УДК 004

ТЕОРИЯ БЛОК-СХЕМ
Theory of block diagram

М. Н. Сандлерман, старший преподаватель
Уральского государственного аграрного университета
(Екатеринбург, ул. Карла Либкнехта, 42)

Рецензент: А. Н. Красовский, доктор физико-математических наук, профессор, зав.кафедрой информационных технологий Уральского государственного аграрного университета

Аннотация

Делается попытка создания теории блок-схем, которая повышает эффективность труда программистов. Теоретические выводы проверены многолетним опытом их применения автором.

Ключевые слова: инфраструктура науки, инфраструктура теории, теория блок-схем, канонический вид блок-схем, блок-схемы, логика блок-схем, блок-схемы для следствия.

Summary

There is given the attempt to develop a theory of block diagrams, which increases the efficiency of programmers. The theoretical conclusions are verified by the author`s longstanding experience of its application.

Keywords: science infrastructure, infrastructure of theory, theory of block diagrams, the canonical form of block diagram, block diagram, logic of block diagrams, block diagram for investigation.

Начнем с сенсационного утверждения: *теории блок-схем не существует*, нет ее, она не создана, не разработана. Тем не менее сам по себе термин «*теория блок-схем*» существует. Однако то, что существует, теорией как таковой не является. Это серьезное заявление, которое мы постараемся обосновать.

Известно, что блок-схема является графическим отражением алгоритма программы. В литературе подробно описаны графические обозначения всех видов логических блоков. Даже созданы специальные программы для графической реализации блок-схем. И все! Это все я бы назвал *инфраструктурой* теории блок-схем. Для теории наличие инфраструктуры необходимо, но недостаточно.

Сравним с шахматами. Представим себе, что нам известно все то, что составляет инфраструктуру шахмат, а именно описание фигур, шахматной доски, правила и цель игры. Можно прибавить к этому описание способов записи ходов (шахматная нотация), несколько образцов практически сыгранных партий. Но для понятия *теории шахмат* этого явно недостаточно. Теория должна давать рекомендации по игре. Как играть в начале, середине и в конце партии. Например, окончания «король и пешка против короля». В одних позициях при правильной игре сильнейшая сторона всегда побеждает, в других – слабейшая сторона добивается ничьей. Раз это все есть, значит, теория шахмат существует.

Что касается блок-схем, то рекомендации по их вычерчиванию (рисованию) существуют. Но рекомендаций по их созданию, целесообразности и эффективности применения (внедрения) пока нет. Постараемся этот недостаток устранить или, точнее, положить начало этому.

Совершим небольшой экскурс в логику. Рассмотрим два суждения.

Эта стена белая.

Эта стена черная.

Оба суждения обладают следующими свойствами. Они по-разному оценивают один и тот же объект. Если истинно одно из них, то другое будет ложным. Но вполне возможно, что они

оба могут быть ложными, если стена имеет какой-нибудь иной цвет в отличие от белого или черного. Суждения такого рода в логике называются **противоположными.**

Еще пара суждений.

Эта стена белая.

Эта стена не белая.

Здесь одно суждение отрицает то, что утверждает другое. Истина всегда есть. Одно из них истинно, другое ложно. Оба эти суждения одновременно не могут быть истинными или ложными. Более того, никакое третье суждение нельзя сюда добавить. Можно, конечно, сказать:

Эта стена красная.

Но это суждение будет совпадать со вторым (*Эта стена не белая*). То есть стена может быть белой или не белой и никакой третьей, потому что любое третье суждение совпадет с первым или вторым. Суждения такого рода в логике называются **противоречащими.** И именно эти суждения нужны нам в излагаемой здесь теории блок-схем. Что касается упомянутых выше **противоположных** суждений, то ими мы не будем пользоваться, а приведены они в целях их разграничения с **противоречащими.**

На этом мы заканчиваем экскурс в логику. Надеемся, что читатель понял суть противоречащих суждений и сможет отличить их от любых других. Более подробно это изложено в учебниках по логике [2].

Цели. Главная цель данной теории – облегчить процесс написания (создания) компьютерных программ. Представляется очевидным, что вначале нужно создать блок-схему будущей программы, затем произвести *отладку программы на уровне блок-схемы*. И только после этого можно приступить к написанию самого текста программы на любом языке программирования. Затем последуют пробные запуски программы, пробная ее эксплуатация, то есть *отладка на машинном уровне.*

Формы блок-схем. Это графические фигуры в виде разных четырехугольников и иных рисунков, связанных между собой стрелками переходов (передач управления). Для изложения сути теории вполне достаточно все многообразие фигур свести к двум: прямоугольнику и ромбу. В ромб принято записывать вопрос, на который нужно ответить *да* или *нет*. Все остальные суждения, предложения, фразы должны носить однозначный утвердительный характер и будут помещаться в прямоугольники.

Далее будет показано, что кроме *графических* существуют еще и *табличные* блок-схемы.

Классы блок-схем. Все блок-схемы разделяются на два вида (класса, типа). Один вид мы назвали *каноническим*. Сам по себе этот термин широко известен и давно используется в науке. У другого вида пока названия нет, поэтому будем называть его *неканоническим*, так как к нему будут относиться буквально все блок-схемы, которые не являются каноническими.

Договоримся, что впредь содержимым логических блоков, изображаемых ромбами, буду только упомянутые ранее *противоречащие суждения.* Причем в блок (ромб) будет записываться только одно из двух суждений и обязательно (!) со знаком вопроса. Пример со стеной:

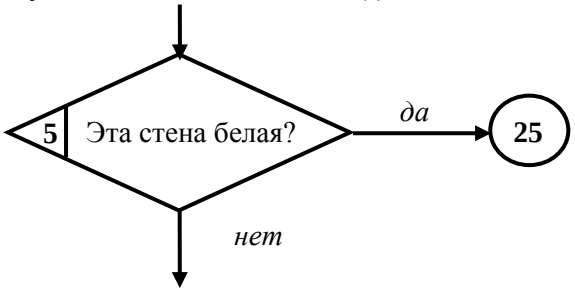

Рис. 1. Блок с противоречащими суждениями

Цифрами на рис. 1 обозначены порядковые номера логических блоков. Верхняя стрелка сверху передает управление блоку № 5, в котором задается вопрос. Если стена окажется белой, управление будет передано блоку № 25, если стена будет иметь любой иной цвет, то управление будет передано блоку вниз по линии «*нет*».

Определим теперь содержание прямоугольных блоков, в которые будем помещать суждения утвердительного характера. Отдельно взятый такой блок может содержать всего одну арифметическую операцию или целый комплекс операций. Например, задача из аналитической геометрии: на плоскости даны координаты трех точек, являющихся вершинами треугольника, площадь которого надо найти. Главное свойство таких блоков: последовательность выполняемых в них операций является однозначной и четко определенной. Блок-схема, состоящая из них и блоков с противоречащими суждениями (как на рис. 1), называется **канонической.**

Итак, **канонической** называется блок-схема, состоящая только из двух видов блоков: утвердительных (прямоугольной формы) и с противоречащими суждениями (ромб), причем из прямоугольников не может быть более одного выхода, а из ромбов всегда есть два (и только два) с обязательным названием «*да*» и «*нет*»; количество входов в любой блок не ограничивается.

На конкретном примере покажем разницу между каноническим и неканоническим видами блок-схем. За основу возьмем алгоритм решения квадратного уравнения в стандартном виде:

$$ax^2 + bx + c = 0.$$

Алгоритм и блок-схемы решения этого уравнения подробно рассмотрены в литературе, поэтому ограничимся двумя видами блок-схем с учетом специфики данной статьи. Каноническая блок-схема будет иметь вид, как на рис. 2.

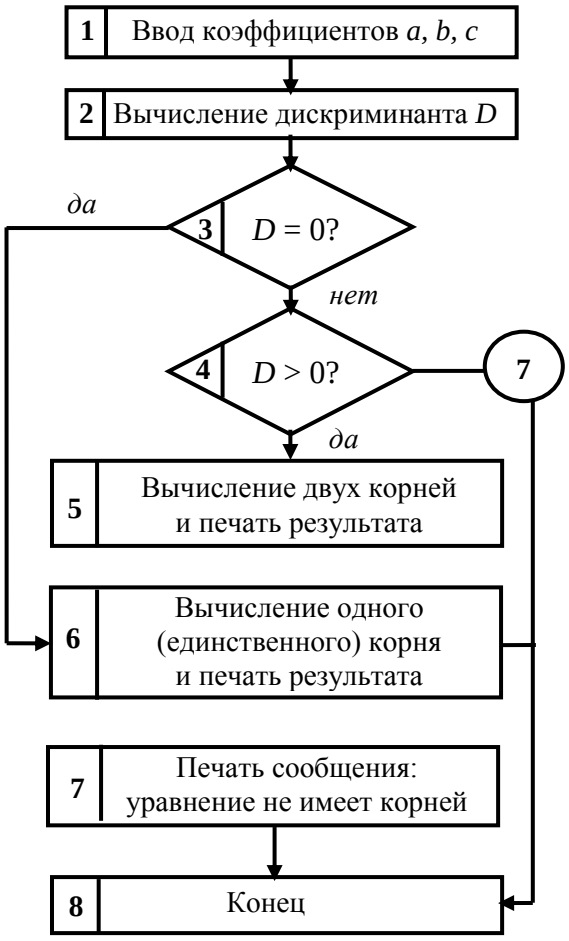

Рис. 2. Каноническая блок-схема

Этот же алгоритм может быть изображен неканонической блок-схемой (рис. 3).

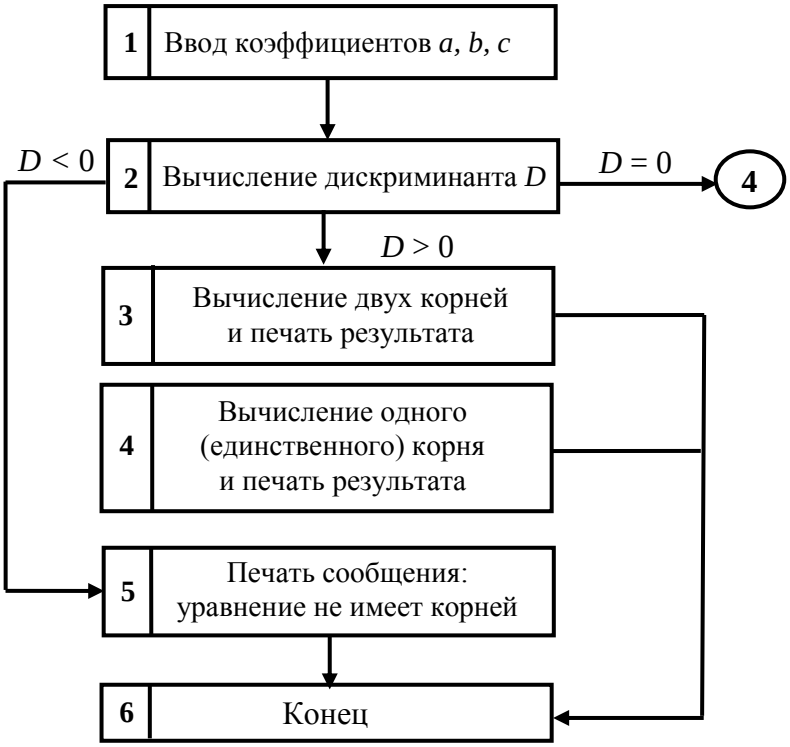

Рис. 3. Неканоническая блок-схема

Принципиальное различие между двумя видами заключается в том, что для канонической блок-схемы выход из любого утвердительного блока (прямоугольная форма) может быть **только один** (если, разумеется, этот блок не последний, у которого нет выходов). В неканонической блок-схеме количество выходов из таких блоков не ограничивается. На рис. 3 видим, что из блока № 2 есть три выхода (разветвления).

Оба вида блок-схем имеют право на существование и могут быть успешно реализованы в конечном тексте программы. Можно доказать, оформив это доказательство как теорему, что **любую неканоническую блок-схему можно привести к каноническому виду.** Смысл разделения всех блок-схем на два вида заключается в том, чтобы специально выделить один так называемый *канонический* вид, который имеет ряд важных для практики программирования достоинств. Отмечу следующие два из них.

1. Автоматически гарантируется логическая полнота. Это означает, что все переходы (передачи управления) будут перечислены и ничего не будет упущено. В неканонической блок-схеме этот эффект достигается труднее. Из блока № 2 (рис. 3) есть три выхода. В этом примере все просто и очевидно, что никакого 4-го выхода здесь нет. Но задача может быть более сложной. Например, нужно обработать переменную *x* в разных промежутках ее существования. Предположим, что таких промежутков в какой-то данной конкретно задаче 5. Значит, из блока, в котором оказалась наша переменная, есть 5 выходов. Начав рассмотрение одного из них, все остальные пометим галочками или иным знаком для памяти, чтобы потом вернуться к ним и ничего не пропустить. При этом можем забыть или упустить одно из направлений или не догадаться дойти до 6-го промежутка, существование которого вначале не предполагалось.

Иное дело в канонической схеме. Из прямоугольного блока есть только один выход. А из ромбического блока всегда два и только два выхода с готовыми метками «*да*» и «*нет*». В таких жестких рамках очень трудно запутаться.

2. Облегчается отладка программы на уровне блок-схем. После того как блок-схема составлена, не нужно торопиться с программированием. Лучше приступить к отладке блок-схемы. В каноническом виде это делать легко и приятно. Мысленно прокручивая операции в каждом блоке, пробегаем по единственным или раздвоенным переходам. В примере из пункта 1 рассматривается ситуация, когда некоторая переменная *x* находится в одном из пяти промежутков. При реализации такого алгоритма в канонической блок-схеме придется создавать ряд противоречащих суждений. Сначала спросим: находится ли *x* в первом промежутке? Ответ может быть только одним из двух: *да* или *нет*. В случае «*нет*» аналогичный вопрос задаем в отношении второго промежутка и так далее, пока не будут рассмотрены все случаи.

Существует еще очень важный для практики программирования *табличный* вид блок-схем (мое маленькое ноу-хау). *Табличным* видом блок-схемы называется изображение канонической блок-схемы в табличной форме. Полагаю, что приведенный ниже пример все объяснит. Блок-схему с рис. 2 переведем в таблицу 1.

Таблица 1

Табличная блок-схема

№	Текст	БП	Да	Нет
1	Ввод коэффициентов *a, b, c*. Вычисление дискриминанта *D*			
2	*D* = 0?		5	3
3	*D* > 0?		4	6
4	Вычисление двух корней и печать результата	7		
5	Вычисление одного (единственного) корня и печать результата	7		
6	Печать сообщения: уравнение не имеет корней			
7	Конец			

Назначение первых двух граф таблицы очевидно. В последних трех графах ставятся номера блоков, в которые передается управление. Если все клетки этих граф пусты (строки 1 и 6), то по умолчанию переход идет к следующему блоку (строке). То есть после строки 1 выполняется строка 2, а после 6 выполняется 7. БП означает безусловный переход.

В чем прелесть табличных блок-схем?

1. Прежде всего, это полная свобода в написании. В любой блок можно записать неограниченный объем текста. Если текст не вмещается на одну строку, его можно продолжить на 2-, 3-й, да хоть на тысячной! Размер любого отдельно взятого блока, заполненного текстом, может занять несколько страниц. Здесь можно не только разместить сам алгоритм, но даже вещи, которые, казалось бы, совсем посторонние. Например, адрес и телефон сотрудника, у которого можно получить дополнительную информацию по данному блоку, ссылки интернета на нужные ресурсы и т. д.

2. После нескольких пробных репетиций написание (создание) всей блок-схемы даже очень сложной задачи будет выглядеть приятным времяпровождением. Просто пишем последовательно блок за блоком. Как только выходим на условный блок с двумя выходами, продолжаем по одному из двух направлений. Например, выбрали «*да*» – ставим в этой графе номер блока, пишем его и далее до какого-нибудь момента или логического конца. В любом

случае начинаем новый просмотр таблицы. Если в графе «Да» стоит номер блока, а на той же строке в графе «Нет» пусто, значит нужно вписать здесь номер того блока, которому будет передано управление по линии «*нет*».

3. Такую схему легко отлаживать. Сначала проверяем, что клетки условных блоков в графах «Да» и «Нет» заполнены переходными номерами. Затем проверяем целостность логических цепочек. Так, в случае $D = 0$ (рис. 4) срабатывает цепочка 1–2–5–7, а если $D > 0$, то 1–2–3–4–7.

4. Облегчаются написание текста программы и ее отладка. Можно устроить проверку или промежуточный просмотр в любом блоке, причем режим этой проверки можно и описать прямо в тексте самого блока в табличной блок-схеме. В графических блоках много текста не напишешь.

5. На первый взгляд, может показаться, что табличная блок-схема теряет наглядность, то есть то, во имя чего и придумана графическая блок-схема. Но здесь надо иметь в виду две вещи. Во-первых, наглядность, даже графическая, имеет смысл, если в схеме число блоков невелико, скажем пять-шесть, а если их десятки или сотни... Во-вторых, табличная блок-схема в наглядности и не нуждается. Зачем она? Наше дело проставить в последних трех графах таблицы *все* номера переходов. Все остальное сделает железная и жестокая логика.

6. Легко пишется и рисуется. Без всяких там прямоугольников, овалов, шестиугольников, стрелок и т. п. Хорошо документируется. Легко допускает коллективный труд, когда отдельные блоки можно раздать разным программистам и в тех же блоках записать имена исполнителей и графики их работы.

7. Можно сразу приступать к созданию табличной блок-схемы, а создав ее и отладив (то есть логически проверив), начинать программировать. Блок-схему следующей задачи сделаем сразу в табличной форме.

Задание – написать игровую программу из серии «Угадай число» по такому алгоритму. Играют двое, каждый по очереди с одной и той же программой. Каждый пытается угадать задуманное его противником число. Программа подсчитывает число неудачных попыток угадать. При этом она выдает слово «МНОГО» или «МАЛО» в зависимости от величины засекреченного числа. Побеждает тот, у кого неудачных попыток меньше.

Прошу прощения у эрудированных читателей за столь простой алгоритм. Но цель этой статьи – рассказать не о сути каких-либо задач, а об их схемной интерпретации. Сейчас самое главное – понять принцип записи табличных блок-схем. Посмотрите, как просто и изящно пишется эта блок-схема. Чтобы не усложнять задачу, договоримся, что угадывается целое трехзначное число из промежутка от 100 по 999 включительно.

Таблица 2

Табличная блок-схема игровой программы

№	Текст	БП	Да	Нет
1	Начало. На экране появляется надпись: «Играть будете?» Программа анализирует ответ. Если «*да*», то управление передается следующему блоку. Любой другой ответ воспринимается как «*нет*»		2	11
2	На экране появляется сообщение для одного играющего в данный момент, чтобы он ввел секретное число для другого игрока. Программа принимает это число и записывает его в ячейку *X*, после чего оно удаляется с экрана. Одновременно создается и обнуляется ячейка *N*, в которой будут подсчитываться неудачные попытки угадывания			

№	Текст	БП	Да	Нет
3	Соответствует ли принятое число X двум условиям: 1) оно целое; 2) $100 \leq X \leq 999$?		5	4
4	Печать на экране: «Число не соответствует правилам игры»	1		
5	Печать на экране: «Угадай число! Сообщи свой вариант мне». Подходит второй игрок и начинает угадывать число			
6	Программа принимает число от второго игрока и записывает его в ячейку Y. Проверка: $X = Y$?		11	7
7	$X < Y$?		8	10
8	Печать сообщения «МНОГО»			
9	Содержимое счетчика N увеличивается на 1	5		
10	Печать сообщения «МАЛО»	9		
11	Печать сообщения «Игра закончена. Число неудачных попыток угадать равно N			

Выводы

1. Введена нумерация всех логических блоков. Если управление передается другому удаленному блоку, нет необходимости вести к нему длинную стрелку. Можно ограничиться короткой стрелкой с номером блока в кружочке. Мне кажется, что это лет 20 или 30 назад уже было известно. Просто после просмотра в Яндексе многих сотен современных блок-схем я не обнаружил нумерации. В любом случае нумерация вещь очень полезная и ее следует применять.

Предлагаю договориться о нумерации первого и последнего блоков. Первый блок всегда должен иметь № 1, располагаться первым. Последний блок всегда должен быть расположен последним и иметь максимальный номер.

2. Из всех видов блок-схем выделен *канонический* вид, созданный с использованием законов логики. **С введением этого понятия можно, по-существу, начать создание самой теории блок-схем.** С переходом на *табличный* вид программисты получают серьезный инструментарий для увеличения производительности труда. Это проверялось мною в течение многих лет работы инженером-программистом. Простые программы писались сходу без блок-схем. Потом применялись блок-схемы. Но когда был осуществлен переход на табличный вид блок-схем, сразу возникло резкое увеличение эффективности метода.

Одно время автор статьи был руководителем группы программистов. Мы запустили в работу свой комплекс бухгалтерских программ на крупном предприятии. Каждый месяц заказчик вносил изменения в алгоритм действующей программы. Хорошо известно, что внесение изменений в действующий комплекс программ нужно проводить очень осторожно. Но благодаря описанной здесь системе все изменения проводились оперативно и удачно.

3. Следователи тоже могут воспользоваться нашими материалами. Если все рассуждения по сложному и запутанному делу преобразовать в канонический вид, то вряд ли разыскиваемый преступник сможет вырваться из железных объятий логики. Нужно лишь проявить нормальную аккуратность и не путать противоположные суждения с противоречащими.

4. Термин *инфраструктура теории* применен в данной статье, видимо, впервые. Во всяком случае поиски определения этого понятия в интернете оказались безуспешными. Возникает естественное желание дать толкование новому термину «*инфраструктура науки*», под которым следует понимать всю совокупность материалов, объектов и их описания, правила их использования. Например, аналитическая геометрия. Инфраструктура здесь – вся геометрия, правила алгебры. Даже соединенные вместе, они еще не являются аналитической

геометрией. Но вот задача: даны координаты двух точек, нужно найти расстояние между ними. Геометрия налицо. Применяем правила алгебры и выводим нужную формулу, которая уже является признаком *теории*.

Вопрос: а нужен ли этот термин науке? Есть ли в нем необходимость? Полагаем, что необходимость есть. В Википедии есть замечательное определение *науки* [3]. Об инфраструктуре там ничего не сказано. И дело не в том, что можно теперь написать нечто подобное формуле:

Наука = инфраструктура + теория.

Возьмем две науки: геологию и медицину. В свете сказанного становится очевидным, какова инфраструктура обеих наук, где и как появляется теория в каждой из них. Интересна история, инфраструктура которой – это летописи, документы, археологические раскопки и т. д. Но вот вопрос: может ли возникнуть новая наука, если известна (в какой-то степени) ее теория, но неизвестна инфраструктура? Можно ли восстановить инфраструктуру, без которой наука не мыслится?

Понимаем, что сказанное далеко от идеала. Но если ученый мир примет к сведению эту терминологию, тогда и будут кем-нибудь разработаны более четкие определения.

5. Представляется очевидным, что далее нужно написать учебное пособие по теории блок-схем, в котором следует отразить все то, что уже известно с учетом здесь сказанного. Любому автору, который согласится на такой труд, мы готовы со своей стороны оказать поддержку и готовы быть одним из рецензентов его будущего пособия.

Библиографический список

1. *Виноградов С. Н., Кузьмин А. Ф.* Логика. Учебник для средней школы. М. : Учпедгиз, 1954.

2. *Гетманова А. Д.* Учебник по логике. М. : Владос, 1955.

3. Наука // Википедия [Электронный ресурс]. Режим доступа : https://ru.wikipedia.org/wiki/%CD%E0%F3%EA%E0.

4. *Сорокина Н. И.* Английский язык для пользователей ЭВМ : практикум для студентов специальности 071800 «Мехатроника». Екатеринбург, 2009.