 2).

§5. РЕКУРСИВНЫЙ АЛГОРИТМ ВЫЧИСЛЕНИЯ ФАКТОРИАЛА

Напишем рекурсивную функцию факториала *fact*:

$$fact := n! = n * (n - 1)!$$

Соответствующий алгоритм показан на рис. 217. Рекурсивный вызов $fact := (n - 1)!$ записывается в иконе «вставка».

§6. КАК ИЗОБРАЗИТЬ РЕКУРСИЮ НА ЯЗЫКЕ ДРАКОН?

На дракон-схемах рекурсию можно отображать традиционным способом. Это значит, что используется стандартный графический алфавит языка ДРАКОН. Иконы, предназначенные специально для изображения рекурсии, не предусмотрены.

Вместе с тем на дракон-схеме при желании можно сделать графическую подсказку, указывающую на наличие рекурсии.

Вопрос. Как выбрать графические средства для отображения рекурсии?

Ответ. Рекомендуется использовать две иконы комментариев. Первая икона устанавливается до начала рекурсии. В иконе пишут «Начало рекурсии». Вторая икона устанавливается после конца рекурсии. В ней пишут «Конец рекурсии». Она помещается перед иконой «конец».

Описанный способ показан на рис. 217.

Сказанное подразумевает, что необходимо ввести

Правило. Разрешается помещать икону комментариев с текстом «Начало рекурсии» выше иконы «заголовок», соединив указанные иконы вертикальной линией.

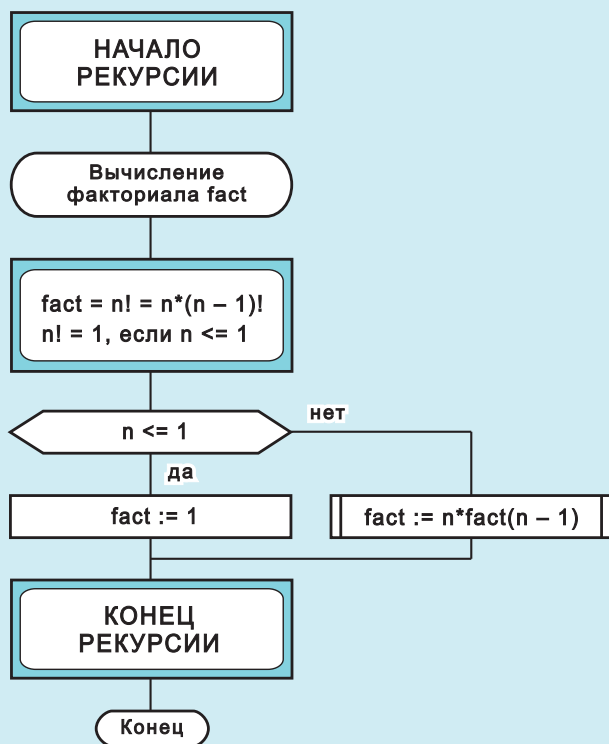


Рис. 217. Рекурсивная функция «Вычисление факториала fact»

§7. ВЫВОДЫ

1. Рекомендуется использовать специальный графический признак, позволяющий легко обнаружить, что на дракон-схеме изображен рекурсивный алгоритм.
2. В качестве графического признака рекурсии предлагается использовать две иконы «комментарий».
3. Первая икона «комментарий» устанавливается до начала рекурсии. В ней пишут «Начало рекурсии».
4. Вторая икона «комментарий» располагается после конца рекурсии. В ней пишут «Конец рекурсии». Она помещается перед иконой «конец».

Часть V

**ЗАКЛЮЧИТЕЛЬНЫЕ
РЕКОМЕНДАЦИИ
ПО СОЗДАНИЮ
ДРАКОН-СХЕМ**

РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ АЛГОРИТМИЧЕСКИХ СТРУКТУР «СИЛУЭТ» И «ПРИМИТИВ»

§1. ЧТО ЛУЧШЕ: СИЛУЭТ ИЛИ ПРИМИТИВ?

Предположим, вы собираетесь начертить дракон-схему. Какую структуру лучше выбрать: силуэт или примитив?

Этот вопрос в упрощенном виде обсуждался в главе 6. В этой главе мы дадим более подробные и окончательные рекомендации.

§2. ГОЛОВНОЙ АЛГОРИТМ И ДВА МЕТОДА СОЗДАНИЯ АЛГОРИТМОВ

Головной алгоритм – это алгоритм самого верхнего уровня на лестнице декомпозиции. Головной алгоритм может содержать вставки (вызываемые процедуры). Но сам он не может быть вставкой для алгоритма более высокого уровня.

Головной алгоритм – это всегда силуэт. Он не может быть примитивом.

Для создания головного алгоритма используют два метода, которые могут дополнять друг друга:

- метод эргономичной декомпозиции;
- метод многостраничного силуэта.

Метод эргономичной декомпозиции описан в главе 21. Метод многостраничного силуэта изложен ниже.

§3. КАК НАРИСОВАТЬ БОЛЬШОЙ АЛГОРИТМ

Полезно различать три понятия:

- большие алгоритмы;
- средние алгоритмы;
- малые алгоритмы.

1. Чтобы создать большой алгоритм, следует использовать систему силуэтов.
2. Головной алгоритм большой схемы обязательно должен быть силуэтом.
3. Алгоритм надо рассматривать как последовательную декомпозицию силуэтов. В том смысле, что каждый силуэт может содержать много вставок, каждая из которых раскрывается как силуэт.
4. **Ни в коем случае нельзя представлять головной алгоритм 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.
5. Если кто-либо (по ошибке) начнет рисовать головной алгоритм в виде системы примитивов, надо немедленно прекратить работу. И превратить примитивы в силуэт. При этом каждый примитив превращается в ветку силуэта. Иногда часть примитивов превращается в ветки силуэта более низкого уровня на лестнице декомпозиции.
6. *Пример.* На рис. 183–185 показано, что любой, сколь угодно большой и сложный алгоритм можно изобразить в виде системы силуэтов. Головной алгоритм представлен на рис. 183. Силуэты более низкого уровня формируются из головного алгоритма с помощью вставок. При этом используется простой и единообразный прием – *эргономичная декомпозиция*.

Вставки нижних уровней изображаются в виде примитива только как исключение.

§4. ОДНОСТРАНИЧНЫЙ И МНОГОСТРАНИЧНЫЙ СИЛУЭТ

Головной алгоритм большого или среднего размера может быть:

- одностраничным силуэтом;
- многостраничным силуэтом.

Одностраничный силуэт размещается на одном листе бумаги согласно стандарту ЕСКД (например, формата А1, А4×4, А3 и т. д.). Примеры показаны на рис. 183–185.

Многостраничный силуэт размещают на нескольких листах бумаги указанного формата.

Рассмотрим пример. На рис. 218–221 изображен силуэт, показанный в виде комплекта из четырех листов формата А3.

Мысленно расположите на столе листы по горизонтали слева направо. Пронумеруйте листы 1, 2, 3, 4. Все четыре листа связаны двумя горизонтальными шинами через соединители. В данном примере силуэт из четырех листов содержит 19 веток (рис. 218–221).

Число листов в дракон-схеме формально не ограничено. Но практика показывает, что многостраничный силуэт редко содержит более 32 веток.

§5. КАК НАРИСОВАТЬ СРЕДНИЙ АЛГОРИТМ

Средний и большой алгоритмы различаются только размерами. Методика их создания одна и та же. Она описана в §§3 и 4.

§6. КАК НАРИСОВАТЬ МАЛЫЙ АЛГОРИТМ

Алгоритмы малых размеров можно рисовать в виде примитива. Примеры показаны на рис. 32, 56, 79, 80, 90, 91, 128, 168, 169, 186, 201, 202, 205.

Сравнивая силуэт и примитив, можно отметить следующее.

1. Силуэт – главное достоинство языка ДРАКОН. Он обладает мощными выразительными средствами. В реальной работе силуэт используется в подавляющем большинстве случаев.
2. Примитив применяют крайне редко, скорее как исключение.
3. Тем не менее, полностью отказываться от понятия «примитив» не следует по двум причинам.
4. 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.
5. Вторая причина – необходимость описания малых алгоритмов («мелких огрызков»). Откуда берутся «мелкие огрызки»? В процессе декомпозиции силуэта может случиться, 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. СОЕДИНИТЕЛЬ

Уже говорилось, что при создании многостраничного силуэта необходимы соединители.

Икона «соединитель» имеет вид кружка (рис. 17, икона И26). Она позволяет соединять горизонтальные шины (линии), находящиеся на соседних листах. Иначе говоря, соединитель служит для перехода линии с листа на лист.

Рассмотрим пример.

- На первом листе (рис. 218) есть два соединителя (см. справа 1 и 2).
 На втором листе (рис. 219) – четыре соединителя (слева 1, 2, справа 3, 4).
 На третьем листе (рис. 220) – четыре соединителя (слева 3, 4, справа 5, 6).
 На четвертом листе (рис. 221) – два соединителя (слева 5, 6).



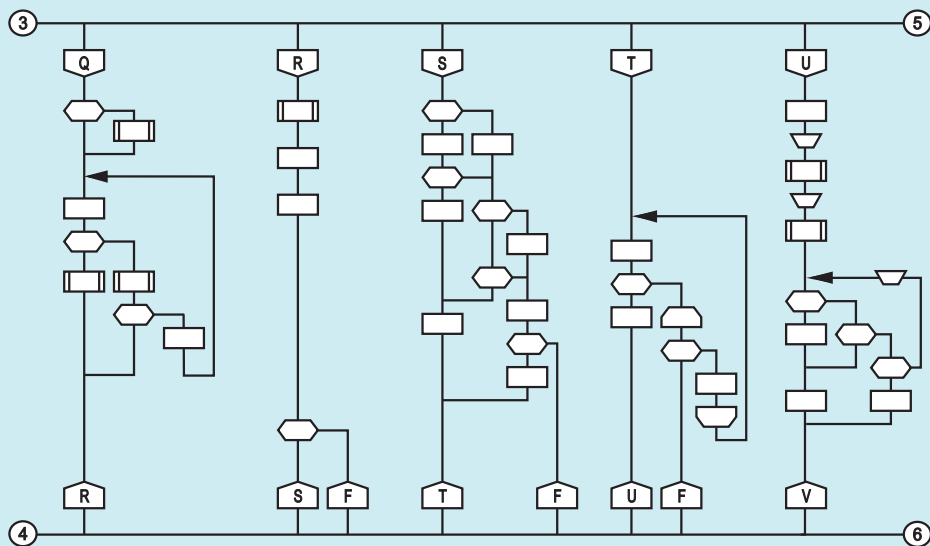
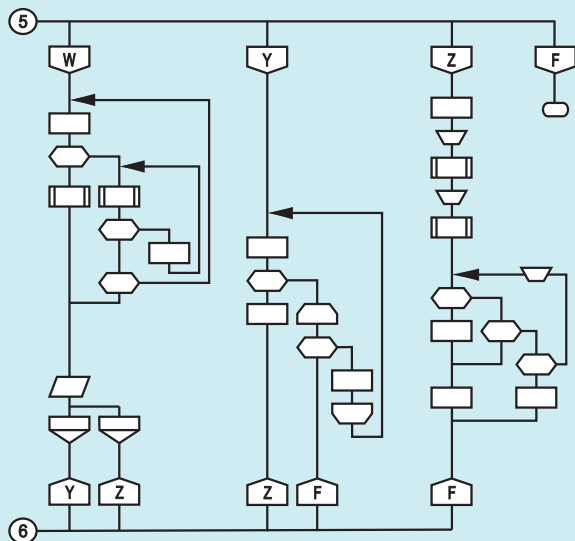


Рис. 220. Лист 3. (Комплект из 4-х листов формата А3)



ПОЯСНЕНИЕ

1. На четырех рисунках (рис. 218–221) показана дракон-схема силуэт. Схема занимает 4 листа формата А3.

2. Мысленно расположите листы по горизонтали слева направо. Пронумеруйте листы 1, 2, 3, 4.

3. Все 4 листа соединены соединителями. (см. рис. 17 икона И26).

4. Дракон-схема на 4-х листах содержит 19 веток

Рис. 221. Лист 4. (Комплект из 4-х листов формата А3)

§8. ВЫВОДЫ

1. Головной алгоритм – это алгоритм самого верхнего уровня на лестнице декомпозиции.
2. Головной алгоритм может содержать вставки (вызываемые процедуры). Но сам он не может быть вставкой для алгоритма более высокого уровня.
3. Головной алгоритм – это всегда силуэт. Он не может быть примитивом или системой примитивов.
4. Для создания головного алгоритма используют:
 - метод эргономичной декомпозиции;
 - метод многостраничного силуэта.
5. Головной алгоритм может быть:
 - одностраничным силуэтом;
 - многостраничным силуэтом.
6. Одностраничный силуэт размещается на одном листе бумаги (на одном экране).
7. Многостраничный силуэт размещают на нескольких листах бумаги. При работе с экраном силуэт прокручивают по горизонтали.
8. Многостраничный силуэт образует целостную зрительную сцену. Сквозь все листы многостраничного силуэта проходят две горизонтальные шины, которые скрепляют листы между собой с помощью пронумерованных соединителей.
9. Силуэт – главное достоинство языка ДРАКОН. Он обладает мощными выразительными средствами.
10. Сложные алгоритмы следует изображать как силуэты, в которых многократно используются иконы «вставка». Последние, в свою очередь, раскрываются как силуэты и т. д. Таким образом, сложный алгоритм надо изображать как последовательную декомпозицию силуэтов.
11. На практике силуэт используют в подавляющем большинстве случаев.
12. Примитив применяют редко, скорее как исключение.
13. Тем не менее, отказываться от примитива не следует, так как он нужен для описания малых алгоритмов.
14. Кроме того, примитив полезен из педагогических соображений. Основные понятия и правила ДРАКОНа удобно объяснять на самой простой модели. То есть на примитиве. И только после этого переходить к рассказу о силуэте.

КАК УЛУЧШИТЬ ПОНЯТНОСТЬ ВЕТОК?

§1. ВВЕДЕНИЕ

Наша цель – облегчить создание, чтение и понимание алгоритмов. На протяжении всей книги мы пытались сделать алгоритмы понятными, приятными для зрительного восприятия, эргономичными.

В этой главе мы сосредоточим внимание на силуэте. И попытаемся выяснить: как сделать силуэты более удобными для работы. Более легкими для понимания.

Проблема в том, что некоторые ветки силуэта могут оказаться громоздкими и сложными. Это нежелательно. Более того, недопустимо. Такой недостаток необходимо устранить.

Для этого существует специальный метод – *метод дробления веток*.

§2. ДРОБЛЕНИЕ ВЕТОК

Метод дробления веток состоит в том, что сложная ветка разбивается на две части.

При этом должно соблюдаться

Правило. При дроблении веток исходная ветка (подлежащая дроблению) должна быть эквивалентной двум новым веткам (появившимся в результате дробления).

Наша задача – показать, что метод дробления веток действительно позволяет улучшить понятность силуэта.

§3. КАК УПРОСТИТЬ СЛОЖНУЮ ВЕТКУ?

Итак, чтобы упростить ветку, надо ее «разорвать» на две половины. И превратить их в две новые ветки.

Эргономическая оптимизация силуэта есть графическое преобразование веток, при котором:

- алгоритм работы силуэта практически не меняется (в том смысле, что исходный и преобразованный силуэты эквивалентны);
- изменяется только форма силуэта (в силуэте увеличивается число веток);
- производится дробление веток с целью их упрощения;
- при необходимости дробление может повторяться несколько раз.

§4. РАЗРЫВ ВЕТКИ И ТОЧКА РАЗРЫВА

Точка разрыва – это точка, в которой соединительная линия разрывается. Данная точка позволяет расчленить ветку на части.

Точки разрыва бывают трех видов:

- простая точка разрыва;
- точка присоединения;
- точка дублирования.

Полезно различать три типа разрывов:

- простой разрыв ветки;
- разрыв ветки с присоединением;
- разрыв ветки с дублированием.

План дальнейшего изложения таков. В §§5–7 описаны простейшие приемы дробления. Затем (в §§8–12), опираясь на эти сведения, будет рассмотрен более сложный пример.

§5. ПРОСТОЙ РАЗРЫВ ВЕТКИ

Простой разрыв показан на рис. 222. Этот рисунок содержит четыре части.

- исходная ветка *A*;
- заготовки для новых веток *A* и *B*;
- построение новых веток *A* и *B*;
- новые ветки *A* и *B*.

Точка разрыва *B* делит исходную ветку *A* на две части:

- верхнюю;
- нижнюю.

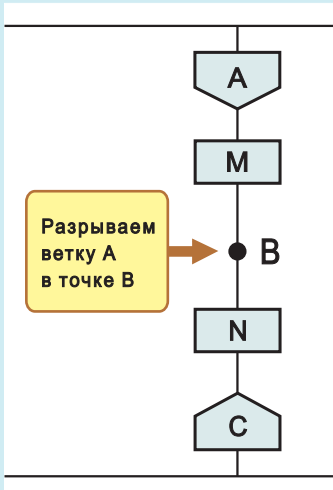
Верхняя часть служит заготовкой для новой ветки *A*. Нижняя – заготовка для новой ветки *B* (рис. 222, сверху).

Зачем нужна точка *B*? Что она делает?

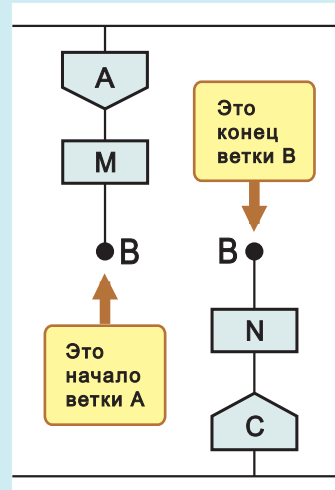
Ответ таков:

- в новой ветке *A* точка *B* порождает икону адрес *B*;
- в новой ветке *B* точка *B* порождает икону имя ветки *B*.

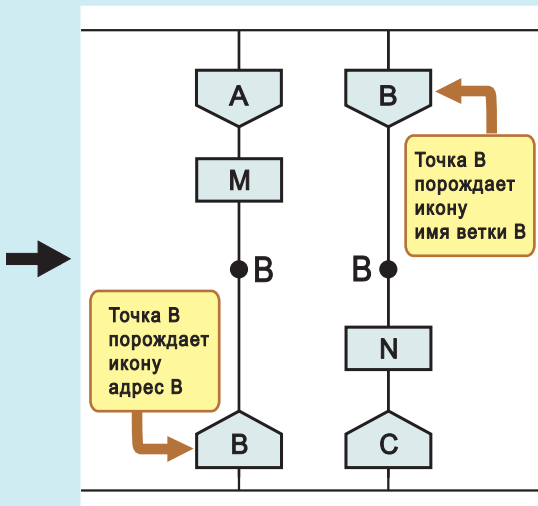
**ИСХОДНАЯ
ВЕТКА А**



**ЗАГОТОВКИ
ДЛЯ НОВЫХ ВЕТОК А и В**



**ПОСТРОЕНИЕ
НОВЫХ ВЕТОК А и В**



**НОВЫЕ
ВЕТКИ А и В**

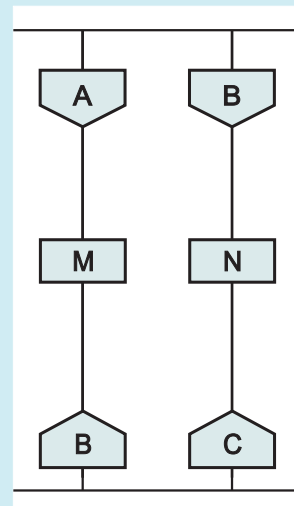


Рис. 222. Простой разрыв ветки.
Точка В разрывает ветку А на две новые ветки А и В

Заготовка отличается от ветки тем, что часть ветки удалена. На рис. 222 (внизу) показано, как заготовки постепенно превращаются в новые ветки.

Сравните исходную ветку *A* и новые ветки *A* и *B*. Легко видеть, что они эквивалентны.

§6. РАЗРЫВ ВЕТКИ С ПРИСОЕДИНЕНИЕМ

Простой разрыв ветки содержит только одну точку разрыва. Как и раньше, обозначим ее буквой *B*.

Перейдем к случаю, когда при делении ветки используется не одна, а две точки разрыва: *B* и *E*.

Вторая точка разрыва (*E*) называется *точкой присоединения*.

Вопрос. В чем состоит проблема присоединения?

Ответ. В исходной ветке *A* имеется одна икона адрес *E*. Но она должна попасть в обе новые ветки. Таким образом, при делении исходной ветки *A* икона адрес *E* должна удвоиться (рис. 223).

Вопрос. Каким образом можно удвоить икону адрес *E*?

Ответ. На рис. 223 (вверху) икона адрес *E* включена в состав новой ветки *A*. Поэтому заготовка новой ветки *B* осталась “ни с чем”. Следовательно, к этой заготовке надо *присоединить* икону адрес *E* (рис. 223, снизу).

Теперь мы можем разъяснить понятие «точка присоединения».

Точка присоединения – это вторая точка разрыва, к которой (в новой ветке *B*) надо присоединить икону адрес *E*, чтобы продублировать эту икону. Дублирование необходимо, так как икона адрес *E* должна находиться в обеих новых ветках (рис. 223).

§7. РАЗРЫВ ВЕТКИ С ДУБЛИРОВАНИЕМ

В предыдущем параграфе рассмотрен случай, когда при дроблении необходимо дублировать иконы.

Следует различать:

- дублирование одной иконы (этот случай уже описан);
- дублирование нескольких икон.

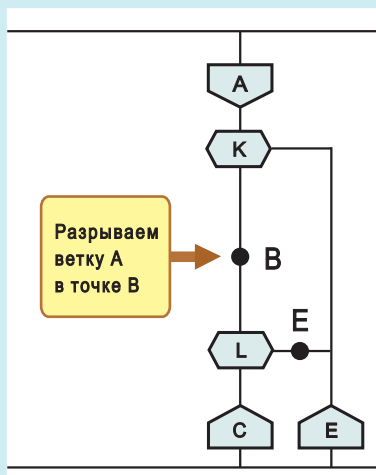
В последнем случае вводится понятие «точка дублирования».

Точка дублирования *X* – это вторая точка разрыва, к которой (в новой ветке *B*) надо присоединить не одну, а несколько икон, включая икону адрес *E*.

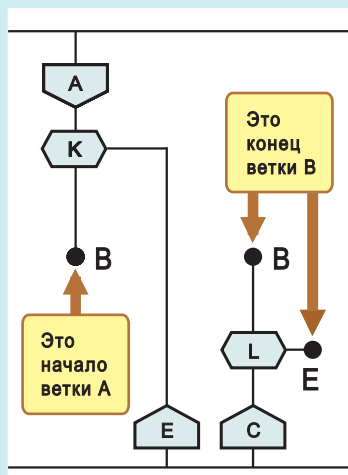
На рис. 224 показан случай, когда наряду с простой точкой разрыва *B*, используется точка дублирования *X*.

Точка разрыва *X* (точка дублирования) указывает на необходимость дублирования в новой ветке *B* икон *N* и *E*.

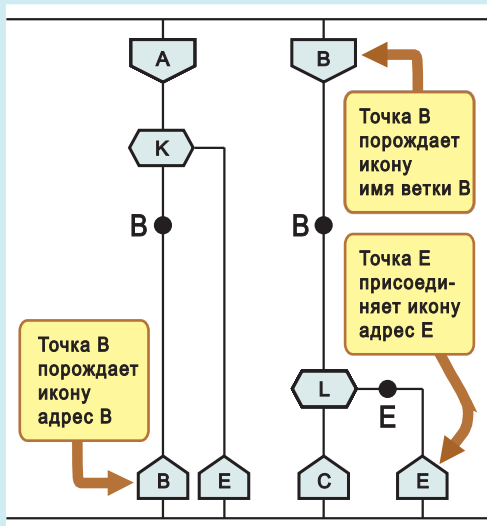
ИСХОДНАЯ ВЕТКА А



ЗАГОТОВКИ ДЛЯ НОВЫХ ВЕТОК А и В



ПОСТРОЕНИЕ НОВЫХ ВЕТОК А и В



НОВЫЕ ВЕТКИ А и В

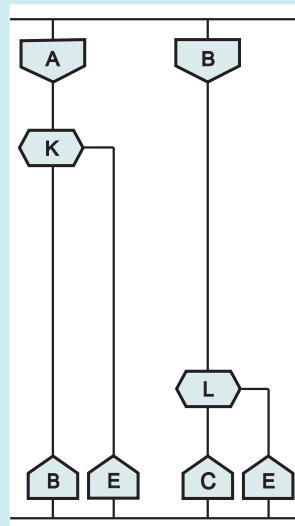
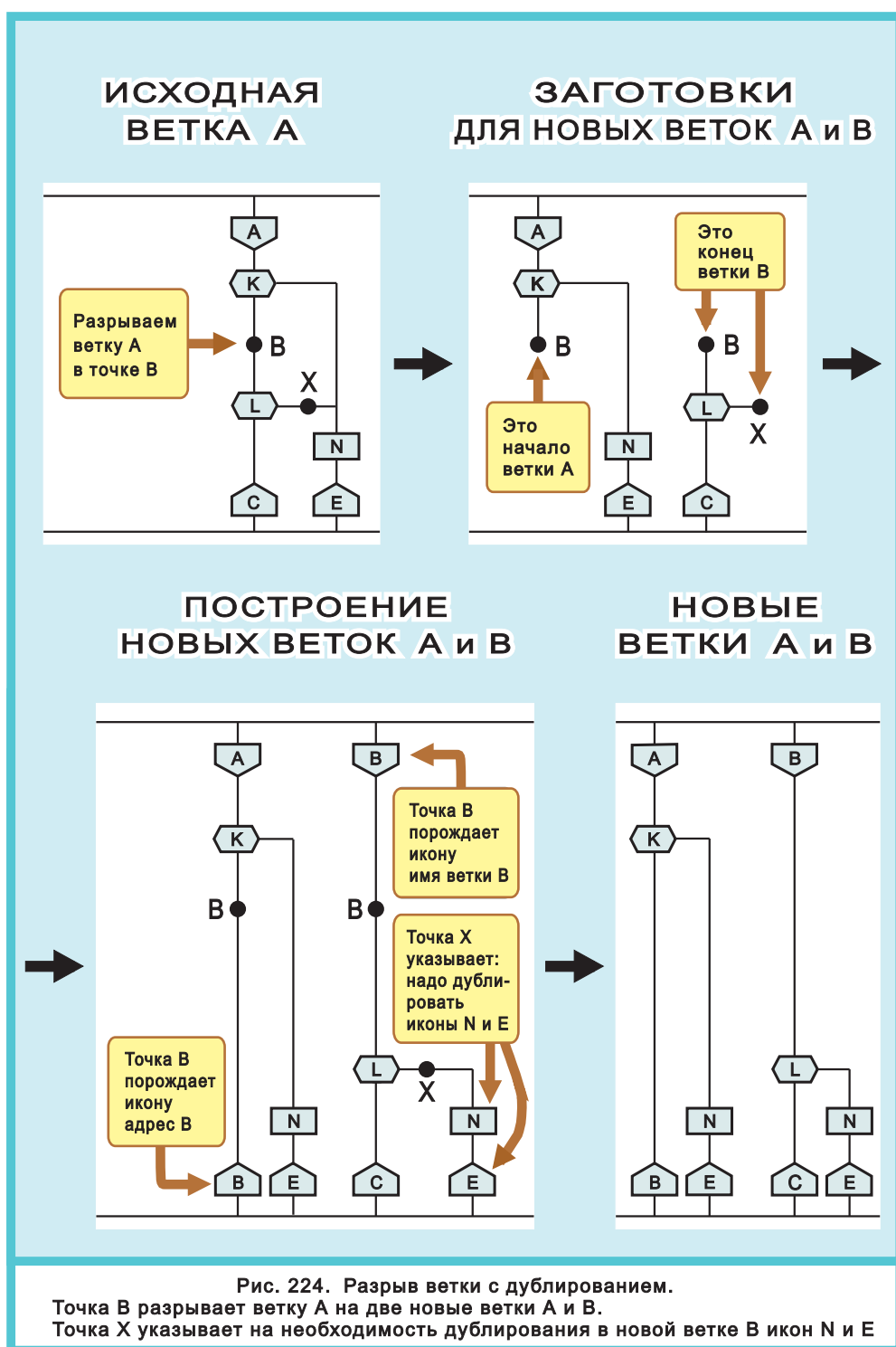


Рис. 223. Разрыв ветки с присоединением.
Точка В разрывает ветку А на две новые ветки А и В.
Точка Е присоединяет к новой ветке В икону адрес Е



Поясним. В исходной ветке *A* имеется одна икона *N* и одна икона *E*. Эти иконы должны попасть в обе новые ветки. Таким образом, при делении исходной ветки *A* иконы *N* и *E* должны удвоиться. Этот факт означает, что при дроблении ветки иконы *N* и *E* дублируются (рис. 224).

§8. ПРИМЕНЕНИЕ МЕТОДА ДРОБЛЕНИЯ К СЛОЖНОМУ СЛУЧАЮ

Рассмотрим пример. *Исходная схема* – трехветочная схема на рис. 225. В этой схеме две сложные ветки (*A* и *C*), которые нуждаются в дроблении.

Третья ветка (*E*) – простая. Она состоит всего из двух икон и не требует дробления.

Результирующая схема – пятиветочная схема на рис. 229, полученная путем преобразования исходной схемы.

Таким образом, в результате дробления трехветочный силуэт на рис. 225 превращается в эквивалентный ему пятиветочный на рис. 229.

В последующих параграфах описана пошаговая технология преобразования исходной схемы в результирующую.

§9. РАЗМЕТКА ВЕТОК ДЛЯ ДРОБЛЕНИЯ

Зачем нужна разметка? Чтобы выявить точки, в которых соединительные линии разрываются. Можно сказать и по-другому. Разметка нужна, чтобы указать точки разрыва.

На рис. 226 показана разметка веток для дробления. Точки разрыва обозначены жирными точками *B*, *E*, *D*, *X*.

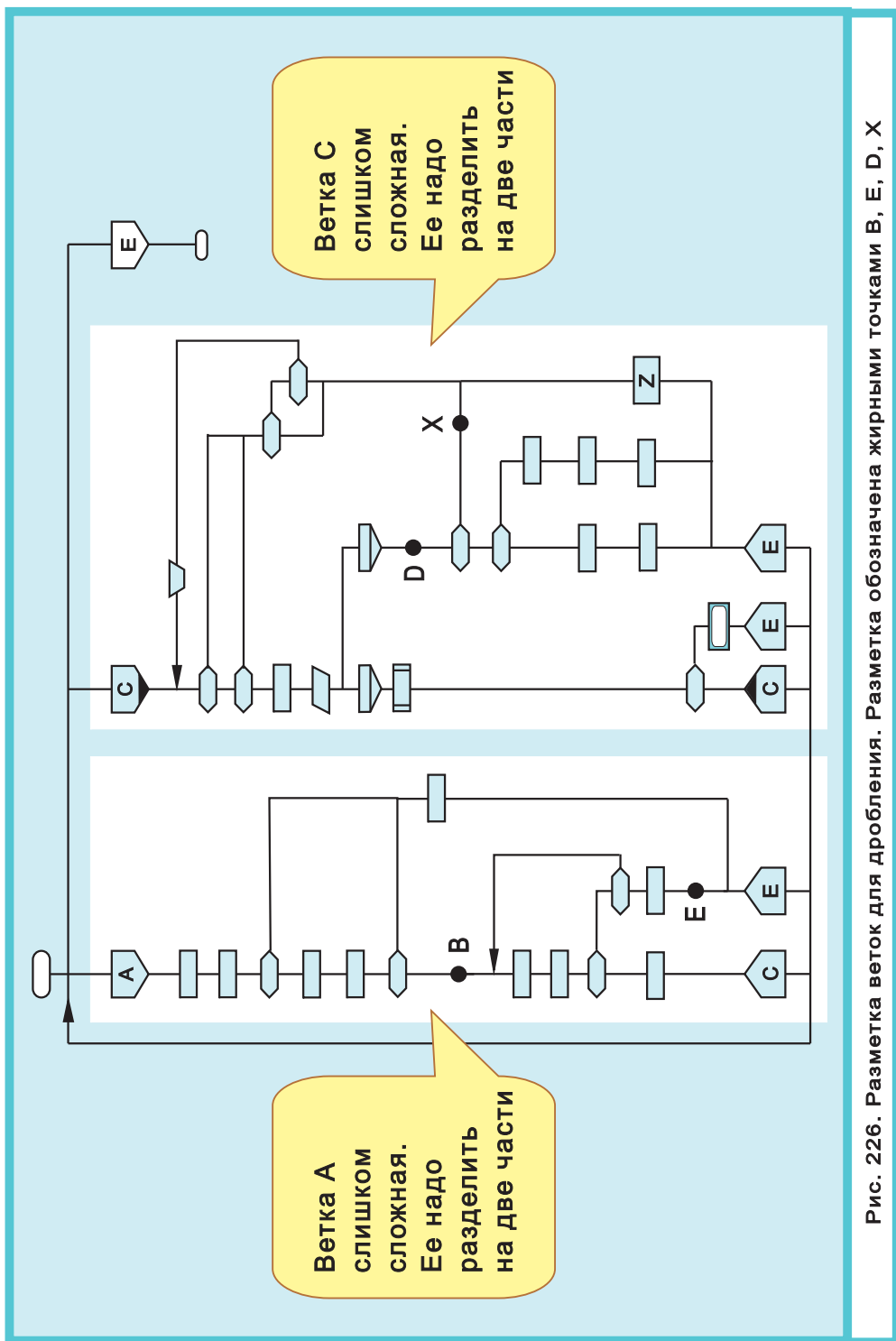
Разметка выполняется за четыре шага.

Шаг 1. Ветку *A* делим на две ветки. Точку разрыва (границу между верхней и нижней частями ветки *A*) обозначим буквой *B*.

Шаг 2. Мысленно выделим нижнюю часть ветки *A*. Точку присоединения (к нижней шине через икону *E*) обозначим буквой *E*.

Шаг 3. Ветку *C* делим на две ветки. Точку разрыва (границу между верхней и нижней частями ветки *C*) обозначим буквой *D*.

Шаг 4. Мысленно выделим нижнюю часть ветки *C*. Точку дублирования (которая указывает на необходимость дублировать иконы *Z* и *E* и присоединение последней к нижней шине) обозначим буквой *X*.



§10. ДЕЛЕНИЕ СЛОЖНОЙ ВЕТКИ А НА ДВЕ ВЕТКИ

На рис. 227 показано, как сложная ветка *A* превратилась в две простые ветки *A* и *B*. Эта операция выполняется за два шага.

Шаг 5. Точка разрыва *B* выполняет две функции:

- В новой ветке *A* она порождает новую икону адрес *B*.
- В новой ветке *B* она порождает новую икону имя ветки *B*.

Шаг 6. Точка присоединения *E* выполняет две функции:

- в нижней части исходной ветки *A* точка *E* обрывает соединительную линию (то есть является точкой разрыва), чтобы сохранить старую икону адрес *E* в составе новой ветки *A*;
- в новой ветке *B* порождает новую икону адрес *E* и присоединяет ее к нижней шине (рис. 227).

§11. ДЕЛЕНИЕ СЛОЖНОЙ ВЕТКИ С НА ДВЕ ВЕТКИ

На рис. 228 показано, как сложная ветка *C* превратилась в две простые ветки *C* и *D*. Эта операция выполняется за два шага.

Шаг 7. Точка разрыва *D* выполняет две функции:

- В новой ветке *C* она порождает новую икону адрес *D*.
- В новой ветке *D* она порождает новую икону имя ветки *D*.

Шаг 8. Точка дублирования *X* выполняет две функции:

- в нижней части старой ветки *C* точка *X* обрывает соединительную линию (то есть является точкой разрыва), чтобы сохранить старую икону действие *Z* и старую икону адрес *E* в составе новой ветки *C*;
- в новой ветке *D* дублирует иконы *Z* и *E* и присоединение последней к нижней шине (рис. 228).

§12. ПРОЦЕСС И РЕЗУЛЬТАТ ДРОБЛЕНИЯ ВЕТОК

Итак, мы рассмотрели сложный пример, представленный на рис. 225–229. Дадим общий обзор сделанного.

На рис. 225 показана *исходная схема*. В этой схеме есть недостаток. Силуэт содержит две неоправданно сложные ветки *A* и *C*. Эти ветки нуждаются в расчленении на более простые части (дроблении).

На рис. 226 указаны точки разрыва, намечающие границу, по которой рассекается исходная ветка, чтобы превратиться в две новых ветки.

На рис. 227 и 228 показано дробление веток *A* и *C*.

Сделаем пояснение для исходной ветки *C* (рис. 228). Заготовка новой ветки *C* почти полностью укомплектована всеми необходимыми частями (иконами). Не хватает лишь одной – иконы адрес *D*.

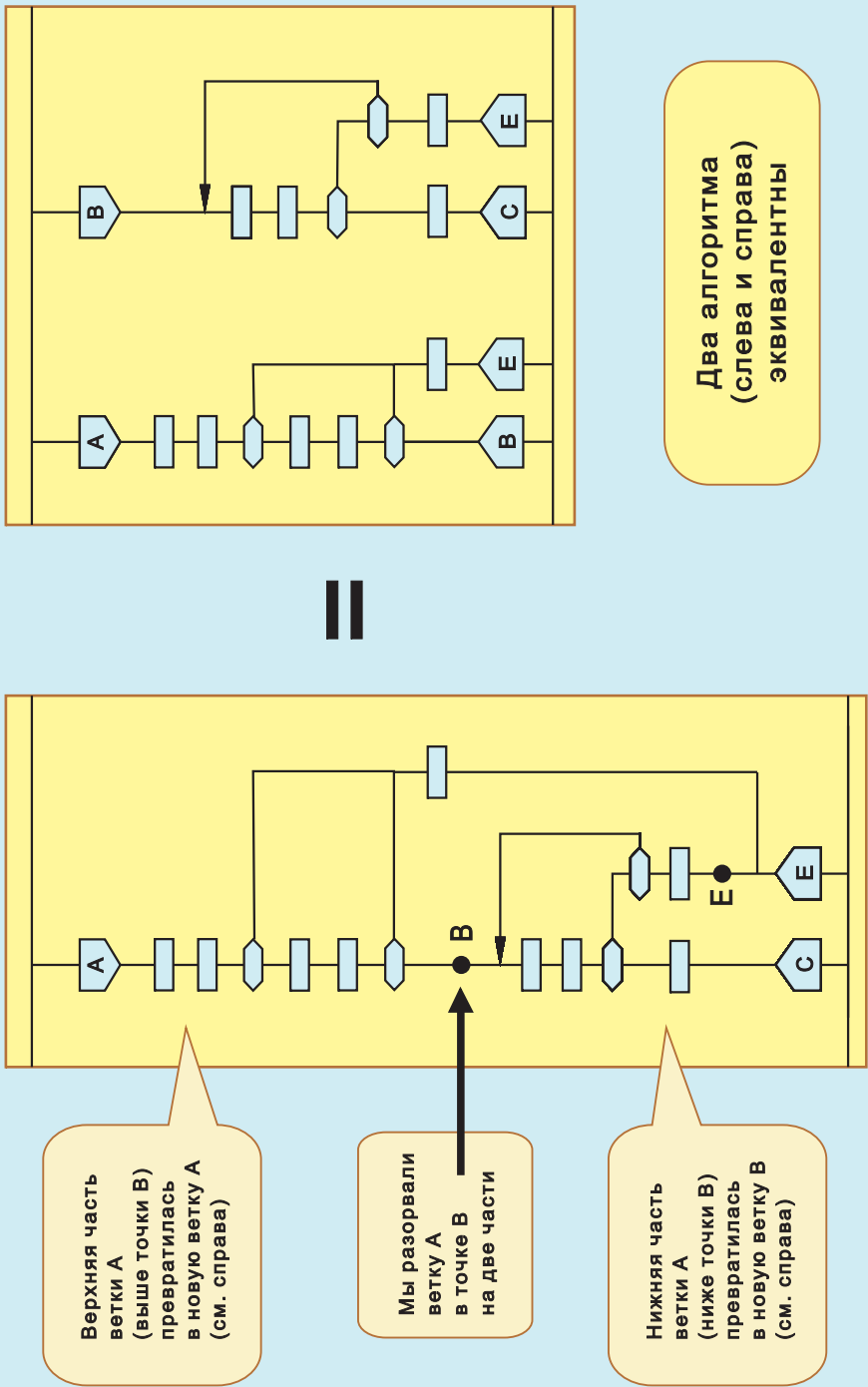
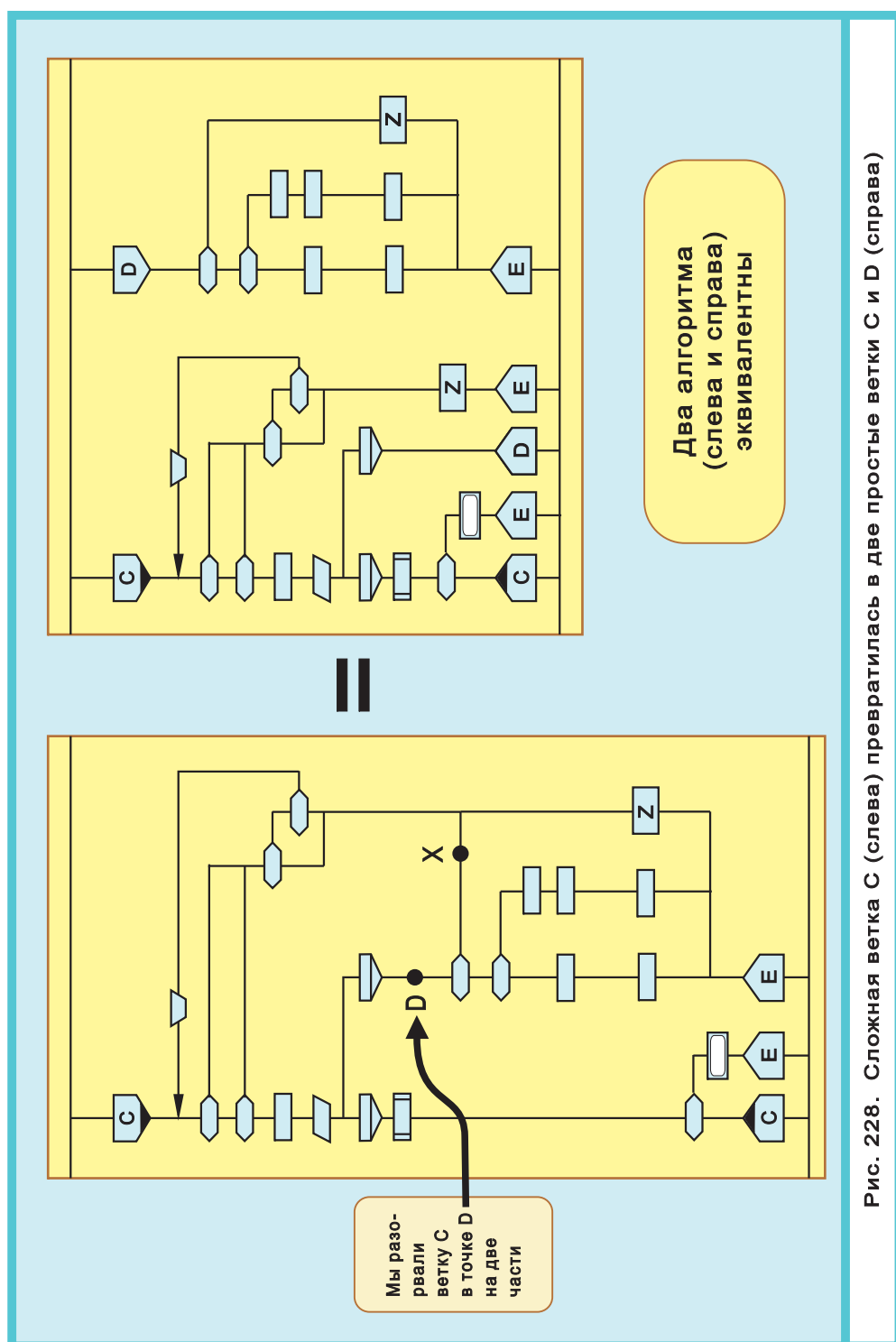


Рис. 227. Сложная ветка A (слева) превратилась в две простые ветки A и B (справа)



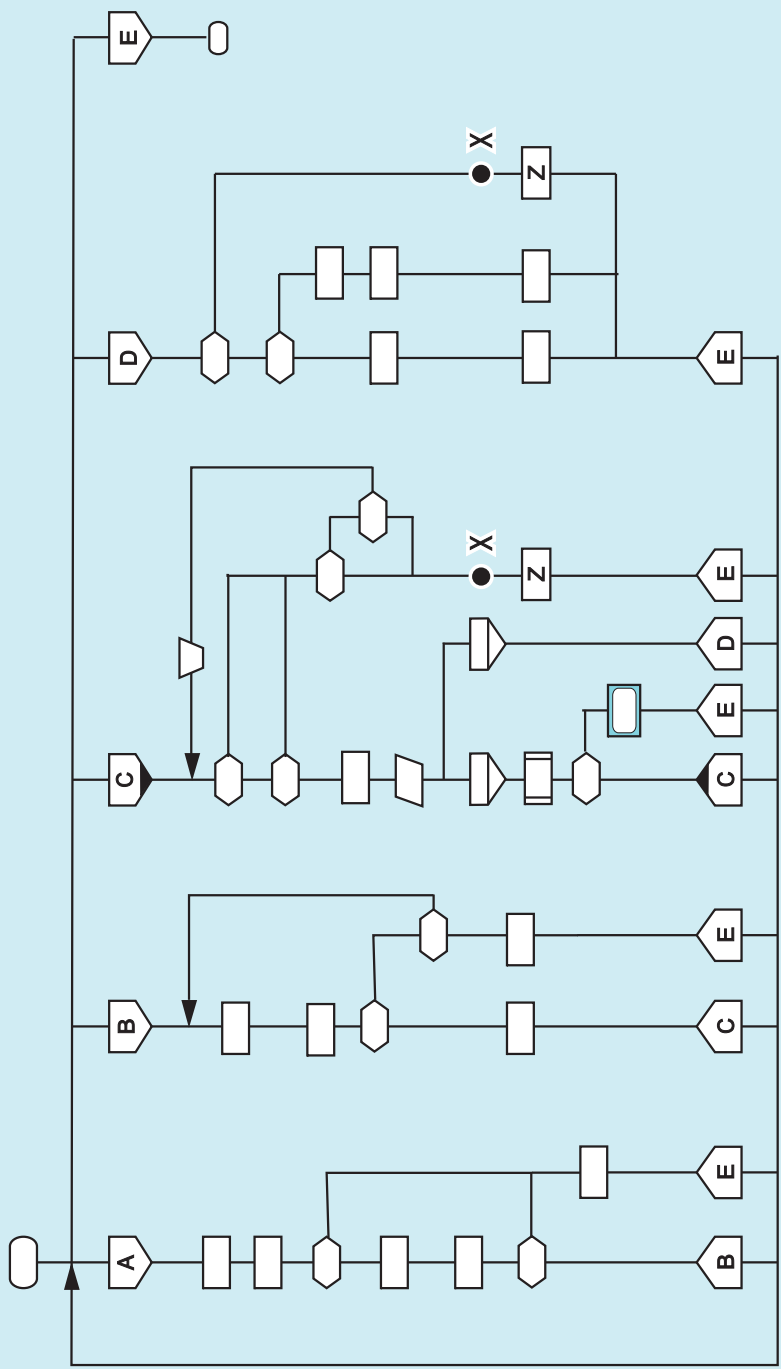


Рис. 229. Результат дробления веток.
Две сложных ветки (A и C) на рис. 225 превратились в четыре простых ветки (A, B, C, D) на этом рисунке. В итоге алгоритм стал более понятным

В отличие от нее, заготовка новой ветки D является «неполноценной». В ней не хватает многих деталей:

- иконы имя ветки D ;
- иконы действие Z ;
- иконы адрес E ;
- соединительных линий, связывающих иконы Z и E между собой и с нижней шиной.

Впрочем, добавление недостающего не составляет труда. Заготовки легко превращаются в новые ветки C и D .

На рис. 229 в ветках C и D для наглядности показаны две точки дублирования X . Под ними ясно видны дублированные иконы Z и E .

После выполнения необходимых шагов исходная трехветочная схема превратилась в преобразованную схему, содержащую пять веток. Зато каждая ветка стала более прозрачной.

Таким образом, хотя число веток увеличилось (с трех до пяти), но в итоге схема стала более простой и более легкой для понимания.

В этом нетрудно убедиться, сравнив рис. 225 и рис. 229.

§13. МЕТОД УКРУПНЕНИЯ ВЕТОК

В этой главе описан метод дробления веток. Он позволяет устранять сложные и громоздкие ветки, расчлняя их на более мелкие и удобные для работы.

Существует и диаметрально противоположный метод – *метод укрупнения веток*. Представим себе, что дробление веток доведено до логического предела и даже до абсурда. Каждая ветка раздроблена на мельчайшие части. Так что тело каждой ветки содержит только один элемент. Например, ветка A содержит одну икону «действие». Ветка B содержит одну икону «вопрос». И так далее.

Силуэт, содержащий столь простые ветки, будем считать черновым.

Черновой силуэт – это структура, расчлняющая алгоритм на простейшие элементы. Разумеется, такие элементарные (одноэлементные) ветки не годятся для работы. К счастью, их можно укрупнить. И довести до приемлемой кондиции. Укрупнение заключается в том, что производится последовательное объединение веток. При этом количество веток в силуэте уменьшается.

Таким образом, алгоритмисту предлагаются два взаимодополняющих метода. Слишком мелкие ветки можно объединять и укрупнять. А слишком сложные и громоздкие можно дробить и измельчать.

В совокупности оба метода (дробление и укрупнение) позволяют выбрать оптимальные размеры веток, обеспечивающие наибольшее удобство для работы и максимальную производительность труда.

Метод укрупнения веток описан в главе 35, §9 и показан на рис. 256–258.

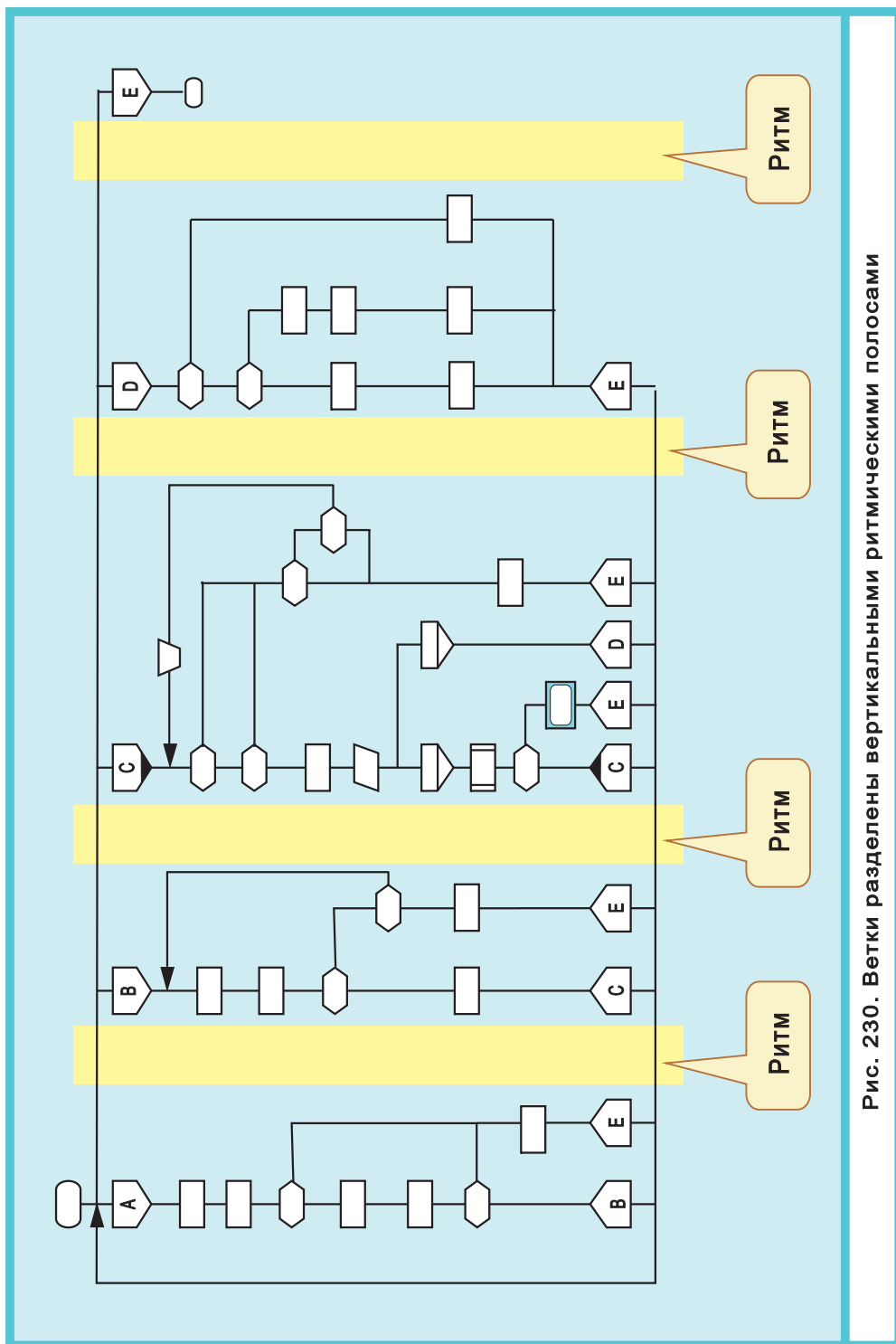


Рис. 230. Ветки разделены вертикальными ритмическими полосами

§14. РИТМИЧЕСКИЕ ПРОМЕЖУТКИ МЕЖДУ ВЕТКАМИ

Чтобы силуэт сделать удобным для зрительного восприятия, желательно между ветками предусмотреть небольшой промежуток, который слегка отодвигает ветки друг от друга. Этот промежуток называется *ритм*. Ритмический промежуток между ветками можно сравнить с пробелами, которые отделяют слова друг от друга при чтении любой книги. Или со свободным от текста «белым пространством», предшествующим началу каждой новой главы в книге.

На рис. 230 показаны четыре широкие ритмические полосы. Каждая ритмическая полоса зрительно отделяет ветки друг от друга. Промежуток между ветками облегчает чтение.

Ритмические полосы на дракон-схемах не изображаются, а подразумеваются.

§15. ВЫВОД

1. Во многих случаях сложную и громоздкую ветку можно заменить на две ветки. В итоге ветка становится более простой и легкой для понимания.
2. При дроблении веток исходная ветка (подлежащая дроблению) и две новые ветки (являющиеся результатом дробления) должны быть эквивалентны.
3. При дроблении число веток в силуэте увеличивается. Зато каждая ветка становится более простой и понятной. В итоге сложность силуэта уменьшается. Он становится более удобным для работы.
4. Наряду с дроблением веток, можно использовать метод укрупнения веток (см. главу 35, §9).
5. Чтобы облегчить зрительное восприятие силуэта, желательно между ветками предусмотреть небольшой промежуток, который называется *ритм*.

Часть VI

**КОНСТРУКТОР
АЛГОРИТМОВ
И ФОРМАЛЬНОЕ
ОПИСАНИЕ ЯЗЫКА**

КОНСТРУКТОР АЛГОРИТМОВ (ПОМОЩНИК ЧЕЛОВЕКА)

§1. ЗАЧЕМ НУЖЕН ДРАКОН-КОНСТРУКТОР?

Разумеется, в случае крайней нужды дракон-схему можно нарисовать и вручную. Либо использовать универсальный графический редактор, например, Visio.

Однако это не лучший способ. Гораздо удобнее воспользоваться специальной программой, которая называется «дракон-конструктор».

В состав дракон-конструктора входит меню графоэлементов (рис. 231). Чтобы нарисовать дракон-схему, пользователь сначала вызывает меню на экран персонального компьютера. А затем с его помощью рисует или, как говорят, конструирует схему. При этом важную роль играют так называемые заготовки.

§2. ЗАГОТОВКА-СИЛУЭТ И ЗАГОТОВКА-ПРИМИТИВ

Чтобы вырастить огромное дерево, нужно бросить в землю маленькое семечко. Любая сколько угодно сложная дракон-схема тоже вырастает из семечка, которое называется *заготовкой*.

Заготовки бывают двух сортов. Одна используется для построения дракон-схемы «силуэт». Из другой получается примитив (рис. 232).

Построение любой дракон-схемы выполняется за конечное число шагов путем соответствующих преобразований выбранной заготовки.

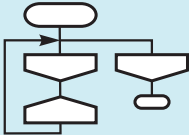


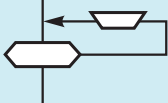








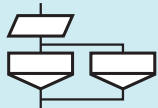


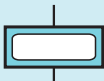
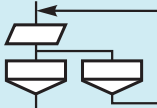



 <div>Силуэт</div>	 <div>Примитив</div>	 <div>Пауза</div>	 <div>Цикл ЖДАТЬ</div>
 <div>Действие</div>	 <div>Развилка</div>	 <div>Таймер</div>	 <div>Формальные параметры</div>
 <div>Вставка</div>	 <div>Обычный цикл</div>	 <div>Синхронизатор</div>	 <div>Полка</div>
 <div>Переключатель</div>	 <div>Цикл ДЛЯ</div>	 <div>Параллельный процесс</div>	 <div>Комментарий</div>
 <div>Переключающий цикл</div>	 <div>Вывод</div>	 <div>Ввод</div>	 <div>Соединитель</div>

Рис. 231. Меню дракон-конструктора

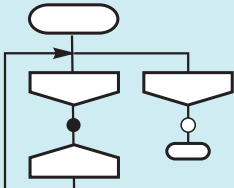

<div>Заготовка для построения силуэта</div> 	<div>Заготовка для построения примитива</div> 
---	---

Рис. 232. Преобразуя заготовки с помощью фиксированного набора формальных визуальных операций, можно построить любую дракон-схему

§3. ЧТО ТАКОЕ АТОМ?

Атом

Это элемент меню на рис. 231, который имеет два вертикальных отростка, лежащие на одной вертикали

Дракон-конструктор может выполнять несколько операций, среди которых важную роль играет команда «ввод атома» (рис. 233). Операция выполняется в два этапа:

- сначала пользователь выбирает нужный атом из меню;
- затем обращается к дракон-схеме и указывает точку, в которую нужно его ввести.

Атомы вставляются не куда попало, а только в разрешенные места, которые называются *валентными точками* дракон-схемы.

Перечень точек включает:

- валентные точки заготовок (отмечены на рис. 232);
- валентные точки макроикон (отмечены на рис. 18);
- входы и выходы атомов.

Ввод атома производится так. Сначала происходит разрыв соединительной линии в выбранной пользователем валентной точке. Затем в место разрыва вставляется атом, как показано на рис. 233.

В реальных дракон-схемах валентные точки не изображаются, а подразумеваются.

§4. ПРИМЕР ПОСТРОЕНИЯ ДРАКОН-СХЕМЫ «ПРИМИТИВ»

Дракон-схема строится на экране компьютера методом «сборки из кубиков». В начале работы пользователь вызывает на экран меню (рис. 231). И размещает его в удобном для себя месте, например в правом верхнем углу экрана. Остальная часть экрана используется как рабочее поле для построения схемы.

Предположим, нужно построить примитив. Пользователь выбирает макроикону «заготовка-примитив» и помещает ее в рабочее поле экрана.

Рассмотрим конкретный пример на рис. 234 и 235.

На первом шаге пользователь вызывает из меню макроикону «обычный цикл». Но куда ее поместить? Пользователь подводит курсор к нужной точке в заготовке-примитив. К той самой точке, в которой следует разорвать соединительную линию, чтобы в образовавшийся разрыв вставить выбранную икону. Результат операции виден на рисунке 234, шаг 1, справа.

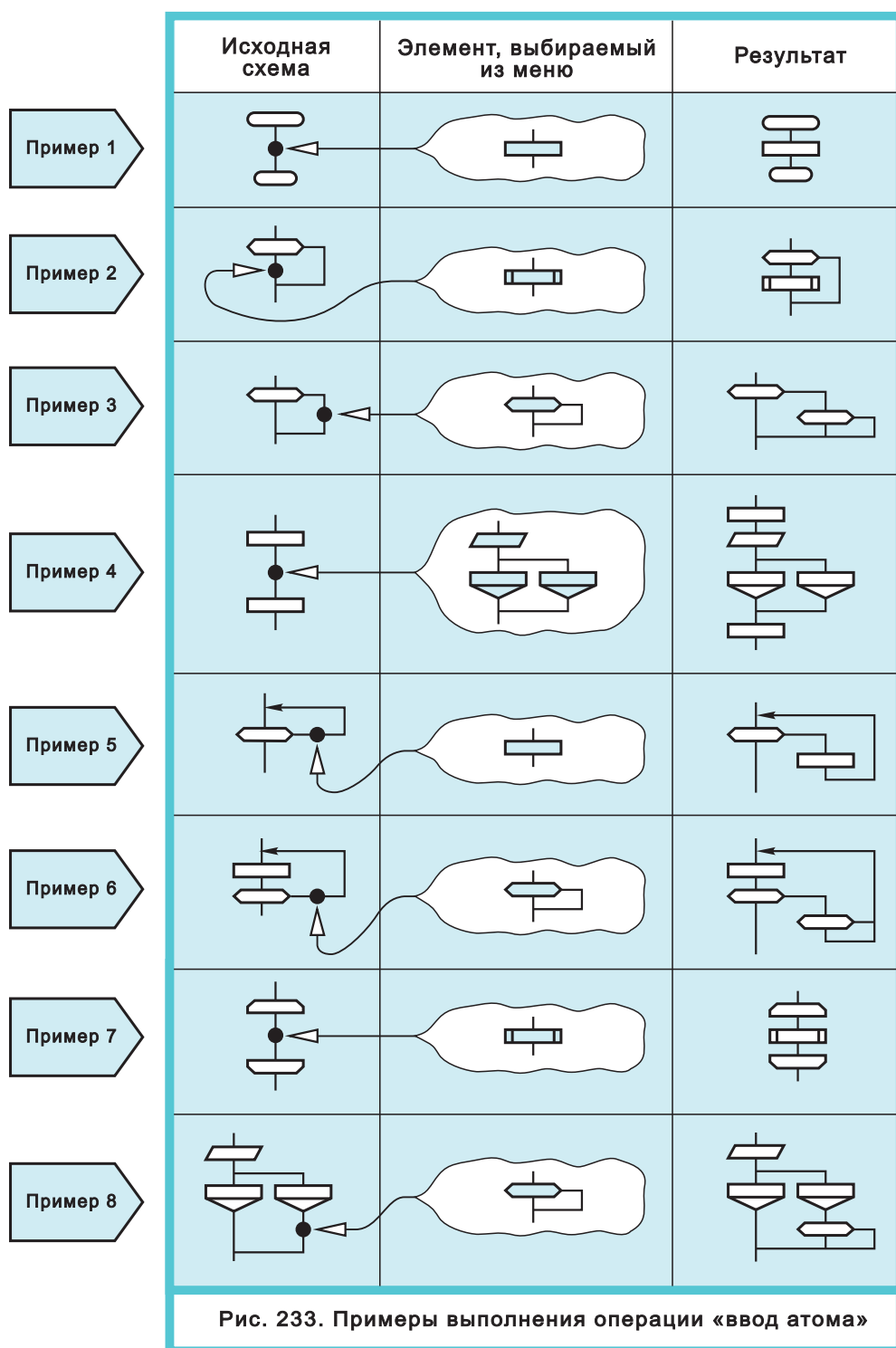


Рис. 233. Примеры выполнения операции «ввод атома»

Два следующих шага выполняются аналогично. В дракон-схему последовательно вводятся две иконы «действие» (рис. 234, шаги 2 и 3).

Далее следует ввод макроиконки «обычный цикл» (рис. 234, шаг 4).

Затем вводятся «вставка», «развилка» и «действие» (рис. 235, шаги 5–7). Результат последней операции показан на рис. 235 (шаг 7, справа).

После того как графический узор (слепыш) дракон-схемы построен, производится заполнение его текстом.

§5. ЧТО ТАКОЕ ЛИАНА?

Обезьяна, сидевшая на дереве, поймала свисавшую сверху лиану. Однако нижняя часть лианы приросла к стволу и не поддавалась. Обезьяна перегрызла ее зубами, уцепилась за конец и мигом перелетела на соседнее дерево, где прочно привязала лиану к ветке.

Нечто подобное умеет делать и дракон-конструктор.

Лиана – это часть дракон-схемы, которая:

- имеет начало и конец (*начало лианы* и *конец лианы*);
- начало лианы находится вверху, конец – внизу;
- от начала к концу проходит хотя бы один маршрут;
- началом лианы служит выход иконы «вопрос» или «вариант» (если этот выход не является петлей цикла);
- концом лианы служит точка слияния.

Лиана есть последовательность шампур-блоков или просто соединительная линия.

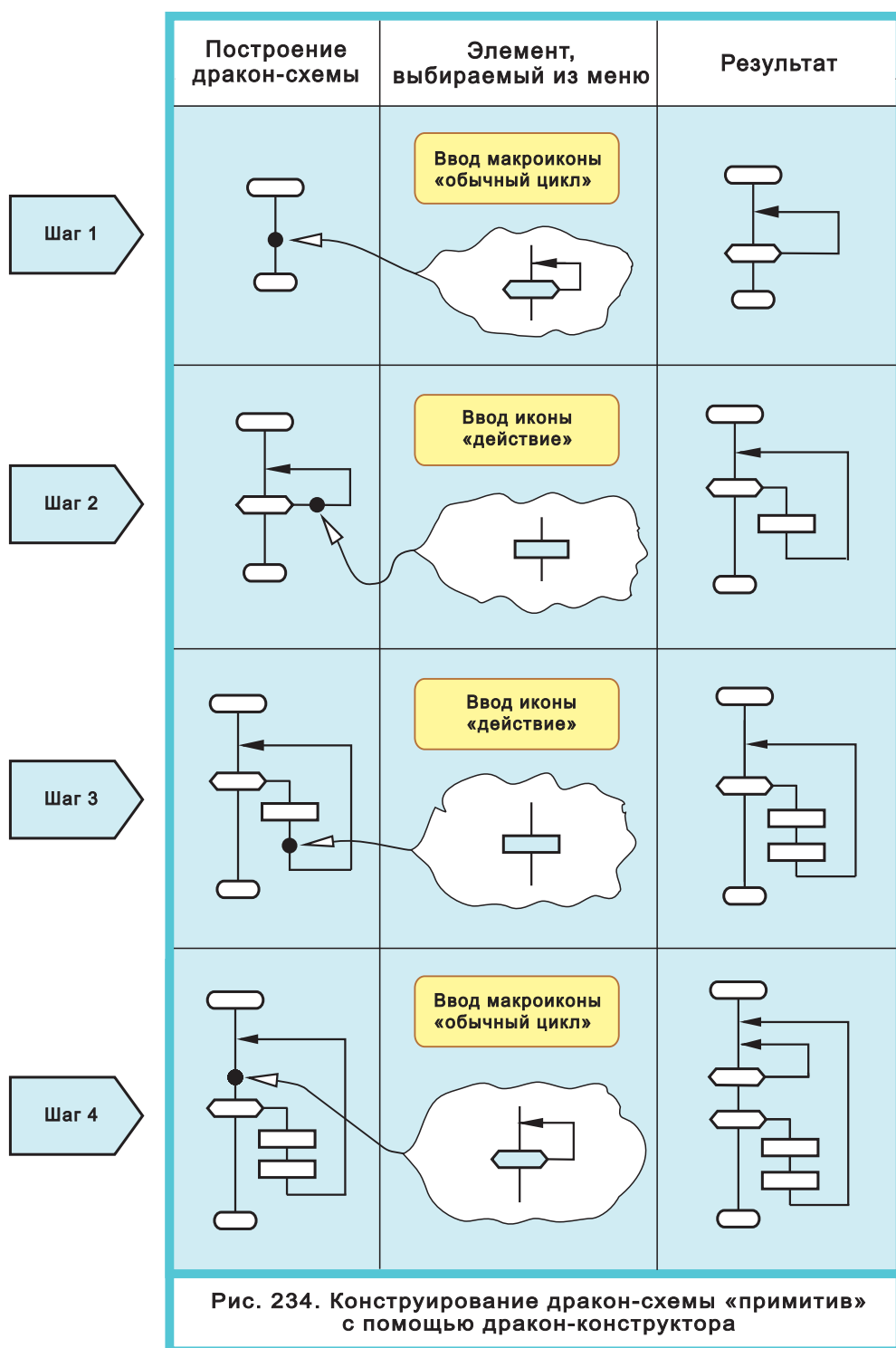
§6. ОПЕРАЦИЯ «ПЕРЕСАДКА ЛИАНЫ»

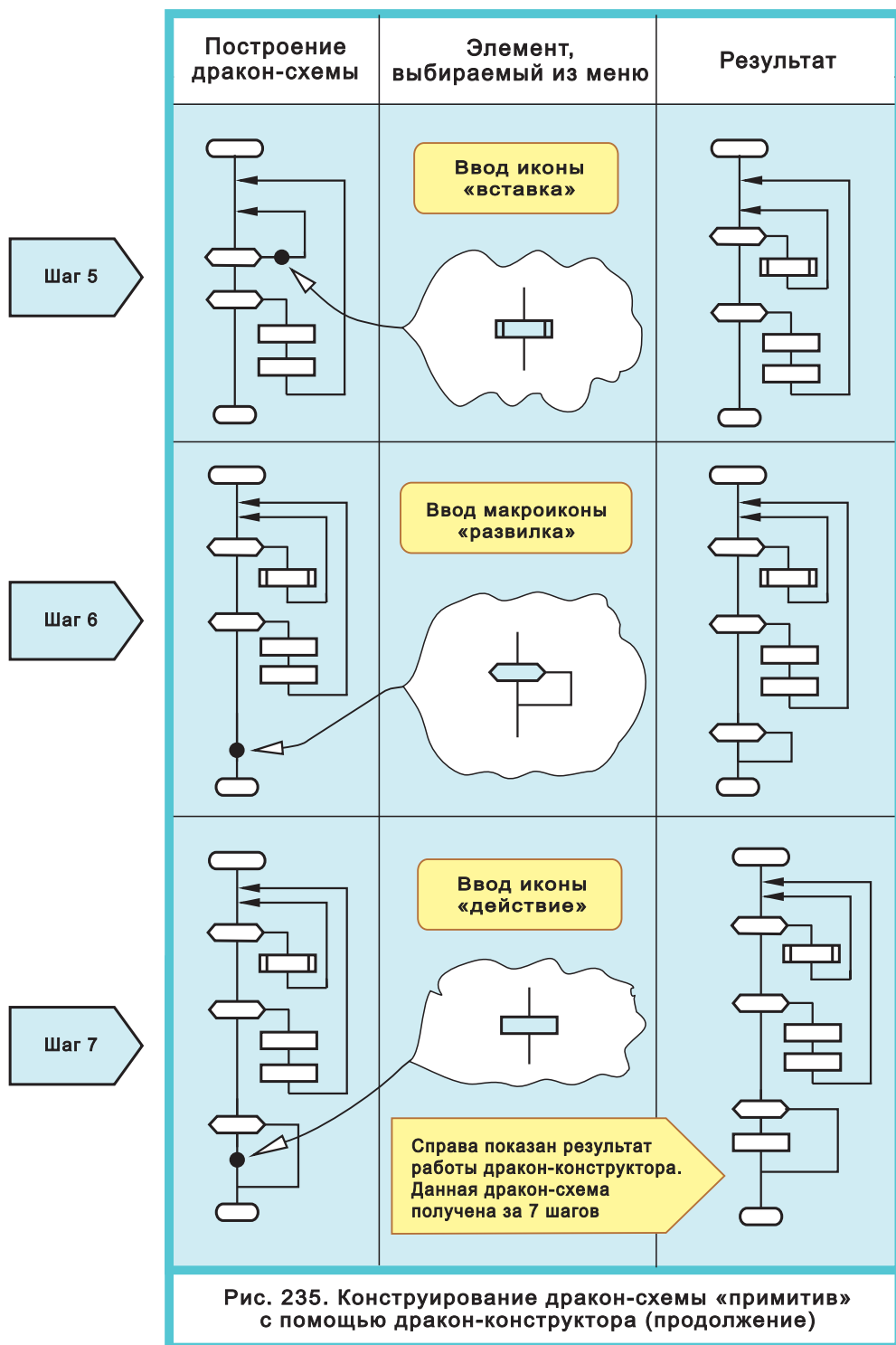
При некоторых условиях (подробно описанных в следующей главе) конец лианы можно оторвать от своего места и присоединить в другую точку дракон-схемы. Такая операция называется *пересадка лианы*. Примеры показаны на рис. 236 и 237.

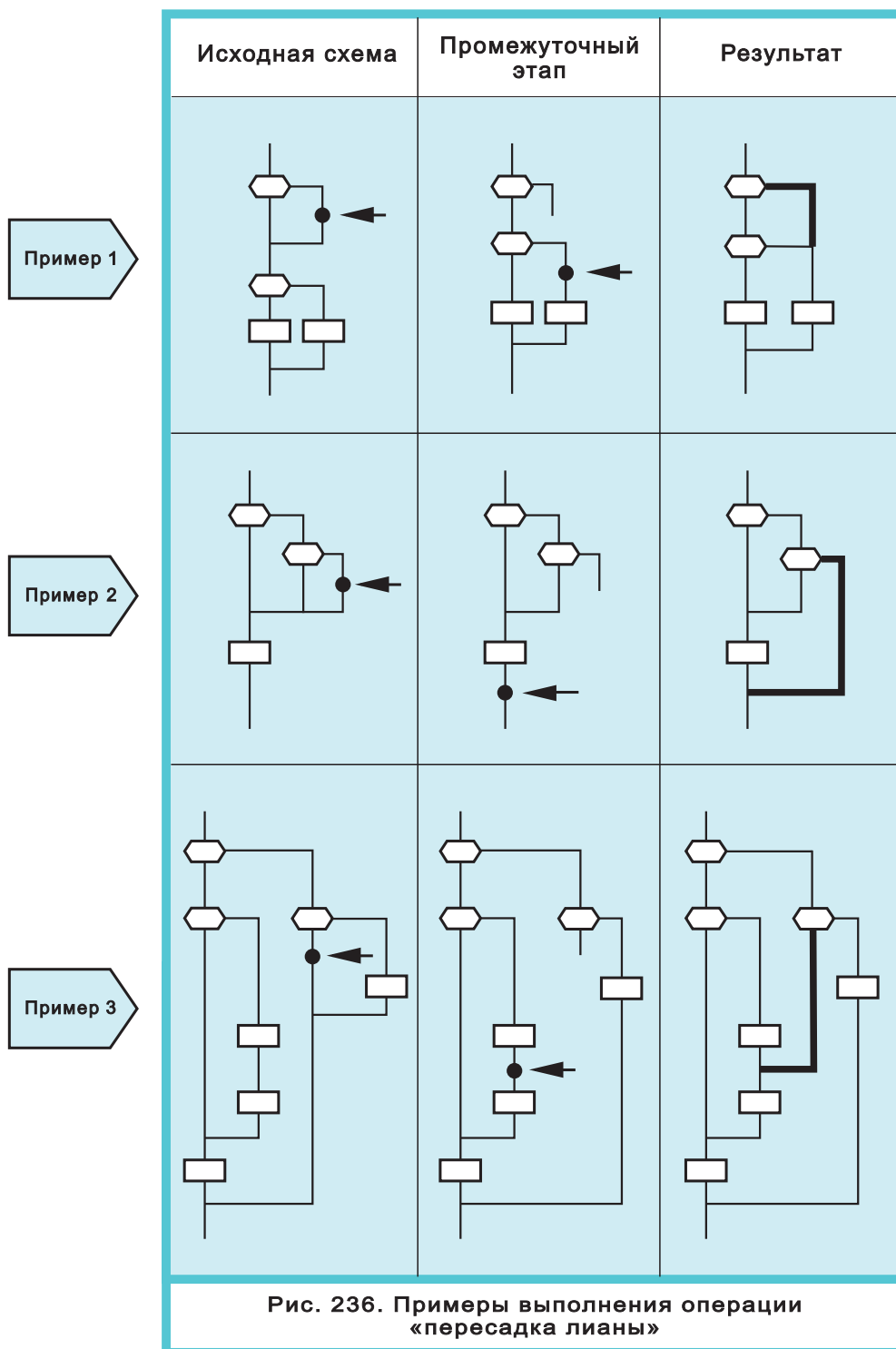
Выполнение этой операции осуществляется в два этапа. Сначала курсор подводится к лиане, конец которой надо освободить (рис. 236, левая графа).

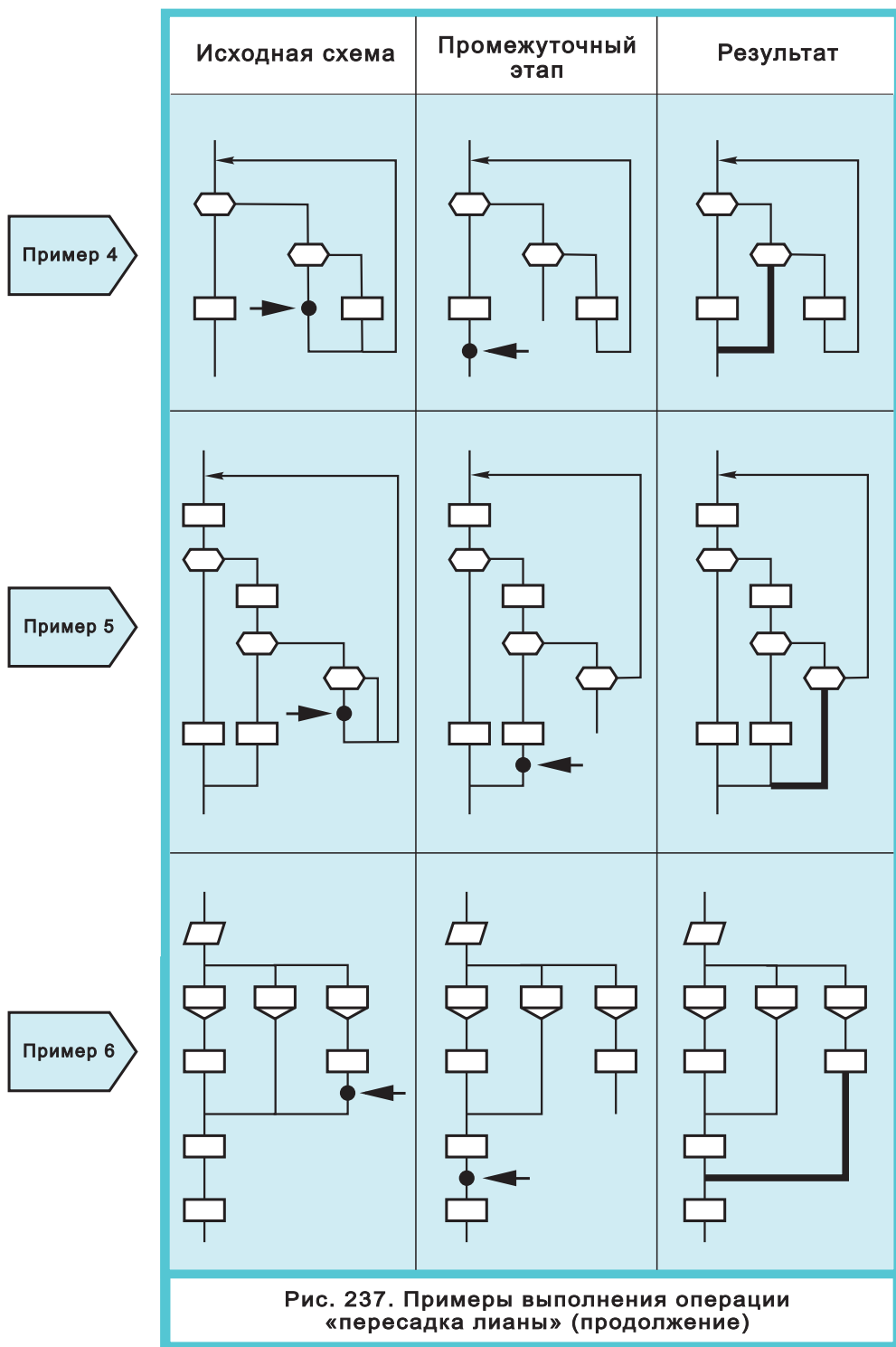
Куда его присоединить? Пользователь выбирает желаемую точку и отмечает ее курсором (рис. 236, средняя графа). Результат операции «пересадка лианы» показан на том же рисунке в правой графе.

Многие дракон-схемы, представленные в этой книге, построены с помощью пересадки лианы. Укажем некоторые из них: рис. 34, 53, 56, 60, 62, 82–84, 87, 105, 107, 108, 110, 123–125, 159.









§7. ОПЕРАЦИЯ «ЗАЗЕМЛЕНИЕ ЛИАНЫ»

Пересадка лианы применима и к примитиву, и к силуэту. В отличие от нее операция *заземление лианы* относится только к силуэту. Она служит для построения веток, имеющих несколько выходов (многоадресных веток).

Чтобы заземлить лиану, необходимо:

- организовать в ветке разветвление (с помощью макроикон «развилка» или «переключатель»);
- оторвать присоединенную к ним лиану от прежнего места;
- присоединить ее через икону «адрес» к нижней горизонтальной линии силуэта, то есть «заземлить» ее.

Операция «заземление лианы» проводится в два этапа.

Первый этап (отрыв нижнего конца лианы от своего места) осуществляется точно так же, как при пересадке лианы (рис. 238, левая графа).

На втором этапе пользователь подводит курсор к нижней линии силуэта, указывая точку, куда лиана может дотянуться, *не пересекая* других линий (рис. 238, средняя графа).

Это действие порождает автоматическое появление в нужном месте иконы «адрес». Лиана автоматически присоединяется к иконе «адрес». И через нее – к нижней линии силуэта (рис. 238, правая графа).

Заземление лианы используется при построении силуэтов с многоадресными ветками. См., например, рис. 25, 34, 85–89, 140, 144, 148, 170.

§8. ПРИМЕР ПОСТРОЕНИЯ ДРАКОН-СХЕМЫ «СИЛУЭТ»

Давайте построим силуэт, изображенный на рис. 85.

Вначале вызовем в рабочее поле заготовку-силуэт (рис. 239, вверху слева).

На первом шаге выполним операцию «добавление ветки». Для этого модифицируем заготовку, автоматически вставляя в нее еще одну ветку (рис. 239, шаг 1).

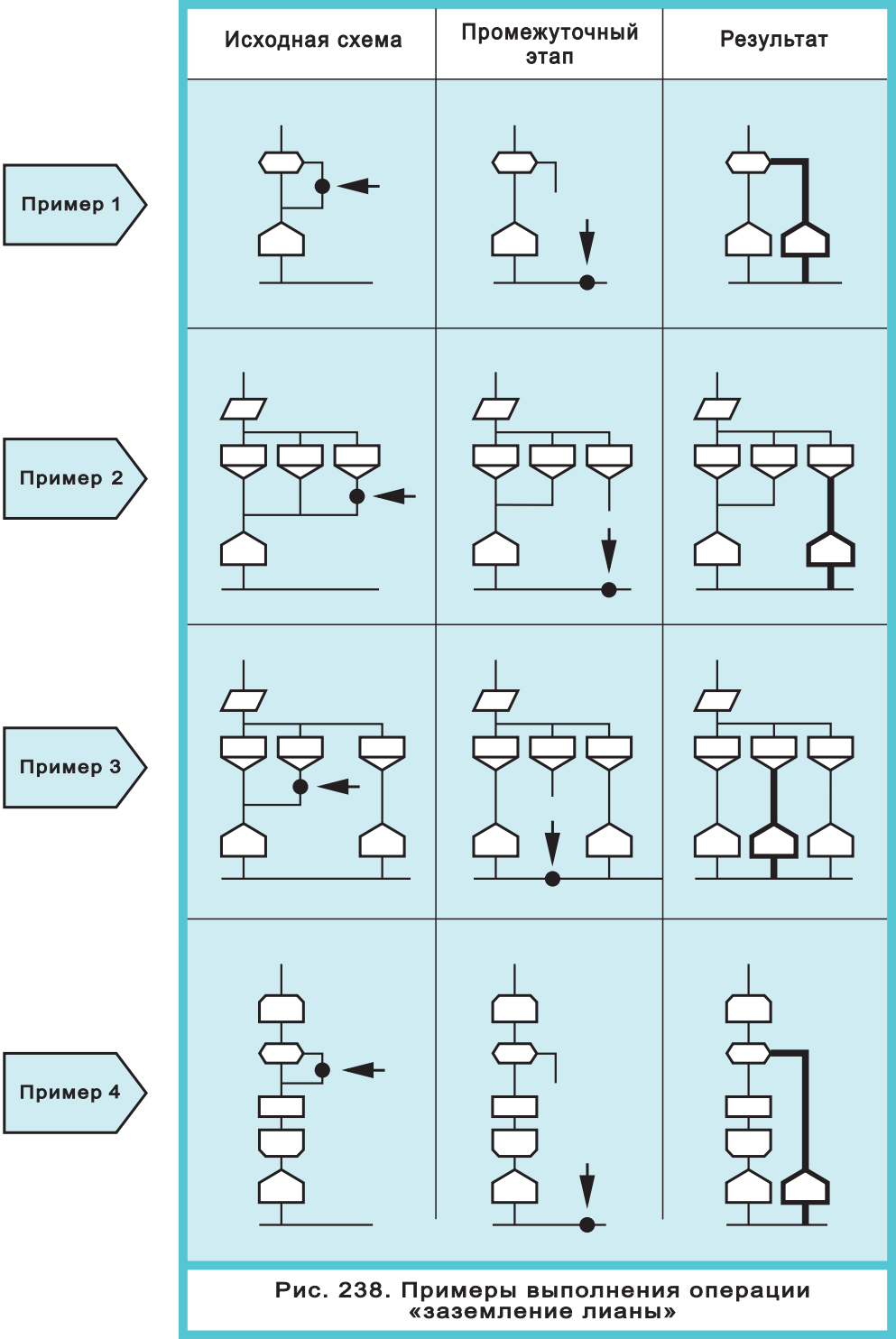
Дальнейший ход строительства ясен из рисунков 239–241.

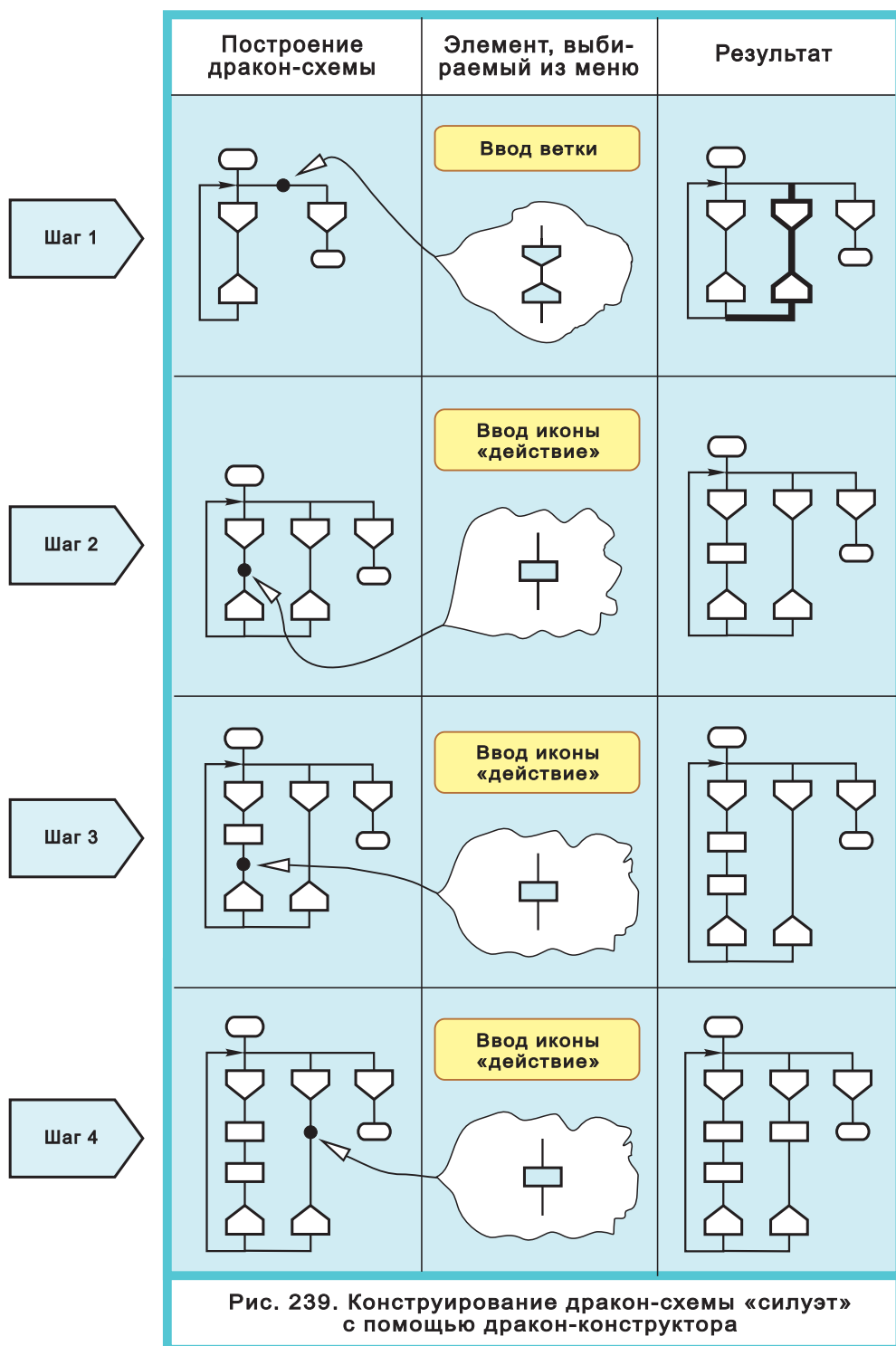
В шагах 2–5 вставим 4 иконы «действие». В шаге 6 вставим макроикону «развилка». Результат операции виден на рис. 240, шаг 6, справа.

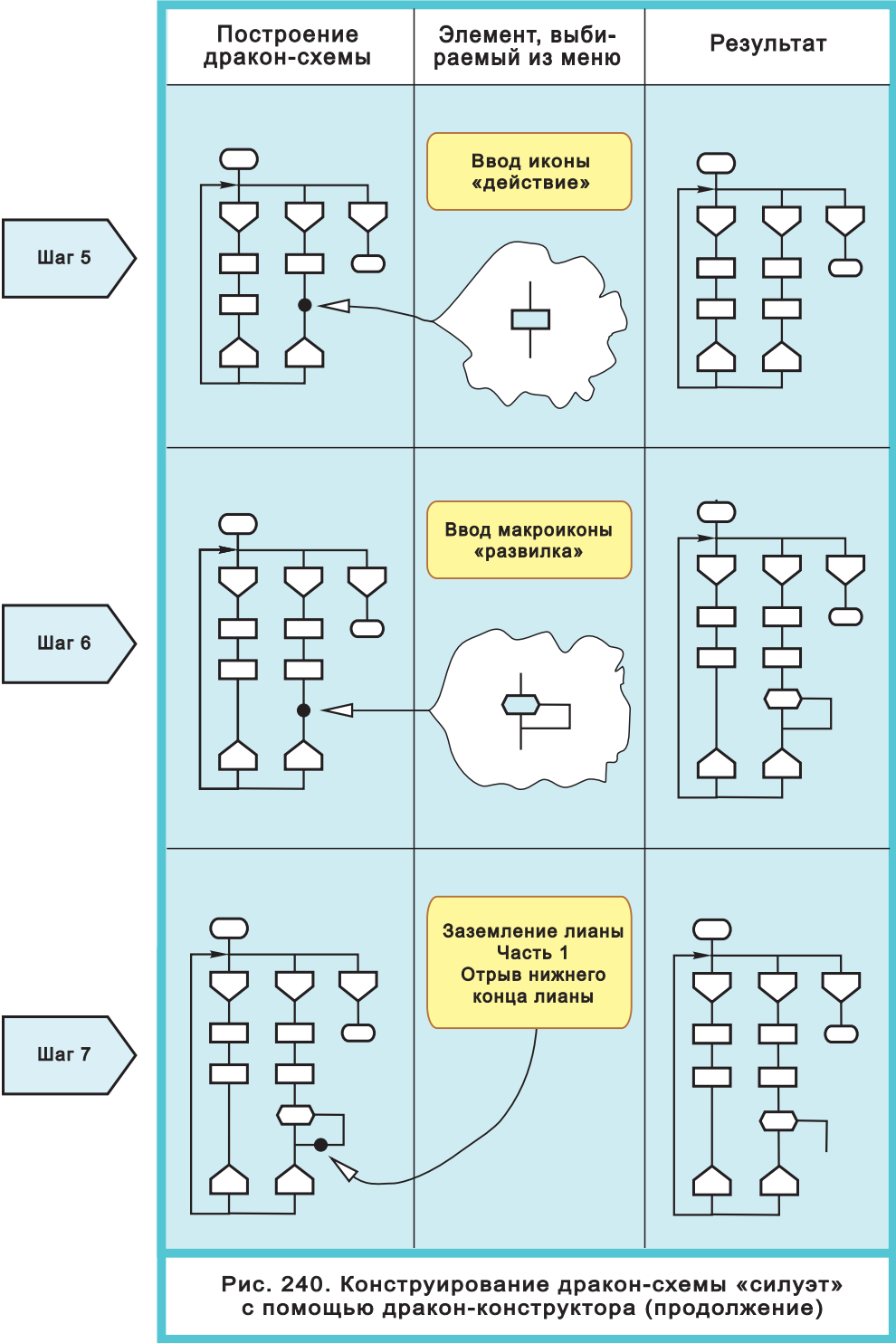
Затем выполним заземление лианы (шаги 7, 8). В шаге 9 добавляем признак веточного цикла (черный треугольник). Для этого помещаем два треугольника в иконы: «имя ветки» и «адрес». В заключение вставим икону «комментарий» в последнюю ветку (шаг 10).

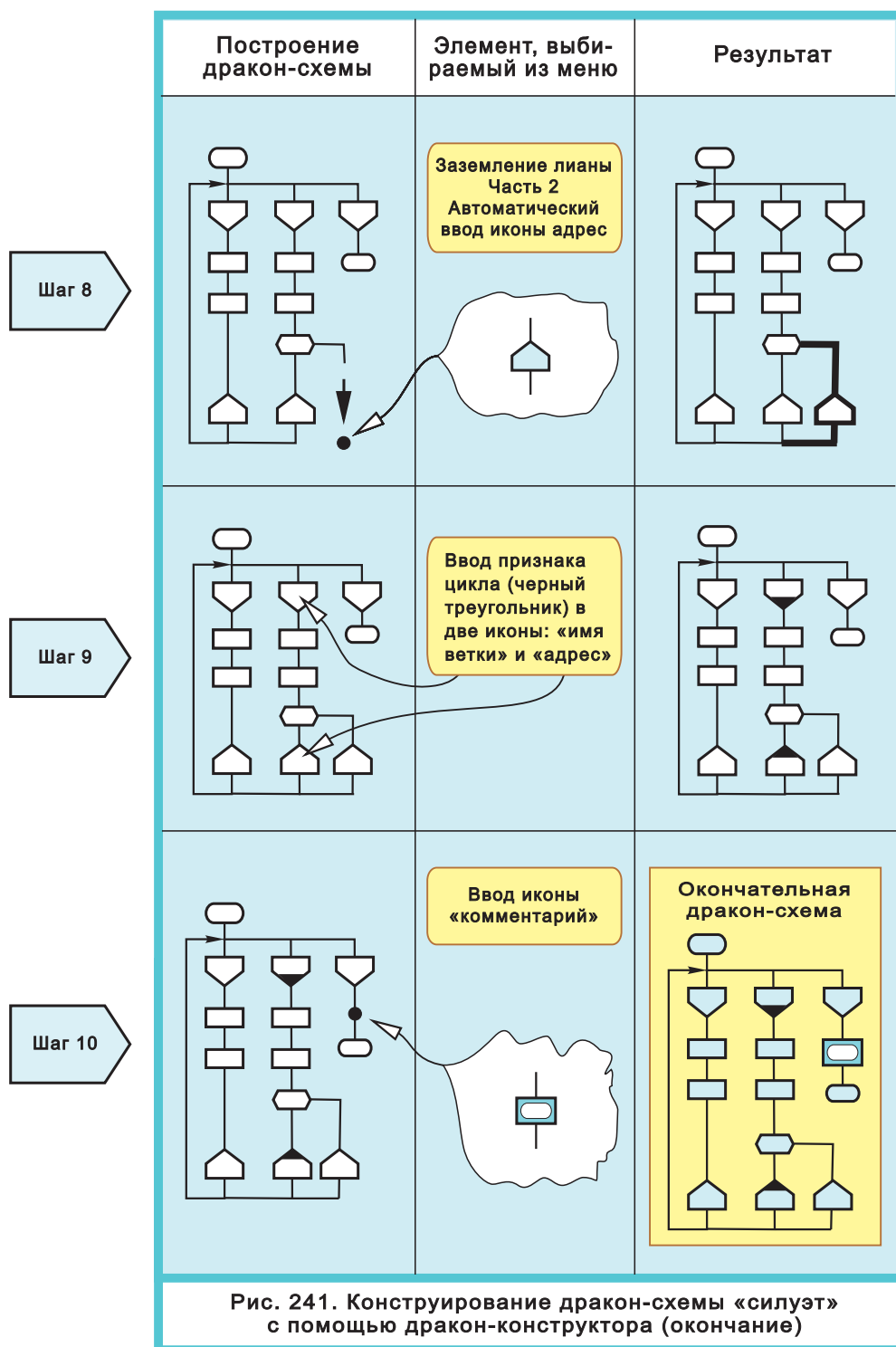
Графическое конструирование заканчивается в момент получения желаемого слепыша (рис. 241, шаг 10, справа).

Затем в иконах записываются текстовые операторы, после чего визуальный алгоритм приобретает окончательный вид, показанный на рис. 85.









§9. ФОРМИРОВАНИЕ НАДПИСЕЙ «ДА» И «НЕТ»

Возле каждой иконы «вопрос» обязательно должны быть надписи «да» и «нет». Эти надписи появляются на дракон-схеме всякий раз, когда из меню вызывается элемент, содержащий икону «вопрос» (рис. 231).

Первоначально дракон-конструктор пишет «да» у нижнего выхода иконы «вопрос» и «нет» – у правого. Пользователь может менять их местами. Для этого в конструкторе предусмотрена операция «да/нет». При выполнении этой операции, слова «да» и «нет» у выходов иконы «вопрос» меняются местами. (При этом все остальные элементы дракон-схемы остаются на своих местах).

Множественное нажатие на кнопку «да/нет» приводит к тому, что «да» и «нет» поочередно меняют свое положение.

§10. ЧЕМ ОТЛИЧАЕТСЯ ОПЕРАЦИЯ «ДА/НЕТ» ОТ РОКИРОВКИ?

При рокировке алгоритм не меняется. Потому что – вместе со сменой «да» и «нет» – плечи развилки тоже меняются местами. Следовательно, рокировка – эквивалентное преобразование алгоритма (см. главу 7).

При операции «да/нет» дело обстоит иначе, так как все остальные элементы остаются на своих местах. Следовательно, операция «да/нет» *изменяет* алгоритм.

§11. КАК ВСТАВИТЬ И УДАЛИТЬ ВЕТКУ

Рассмотрим две задачи:

- вставить ветку в силуэт;
- удалить ветку из силуэта.

Эти задачи можно решать по-разному, например, так.

Чтобы вставить ветку, надо на панели инструментов щелкнуть кнопку «вставить слева» или «вставить справа». Затем щелкнуть на иконе «имя ветки». Новая ветка будет вставлена соответственно СЛЕВА (или СПРАВА) от указанной иконы.

Чтобы удалить ветку, надо на панели инструментов щелкнуть кнопку «удалить». Затем щелкнуть на иконе «имя ветки». Данная ветка будет удалена полностью, включая все входящие в ее состав иконы.

§12. ГРАФИЧЕСКИЕ ПРАВИЛА : ХОРОШО И ПЛОХО

Рисунки, представленные в книге, во многих случаях выполнены по эргономическим правилам дракон-конструктора. Но не всегда. В ряде случаев из-за недостатка места на странице пришлось уплотнять и «искажать» рисунки. То есть нарушать эргономические правила.

Подобные искажения можно видеть на рис. 25. Укажем две ошибки:

- Многие иконы имеют разную ширину (разный горизонтальный размер).
- У некоторых икон вертикальная линия не является осью симметрии. Например, икона комментариев в ветке «Ожидание клева» и правая икона «Насади червяка» в ветке «Рыбацкая работа» расположены несимметрично из-за недостатка места.

Один из вариантов исправленной схемы дан на рис. 241а.

Рекомендация. Ширина всех икон данной ветки силуэта должна быть одинаковой. Пользователь может задать ширину по своему усмотрению. Разные ветки могут иметь разную ширину икон. (Исключениями являются иконы: «конец», «пауза», «период», «пуск таймера», «синхронизатор»).

Пример. На рис. 241а первые три ветки (Рыбная ловля, Ожидание клева, Рыбацкая работа) имеют ширину икон 19 мм. В последней ветке (Обратная дорога) ширина икон увеличена до 22 мм.

Зачем увеличена ширина? Для того, чтобы текст в двух последних иконах уместился на четырех или пяти строках (рис. 241а). Иначе (при ширине 19 мм) текст занял бы семь или восемь строк. И, следовательно, был бы неудобен для чтения (рис. 25).

Во многих случаях подобные сложности не нужны. Потому действует более простая

Рекомендация. Конструктор алгоритмов автоматически задает ширину всех икон силуэта.

Конструктор алгоритмов должен обеспечить максимальные удобства для пользователя. Программа должна по выбору пользователя выполнять любую из указанных рекомендаций (а также множество других рекомендаций, которые здесь для краткости не рассматриваются).

§13. СТРУКТУРИЗАЦИЯ ТЕКСТА В ИКОНЕ «КОММЕНТАРИЙ»

Прочитайте комментарий на рис. 171. Обратите внимание на отбивки (увеличенные интервалы между абзацами) и маркеры (черные кружки).

Отбивки и маркеры играют важную роль. Если их убрать, получим сплошной, монолитный текст, который неудобно читать. Чтобы помочь читателю, облегчить его труд и уменьшить нагрузку на глаза, конструктор алгоритмов – по указанию пользователя – должен уметь выполнять *структуризацию* текста в иконе «комментарий».

Структуризация улучшает читабельность текста за счет дробления монолитного текста на несколько приятных для глаза блоков. При этом взгляд читателя *легко* переходит от одного абзаца к другому.

Интервал до или после абзаца служит для визуального отделения одного абзаца от другого за счет пустого пространства между ними.

Структуризация призвана сделать длинный текст удобочитаемым. Обратите внимание на следующие детали (см. комментарий на рис. 171):

- Текст в иконе комментарий разбивается на части (в нашем случае, на 6 частей).
- Все части сверху и снизу разделяются отбивками (число отбивок на единицу больше числа частей; на данном рисунке 7 отбивок).
- Первая часть, если она есть, играет роль заголовка (например, «Промывать глаз можно»). Заголовок не имеет маркера.
- Остальные части начинаются с маркера.

Более простой случай показан на рис. 196 в правой ветке. Там изображены 3 маркированные части и 4 отбивки. Заголовок нет.

Возможны вариации. На рис. 186 комментарий имеет 3 части и 4 отбивки. Третья часть в свою очередь делится на три однострочных абзаца, выделяемые маркерами. Обратите внимание: однострочные абзацы не имеют разделяющих отбивок. Кроме того, в данном случае слева от маркера для выразительности делается отступ.

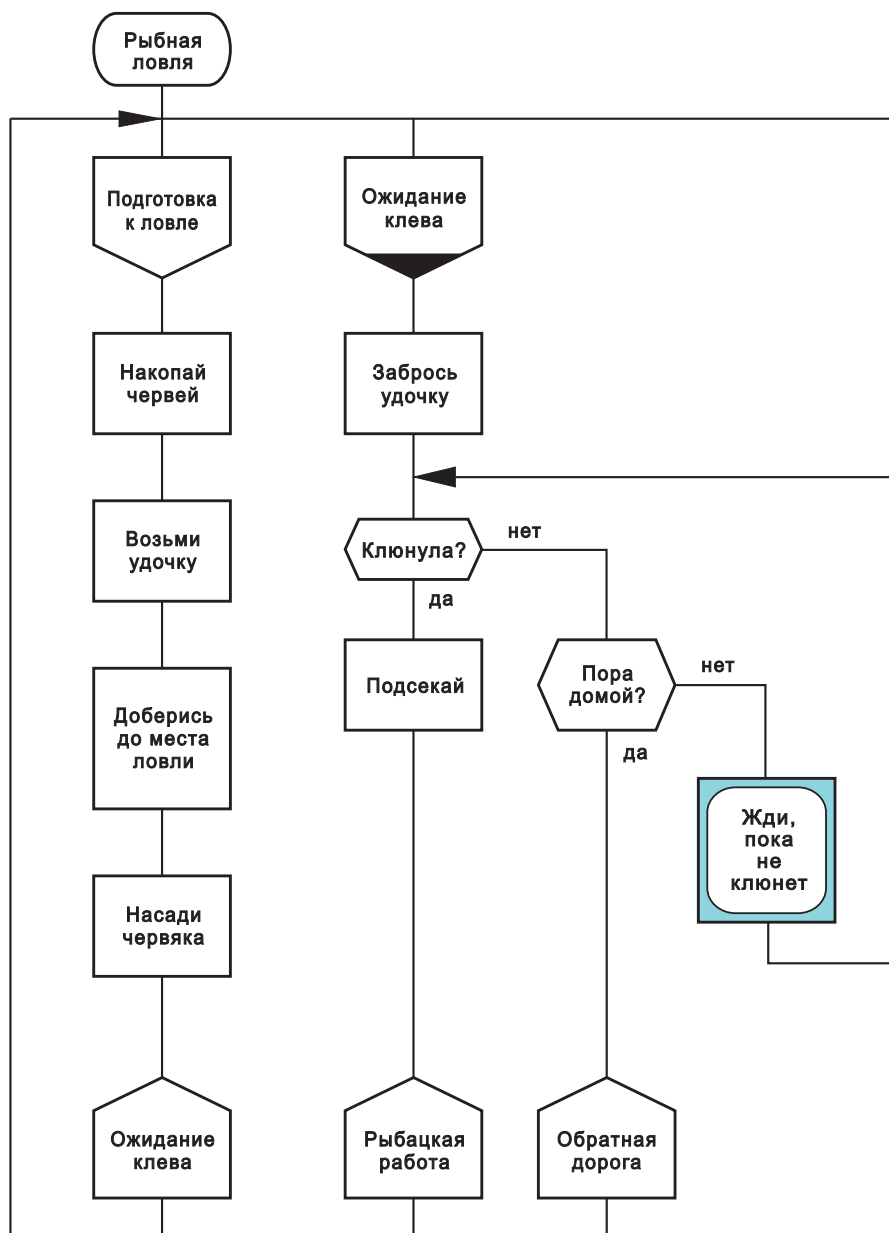
См. также рис. 187 и 197.

Чтобы обеспечить эргономичную структуризацию текста в иконе комментарий, конструктор алгоритмов должен поддерживать, как минимум, три инструмента: *отбивки, маркеры и отступы*.

Можно предложить и более серьезные требования, но здесь не место говорить о деталях.

§14. СТРУКТУРИЗАЦИЯ ТЕКСТА В ИКОНЕ «ДЕЙСТВИЕ»

Рассмотрим случай, когда надо создать дракон-схему алгоритма и больше ничего. То есть нужна просто «картинка», а трансляция в программный код и исполнение на компьютере не требуются.



Сравните рис. 25 и рис. 241а.

На рис. 25 есть недочеты:

- многие иконы имеют неодинаковый горизонтальный размер (ширину);
- у некоторых икон вертикальная линия не является осью симметрии

На рис. 241а недочеты исправлены так:

- первые три ветки имеют ширину икон 19 мм;
 - в последней ветке ширина увеличена до 23 мм.
- Эти изменения позволили улучшить эргономичность схемы и читабельность текста в двух последних иконах.

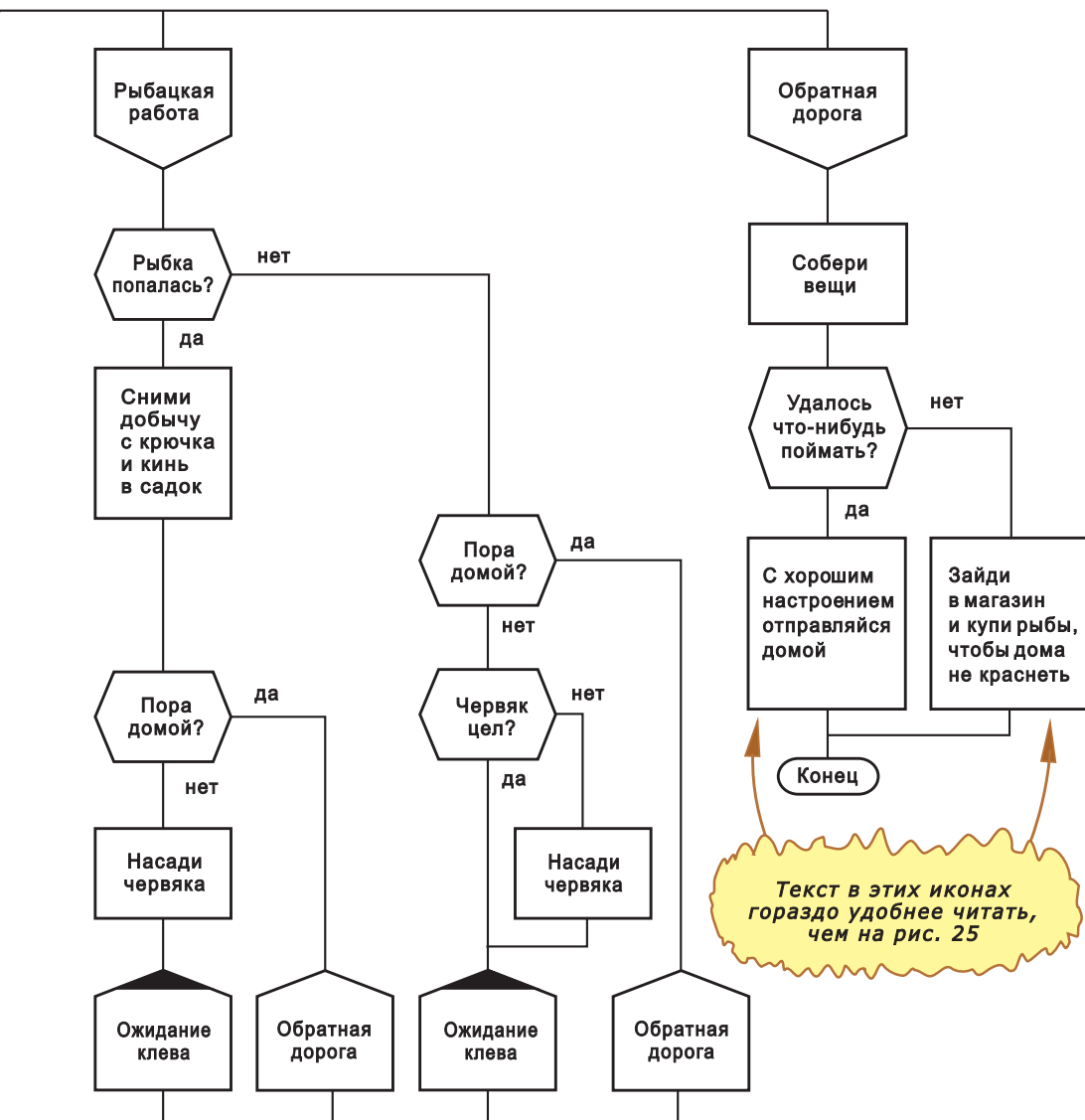


Рис. 241а. Алгоритм «Рыбная ловля»

При таких условиях иногда можно ослабить формальные (математические) требования к дракон-схеме. И разрешить писать в иконе «действие» словесное описание действий. Это может привести к ситуации, когда в иконе «действие» появится длинный текст.

Длинный, то есть сплошной, монолитный текст неудобен для чтения. Возникает уже знакомая нам ситуация. Но на этот раз применительно не к комментарию, а к тексту в иконе «действие».

Решение проблемы подробно описано в §13.

Примеры структуризации текста в иконе «действие» приведены на рис. 170, 196, 197, 206.

§15. ГДЕ СКАЧАТЬ ДРАКОН-КОНСТРУКТОР?

Первоначально среда разработки алгоритмов и программ на языке ДРАКОН была создана в Федеральном космическом агентстве для проектирования систем управления ракетно-космической техники. Она успешно применяется для разработки алгоритмов и программ бортового компьютера «Бисер», установленного на борту космических ракет – см. с. 515.

На втором этапе возникла необходимость приспособить эту среду для гражданских нужд широкого применения – для эксплуатации на персональных компьютерах, ноутбуках и др.

Эту задачу попытался решить Геннадий Тышов. Созданная им программа называется ИС Дракон. Ее можно скачать здесь:

<http://forum.oberoncore.ru/viewtopic.php?p=22669#p22669>

<http://forum.oberoncore.ru/viewforum.php?f=79>

Интернет-консультации можно получить по адресам:

Визуальный язык ДРАКОН.

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

Алгоритмы в дракон-схемах. Обсуждение способов и правил изображения алгоритмов. Примеры.

<http://forum.oberoncore.ru/viewforum.php?f=78>

§16. ВЫВОДЫ

1. Хотя общее число икон и макроикон языка ДРАКОН равно 45, для построения любой дракон-схемы достаточно иметь небольшое меню, содержащее всего 20 графоэлементов.
2. Графическое меню существенно облегчает работу пользователя – оно дает возможность конструировать дракон-схему методом «сборки из кубиков». Для этого служит операция «ввод атома», позволяющая доставать кубики из меню и укладывать их в проектируемую дракон-схему.
3. Другие операции («пересадка лианы», «заземление лианы» и т. д.) разрешают вносить в схему логические детали, расширяющие ее функциональные возможности и улучшающие эргономическое качество.
4. Во внутренних алгоритмах дракон-конструктора реализован полный набор правил визуального синтаксиса языка ДРАКОН, что освобождает пользователя от необходимости подробно запоминать синтаксические правила.
5. Дракон-конструктор создает только правильно построенные графические схемы и исключает возможность появления запрещенных схем. Отсюда вытекает, что при работе с дракон-конструктором ошибки визуального синтаксиса *принципиально невозможны*.

ГРАФИЧЕСКИЙ СИНТАКСИС ЯЗЫКА ДРАКОН

§1. ОБЩИЕ ПОНЯТИЯ

Тезис 1. Иконы – визуальные буквы, образующие визуальный алфавит языка ДРАКОН, представленный на рис. 17.

Тезис 2. Заготовка-примитив и заготовка-силуэт – фигуры, показанные на рис. 232.

Предварительный тезис 3. Примитив – фигура, полученная путем преобразования заготовки-примитив за конечное число шагов с помощью фиксированного набора операций (перечисленных ниже в тезисе 36).

Предварительный тезис 4. Силуэт – фигура, полученная путем преобразования заготовки-силуэт за конечное число шагов с помощью фиксированного набора операций (перечисленных ниже в тезисе 37).

Тезис 5. Дракон-схема – общее понятие для обозначения примитива и силуэта.

§2. ШАМПУР-БЛОК

Тезис 6. Шампур-блок – часть дракон-схемы, имеющая один вход сверху и один выход снизу, содержащая одну или несколько икон, причем:

- вход и выход лежат на одной вертикали, через которую проходит путь от входа к выходу;
- через каждую икону, входящую в состав шампур-блока, проходит хотя бы один путь от входа к выходу;
- в состав шампур-блока могут входить любые иконы за исключением следующих: заголовок, конец, формальные параметры, петля силуэта, соединитель.

Тезис 7. Главная вертикаль шампур-блока – вертикаль, соединяющая его вход и выход.

Остальные вертикали, если они есть, находятся правее главной. Все вертикали шампур-блока ориентированы сверху вниз, кроме цепей, используемых для организации петли цикла.

§3. ОПЕРАЦИЯ «ВВОД АТОМА»

Тезис 8. Атомы – фигуры, изображенные на рис. 242. Эти фигуры используются в операции «ввод атома». Любой атом является шампур-блоком.

Тезис 9. Валентная точка – точка, принадлежащая заготовке или дракон-схеме, в которой разрешается произвести разрыв соединительной линии, чтобы в место разрыв вставить атом с помощью операции «ввод атома». Для краткости валентную точку можно называть *точкой ввода*.

Тезис 10. Ввод атома – преобразование заготовки или дракон-схемы, выполняемое следующим образом: производится разрыв соединительной линии в валентной точке и в это место вставляется атом, как показано на рис. 233.

§4. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ ОБ АТОМАХ

Тезис 11. Атомы делятся на простые и составные. Простой атом содержит одну икону, составной – не менее двух или разветвление (рис. 242).

Тезис 12. Функциональный атом – простой атом, не являющийся пустым оператором. Таковы все простые атомы, кроме комментария.

Тезис 13. Составные атомы бывают пустые и непустые. В непустом есть хотя бы один функциональный атом. В пустом нет ни одного.

Тезис 14. В полностью законченной дракон-схеме не должно быть ни одного пустого атома (так как последний эквивалентен пустому оператору). Пустые атомы разрешается использовать на всех этапах построения дракон-схемы, кроме заключительного.

§5. КРИТИЧЕСКИЕ И НЕЙТРАЛЬНЫЕ ТОЧКИ

Тезис 15. Валентные точки (точки ввода) делятся на нейтральные и критические.

Тезис 16. Точка называется нейтральной, если применение операции «ввод атома» к данной точке является возможным, но не обязательным. В отличие от нее критическая точка требует обязательного ввода атома.

Тезис 17. Валентные точки находятся в заготовках и атомах. Они показаны на рис. 232 и 242, где нейтральные точки обозначены светлыми кружками, критические – жирными точками.

Тезис 18. Если в фигуре (заготовке или атоме) одна критическая точка, ввод атома обязательно производится именно в нее; при этом крити-

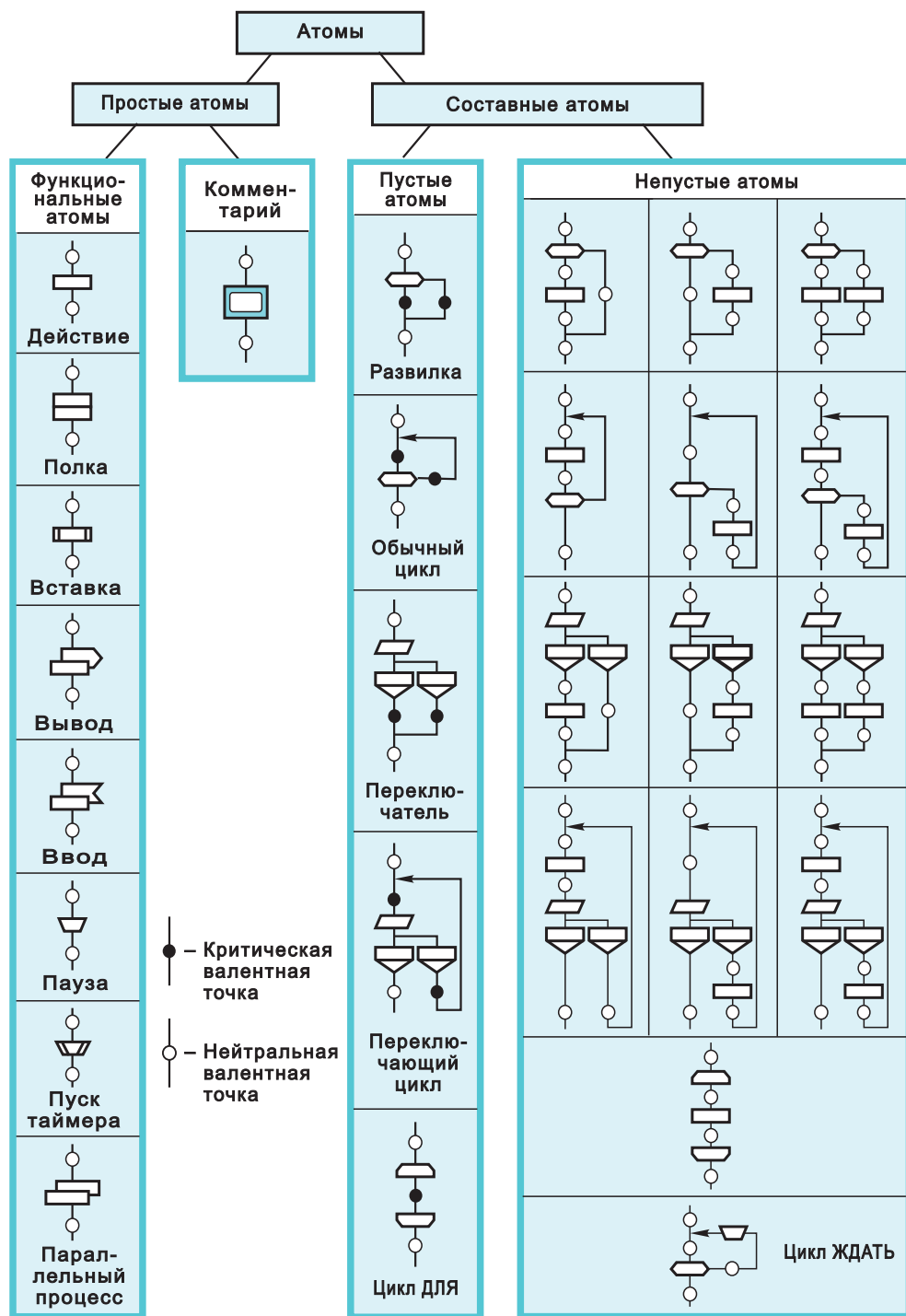


Рис. 242. Атомы и валентные точки

ческая точка уничтожается. Если фигура имеет две критические точки, обязательный ввод атома делается только в одну из них; при этом критическая точка, в которую произведен ввод, уничтожается, а другая критическая точка нейтрализуется, т. е. становится нейтральной.

Тезис 19. Полная совокупность критических точек охватывает:

- критические точки в пустых атомах;
- одну критическую точку в заготовке-примитив;
- одну критическую точку в заготовке-силуэт.

Тезис 20. Полная совокупность нейтральных точек охватывает:

- входные и выходные точки атомов;
- две внутренние точки в атоме «цикл ЖДАТЬ»;
- одну точку в заготовке-силуэт;
- точки, полученные в результате нейтрализации критических точек.

§6. ПРАВИЛА ИСПОЛЬЗОВАНИЯ ОПЕРАЦИИ «ВВОД АТОМА» ПРИ ПОСТРОЕНИИ ДРАКОН-СХЕМЫ

Тезис 21. Операция «ввод атома» применяется для ввода только простых и пустых атомов, а также цикла ЖДАТЬ. Ввод непустого атома осуществляется в два этапа; сначала вводится пустой атом, затем в его критическую точку вводится функциональный атом.

Пояснение. Ввод пустого атома – очень удобный строительный прием. Он позволяет обеспечить богатство и разнообразие создаваемых дракон-схем и используемых в них конфигураций. Среди последних особую роль играет так называемая «матрешка».

Тезис 22. *Матрешка* – фигура, полученная путем ввода пустого атома в критическую точку пустого атома, а также путем многократного вложения пустых и непустых атомов друг в друга (рис. 243).

Тезис 23. Матрешка бывает пустой (если все содержащиеся в ней атомы пустые), частично пустой (если в ней есть как пустые, так и непустые атомы) и непустой (если все ее атомы непустые). См. рис. 244–246.

Пояснение. После того как пользователь эффективно использовал пустые атомы для придания дракон-схеме желаемой конфигурации, он должен убрать их из схемы.

Тезис 24. Чтобы устранить пустые атомы из дракон-схемы, есть два способа:

- превратить пустой атом в непустой;
- преобразовать пустой атом в пустую матрешку, затем превратить ее в непустую.

Тезис 25. Устранение из дракон-схемы пустых атомов автоматически приводит к уничтожению всех критических точек.

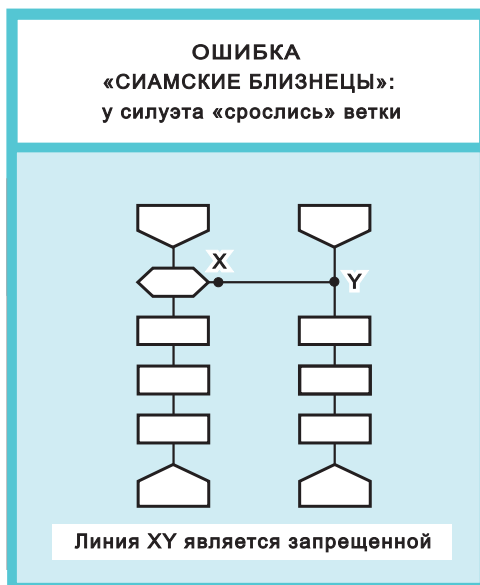
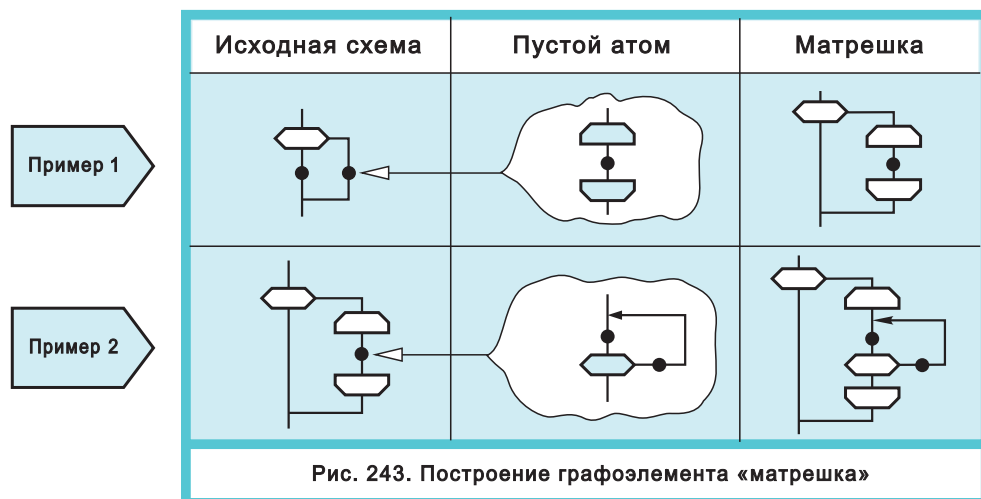


Рис. 247. Ошибка «Сиамские близнецы» — это запрещенная связь между ветками

§7. ЛИАНА

Тезис 26. Лиана – часть дракон-схемы, имеющая один вход сверху и один выход снизу, именуемые «началом лианы» и «концом лианы» соответственно. Началом лианы может быть любой выход икон «вопрос» и «вариант», если он (выход) не является петлей цикла. Концом лианы считается точка слияния, в которой нижняя часть лианы соединяется с другой линией (концом лианы не может быть неразветвленный вход иконы).

Тезис 27. Лиана может быть нагруженной (если она содержит иконы) и ненагруженной (если это просто линия).

§8. ПЕРЕСАДКА ЛИАНЫ

Тезис 28. Пересадка лианы – преобразование дракон-схемы, выполняемое за четыре шага.

Шаг 1. Производится отрыв конца лианы от точки присоединения (рис. 236, 237).

Шаг 2. Конец лианы с помощью вертикальных и горизонтальных линий присоединяется к любой валентной точке, куда лиана может дотянуться без пересечения с другими линиями (рис. 236, 237). При этом запрещается:

- формировать второй вход в ветку (ошибка «сиамские близнецы» – см. рис. 247);
- образовывать новый цикл;
- создавать второй вход в цикл.

Однако разрешается строить новый путь из середины обычного цикла к единственному входу в этот цикл, создавая визуальный эквивалент оператора *continue* языка Си (см. рис. 167, пример 7, а также рис. 83).

Шаг 3. Производится эквивалентное преобразование топологии дракон-схемы, чтобы

- лиане не пришлось загибаться наверх (рис. 248);
- соблюдались правила построения шампур-блока (рис. 249).

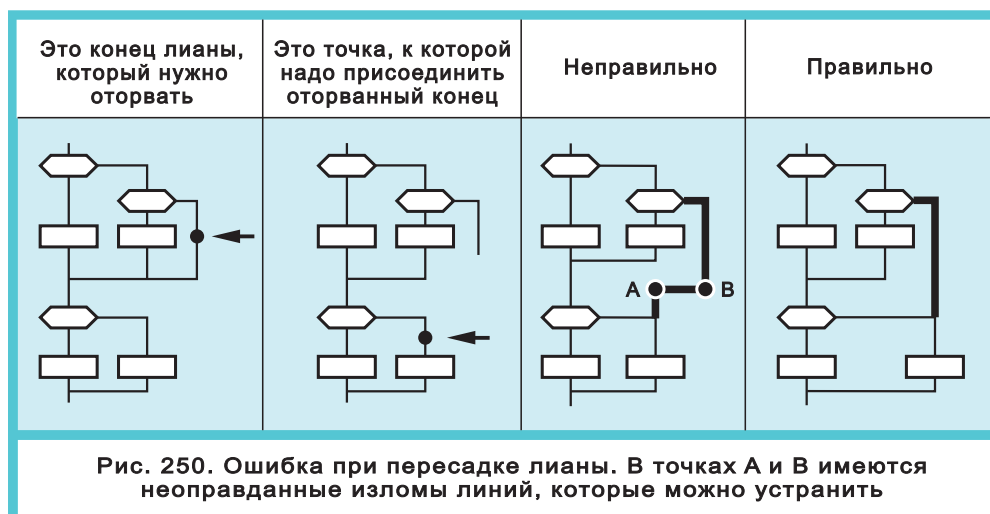
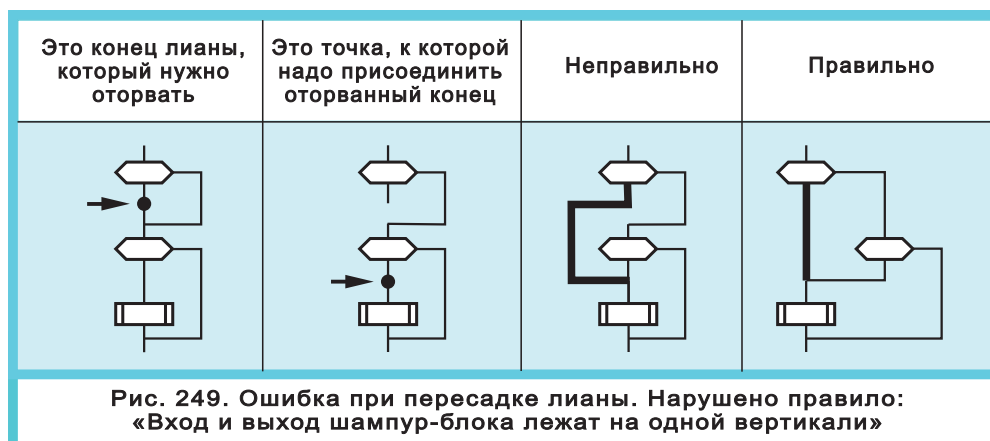
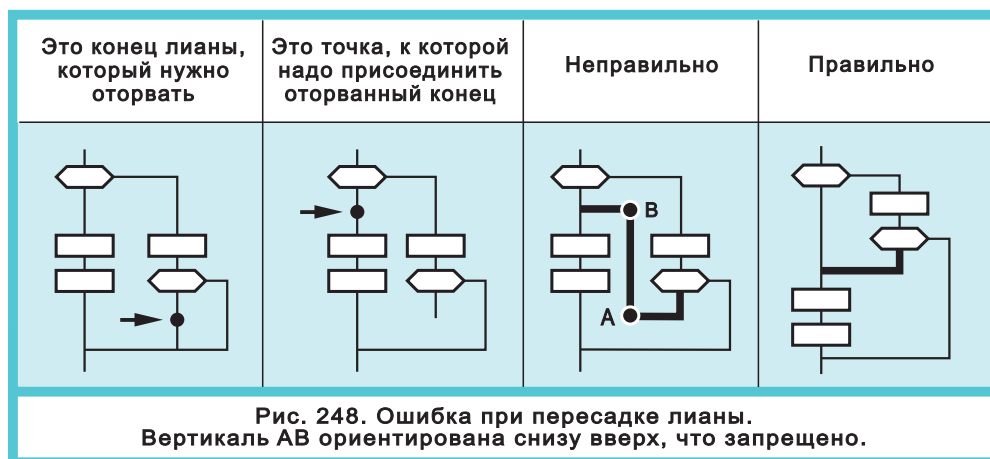
Шаг 4. Устраняются неоправданные изломы линий (рис. 250).

§9. ЗАЗЕМЛЕНИЕ ЛИАНЫ

Тезис 29. Заземление лианы – преобразование дракон-схемы, выполняемое за четыре шага.

Шаг 1. Производится отрыв конца лианы от точки присоединения (рис. 238).

Шаг 2. Конец лианы с помощью вертикальной линии присоединяется к любой точке нижней горизонтальной линии силуэта, куда он



может дотянуться, не пересекая другие линии.

Шаг 3. Производится разрыв линии в нижней части лианы и в место разрыва вставляется икона «адрес» (рис. 238, 240 (шаг 7), 241 (шаг 8).

Шаг 4. Устраняются неоправданные изломы линий.

§10. ПРОЧИЕ ОПЕРАЦИИ

Тезис 30. Боковое присоединение – преобразование дракон-схемы, с помощью которого в схему добавляются иконы «синхронизатор» или «формальные параметры».

Икона «синхронизатор» размещается слева от другой иконы и соединяется с ней горизонтальным отростком. Перечень икон, к которым осуществляется боковое присоединение синхронизатора, показан на рис. 18 (п. 8–20).

Икона «формальные параметры» размещается справа от иконы «заголовок» и соединяется с ней горизонтальным отростком, как показано на рис. 18 (п. 1).

Тезис 31. Добавление варианта – преобразование дракон-схемы, с помощью которого в атом «переключатель» добавляется еще одна икона «вариант». Число добавлений не более 14, так что максимальное число вариантов в переключателе равно 16. Эти цифры могут быть увеличены.

Тезис 32. Добавление ветки – преобразование силуэта, в который добавляется еще одна ветка. Число добавлений не более 30, так что максимальное число веток в силуэте равно 32. Эти цифры могут быть увеличены.

Тезис 33. Удаление последней ветки – преобразование силуэта, при котором удаляется крайняя правая ветка. Этот прием используется при описании бесконечного процесса, как показано в примерах на рис. 140 и 148.

О г р а н и ч е н и е. В силуэте не может быть меньше двух веток.

Тезис 34. Удаление конца примитива – преобразование примитива, при котором удаляется икона «конец». Это необходимо для описания бесконечного параллельного процесса.

Тезис 35. Дополнительный вход – преобразование силуэта, с помощью которого добавляется еще одна икона «заголовок», которая размещается над любой иконой «имя ветки» (кроме левой) и соединяется с ней вертикальным отростком. При этом на верхней горизонтальной линии силуэта рисуют направленную вправо стрелку, как показано в примере на рис. 144 справа.

Ограничение. При наличии веточного цикла запрещается присоединять дополнительный заголовок к середине веточного цикла.

§11. ОСНОВНЫЕ РЕЗУЛЬТАТЫ

Тезис 36. Любая правильно построенная дракон-схема «примитив» является результатом преобразования заготовки-примитив с помощью конечного числа операций: ввод атома, пересадка лианы, добавление варианта, боковое присоединение, удаление конца примитива.

Тезис 37. Любая правильно построенная дракон-схема «силуэт» является результатом преобразования заготовки-силуэт с помощью конечного числа операций: ввод атома, добавление ветки, пересадка лианы, заземление лианы, добавление варианта, боковое присоединение, удаление последней ветки, дополнительный вход.

Пояснение. Тезисы 36 и 37 могут рассматриваться как окончательные определения понятий «примитив» и «силуэт».

§12. ВЫВОДЫ

1. Изложенные выше 37 тезисов (вместе с рисунками, на которые они ссылаются) дают однозначное описание визуального синтаксиса.
2. Это описание является достаточным для построения дракон-конструктора, способного решить две задачи:
 - нарисовать (в соответствии с указаниями пользователя) любую абстрактную дракон-схему (слепыш), принадлежащую множеству правильно построенных (удовлетворяющих требованиям визуального синтаксиса) дракон-схем;
 - создать в памяти компьютера формальное описание построенной схемы, пригодное (после заполнения икон текстовыми операторами) для трансляции в объектные коды или для выполнения программы в режиме интерпретации.

Часть VII

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ЯЗЫКА ДРАКОН



Это самая сложная часть во всей книге. Если вам не по душе математика, пропустите ее.

Данная часть предназначена для тех, кого интересуют вопросы теории и математическое обоснование языка ДРАКОН, включая математическую логику, исчисление икон, метод Ашкрофта-Манни и т.д.

ИСЧИСЛЕНИЕ ИКОН

Для кого предназначены главы 34, 35 и 36?

Данный материал предназначен не для начинающих, а для профессионалов, которые хотят познакомиться с теоретическим фундаментом языка ДРАКОН.

Если же вы только начинаете изучать язык ДРАКОН, если ваша цель – научиться рисовать дракон-схемы и пользоваться программой «дракон-конструктор», вам не стоит тратить время и читать главы 34, 35 и 36.

§1. ВВЕДЕНИЕ

Глава посвящена исследованию связи языка ДРАКОН и математической логики. Мы покажем, что:

- графический синтаксис языка ДРАКОН опирается на идеи математической логики;
- исчисление икон – это раздел математической логики;
- внутренние алгоритмы дракон-конструктора реализуют исчисление икон, то есть опираются на положения математической логики;
- математическую логику можно рассматривать как одно из теоретических оснований языка ДРАКОН.

§2. ШАМПУР-СХЕМА

Шампур-схема – абстрактная дракон-схема. Подчеркнем, что шампур-схема по определению является абстрактной, то есть полностью лишенной текста.

§3. ВИЗУАЛЬНОЕ ЛОГИЧЕСКОЕ ИСЧИСЛЕНИЕ

Попытаемся взглянуть на графический (визуальный) синтаксис языка ДРАКОН с позиций математической логики. Нашему взору откроется необычная картина. Оказывается, любая абстрактная дракон-схема (шампур-схема) представляет собой теорему, которая строго выводится (доказывается) из двух аксиом, каковыми являются заготовка-примитив и заготовка-силуэт.

– Какие же это аксиомы? – вправе удивиться читатель.
– Ведь это просто картинки-слепыши! А шампур-схемы вовсе не похожи на теоремы! Кто и зачем их должен доказывать? Наверно, это шутка или метафора.

– Вовсе нет, отнюдь не метафора. Ниже будет показано, что визуальный синтаксис ДРАКОНА построен как логическое исчисление (назовем его «исчисление икон»). Данное исчисление можно рассматривать как раздел *визуальной математической логики*.

Последнее понятие не является традиционным. Математическая логика и ее основные понятия (исчисление, логический вывод и т. д.) сформировались в рамках текстовой парадигмы. В данной главе, по-видимому, впервые вводятся визуальные аналоги этих понятий. И на их основе строится исчисление икон.

§4. ОБЩЕИЗВЕСТНЫЕ СВЕДЕНИЯ О МАТЕМАТИЧЕСКОЙ ЛОГИКЕ

Принципиальным достижением математической логики является разработка современного аксиоматического метода, который характеризуется тремя чертами:

- явной формулировкой исходных положений (аксиом) развиваемой теории (формальной системы);
- явной формулировкой правил логического вывода, с помощью которых из аксиом выводятся теоремы теории;
- использованием формальных языков для изложения теорем рассматриваемой теории [1].

Основным объектом изучения в математической логике являются логические исчисления. В понятие исчисления входят:

- а) формальный язык, который задается с помощью алфавита и синтаксиса,
- б) аксиомы,
- в) правила вывода [1].

Таким образом, исчисление позволяет, зная аксиомы и правила вывода, получить (то есть вывести, доказать) все теоремы теории. Причем теоремы, как и аксиомы, записываются на формальном языке.

Напомним, что в рамках математической логики следующие термины можно рассматривать как синонимы:

- логическое исчисление,
- формальная система
- теория.

Следовательно, теоремы исчисления, теоремы формальной системы и теоремы теории – одно и то же.

§5. ОБ ОДНОМ РАСПРОСТРАНЕННОМ ЗАБЛУЖДЕНИИ

Существуют два подхода к формализации человеческих знаний: визуальный и текстовый. С этой проблемой связано любопытное противоречие. С одной стороны, преимущество графики перед текстом для человеческого восприятия считается общепризнанным. Человеческий мозг в основном ориентирован на визуальное восприятие, и люди получают информацию при рассмотрении графических образов быстрее, чем при чтении текста.

Игорь Вельбицкий справедливо указывает:

«Текст – наиболее общая и наименее информативная в смысле наглядности и скорости восприятия форма представления информации», а чертеж – «наиболее развитая интегрированная форма представления знаний» [2].

С другой стороны, теоретическая разработка принципов визуальной формализации знаний все еще не развернута в должной мере. Причину отставания следует искать в истории науки, в частности в особенностях развития математики и логики.

В этих дисциплинах с давних пор (иногда явно, чаще неявно) предполагалось, что результаты математической и логической формализации знаний в подавляющем большинстве случаев должны иметь форму текста (а не чертежа, не изображения). Например, Стефен Клини пишет:

«Будучи формализованной, теория по своей структуре является уже не системой осмысленных предложений, а системой фраз, рассматриваемых как последовательность слов, которые, в свою очередь, являются последовательностями букв... В символическом языке символы будут обычно соответствовать целым словам, а не буквам, а последовательности символов, соответствующие фразам, будут называться «формулами»... Теория доказательств... предполагает... построение произвольного длинных последовательностей символов» [3].

Из этих рассуждений видно, что Клини (как и многие другие авторы) ставит в центр исследования проблему *текстовой* формализации и полностью упускает из виду всю совокупность проблем, связанных с *визуальной* формализацией.

Анализ литературы, посвященной данной теме, показывает, что большинство ученых исходит из неявного предположения, что научное знание – это, прежде всего, «текстовое» знание. Они полагают, что наиболее адекватной (или даже единственно возможной) формой для представления результатов научного исследования является последовательность формализованных и неформализованных фраз. То есть текст (а отнюдь не графические, визуальные образы).

Вопреки этому мнению, язык ДРАКОН опирается не на текстовую, а на визуальную формализацию знаний. Визуальный синтаксис языка ДРАКОН является не текстовым, а *формальным визуальным объектом*.

Таким образом, упомянутые выше ученые защищают ошибочную точку зрения, которую можно охарактеризовать как «принцип абсолютизации текста».

§6. ПРИНЦИП АБСОЛЮТИЗАЦИИ ТЕКСТА

Суть его можно выразить, например, в форме следующих рассуждений.

Прогресс науки обеспечивается успехами логико-математической формализации и разработкой новых научных понятий и принципов, а не усовершенствованием рисунков.

Формулы и слова выражают сущность научной мысли.

Рисунки – это всего лишь иллюстрации к научному тексту. Они облегчают понимание уже готовой, сформированной научной мысли, но не участвуют в ее формировании. Короче говоря, язык науки – это формулы и предложения, но никак не картинки.

В науке есть суть, сердцевина, от которой зависит успех научного творчества и получение новых научных результатов. Она выражается логико-математическими формализмами, научными понятиями и суждениями, выраженными в словах.

И есть вспомогательные задачи – обучение новичков, обмен информацией между учеными. Здесь-то и помогают картинки, облегчая взаимопонимание.

Кроме того, рисунки имеют необязательную, свободную и нестрогую форму, их невозможно формализовать. Поэтому формализация научного знания несовместима с использованием рисунков.

Таким образом, рисунки и чертежи есть нечто внешнее по отношению к науке. Совершенствование языка рисунков и научный прогресс – разные вещи, они не связаны между собой.

Существует ряд работ, косвенным образом доказывающих, что принцип абсолютизации текста является ошибочным и вредным [4, 5 и др.]. Сегодня все больше ученых приходит к выводу, что визуальную формализацию знаний нельзя рассматривать как нечто второстепенное для научного познания, поскольку она входит в самую ткань мысленного процесса ученого и «может опосредовать самые глубинные, творческие шаги научного познания» [4].

Вместе с тем в математической логике визуальные методы, насколько нам известно, пока еще не нашли широкого применения. Иными словами, математическая логика по сей день остается оплотом текстового мышления и текстовых методов формализации знаний. Это обстоятельство играет отрицательную роль, мешая поставить последнюю точку в доказательстве ошибочности «принципа абсолютизации текста».

Далее мы попытаемся на частном примере исчисления икон продемонстрировать принципиальную возможность визуализации, по крайней мере, некоторых разделов или, скажем аккуратнее, вопросов математической логики.

§7. ВИЗУАЛИЗАЦИЯ ПОНЯТИЙ МАТЕМАТИЧЕСКОЙ ЛОГИКИ

Нам понадобится определение двух понятий:

- *визуальный логический вывод* (для краткости – видеовывод);
- *визуальное логическое исчисление* (для краткости – видеоисчисление).

Чтобы облегчить изучение материала, используем метод «очной ставки». Поместим в левой графе таблицы 1 хорошо известное «текстовое» понятие. А в правой – определение нового, визуального понятия.

Таблица 1

Определение понятия «логический вывод»	Определение понятия «видеовывод» (визуальный логический вывод)
<p>Вывод в исчислении V есть последовательность C_1, \dots, C_n формул, такая, что для любого i формула C_i есть:</p> <ul style="list-style-type: none"> ● либо аксиома исчисления V; ● либо непосредственное следствие предыдущих формул по одному из правил вывода. <p>Формула C_n называется теоремой исчисления V, если существует вывод в V, в котором последней формулой является C_n [6].</p>	<p>Видеовывод в видеоисчислении V есть последовательность C_1, \dots, C_n видеоформул, такая, что для любого i видеоформула C_i есть:</p> <ul style="list-style-type: none"> ● либо видеоаксиома видеоисчисления V; ● либо непосредственное следствие предыдущих видеоформул по одному из правил видеовывода. <p>Видеоформула C_n называется видео-теоремой видеоисчисления V, если существует видеовывод в V, в котором последней видеоформулой является C_n.</p>

Нетрудно заметить, что новое определение (справа) почти дословно совпадает с классическим (слева). Разница состоит лишь в добавлении приставки «видео».

Развивая этот подход и опираясь на «текстовое» определение логического исчисления, можно по аналогии ввести понятие «видеоисчисление» (табл. 2).

Таблица 2

<p>Определение понятия «логическое исчисление»</p>	<p>Определение понятия «видеоисчисление» (визуальное логическое исчисление)</p>
<p>Логическое исчисление может быть представлено как формальная система в виде четверки</p> $V = \langle I, S_0, A, F \rangle$ <p>где</p> <ul style="list-style-type: none"> ● I – множество базовых элементов (букв алфавита); ● S_0 – множество синтаксических правил, на основе которых из букв строятся правильно построенные формулы; ● A – множество правильно построенных формул, элементы которого называются аксиомами; ● F – правила вывода, которые из множества A позволяют получать новые правильно построенные формулы – теоремы [7]. 	<p>Видеоисчисление может быть представлено как формальная система в виде четверки</p> $V = \langle I, S_0, A, F \rangle$ <p>где</p> <ul style="list-style-type: none"> ● I – множество икон (букв визуального алфавита); ● S_0 – множество правил визуального синтаксиса, на основе которых из икон строятся правильно построенные видеоформулы; ● A – множество правильно построенных видеоформул, элементы которого называются видеоаксиомами; ● F – правила видеовывода, которые из множества A позволяют получать новые правильно построенные видеоформулы – <i>видеотеоремы</i>. (Множество теорем обозначим через T.)

§8. ИСЧИСЛЕНИЕ ИКОН

Итак, мы определили нужные понятия визуальной математической логики. С их помощью можно построить исчисление икон.

- Множество икон I (букв визуального алфавита) задано тезисом 1 (см. главу 33) и показано на рис. 17.
- Множество S_0 правил визуального синтаксиса описано в главе 33 в тезисах 2–37.
- Множество A визуальных аксиом включает всего два элемента: заготовку-примитив и заготовку-силуэт (рис. 232). Далее будем называть их *аксиома-примитив* и *аксиома-силуэт*.

- Множество T , охватывающее все видеотеоремы исчисления V , есть не что иное как множество шампур-схем (абстрактных дракон-схем). Заметим, что множество T не включает аксиомы, так как последние содержат незаполненные критические точки и, следовательно, эквивалентны пустым операторам. Множество T распадается на две части: множество примитивов T_1 и множество силуэтов T_2 .
- Множество F правил видеовывода также делится на две части F_1 и F_2 . Множество F_1 позволяет выводиться все теоремы-примитивы, принадлежащие множеству T_1 , из единственной аксиомы-примитива. Оно содержит пять правил вывода: ввод атома, добавление варианта, пересадка лианы, боковое присоединение, удаление конца примитива. Эти правила описаны в тезисах 10, 21, 28, 30, 31, 34 главы 33.
- Множество F_2 дает возможность выводиться все теоремы-силуэты множества T_2 из единственной аксиомы-силуэта. Оно содержит восемь правил вывода: ввод атома, добавление варианта, добавление ветки, пересадка лианы, заземление лианы, боковое присоединение, удаление последней ветки, дополнительный вход. Правила вывода для силуэта описаны в тезисах 10, 21, 28–33, 35 главы 33.

На этом построение исчисления икон заканчивается.

§9. СЕМАНТИКА ШАМПУР-СХЕМ

Известно, что изучение исчислений составляет синтаксическую часть математической логики. Кроме того, последняя занимается семантическим изучением формальных языков, причем основным понятием семантики служит понятие истинности [1].

В исчислении икон семантика тривиальна. Различные видеоформулы (блок-схемы) могут быть истинными или ложными.

Видеоформула называется *истинной*, если она – либо аксиома, либо выводится из аксиом с помощью правил вывода (то есть является теоремой). И *ложной* в противном случае.

Таким образом, все правильно построенные шампур-схемы (теоремы) истинны. И наоборот, неправильно построенные схемы, не удовлетворяющие визуальным правилам языка ДРАКОН, считаются ложными.

Примеры ложных схем показаны на рис. 156, 158, 160, 162, 164.

§10. ДОКАЗАТЕЛЬСТВО ПРАВИЛЬНОСТИ АЛГОРИТМОВ

Роберт Андерсон подчеркивает: «целью многих исследований в области доказательства правильности программ является... механизация таких доказательств» [8]. Дэвид Грис указывает, что «доказательство должно опережать построение программы» [9].

Объединив оба требования, получим, что автоматическое доказательство правильности должно опережать построение алгоритма. Нетрудно убедиться, что предлагаемый метод обеспечивает частичное выполнение этого требования. В самом деле, в начале главы было показано, что любая правильно построенная шампур-схема является строго доказанной теоремой. В алгоритмах дракон-конструктора закодировано исчисление икон. Поэтому любая шампур-схема, построенная с его помощью, является истинной, то есть правильно построенной. Этот результат очень важен. Он означает, что:

Дракон-конструктор осуществляет 100%-е автоматическое доказательство правильности шампур-схем, гарантируя принципиальную невозможность ошибок визуального синтаксиса.

В начале главы мы задали смешной вопрос: если шампур-схемы – это теоремы, кто должен их доказывать? Ответ прост. Их никто не должен доказывать, так как все они раз и навсегда доказаны. Потому что работа дракон-конструктора построена как реализация исчисления икон.

А теперь добавим ложку дегтя в бочку меда. К сожалению, данный метод позволяет доказать правильность шампур-схемы и только. Это составляет лишь часть от общего объема работы, которую нужно выполнить, чтобы доказать правильность алгоритма на 100%. Здесь необходима оговорка.

Частичное доказательство правильности алгоритма с помощью дракон-конструктора осуществляется без какого-либо участия человека и достигается совершенно бесплатно, так как дополнительные затраты труда, времени и ресурсов не требуются. Так что полученный результат (безошибочное автоматическое проектирование графики дракон-схем) следует признать значительным шагом вперед.

§11. ВЫВОДЫ

1. Построено визуальное логическое исчисление, которое мы назвали «исчислением икон».
2. Данное исчисление можно рассматривать как раздел нового направления – *визуальной математической логики*.
3. Показано, что графический синтаксис языка ДРАКОН представляет собой исчисление икон.
4. Исчисление икон является теоретическим обоснованием языка ДРАКОН.
5. Исчисление икон полностью реализовано во внутренних алгоритмах работы дракон-конструктора.
6. Дракон-конструктор осуществляет 100%-е автоматическое доказательство правильности шампур-схем, гарантируя принципиальную невозможность ошибок визуального синтаксиса.

7. Безошибочное автоматическое проектирование графики дракон-схем следует признать значительным шагом вперед, повышающим производительность труда при практической работе.
8. Вторым (наряду с математическим) направлением, использованным при создании языка ДРАКОН, является научный подход к эргономизации конструкций языка. Такой подход позволяет улучшить визуальные образы языка (визуальные формы фиксации знаний), согласовав их с известными характеристиками глаза и мозга.
9. Разработка исчисления икон говорит в пользу этой идеи и служит примером, подтверждающим актуальность когнитивной эргономики.

МЕТОД АШКРОФТА-МАННЫ И АЛГОРИТМИЧЕСКАЯ СТРУКТУРА «СИЛУЭТ»

§1. МЕТОД Эдварда АШКРОФТ И Зоухара МАННА

К языку ДРАКОН ведут несколько математических тропинок. Одна из них – метод Ашкрофта-Манны [1]. С помощью этого метода любой неструктурный алгоритм можно превратить в структурный. Это чисто теоретический метод, который в реальной работе обычно не применяется.

Более того, известный специалист по надежности программ Гленфорд Майерс подчеркивает, что метод Ашкрофта-Манны «никогда не следует использовать на практике» [2]

Почему же мы о нем вспомнили? Потому, что результат, получаемый с помощью метода Ашкрофта-Манны, почти полностью совпадает с дракон-схемой «силуэт». А силуэт служит мощным средством для повышения производительности труда алгоритмистов и программистов.

Ниже мы покажем, что метод Ашкрофта-Манны можно использовать для математического обоснования языка ДРАКОН.

Наиболее удобное описание метода Ашкрофта-Манны дал Эдвард Йодан [3]. На это описание мы и будем опираться (с небольшими изменениями).

§2. НАЧАЛЬНЫЕ СВЕДЕНИЯ О МЕТОДЕ АШКРОФТА-МАННЫ

Метод Ашкрофта-Манны применим к любым алгоритмам (в частности, содержащим циклы и другие сложные конструкции).

На рис. 251 представлен пример неструктурной схемы (назовем ее *исходной*). В параграфах §§2–7 описан процесс преобразования исходной схемы в структурную схему на рис. 253. Последнюю можно назвать «схемой Ашкрофта-Манны».

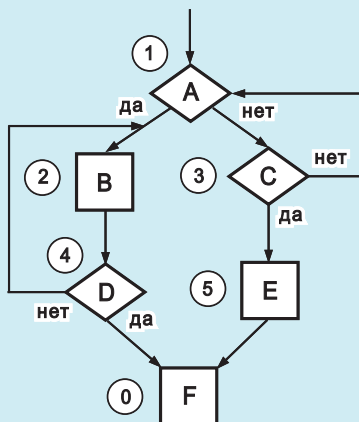
Каждому блоку неструктурной схемы приписывается номер. Заметим, что на рис. 251 это уже сделано. Способ присваивания номеров произвольный. Но обычно принимают соглашение: обозначать номером 1 первый исполняемый блок алгоритма. И номером 0 – последний исполняемый блок.

На рис. 252 исходная схема нарисована более аккуратно. Наклонные линии заменены на вертикальные и горизонтальные.

Все блоки исходной схемы делятся на две части:

- функциональные блоки (простейшим функциональным блоком служит икона «действие»);
- икона «вопрос».

В исходной схеме на рис. 251 присутствуют три функциональных блока: *B*, *E*, *F*. И три иконы вопрос: *A*, *C*, *D*.



На рис. 251–258 показано преобразование исходной неструктурной схемы в алгоритмическую структуру силуэт.

В ходе преобразования можно выделить четыре схемы:

1. Исходная неструктурная схема (рис. 251);
2. Структурная схема Ашкрофта-Манна (рис. 253);
3. Черновой силуэт (рис. 256);
4. Окончательный силуэт (рис. 258)

Шаг 1. Каждому блоку исходной схемы приписывается номер (в кружках)

Рис. 251. Исходная неструктурная схема, которую надо превратить в силуэт

§3. ЧТО ТАКОЕ БЛОК-ПРЕЕМНИК?

Каждый блок исходной схемы передает управление некоторому другому блоку. Последний называется *блоком-преемником*. Все блоки-преемники, показанные на рис. 251 и 252, перечислены в таблице.

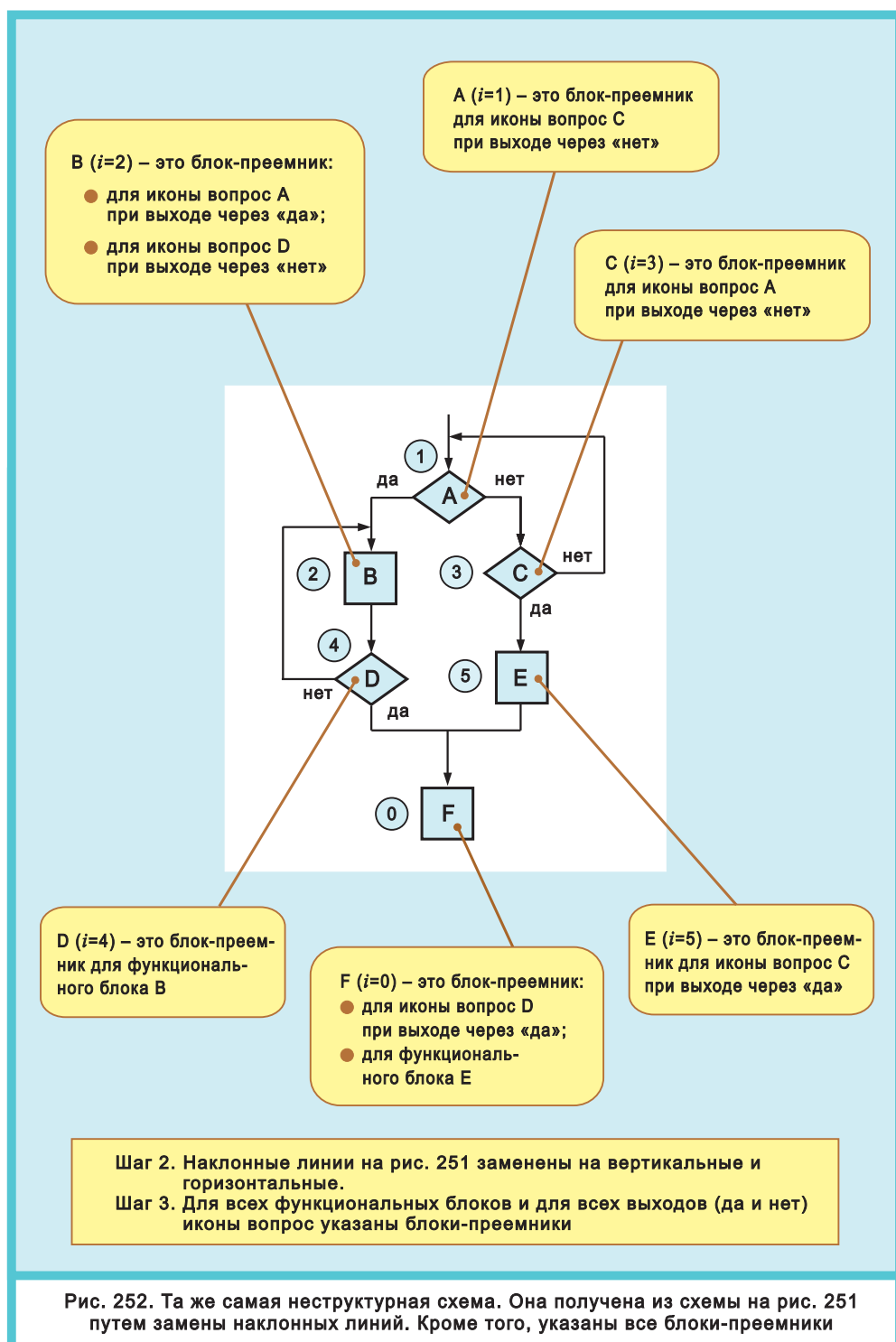


Рис. 252. Та же самая неструктурная схема. Она получена из схемы на рис. 251 путем замены наклонных линий. Кроме того, указаны все блоки-преемники

Блок, передающий управление	Блок-посылка
Икона-вопрос А при выходе через «да»	В
Икона-вопрос А при выходе через «нет»	С
Функциональный блок В	Д
Икона-вопрос С при выходе через «да»	Е
Икона-вопрос С при выходе через «нет»	А
Икона-вопрос Д при выходе через «да»	Г
Икона-вопрос Д при выходе через «нет»	В
Функциональный блок Е	Г
Функциональный блок Г (это последний блок схемы)	Последний блок Г не имеет блока-посылки

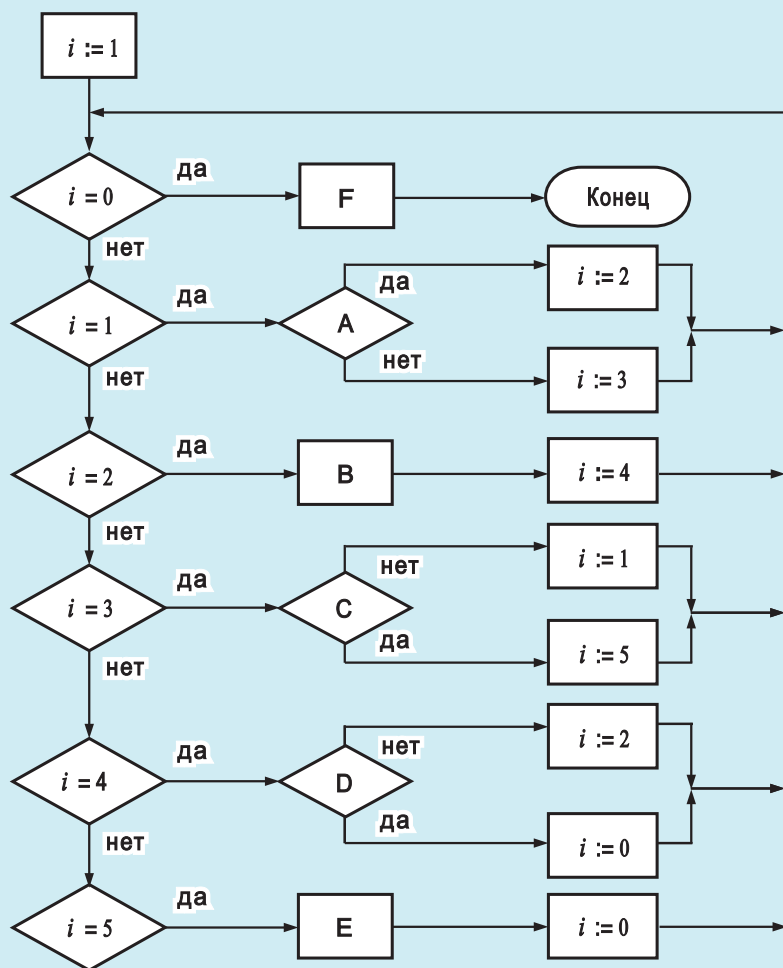
§4. НОВАЯ ПЕРЕМЕННАЯ

В алгоритм вводится новая переменная. Для нашей цели требуется переменная целого типа. Имя переменной произвольное. В нашем примере новая переменная обозначена через i (рис. 253).

§5. НОМЕРА БЛОКОВ-ПОСЫЛКИ ДЛЯ ФУНКЦИОНАЛЬНЫХ БЛОКОВ

Функциональные блоки присваивают переменной i номер блока-посылки в исходной схеме на рис. 251.

Номера блоков-посылок для функциональных блоков	
Блок-посылка	Номер блока-посылки на рис. 251
В	$i = 2$
Е	$i = 5$
Г	$i = 0$
Переменная i принимает значение, равное номеру блока-посылки на рис. 251	



Шаг 4. В схему вводится новая переменная i .

Шаг 5. В схему вводятся служебные блоки, которые присваивают переменной i целое значение, указывающее номер блока-преемника в исходной схеме на рис. 251.

Шаг 6. Начальным значением переменной i выбрано число 1. Затем мы последовательно опрашиваем значения переменной i (см. слева), исполняем соответствующее действие, повторяем опрос i и т.д. В результате неструктурная схема на рис. 251 превращается в структурную схему на рис. 253

Рис. 253. Структурная схема. Она получена из неструктурной схемы на рис. 251 с помощью метода Ашкрофта-Манна. Назовем эту схему «схемой Ашкрофта-Манна»

§6. НОМЕРА БЛОКОВ-ПРЕЕМНИКОВ ДЛЯ ИКОН «ВОПРОС» (С УЧЕТОМ «ДА» И «НЕТ»)

Логические блоки (икона «вопрос») присваивают переменной i номер блока-*преемника* в исходной схеме (рис. 251). Следует помнить, что выбор блока-преемника зависит от выхода иконы «вопрос» через «да» или «нет».

Номера блоков-преемников для икон «вопрос» (с учетом «да» и «нет»)		
Блок-преемник	Блок, передающий управление	Номер блока-преемника на рис. 251
Ф	Икона-вопрос D при выходе через «да»	$i = 0$
А	Икона-вопрос С при выходе через «нет»	$i = 1$
В	Икона-вопрос А при выходе через «да»	$i = 2$
В	Икона-вопрос D при выходе через «нет»	$i = 2$
С	Икона-вопрос А при выходе через «нет»	$i = 3$
Е	Икона-вопрос С при выходе через «да»	$i = 5$
Переменная i принимает значение, равное номеру блока-преемника на рис. 251		

§7. МЕТОД ПРЕОБРАЗОВАНИЯ НЕСТРУКТУРНЫХ АЛГОРИТМОВ В СТРУКТУРНЫЕ С ПОМОЩЬЮ ВВЕДЕНИЯ ПЕРЕМЕННОЙ СОСТОЯНИЯ

В параграфах §§2–7 описан процесс преобразования исходной схемы в структурную схему на рис. 253. В этом параграфе мы перестроим всю блок-схему на рис. 251, придав ей форму, показанную на рис. 253.

Начальным значением переменной i выбрано (в соответствии с принятым ранее соглашением) целое число 1. Затем мы последовательно опрашиваем значения переменной i , исполняем соответствующее действие, повторяем опрос i и т. д. В результате неструктурная схема на рис. 251 превращается в структурную схему на рис. 253.

Преимущество метода состоит в том, что описанное выше преобразование может быть неограниченно продолжено. Например, вместо шести блоков на рис. 251, мы могли бы рассмотреть пример с шестьюдесятью блоками, не усложняя при этом общего подхода.

Каждому блоку исходной схемы на рис. 251 соответствует определенное *состояние* алгоритма на рис. 253. В каждом *состоянии* либо дается ответ на да-нетный вопрос, либо выполняется некоторая обработка.

Переменная i указывает номер состояния. Поэтому она и называется *переменной состояния*. А описываемый метод называется методом *введения переменной состояния*.

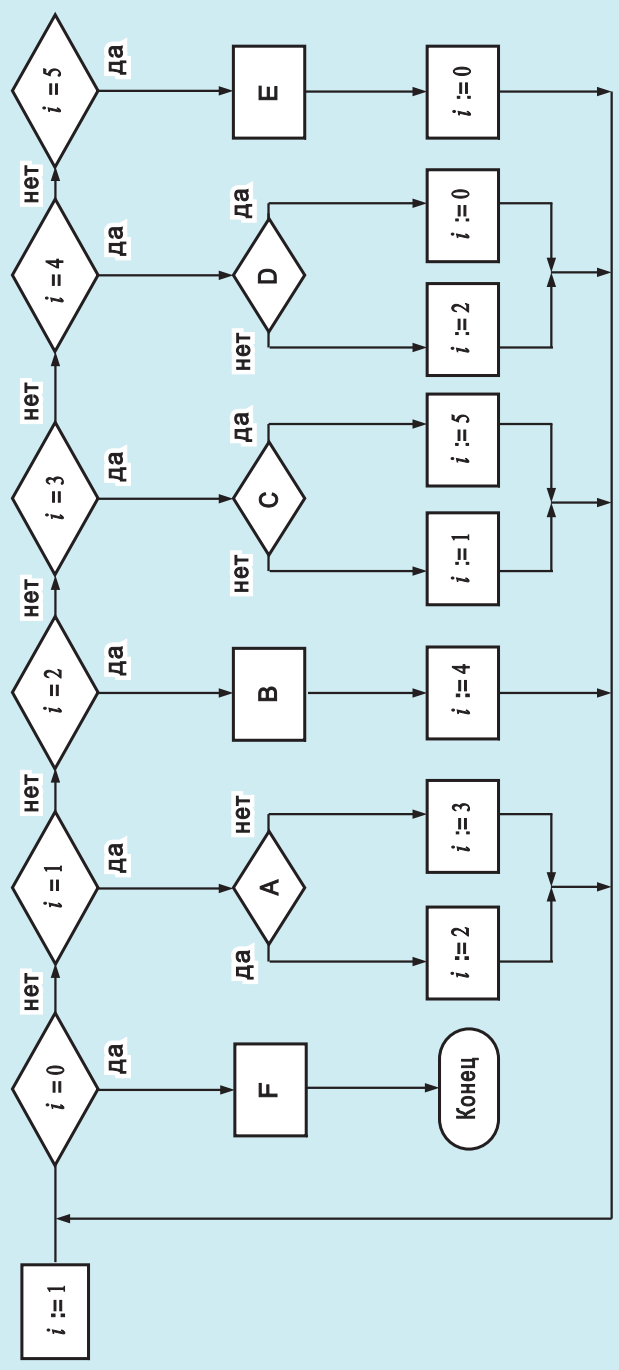
Многие специалисты (глядя на рис. 253), делают вывод, что реализация метода введения переменной состояния предполагает использование вложенных конструкций *if-then-else*. Хотя этот способ возможен, более вероятно реализация с использованием сочетания конструкции *do-while* и конструкции *case* [3].

§8. ПРЕОБРАЗОВАНИЕ СХЕМЫ АШКРОФТА-МАННЫ В ЧЕРНОВУЮ СХЕМУ «СИЛУЭТ»

Итак, на рис. 253 получена структурная схема Ашкрофта-Манн. Следующий этап – преобразование схемы Ашкрофта-Манн в черновую схему силуэт на рис. 256. Это преобразование выполняется за десять шагов.

- Шаг 1.* Схема на рис. 253 поворачивается на 90° против часовой стрелки.
- Шаг 2.* Схема отражается сверху вниз. (Результат показан на рис. 254).
- Шаг 3.* Вертикаль, содержащая икону конец, перемещается в крайнюю правую позицию.
- Шаг 4.* Каждая икона «вопрос» (в верхнем ряду) устанавливается на свой шампур.
- Шаг 5.* Удаляются три Т-образные линии в нижней части схемы, которые заменяются вертикалями. (Результат показан на рис. 255).
- Шаг 6.* Верхние фигуры на рис. 255 превращаются в иконы «имя ветки».
- Шаг 7.* Нижние фигуры на рис. 255 превращаются в иконы «адрес».
- Шаг 8.* Внутри икон «имя ветки» записываются метки, обозначающие названия веток.
- Шаг 9.* Внутри икон «адрес» записывается имя ветки, куда происходит передача управления в соответствии с логикой работы алгоритма.
- Шаг 10.* Имя переменной состояния удаляется.

После выполнения 10 шагов получаем черновую схему силуэт, показанную на рис. 256.



Шаг 1. Схема на рис. 253 поворачивается на 90° против часовой стрелки.
Шаг 2. Схема отражается сверху вниз. (Результат показан на рис. 254).

Рис. 254. Данная схема получена из схемы Ашкрфта-Манна на рис. 253 в результате поворота схемы на 90° и отражения сверху вниз. Наша цель – преобразовать схему Ашкрфта-Манна в схему силуэт

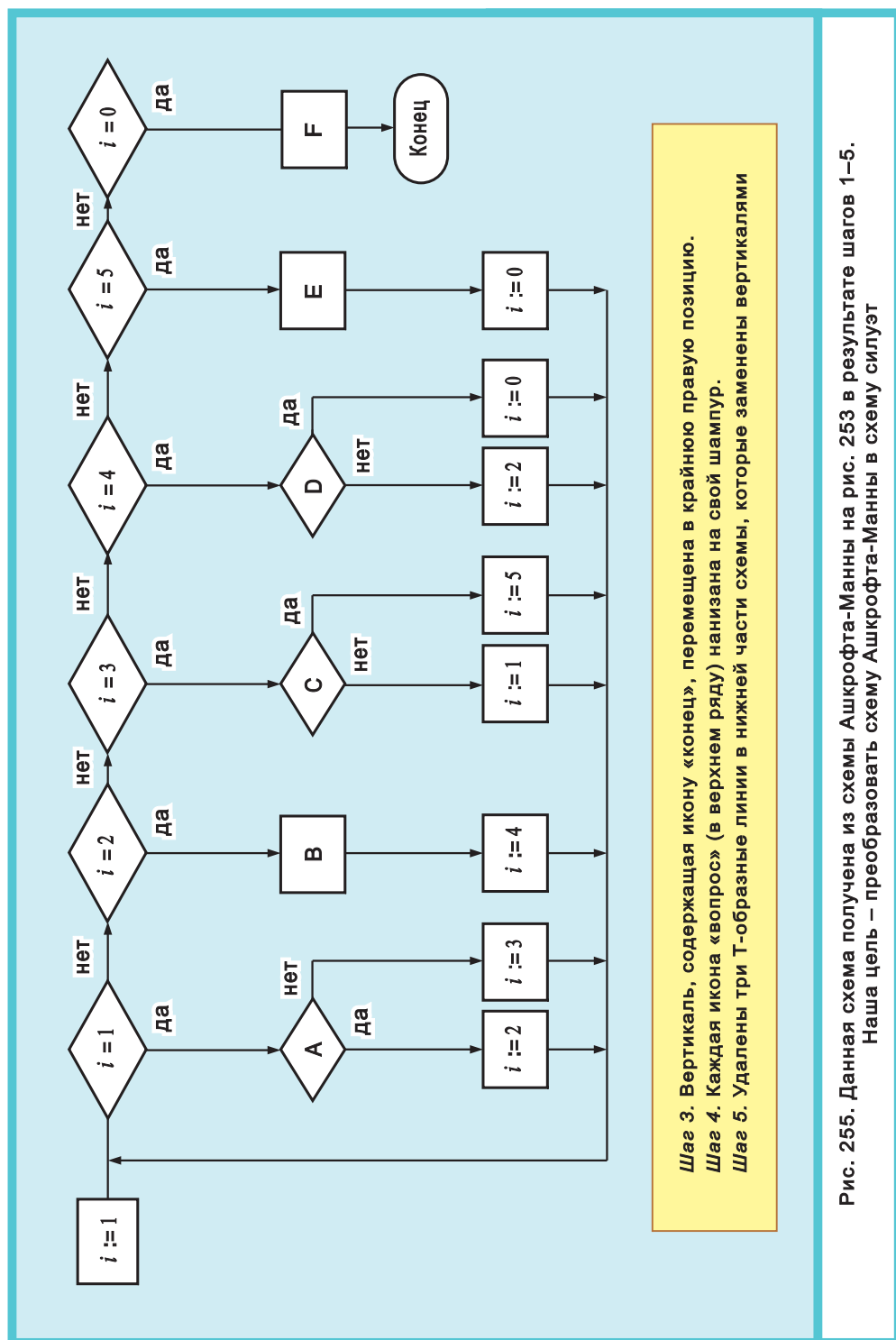


Рис. 255. Данная схема получена из схемы Ашкрофта-Манна на рис. 253 в результате шагов 1–5.
 Наша цель – преобразовать схему Ашкрофта-Манна в схему силуэт

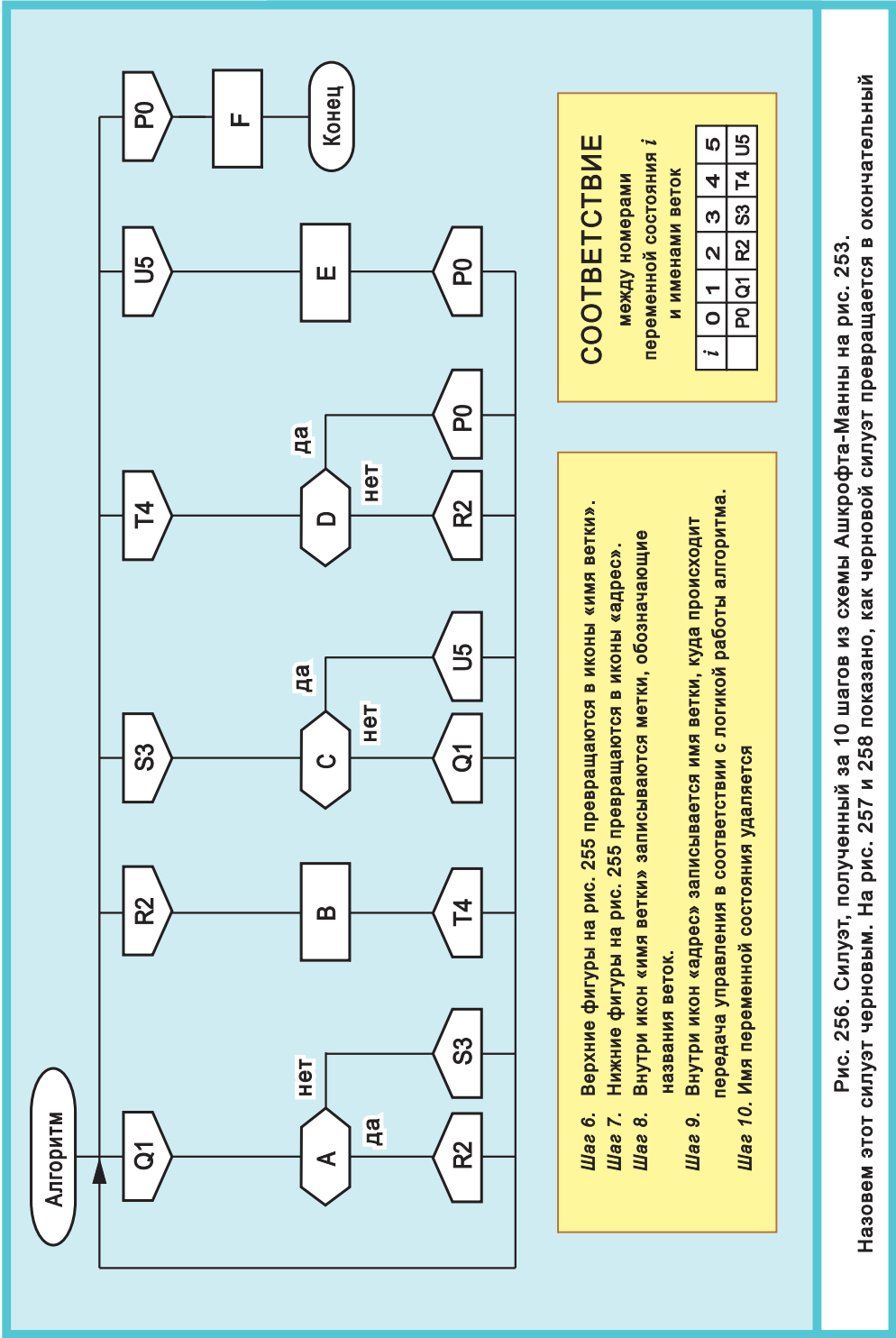


Рис. 256. Силуэт, полученный за 10 шагов из схемы Ашкрофта-Манна на рис. 253. Назовем этот силуэт черновым. На рис. 257 и 258 показано, как черновой силуэт превращается в окончательный

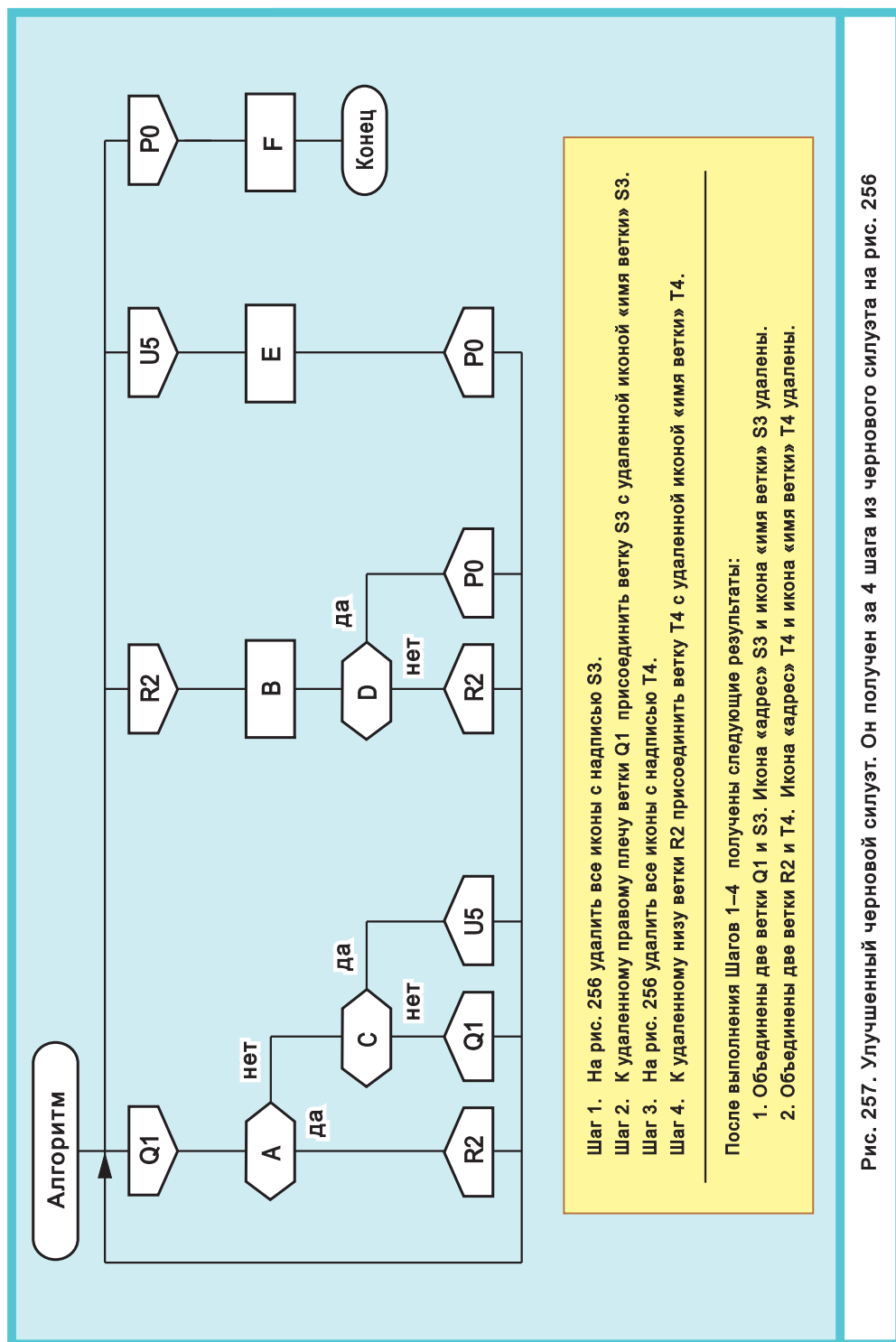
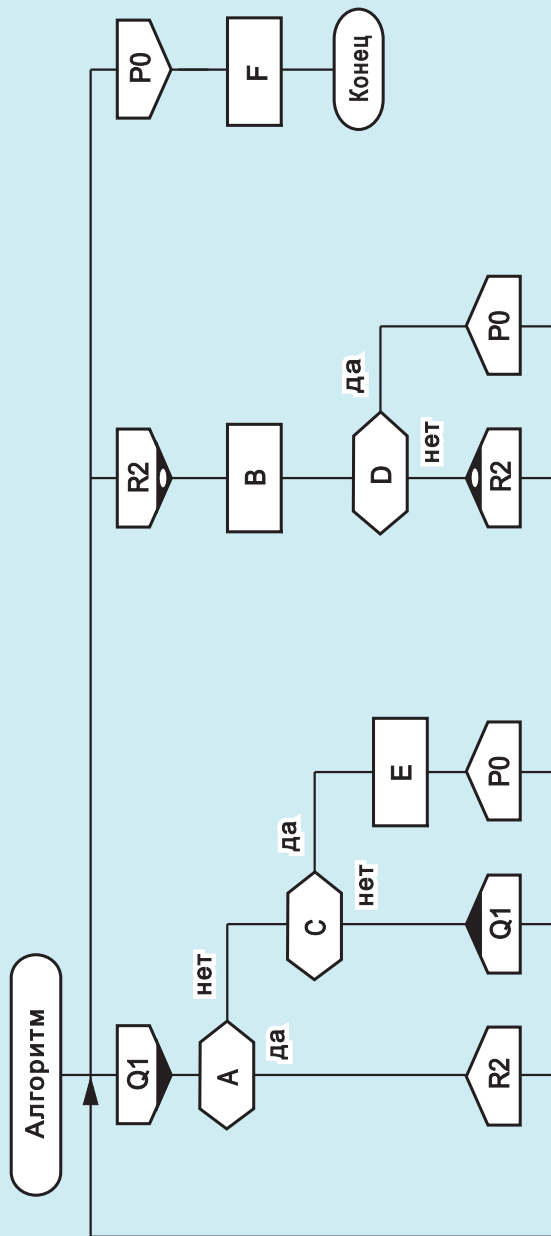


Рис. 257. Улучшенный черновой силуэт. Он получен за 4 шага из чернового силуэта на рис. 256



Шаг 5. На рис. 257 удалить все иконы с надписью U5.
Шаг 6. К удаленному правому плечу ветки Q1 присоединить ветку U5 с удаленной иконой «имя ветки» U5.
Шаг 7. Обозначить черными треугольниками веточные циклы Q1 и R2.

После выполнения Шагов 5–7 получены следующие результаты:
1. Объединены две ветки Q1 и U5. Икона «адрес» U5 и икона «имя ветки» U5 удалены.
2. Веточные циклы Q1 и R2 выделены и обозначены черными треугольниками.
3. Получена дракон-схема силуэт в окончательном виде.

Рис. 258. Окончательный силуэт. Он получен за 7 шагов из черногого силуэта на рис. 256

§9. ПРЕОБРАЗОВАНИЕ ЧЕРНОВОЙ СХЕМЫ В ОКОНЧАТЕЛЬНУЮ СХЕМУ СИЛУЭТ

Опишем преобразование чернового силуэта на рис. 256 в окончательный силуэт на рис. 258.

1. Объединяем ветки Q1 и S3.
2. Объединяем ветки R2 и T4.
3. Объединяем ветки Q1 и U5.
4. Веточные циклы выделяем с помощью черных треугольников.

Сравним рис. 256 и 258. Из первой схемы удалены три ветки. Вместо шести веток осталось только три. В итоге схема на рис. 258 стала выразительнее и проще.

Проведенные рассуждения позволяют сделать

Вывод. Метод Ашкрофта-Манн можно рассматривать как математическое обоснование основной алгоритмической структуры языка ДРАКОН – структуры «силуэт».

§10. ВЫВОДЫ

1. В данной главе показано преобразование исходного неструктурного алгоритма в алгоритм «силуэт».
2. На первом этапе выполняется преобразование исходного неструктурного алгоритма в структурный алгоритм Ашкрофта-Манн с помощью введения переменной состояния.
3. На втором этапе производится преобразование алгоритма Ашкрофта-Манн в черновой алгоритм «силуэт».
4. На третьем этапе выполняется преобразование чернового алгоритма «силуэт» в окончательный алгоритм «силуэт».
5. Метод Ашкрофта-Манн является математическим обоснованием алгоритмической структуры «силуэт».

ВИЗУАЛЬНЫЙ СТРУКТУРНЫЙ ПОДХОД К АЛГОРИТМАМ И ПРОГРАММАМ (ШАМПУР-МЕТОД)

§1. ВВЕДЕНИЕ

Напомним, что книга посвящена алгоритмам. Вопросы программирования в ней не рассматриваются. Данная глава является исключением. Глава посвящена исследованию и развитию фундаментального понятия «структурное программирование». Мы будем использовать это понятие в первую очередь применительно к алгоритмам. И лишь иногда – к программам.

§2. ТЕРМИНОЛОГИЯ

Введем понятие «визуальный структурный подход к алгоритмам и программам». Определим его как набор правил, совпадающий с визуальным синтаксисом языка ДРАКОН. В концентрированном виде эти правила изложены в главе 33. Данный подход предназначен для решения двух задач:

- создания наглядных, понятных и удобных для человеческого восприятия алгоритмов и программ;
- автоматического доказательства правильности шампур-схем (то есть графической части дракон-схем).

Наряду с термином «визуальный структурный подход к алгоритмам и программам» мы будем для краткости (в качестве синонима) использовать термин «шампур-метод».

Можно сказать, что данная книга – это подробное описание шампур-метода.

§3. ДВА СТРУКТУРНЫХ ПОДХОДА

Как показано в работе [1], следует различать два структурных подхода:

- одномерный (текстовый) подход;
- двумерный (визуальный) подход.

Первый подход создан основоположниками компьютерной науки и уточнен их последователями. Одномерный структурный подход известен под названием «структурное программирование». Он широко используется в мировой практике программирования.

Второй подход разработан автором, изложен в этой книге и других работах (см. раздел «Основная литература по языку ДРАКОН»).

§4. ПОСТАНОВКА ПРОБЛЕМЫ

Размышляя над проблемой, автор пришел к следующим предварительным выводам или, лучше сказать, предположениям.

- Несмотря на наличие целого ряда общих признаков, текстовое и визуальное структурное программирование – существенно разные вещи.
- Традиционный (текстовый) структурный подход является, по-видимому, наилучшим решением соответствующей задачи для традиционного (текстового) структурного программирования
- Для визуального подхода подобное утверждение неправомерно. Можно, конечно, тупо перенести правила текстового структурного программирования на визуальный случай. Но это не будет хорошим решением.
- Чтобы разработать эффективный метод структуризации для визуального варианта, необходимо, взяв за основу правила текстового структурного программирования, значительно модифицировать их.
- Принципы структуризации, положенные в основу визуального синтаксиса языка ДРАКОН, являются искомым решением.

В данной главе сделана попытка обосновать заявленные выводы.

§5. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Понятие «структурное программирование» во многом связано с именем Эдсгера Дейкстры. Данное понятие охватывает две темы:

- доказательство правильности программ (*program correctness proof*);
- метод структуризации программ.

Ниже рассмотрены обе темы. Первая тема изложена в §§6 и 7. Вторая описана, начиная с §8.

§6. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ КАК ДОКАЗАТЕЛЬСТВО ПРАВИЛЬНОСТИ ПРОГРАММ

В середине XX века появились компьютеры и компьютерные программы. В ту раннюю эпоху теория программирования еще не существовала. Разработка программ велась методом проб и ошибок. И выглядела примерно так: «написать код программы – проверить код на компьютере (протестировать) – найти ошибки – исправить код – снова протестировать – снова найти ошибки – снова исправить и т. д.».

Эдсгер Дейкстра осудил подобную практику и указал, что господствующий в компьютерной индустрии подход к программированию как к процессу достижения результата методом проб и ошибок порочен, поскольку стимулирует программистов не думать над задачей, а сразу писать код.

Чтобы исправить положение, Дейкстра предложил использовать математический подход к программированию. Такой подход, в частности, подразумевает формальное доказательство правильности выбранного алгоритма и последующую реализацию алгоритма в виде структурной программы, правильность которой должна быть формально доказана.

В «Заметках по структурному программированию» Дейкстра пишет:

«... необходимость продуманной структурной организации программы представляется как следствие требования о доказуемости правильности программы» [2, с. 52].

В толковом словаре читаем:

«Основным назначением общего метода структурного программирования, разработанного в значительной степени Э. Дейкстрой, является обеспечение доказательства правильности программы... (*program correctness proof*)» [3, с. 463].

Таким образом, согласно Дейкстре, структурное программирование подразумевает *доказательство правильности программ*.

§7. ЧЕМ РАЗЛИЧАЮТСЯ НАШ ПОДХОД К ДОКАЗАТЕЛЬСТВУ ПРАВИЛЬНОСТИ И ПОДХОД ДЕЙКСТРЫ?

Мы используем идею Дейкстры о доказательстве правильности программ, развивая ее и перенося на абстрактные дракон-схемы (шампур-схемы).

В главе 34 мы разработали логическое исчисление икон, которое определяет порядок работы дракон-конструктора. Благодаря этому дракон-конструктор осуществляет автоматическое доказательство правильности шампур-схем.

В чем отличие от подхода Дейкстры? Во-первых, мы говорим не о полном, а о частичном доказательстве правильности.

Во-вторых, Дейкстра предлагает программистам доказывать правильность программ *не автоматически, а вручную* с помощью разработанных им математических методов (преобразование предикатов и др.) [4].

Мы же предлагаем доказывать правильность *не вручную, а автоматически*.

В главе 34 мы доказали, что исчисление икон истинно. Отсюда вытекает, что если дракон-конструктор спроектирован правильно (то есть, если он точно реализует исчисление икон), то любая дракон-схема, созданная пользователем с помощью дракон-конструктора, будет гарантированно иметь правильную графическую часть.

Таким образом, *однократное* доказательство истинности исчисления икон влечет за собой тот факт, что десятки и сотни тысяч дракон-схем, (созданных с помощью дракон-конструктора) будут иметь автоматически доказанную правильную графику. Нет никакой необходимости *многократно* повторять доказательство, то есть доказывать правильность графической части для *каждой* из десятков или сотен тысяч дракон-схем.

Частичное доказательство правильности алгоритма с помощью дракон-конструктора осуществляется без какого-либо участия человека и достигается совершенно бесплатно, так как дополнительные затраты труда, времени и ресурсов не требуются. Так что полученный результат (безошибочное автоматическое проектирование графики алгоритмов) следует признать заметным шагом вперед.

§8. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ КАК МЕТОД СТРУКТУРИЗАЦИИ ПРОГРАММ

Основные положения метода общеизвестны:

- Алгоритм (программа) строится из частей (базовых управляющих структур).
- Каждая базовая управляющая структура имеет один вход и один выход.

- Для передачи управления используются три базовые управляющие структуры: последовательность, выбор, цикл.

§9. РАЗВИТИЕ КОНЦЕПЦИИ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Работы основоположников структурного программирования послужили исходной идеей для разработки шампур-метода и языка ДРАКОН. Предлагаемый нами двумерный структурный подход – это непосредственное развитие классического одномерного структурного программирования.

Почему возникла необходимость в таком развитии, то есть в существенной доработке классических идей?

- Идеи структурного программирования разрабатывались, когда компьютерная графика фактически еще не существовала и основным инструментом программиста был одномерный (линейный или ступенчатый) текст.
- Создатели структурного программирования недооценили потенциальные возможности блок-схем (*flow-charts*). И не сумели дать блок-схемам строгое математическое обоснование, пригодное для трансляции блок-схем в объектные коды.
- Авторы структурного программирования не были знакомы с когнитивной эргономикой и не смогли превратить блок-схемы одновременно и в эргономичный, и в математический объект. Впрочем, они и не ставили перед собой такой задачи.

§10. ПЕРЕХОД ОТ ОДНОМЕРНОГО ТЕКСТА К ДВУМЕРНОЙ ГРАФИКЕ РОЖДАЕТ НОВЫЕ ВОЗМОЖНОСТИ

Текстовые структурные управляющие конструкции сыграли позитивную роль. Они позволили улучшить структуру и удобочитаемость программ, уменьшить число ошибок и т. д. В сочетании с другими средствами они помогли, как иногда говорят, «превратить программирование из шаманства в науку».

Но у медали есть и другая сторона. Слабое место текста заключается в недостатке выразительных средств. Следствием являются ограничения и запреты. Эти ограничения и запреты вытекают из природы текста, из природы текстового структурного программирования.

Недостаток выразительных средств, проявляющийся через ограничения и запреты, тормозит повышение производительности труда алгоритмистов и программистов.

В рамках одномерного текста устранить эти ограничения и запреты невозможно. Но это вовсе не значит, что ситуацию в принципе нельзя улучшить. Чтобы добиться улучшения, надо перейти от текста к графике. Точнее, перейти от одномерного текстового структурного программирования к двумерному визуальному структурному программированию.

Задача состоит в том, чтобы значительно уменьшить число запретов и ограничений. То есть предоставить алгоритмистам и программистам дополнительную свободу действий.

Эту задачу и решает язык ДРАКОН. Следуя по мудрому пути, начертанному основоположниками структуризации, визуальный язык ДРАКОН устраняет ненужные барьеры и препятствия. И позволяет совершить прыжок из царства необходимости в царство свободы.

Каким образом? Благодаря замене одномерного (текстового) структурного подхода на двумерный (визуальный) структурный подход. Последний, разумеется, также является системой правил. Но в «двумерной» системе правил значительная часть запретов снимается. То, что раньше было нельзя, теперь можно.

Многие запреты (неизбежные при текстовом структурном программировании) «перегибают палку». Они противоречат здравому смыслу и затрудняют понимание алгоритмов и программ.

§11. УПРАВЛЯЮЩИЕ СТРУКТУРЫ, КОТОРЫЕ ЗАПРЕЩЕНЫ В ТЕКСТОВОМ СТРУКТУРНОМ ПРОГРАММИРОВАНИИ И РАЗРЕШЕНЫ В ВИЗУАЛЬНОМ

В книге приведено большое количество примеров таких структур. Здесь нет необходимости их повторять. Поэтому мы приведем только один пример. Рассмотрим схему на рис. 260.

В классическом структурном программировании управляющая структура на рис. 260 и *многие другие* запрещены. Они считаются неструктурными. Но такие алгоритмы часто встречаются на практике.

Что отсюда следует?

Наш ответ таков: текстовые управляющие структуры играли важную роль в эпоху текстового программирования. В ту пору они были единственно возможным решением. Но сегодня, в эпоху компьютерной графики и визуальной алгоритмизации (визуального программирования) подобные ограничения и запреты следует признать устаревшими. Потому что во многих случаях (хотя и не всегда) они искажают нормальный ход человеческой мысли.

Недопустимо запрещать правильный процесс мышления. Его надо разрешить. И ДРАКОН разрешает.

Подчеркнем еще раз. Наши предложения стали возможными благодаря предыдущим достижениям. Благодаря усилиям выдающихся ученых, создавших метод структурного программирования. Они опираются на фундамент, разработанный в эпоху текстового программирования. Поэтому наши предложения нельзя считать полностью новыми. Они лишь развивают разработанные классиками идеи и снимают ограничения, которые в ту эпоху (которая, кстати, еще не закончилась) были неизбежными.

§12. НОВАЯ ФИЛОСОФИЯ ПРОЦЕДУРНОГО ПРОГРАММИРОВАНИЯ

В рамках философии языка ДРАКОН ключевые слова управляющих конструкций становятся ненужными. Они рассматриваются как лишние и даже вредные. В самом деле, глядя на дракон-схему, нельзя обнаружить эти ключевые слова даже под микроскопом. Почему? Потому что в языке ДРАКОН применяется совершенно другой понятийный аппарат (атомы, валентные точки и т. д.) – см. главы 32 и 33.

Смена понятийного аппарата – это переход к новому пониманию глубинной сущности вещей. То есть к новому мировоззрению в области императивного, процедурного программирования.

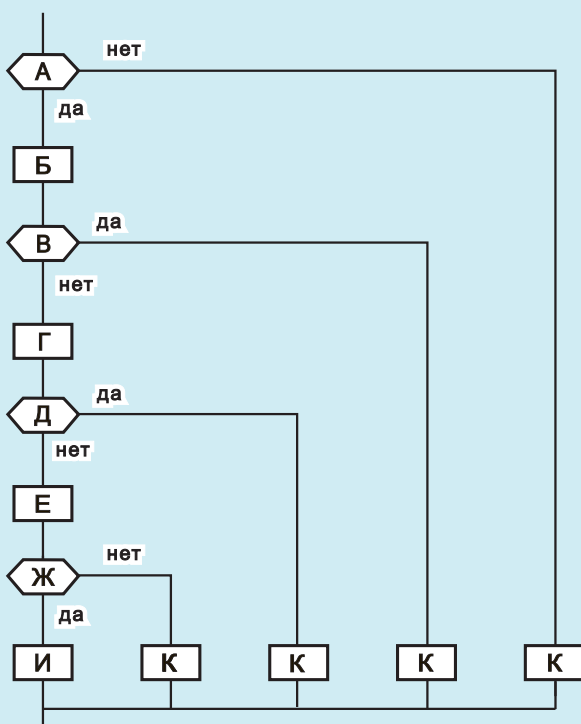
ДРАКОН – это не просто новый язык (новое семейство языков). Это новый взгляд на процедурное программирование. Если угодно – новое мировоззрение.

§13. ЗАМЕНИТЕЛИ GOTO

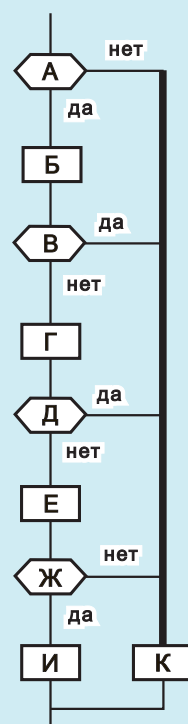
Согласно классической теореме Бома и Джакопини, всякий реальный алгоритм (программа) может быть построена из функциональных блоков (операций) и двух конструкций: цикла и дихотомического выбора (развилки) [5]. Эдсгер Дейкстра обогатил и усилил эту идею, предложив отказаться от оператора безусловного перехода *goto* и ограничиться тремя управляющими конструкциями: последовательность, выбор, цикл [2, с. 25–28].

Дональд Кнут подверг критике тезис Дейкстры о полном исключении *goto*, продемонстрировав случаи, где *goto* полезен [6]. В итоге возникла плодотворная дискуссия, в ходе которой выявились четыре варианта мнений (табл. 1).

НЕПРАВИЛЬНО



ПРАВИЛЬНО



Равносильное преобразование «вертикальное объединение» улучшает эргономичность дракон-схемы

Рис. 259. Плохая схема.

В ней слишком много вертикалей (пять) и одинаковых икон (четыре). Кроме того, есть три лишних излома, а три горизонтали и вертикали неоправданно длинные

Рис. 260. Хорошая схема.

Число вертикалей, икон и изломов удалось значительно сократить

Таблица 1

Позиция участников дискуссии	Используются три структурные конструкции?	Используются заменители <i>goto</i> ?	Используются <i>goto</i> ?
Вариант 1	Да	Нет	Нет
Вариант 2	Да	Нет	Да
Вариант 3	Да	Да	Да
Вариант 4	Да	Да	Нет

Вариант 1 описывает ортодоксальную позицию Дейкстры, согласно которой оператор *goto* имеет «гибельные последствия» и поэтому «должен быть исключен из всех языков программирования высокого уровня». Исходя из этого, были разработаны языки без *goto*: Джава, Питон, Tcl, Модула-2, Оберон и др.

Вариант 2 отражает мнение ранних критиков Дейкстры, позиция которых выражается словами:

«использование оператора *goto* может оказаться уместным в лучших структурированных программах» [7];

«всегда были примеры программ, которые не содержат *goto* и аккуратно расположены лесенкой в соответствии с уровнем вложенности операторов, но совершенно непонятны, и были другие программы, содержащие *goto*, и все же совершенно понятные» [8, с. 134].

Нужно «избегать использования *goto* всюду, где это возможно, но не ценой ясности программы» [8, с. 137–138].

Как известно, полемика по *goto* «растревожила осиное гнездо» и всколыхнула «весь программистский мир». Варианты 1 и 2 выражают крайние позиции участников дискуссии, между которыми, как казалось вначале, компромисс невозможен. Однако ситуация изменилась с изобретением и широким распространением *заместителей goto*, примерами которых являются:

- в языке Си – операторы *break*, *continue*, *return* и функция *exit ()*;
- в языке Модула-2 – операторы *RETURN*, *EXIT*, процедура *HALT* и т. д.

Заместители – особый инструмент, который существенно отличается как от трех структурных управляющих конструкций, так и от *goto*. Они обладают двумя важными преимуществами по сравнению с *goto*:

- 1) не требуя меток, заместители исключают ошибки, вызванные путаницей с метками;

- 2) каждый заменитель может передать управление не куда угодно (как *goto*), а в одну-единственную точку, однозначно определяемую ключевым словом заменителя и его местом в тексте. В результате множество точек, куда разрешен переход, сокращается на порядок, что также предотвращает ошибки.

Вариант 3 описывает языки Си, Дельфи, Ада, и др., где имеются заменители и на всякий случай сохраняется *goto*.

Вариант 4 соответствует языкам Модуль-2, Джава, где также есть заменители, однако *goto* исключен. Следует подчеркнуть, что при наличии заменителей сфера применения *goto* резко сужается. Так что удельный вес ошибок, связанных с его применением, заметно уменьшается. Поэтому вопрос об исключении *goto*, по-видимому, теряет прежнюю остроту.

Идея структуризации оказала большое влияние на разработку алгоритмов и программ. Практически во все процедурные языки (точнее, во все процедурные фрагменты языков) были введены структурные конструкции цикла и ветвления, а также различные заменители *goto*.

§14. ЧЕТЫРЕ ПРИНЦИПА СТРУКТУРИЗАЦИИ БЛОК-СХЕМ, ПРЕДЛОЖЕННЫЕ Э. ДЕЙКСТРОЙ

Попробуем еще раз заглянуть в темные переулки истории и внимательно перечитаем классический труд Дейкстры «Заметки по структурному программированию» [2]. К немалому удивлению, мы обнаружим, что основной тезис о структурных управляющих конструкциях излагается с прямой апелляцией к *визуальному языку блок-схем*.

Непосредственный анализ первоисточника со всей очевидностью подтверждает простую мысль. Дейкстрианская «структурная революция» началась с того, что Дейкстра, используя блок-схемы как инструмент анализа структуры программ, предложил наряду с другими важными идеями четыре принципа структуризации блок-схем! К сожалению, в дальнейшем указанные принципы были преданы забвению или получили иное, по нашему мнению, слишком вольное толкование.

Эти принципы таковы.

1. *Принцип ограничения топологии блок-схем*. Структурный подход должен приводить

«к ограничению топологии блок-схем по сравнению с различными блок-схемами, которые могут быть получены, если разрешить проведение стрелок из любого блока в любой другой блок. Отказавшись от большого разнообразия блок-схем и ограничившись ...тремя типами операторов управления, мы следуем тем самым некоей последовательностной дисциплине» [2, с. 28].

2. *Принцип вертикальной ориентации входов и выходов блок-схемы.* Имея в виду шесть типовых блок-схем (*if-do, if-then-else, case-of, while-do, repeat-until*, а также «действие»), Дейкстра пишет:

«Общее свойство всех этих блок-схем состоит в том, что у каждой из них один вход вверху и один выход внизу» [2, с. 27].

3. *Принцип единой вертикали.* Вход и выход каждой типовой блок-схемы должны лежать на одной вертикали.
4. *Принцип нанизывания типовых блок-схем на единую вертикаль.* При последовательном соединении типовые блок-схемы следует соединять, не допуская изломов соединительных линий, чтобы выход верхней и вход нижней блок-схемы лежали на одной вертикали.

Хотя Дейкстра не дает словесной формулировки третьего и четвертого принципов, они однозначно вытекают из имеющихся в его работе иллюстраций [2, с. 25–28]. Чтобы у читателя не осталось сомнений, мы приводим точные копии подлинных рисунков Дейкстры (рис. 261, средняя и левая графа).

Таким образом, можно со всей определенностью утверждать, что две идеи (текстовый и визуальный структурный подход), подобно близнецам, появились на божий свет одновременно. Однако этих близнецов ожидала разная судьба – судьба принца и нищего.

§15. ПОЧЕМУ НАУЧНОЕ СООБЩЕСТВО НЕ ПРИНЯЛО ВИДЕОСТРУКТУРНУЮ КОНЦЕПЦИЮ Э. ДЕЙКСТРЫ?

Далее события развивались довольно загадочным образом, поскольку вокруг видеоструктурной¹ концепции Дейкстры образовался многолетний заговор молчания.

Неприятность в том, что изложенные выше рекомендации Дейкстры не были приняты во внимание разработчиками национальных и международных стандартов на блок-схемы (ГОСТ 19.701–90, стандарт *ISO 5807–85* и т. д.). В итоге метод стандартизации, единственный метод, который мог бы улучшить массовую практику вычерчивания блок-схем, не был использован. Вследствие этого идея Дейкстры оказалась наглухо изолированной от реальных блок-схем, используемых в реальной практике разработки. В чем причина этой более чем странной ситуации?

Здесь уместно сделать отступление. Американский историк и философ Томас Кун в книге «Структура научных революций» говорит о том, что в истории науки время от времени появляются особые периоды, когда выдвигаются «несоизмеримые» с прежними взглядами научные идеи. Последние он называет *парадигмами* [9].

¹ В дальнейшем мы будем нередко использовать приставку «видео», трактуя ее как «относящийся к визуальному программированию».

Структурные операторы Дейкстры	Структурные блок-схемы Дейкстры	Дракон-схемы
if ? do S1		
if ? then S1 else S2		
case i of (S1;S2;...Sn)		
while ? do S		
repeat S until ?		

Рис. 261. Структурные блок-схемы Дейкстры и эквивалентные им визуальные операторы языка ДРАКОН

История науки – история смены парадигм. Разные парадигмы – это разные образцы мышления ученых. Поэтому сторонникам старой парадигмы зачастую бывает сложно или даже невозможно понять сторонников новой парадигмы (новой системы идей), которая приходит на смену устоявшимся стереотипам научного мышления.

По нашему мнению, одномерный (текстовый) и двумерный (визуальный) подход – это две парадигмы. Причем нынешний этап развития программирования есть болезненный процесс ломки прежних взглядов, в ходе которого прежняя *текстовая парадигма* постепенно уступает место новой *визуальной парадигме*. При этом – в соответствии с теорией Куна – многие, хотя и не все, видные представители прежней, отживающей парадигмы проявляют своеобразную слепоту при восприятии новой парадигмы, которая в ходе неустанной борьбы идей в конечном итоге утверждает свое господство.

Этот небольшой экскурс в область истории и методологии науки позволяет лучше понять причины поразительного невнимания научного сообщества к изложенным Дейкстрой принципам структуризации блок-схем.

Начнем по порядку. Формальная блок-схема – это двумерный чертеж. Следовательно, она является инструментом визуального программирования. Отсюда следует, что предложенные Дейкстрой принципы структуризации блок-схем есть не что иное, как исторически первая попытка сформулировать основные (пусть далеко не полные и, возможно, нуждающиеся в улучшении) принципы визуального структурного программирования.

Однако в 1972 году, в момент публикации работы Дейкстры [2], визуальное программирование как термин, понятие и концепция фактически еще не существовало. Поэтому, что вполне естественно, суть концепции Дейкстры была воспринята сквозь призму господствовавших тогда догматов текстового программирования со всеми вытекающими последствиями.

Так родилась приписываемая Дейкстре и по праву принадлежащая ему концепция текстового структурного программирования. При этом (что также вполне естественно) в означенное время никто не обратил внимания на тот чрезвычайно важный для нашего исследования факт, что исходная формулировка концепции Дейкстры имела явно выраженную визуальную компоненту. Она представляла собой метод структуризации блок-схем, то есть была описана в терминах видеоструктурного программирования.

Подобное невнимание привело к тому, что авторы стандартов на блок-схемы посчитали, что данная идея их не касается, ибо относится исключительно к тексту программ. Они дружно проигнорировали или, скажем мягче, упустили из виду визуальную компоненту структурного метода Дейкстры.

Справедливости ради добавим, что и сам отец-основатель (Э. Дейкстра), обычно весьма настойчивый в продвижении и популяризации своих

идей, отнесся к своему видеоструктурному детищу с удивительным безразличием. И ни разу не выступил с предложением о закреплении структурной идеи в стандартах на блок-схемы.

§16. ПАРАДОКС СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Мы подошли к наиболее интригующему пункту в истории структурного подхода. Чтобы выявить главное звено проблемы, зададим вопрос. Являются ли блок-схемы и структурное программирование взаимно исключающими, несовместимыми решениями? В литературе по этому вопросу наблюдается редкое единодушие. Да, они несовместимы. Вот несколько отзывов. По мнению многих авторов, блок-схемы:

«не согласуются со структурным программированием, поскольку в значительной степени ориентированы на использование *goto*» [8, с. 150].

Они «затемняют особенности программ, созданных по правилам структурного программирования» [3, с. 193].

«С появлением языков, отвечающих принципам структурного программирования, ... блок-схемы стали отмирать» [10].

Парадокс в том, что приведенные высказывания основываются на недоразумении. Чтобы логический дефект стал очевидным, сопоставим две цитаты по методу «очной ставки» (табл. 2). Сравнивая мнение современных авторов с позицией Дейкстры, нетрудно убедиться, что описываемый критиками изъян действительно имеет место.

Но! Лишь в том случае, если правила вычерчивания блок-схем игнорируют предложенный Дейкстрой принцип ограничения топологии блок-схем. И наоборот, соблюдение указанного принципа сразу же ликвидирует недостаток.

Таблица 2

Мнение критиков, убежденных в невозможности структуризации блок-схем	Предложение Э. Дейкстры о структуризации блок-схем
«Основной недостаток блок-схем заключается в том, что они не приучают к аккуратности при разработке алгоритма. Ромб можно поставить в любом месте блок-схемы, а от него повести выходы на какие угодно участки. Так можно быстро превратить программу в запутанный лабиринт, разобраться в котором через некоторое время не сможет даже сам ее автор» [10].	Структуризация блок-схем с неизбежностью приводит «к ограничению топологии блок-схем по сравнению с различными блок-схемами, которые могут быть получены, если разрешить проведение стрелок из любого блока в любой другой блок. Отказавшись от большого разнообразия блок-схем и ограничившись... тремя операторами управления, мы следуем тем самым некоей последовательностной дисциплине» [2, с. 28]

§17. ПЛОХИЕ БЛОК-СХЕМЫ ИЛИ ПЛОХИЕ СТАНДАРТЫ?

Проведенный анализ позволяет сделать несколько замечаний.

- Обвинения, выдвигаемые противниками блок-схем, неправомерны, потому что ставят проблему с ног на голову. Дело не в том, что блок-схемы по своей природе противоречат принципам структуризации. А в том, что при разработке стандартов на блок-схемы указанные принципы не были учтены. На них просто не обратили внимания, поскольку в ту пору – именно в силу парадигмальной слепоты – считалось, что на практике структурный подход следует применять к текстам программ, а отнюдь не к блок-схемам.
- Чтобы сделать блок-схемы пригодными для структуризации, необходимо, в частности, ограничить и регламентировать их топологию с учетом видеоструктурных принципов Дейкстры.
- Видеоструктурная концепция Дейкстры – это фундаментальная идея, высказанная более тридцати лет назад и оказавшаяся невос требованной, поскольку она значительно опередила свое время.
- Эту задачу решает предлагаемый нами шампур-метод, понимаемый как *метод формализации блок-схем*, который *развивает* видеоструктурную концепцию Дейкстры. С помощью шампур-метода разработана новая топология блок-схем (дракон-схемы), регламентация которой произведена на основе принципа когнитивной формализации знаний.

Наибольшую трудность в течение длительного времени представляли математика и эргономика блок-схем. Нужно было создать математически строгий метод формализации блок-схем, учитывающий правила эргономики и позволяющий превратить блок-схемы в программу, пригодную для ввода в компьютер и трансляции в объектный модуль программы.

Язык ДРАКОН позволил эффективно решить эту задачу. Одновременно была решена задача эргономизации блок-схем, то есть задача приспособления блок-схем для наиболее удобного человеческого восприятия и понимания.

§18. ВИЗУАЛЬНОЕ И ТЕКСТОВОЕ СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Можно предложить ряд правил, устанавливающих соответствие между понятиями двумерного (визуального) и одномерного (текстового) структурного подхода.

1. Макроикона «развилка» (рис. 242), в которую произведен ввод функционального атома в левую или обе критические точки, эк-

- вивалентна конструкциям *if-then* и *if-then-else* соответственно [11, с. 120, 121]. См. также две верхние графы на рис. 261.
2. Макроикона «обычный цикл» (рис. 242), в которую произведен ввод функционального атома в одну или обе критические точки, эквивалентна конструкциям *do-until*, *while-do*, *do-while-do* соответственно [11, с. 120, 121]. См. также две нижние графы на рис. 261.
 3. Непустая макроикона «переключатель» (рис. 242), в которую введено нужное число икон «вариант» с помощью операции «добавление варианта», эквивалентна конструкции *case* [2, с. 27]. См. также рис. 261.
 4. Непустая макроикона «цикл ДЛЯ» (рис. 242) эквивалентна циклу *for* языка Си.
 5. Непустая макроикона «переключающий цикл» (рис. 242), в которую введено нужное число икон «вариант», эквивалентна циклу с вложенным оператором *case*, используемым, например, при построении рекурсивных структурированных программ по методу Ашкрофта-Манна [11, с. 141, 142].
 6. Шампур-блок – видеоструктурный эквивалент понятия «простая программа» [11, с. 102].
 7. Атом – общее понятие для основных составляющих блоков, необходимых для построения программы согласно структурной теореме Бомы и Джакопини [5]. Оно охватывает также все элементарные конструкции структурного подхода, кроме последовательности [11, с. 119–121]. (Последняя в визуальном структурном подходе считается неэлементарной структурой).
 8. Операция «ввод атома» позволяет смоделировать две операции:
 - построить последовательность из двух и более атомов;
 - методом заполнения критических точек осуществить многократное вложение составных операторов друг в друга.

Благодаря этому единственный функциональный блок «постепенно раскрывается в сложную структуру основных элементов».

§19. СТРУКТУРНЫЕ, ЛИАННЫЕ И АДРЕСНЫЕ БЛОКИ

Введем понятие «базовые операции» для обозначения операций «ввод атома», «добавление варианта» и «боковое присоединение» (см. главу 33). Применяя к заготовке-примитив базовые операции любое число раз, мы всегда получим структурную дракон-схему в традиционном смысле слова (рис. 262).

Введем три понятия.

Структурный
блок

Это шампур-блок, построенный только с помощью базовых операций, без использования действий с лианой (рис. 262).

Лианный
блок

Это шампур-блок, полученный из структурного блока с помощью операции «пересадка лианы» (рис. 263).

Адресный
блок

- Это результат преобразования структурного блока с помощью операции «заземление лианы» и, возможно, «пересадка лианы».
- Адресный блок используется только в силуэте и представляет собой многоадресную ветку (или ее нижнюю часть).
- Он имеет один вход и не менее двух выходов, присоединенных к нижней горизонтальной линии силуэта через иконы «адрес» (рис. 265).

- В примитиве могут использоваться только структурные и лианные блоки (рис. 262, 263).
- В силуэте применяются все три типа блоков: структурные, лианные и адресные (рис. 264, 265).

Шампур-метод позволяет строить как структурные, так и лианные конструкции. Выше мы выяснили, что базовые операции моделируют классический структурный подход.

Чтобы смоделировать полезные функции оператора *goto*, в шампур-методе предусмотрены операции «пересадка лианы» и «заземление лианы».

§20. СИЛУЭТ КАК КОНЕЧНЫЙ АВТОМАТ И ДИАГРАММА СОСТОЯНИЙ

Силуэт на рис. 265 можно интерпретировать как детерминированный конечный автомат (рис. 266). Каждую ветку при желании можно характеризовать как состояние автомата. Переход с ветки на ветку означает переход из одного состояния в другое.

Силуэт можно также рассматривать как диаграмму состояний (*state machine diagram*).

§21. АНАЛОГИ ДРАКОН-СХЕМ

Аналогом дракон-схем являются диаграммы поведения (*behaviour diagrams*) языка *UML*, в частности, диаграмма деятельности (*activity diagram*), диаграмма состояний (*UML state machine diagram*) и некоторые

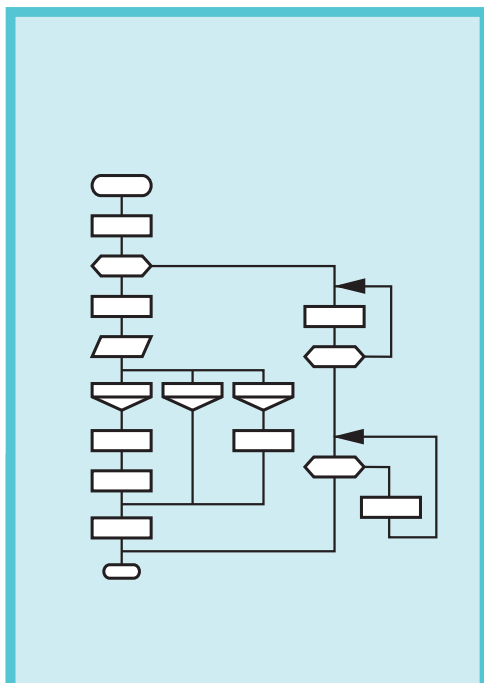


Рис. 262. Примитив, построенный из структурных блоков

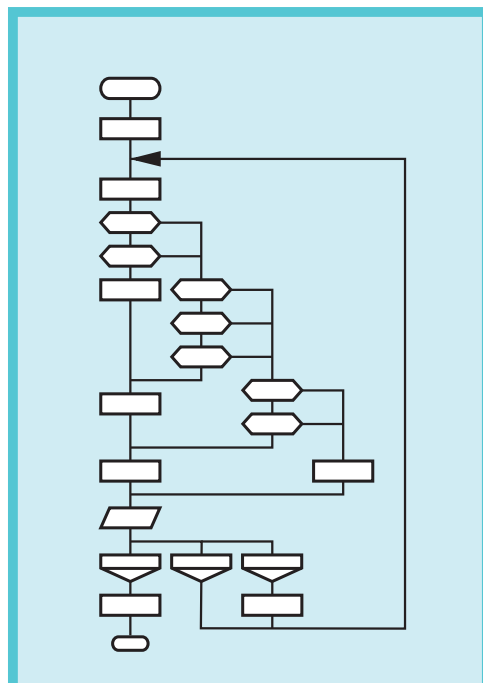


Рис. 263. Примитив, построенный из структурных и лианых блоков

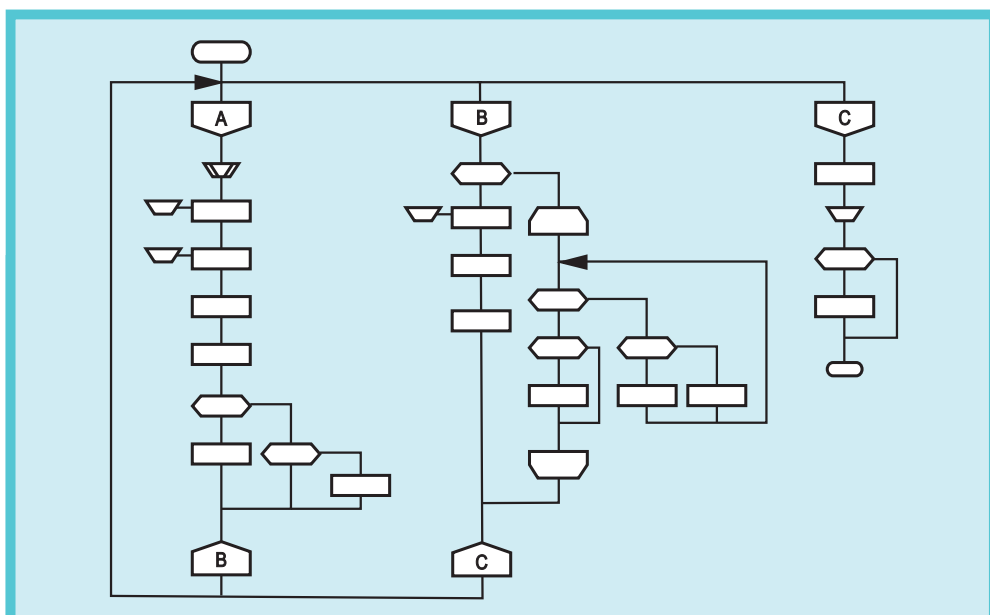


Рис. 264. Силуэт, построенный из структурных блоков

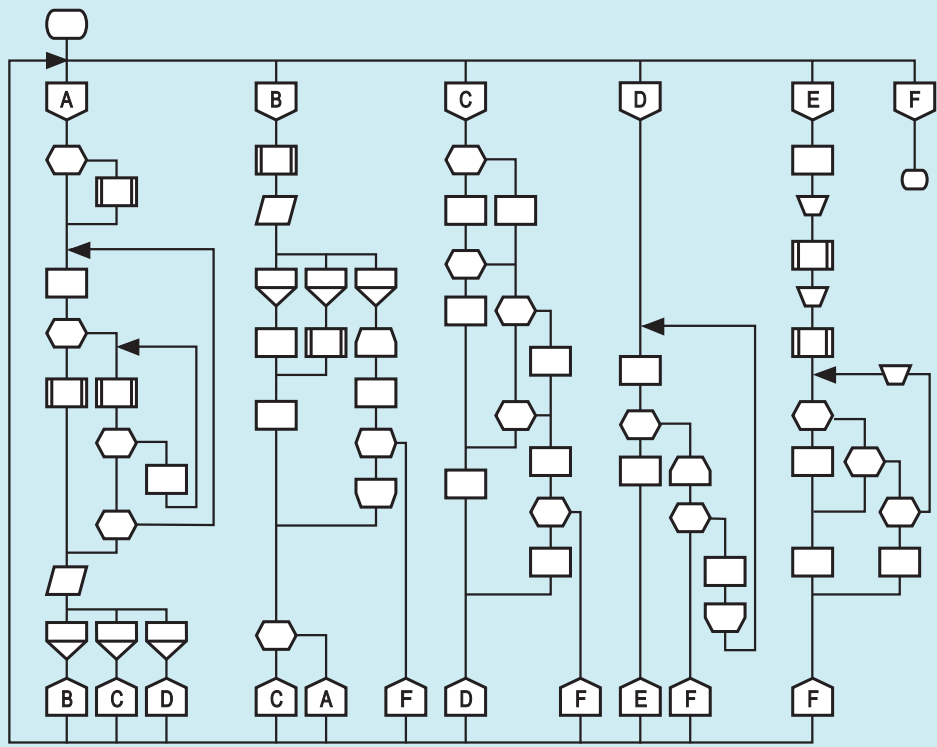


Рис. 265. Силуэт, построенный из структурных, лианных и адресных блоков

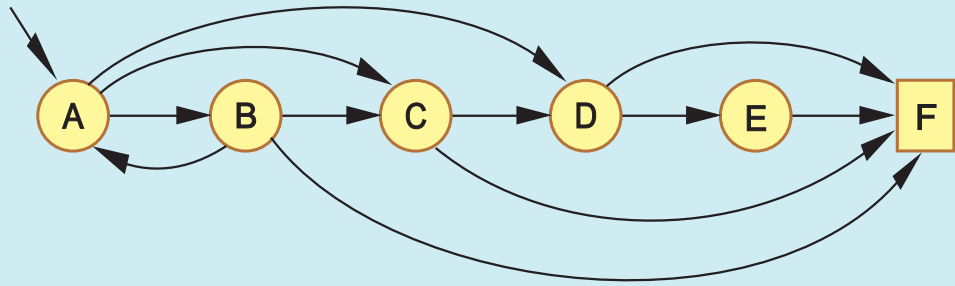


Рис. 266. Детерминированный конечный автомат, соответствующий силуэту на рис. 265

диаграммы взаимодействия (*interaction diagrams*), например, диаграмма синхронизации (*timing diagram*).

Другая группа аналогов дракон-схем охватывает блок-схемы алгоритмов по ГОСТ 19.701–90, диаграмму Насси-Шнейдермана, псевдокод (язык описания алгоритмов) и др.

§22. ОПЕРАЦИИ С ЛИАНОЙ И ОПЕРАТОР GOTO

Операции с лианой моделируют все без исключения функции заменителей *goto* (например, досрочный выход из цикла), а также некоторые функции *goto*, которые невозможно реализовать с помощью заменителей. Однако они не приводят к хаосу, вызванному бесконтрольным использованием *goto*.

С эргономической точки зрения, действия с лианой на порядок эффективнее и удобнее, чем *goto* и заменители. Кроме того, они весьма эффективно корректируют недостатки классического (текстового) структурного программирования.

Приведем пример. В главе 7 мы рассмотрели эргономические преимущества схемы на рис. 62 по сравнению с рис. 61. Показано, что улучшение эргономичности достигнуто за счет использования равносильных преобразований алгоритмов: вертикального и горизонтального объединения. При этом за кадром осталась важная проблема – проблема синтаксиса.

Как построить указанные схемы? Теперь мы имеем возможность осветить этот вопрос. Схема на рис. 61 представляет собой структурный блок, полученный с помощью операции «ввод атома». В отличие от нее схема на рис. 62 – это лианный блок, построенный методом пересадки лианы.

Уместно вспомнить предостережение Гленфорда Майерса:

«Правила структурного программирования часто предписывают повторять одинаковые фрагменты программы в разных участках модуля, чтобы избавиться от употребления операторов *goto*. В этом случае лекарство хуже болезни. Дублирование резко увеличивает возможность внесения ошибок при изменении модуля в будущем» [8, с. 138].

Как видно на рис. 53, 56, 62 пересадка лианы позволяет элегантно и без потерь решить эту непростую проблему, одновременно улучшая наглядность и понятность алгоритма, обеспечивая более эффективное топологическое упорядочивание маршрутов.

Пересадка лианы узаконивает лишь некоторые, отнюдь не любые передачи управления, поскольку определение операции «пересадка лианы» (см. главу 33, тезис 28) содержит ряд ограничений.

Запрет на образование нового цикла вызван тем, что переход на оператор, расположенный выше (раньше) в тексте программы, считается

«наихудшим применением оператора *goto* [8, с. 135]. Указанный запрет вводится, чтобы выполнить требование: использовать *goto* только для передачи управления вперед по программе, которое считается более приемлемым.

Запрет на образование второго входа в цикл соответствует требованию структурного программирования, согласно которому цикл, как и любая простая программа, должен иметь не более одного входа.

Лишь третий запрет является оригинальной особенностью шампур-метода: он запрещает передачи управления, изображение которых с помощью лианы ведет к пересечению линий.

Таким образом, пересадка лианы разрешает только те переходы вниз по дракон-алгоритму, которые образуют связи с валентными точками и изображаются легко прослеживаемыми маршрутами, то есть не пересекающимися линиями.

В визуальном структурном подходе аналогом *goto* и заменителей служат формальные операции «пересадка лианы» и «заземление лианы».

§23. ПОЧЕМУ САМОЛЕТ НЕ МАШЕТ КРЫЛЬЯМИ?

Говоря о будущем шампур-метода, необходимо осознать, что одномерный (текстовый) и двумерный (визуальный) подходы опираются на разные системы понятий, которые по-разному расчленяют действительность. Поэтому визуальный подход нельзя рассматривать как результат механического перевода устоявшихся понятий классического текстового структурного программирования на новый язык.

Поясним. При визуальном структурном подходе программист работает только с чертежом программы, не обращаясь к ее тексту. Точно так же программист, работающий, скажем, на Дельфи, не обращается к ассемблеру и машинному коду — они для него просто не существуют!

Это значит, что столь тщательно обоснованная Дейкстрой и его коллегами коллекция ключевых слов структурного программирования (*if, then, else, case, of, while, do, repeat, until, begin, end* [2, с. 22, 26, 27]) при переходе к визуальному подходу становится ненужной. Для программиста она просто перестает существовать!

В равной степени становятся лишними и отмирают и другие ключевые слова, используемые оппонентами Дейкстры в различных вариантах расширения структурного подхода: *goto, continue, break, exit* и т. д.

Поскольку в визуальном варианте структурного подхода ключевое слово *goto* не используется, теряют смысл и все споры относительно за-

конности или незаконности, опасности или безопасности его применения. Становится ненужной обширная литература, посвященная обсуждению этого, некогда казавшегося столь актуальным вопроса.

Предвижу возражения: хотя названные ключевые слова действительно исчезают, однако выражаемые ими понятия сохраняют силу и для визуального случая. Например, два визуальных алгоритма на рис. 60, 62 можно построить с помощью понятий *if-then-else* и *goto*.

Данное возражение нельзя принять, поскольку произошла подмена предмета обсуждения. С помощью указанных понятий можно построить не визуальные алгоритмы, а текстовые алгоритмы, эквивалентные алгоритмам на рис. 60, 62.

Что касается собственно визуальных программ, то они, будучи «выполнимой графикой», строятся из «выполняемых» икон и «выполняемых» соединительных линий. Подчеркнем еще раз: при построении алгоритма (программы) с помощью дракон-конструктора пользователь не применяет понятия *if-then-else* и *goto*, ибо в этом нет никакой необходимости.

Нельзя путать задачу и систему понятий, на которую опирается метод ее решения. В обоих случаях – и при текстовом, и при визуальном структурном подходе – ставится одна и та же задача: улучшить понятность программ и обеспечить более эффективный интеллектуальный контроль за передачами управления.

Однако система понятий коренным образом меняется. Ту функцию, которую в текстовой программе выполняют ключевые слова, в визуальной программе реализуют совершенно другие понятия: иконы, макроиконы, соединительные линии, шампур, главная вертикаль шампура-блока, лиана, атом, пересадка лианы, запрет пересечения линий и т. д.

Очевидно, что использование понятий, выражаемых ключевыми словами текстового структурного программирования, не является самоцелью. Оно служит для того, чтобы «делать наши программы разумными, понятными и разумно управляемыми» [4, с. 272]. Указанные понятия решают эту задачу не во всех случаях, а только в рамках текстового программирования.

При переходе к визуальному подходу задача решается по-другому, с помощью другого набора понятий. Отказ от старого набора понятий и замена его на новый позволяет добиться новой постановки задачи и более эффективного ее решения. Поэтому высказываемое иногда критическое замечание: «недостаток шампура-метода в том, что он не удовлетворяет требованиям структурного программирования» является логически неправомерным. Нельзя упрекать самолет за то, что он не машет крыльями.

§24. ВЫВОДЫ

1. Императивная (процедурная) часть языка ДРАКОН опирается на новый метод – двумерное структурное программирование. Или, что одно и то же, *шампур-метод*.
2. Правила двумерного структурного программирования существенно отличаются от традиционного одномерного (текстового) структурного программирования.
3. Идеи структурного программирования разрабатывались, когда компьютерная графика фактически еще не существовала и основным инструментом алгоритмиста и программиста был одномерный (линейный или ступенчатый) текст.
4. До появления компьютерной графики методология текстового структурного программирования была наилучшим решением.
5. С появлением компьютерной графики ситуация изменилась. Появилась возможность заменить текстовые управляющие структуры на управляющую графику, то есть использовать двумерное структурное программирование.
6. Слабое место традиционного структурного программирования и текстового представления алгоритмов и программ заключается в недостатке выразительных средств. Следствием являются ограничения и запреты. Эти ограничения и запреты вытекают из природы текста, из природы текстового представления управляющих структур.
7. Недостаток выразительных средств, проявляющийся через ограничения и запреты, тормозит повышение производительности труда алгоритмистов и программистов.
8. В рамках текстового представления управляющих структур устранить эти ограничения и запреты невозможно. Чтобы добиться улучшения, надо перейти от одномерного текстового структурного программирования к двумерному визуальному структурному программированию.
9. Многие ограничения и запреты, неизбежные при текстовом структурном программировании, во многих случаях противоречат здравому смыслу, затрудняют понимание алгоритмов и программ, искажают нормальный ход человеческой мысли.
10. Недопустимо запрещать правильный процесс мышления. Его надо разрешить. Шампур-метод и язык ДРАКОН устраняют этот недостаток.
11. При использовании шампур-метода набор управляющих ключевых слов текстового структурного программирования становится ненужным.
12. При визуальном структурном подходе программист работает только с чертежом программы (дракон-схемой), не обращаясь к ее тек-

стовому представлению. Точно так же программист, работающий, скажем, на Дельфи, не обращается к ассемблеру и машинному коду – они для него просто не существуют.

13. Во многих случаях (список которых еще предстоит уточнить) желательно отказаться от текстовых управляющих структур, заменив их управляющей графикой.
14. ДРАКОН – это не просто новый язык (новое семейство языков). Это новый взгляд на процедурное программирование. Если угодно – новое мировоззрение.
15. Наибольшую трудность в течение длительного времени представляли математика и эргономика блок-схем. Нужно было создать математически строгий метод формализации блок-схем, позволяющий превратить блок-схемы в программу, пригодную для ввода в компьютер и трансляции в объектный модуль программы.
16. Язык ДРАКОН позволил эффективно решить эту задачу.
17. Одновременно была решена задача эргономизации блок-схем, то есть задача приспособления блок-схем для наиболее удобного человеческого восприятия и понимания.

Часть VIII

КАКУЮ РОЛЬ ИГРАЮТ АЛГОРИТМЫ В ЧЕЛОВЕЧЕСКОЙ КУЛЬТУРЕ?

Задача данной книги – изложить «Основы алгоритмизации». Эта задача уже выполнена (в предыдущих семи частях). В принципе, на этом можно было бы поставить точку.

Однако нельзя оставить без внимания социальные, гуманитарные и культурные аспекты алгоритмизации. Этой важной теме посвящена последняя (восьмая) часть.

АЛГОРИТМИЧЕСКОЕ МЫШЛЕНИЕ

§1. АЛГОРИТМЫ И ЧЕЛОВЕЧЕСКАЯ КУЛЬТУРА

Алгоритмы играют в человеческой культуре огромную роль. Они выполняют две функции.

Первая очевидна и общеизвестна. В XX веке нашу планету, словно волшебные цветы, усеяли сотни миллионов компьютеров. Компьютеры не могут работать без программ, а в программах «спрятаны» алгоритмы. Следовательно, без алгоритмов не может существовать современная цивилизация.

Вторая функция не менее важна. Алгоритмы оказывают влияние на человеческое мышление, улучшая работу ума. К сожалению, этот процесс идет очень медленно, затрагивая преимущественно программистов и математиков. Все остальные (то есть НЕпрограммисты), как правило, не знакомы с алгоритмами. Это обстоятельство существенно тормозит интеллектуальное развитие населения.

§2. АЛГОРИТМИЧЕСКАЯ НЕГРАМОТНОСТЬ

Оценивая ситуацию в целом, можно сказать, что на нашей планете господствует *алгоритмическая неграмотность*.

Во многих странах были предприняты попытки поднять алгоритмическую грамотность с помощью системы образования. К сожалению, эти попытки не имели заметного успеха.

Естественно возникает вопрос: в чем причина неудачи? Почему алгоритмическая неграмотность является столь живучей? И как можно поправить дело?

Это очень важные вопросы, на которые следует дать тщательный и аргументированный ответ.

В главах 37–41 проделан подробный анализ проблемы, выявлены причины трудностей и намечен путь, который, как нам кажется, позволит переломить ситуацию.

Алгоритмическая неграмотность волнует многих дальновидных людей. Рассмотрим простой пример, заимствованный из сети Интернет.

§3. АЛГОРИТМ «ЧАШКА ЧАЯ»

Заглянем в один из блогов Живого Журнала. Нас встречает хозяин блога Митрич (meatreach).

Лукаво улыбаясь, Митрич говорит:

– Объясните мне, как приготовить чашку чая? По-простому, из пакетика. Я задавал этот вопрос разным людям. И все время повторял: «дайте мне инструкцию, научите меня».

Чаще всего ответ выглядел так:

– Берем чашку, берем пакетик, кладем в чашку. Заливаем водой из чайника.

– А если в чайнике вода холодная?

– Ну, тогда сначала кипятим.

– А если там воды нет?

– Тогда наливаем воду, потом кипятим, потом наливаем в чашку. Но если ты хочешь, чтобы чай был «правильный», сначала надо, чтобы чашка была горячая.

Что же получается? Действуя по такой инструкции, я сначала залью пакетик холодной водой. Потом сожгу чайник, включив его без воды. Потом все-таки заварю чашку чая. И тут же выясню, что чай получился «неправильный». И надо все начинать сначала. А ведь задачка не такая уж и сложная.

Беда в том, что обычный человек не способен сформулировать простой пошаговый алгоритм [1].

§4. АЛГОРИТМ МИТРИЧА ИЗ ВОСЬМИ ШАГОВ

Шаг 1. Если в чайнике вообще нет или слишком мало воды, надо налить ее.

Шаг 2. Если вода в чайнике холодная, надо вскипятить ее.

Шаг 3. Если нет чистой чашки, надо вымыть грязную.

Шаг 4. Взять чистую чашку.

Шаг 5. Налить полчашки горячей воды, чтобы чашка согрелась.

Шаг 6. Положить в чашку пакетик, долить водой.

Шаг 7. Ждать минуту, помешивать пакетик ложкой.

Шаг 8. Выкинуть пакетик [1].

Внимательный читатель легко заметит, что Митрич несколько упростил ситуацию. Сравните алгоритм Митрича и более подробный алгоритм на рис. 267.

§5. ОСОБЕННОСТЬ АЛГОРИТМИЧЕСКОГО МЫШЛЕНИЯ

О чем говорит Митрич? О том, что есть люди, и их немало, которые не обладают алгоритмическим мышлением.

Как же ведут себя эти люди? В каком-то смысле они похожи на полуслепых. Они замечают только самое главное:

«Берем чашку, берем пакетик, кладем в чашку. Заливаем водой из чайника».

И на этом ставят точку. Они упускают из вида все детали, все условия и подробности (назовем их *сопутствующие действия*).

Говоря упрощенно, алгоритмическое мышление состоит в умении выполнять следующие мыслительные операции.

- Выделить *главные* действия.
- Тщательно продумать задачу и выявить все *сопутствующие* действия.
- Указать сопутствующие действия, которые *предшествуют* главным действиям.
- Указать сопутствующие действия, которые *выполняются после* главных действий.
- Соединить главные и сопутствующие действия в логичную и обоснованную последовательность действий (алгоритм).

Большинство людей не обладают навыками алгоритмического мышления. Эти навыки, как правило, есть только у программистов и математиков.

§6. МИТРИЧ РАЗМЫШЛЯЕТ ОБ АЛГОРИТМИЧЕСКОЙ НЕГРАМОТНОСТИ

Обучение алгоритмическому мышлению, которое дают в школе, ведется неправильно и не достигает цели. К тому же про алгоритмы говорят только на уроках информатики. На всех остальных уроках этой темы не касаются.

К чему это приводит? Подавляющее большинство детей, заканчивающих школу, не владеют алгоритмическим мышлением. Они не способны сформулировать простейшую пошаговую инструкцию.

Вы думаете, такие навыки нужны только программистам?

- О, нет, это совсем не так, – говорит Митрич. И продолжает:
- Почитайте законы. Почитайте нормативные акты, инструкции.

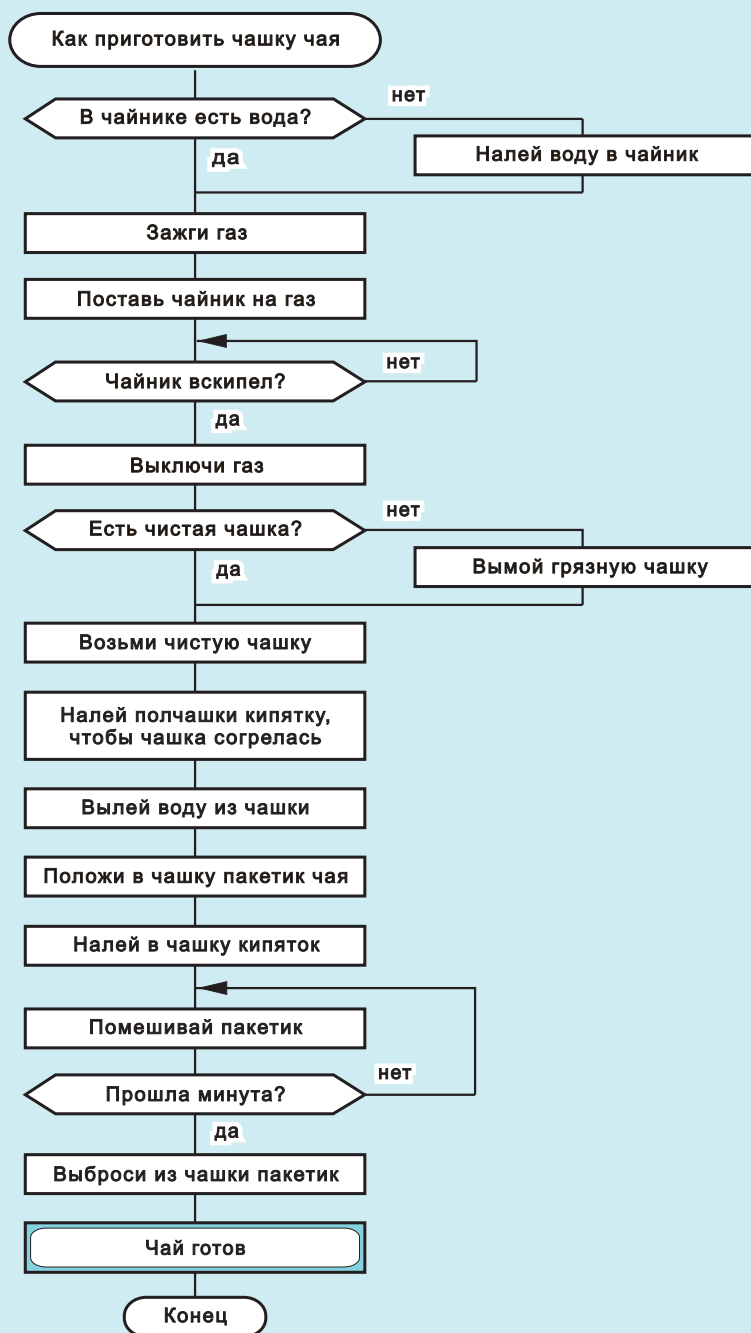


Рис. 267. Алгоритм «Как приготовить чашку чая»

Почитайте письма Министерства финансов, описывающие правила, алгоритмы и методики ведения бухгалтерского учета. Почитайте регламенты и правила в государственных органах и частных компаниях.

Все эти документы написаны людьми, которые, как бы сказать помягче... не очень сильны по части алгоритмов.

Послушайте, как какой-нибудь менеджер, бригадир или прораб объясняет своему сотруднику, что тот должен делать. А еще веселее – найдите менеджера, бригадира или прораба, способного сформулировать пошаговую инструкцию со всеми сопутствующими действиями. И посмотрите на его сотрудника, когда он будет пытаться воспринять эту инструкцию!

Вы увидите поистине удивительную картину. У большинства людей отсутствует важнейший навык. Отсутствует умение мыслить алгоритмически [1].

§7. ТРИ ЗАДАЧИ СИСТЕМЫ ОБРАЗОВАНИЯ В ОБЛАСТИ ИНФОРМАТИКИ

Забудем про частные примеры. И поднимем уровень обсуждения. Среди многих задач системы образования выделим три:

- подготовка квалифицированных пользователей компьютера;
- подготовка программистов (в специализированных классах и школах, а также вузах);
- овладение алгоритмическим мышлением.

Первые две задачи лежат за рамками нашего обсуждения. Сосредоточим внимание на третьей.

Задача овладения алгоритмическим мышлением повсеместно недооценивается и решается неудовлетворительно.

Следствием этого обстоятельства является господствующая в обществе массовая алгоритмическая неграмотность.

§8. ВЫВОДЫ

1. Проблема алгоритмического мышления – это отдельная, самостоятельная и чрезвычайно важная проблема.
2. Существующие средства не позволяют решить проблему.
3. Необходимо коренным образом пересмотреть подход к данной проблеме и использовать *новые* средства для ее успешного решения.

АЛГОРИТМЫ И УЛУЧШЕНИЕ РАБОТЫ УМА

§1. ВВЕДЕНИЕ

Алгоритмы и алгоритмическое мышление содействуют усилению человеческого интеллекта. Развитие алгоритмической грамотности увеличивает интеллектуальный потенциал общества. И наоборот, алгоритмическая неграмотность лишает людей этого важного преимущества.

§2. ПРОБЛЕМА УЛУЧШЕНИЯ РАБОТЫ УМА

Развитие алгоритмического мышления тесно связано с концепцией улучшения работы ума [1–3]. Исходя из этой концепции, ставится задача совершенствования человеческого интеллекта [4].

Однако некоторые авторы считают проблему улучшения интеллекта некорректной и надуманной, поскольку

«за всю историю развития человеческой цивилизации не отмечено сколь-нибудь заметное совершенствование человеческого интеллекта» [5].

Подобное возражение основывается на недоразумении. Ведь речь идет не о генетически заданных предпосылках развития мышления (геном человека, как предполагают ученые, действительно не меняется на протяжении последних тысячелетий), а о механизмах социального наследования и передачи знаний. И связанных с ними *приобретенных* возможностях интеллекта, которые можно значительно усилить [2, с. 298].

§3. ДИСКУССИОННЫЕ ВОПРОСЫ, СВЯЗАННЫЕ С РАБОТОЙ МОЗГА

Специалист по эволюции интеллекта Борис Сергеев пишет:

«Нейроантропология достаточно убедительно доказала, что мозг современного человека и нашего далекого предка за последние 100 000 – 1 000 000 лет не претерпел существенных изменений» [6].

Так ли это? Неужели за миллион лет мозг действительно не изменился? Ведь невозможно представить, чтобы мозг такого примитивного существа, каким был древний человек, создал современную науку и искусство.

Каким образом был получен странный вывод о неизменности мозга? Оказывается, самым примитивным путем. Исследователи взяли старинные черепа, тщательно их измерили (особенно внутри) и сравнили с современными. Результаты показали, что крупные куски мозга древних людей по размерам не слишком отличаются от современных.

Полученный по такой методике вывод о неизменности мозга представляется сомнительным. Ведь при таком подходе упускается главное – микроскопический характер и необозримая сложность межнейронных связей [1, с. 619].

Нет сомнения, что древние полуобезьяны и современные люди имеют почти одинаковые черепа (и основные части мозга).

Но ведь не в этом же суть! Решающее отличие древнего и современного мозга – в изменении карты межнейронных соединений. *Кардинальное изменение межнейронной «кабельной сети» – вот что объясняет гигантский скачок интеллекта* [1, с. 618, 619].

Сегодня люди умнее всех своих предков. Они превосходят и древних египтян, и современников Сократа, и мастеров эпохи Возрождения, и интеллектуалов XIX века. Почему? Потому что внутри их черепов находятся принципиально новые узоры межнейронной паутины [1, с. 619].

Эти новые узоры получены не в результате генетического наследования (за это время гены почти не изменились). А в результате мощного развития культуры, прежде всего знаковой культуры. Создав знаки, люди приобрели мощные инструменты, позволяющие изменять одну из важнейших структур мозга – межнейронные связи. И за счет этого неуклонно наращивать собственный интеллект [1, с. 618, 619].

§4. ИСТОРИЧЕСКИЕ ПРЕДПОСЫЛКИ ПРОБЛЕМЫ УСИЛЕНИЯ ИНТЕЛЛЕКТА

Изучение проблемы улучшения работы ума есть развитие давней научной традиции. В рамках этой традиции, в связи с отсутствием устоявшейся терминологии, ученые используют сходные по смыслу, хотя и не полностью совпадающие понятия и выражения:

- «расширение способностей ума» (*Рене Декарт*);

- «улучшение наших умозаключений», «облегчение процесса нашего мышления» (*Готфрид Лейбниц*);
- «усиление человеческого мышления» (*Эрнст Шредер*);
- «облегчение мышления и придание ему большей точности и силы» (*Готлоб Фреге*);
- «улучшение понимания» (*Макс Вертгеймер*);
- «усиление мыслительных возможностей» (*Джером Брунер*);
- «усиление природных психических процессов» (*Джеймс Верч*);
- «увеличение КПД функционирования человеческого мозга», «возрастание эффективности человеческого мышления», «усиление знаниепорождающих, творческих функций естественного интеллекта человека», «качественная интенсификация массового научного творчества» (*Александр Зенкин*).

§5. МОЖЕТ ЛИ ЧЕЛОВЕК СТАТЬ УМНЕЕ?

ПРОТИВОРЕЧИВЫЕ ТРЕБОВАНИЯ К ЧЕЛОВЕЧЕСКОМУ МОЗГУ

Развитие цивилизации сопровождается значительным усложнением интеллектуальных задач, увеличением нагрузки на мозг. В свою очередь, рост нагрузки, повышение напряженности умственного труда нередко приводят к перегрузке мозга.

В условиях перегрузки людям намного труднее охватить умом отдаленные последствия своих и чужих решений и действий. Поэтому многие важные детали выпадают из поля зрения и не учитываются. Это может привести и нередко приводит к негативным последствиям.

Налицо противоречие. С одной стороны, чтобы избежать нежелательных перегрузок и вызванных ими упущений, необходимо уменьшить нагрузку на мозг. С другой стороны, развитие и усложнение цивилизации приводит к лавинообразному усложнению задач и безостановочному росту их количества. Все это предъявляет к мозгу постоянно растущие требования, направленные на повышение его интеллектуальной производительности.

Как разрешить это противоречие? Можно ли выполнить два противоположных требования:

- облегчить работу мозга
- и одновременно увеличить его умственную продуктивность?

§6. КОМПЬЮТЕРНАЯ МИФОЛОГИЯ:

ОБЛЕГЧАЮТ ЛИ КОМПЬЮТЕРЫ УМСТВЕННЫЙ ТРУД?

Иногда говорят, что компьютеризация и автоматизация умственного труда снимают эту проблему. Дескать, компьютер облегчает работу мозга, принимая на себя значительную часть задач и выполняя их намного быстрее. Благодаря этому нагрузка на человека якобы уменьшается.

Это неверно. В действительности использование компьютеров не приводит к уменьшению напряженности труда. Почему?

Потому что вместо одних заданий (которые удалось переложить на машину), человеческий мозг нередко получает множество новых задач. В итоге суммарная нагрузка не уменьшается и даже возрастает.

Все чаще ученые приходят к выводу, что применение компьютеров во многих или даже большинстве случаев не только не упрощает, а наоборот, резко усложняет интеллектуальные задачи, которые остаются на долю человека.

Например, Эдсгер Дейкстра пишет о «неисчерпаемой» и «беспрецедентной» сложности задач, которые приходится решать программистам [7]. С ним соглашается академик Андрей Ершов: «Программисты составляют первую большую группу людей, работа которых ведет к пределам человеческого знания... и затрагивает глубочайшие тайны человеческого мозга» [8].

Психолог Михаил Ярошевский отмечает:

«Успехи кибернетики приводят к тому, что резко расширяются перспективы передачи техническим устройствам тех умственных операций, которые поддаются формализации. Такие операции раньше поглощали значительную часть интеллектуальных усилий ученого. Однако теперь ситуация изменилась. В новых условиях резко повышаются требования к формированию способностей ученого производить такие действия, которые не могут совершаться компьютерами» [9].

Таким образом, массовая компьютеризация не отменяет интересующую нас проблему усиления интеллекта и повышения продуктивности умственного труда. Напротив, она делает ее еще более актуальной.

§7. ТЕЗИС АНДРЕЯ ЕРШОВА

Настало время поставить вопрос в предельно четкой форме.

Может ли человек улучшить свой интеллект? Или это в принципе невозможно?

Сорок лет назад высказал свое мнение академик Андрей Ершов. Он полагает, что занятия программированием делают людей умнее. Человек резко увеличит свой интеллект, если станет программистом. Вот его слова:

«Человек неизмеримо усилит свой интеллект, если сделает частью своей натуры способность планировать свои действия, вырабатывать общие правила и способ их применения к конкретной ситуации, организовывать эти правила в осознанную и выразимую структуру, – одним словом, сделается программистом» [10].

§8. В ЧЕМ СИЛА АЛГОРИТМИЧЕСКОГО МЫШЛЕНИЯ?

Внимательно прочитав слова Ершова, можно убедиться, что выдвинутый им тезис относится не только к программистам, но и к алгоритмистам. То есть к людям, обладающим алгоритмическим мышлением.

Перефразируя тезис Ершова, получим:

«Человек неизмеримо усилит свой интеллект, если сделает частью своей натуры способность планировать свои действия, вырабатывать общие правила и способ их применения к конкретной ситуации, организовывать эти правила в осознанную и выразимую структуру», – одним словом, овладеет алгоритмическим мышлением и научится применять его на практике.

Если Ершов прав, мы вправе сделать вывод:

Овладев алгоритмическим мышлением, человек неизмеримо усилит свой интеллект.

§9. ПРОФЕССИОНАЛЬНЫЕ ЯЗЫКИ

Алгоритмический язык относится к классу профессиональных языков.

Профессиональный язык – богатое множество знаково-символических средств, включающее естественный человеческий язык, научные понятия и термины, математические и иные формулы, логико-математические исчисления, различные искусственные языки, а также всевозможные схемы, чертежи, графики, диаграммы, карты, современные приемы компьютерной визуализации знаний и пр. [1, с. 255].

Существует обширное множество профессиональных языков, так как специалисты в разных областях знания обычно используют разные языки.

§10. О ВЛИЯНИИ ПРОФЕССИОНАЛЬНЫХ ЯЗЫКОВ НА МЫШЛЕНИЕ ЛЮДЕЙ

Уже давно замечено, что профессиональные языки оказывают заметное влияние на мышление.

Известно, например, что «программист мыслит категориями, которые дает ему в распоряжение язык программирования» [11]. По мнению экспертов, влияние языка «независимо от нашего желания сказывается на нашем способе мышления» [7, с. 9]. Язык оказывает глубокое воздействие «на навыки мышления и изобретательские способности», причем «царящий в существующих языках беспорядок» непосредственно отражается на стиле и эффективности труда [11].

Чем определяется тот предел, до которого может усилить свой интеллект программист? Эта грань жестко регламентируется его личным интеллектуальным опытом и характеристиками тех языков, которые он использует в работе. Между тем все без исключения известные языки программирования наряду с многочисленными достоинствами имеют существенный изъян – это кастовые языки ограниченного применения [2, с. 299].

§11. НЕДОСТАТОК ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Языки программирования предназначены только для «элиты» (только для программистов). Следовательно, это «праздник не для всех». Миллионы людей, не знакомых с программированием, не имеют к этому «празднику» никакого отношения и ничего не выигрывают.

Таким образом, языки программирования не могут оказать облагораживающее воздействие на интеллектуальную жизнь общества, преодолеть раздробленность индивидуальных интеллектов и обеспечить необходимое усиление могущества коллективного разума, соответствующее современным требованиям [1, с. 431, 2, с. 299].

Ясно, однако, что задача улучшения работы ума должна решаться не только применительно к программистам, но и к миллионам других людей.

Каким образом можно решить эту задачу? Об этом пойдет речь ниже.

§12. МОЖНО ЛИ УВЕЛИЧИТЬ УМСТВЕННУЮ МОЩЬ ЦИВИЛИЗАЦИИ?

Социальный успех любого искусственного языка, его укорененность в культуре, возможность крупномасштабного расширения сферы его применения и международного признания зависят от общедоступности и полезности языка. Полезность хороших профессиональных языков определяется тем, что они должны облегчить понимание и взаимопонимание, обеспечить стратегический интеллектуальный прорыв, позволяющий качественно образом увеличить умственную мощь цивилизации.

Можно ли решить подобную задачу в принципе? Строго говоря, доказательный ответ на этот вопрос пока отсутствует. Вместе с тем можно высказать некоторые предположения.

Письменный язык – это система нотаций, а нотация, как утверждает Кеннет Айверсон, есть «средство мышления» [12].

Анализируя проблему улучшения нотаций, известный английский логик, математик и философ Альфред Норт Уайтхед пишет:

«Освобождая мозг от всей необязательной работы, хорошая нотация позволяет ему сосредоточиться на более сложных проблемах и в результате увеличивает умственную мощь цивилизации» [12].

§13. РАБОЧАЯ ГИПОТЕЗА: КАЧЕСТВО ЯЗЫКА ВЛИЯЕТ НА СИЛУ УМА

Эргономическое качество алгоритмических языков имеет большое значение. Если качество низкое, алгоритмы оказываются слишком сложными и трудными для понимания. Они содержат смысловые пробелы, упущения и непонятные места. Это порождает многочисленные неясности, заблуждения, ошибки и взаимное непонимание между соисполнителями работ.

В итоге производительность умственного труда создателей алгоритмов падает. Но это не все. Чрезмерная трудность понимания алгоритмов ставит непреодолимую стену перед всеми, кто хочет освоить алгоритмизацию.

И наоборот, если качество языка высокое, алгоритмы становятся легкими для восприятия. Растет глубина и скорость понимания и взаимопонимания. Смысловые ошибки, противоречия и слабые места «сами бросаются в глаза». Их можно гораздо быстрее обнаружить и устранить. В результате продуктивность мозга существенно возрастает.

Сказанное позволяет выдвинуть гипотезу.

Рабочая гипотеза

Производительность мозга разработчика алгоритмов зависит от эргономичности алгоритмического языка. Улучшая язык, можно поднять производительность.

§14. ОБОСНОВАНИЕ ГИПОТЕЗЫ

Данная гипотеза нуждается в тщательном научном обосновании. Такое обоснование представлено в литературе. Заинтересованных читателей мы отсылаем к монографиям.

1. Паронджанов В. Д. Как улучшить работу ума. (Новые средства для образного представления знаний, развития интеллекта и взаимопонимания). М.: Радио и связь, 1998. – 352 с.
2. Паронджанов В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! М.: Дело, 2001. – 360 с.

3. Паронджанов В. Д. Почему мудрец похож на обезьяну, или Парадоксальная энциклопедия современной мудрости. М.: РИПОЛ Классик, 2007. – 1154 с. – См. главы 5–26.
4. Паронджанов В. Д. Дружелюбные алгоритмы, понятные каждому (Как улучшить работу ума без лишних хлопот). М.: ДМК-пресс, 2010. – 464 с.

См. также Безель Я. Б. Можно ли улучшить работу ума? Новый взгляд на проблему. Размышления над новой книгой // Вестник Российской академии наук, том 73, № 4, 2003. – С. 363–365. *Рецензия на книгу: Паронджанов В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! М.: Дело, 2001.*

§15. ВЫВОДЫ

1. Овладение алгоритмическим мышлением позволяет человеку усилить свой интеллект.
2. Производительность мозга разработчика алгоритмов зависит от эргономичности алгоритмического языка. Улучшая эргономичность языка, можно поднять производительность мозга.
3. Традиционные алгоритмические языки создавались без учета требований когнитивной эргономики.
4. Одновременное выполнение правил математики и эргономики позволяет получить *эргономичный* алгоритмический язык.
5. Эргономичный язык существенно превосходит традиционные алгоритмические языки по критерию «понятность алгоритмов для человеческого зрительного восприятия».
6. Эргономичный алгоритмический язык позволяет легче и быстрее овладеть алгоритмическим мышлением. В результате алгоритмическое мышление становится доступным для более широких кругов населения.
7. Примером эргономичного алгоритмического языка является язык ДРАКОН.

АЛГОРИТМИЧЕСКОЕ МЫШЛЕНИЕ И ДВЕ ГРУППЫ ЛЮДЕЙ

§1. ПРОГРАММИСТЫ И НЕПРОГРАММИСТЫ

Алгоритмы можно условно разбить на две части:

- алгоритмы, которые находятся в компьютерах в составе компьютерных программ. Создателями этих алгоритмов являются программисты и математики;
- алгоритмы, которые обычно «живут» вне программ и вне компьютеров. К этим алгоритмам имеют отношение НЕпрограммисты.

Таким образом, мы различаем две группы людей. Первая группа (программисты и математики) владеет алгоритмическим мышлением. Вторая нет.

Для нас наибольший интерес представляют НЕпрограммисты, то есть люди, не умеющие программировать. Сегодня они лишены возможности приобщиться к алгоритмическому мышлению. В результате интеллектуальный потенциал общества снижается.

Но ситуацию можно исправить. Мы полагаем, что непрограммистов можно быстро обучить навыкам алгоритмического мышления. И за счет этого повысить интеллектуальный уровень населения.

§2. СООТНОШЕНИЕ ПРОГРАММИСТОВ И НЕПРОГРАММИСТОВ

Численность программистов и математиков, видимо, не превышает 10% от общего числа специалистов. Остальные 90% – непрограммисты.

Иначе говоря, подавляющее большинство специалистов не владеют алгоритмическим мышлением. На наш взгляд, этот факт представляет собой серьезную проблему.

§3. УРОКИ ИСТОРИИ

Каким образом можно нащупать путь к решению проблемы алгоритмического мышления? В поисках ответа попробуем перенестись на тридцать лет назад. Именно тогда впервые прозвучал знаменитый, но ошибочный лозунг академика Андрея Ершова «Программирование – вторая грамотность!». В свое время этот лозунг облетел весь мир.

Вспомним, как это было.

В 1980 году Ершов получил письмо от Президента ИФИП Пьера Бобилье с приглашением выступить с ключевым докладом на 3-й Всемирной конференции ИФИП и ЮНЕСКО по применению компьютеров в обучении:

«Ваши взгляды на роль компьютеров в обучении будут с признательностью восприняты аудиторией в более чем 1000 человек, которые приедут из 70 стран мира, как развитых, так и развивающихся».

Конференция состоялась в июле 1981 года в Лозанне (Швейцария) [1]. В своем выступлении Ершов, в частности, заявил:

«развитие и распространение ЭВМ приведет ко всеобщему умению программировать». И добавил: «...программировать сумеет каждый, что я и называю второй грамотностью» [2].

§4. В ЧЕМ ОШИБСЯ ЕРШОВ?

В то время многие специалисты опасались, что стремительный рост числа компьютеров потребует резко увеличить число требуемых программ. А для этого чуть ли не все население должно превратиться в программистов. Соглашаясь с этим, Ершов отмечал:

«элементарные расчеты показывают, что для того чтобы через двадцать лет запрограммировать все производимые микропроцессоры, нам нужно посадить за программирование все взрослое население земного шара» [2].

Сегодня мы знаем, что это ошибочный вывод. Специалист по искусственному интеллекту Александр Нариньяни с нескрываемой иронией пишет:

«Казавшийся когда-то передовым лозунг “программирование – вторая грамотность” выглядит первоапрельской шуткой по отношению к сотням миллионов наивных пользователей: школьников, чиновников, солдат, домашних хозяек. Тут и с первой-то грамотностью не все просто. Так что насчет второй и речи быть не может» [3].

§5. ВАЖНЫЙ МОМЕНТ В ПОЗИЦИИ ЕРШОВА

С формальной точки зрения, Ершов в своем докладе вел речь о программах. Однако если подойти к делу не формально, а по существу, легко убедиться, что в большинстве случаев он говорил не о программах, а об *алгоритмах*.

Для примера рассмотрим высказывание Ершова:

«Повседневная жизнь человека, особенно городская, – это деятельность по ПРОГРАММАМ. Каждый человек, придерживающийся режима, с гордостью почувствует себя ПРОГРАММИСТОМ, если вспомнит свои заполненные до предела утренние процедуры, начиная от звонка будильника и кончая началом работы. Поразмышляйте над процедурой уборки в квартире, и вы увидите, что разработка этой программы сделает честь любому профессиональному ПРОГРАММИСТУ...» [2].

Этот отрывок из доклада Ершова будет выглядеть более корректно, если в нем вместо слова «программа» всюду подставить слово «алгоритм». В результате такой замены получим:

Повседневная жизнь человека, особенно городская, – это деятельность по АЛГОРИТМАМ. Каждый человек, придерживающийся режима, с гордостью почувствует себя АЛГОРИТМИСТОМ, если вспомнит свои заполненные до предела утренние процедуры, начиная от звонка будильника и кончая началом работы. Поразмышляйте над процедурой уборки в квартире, и вы увидите, что разработка этой программы сделает честь любому профессиональному АЛГОРИТМИСТУ.

В данном отрывке Ершов говорит о множестве *бытовых* алгоритмов, которые не имеют отношения ни к программам, ни к компьютерам. Развивая мысль Ершова, следует указать на обширный класс подобных алгоритмов, которые связаны не с бытом человека, а с его работой.

§6. ЧАСТИЧНО ФОРМАЛИЗОВАННЫЕ АЛГОРИТМЫ

Примерами частично формализованных алгоритмов являются почти все алгоритмы практической жизни, описанные в главах 17–25. Рассмотрим медицинские алгоритмы на рис. 170, 171, а также нерукотворные биологические алгоритмы на рис. 186, 187, 192, 193.

Чем интересны медицинские алгоритмы? Тем, что они отвечают на насущную потребность практической медицины. Откроем «Практическое руководство для врачей общей (семейной) практики / Под редакцией академика РАМН А. Н. Денисова. – М.: ГЭОТАР-МЕД, 2001. – 720с.».

Для кого предназначена эта книга? Правильно, «для врачей общей практики (семейных врачей), участковых терапевтов и педиатров, амбу-

латорно-поликлинических врачей других специальностей, интернов, клинических ординаторов, студентов медицинских вузов».

Слава богу, в книге собраны алгоритмы, позарез нужные врачам. Это диагностические алгоритмы клинических синдромов, наиболее часто встречающихся в общей врачебной практике. Все алгоритмы представлены на языке блок-схем. Вот примеры:

1. алгоритм «Остро возникшая головная боль»,
2. алгоритм «Хроническая головная боль»,
3. алгоритм «Головокружение»,
4. алгоритм «Синкопальное состояние»,
5. алгоритм «Нарушение остроты зрения»,
6. алгоритм «Красный глаз»,
7. алгоритм «Боль в ухе»,
8. алгоритм «Боль в горле»,
9. алгоритм «Боль в груди»,
10. алгоритм «Боль в суставах»,
11. алгоритм «Острый кашель»,
12. алгоритм «Хронический кашель»,
13. алгоритм «Кровохарканье»,
14. алгоритм «Одышка»,
15. алгоритм «Дисфагия»
16. алгоритм «Тошнота и рвота»;
17. алгоритм «Увеличение живота»;
18. алгоритм «Запор»;
19. алгоритм «Острая диарея»;
20. и т. д.

Сделаем важное пояснение. Хорошо, что авторы медицинских учебников (наконец-то!) стали использовать графический язык для изображения алгоритмов. Но очень плохо, что выбраны давно устаревшие блок-схемы. Авторы учебников должны знать, что **вместо блок-схем следует использовать язык ДРАКОН, который имеет неоспоримые преимущества.**

§7. МОЖНО ЛИ ГОВОРИТЬ О «ВТОРОЙ ГРАМОТНОСТИ»?

Тезис Ершова «Программирование – вторая грамотность» не выдержал испытания временем. Вместе с тем упомянутый доклад Ершова и другие его работы позволяют по-новому взглянуть на алгоритмы и проблему алгоритмического мышления.

Развивая мысль Ершова, следует обратить особое внимание на частично формализованные алгоритмы. И на необходимость развития алгоритмического мышления у обширной группы (90%) людей, соприкасающихся с алгоритмами.

Мы предполагаем, что развитие этого направления со временем позволит сделать первый шаг к решению важной задачи – попытаться воплотить в жизнь идею Андрея Ершова в ее современном звучании: «Алгоритмизация – вторая грамотность».

§8. ВЫВОДЫ

1. Алгоритмы можно разбить на две части:
 - алгоритмы, создаваемые программистами и математиками при разработке программ;
 - частично формализованные алгоритмы, не имеющие отношения ни к программам, ни к компьютерам.
2. Большое значение имеет обучение алгоритмическому мышлению миллионов людей, соприкасающихся с частично формализованными алгоритмами.
3. Обучение программированию является важной задачей. Но эта задача касается сравнительно небольшого числа людей (менее 10%). Обучение программированию не может и не должно быть массовым.
4. Что касается алгоритмизации (понимаемой, как овладение алгоритмическим мышлением), то эта задача может и должна быть предметом *массового* обучения.
5. Сегодня обучение алгоритмизации не рассматривается как самостоятельная задача, необходимая обществу.
6. Принято считать, что алгоритмизация – это часть курса программирования и вне последнего не имеет ценности для общества.
7. Это неверно. Частично формализованные алгоритмы способны принести большую пользу медикам, биологам, работникам государственного, муниципального и корпоративного управления, гуманитариям и многим другим людям, которым не нужно знать программирование, но нужно уметь мыслить алгоритмически.
8. Таким образом, перед системой образования возникает принципиально новая масштабная задача: организовать обучение алгоритмическому мышлению для непрограммистов. То есть для подавляющего большинства специалистов.

КАК ЛИКВИДИРОВАТЬ АЛГОРИТМИЧЕСКУЮ НЕГРАМОТНОСТЬ

§1. ПРОБЕЛ В ЗНАНИЯХ СПЕЦИАЛИСТОВ

Мы живем в обществе знаний. Носителями профессиональных знаний являются специалисты, например, агрономы, нефтяники, микробиологи, технологи, экономисты, химики, врачи.

Парадокс в том, что эти люди прекрасно знают свою работу. Они четко знают последовательность выполняемых ими действий, то есть алгоритмы своей работы. А также, что очень важно, алгоритмы предметной области.

Но они, увы, не знают, что их знания называются алгоритмами. Эти замечательные специалисты не умеют описывать алгоритмы. Они не умеют расчленять свои знания и выделять среди них алгоритмическую (процедурную) часть. Они не в состоянии выразить свои знания на бумаге в форме алгоритмов.

Хорошо ли это? Допустима ли подобная алгоритмическая неосведомленность?

§2. УСТАРЕВШАЯ ТОЧКА ЗРЕНИЯ

На этот вопрос обычно отвечают так:

Конечно, хорошо. Потому что существует разделение труда. Каждый должен заниматься своим делом. Алгоритмы – дело математиков и программистов. А нефтяники, химики, врачи и экономисты совершенно не обязаны знать алгоритмические тонкости. У них, слава богу, своих забот хоть отбавляй!

Данная точка зрения требует непредвзятого анализа. Прежде всего, надо ответить на вопрос. В чем причина повсеместно распространенной

алгоритмической неграмотности? Неграмотности подавляющего большинства специалистов?

§3. НЕОПРАВДАННО БОЛЬШИЕ ТРУДОЗАТРАТЫ

Суть в том, что нынешние алгоритмы (используемые во всем мире) имеют серьезный недостаток. Они *чрезвычайно трудны для понимания*.

Поэтому существующая практика изучения, разработки и эксплуатации алгоритмов является неудовлетворительной. Она требует слишком больших затрат труда и времени. Эти трудозатраты настолько велики, что во много раз превышают реальные резервы времени, которыми располагают люди.

Алгоритмическая неграмотность объясняется тем, что в современных условиях изучение алгоритмов является слишком сложным и зачастую непосильным делом. Поэтому работа с алгоритмами для подавляющего большинства профессионалов оказывается невозможной. Данное обстоятельство ставит непреодолимый барьер и не позволяет перейти к массовому овладению алгоритмизацией.

Это плохо. Алгоритмическая неграмотность многих умных людей неблагоприятно отражается на развитии общества.

§4. ПОЧЕМУ АЛГОРИТМЫ ТРУДНЫ ДЛЯ ПОНИМАНИЯ?

Что такое понятность алгоритма? В главе 4 было дано определение.

Понятность алгоритма – свойство алгоритма минимизировать интеллектуальные усилия, необходимые для его понимания человеком при зрительном восприятии текста алгоритма.

Как увеличить понятность алгоритма? Какая научная дисциплина может подсказать ответ на этот вопрос? Это – психология. Если точнее – эргономика. А еще точнее – когнитивная эргономика.

Применительно к данному случаю когнитивную эргономику можно определить так. Это наука о том, как облегчить и улучшить умственную работу при зрительном восприятии искусственных зрительных сцен, передающих человеку интеллектуальное содержание. Например, драконсхем. Когнитивная эргономика позволяет создавать эргономичные алгоритмические языки и эргономичные алгоритмы.

На заре развития эргономики вместо термина «эргономичные средства» говорили «средства художественного конструирования». Вспомним еще раз мудрые слова одного из пионеров эргономики, основателя Института психологии РАН 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, об-

легчаем 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].

Теперь мы можем ответить на вопрос: «почему алгоритмы трудны для понимания?». Потому что при создании алгоритмов используются *неэргономичные* алгоритмические языки.

§5. АЛГОРИТМИЗАЦИЯ – ЭТО ТРУД

Разработка алгоритмов – это труд, производительность которого играет важную роль. Если труд слишком сложен (производительность труда мала), то алгоритмизацию могут выполнять только опытные специалисты. При таких условиях массовая работа с алгоритмами невозможна.

И наоборот, если данный труд удастся резко облегчить, алгоритмизация станет посильной почти для каждого. В этом случае создаются необходимые предпосылки для массового овладения методами алгоритмизации.

§6. АЛГОРИТМИЧЕСКОЕ МЫШЛЕНИЕ И ЯЗЫК ДРАКОН

Многие группы учащихся по разным причинам не знакомы с алгоритмами и не умеют с ними работать. Они не могут быстро и безошибочно записывать свои знания в форме алгоритмов.

Этот недостаток можно устранить. Алгоритмы можно и нужно делать дружелюбными с помощью когнитивно-эргономических методов.

Алгоритмический язык ДРАКОН построен в строгом соответствии с рекомендациями когнитивной эргономики. Благодаря этому ДРАКОН кардинально облегчает труд алгоритмизации и повышает его производительность.

Вследствие этого трудоемкость изучения, разработки и понимания алгоритмов значительно уменьшается. Алгоритмическое мышление становится доступным для всех желающих. Учащиеся приобретают необходимые знания и умения, позволяющие писать, читать, понимать и использовать алгоритмы.

§7. ВЫВОДЫ

1. Современные алгоритмы, используемые во всем мире, чрезвычайно трудны для понимания.
2. Существующая практика изучения, разработки и эксплуатации алгоритмов требует неоправданно больших затрат труда и времени.
3. Эти трудозатраты настолько велики, что во много раз превышают возможности людей.
4. Алгоритмическая неграмотность работников объясняется тем, что изучение алгоритмов является слишком сложным делом. Поэтому работа с алгоритмами для подавляющего большинства профессионалов оказывается невозможной.
5. Чтобы поправить дело, необходимо существенно облегчить работу с алгоритмами с помощью когнитивно-эргономических методов. В результате непосильная работа может стать посильной.
6. Алгоритмический язык ДРАКОН построен в соответствии с рекомендациями когнитивной эргономики.
7. Можно предположить, что язык ДРАКОН окажется эффективным средством, позволяющим (в сочетании с другими методами) в разумные сроки устранить алгоритмическую неграмотность.

НЕОБХОДИМОСТЬ КУЛЬТУРНЫХ ИЗМЕНЕНИЙ

§1. НА ПОРОГЕ АЛГОРИТМИЧЕСКОЙ КУЛЬТУРНОЙ РЕВОЛЮЦИИ

На нашей планете работают десятки или, возможно, сотни миллионов людей, которые оказались в парадоксальном положении. С одной стороны, эти люди не обладают знаниями об алгоритмах. С другой стороны, они постоянно имеют дело с алгоритмами, не догадываясь об этом.

Такое положение дел неприемлемо и нуждается в исправлении. Нельзя скрывать от миллионов людей, что их работа связана с алгоритмами. Нужно познакомить их с удивительными (для них) достижениями человеческого разума. Нужно объяснить этим уважаемым людям, что их работа имеет теснейшую связь с алгоритмами.

Таким образом, десяткам или сотням миллионов людей надо привить навыки алгоритмического мышления. Это большая работа. Работа по развитию мышления.

Поскольку речь идет о крупномасштабном преобразовании мышления, мы вправе назвать грядущее преобразование культурной революцией. Точнее говоря, *алгоритмической культурной революцией*.

Разумеется, алгоритмическая революция должна иметь предпосылки. Она должна иметь веские основания. Должны возникнуть условия, которые позволят осуществить культурную революцию на практике.

Что это за условия? Главным условием, главной предпосылкой алгоритмической революции является тот факт, что появился прямой и широкий путь, позволяющий всем (или почти всем) желающим овладеть алгоритмическим мышлением. Причем сделать это в разумные и реальные сроки.

Раньше такого пути не было. Путь к сокровищнице алгоритмических знаний был закрыт для подавляющего большинства людей. Новые возможности открылись тогда, когда алгоритмизация стала опираться не только на математику, но и на когнитивную эргономику. Последняя сделала алгоритмы дружелюбными, человеческими, понятными для «народа».

§2. ТЕХНОЛОГИ И ТЕХНОЛОГИЧЕСКИЕ АЛГОРИТМЫ

В главе 40 шла речь о непрограммирующих специалистах (медиках, биологах, чиновниках, экономистах, нефтяниках, энергетиках и многих других). Обозначим этих специалистов термином «технологи».

Для нас важно, что «технологи»:

- прекрасно знают свою работу;
- знают последовательность выполняемых ими действий, то есть алгоритмы своей работы. А также алгоритмы предметной области;
- не обладают знаниями об алгоритмах;
- не умеют анализировать свои знания и выделять среди них алгоритмическую часть;
- не умеют выразить свои знания на бумаге в форме алгоритмов.

Назовем указанные алгоритмы (то есть алгоритмы, с которыми соприкасаются «технологи») *технологическими*. Здесь уместно вспомнить частично формализованные алгоритмы. Частичная формализация означает, что графический синтаксис языка является математически строгим. А текстовые надписи, расположенные внутри или снаружи икон, пишут на естественном языке. Это очень удобно для читателей.

§3. ЧТО ТАКОЕ ТЕХНОЛОГИЧЕСКИЙ ЯЗЫК?

Мы исходим из того, что технологические алгоритмы являются частным случаем общего понятия «алгоритм».

Сходство понятий «алгоритм» и «технологический алгоритм» имеет большое значение. К сожалению, это сходство до сих пор не привлекало к себе должного внимания, что в немалой степени способствовало разделению науки на изолированные «клетки», создавая неоправданные препятствия для межотраслевых и междисциплинарных контактов.

Сегодня программисты и «технологи» (в широком смысле слова, включая агрономов, медиков, педагогов, управленцев и т. д.) – это разные «касты», которые получают разное образование и говорят на разных профессиональных языках. Подобные барьеры сильно затрудняют взаимопонимание между специалистами.

Названный недостаток (трудности взаимопонимания) можно ослабить или устранить, создав язык, одинаково удобный для «технологов»,

программистов и других специалистов. Для обозначения этого языка предлагается термин *техноязык* (технологический язык).

Первым кандидатом на роль техноязыка является ДРАКОН.

§4. ТЕХНОЯЗЫК – ЯЗЫК НОВОГО ТИПА

Техноязык имеет двойное назначение. С одной стороны, он дает возможность (как и любой другой алгоритмический язык) проектировать алгоритмы. С другой стороны, он позволяет стандартизовать запись технологических алгоритмов (технологических процессов) любой природы в любой предметной области. Причем делать это таким образом, что стандартная запись оказывается, во-первых, более строгой, свободной от пробелов и двусмысленностей, во-вторых, более наглядной, доходчивой и очень удобной для читателя.

Таким образом, *техноязык* – это язык нового типа, который сочетает математическую строгость алгоритмического языка с эргономичностью языка межатраслевого и междисциплинарного общения, пригодного для наглядного описания технологий и взаимопонимания между специалистами.

Уточним понятие. Техноязык – общий термин, обозначающий все языки ДРАКОН-семейства, в том числе:

- гибридные языки программирования (см. главу 16);
- частично формализованные языки (см. главу 39).

Все дракон-языки имеют единый графический синтаксис. Это обеспечивает зрительное сходство дракон-схем различных дракон-языков. Каждый язык дракон-семейства отличается тем, что имеет свой собственный текстовый синтаксис. Разнообразие текстовых синтаксисов обеспечивает богатство выразительных средств.

Подведем итоги. В настоящее время отсутствует простой и удобный язык, рассчитанный на огромную армию непрограммистов и программистов, улучшающий работу ума и облегчающий понимание и взаимопонимание. Техноязык позволяет заполнить этот пробел.

§5. ДЕКЛАРАТИВНЫЕ ЗНАНИЯ

Человеческие знания, выраженные с помощью письменного языка, можно разбить на две части: технологические (процедурные, алгоритмические) и декларативные (см. главу 2).

Для изложения декларативных знаний представители разных профессий используют различные декларативные языки, в том числе графические. Например, декларативные знания конструктора выражаются на языке конструкторских чертежей, электрика – на языке электрических схем, географа – на языке географических карт.

§6. МОЖНО ЛИ СОЗДАТЬ УНИВЕРСАЛЬНЫЙ ТЕХНОЯЗЫК?

Большой интерес представляет вопрос: можно ли создать универсальный язык представления профессиональных знаний, удобный для специалистов любой профессии и позволяющий улучшить взаимопонимание между ними?

Для декларативных знаний ответ, очевидно, будет отрицательным. Потому что нельзя скрестить ужа с ежом и придумать разумный и полезный гибрид электрической схемы и географической карты (или конструкторского чертежа). Такой путь неизбежно ведет в тупик.

Поэтому придется смириться с выводом, что специалисты разных профессий будут и впредь использовать множество самых разнообразных декларативных языков. Унификация здесь невозможна.

В отличие от декларативных знаний технологические (алгоритмические, императивные, процедурные) знания специалистов любого профиля имеют в точности одинаковую структуру, которая несколько не зависит от конкретной специальности и предметной области.

Отсюда проистекает важный вывод:

Для отображения любых технологических (алгоритмических) знаний можно использовать один и тот же техноязык, общий для всех научных и учебных дисциплин.

§7. АРГУМЕНТЫ В ЗАЩИТУ ТЕХНОЯЗЫКА

1. Техноязык позволяет выражать любые технологические (алгоритмические) знания в любой предметной области в *единой стандартной графической форме*. Он имеет стандартный графический синтаксис.

2. Техноязык может играть роль междотраслевого и междисциплинарного языка, содействующего решению важнейшей проблемы – проблемы взаимопонимания между учащимися, специалистами и учеными.

3. Технологические знания (то есть средства для описания структуры деятельности) играют особую роль в человеческой жизни. В самом деле, человек – деятельное существо. От рождения до смерти он непрерывно действует. Деятельность выражает сущность жизни. Бездеятельность – это смерть. Поэтому знания о структуре деятельности (технологические знания) составляют важный компонент человеческих знаний.

Можно предположить, что в системе человеческих знаний технологические знания играют роль несущей конструкции или каркаса, который скрепляет между собой (склеивает, цементирует) отдельные фрагменты декларативных знаний. Сказанное хорошо согласуется с известным мнением, согласно которому

«большинство знаний об окружающем мире можно выразить в виде процедур или последовательности действий, направленных на достижение конкретных целей» [1].

§8. ЯЗЫК ДЛЯ ВЗАИМОПОНИМАНИЯ ОЧЕНЬ НУЖЕН

Социально-экономические успехи общества сильно зависят от разработки и внедрения новых технологий (новых алгоритмов). Между тем способы описания структуры деятельности и новых технологических процессов (алгоритмов), используемые специалистами-технологами, недостаточно формализованы и слабо используют опыт, накопленный в алгоритмизации и программировании.

С другой стороны, многие математики, алгоритмисты и программисты испытывают серьезные затруднения при необходимости наглядно и доходчиво описать сущность, структуру и содержание предлагаемых математических решений, алгоритмов и создаваемых программных комплексов и передать соответствующие знания другим людям.

В обоих случаях причиной этого изъяна является отсутствие необходимых средств взаимопонимания. В данном случае под средством взаимопонимания понимается техноязык. Он может обеспечить высокоэффективное интеллектуальное взаимопонимание и производственное взаимодействие между людьми, что особенно важно, в частности, при создании крупномасштабных проектов.

Нужда в таком языке весьма велика.

Изложенные соображения позволяют сделать два вывода.

- Создание техноязыка является осуществимой задачей.
- Искомый язык следует строить не как язык отображения декларативных знаний, а именно как техноязык, то есть язык технологических (алгоритмических) профессиональных знаний.

Содержание книги свидетельствует, что все описанные качества приписуи языку ДРАКОН.

§9. НЕОБХОДИМОСТЬ КУЛЬТУРНЫХ ИЗМЕНЕНИЙ

Современные компьютерные языки представляют собой замечательное достижение и эффектный прорыв в будущее. Вместе с тем для них характерна определенная ограниченность, так как в их задачу не входит создание подлинно «народного» межатраслевого языка (техноязыка), предназначенного:

- для интеллектуального взаимопонимания и взаимодействия специалистов разных профессий;
- для улучшения работы ума;
- для эффективного описания структуры деятельности;
- для проектирования технологических алгоритмов и социальных технологий;
- для мощного роста производительности в процессе формализации алгоритмических профессиональных знаний.

На наш взгляд, техноязык должен повлечь за собой определенные изменения в культуре, иначе говоря – стать *новым элементом языковой культуры*.

Представляется уместной следующая аналогия. Несколько столетий назад развитие машиностроения и строительного дела сдерживалось наряду с другими причинами отсутствием языка, позволяющего эффективно фиксировать необходимые знания. Общественная потребность в таком языке была чрезвычайно велика.

Когда французский математик Гаспар Монж впервые предложил идеи начертательной геометрии и появились три проекции (фасад, план, профиль), это привело к формированию, закреплению в стандартах и всемирному распространению языка конструкторских и строительных чертежей. Последний стал одним из важнейших языковых средств технической цивилизации и одновременно достоянием мировой культуры.

Человечество получило столь необходимый и давно ожидаемый языковой инструмент для фиксации и обогащения соответствующих знаний. В итоге развитие промышленности заметно ускорилось.

Думается, сегодня имеет место примерно такая же ситуация, ибо сформировалась общественная потребность в техноязыке, решающего перечисленные выше задачи.

Известно, что

«многие люди с активным умом и блестящими идеями оказываются не в состоянии заставить своих коллег точно понять, что они имеют в виду», причем подобная ситуация, вызванная неадекватностью используемых языковых средств, является одной из причин недостаточной восприимчивости современной индустрии к новым идеям [2].

Эти и другие соображения свидетельствуют о том, что для улучшения взаимопонимания необходимо провести отнюдь не простые системные изменения в культуре. По-видимому, в качестве одного из первых шагов целесообразно сделать техноязык частью системы образования в школе и вузе. Причем не как факультатив, а как обязательный компонент учебной программы.

§10. ВЫВОДЫ

Ниже излагаются тезисы о возможной роли языка ДРАКОН в человеческой культуре.

1. Языки представления декларативных знаний не поддаются унификации. С технологическими (алгоритмическими, императивными, процедурными) знаниями ситуация иная. Существует возможность построить универсальный технологический язык – *техноязык*.

2. В настоящее время ощущается острая необходимость в высокоточном средстве для улучшения интеллектуального взаимодействия между людьми – техноязыке.
3. Техноязык может стать новым элементом языковой культуры человечества.
4. Ближайшими аналогами языка ДРАКОН служат блок-схемы алгоритмов, диаграммы поведения языка *UML* и псевдокод. Однако они обладают серьезными недостатками и по своим эргономическим и дидактическим характеристикам значительно уступают языку ДРАКОН. Поэтому в качестве техноязыка и нового элемента культуры следует использовать ДРАКОН, а не указанные схемы.
5. Ставка на язык ДРАКОН является оправданной, так как она позволяет осуществить системные изменения в культуре: в системе образования, на производстве (программы и технологии), в науке (улучшение общения между учеными), в социальных технологиях и других областях.
6. Изучение техноязыка ДРАКОН должно быть предусмотрено в средней и высшей школе. Для этого техноязык нужно специально приспособить, сделать легким, доступным для школьников и студентов.

АЛГОРИТМЫ ДОЛЖНЫ БЫТЬ ПОНЯТНЫМИ (вместо заключения)

ЗАЧЕМ НАПИСАНА ЭТА КНИГА?

В этой книге мы попытались:

- провести четкую грань между алгоритмизацией и программированием;
- сосредоточить внимание на алгоритмах, оставив программирование за рамками книги;
- изложить основы алгоритмизации;
- создать средства, обеспечивающие максимально возможную понятность алгоритмов. И за счет этого *сделать алгоритмы доступными для «народа»*.

КРИТИКА ТРАДИЦИОННЫХ ПОДХОДОВ

Прежние способы записи алгоритмов устарели. Они слишком трудны для понимания и требуют неоправданно больших трудозатрат.

Древние, но живучие привычки ставят непреодолимый барьер для большинства людей, которые хотят научиться выражать свои знания, мысли и планы в форме алгоритмов.

Традиционные формы представления алгоритмов отжили свой век и должны сойти со сцены. Именно они несут ответственность за господствующую на нашей планете алгоритмическую неграмотность.

КАКИЕ РЕЗУЛЬТАТЫ ПОЛУЧЕНЫ?

- Предложен новый способ записи алгоритмов – *дракон-схемы*.
- Благодаря этому новшеству алгоритмы становятся значительно более понятными, общедоступными, кристально ясными.

- Использование дракон-схем позволяет повысить производительность труда при разработке, анализе и проверке алгоритмов (возможно, в несколько раз).
- Дракон-схемы облегчают и ускоряют обучение алгоритмизации.
- Новый способ записи дает возможность коренным образом изменить систему образования в области алгоритмизации. И познакомиться с алгоритмами более широкие слои населения;

Можно предположить, что внедрение дракон-схем в массовую практику поможет обеспечить ликвидацию алгоритмической неграмотности.

ПОНЯТНОСТЬ АЛГОРИТМОВ

При разработке языков для записи алгоритмов (алгоритмических языков) обычно выдвигается ряд требований. К сожалению, среди них, как правило, отсутствует самое важное для человека:

«Алгоритмы, записываемые на алгоритмическом языке, должны быть понятны для человеческого зрительного восприятия и удобны для человеческого мышления».

Слово «понятны» следует пояснить. Нужны не просто понятные, а *в высшей степени* понятные алгоритмы. Это значит, что должен выполняться принцип: «Взглянул – и сразу понял!», «Посмотрел – и мигом во всем разобрался!».

С учетом этих пояснений вводится термин «критерий сверхвысокой понятности».

Отличие языка ДРАКОН состоит в том, что язык должен удовлетворять данному критерию. Это значит, что требование понятности алгоритмов рассматривается как *главное, приоритетное, наиболее важное требование к языку*.

Чтобы выполнить указанное требование, одной математики мало. Наряду с математикой, необходимо использовать когнитивную эргономику.

КОГНИТИВНАЯ ЭРГОНОМИКА

Язык ДРАКОН имеет две опоры. Первая – математика. Вторая – психология, точнее, когнитивная эргономика. Именно эргономика позволяет сделать дракон-схемы изящными и доступными. При создании ДРАКОНа был использован научный подход к эргономизации конструкций языка. Такой подход позволил улучшить визуальные образы языка (визуальные формы фиксации знаний), согласовав их с тонкими характеристиками

глаза и мозга. Тонкими, но хорошо известными в когнитивной эргономике, психофизиологии, нейробиологии.

Когнитивная эргономика позволила преобразовать неудобные и устаревшие блок-схемы в элегантные очертания приятных и доходчивых дракон-схем.

С появлением дракон-схем разработка алгоритмов существенно облегчается.

ДРАКОН – качественно новый этап работы с алгоритмами.

СТАНЕТ ЛИ ДРАКОН ЧЕМПИОНОМ МИРА ПО КРИТЕРИЮ «ПОНЯТНОСТЬ АЛГОРИТМОВ»?

Претензия ДРАКОНа на «мировое господство» ограничена. Он вступает в конкурентную борьбу только с императивными и процедурными языками (точнее, с императивно-процедурными частями языков). И только в том случае, когда *понятность алгоритмов* является главным требованием к языку. Тем, кто желает писать непонятные или трудные для понимания алгоритмы, ДРАКОН не нужен.

Требование удобопонятности алгоритмов все чаще выходит на передний план. Поэтому шансы ДРАКОНа на победу в конкурентной борьбе с другими языками растут.

ДРАКОН-КОНСТРУКТОР

Дракон-конструктор – верный слуга алгоритмиста. Эта компьютерная программа способна оказать человеку огромную помощь при создании алгоритмов.

Внутри программы спрятана сложная математика (исчисление икон и др.), но прелесть в том, что пользователю знать эту математику не нужно. Чтобы обеспечить максимальные удобства для человека, большинство функций по созданию алгоритмов (кроме творческих операций) берет на себя дракон-конструктор.

Кроме того, дракон-конструктор осуществляет автоматическое доказательство правильности дракон-схем, гарантируя принципиальную невозможность ошибок визуального синтаксиса.

Безошибочное проектирование графики дракон-схем – важное преимущество, повышающее производительность труда при практической работе.

ГДЕ СКАЧАТЬ ДРАКОН-КОНСТРУКТОР?

Ответ дан на стр. 414.

Не исключено, что к моменту выхода этой книги на сайтах «Визуальный язык ДРАКОН» <http://drakon.su/>, «Алгоритмический язык ДРАКОН» <http://drakon-practic.ru/> и форуме <http://forum.oberoncore.ru/viewforum.php?f=77> появится новая информация о разработке общедоступных инструментальных программ языка ДРАКОН.

См. также электронную энциклопедию «Википедия», статья «ДРАКОН (алгоритмический язык)».

Как связаться с автором?
Электронная почта: vdp2007@bk.ru
Тел. 8 (495) 331-50-72
8 (495) 535-34-13

ЛИТЕРАТУРА

Введение (с. 6–12)

1. Робертсон Л. А. Программирование – это просто. Пошаговый подход. Перевод с 4-го английского издания. М.: БИНОМ Лаборатория знаний, 2008. – 383 с.
2. Канцедал С. А. Алгоритмизация и программирование. Учебное пособие. М.: Форум, Инфра-М, 2008. – 352 с.
3. Князева М. Д. Алгоритмика: от алгоритма к программе. Учебное пособие. М.: Кудиц-Образ, 2006. – 192 с.
4. International Organization for Standardization, ISO 5807:1985 Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts.
5. ГОСТ 19.701–90. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. М.: Изд-во стандартов, 1991. – 26 с.
6. Паронджанов В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! М.: Дело, 2001. – 360с.
7. Паронджанов В. Д. Дружелюбные алгоритмы, понятные каждому. Как улучшить работу ума без лишних хлопот. М.: ДМК Пресс, 2010. – 464 с.
8. Примерная программа дисциплины «Информатика». Издание официальное. – М.: Госкомвуз, 1996. – С. 3, 4, 15, 16.

Глава 7 (с. 78–112)

1. Йодан Э. Структурное проектирование и конструирование программ. М.: Мир, 1979. – С. 192–196, 226.

Глава 11 (с. 153–183)

1. Йодан Э. Структурное проектирование и конструирование программ. М.: Мир, 1979. – С. 252.

Глава 12 (с. 184–204)

1. *Вайсер М., Шнейдерман Б.* Человеческий фактор в программировании для ЭВМ. – В кн.: Человеческий фактор. В 6-и т. Т. 6. Эргономика в автоматизированных системах. М.: Мир, 1992. – С. 15.
2. *Ван Тассел Д.* Стил, разработка, эффективность, отладка и испытание программ. М.: Мир, 1981. – С. 18.

Глава 15 (с. 242–254)

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. 9.
3. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию. М.: Мир, 1980. – С. 80.
4. ГОСТ 19.701–90. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. М.: Изд. стандартов, 1991.
5. *Криницкий Н. А.* Алгоритмы вокруг нас. М.: Наука, 1984. – С. 102.
6. *Питерс Л. Дж.* Методы отображения и компоновки программных средств // ТИИЭР. 1980. Т. 68, №9. – С. 60.
7. *Семакин И. Г.* Основы алгоритмизации и программирования. М.: ИЦ Академия, 2008. – С. 32, рис. 113в.
8. *Дайтибегов Д. М., Черноусов Е. А.* Основы алгоритмизации и алгоритмические языки. М.: Финансы и статистика, 1992. – С. 81.

ГЛАВА 16 (с. 255–266)

1. Практический вывод по результатам эксплуатации системы ИС Дракон – Транслятор Дракон-Си – Keil. <http://forum.oberoncore.ru/viewtopic.php?p=68616#p68616>.
2. *Приклонский П.* Транслятор файла *.drt ИС Дракон в текст Си-программ. <http://forum.oberoncore.ru/viewtopic.php?f=79&t=2718>.

Глава 17 (с. 269–276)

1. Руководство по профилактической медицине. М.: Новая слобода, 1993. – С. 40.
2. Практическое руководство для врачей общей (семейной) практики. / Под редакцией академика РАМН И.Н. Денисова. М.: Геотар-Мед, 2001. – С. 501–504.

Глава 18 (с. 277–285)

1. Математическая энциклопедия. М.: Сов. энциклопедия, 1977. Т. 1. – С. 202–210.

Глава 22 (с. 312–324)

1. Сонин Н. И., Сапин М. Р. Биология. Человек. Учебник для 8 класса средней школы. М.: Дрофа, 2004. – С. 130.
2. Кемп П., Арнс К. Введение в биологию. М.: Мир, 1988. – С. 612, 613.
3. Пташине М. Переключение генов. Регуляция генной активности и фаг λ . М.: Мир, 1989. – С. 20.

Глава 23 (с. 325–330)

1. Марценюк В. Б. Книга, которая учит, как вырастить помидоры. Учебное пособие. Пермь: ПС Гармония, 2002. – С. 7–21.

Глава 25 (с. 344–348)

1. Паронджанов В. Д. Почему мудрец похож на обезьяну, или Парадоксальная энциклопедия современной мудрости. М.: РИПОЛ классик, 2007. – 1154 с.
2. Паронджанов В. Д. Технология структурирования алгоритмов выработки управленческих решений (САВУР-технология) и ее применение в системах организационного управления. – В кн.: Информационно-аналитическое обеспечение стратегического управления: теория и практика. Труды второй всероссийской научно-практической конференции 19–20 мая 2005 г. М.: ИПКгосслужбы, ИНИОН РАН, 2006. – С. 315–319.
3. Павлова Н. Ф. Стратегическое планирование развития территориальных социальных образований в схемах. Екатеринбург, Уральское отделение РАН, 2002. – 119 с.

Глава 34 (с. 427–435)

1. Ершов Ю. Л., Палютин Е. А. Математическая логика. М.: Наука, 1979. – С. 12, 13.
2. Вельбицкий И. В. Алгебра конструирования алгоритмов и программ // Управляющие системы и машины. 1987, №6. – С. 100.
3. Клини С. К. Введение в метаматерику. М.: ИЛ, 1957. – С. 59–61.
4. Колеватов В. А. Социальная память и познание. М.: Мысль, 1984. – С. 133.
5. Зенкин А. А. Когнитивная компьютерная графика. М.: Наука, 1991.
6. Кондаков Н. И. Логический словарь-справочник. М.: Наука, 1976. – С. 101, 285.

7. *Поспелов Г. С.* Искусственный интеллект основа новой информационной технологии. М.: Наука, 1988. – С. 41.
8. *Андерсон Р.* Доказательство правильности программ. М.: Мир, 1988. – С. 152.
9. *Грис Д.* Наука программирования. М.: Мир, 1984. – С. 303.

Глава 35 (с. 436–448)

1. *Aschcroft E, Manna Z.* The Translation of «Goto» Programs into «While» Programs. Proceedings of 1971 IFIP Congress.
2. *Майерс Г.* Надежность программного обеспечения. М.: Мир, 1980. – С. 137.
3. *Йодан Э.* Структурное проектирование и конструирование программ. М.: Мир, 1979. – С. 192–196.

Глава 36 (с. 449–472)

1. *Ермаков И. Е., Жигуненко Н. А.* Двумерное структурное программирование; класс устремлённых графов (теоретические изыскания из опыта языка «Дракон») // Сборник докладов конференции «Современные информационные технологии и ИТ-образование». V Международная научно-практическая конференция. 8–10 ноября 2010 г. Москва, МГУ.
2. *Дейкстра Э.* Заметки по структурному программированию. – В кн.: Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975. – С. 7–97.
3. Толковый словарь по вычислительным системам / Под ред. В. Иллингуорта и др.: Перевод с англ. М.: Машиностроение, 1991. – 560 с.
4. *Дейкстра Э.* Дисциплина программирования. М.: Мир, 1978. – 275 с.
5. *Bohm C., Jacopini G.* Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules // Communications of the ACM. 1965. Vol. 9, №5. – P. 366–371.
6. *Knuth D.* Structured Programming with GOTO Statements // Computing Survys, 6 (4), 1974 – P. 261–301.
7. *Ван Тассел Д.* Стиль, разработка, эффективность, отладка и испытание программ. М.: Мир, 1981. – С. 89.
8. *Майерс Г.* Надежность программного обеспечения. М.: Мир, 1980. – 360 с.
9. *Кун Т.* Структура научных революций. М.: АСТ, 2003. – 608 с.
10. *Очков В. Ф., Пухначев Ю. В.* 128 советов начинающему программисту. М.: Энергоатомиздат, 1992. – С. 21.
11. *Лингер Р., Миллс Х., Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982. – 406 с.

Глава 37 (с. 475–479)

1. Живой Журнал. Блог meatreach. 2007-02-12: Как заварить чашку чая? <http://meatreach.livejournal.com/120342.html>

Глава 38 (с. 480–487)

1. *Паронджанов В. Д.* Почему мудрец похож на обезьяну, или Парадоксальная энциклопедия современной мудрости. М.: РИПОЛ Классик, 2007. – 1154 с.
2. *Паронджанов В. Д.* Как улучшить работу ума. Алгоритмы без программистов – это очень просто! М.: Дело, 2001. – 360 с.
3. *Паронджанов В. Д.* Как улучшить работу ума. (Новые средства для образного представления знаний, развития интеллекта и взаимопонимания). М.: Радио и связь, 1998, 1999. – 352 с.
4. *Паронджанов В. Д.* Кризис цивилизации и нерешенные проблемы информатизации // Научно-техническая информация. Серия 2. 1993. №12. – С. 1–9.
5. *Болошин И. А.* Еще раз о кризисе цивилизации и нерешенных проблемах информатизации // Научно-техническая информация. Серия 2. 1994. №9 – С. 32.
6. *Сергеев Б. Ф.* Ступени эволюции интеллекта. Л.: Наука, 1986. – С. 179.
7. *Дейкстра Э.* Дисциплина программирования. М.: Мир, 1972. – С. 265, 266.
8. *Шнейдерман Б.* Психология программирования. Человеческие факторы в вычислительных и информационных системах. М.: Радио и связь, 1984. – С. 11.
9. Психологические проблемы автоматизации научно-исследовательских работ. М.: Наука, 1987. – С. 10.
10. *Ершов А. П.* О человеческом и эстетическом факторах в программировании // Информатика и образование, 1993, №6. – С. 7.
11. *Перминов О. Н.* Программирование на языке Паскаль. М.: Радио и связь, 1988. – С. 4, 5.
12. *Айверсон К. Е.* Нотация как средство мышления // Лекции лауреатов премии Тьюринга. М.: Мир, 1993. – С. 392, 393.

Глава 39 (с. 488–492)

1. Архив академика А. П. Ершова. Программирование – вторая грамотность. Предисловие архивариуса. http://ershov.iis.nsk.su/russian/second_literacy/pred.html
2. Ершов А. П. Программирование – вторая грамотность. Доклад на 3-й Всемирной конференции ИФИП и ЮНЕСКО по применению ЭВМ в обучении, 1981, Лозанна (Швейцария).

http://ershov.iis.nsk.su/russian/second_literacy/article.html

3. Нариньяни А.С. Между эволюцией и сверхвысокими технологиями: новый человек ближайшего будущего // Вопросы философии, 2008, №4. – С. 5.

Глава 40 (с. 493–496)

1. Ломов Б. Ф. Эргономические (инженерно-психологические) факторы художественного конструирования. – В кн.: Учебно-методические материалы по художественному конструированию. М., 1965.

Глава 41 (с. 497–503)

1. Джордж М. П., Лэнски Э. Л. Процедурные знания // ТИИЭР. 1986. Т. 74. № 10. – С. 101.
2. Woodward J. F. Science in Industry: Science of Industry. An Introduction to the Management of Technology-Based Industry. Aberdeen: Aberdeen University Press, 1982. – P. 13.

ОСНОВНАЯ ЛИТЕРАТУРА ПО ЯЗЫКУ ДРАКОН

(в хронологическом порядке)

1. Паронджанов В. Д. Графический синтаксис языка ДРАКОН // Программирование, 1995, №3. – С. 45–62. (*Первая публикация о языке ДРАКОН в журнале Российской академии наук*).
2. Паронджанов В. Д. Как улучшить работу ума. (Новые средства для образного представления знаний, развития интеллекта и взаимопонимания). М.: Радио и связь, 1998, 1999. – 352 с., ил.: 154. (*Первая книга по языку ДРАКОН*).
3. Паронджанов В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! М.: Дело, 2001. – 360 с., ил. 154.
http://arbinada.com/main/system/files/Parondzhanov.Kak_uluchsit_rabotu_uma.pdf
4. Паронджанов В. Д. Почему мудрец похож на обезьяну, или Парадоксальная энциклопедия современной мудрости. М.: РИПОЛ Классик, 2007. – 1154 с. ил. 245. (*Дано научно-популярное описание языка ДРАКОН. – С. 297–434*).
5. Паронджанов В. Д. Язык ДРАКОН. Краткое описание. М.: 2009. – 124 с. (*Краткое описание позволяет быстро ознакомиться с основными идеями языка ДРАКОН*).
<http://drakon.su/biblioteka/start>
<http://drakon-practic.ru/dragon.pdf>
6. Паронджанов В. Д. Дружелюбные алгоритмы, понятные каждому (Как улучшить работу ума без лишних хлопот). М.: ДМК-пресс, 2010. – 464 с. ил. 233.
7. Паронджанов В. Д. Учись писать, читать и понимать алгоритмы. Алгоритмы для правильного мышления. Основы алгоритмизации. – М.: ДМК Пресс, 2012. – 520 с. Иллюстраций: 272.
8. Статья «ДРАКОН (алгоритмический язык)» в электронной энциклопедии «Википедия».

ЛИТЕРАТУРА О ЯЗЫКЕ ДРАКОН ДЛЯ СИСТЕМЫ ОБРАЗОВАНИЯ

9. Примерная программа дисциплины «Информатика». Издание официальное. М.: Госкомвуз, 1996. 21 с. / Авторы программы: Кузнецов В. С., Падалко С. Н., Паронджанов В. Д., Ульянов С. А. Научные редакторы: Падалко С. Н., Паронджанов В. Д. (*Это официальный документ Государственного комитета по высшему образованию Российской Федерации, который официально «узаконил» язык ДРАКОН*. См. – С. 3, 4, 15, 16).
10. Паронджанов В. Д. Каким будет школьный алгоритмический язык XXI века? // Информатика и образование, 1994, №3. – С. 77–92.
11. Паронджанов В. Д. Возможна ли новая революция в образовании? // Высшее образование в России, 1997, №2. – С. 1–11.
12. Паронджанов В. Д. Занимательная информатика. М.: Росмэн, 1998. – 152 с. Иллюстраций: 200. (*Язык ДРАКОН для школьников*).
13. Паронджанов В. Д. Занимательная информатика. М.: Росмэн, 2000. – 160 с. Иллюстраций: 207. (*Язык ДРАКОН для школьников*).
14. Паронджанов В. Д. Занимательная информатика. М.: Дрофа, 2007. – 192 с. Иллюстраций: 240. (*Пособие по теме «Алгоритмы» для учащихся 5–9 классов, построенное на основе языка ДРАКОН*).

ПРИМЕНЕНИЕ ЯЗЫКА ДРАКОН В РАКЕТНО-КОСМИЧЕСКОЙ ОТРАСЛИ

Язык ДРАКОН успешно используется во многих космических программах:

- разгонный блок космических аппаратов ДМ-SL (проект «Морской старт»);
- разгонный блок космических аппаратов «Фрегат»;
- модернизированная ракета-носитель тяжелого класса «Протон-М»;
- разгонный блок космических аппаратов ДМ-SL-Б (проект «Наземный старт»);
- разгонный блок космических аппаратов ДМ-03;
- первая ступень KSLV-1 для южнокорейской ракеты-носителя легкого класса KSLV (Korean Space Launch Vehicle);
- ракета-носитель легкого класса Ангара 1,2;
- ракета-носитель тяжелого класса Ангара-А5;
- разгонный блок космических аппаратов КВТК (кислородно-водородный тяжелого класса) и др.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

А

Алгоритм

- бесконечный 206, 207
- биологический 313-323, 330, 492
- классический 278, 279, 283
- линейный 80, 107, 113, 123
- медицинский 269-271, 490
- неклассический 278, 279, 283
- нерукотворный 323, 492
- разветвленный 58, 59, 71, 80, 113, 123
- рукотворный 312
- циклический 11, 113-152
- частично формализованный 490-492, 498, 499
- эргономичный 9, 54, 336

Алгоритмическая структура 41, 56, 254, 436

- примитив 56-77
- силуэт 41-55

Алиас 263, 265

Атом 397, 415, 417-419, 423, 424, 433, 455, 463, 464, 468, 470

- непустой 417-419
- простой 417, 418
- пустой 417-419
- составной 417, 418
- функциональный 417, 418

Б

Блок-схема 7, 8, 15-18, 31-37, 52, 53, 60, 112, 242-254, 294, 345, 433, 441, 453-463, 468, 471, 472, 490, 503, 506

В

Валентные точки 397, 417, 418, 421, 455, 469

- заготовок 397, 417
- макроикон 397
- критические 417-419, 433, 463, 464
- нейтральные 417-419

Ввод атома 397, 415, 417, 419, 424, 433, 464, 468

Ветка 41-55, 67, 70-77, 130, 138, 178, 183, 202

- безадресная 52
- одноадресная 52
- многоадресная 52, 72, 73, 178, 183, 287, 405, 465

Веточный цикл 52, 113, 123, 124, 130, 132, 139, 148, 219

Вертикальное объединение 93, 99, 105, 111, 112, 468

Вертикальное разъединение 105

Визуальная формула 84, 85, 156, 157, 161, 162, 164, 169, 170

Визуализация логических формул 153, 159, 178, 183

Визуальный синтаксис 75, 428, 430

Внешние соединители 53

Вставить ветку 409

Вход ветки 46, 55

Выход ветки 46

Выход из цикла досрочный 124, 125, 132, 139, 148, 152, 468

Выход из цикла основной 124, 125, 132, 139, 148, 152

Г

Гибридные языки программирования 255, 259, 263, 265, 266, 499

Главная вертикаль 130, 139, 247, 250, 416, 470

Главный вопрос условного оператора 192, 194, 195, 198, 204

Главный маршрут

- алгоритма 58, 60, 62, 66, 88, 246, 247
- ветки 70, 71, 77
- примитива 58, 60, 62, 66
- силуэта 70, 71, 77, 103, 105

Горизонтальные шины силуэта 50, 53, 373, 376

Горизонтальное объединение 93, 103, 112, 468

Графические буквы 37

Графический (визуальный) синтаксис 75,

77, 259, 263, 265, 266, 416-424, 427,
428, 434, 498, 499, 500, 514

Графоэлементы 37, 395, 414

Д

Да-нетный вопрос 20, 21, 122, 155, 442

Досрочный выход из цикла 124, 125,
132, 139, 148, 152, 468

Дракон-схема

– – буквенная 80, 88

– – смысловая 80, 88

Дробление веток 377- 392

Ж

Жирная горизонтальная линия 227, 228,
230, 232, 237-240

З

Завершение 130, 138, 139, 178, 183, 202

Заготовка-примитив 395, 404, 416, 419,
424, 428, 432

Заготовка-силуэт 395, 397, 416, 419, 424,
428, 432, 464

Заземление лианы 404, 415, 421, 433,
465, 469

Знание

– декларативное 23, 24, 28, 34, 285, 345,
499, 500, 502

– процедурное 10, 23-27, 34, 269, 275,
276, 285, 345, 499, 500, 502

Зрительная зона 197

Зрительная сцена 194, 197, 202, 242, 331,
334, 336, 376, 494

Зрительное восприятие 242, 253, 257,
377, 392, 487, 494

Зрительно-смысловой образ 51, 60, 250, 253

И

Идентификатор 184-201, 204, 263

– ветки 75

Избыточные элементы 75, 194, 243, 247,
250, 257, 258

Икона 18, 37, 38

– адрес 38, 44-46, 49-52, 55

– вариант 28-31, 38, 45

– ввод 38, 207-210

– вопрос 20-22, 28, 30, 38

– вставка 38, 107-110, 279, 310

– выбор 28-31, 38

– вывод 38, 207-210

– действие 18-21, 25, 38, 114

– заголовок 18, 25, 38,

– имя ветки 38, 44-48, 55

– комментарий 18, 38, 75, 138, 279, 330

– конец 18, 19, 25, 38, 45, 46

– конец цикла ДЛЯ 38, 132, 262, 263

– начало цикла ДЛЯ 38, 132, 262

– параллельный процесс 38, 207, 215,
216, 222-241

– пауза 38, 205-209

– период 38, 207, 214-216, 219-221

– петля цикла 38, 130, 139

– петля силуэта 38

– полка 38, 197, 198, 201-203, 262

– пуск таймера 38, 207-214, 219-221

– синхронизатор 38, 207-213, 219-221

– соединитель 38, 53, 372, 373, 376, 416

– формальные параметры 38, 416, 423

К

Каноническая дракон-схема 171-176

Когнитивная эргономика 9, 253, 302,
435, 487, 494-498, 505

Команда управления

параллельными процессами

– – – – останов 227

– – – – пуск 227

– – – – рестарт 227

– – – – стоп 227

Конец лианы 399, 404, 421

Конструктор алгоритмов 11, 152, 395-415

Критерий сверхвысокого понимания 54,
505

Л

Лиана 399, 421

– ненагруженная 421

– нагруженная 421

Логическая последовательность испол-
нения веток 51

Логическая функция И 153-159, 165

– – стандартная 166-168, 170, 172,
177, 178

– – нестандартная 166-168, 170, 177, 178

Логическая функция ИЛИ 159-163,
165, 166

– – стандартная 167, 168, 170, 172,
177, 183

– – нестандартная 167, 168, 170, 177, 183

Логическая функция НЕ 163-165

М

Макроикона 39

– ввод по таймеру 39

– вставка по таймеру 39

– вывод по таймеру 39

- действие по таймеру 39
- заголовок с параметрами 39
- обычный цикл 39, 113, 123, 125, 132, 139, 214
- обычный цикл по таймеру 39
- параллельный процесс по таймеру 39
- переключатель 28, 31, 39, 132, 404
- переключатель по таймеру 39
- переключающий цикл 39, 113, 123, 124, 132, 137, 464
- переключающий цикл по таймеру 39
- полка по таймеру 39
- пуск таймера по таймеру 39
- развилка 28, 30, 39, 78, 79, 83, 85
- развилка по таймеру 39
- цикл ДЛЯ 39, 113, 123, 124, 132, 136, 262, 264
- цикл ДЛЯ по таймеру 39
- цикл ЖДАТЬ 39, 75, 113, 123, 212-215, 219, 233, 236
- цикл ЖДАТЬ по таймеру 39
- Маршрут 58-62, 66, 70, 71
- Массив 262, 264, 356-359
- Матрешка 419, 420
 - непустая 419, 420
 - пустая 419, 420
 - частично пустая 419, 420
- Многостраничный силуэт 311, 371, 372, 373, 376

Н

- Набор маршрутов 81-83, 103, 111
- Набор формул 81-83, 111
- Начало лианы 399, 421

О

- Одностраничный силуэт 372, 376
- Оператор
 - адрес 75
 - ввод 207, 209
 - ветка 45, 75
 - вывод 207, 209
 - вызов процедуры 108
 - имя ветки 46
 - обычный цикл 125
 - параллельный процесс 215
 - пауза 209, 212, 219-221
 - перехода (*goto*) 46
 - период 214, 215, 219-221
 - полка 197, 201, 202
 - присваивания (присвоить) 195, 219
 - пуск таймера 209, 211, 212, 214, 219, 221
 - пустой 417

- реального времени 207, 216
- синхронизатор 209, 211, 212, 219, 221
- снять признак 197, 198, 201
- составной 207
- текстового языка 75
- установить признак 197, 198, 201
- цикл ДЛЯ 132
- цикл ЖДАТЬ 214, 219
- языка ДРАКОН 75
- Ошибка «нет шампура» 57
- Ошибка «сиамские близнецы» 420, 421

П

- Пересадка лианы 399, 402-404, 415, 421-424, 433, 465, 468-470
- Передача управления 46, 75, 110, 220, 437, 442, 458
- Переменная цикла 262, 263
- Плечо развилки 78, 79, 83-85, 111, 409
 - левое 78, 79
 - правое 78, 79
- Побочные маршруты 58, 59, 88, 103
 - ветки 70, 71, 77
 - примитива 58-66
- Понятность алгоритмов 10, 11, 37, 51, 54, 55, 60, 67, 78-112, 162, 184, 192, 194, 242, 263, 266, 269, 276, 286, 294, 300, 344, 345, 348, 442, 468, 487, 494, 498, 504-506
- Понятность веток 377-392
- Поиск данных 360-364
- Правило
 - главного маршрута 58
 - «Движение вверх запрещено» 107
 - «Пересечения линий запрещены» 52, 55, 99, 103, 105, 112, 148, 246, 469, 470
 - «Повторы запрещены» 93
 - хорошей хозяйки 62
 - «Чем ниже, тем позже» 24, 25
 - «Чем правее, тем хуже» 59-62, 66, 71, 77, 88, 105
 - шампура 57
- Примитив 56-77, 105, 130, 250, 371-376, 395, 404, 416-433, 464, 465, 481
- Принцип «Посмотрел – и сразу понял!» 2, 37, 165, 302, 505
- Процедура 195, 197, 202, 212, 215, 223, 294, 295, 310, 325, 371, 457, 500
- Процесс 37, 279, 285, 312, 313, 318, 344, 348
 - параллельный 207, 215, 216, 222-241
 - последовательный 19, 24, 25, 41, 215, 217
- Псевдокод 34, 73, 468, 503
- Пункт разделения (*concurrent fork*) 236, 240

Пункт слияния (concurrent join) 237, 240

Р

Равносильные алгоритмы 82-84, 88, 99,
103, 108, 111, 112, 157, 162, 166, 167

Разметка веток для дробления 383

Разрыв ветки

– – простой 378, 380

– – с присоединением 378, 380

– – с дублированием 378, 380

Ритмические промежутки 392

Рекурсивные алгоритмы 365-368

Рокировка 83-88, 105, 111, 112, 166, 167,
171, 172, 183, 409

С

Силуэт 41-55, 67-77, 105, 108, 112, 130,
139, 250, 254, 286, 292, 311, 371-
395, 404, 409, 410, 416, 419, 421-
424, 428, 432, 433, 436-449, 465

Смысловые части алгоритма 41, 45, 46,
48, 49, 67

Структура деятельности 277, 278, 285,
302, 311, 500, 501

Т

Текст

– декларативный 23

– процедурный 23

Текстовый синтаксис 75, 77, 132, 259,
263-266, 499

Текстовый язык 73, 75

Текстоэлементы 262

Тело ветки 44, 45, 390

Тело цикла 119, 130, 139

Техноязык 499-503

Точка ввода 417

Точка дублирования 378, 380, 383, 386

Точка присоединения 378, 380, 383, 386

Точка разрыва простая 378, 380

Точка слияния 78, 79, 111, 399, 421

Треугольник 227-231, 237, 240, 241

У

Удалить ветку 409

Укрупнение веток 390

Управляющая графика 12, 471

Управляющая структура (управляющая
конструкция) 452-454, 457, 458, 471

Условие окончания цикла 120, 121, 122,
123, 130, 139, 141, 144

Условие продолжения цикла 118, 120,
121, 122, 123, 130, 139, 144

Ф

Формула маршрута 78, 81, 83, 99

Ц

Цикл в цикле 132, 140-152

Цикл гибридный 114, 123, 132

Цикл ДО 113, 114, 123, 132, 140, 141,
146, 148, 152, 258

Цикл ПОКА 114, 119, 123, 141, 146, 148,
152, 250, 258

Ч

Черный треугольник 52, 132, 139, 404

Ш

Шампур 56-60, 66, 70-72, 77, 82-85, 88,
103, 105, 125, 130, 139, 166-168,
183, 246, 247, 250, 253, 399, 416,
417, 421, 427, 428, 433, 434, 442, 449

Шампур-блок 40, 125, 139, 399, 416, 417,
421, 464, 470

Шампур ветки 70, 71, 77

Шампур-метод 449-472

Шампур-схема 427, 428, 433, 434, 449, 451

Шампур цикла 130, 139

Шапка 46, 48, 205, 302, 303

Э

Эквивалентное преобразование алгоритмов 84, 108, 189, 421

Эквивалентные алгоритмы 75, 84, 93,
112, 165, 198, 244, 247, 256, 258,
377, 378, 380, 383, 392, 464, 470

Эргономичная декомпозиция 310, 311,
371, 372, 376

Эргономичный

– алгоритм 9, 54, 336

– алгоритмический язык 9, 11, 301, 487

– текст 184-204

Книги издательства «ДМК Пресс» можно приобрести в торгово-издательском холдинге «АЛЪЯНС-КНИГА» (АЛЪЯНС БУКС) наложенным платежом или выслать письмо на почтовый адрес: Россия, 115487, Москва, 2-ой Нагатинский пр-зд, д. 6А.

При оформлении заказа в письме следует указать полностью Ф.И.О. и почтовый адрес заказчика (с индексом).

Эти книги вы также можете заказать на сайте: **www.alians-kniga.ru**.

Оптовые продажи: тел. **(499) 725-50-27, 725-54-09**.

Электронный адрес: **books@alians-kniga.ru**.

Паронджанов Владимир Данилович

e-mail: vdp2007@bk.ru

УЧИТЬСЯ ПИСАТЬ, ЧИТАТЬ И ПОНИМАТЬ АЛГОРИТМЫ

АЛГОРИТМЫ ДЛЯ ПРАВИЛЬНОГО МЫШЛЕНИЯ

Основы алгоритмизации

Главный редактор Мовчан Д. А.
 dm@dmk-press.ru

Корректор Синяева Г. И.

Верстка Паранская Н. В.

Дизайн обложки Мовчан А. Г.

Подписано в печать 26.03.2012. Формат 70×100 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 42,25. Тираж 1000 экз.

№

Web-сайт издательства: www.dmk-press.ru