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-Манны и т.д.

Глава 34

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 произвольно 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 саму 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)
Вывод в исчислении V есть последовательность C_1, ... C_n формул, такая, что для любого i формула C_i есть: ● либо аксиома исчисления V; ● либо непосредственное следствие предыдущих формул по одному из правил вывода. Формула C_n называется теоремой исчисления V, если существует вывод в V, в котором последней формулой является C_n [6].	Видеовывод в видеоисчислении V есть последовательность C_1, ... C_n видеоформул, такая, что для любого i видеоформула C_i есть: ● либо видеоаксиома видеоисчисления V; ● либо непосредственное следствие предыдущих видеоформул по одному из правил видеовывода. Видеоформула C_n называется видеотеоремой видеоисчисления V, если существует видеовывод в V, в котором последней видеоформулой является C_n.

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

Определение понятия «логическое исчисление»	**Определение понятия «видеоисчисление» (визуальное логическое исчисление)**
Логическое исчисление может быть представлено как формальная система в виде четверки $V = < И, S_0, A, F >$ где ● *И* – 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].	Видеоисчисление может быть представлено как формальная система в виде четверки $V = < И, S_0, A, F >$ где ● *И* – 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 – *видеотеоремы*. (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 *И* (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 драконсхем). 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 выводить 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 выводить 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 драконсхем следует признать значительным шагом вперед, повышающим производительность труда при практическом 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.

Глава 35

МЕТОД АШКРОФТА-МАННЫ И АЛГОРИТМИЧЕСКАЯ СТРУКТУРА «СИЛУЭТ»

§1. МЕТОД Эдварда АШКРОФТ И Зохара 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*.

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.

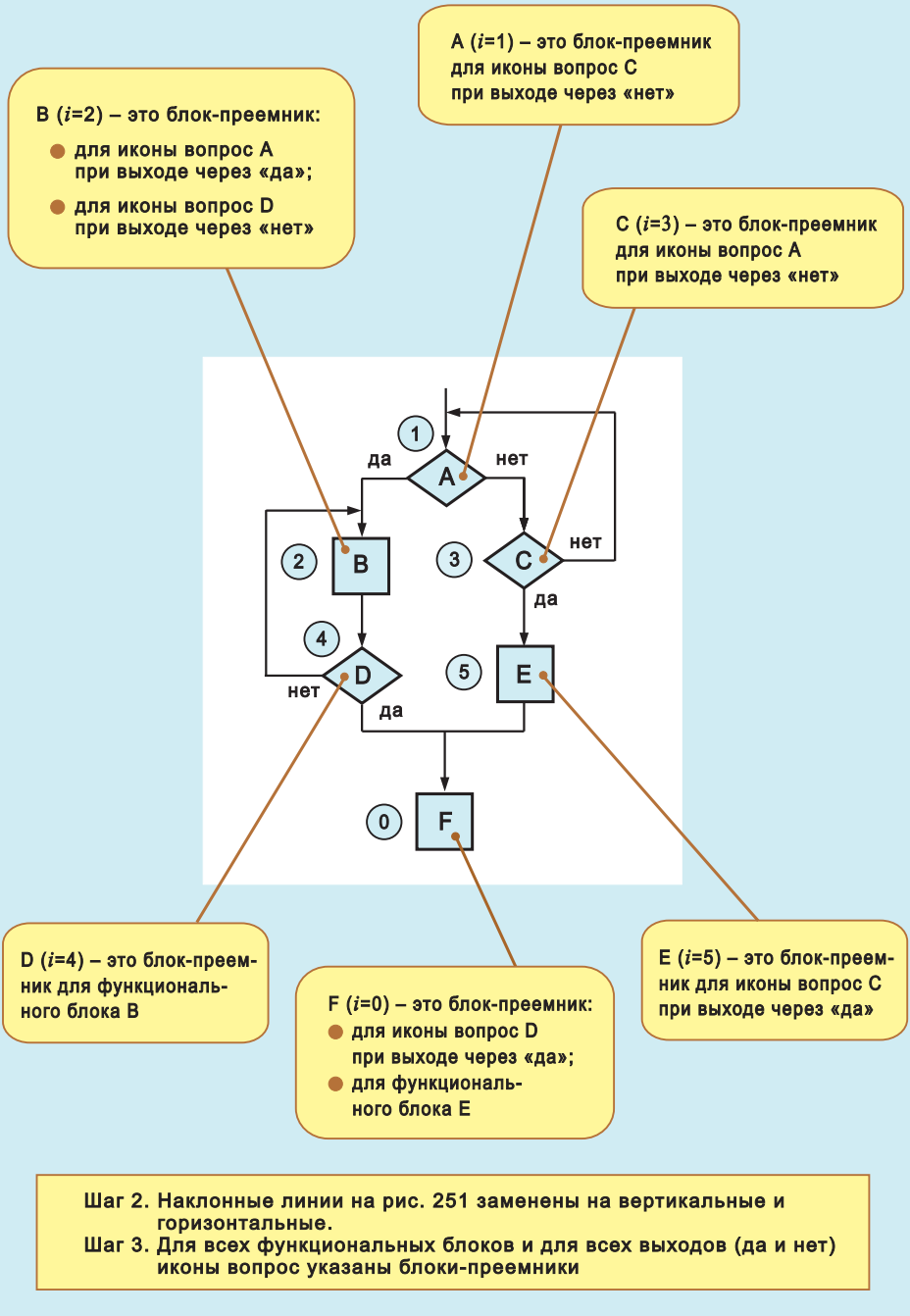


Рис. 252. Та же самая неструктурная схема. Она получена из схемы на рис. 251 путем замены наклонных линий. Кроме того, указаны все блоки-преемники

Блок, передающий управление	Блок-преемник
Икона-вопрос A при выходе через «да»	B
Икона-вопрос A при выходе через «нет»	C
Функциональный блок B	D
Икона-вопрос C при выходе через «да»	E
Икона-вопрос C при выходе через «нет»	A
Икона-вопрос D при выходе через «да»	F
Икона-вопрос D при выходе через «нет»	B
Функциональный блок E	F
Функциональный блок F (это последний блок схемы)	Последний блок F не имеет блока-преемника

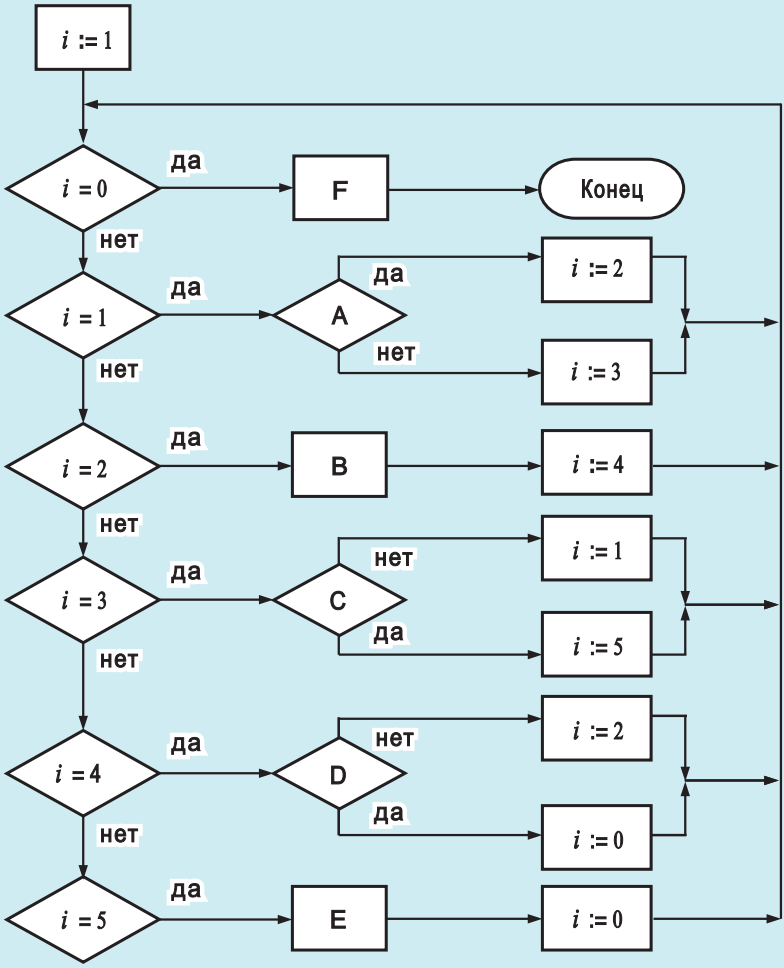
§4. НОВАЯ ПЕРЕМЕННАЯ

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-ПРЕЕМНИКОВ ДЛЯ ФУНКЦИОНАЛЬНЫХ БЛОКОВ

Функциональные блоки присваивают переменной *i* номер блока-*преемника* в исходной схеме на рис. 251.

Номера блоков-преемников для функциональных блоков	
Блок-преемник	**Номер блока-преемника на рис. 251**
B	$i = 2$
E	$i = 5$
F	$i = 0$
Переменная *i* принимает значение, равное номеру блока-преемника 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-Манны. Назовем эту схему «схемой Ашкрофта-Манны»

§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**
F	Икона-вопрос D при выходе через «да»	$i = 0$
A	Икона-вопрос C при выходе через «нет»	$i = 1$
B	Икона-вопрос A при выходе через «да»	$i = 2$
B	Икона-вопрос D при выходе через «нет»	$i = 2$
C	Икона-вопрос A при выходе через «нет»	$i = 3$
E	Икона-вопрос C 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 вероятна 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-Манны. Следующий этап – преобразование схемы Ашкрофта-Манны 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.

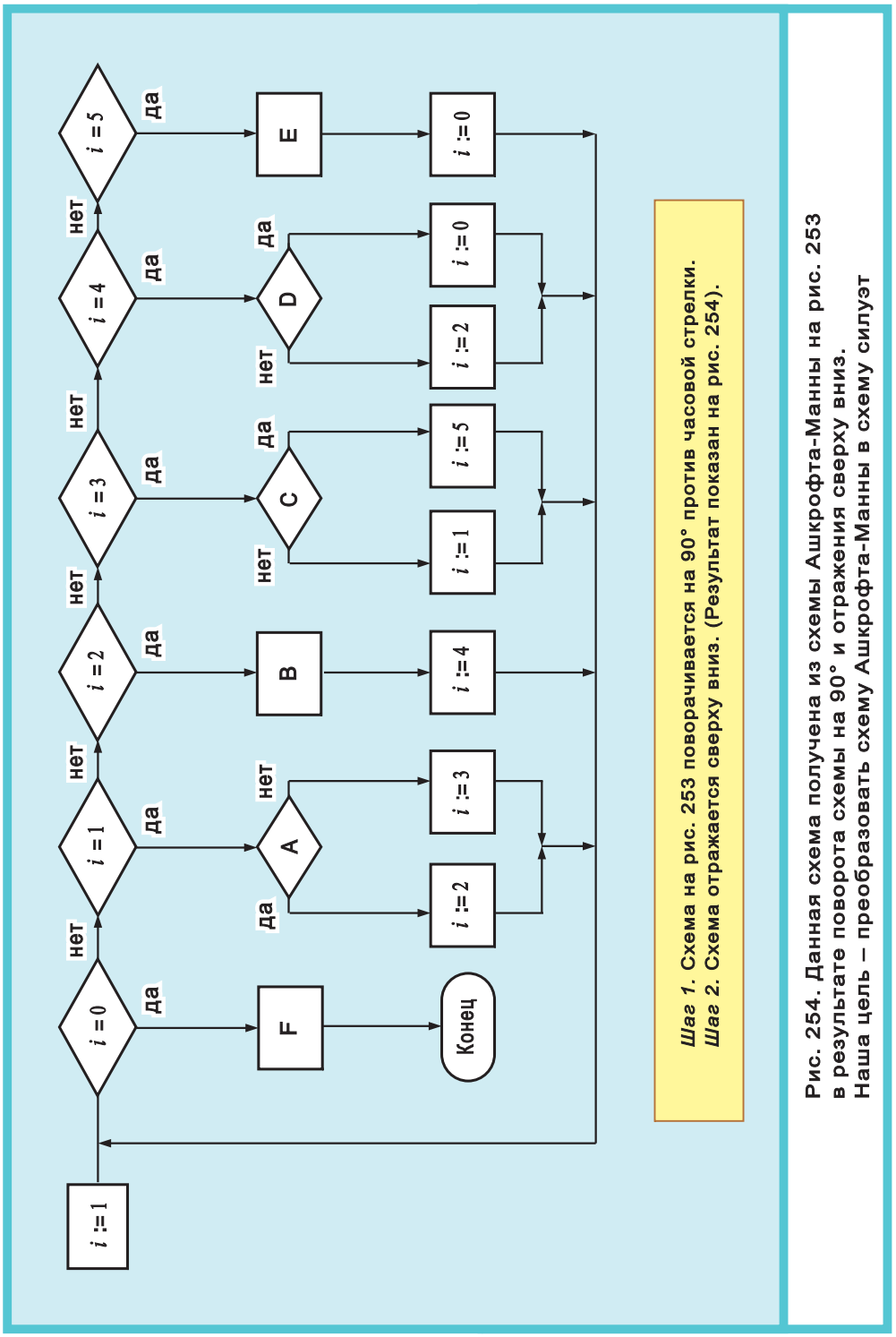
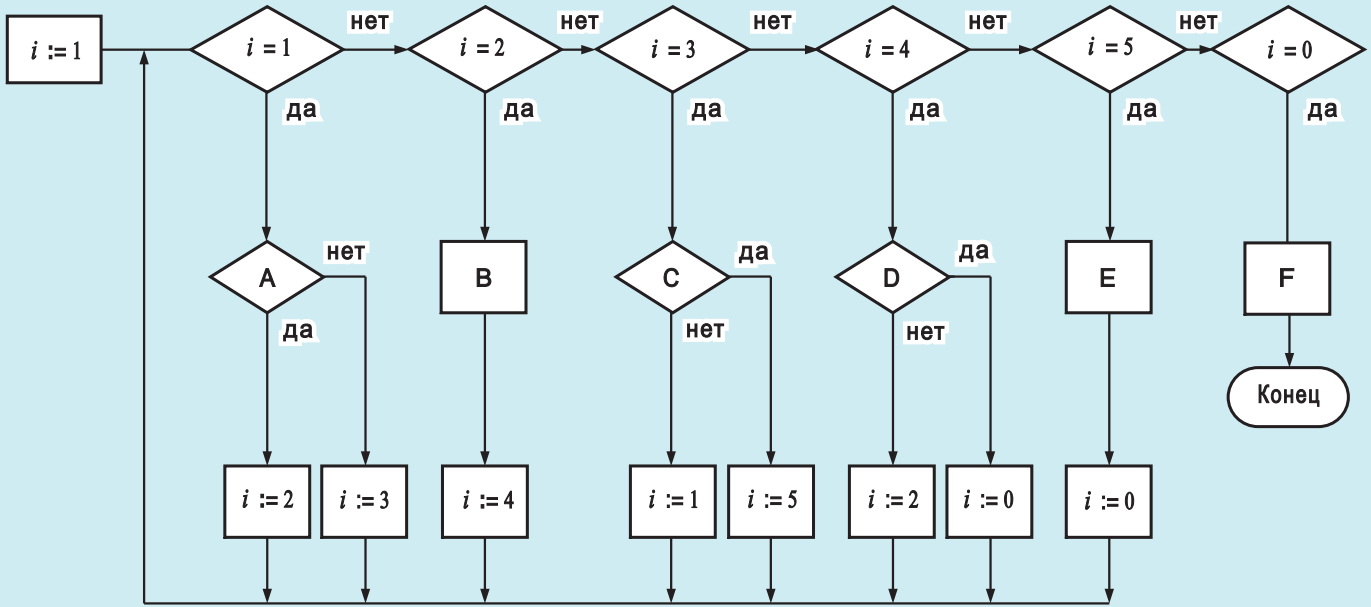
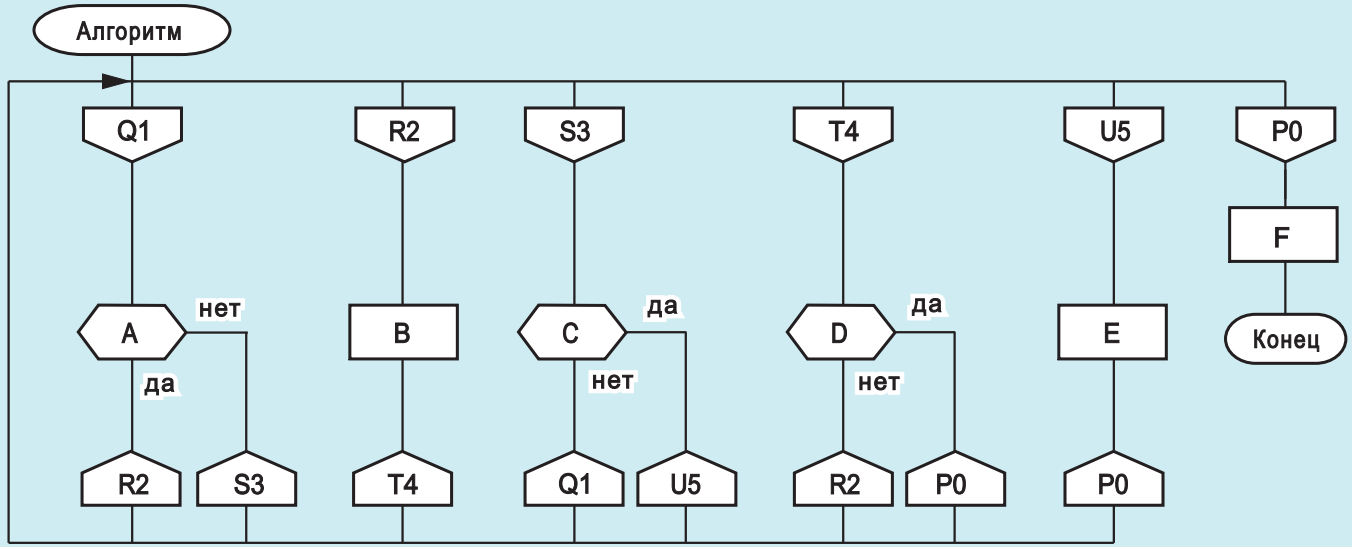


Рис. 254. Данная схема получена из схемы Ашкрофта-Манны на рис. 253 в результате поворота схемы на 90° и отражения сверху вниз. Наша цель – преобразовать схему Ашкрофта-Манны в схему силуэт



Шаг 3. **Вертикаль, содержащая икону «конец», перемещена в крайнюю правую позицию.**
Шаг 4. **Каждая икона «вопрос» (в верхнем ряду) нанизана на свой шампур.**
Шаг 5. **Удалены три Т-образные линии в нижней части схемы, которые заменены вертикалями**

Рис. 255. Данная схема получена из схемы Ашкрофта-Манны на рис. 253 в результате шагов 1–5. Наша цель – преобразовать схему Ашкрофта-Манны в схему силуэт

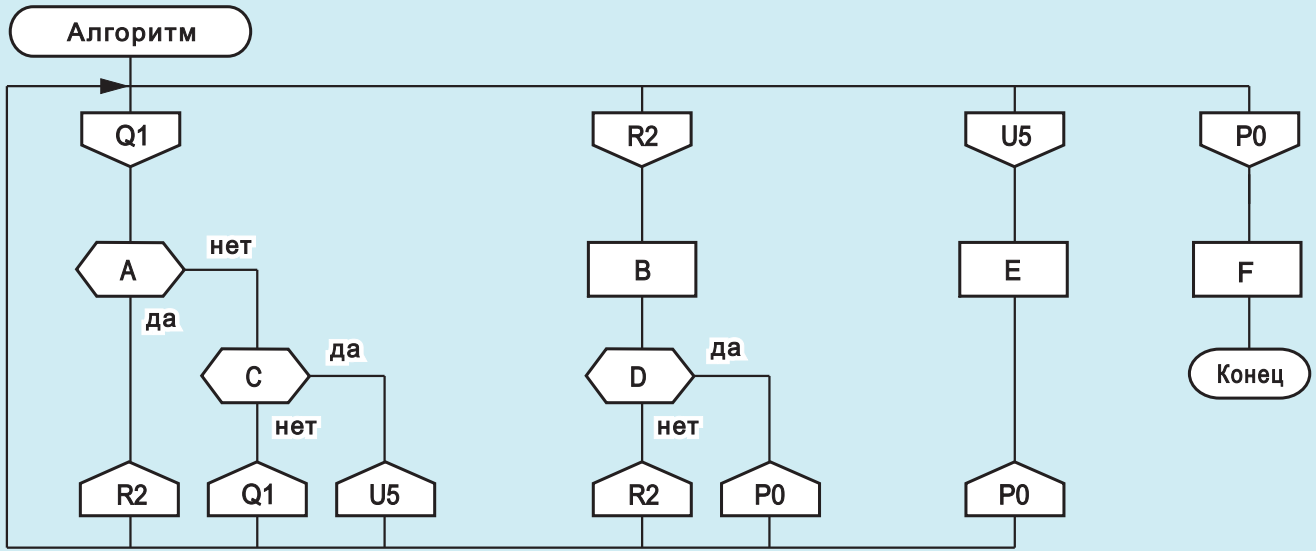


Шаг 6. Верхние фигуры на рис. 255 превращаются в иконы «имя ветки».
Шаг 7. Нижние фигуры на рис. 255 превращаются в иконы «адрес».
Шаг 8. Внутри икон «имя ветки» записываются метки, обозначающие названия веток.
Шаг 9. Внутри икон «адрес» записывается имя ветки, куда происходит передача управления в соответствии с логикой работы алгоритма.
Шаг 10. Имя переменной состояния удаляется

СООТВЕТСТВИЕ
между номерами переменной состояния *i* и именами веток

i	0	1	2	3	4	5
	P0	Q1	R2	S3	T4	U5

Рис. 256. Силуэт, полученный за 10 шагов из схемы Ашкрофта-Манны на рис. 253. Назовем этот силуэт черновым. На рис. 257 и 258 показано, как черновой силуэт превращается в окончательный

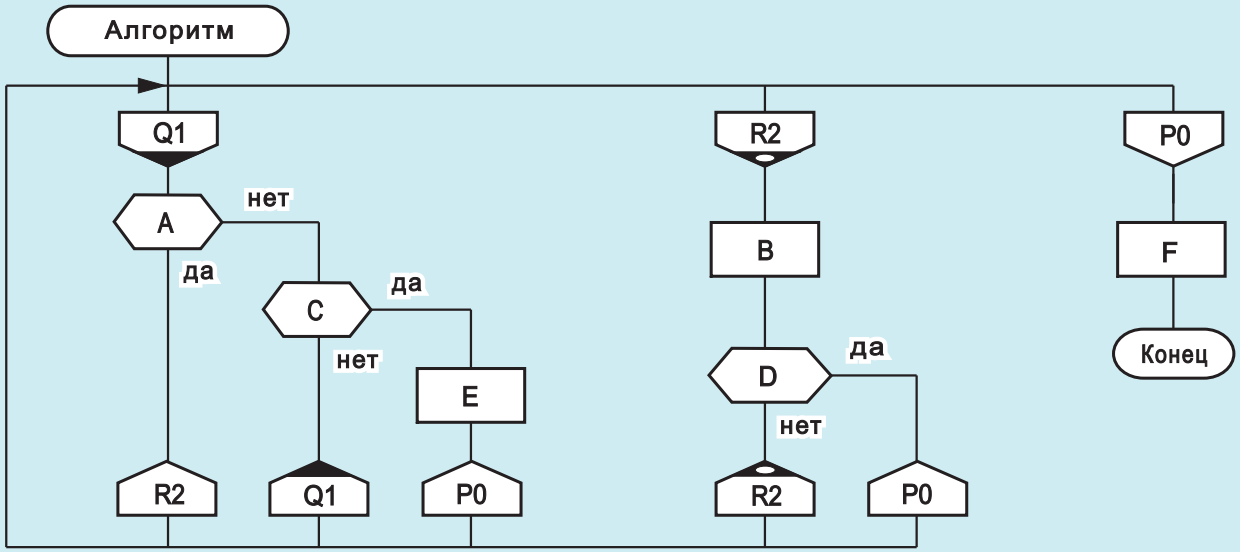


Шаг 1. На рис. 256 удалить все иконы с надписью S3.
Шаг 2. К удаленному правому плечу ветки Q1 присоединить ветку S3 с удаленной иконой «имя ветки» S3.
Шаг 3. На рис. 256 удалить все иконы с надписью T4.
Шаг 4. К удаленному низу ветки R2 присоединить ветку T4 с удаленной иконой «имя ветки» T4.

После выполнения Шагов 1–4 получены следующие результаты:

1. Объединены две ветки Q1 и S3. Икона «адрес» S3 и икона «имя ветки» S3 удалены.
2. Объединены две ветки R2 и T4. Икона «адрес» T4 и икона «имя ветки» T4 удалены.

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 чернового 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-Манны 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-Манны с помощью введения переменной состояния.
3. На втором этапе производится преобразование алгоритма Ашкрофта-Манны в черновой алгоритм «силуэт».
4. На третьем этапе выполняется преобразование чернового алгоритма «силуэт» в окончательный алгоритм «силуэт».
5. Метод Ашкрофта-Манны является математическим обоснованием алгоритмической структуры «силуэт».

Глава 36

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

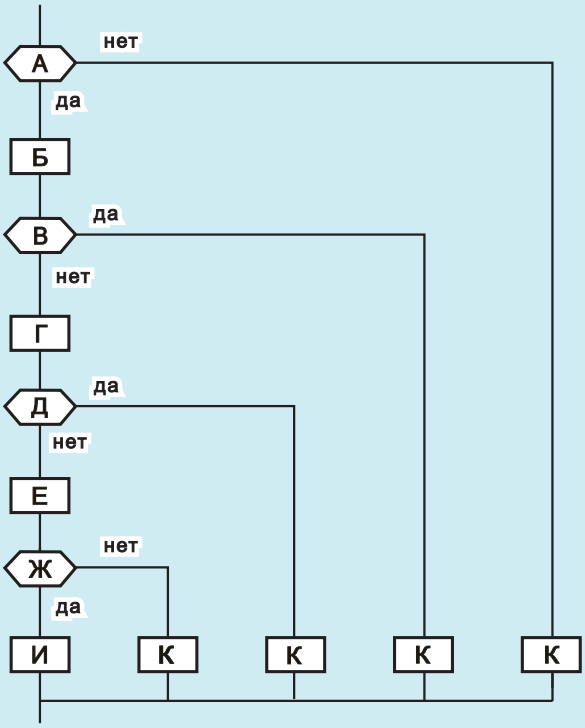


Рис. 259. Плохая схема.
В ней слишком много вертикалей (пять) и одинаковых икон (четыре). Кроме того, есть три лишних излома, а три горизонтали и вертикали неоправданно длинные

ПРАВИЛЬНО

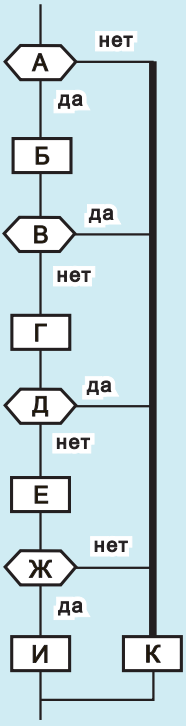


Рис. 260. Хорошая схема.
Число вертикалей, икон и изломов удалось значительно сократить

Равносильное преобразование «вертикальное объединение» улучшает эргономичность дракон-схемы

Таблица 1

Позиция участников дискуссии	Используются три структурные конструкции?	Используются заменители *goto*?	Используются *goto*?
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 *заменителей 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.

Заменители – 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, заменители 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 Модула-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

Попытаемся 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, использовав 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 видеоструктурной[1] 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».

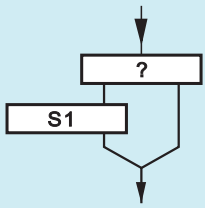
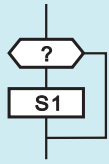
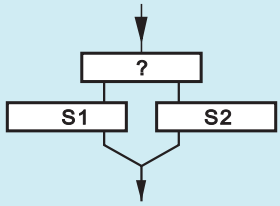
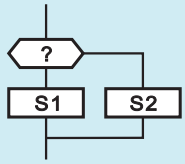
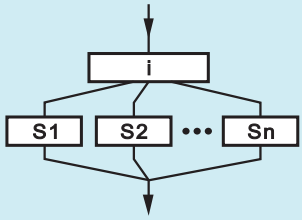
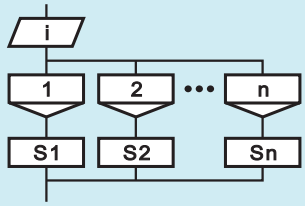
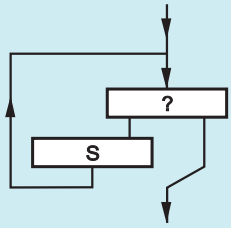
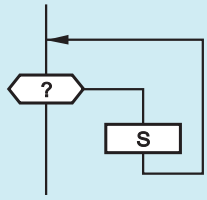
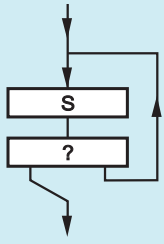
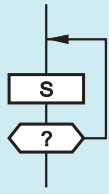
Структурные операторы Дейкстры	Структурные блок-схемы Дейкстры	Дракон-схемы
if ? do S1		
if ? then S1 else S2		
case i of (S1;S2;...Sn)		
while ? do S		
repeat S until ?		

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 невостребованной, 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-Манны [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 Бома 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

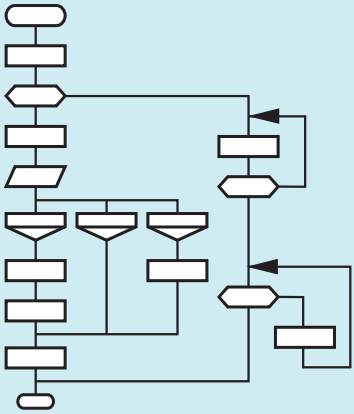


Рис. 262. Примитив, построенный из структурных блоков

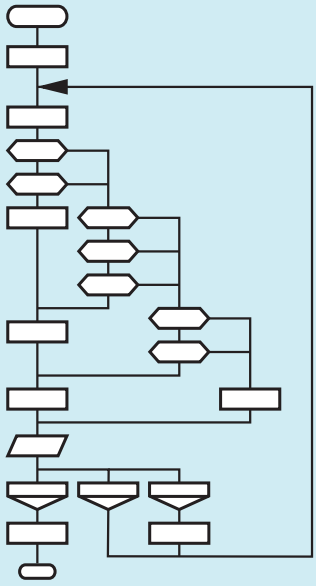


Рис. 263. Примитив, построенный из структурных и лианных блоков

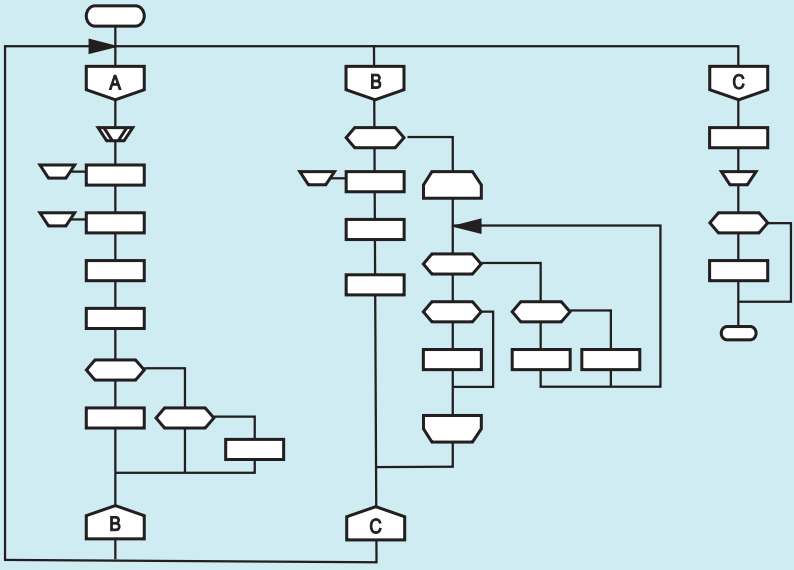


Рис. 264. Силуэт, построенный из структурных блоков

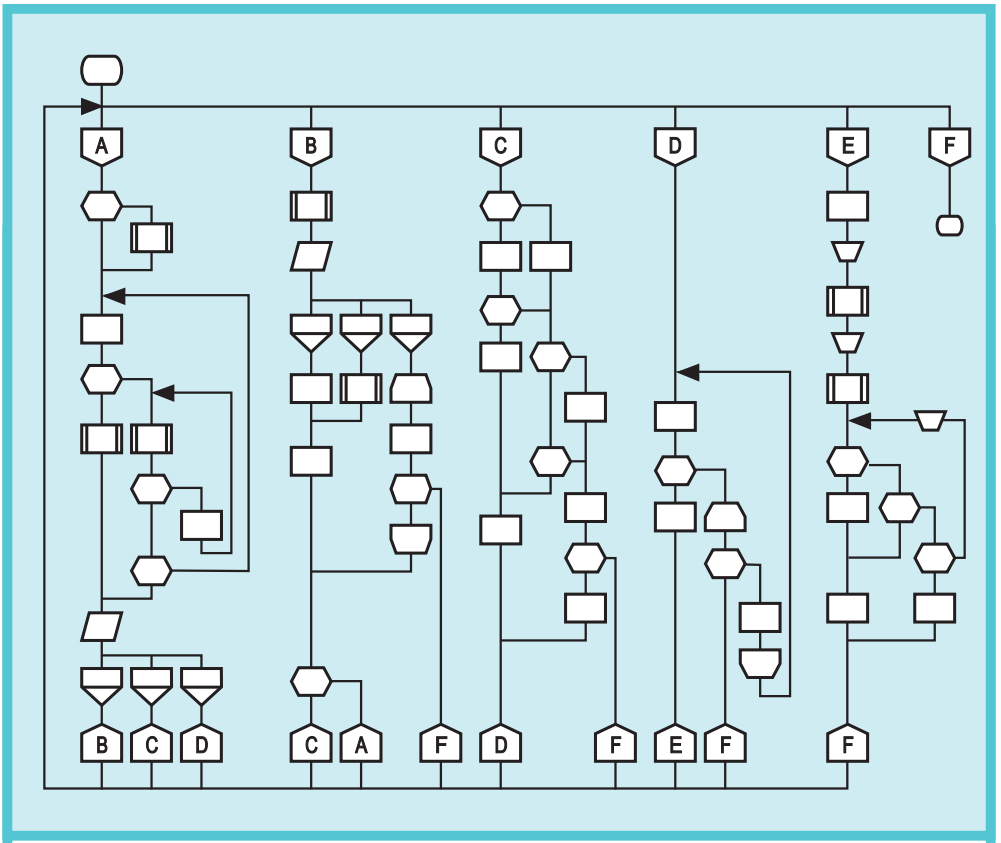


Рис. 265. Силуэт, построенный из структурных, лианных и адресных блоков

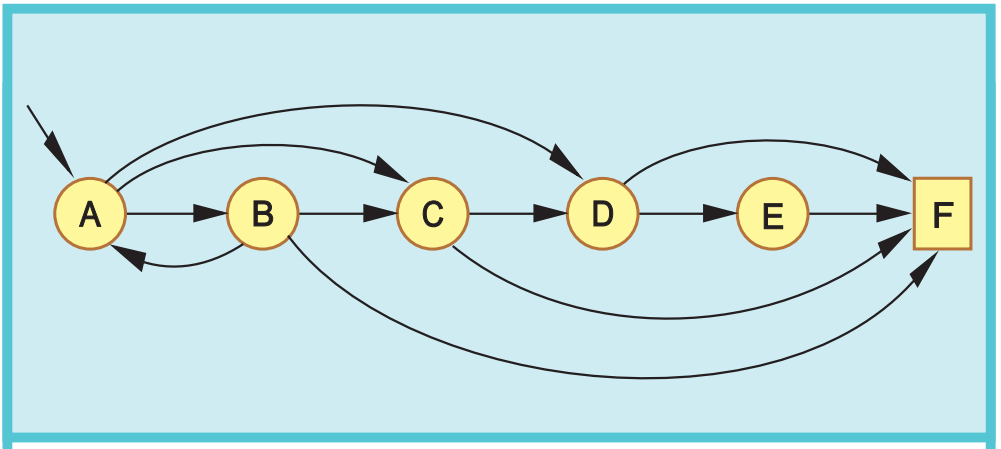


Рис. 266. Детерминированный конечный автомат, соответствующий силуэту на рис. 265

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 шампур-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 шампур-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.