ИНФОРМАТИКА

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ СТРОИТЕЛЬНЫЙ УНИВЕРСИТЕТ

В.Г. Куликов, В.С. Евстратов

ТЕОРИЯ АЛГОРИТМОВ

Учебно-методическое пособие

ISBN 978-5-7264-2963-2

Москва
Издательство МИСИ – МГСУ
2022

УДК 510.5
ББК 22.12
К90

Рецензенты:
доктор технических наук, профессор *В.И. Римшин*, член-корреспондент РААСН;
кандидат технических наук, доцент *Н.А. Гаряев*,
доцент кафедры информационных систем, технологий и автоматизации в строительстве НИУ МГСУ

Куликов, Владимир Георгиевич.

К90 Теория алгоритмов [Электронный ресурс] : учебно-методическое пособие / В.Г. Куликов, В.С. Евстратов ; Министерство науки и высшего образования Российской Федерации, Национальный исследовательский Московский государственный строительный университет, кафедра информационных систем, технологий и автоматизации в строительстве. — Электрон. дан. и прогр. (1,2 Мб). — Москва : Издательство МИСИ – МГСУ, 2022. — Режим доступа: http://lib.mgsu.ru — Загл. с титул. экрана.

ISBN 978-5-7264-2963-2 (сетевое)
ISBN 978-5-7264-2964-9 (локальное)

В учебно-методическом пособии по дисциплине «Теория алгоритмов» представлены разделы, традиционно изучаемые в курсе теории алгоритмов: машины Тьюринга, нормальные алгоритмы Маркова, рекурсивные функции и т.д. Рассмотрены вопросы интуитивного и формального определения алгоритмов, сложности и нумерации алгоритмов, алгоритмически неразрешимых проблем, конструирования машин Поста.

Для обучающихся по направлению подготовки 09.03.02 Информационные системы и технологии.

Учебное электронное издание

Редактор *Л.В. Себова*
Корректор *В.К. Чупрова*
Компьютерная правка и верстка *О.В. Суховой*
Дизайн первого титульного экрана *Д.Л. Разумного*

Для создания электронного издания использовано:
Microsoft Word 2010, ПО Adobe Acrobat Pro.

Подписано к использованию 16.01.2022. Объем данных 1,2 Мб.

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Национальный исследовательский
Московский государственный строительный университет»
129337, Москва, Ярославское ш., 26.

Издательство МИСИ – МГСУ.
Тел. (495) 287-49-14, вн. 14-23, (499) 183-91-90, (499) 183-97-95.
E-mail: ric@mgsu.ru, rio@mgsu.ru

Оглавление

ВВЕДЕНИЕ 5
1. ОСНОВНЫЕ ПОНЯТИЯ И МОДЕЛИ АЛГОРИТМОВ 6
2. МАШИНА ТЬЮРИНГА 21
3. МАШИНА ПОСТА 22
4. РЕКУРСИВНЫЕ ФУНКЦИИ 23
5. АССОЦИАТИВНЫЕ ИСЧИСЛЕНИЯ 25
6. КЛАССЫ СЛОЖНОСТИ 26
7. ЛОГИЧЕСКИЙ СИНТЕЗ ВЫЧИСЛИТЕЛЬНЫХ СХЕМ 29
Библиографический список 37
ПРИЛОЖЕНИЕ 38

ВВЕДЕНИЕ

Современное формальное определение алгоритма было дано в 1930–1950-х годах в произведениях А. Тьюринга, Э. Поста, А. Черча, Н. Винера, А.А. Маркова.

Алгоритм (процедура) — решение задач в виде точных последовательно выполняемых предписаний. Это интуитивное определение сопровождается описанием интуитивных **свойств (признаков)** алгоритмов: эффективность, определенность, конечность [1].

Эффективность — возможность исполнения предписаний за конечное время.

Например, алгоритм — процедура, состоящая из «конечного числа команд, каждая из которых выполняется механически за фиксированное время и с фиксированными затратами»[1].

Функция может быть эффективно вычислена с помощью алгоритмов, если существует механическая процедура, из которой для конкретных значений ее аргументов может быть найдено значение этой функции.

Определенность — возможность точного математического определения или формального описания содержания команд и последовательности их применения в этой процедуре.

Конечность — выполнение алгоритма для конкретных исходных данных за конечное количество шагов.

Для доказательства алгоритмов в теории используются алгоритмические преобразования слов и фраз формального языка.

В формальных описаниях алгоритм конструктивно связан с концепцией машины, предназначенной для автоматизированных преобразований символьной информации.

Для автоматических расчетов разрабатываются модели алгоритмов распознавания языков и машина, которая работает с этими моделями. Таким образом, они сочетают математическое и формальное определение алгоритма с конструктивным, что позволяет создавать модели на компьютере.

Алгоритмические вычисления используются во всех областях науки и техники. Например, в монографии Д. Кнута [1] рассматривается множество проблемно ориентированных алгоритмических решений.

Интуитивно понятный графический метод построения алгоритмов в виде диаграмм и схем по-прежнему актуален и поддерживается стандартами языкового редактирования.

Объектно ориентированное обобщение алгоритмических языков позволяет включить алгоритмическое мышление и организовать крупномасштабное программирование с огромной производительностью и ресурсами компьютерной памяти, что постоянно увеличивает стандартные библиотеки.

Теория алгоритмических языков и компиляторов действительно связана с алгоритмами, но это независимая область знания и применения алгоритмов.

Общая теория алгоритмов обращается к проблеме эффективной вычислимости. Было разработано несколько формальных определений алгоритма, в которых эффективность и окончательность вычислений можно количественно оценить с помощью количества элементарных шагов и требуемого объема памяти.

Аналогичными моделями алгоритмических преобразований символьной информации являются:

– конечные автоматы;

– машина Тьюринга;

– машина Поста;

– ассоциативное исчисление или нормальные алгоритмы Маркова;

– рекурсивные функции.

Некоторые из этих моделей составляют основу методов программирования и используются в алгоритмических языках.

В современной **программной инженерии** алгоритмы как методы решения задач занимают видное место по сравнению с традиционной математикой. И неважно, есть ли в абстрактных алгоритмических моделях чистое алгоритмическое решение. Если необходимо решение проблемы, широко используются эвристики, и «доказательством» работоспособности алгоритма является его успешное тестирование [2].

[1] Ахо А. Теория синтаксического анализа, перевода и компиляции : в 2 т. Т. 1 / А. Ахо, Дж. Ульман ; пер. с англ. В.Н. Агафонова ; под ред. В.М. Курочкина. — Москва : Мир, 1978. — 612 с.

1. ОСНОВНЫЕ ПОНЯТИЯ И МОДЕЛИ АЛГОРИТМОВ

Для формального описания алгоритма необходимо формальное описание решаемой проблемы. В большинстве случаев описание проблемы носит неформальный (вербальный) характер, следовательно, переход к алгоритму неформальный и требует проверки и тестирования, а также нескольких итераций для приближенного решения.

Верификация (от лат. verus — *истинный* и facere — *делать*) — способ подтверждения любых теоретических положений, алгоритмов, программ и процедур путем сравнения их с экспериментальными данными (справочными или эмпирическими), алгоритмами и программами.

Тестирование используется для определения соответствия объекта испытаний указанным спецификациям.

Теория алгоритмов не может предоставить универсального и формального способа описания проблемы и ее алгоритмического решения. Однако ценность теории состоит в том, что она дает примеры таких описаний и определения алгоритмически неразрешимых задач.

Один из примеров — на обычном языке, и задача формулируется как разработка алгоритма распознавания принадлежности любого предложения к определенному регулярному языку. Доказано, что регулярный язык можно формально преобразовать в **модель алгоритма** для решения этой проблемы за конечное число шагов. Эта модель представляет собой **конечный автомат**.

Регулярные языковые расширения используются: при описании словаря формальных алгоритмических языков и алгоритмов, при исследовании шаблонов редактирования, в современном программировании (Java, Phyton, C#, PHP, JS), в компиляторах и системах управления обработкой данных, в качестве интерактивных языков в операционных системах. Следовательно, конечный автомат может использоваться для алгоритмического решения проблем в этих областях.

Алфавит языка обозначается как конечное множество символов. Например: $\sum = \{a, b, c, d\}$, $\sum = \{0, 1\}$.

Символ и цепочка символов образуют слово — *a*, *b*, 0, *abbcd*, 0111000. Пустое слово **(*e*)** не содержит символов.

Множество слов $S = \{a, ab, aaa, bc\}$ в алфавите Σ называют языком $L(\Sigma)$.

Язык $S = L(\Sigma)$ может содержать неограниченное количество слов.

Для их определения используются различные формальные правила. В простейшем случае это **алгебраическая формула**, которая содержит операции по формированию слов из символов алфавита и ранее полученных слов.

Рассмотрите следующие шаги, чтобы сформировать новые наборы из существующих наборов слов.

1. Символы алфавита могут быть связаны путем **конкатенации** (присоединения) к строкам словесных символов, связанных с новыми словами.

Конкатенация двух слов *x*|*y* обозначает, что к слову *x* справа приписано слово *y* или *x*|*y* = *xy*, причем $xy \neq yx$.

Произведение $S_1|S_2 = S_1S_2$ множеств слов S_1 и S_2 — это множество всех различимых слов, построенных конкатенацией соответствующих слов из S_1 и S_2.

Если $S_1 = \{a, aa, ba\}$, $S_2 = \{e, bb, ab\}$, то $S_1S_2 = \{a, aa, ba, abb, aabb, baab, ...\}$. Для конкатенации выполняется ассоциативность, но коммутативность и идемпотентность не выполняются: $S_1S_2 \neq S_2S_1$; $SS \neq S$.

2. **Объединение** $(S_1 \cup S_2)$ или $(S_1 + S_2)$ множеств:

$S_1 = \{a, aa, ba\}$, $S_2 = \{e, bb, ab\}$, $S_1 \cup S_2 = \{a, aa, ba, e, bb, ab\}$. Для операции объединения выполняются следующие законы:

коммутативность объединения: $S_1 \cup S_2 = S_2 \cup S_1$;

идемпотентность объединения: $S \cup S = S$;

ассоциативность объединения: $S_1 \cup (S_2 \cup S_3) = (S_1 \cup S_2) \cup S_3$;

дистрибутивность конкатенации (умножения) и объединения: $S_1(S_2 \cup S_3) = S_1S_2 \cup S_1S_3$.

3. **Итерация множества** $\{S\}^*$ состоит из пустого слова и всех слов вида $S^\wedge$: $S^1 = S$, $S^2 = SS$, $S^3 = SSS$.

Формулы, содержащие эти операции с множествами слов, называют **регулярными выражениями**.

Ассоциативность итерации: $S_1{*}(S_2{*}S_3) = (S_1{*}S_2){*}S_3$.

Дистрибутивность объединения с итерацией: $S_1{*}(S_2 \cup S_3) = S_1{*}S_2 \cup S_1{*}S_3$. Если a, b — любые регулярные выражения, то $(a \cup b)^* = (a^* \cup b^*)^* = (a^*b^*)^* = (a^*b)^*a^*$; $a^* = a^*a^* = (a^*)^* = (a \cup a^2 \cup \ldots \cup a^k)^*$; $(a^*b)^* = (a \cup b)^*b$.

Следовательно, формулы могут содержать круглые скобки и могут быть преобразованы с использованием этих законов.

Регулярные выражения допускают формальные алгебраические преобразования.

Языки, определяемые регулярными выражениями, называются регулярными языками, а набор слов — регулярными множествами.

Пример 1.1

Регулярные выражения регулярного языка в алфавите Σ = {0,1}:

(0 ∪ (1(0)*)) = 0 ∪ 10*;

(0 ∪ 1)* = (0* ∪ 1*)*;

(0 ∪ 1)*011 — все слова из 0 и 1, заканчивающиеся на 011;

(a ∪ b)(a ∪ + b)* = (a ∪ b)(a* ∪ b*)* — слова, начинающиеся с a или b;

(00 ∪ 11)*((01 ∪ 10)(00 ∪ 11)*(01 ∪ 10)(00 ∪ 11)*)* — все слова, содержащие четное число 0 и 1.

Пример 1.2

Входной алфавит: Σ = {i, +, –},

где i — идентификатор;

(+, –) — знаки арифметических операций.

Примеры правильных арифметических выражений:

i, – i, i + i, i – i, – i – i, i + i – i, ...

Обозначим знаки арифметических действий буквами p = (+), m = (–).

Тогда соответствующие правильные (регулярные) арифметические выражения имеют вид i, mi, ipi, imi, $mimi$, $ipimi$, ... Регулярное выражение, определяющее регулярный язык:

$L(M) = (mi + i)((p + m)i)^*$.

Утверждение. Для каждого регулярного множества (языка $L(\Sigma)$) можно построить по крайней мере одно регулярное выражение, но для каждого регулярного выражения существует только одно регулярное множество.

Основные модели алгоритмов

Алгоритм распознавания предложений регулярного языка называют конечным автоматом **(КА)**.

Конечный автомат определяется символами: $M = (Q, \sum, \delta, q_0, F)$,

где $Q = \{q_0, q_1, \ldots, q_n\}$ — конечное множество состояний;

$\sum = \{a, b, c, \ldots\}$ — входной алфавит (конечное множество);

δ: $Q^* \sum \rightarrow \{P_j\}$ — функция переходов, P_j — подмножество Q.

Конечное множество значений для этого функционального отношения может быть определено перечислением в **таблице переходов** (табл. 1.1).

Таблица 1.1

Q	$\sum$	P_j
q_i	a	q_j

q_0 — начальное состояние;

F — множество заключительных состояний.

Конечный автомат называется **недетерминированным** (НДКА), если P_j содержит более одного состояния.

КА называется **детерминированным** (ДКА), если P_j содержит не более одного состояния.

КА **полностью определенный**, если P_j в детерминированном автомате не пустое. Если есть пустые элементы множества P_j, то автомат **частично определен**.

Работа КА или выполнение обычного алгоритма распознавания слов на языке может быть представлена последовательностью шагов, которые определяются текущим состоянием Q, входным символом $\sum$ и последующим состоянием P_j.

Используется конструктивное описание принципа работы КА, например машины M, со следующей организацией (рис. 1.1).

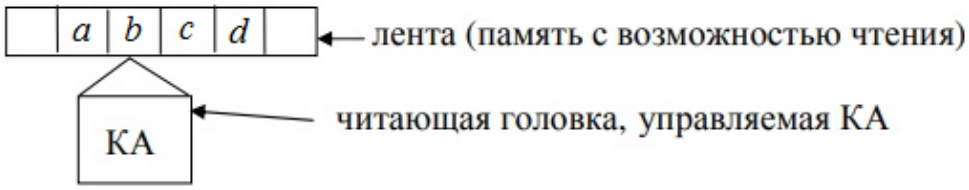


Рис. 1.1. Машина, исполняющая конечный автомат

КА считывает символ для входа в текущее состояние $q_i \in Q$, переходит в следующее состояние $q_j \in Q$ и меняет считывающую головку на следующий символ.

Автомат принимает входное слово, если оно достигает конечного состояния F, последовательно считывая символы из памяти и переходя в следующие состояния согласно таблице переходов. В этом случае входное слово заканчивается и автомат останавливается.

Конфигурация КА: $k = (q, \omega)$, где q — текущее состояние КА, ω — непрочитанная цепочка символов слова на ленте, включая символ под читающей головкой;

$k = (q, \omega)$ — текущая конфигурация;

$k_0 = (q_0, \omega_0)$ — начальная конфигурация;

$k_f = (q, e)$ — заключительная конфигурация, $q \in F$, (e) — символ, обозначающий конец строки.

Шаг алгоритма — переход из одной конфигурации КА в другую:

$K_i \to K_j$ или $(q_i, \omega_i) \to (q_j, \omega_j)$.

Функция переходов, заданная в табличной форме, может быть представлена графом переходов $kG = (Q, R)$, где Q — вершины графа, R — бинарное отношение между парой вершин, которое представлено множеством дуг (q_i, q_j).

$(q_i, q_j) \in Q^*Q$, если существует символ $a \in \sum$ и $\delta(q_i, a) = q_j$. На дугах графа (q_i, q_j) отмечаются соответствующие символы алфавита.

Пример 1.3

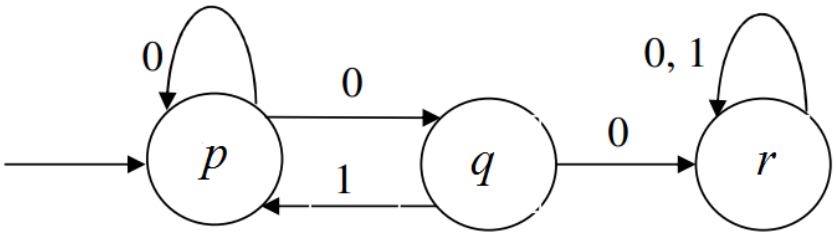


Рис. 1.2. Детерминированный читающий КА

Для ДКА, приведенного на рис. 1.2: состояния $Q = \{p, q, r\}$, входной алфавит $\sum = \{0, 1\}$, начальное состояние p, конечное — r, таблица переходов указана в табл. 1.2.

Таблица 1.2

Q	Σ	
	0	1
p	q	p
q	r	p
r	r	r

Исполнение алгоритма — это последовательность шагов, в которых изменяется конфигурация КА:

$(p, 01001) \rightarrow (q, 1001) \rightarrow (p, 001) \rightarrow (q, 01) \rightarrow (r, 1) \rightarrow (r, e)$,

где $(p, 01001)$ — начальная конфигурация;

(r, e) — конечная конфигурация.

В результате применения слова 01001 в начальном состоянии p автомат переходит в следующее состояние q и следующее значение цепочки символов на входе 1001.

Автомат *M* допускает слово ω_0, если существует $(q_0, \omega_0) \rightarrow {}^*(q_f, e)$, где $\rightarrow {}^*( )$ обозначает **транзитивное замыкание** и существует путь, соединяющий q_0 и q_f для входного слова ω_0.

Язык *L*(*M*), определяемый (распознаваемый, допускаемый) автоматом *M*, включает множество всех слов, допускаемых *M*.

Преобразование регулярных выражений в конечный автомат

Утверждение. Язык *L* является регулярным тогда и только тогда, когда он определяется КА. Для любого регулярного языка, представленного регулярным выражением, можно построить КА — распознаватель слов, допустимых языком.

Метод Ямады преобразования *L*(*M*) → *M* позволяет формально выполнить преобразование [3; 4]:

1) разметка состояний устанавливает позиции символов в регулярном выражении (пример 1.2):

$L(M) = (mi + i)\big((p + m)i\big)^*0123456$;

2) рассмотрим 2^6 + 1 подмножеств мест предположения состояния и сформируем регулярным выражением возможные переходы между состояниями. Явная избыточность состояний уже устранена, так как для некоторых состояний нет переходов.

Общие оценки числа состояний и переходов КА:

– число переходов (ребер графа) КА равно (*n* + 1), где *n* — число букв в регулярном выражении (в данном примере *n* = 7 состояний);

– в полностью определенном КА нижнюю границу числа состояний m определяем из условия $m^*s = n + 1$, где s — число символов входного алфавита (в данном случае $3m = 7$ и $m =]7/3[= 3$);

– если $m^*s > n + 1$, то автомат частично определенный и верхняя граница $m \leq n + 1$ — количество позиций в регулярном выражении (в данном примере m = 7).

Для рассматриваемого примера можно построить КА с использованием верхней оценки (рис. 1.3).

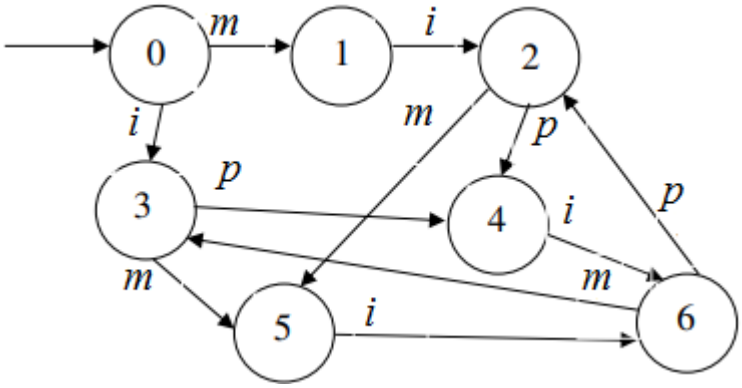


Рис. 1.3. Преобразование регулярного выражения в КА

Утверждение. Для каждого регулярного множества существует определяющий его КА с минимальным числом состояний.

Слово *различает* состояния q_i и q_j, если $(q_i, \omega) \to {}^*(q_m, e)(q_j, \omega) \to {}^*(q_n, e)$ и одно из состояний q_m или q_n принадлежит F.

Состояния k *не различимы*, если они не различимы цепочкой длины k.

Состояния не различимы (*эквивалентны* $q_i = q_j$), если они не различимы при $k \to \infty$.

В автомате на рис. 1.3 каждую пару эквивалентных состояний $\{2, 3\}$, $\{4, 5\}$ заменяем одним состоянием $\{2, 3\} \to 2$, $\{4, 5\} \to 3$.

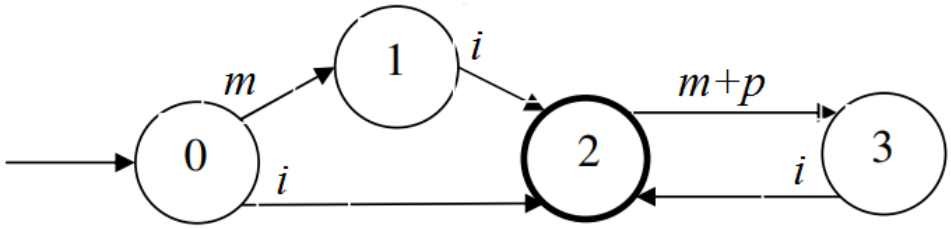


Рис. 1.4. ДКА распознавания $L(M)$

Рассмотренную технику преобразования можно упростить, если учесть очевидные алгебраические свойства операций с регулярными выражениями:

– строки в круглых скобках с операцией (+) имеют общее начальное состояние перед открывающей круглой скобкой и конечное состояние после закрывающей круглой скобки;

– итерация обозначает цикл с произвольным количеством повторений, с пустым количеством циклов сохраняется состояние, с которым мы вошли в цикл;

– при конкатенации нескольких символов различимые состояния заменяют конкатенацию между парой символов:

$L(M) = (mi + i)\big((p + m)i\big)^*01232.$

Состояния размещаем в найденные для них позиции, исключая повторения:

$L(M) = (mi + i)\big((p + m)i\big)^* = 0(m1i + i)2\big((p + m)3i2\big)^*01232.$

Заменяем линейную помеченную строку графом КА (рис. 1.4). Начальное состояние — 0, конечное состояние — 2.

В общем случае регулярное выражение может быть преобразовано в недетерминированный КА (НДКА).

Пример 1.4

Рассмотрим НДКА распознавания $L(M)$ (рис. 1.5).

$L(M) = aa*bd* + ad* = (aa*b + a)d* = (0a_1(a_1)*b + 0a)2(d_2)*012.$

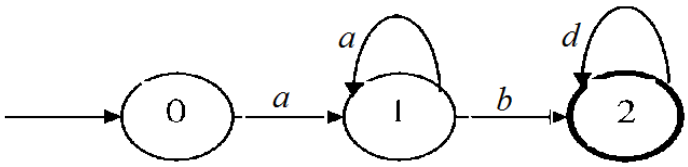


Рис. 1.5. НДКА распознавания $L(M)$

При выполнении алгоритма (НДКА) по методу поиска по образцу сохраняются номера позиций, в которых можно выбрать несколько переходов. При неудачном поиске осуществляется возврат на ближайшую позицию (назад). Шаги по разным ветвям выполняются параллельно:

(0, *add*) → (1, *dd*)

→ (2, *dd*) → (2, *d*) → (2, *e*).

Пример 1.5

Всякая подцепочка из трех символов содержит два нуля:

$\big((001)*(100)*(010)*\big)*$.

Число 0 делится на 2, а число 1 — на 3:

$\big((00)*(111)*\big)*$ = (00 + 111).

Множество цепочек, в которых каждая пара 00 находится перед парой:

$11\big((00)(11)*01*\big)*$.

Множество цепочек, в которых количество единиц четно:

(0* + 11 + 101)*.

Утверждение. Два автомата M и M' эквивалентны, если допускают один и тот же язык $L(M) = L(M')$.

Утверждение. Для НДКА M можно построить эквивалентный ДКА M'.

Преобразование представим следующей процедурой:

– на i-м шаге множество состояний ДКА обозначим Q_i;

– Q_0 обозначает множество состояний в НДКА;

– находим различимые подмножества следующих состояний для всех условий, применяемых к Q_i, включаем их в Q_{i+1}. Итерация повторяется, пока $Q \neq Q_{i+1}$.

Лемма. Число состояний в ДКА не превышает $2^n - 1$, где n — число состояний в НДКА.

Число $2^n - 1$ — количество собственных подмножеств для множества, состоящего из n различимых элементов.

Следовательно, процедура конечна.

Пример 1.6

Эквивалентный детерминированный автомат (рис. 1.6) содержит четыре состояния и определен тем же регулярным выражением $L(M') = L(M)$.

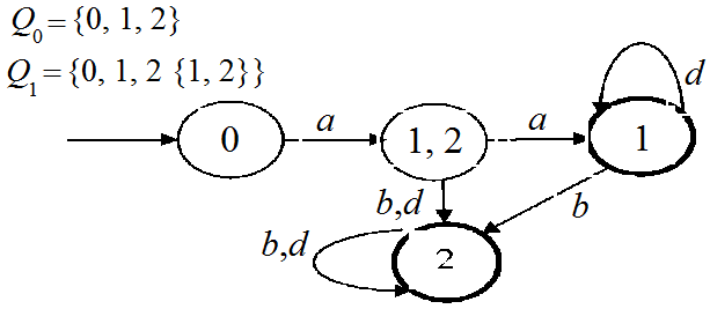


Рис. 1.6. ДКА получен из НДКА

Пример 1.7

НДКА задан таблицей переходов (табл. 1.3).

Таблица 1.3

Переходы	0	1
P	q, s	q
Q	r	q, r
R	s	p
S	q	p

Преобразование в ДКА:

$Q_0 = \{p, q, r, s\}$;

$Q_1 = \{p, q, r, s, \{q, s\}, \{r, q\}\}$;

$Q_2 = \{p, q, r, s, \{q, s\}, \{r, q\}, \{r, s\}\}$;

$Q_3 = \{p, q, r, s, \{q, s\}, \{r, q\}, \{q, r, p\}, \{r, s\}\}$;

$Q_4 = \{p, q, r, s, \{q, s\}, \{r, q\}, \{q, r, p\}, \{r, s\}, \{q, r, s\}\}$;

$Q_5 = Q_4$;

$Q_4 = \{p, q, r, s, \{q, s\}, \{r, q\}, \{q, r, p\}, \{r, s\}, \{q, r, s\}\} = \{P, Q, R, S, X, Y, Z, M, F\}$.

Таблица переходов КА представлена в табл. 1.4.

Таблица 1.4

0		1
P	X	Q
Q	R	Y
R	S	P
S	Q	P
X	Y	Z
Y	M	Z
Z	F	Z
M	X	P
F	F	Z

ДКА содержит девять состояний. Если S — конечное состояние в НДКА, то в ДКА выделены конечные состояния $\{S, X, M, F\}$, содержащие S.

Преобразование конечного автомата в регулярное выражение

Утверждение. Для любого конечного читающего автомата M можно построить регулярное выражение $L(M)$.

Для допускающего (финального) состояния исключим промежуточное состояние (кроме начального q_0) следующим образом.

Пусть q_i — предшествующее состояние, q_s — промежуточное и q_j — следующее, таким образом:

1) над каждой дугой (q_i, q_s) запишем регулярное выражение Q_{is};

2) на дуге (q_s, p_j) — выражение Q_{sj};

3) петлю в s обозначим регулярным выражением S^*;

4) дугам (q_i, q_j) после исключения промежуточного состояния q_s припишем регулярные выражения $Q_{is}S^*Q_{sj}$.

После удаления всех промежуточных вершин исходное и конечное состояния остаются. Конечные состояния объединяются путем добавления соответствующих регулярных выражений. В частном случае, если $q_0 \in F$, состояние q_0 останется, и дуге будет присвоено регулярное выражение.

Когда остаются два состояния, начальное и конечное, промежуточное регулярное выражение можно записать как $(R + SU^*T)^*SU^*$.

Полученный КА представлен на рис. 1.7.

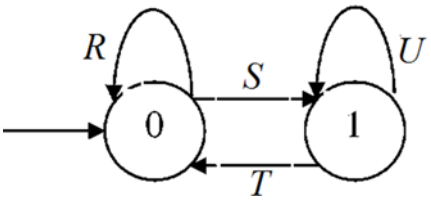


Рис. 1.7. Формирование регулярного выражения

Пример 1.8

Преобразуем ДКА в регулярное выражение (рис. 1.8).

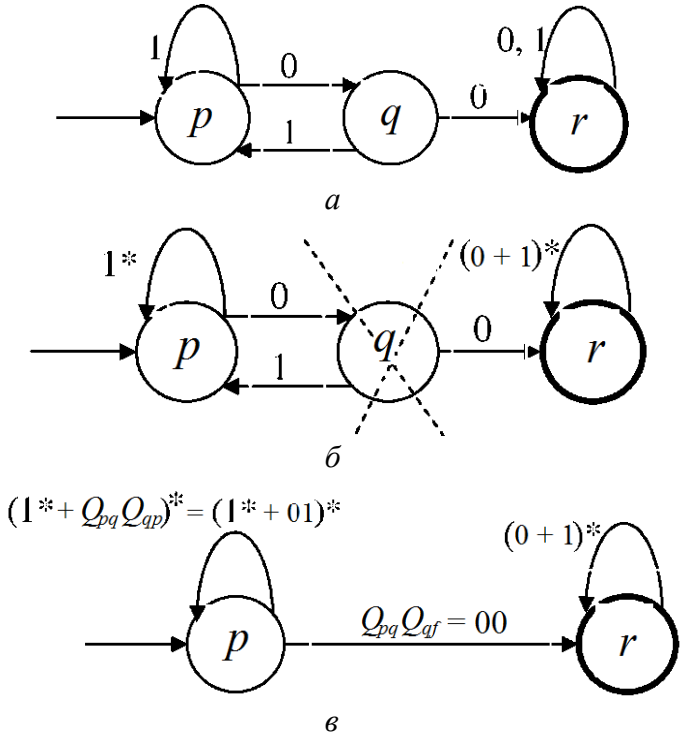


Рис. 1.8. Три стадии преобразования КА в регулярное выражение:
а — исходный КА; *б* — исключение состояния; *в* — свертка для двух состояний

Завершающий шаг — соединение (конкатенация) выражений для начального и финального состояний:

$R = (1^* + Q_{pq}Q_{qp})^* = (1^* + 01)$;

$U = (0 + 1)^*$;

$S = Q_{pq}Q_{qf} = 00$;

T отсутствует;

$(R + SU^*T)^*SU^* = (1^* + 01)^*00(0 + 1)^*$;

$L(M) = (1^* + 01)^*00(0 + 1)^*001023$.

Состояния q_i и q_j связаны дугой (q_i, q_j), если существует смежный символ в регулярном выражении. В этом случае дуга помечается соответствующим символом.

Получен НДКА:

$Q_0 = \{q_0, q_1, q_2, q_3\}$;

$Q_1 = \{q_0, q_1, q_2, q_3, \{q_{1,2}\}\}$.

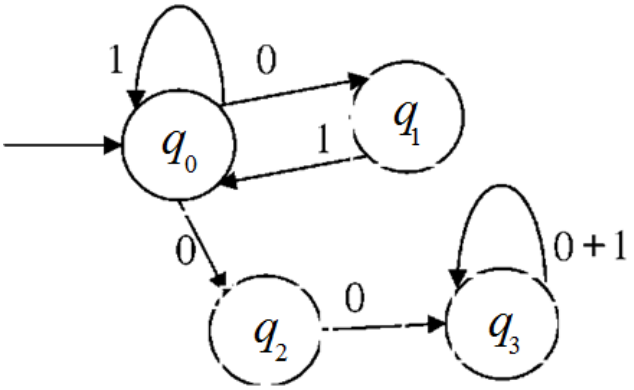


Рис. 1.9. Преобразование КА
в регулярное выражение и обратно в КА

Состояния q_1 и q_2 не достижимы из начального состояния q_0 и могут быть исключены. Получим исходный КА (рис. 1.9) с состояниями $Q_1 = \{q_0, q_3, \{q_{1,2}\}\}$, что подтверждает корректность преобразования КА в регулярные выражения.

Распознавание нерегулярных языков

Нерегулярные языки различаются, потому что есть ограничения на повторение символов.

Например, $L(M) = \{0^n 1^n$, где $n \geq 0\}$.

Для сохранения числа нулей требуется память.

Для распознавания принадлежности некоторого языка к классам регулярных или нерегулярных языков используется лемма о накачке.

Лемма о накачке. Пусть L — регулярный язык. Тогда существует константа n, для которой каждое слово языка L длиной $m \geq n$ можно разделить на три слова xyz так, что длина $xy \leq n$ и для любого $k \geq 0$ слово $xy^k z$ принадлежит L.

Таким образом, если слово y повторить (накачать) любое число раз ($k \geq 0$), то слово $xy^k z$ принадлежит регулярному языку L и не принадлежит нерегулярному.

Так, например, для $L(M) = \{0^n 1^n$, где $n > 0\}$, если $k = n = 5$, то $0^5 1$ не принадлежит $L(M)$.

Примеры нерегулярных языков:

– все двоичные коды, длины которых являются простыми числами;

– набор двоичных кодов, которые начинаются с единицы и представляют собой простые целые числа.

Распознавание нерегулярных языков рекурсивным запуском конечного автомата

Система S состоит из двух и более КА: A_1, ..., A_n. При этом:

– один из автоматов A_1 начальный;

– входной алфавит автоматов расширен символами, обозначающими начальные состояния других автоматов.

Рекурсивный КА вызывает сам себя.

Пример 1.9

Пусть система состоит из однотипных автоматов:

$L(\Sigma), \Sigma = \{a, b, L\}$;

$L(A) = a^*b + a^*L^*b$, где L запускает новый образец автомата.

Нерегулярный язык, распознаваемый рекурсивным вызовом автомата L (рис. 1.10):
$L(S) = ab + aabb + aaabbb + \ldots = \{a^n b^n\}$, где $n > 0$.

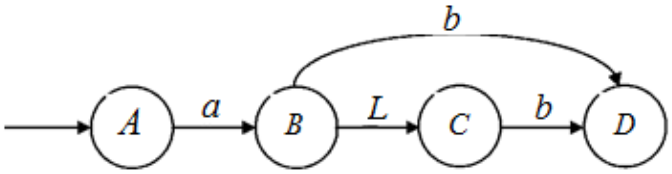


Рис. 1.10. Рекурсивный КА запускает себя в L

Работа КА при исполнении алгоритма распознавания нерегулярного язык происходит следующим образом (рис. 1.11):

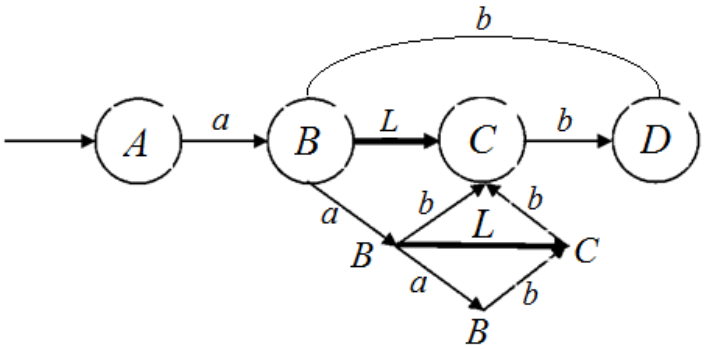


Рис. 1.11. Распознавание нерегулярного языка рекурсивным КА

Конечные автоматы с выходом

Для автоматического распознавания слов на регулярном языке необходимы внешние сигналы состояний, в которых автомат находится после завершения чтения слова:

– КА должен быть полностью определен, дополнен состоянием Z, в которое автомат переходит, если слово не применимо к автомату;

– во входном алфавите Z^* есть символ e, заключающий в себе любое слово;

– на выходе автомата формируется символ алфавита W, который по последнему символу идентифицирует состояние, в которое приходит КА.

Рассмотрим КА (рис. 1.12). При доопределении $Q = \{A, B, F, D, Z\}$, $1^* = \{a, b, d, e\}$, $W = \{\alpha, \beta, \gamma\}$.

$W = \{\alpha, \beta, \gamma\}$ = {начальное состояние α, завершение ввода β (слово не принадлежит регулярному языку), слово принадлежит регулярному языку γ}.

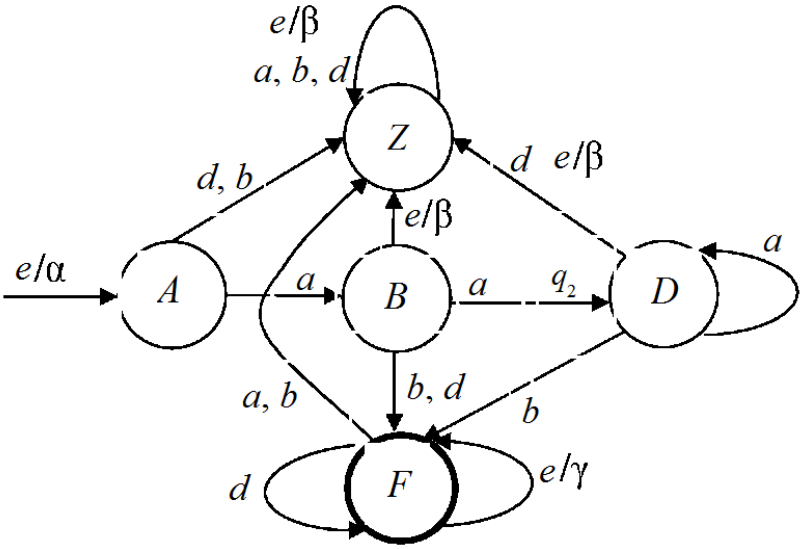


Рис. 1.12. КА с выходным алфавитом

Событийная интерпретация конечного автомата с выходом

Схема машины для выполнения алгоритма распознавания может быть определена путем интерпретации табличного описания КА событиями в виде логических уравнений.

Событиями являются предикаты:

– идентификации и выбора состояний A = (is A);

– идентификации входов a = (is a);

– выбора выходов a = (is a).

Уравнения перехода в состояния:

$A' = Ae$;

$B' = Aa \vee Ba$;

$D' = Ba \vee Da$;

$Z' = a(b \vee d) \vee Be \vee D(d \vee e) \vee Z(a \vee b \vee d \vee e)$;

$F' = B(b \vee d) \vee F(d \vee e)$.

Уравнения выхода:

$\alpha = Ae$;

$\beta = Be \vee Ze \vee De$;

$\gamma = Fe$.

При этом для выполнения КА решается система уравнений для заданного входного слова, работа заканчивается формированием выхода W.

В итерационных вычислениях с использованием уравнений необходимо различать переменные, представляющие текущее и следующее состояния. Например, в $B' = Aa \vee Ba$ справа — текущие состояния, а слева — следующие.

Более простая схема вычислений в виде Switch-программы рассмотрена в [5], где Q — числовая переменная, нумерующая состояния $Q = \{A, B, D, F, Z\} = \{0, 1, 2, 3, 4\}$.

При вычислениях решается система уравнений с переключателем состояний Q (рис. 1.13).

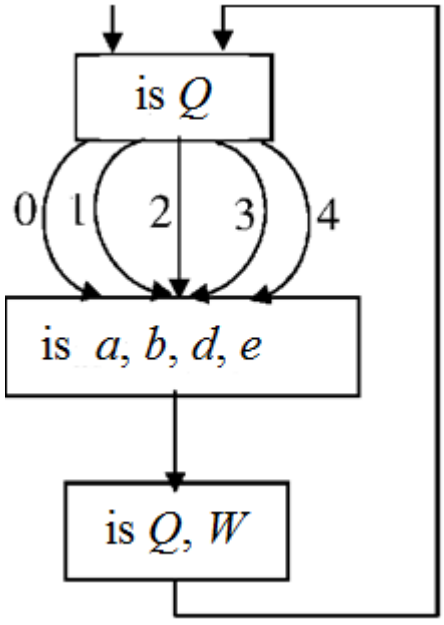


Рис. 1.13. Switch-программа интерпретации КА
системой уравнений в событиях

Записывающие конечные автоматы

Из-за ограниченного количества конечных автоматов с выводом можно определять алгоритмы распознавания, а также выполнять простые операции для преобразования (редактирования) слов в языке. Символы для этих операций представляют собой команды для ручного преобразования или вводятся для формирования записанных выходных слов.

Конструкцию КА для этой цели можно улучшить, добавив записывающую ленту (записывающую память) (рис. 1.14).

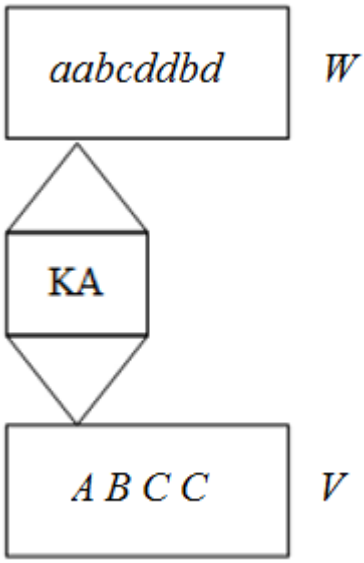


Рис. 1.14. Записывающий КА,
где W — цепочка символов,
считываемая с входной памяти (ленты),
V — цепочка символов, записываемых
в выходную память (ленту)

Конфигурация автомата $K(q_i, \omega, w)$ включает текущее состояние, оставшуюся входную цепочку символов и выходную строку символов.

Пример 1.10

Разработаем КА, реализующий преобразователь, устраняющий избыточные операции в арифметических выражениях.

Входной алфавит $\sum = \{i, +, -\}$, выходной алфавит — соответствующие подстановки, заменяющие входные символы в цепочках регулярного языка арифметических выражений.

Обозначим арифметические операции символами $p(+), m(-)$.

Пример цепочки входного регулярного языка, для которой необходимо редактирование: $-i+-i-++-i$ = *mipmimppmi*, регулярное выражение, для которого требуется редактирование:

$L(M) = i\text{^}2*(p+m)\text{^}3$;

$L(M = (p+m)*i*((p+m)*(p+m)*i)$.

Строка должна быть преобразована в $-i-i+i = mimipi$ следующими подстановками x/W. При этом символы $W = \{i, e, mi\}$ записываются в выходной памяти при чтении входного слова регулярного языка: p/e — нет записи (фактически стирание во входной строке), i/mi — запись mi в выходную память.

Выходной регулярный язык в примере $L(M) = (mi+i)\big((p+m)i\big)*$.

КА-преобразователь исходного предложения представлен на рис. 1.15.

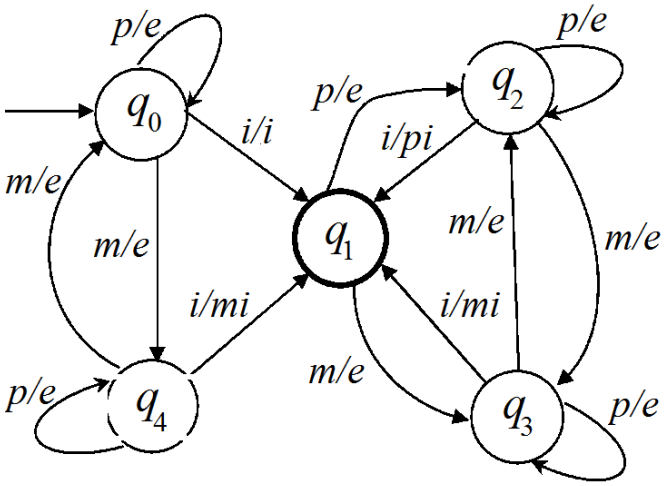


Рис. 1.15. Записывающий КА

Применение конечных автоматов в программировании

Конечные автоматы применимы в качестве алгоритмических моделей для непосредственного программирования задач управления объектами.

Основная проблема, возникающая при программировании в модели автомата, — определение набора состояний КА и входного алфавита. Эту проблему легко решить, преобразовав блок-схему программы в КА.

Утверждение. Существует конструктивный метод преобразования блок-схем алгоритмов в КА.

Рассмотрим преобразование блок-схемы алгоритма умножения $S = A*B$, построенной по школьному методу, где $A = (a_{n-1}a_{n-2}\ldots a_1a_0)$; $B = (b_{n-1}b_{n-2}\ldots b_1b_0)$; n — разрядные десятичные числа.

Произведение S вычисляется суммированием частичных произведений со сдвигом вправо на один десятичный разряд:

$$
\begin{array}{l}
A^* b_{n-2} \\
\quad A^* b_{n-1} \\
\qquad \cdots \\
\qquad\quad A^* b_1 \\
\qquad\qquad A^* b_0 \\
\hline
S_{2n-1}\, S_{2n-2} \ldots S_0
\end{array}
$$

Вычисление частичных произведений, сдвиг и суммирование считаются эффективными операциями.

Схема алгоритма умножения и его преобразования в КА представлена на рис. 1.16.

Операторные вершины, а также начальная (ввод) и конечная (вывод) обозначаются как состояния КА и имеют смысл задержки в выполнении соответствующих операций. Алгоритм может выполнять указанные операции или генерировать команды для их автоматического выполнения на машине. Таким образом, множество состояний равно $Q = \{q_0, q_1, \ldots, q_5\}$. Входной алфавит определяет набор символов, который кодирует предсказания, выраженные в условных вершинах, и получает двоичные значения {true, false}. Переходы из одного состояния в другое могут быть представлены сопряжением значений входных переменных $\{x_1, x_2\}$.

Функция выходов, формируемых в состояниях, обозначим символами-командами выходного алфавита $W = \{\alpha, \beta, \gamma, \delta, \varphi\}$ и поставим им в соответствие состояния КА.

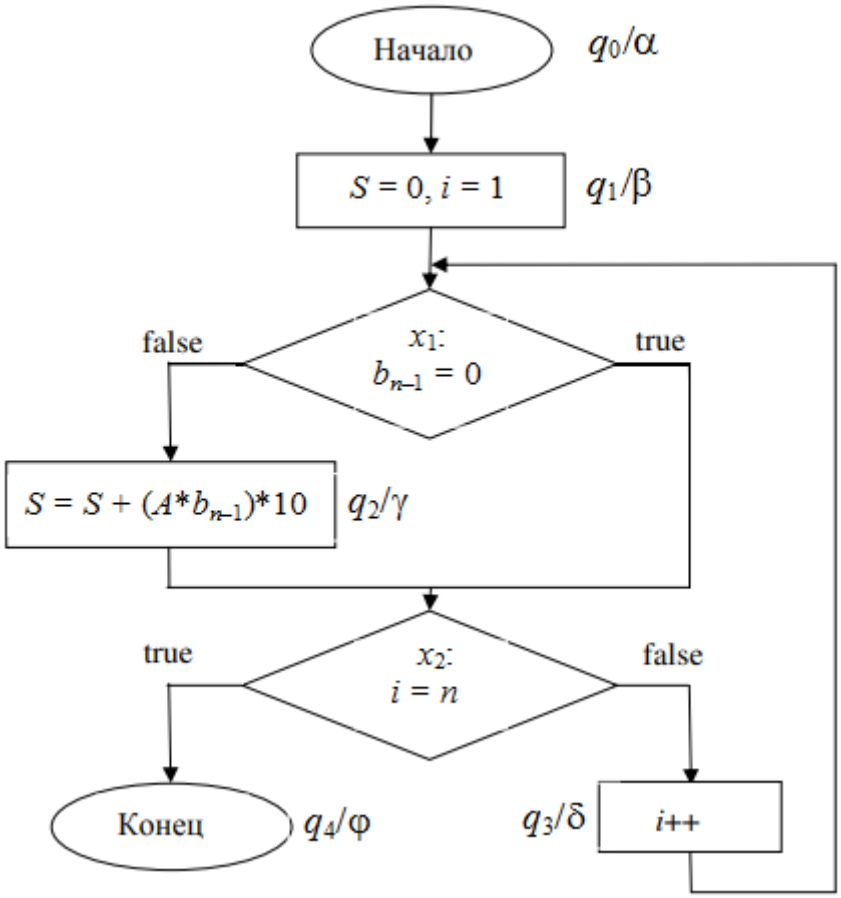


Рис. 1.16. Преобразование блок-схемы в КА

Функция переходов КА определяется в табл. 1.5.

Таблица 1.5

Переходы	q_0	q_1	q_2	q_3	q_4	q_5
q_0	–	T	–	–	–	–
q_1	–	–	$\bar{x}_1$	–	–	$\bar{x}_1$
q_2	–	–	–	$\bar{x}_2$	$\bar{x}_2$	–
q_3	–	–	$\bar{x}_1$	–	–	$\bar{x}_1$
q_4	–	–	–	–	–	–
q_5	–	–	–	$\bar{x}_2$	$\bar{x}_2$	–

Метод преобразования блок-схем в КА хорошо известен и апробирован в основном для дальнейших преобразований алгоритмов в схемы.

Следовательно, модель алгоритма в виде КА легко может быть преобразована в блок-схему, и тогда всегда выполняется переход к формальному описанию алгоритма на алгоритмическом языке и в виде программы ЭВМ.

Частичные КА могут использоваться в преобразованиях алгоритмов, представленных блок-схемами.

Важность существования частичных автоматов в программировании зависит от типа задачи, решаемой алгоритмическим методом:

– для распознавателей языка — ограничение обычного языка;

– для алгоритмов преобразования данных — ограничение диапазона значений данных, т.е. результатов преобразования;

– для алгоритмов управления — ограничение входных данных.

В случае распознавателей свойства частичных автоматов могут использоваться для управления и идентификации неязыковых предложений во время тестирования.

В других случаях для тестирования также применяются частичные автоматы. В реальных программах переходы можно переопределить любым способом, потому что правильные переходы в четко налаженной программе невозможны. Плагин можно использовать для мониторинга и тестирования программ во время выполнения.

Выполним замену логических условий символами входного алфавита $\Sigma^* = \{T, a, b, c, d\}$, где $a = \bar{x}_1$, $b = x_1$, $c = \bar{x}_2$, $d = x_2$; q_0 — начальное состояние, q_4 — финальное состояние (табл. 1.6).

Таблица 1.6

Переходы	q_0	q_1	q_2	q_3	q_4	q_5
q_0	–	T	–	–	–	–
q_1	–	–	a	–	–	b
q_2	–	–	–	c	d	–
q_3	–	–	a	–	–	b
q_4	–	–	–	–	–	–
q_5	–	–	–	c	d	–

Модель алгоритма умножения в символах входного алфавита приведена на рис. 1.17.

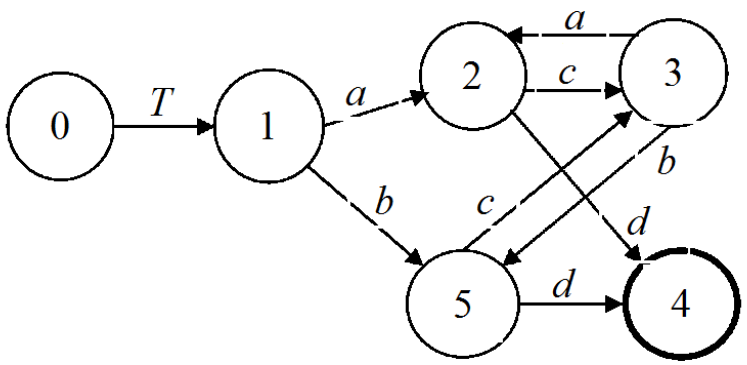


Рис. 1.17. Модель алгоритма умножения

$$L(M) = (0T1)\{[(1a2)(((2c3)(3a2))^*(2d4) + (2c3)((3b5)(5c3))^*(3b5)(5d4)] +$$
$$+ (1b5)[((5c3)(3b5))^*(5d4) + (5c3)((3b5)(5c3))^*((3a2)(2c3))^*(3a2)(2d4)]\} =$$
$$= T[a((ca)^*d + c(bc)^*bd) + b((cb)^*d + c(bc)^*(ac)^*ad)] =$$
$$= T[a((ca)^* + c(bc)^*b) + b((cb)^* + c(bc)^*(ac)^*a)]d =$$
$$= T[a((ca)^* + c(bc)^*b) + b((cb)^* + c(bc)^*(ac)^*a)]d =$$
$$= T[a((ca)^* + tb) + b((cb)^* + t(ac)^*a)]d,$$

где t переводит КА в состояние, которое запускает конечный автомат, распознающий строку $c(bc)^*$.

2. МАШИНА ТЬЮРИНГА

Универсальный КА, используемый для решения любой алгоритмически разрешимой задачи, в теории алгоритмов и вычислений называется **машиной Тьюринга**.

Память машины допускает как чтение, так и запись на ленту в одну и ту же ячейку.

Множество входных и выходных символов не различается, т.е. единый алфавит на входе (чтение) и на выходе (запись) $\sum = W$.

Функция перехода δ: $Q^* \sum \rightarrow Q$.

Функция выхода λ: $Q^* \sum \rightarrow K^* \sum$, где K — команды управления памятью (применительно к ленте: L — сдвиг влево, R — сдвиг вправо, N — лента неподвижна).

Принципы работы машины Тьюринга

Начальное слово — в алфавите ∑ размещается на ленте.

При чтении другого символа с панели выполняется определенная команда K, машина переходит в следующее состояние, и символ записывается в то же место на панели.

Машина обращается к входной последовательности, если она достигает конечного состояния, и останавливается.

Машина Тьюринга реализует алгоритм, если он всегда применяется к исходной информации (слову), которая отражает условия задачи и преобразует входное слово в полученную информацию (выходное слово).

Конфигурация машины при выполнении алгоритма: $K(qi, s)$, где qi — состояние машины; s — текущая строка памяти. Шаг алгоритма — это переход от одной конфигурации к другой после прочтения символа.

Машина Тьюринга чрезвычайно важна в теории алгоритмов как строгое и точное формальное определение алгоритма в виде машинной схемы.

Основная гипотеза теории алгоритмов: машина Тьюринга решает любую проблему алгоритмического решения.

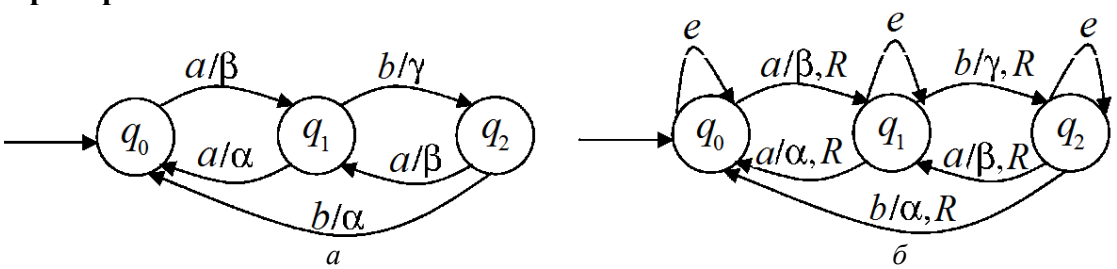
Вопрос об алгоритмическом разрешении (существовании алгоритма решения проблемы) заставляет доказать, что существует машина T, которая решает задачу за конечное число шагов. Если количество шагов бесконечно, проблема трудноразрешима или она не имеет алгоритмического решения.

Однако в этом случае проблема доказательства решения самой задачи является неразрешимой, поскольку нет алгоритма создания такой машины (аналогия с формальным логическим выводом).

Таким образом, проблема сводится к некоторой и более простой задаче, для которой можно показать, что соответствующая машина Тьюринга не может быть построена даже в этом случае, а более общая проблема не имеет алгоритмического решения.

Машина Тьюринга позволяет выполнять алгоритм распознавания общих и нерегулярных языков. Вычисление КА в машине Тьюринга представлено на рисунке ниже.

Пример 2.1



а *б*

Рисунок. Вычисление КА в машине Тьюринга: *а* — конечный автомат; *б* — машина Тьюринга

Такая конструкция не имеет практического применения: любой компьютер является универсальной машиной Тьюринга с физически ограниченной памятью и может интерпретировать любую машину, если мы предположим, что, как и в машине Тьюринга, память бесконечна.

Следующую концепцию русского математика В.Н. Крупского развития гипотезы Тьюринга связывают с программированием и алгоритмическими языками.

Тезис Тьюринга (неформальный): любую вычислительную функцию (для любой программы на языке программирования) типа $\sum^* \rightarrow \sum^*$ можно вычислить на соответствующей машине Тьюринга.

Неформальность тезиса Тьюринга в том, что его нельзя полностью доказать математическими средствами. В то же время многие попытки устранить его, т.е. предложить язык программирования с большой вычислительной мощностью, не увенчались успехом. Причина невозможности полного математического доказательства — неопределенность рассматриваемых языковых семейств. Если их заменить достаточно информативным описанием языков программирования, то соответствующий частный случай тезиса становится общепринятым и «проверенным» математическим понятием.

Пример 2.2

Семейство состоит из языков C и PASCAL, синтаксис и операционная семантика которых подробно описаны. В языках ASSEMBLER есть компиляторы, поэтому для программирования соответствующего частного случая тезиса Тьюринга необходимо создать компилятор, который преобразует код компиляции в программу для машины Тьюринга. Последнее — очень долгая, но точная задача программирования. Для полного теста по математике нужно будет протестировать программу на этом языке.

3. МАШИНА ПОСТА

Конструктивно машина Поста близка к машине Тьюринга. В основном она отличается двоичным алфавитом входных и выходных данных (в машине Тьюринга алфавит не определяется, а выбирается).

В машине Тьюринга конструируется управляющий автомат, а в машине Поста — программа для решения этой проблемы. Очевидно, что преимуществами машины Поста являются универсальность и простота.

Система команд машины Поста:

Shr — движение вправо к соседней позиции;

Shl — движение влево к соседней позиции;

Wr1 — читать текущую ячейку,
если в ячейке 0, то записать 1,
если в ячейке 1, то останов неприменимости;

Wr0 — читать текущую ячейку,
если в ячейке 1, то записать 0,
если в ячейке 0, то останов неприменимости;

Jmp j0, j1 — читать текущую ячейку,
если в ячейке 0, то перейти к команде j0,
если в ячейке 1, то перейти к команде j1;

Stop — остановка.

Следовательно, алгоритмическое описание и решение задачи — это машинная программа. Набор команд (программа) применим к этой проблеме и представляет собой алгоритм ее решения, если выполнение программы завершается командой STOP. Если происходит неработающая остановка, значит, программа неприменима к данной задаче и не имеет алгоритмического решения.

Машина Поста не имеет практического значения. В теории компьютеров она представляет собой универсальную минимальную систему, полные инструкции и элементарный дизайн.

4. РЕКУРСИВНЫЕ ФУНКЦИИ

Если задать алфавит A из букв r, то любое слово R в этом алфавите можно рассматривать как запись некоторого натурального числа в r-й системе счисления.

Таким образом, исходные данные — слова R в алфавите A — можно интерпретировать как натуральные числа. Вместе с тем можно интерпретировать результат передачи алгоритма как числовое значение функции, вычисленное на основе заданного значения аргумента.

Рекурсивные (возвратные) функции позволяют определять значение от неизвестного до неизвестного (от простого к сложному).

Есть измеримая функция, если есть алгоритм — эффективная процедура последовательного расчета от простого к сложному.

Выделяются базис элементарных функций, интуитивно вычислимых, и операторы ресурсов для получения более сложных функций.

1. При вычислении значения натурального числа $N_n = N_{n-1} + 1, N_n = 0$ можно записать ряд выражений для $N_{n-1}, N_{n-2}, \dots, N_4, N_3, N_2, N_1, N_0 = 0$ и затем последовательно вычислить N_n.

2. Для суммы n чисел $\{a_1, a_2, \dots, a_n\}$: $S_n = S_{n-1} + a_n, \dots, S_1 = S_0 + a_1, S_0 = 0$.

К элементарным вычислимым функциям относятся:

1) $S(x) = x + 1$ — следование, т.е. вычисление следующего натурального числа;

2) $0(x) = 0$ — константа нуля;

3) $I_m(x_1 x_2 \dots x_n) = x_m$ — выбор аргумента x_m из n аргументов.

Операторы получения более сложных функций:

1) $H(x, y) = f(x)$ — введение вспомогательной фиктивной переменной, при вычислении стирается y и вычисляется $f(x)$;

2) $\Phi(x) = g(f(x))$ — суперпозиция, для вычисления $\Phi(x)$ используются алгоритмы вычисления более простых функций $y = f(x)$ и $g(y)$;

3) итерация (рекурсия) $\Phi(0) = C$ — базис, $\Phi(x + 1) = f(x, \Phi(x))$ — рекуррентное (возвратное) соотношение, шаг индукции.

Утверждение. Всякая рекурсивная функция эффективно вычислима. Существует метод, который по рекурсивному описанию строит алгоритм вычисления этой функции.

Эффективное вычисление рекурсивных функций

Применение рекурсии и формирование трассы вычислений $\Phi(N), \Phi(N-1), \dots, \Phi(0)$.

Возврат-вычисление функций по трассе:

– если трасса известна, то строится циклическая программа;

– проводится прямое вычисление по формуле $\Phi(N)$, если она известна.

Примером тому служит сумма членов арифметической прогрессии или чисел Фибоначчи.

Примеры построения сложных рекурсивных функций на основе элементарных и рекурсии:

1) **сумма** — вычисляется $\Phi(x, C) = x + C$ для заданного x:

$\Phi(0, C) = C$ — базис,

$\Phi(x + 1, C) = \Phi(x, C) + 1$ — итерация;

2) **произведение** — вычисляется $\Phi(x, C) = x*C$ для заданного множителя x:

$\Phi(0, C) = 0$ — базис,

$\Phi(x + 1, C) = \Phi(x, C) + C$ — итерация;

3) **факториал** — вычисляется $\Phi(x) = x!$:

$\Phi(0) = 1$,

$\Phi(x + 1, C) = (x + 1)*\Phi(x, C)*C$ — итерация;

4) вычисляется $\Phi(x, C) = x!\, C^{x+1}$:

$\Phi(0, C) = C$,

$\Phi(x + 1, C) = (x + 1)*\Phi(x, C)*C$ — итерация;

5) **вычисляется** $\Phi(x, C) = C^x$:

$\Phi(0, C) = 1$,

$\Phi(x + 1, C) = \Phi(x, C)*C$ — итерация;

6) μ-оператор для определения функции, число повторений которой неизвестно, но задается предикатом:

$[\log_r(y)] = \mu x(r^{(x+1)} > y)$, y — искомый аргумент;

$[y^{1/r}] = \mu x((x + 1)^r > y)$.

Теория алгоритмов показывает, что машины Тьюринга могут эффективно вычислять все измеримые ретроспективные функции.

На практике повторяющееся описание рабочих задач интуитивно понятно. В то же время уровень сложности решаемых задач практически доступен, используется в алгоритмических языках высокого уровня и сосредоточен в эффективных вычислительных процессах.

Язык рекурсивных вычислений РЕФАЛ (рекурсивных функций алгоритма) предложен В.Ф. Турчиным.

Рекурсии могут быть использованы в алгоритмических языках Алгол, Пролог, Форт.

Примеры рекурсивных вычислений

1. **Факториал**

Ф(0) = 1;

Ф(x + 1) = (x + 1)*Ф(x).

Формируется и сохраняется трасса

Ф(10) = 10*Ф(9) = 10*(9*Ф(8)) = 10*(9*(8*…1*Ф(0)).

Затем последовательно вычисляется факториал.

Простая циклическая программа в Си:

for(int i = 1; S = 1; i < 11; i++)

S = S*i;

2. **Ряд чисел Фибоначчи** $F_1, F_2, \ldots, F_n$.

Трасса строится по рекуррентной формуле

$F_n = F_n - 1 + F_n - 2$, $n > 2$.

Общая формула Бине:

$F_n = \dfrac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}$, где $\varphi = \dfrac{1+\sqrt{5}}{2}$.

3. **Арифметическая и геометрическая прогрессии**

Трассы членов ряда $A_n, A_{n-1}, \ldots, A_1, A_0$ и рекуррентные формулы:

– возрастающей арифметической прогрессии: $A_n = A_{n-1} + d$;

– возрастающей геометрической прогрессии: $A_n = A_{n-1}*d$.

Формулы для общего члена ряда:

– арифметической прогрессии: $A_n = A_0 + d(n - 1)$;

– геометрической прогрессии: $A_n = A_0 d^{n-1}$.

4. **Степенные ряды и полиномы**

Степенные ряды и полиномы широко используются в виде производящих функций, в приближениях функций ряда Тейлора, в позиционных системах счисления.

Вычисления массивов во многих случаях могут быть представлены итерационными формулами и сведены к итерационным алгоритмам.

Пример 4.1

Десятичное число $a_{m-1}\ a_{m-2}...a_1a_0$, представленное в полиномиальной форме, обозначает количество N:

$$N = \sum_{i=0}^{m-1} a_i d^i = a_{m-1}10^{m-1} + a_{m-2}10^{m-2} + \cdots + a_1 10 + a_0 =$$
$$= \big(\ldots\big((0 + a_{m-1})10 + a_{m-2}\big)10 + \cdots + a_1\big)10 + a_0.$$

С использованием обобщенной схемы Горнера полином преобразуется в скобочную форму, которую можно определить рекуррентной формулой, применяемой для преобразования числа в любую другую позиционную однородную систему счисления:

$S_{i-1} = S_i*10 + a_{i-1}$, $i = m, m-1, \ldots, 0$, $S_m = 0$.

5. АССОЦИАТИВНЫЕ ИСЧИСЛЕНИЯ

Пусть задан алфавит $\sum$. Слово — последовательность букв алфавита. Допустимые преобразования слов задаются подстановками. Ориентированные подстановки $P \to Q$ — замена любой части слова P на Q.

Неориентированные подстановки $P \leftrightarrow Q$ — замена как P на Q, так и наоборот.

Ассоциативное исчисление — совокупность слов в алфавите.

Схема алгоритма — конечная система подстановок.

Алгоритм в ассоциативном исчислении — формальное решение задачи распознавания эквивалентности слов $A \sim Q$ с использованием подстановок.

Прикладные задачи, близкие к ассоциативным исчислениям:

– логический вывод;

– компиляция формальных языков;

– задачи на графах, например поиск пути в лабиринте.

Примеры ассоциативных исчислений

1. **Нормальные алгорифмы А.А. Маркова**

Алфавит $\sum = \{a, b, c\}$.

Схема алгоритма — упорядоченная система ориентированных подстановок:

$b \to acc$

$ca \to accc\ aa \to \emptyset$

$bb \to \emptyset$

$cccc \to \emptyset$

– используется первая по порядку подстановка;

– рассматривается самое левое вхождение:

$\underline{b}b \to acc\underline{b} \to ac\underline{ca}cc \to ac\underline{a}ccccc \to a\underline{a}ccccccccc \to ccccccccc \to cccc \to \emptyset$

2. **Исчисления Г.С. Цейтина**

Алфавит $\sum = \{a, b, c, d, e\}$.

Схема алгоритма:

$ac \to ca$

$ad \to da$

$bc \to cb$

$bd \to db$

$abac \to abaceeca \to ae$

$edb \to be$

$abcde \to acbde \to cabde \to cadbe \to$

3. **Проблема выводимости в аксиоматической теории** — аналогия с ассоциативным исчислением — представляет процесс формальных преобразований с использованием подстановок в аксиомах. Для любых слов R и S требуется доказать, что существует цепочка подстановок (алгоритм), преобразующая $R \rightarrow S$.

Теорема Черча [6] доказывает, что вывод недоступен в форме конечного алгоритма.

4. **Анализ предложений алгоритмического языка**

Синтаксис определяется подстановочным набором, применимым к последовательности символов языкового алфавита и расширенным промежуточными терминальными символами. Если символьная строка считывается и идентифицируется с помощью подстановки, строка принадлежит языку. Схема называется **компилятором** для использования центров, которые выполнены в виде читающей и пишущей машины, представляющей символьный командный поток.

6. КЛАССЫ СЛОЖНОСТИ

В рамках классической теории алгоритмические задачи различаются по классам сложности (*P*-сложный, *NP*-жесткий, экспоненциально сложный и т. д.).

Курсы сложности — это множество вычислительных задач, которые примерно одинаковы с точки зрения вычислительной сложности. Более конкретно, классы сложности — это наборы категорий (функции, принимающие слово в качестве входных данных и возвращают ответ 0 или 1), которые используют примерно одинаковое количество ресурсов для вычислений. Каждый класс сложности (в строгом смысле) определяется как набор категорий с определенными свойствами.

Класс сложности X — это набор классов $P(x)$, вычисленных на машинах Тьюринга, которые используют ресурсы $O(f(n))$ для вычисления, где n — длина слова x.

Время вычисления (количество циклов машины Тьюринга) или рабочая область (количество ячеек, используемых на ленте во время работы) обычно берутся в качестве ресурсов.

Класс *P* — это задачи, которые могут быть решены с помощью множителя времени в зависимости от количества исходных данных с помощью детерминированного компьютера (например машины Тьюринга).

Класс *NP* — это задачи, которые могут быть решены за полиномиальное время с помощью недетерминированного компьютера, т.е. машины, следующее состояние которой не всегда четко определяется предыдущими. Класс *NP* включает задачи, решением которых с помощью дополнительной информации о длине полинома, данной ранее, мы можем управлять за полиномиальное время. Класс *P* входит в категорию *NP*. Классическими примерами задач *NP* являются: задача коммивояжера, нахождение гамильтонова круга, раскраска верхней части графика.

Функционирование такой машины можно представить как ветвящийся процесс в любой неоднозначности: проблема считается решенной, если ответила хотя бы одна ветвь процесса.

Поскольку класс *P* содержится в классе *NP*, принадлежность к проблеме в классе *NP* часто отражает наше текущее понимание того, как решить эту проблему, и не является окончательным. В общем случае нет оснований полагать, что решение *P* не может быть найдено для конкретной *NP*-задачи. Вопрос о возможной эквивалентности классов *P* и *NP* (т.е. о возможности найти решение *P* любой *NP*-задачи) многие считают одним из основных вопросов современной теории сложности алгоритмов. На этот вопрос нет ответа. Сама постановка вопроса об эквивалентности классов *P* и *NP* возможна благодаря введению понятия *NP*-полных задач. Полные проблемы *NP* представляют собой подмножество проблем *NP* и отличаются тем, что все проблемы *NP* могут быть сведены к ним тем или иным способом. Из этого следует, что если решение *P* найдено для *NP*-полной задачи, то решение *P* будет найдено для всех задач класса *NP*. Примером *NP*-полной проблемы является проблема конъюнктивной формы.

Наиболее часто встречающиеся классы сложности в зависимости от числа входных данных n (в порядке нарастания сложности, т.е. увеличения времени работы алгоритма, при стремлении n к бесконечности): O(1) — количество шагов алгоритма не зависит от количества входных данных. Обычно это алгоритмы, использующие определенную часть данных входного потока и игнорирующие все остальные данные.

Ряд алгоритмов имеет порядок, включающий $\log_2 n$, и называется логарифмическим. Эта сложность возникает, когда алгоритм неоднократно подразделяет данные на подсписки длиной 1/2, 1/4, 1/8 и т.д. от оригинального размера списка. Логарифмические порядки возникают при работе с бинарными деревьями. Бинарный поиск имеет сложность среднего и наихудшего случаев $O(\log_2 n)$.

Сложность $O(n\log_2 n)$ имеют алгоритмы быстрой сортировки, сортировки слиянием и «кучной» сортировки, а также алгоритм Краскала — построение минимального связывающего дерева, n — число ребер графа.

Алгоритм со сложностью $O(n)$ — алгоритм линейной сложности, т.е. для каждого входного объекта выполняется только одно действие.

Алгоритмы, имеющие порядок $O(n^2)$, являются квадратичными. К ним относятся наиболее простые алгоритмы сортировки, алгоритм Дейкстры — нахождение кратчайших путей в графе [7]. Алгоритм Прима — построение минимального связывающего дерева. Квадратичные алгоритмы используются на практике только для относительно небольших значений n. Всякий раз, когда число вершин графа n удваивается, время выполнения такого алгоритма увеличивается на множитель четыре.

Алгоритм показывает кубическое время, если его порядок равен $O(n^3)$, и такие алгоритмы очень медленные. Всякий раз, когда n удваивается, время выполнения алгоритма увеличивается в восемь раз. Таким примером может служить алгоритм Флойда – Уоршелла — динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа.

Алгоритм со сложностью $O(2^n)$ имеет экспоненциальную сложность. Такие алгоритмы выполняются настолько медленно, что они используются только при малых значениях n. Этот тип сложности часто ассоциируется с проблемами, требующими неоднократного поиска дерева решений.

Алгоритмы со сложностью $O(n!)$ — факториальные алгоритмы, в основном используются в комбинаторике для определения числа сочетаний, перестановок.

В табл. 6.1 сравниваются значения n^2 и $n\log_2 n$. Более эффективным является алгоритм сортировки $O(n\log_2 n)$, чем обменная сортировка. Например, в случае со списком из 10 000 элементов количество сравнений для обменной сортировки ограничивается величиной 100 000 000, тогда как более эффективный алгоритм имеет количество сравнений, ограниченное величиной 132 000. Новая сортировка приблизительно в 750 раз более эффективна.

Таблица 6.1

n	n^2	$n \log_2 n$
5	25	11,6
10	100	33,2
100	10 000	664,3
1000	1 000 000	9965,7
10 000	100 000 000	132 877,1

Таблица 6.2

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65 536
32	5	160	1024	32 768	4 294 967 296
128	7	896	16 384	2 097 152	$3{,}4 \cdot 10^{38}$
1024	10	10 240	1 048 576	1 073 741 824	$1{,}8 \cdot 10^{308}$
65 536	16	1 048 576	4 294 967 296	$2{,}8 \cdot 10^{14}$	Избегайте!

В табл. 6.2 приводятся линейный, квадратичный, кубический, экспоненциальный и логарифмический порядки величины для выбранных значений n. Очевидно, что следует избегать использования кубических и экспоненциальных алгоритмов, если только значение n не мало.

Важность проведения резкой границы между полиномиальными и экспоненциальными алгоритмами вытекает из сопоставления числовых примеров роста допустимого размера задачи с увеличением быстродействия Б используемых ЭВМ (табл. 6.3, в которой указаны размеры задач, решаемых за одно и то же время T на ЭВМ с быстродействием $Б_1$ при различных зависимостях сложности Q от размера n). Эти примеры показывают, что, выбирая ЭВМ в K раз более быстродействующую, получаем увеличение размера решаемых задач при линейных алгоритмах в K раз, при квадратичных алгоритмах в $K^{1/2}$ раз и т.д.

Таблица 6.3

$Q(n)$	$Б_1$	$Б_2 = 100\ Б_1$	$Б_2 = 1000\ Б_1$
n	n_1	$100n_1$	$1000n_1$
n^2	n_2	$10n_2$	$31{,}6n_2$
n^3	n_3	$4{,}64n_3$	$10n_3$
2^n	n_4	$6{,}64 + n_4$	$9{,}97 + n_4$

Иначе обстоит дело с неэффективными алгоритмами. Так, в случае сложности 2^n для одного и того же процессорного времени размер задачи увеличивается только на $\lg K / \lg 2$ единиц. Следовательно, переходя от ЭВМ с $Б_1 = 1$ Гфлопс к супер-ЭВМ с $Б_3 = 1$ Тфлопс, можно увеличить размер решаемой задачи только на 10, что совершенно недостаточно для практических задач. Действительно, в таких задачах, как, например, синтез тестов для больших интегральных схем, число входных двоичных переменных может составлять более 100, и поэтому полный перебор всех возможных проверяющих кодов потребует выполнения более 2^{100} вариантов моделирования схемы.

Изучение сложности алгоритма позволяет нам иначе взглянуть на многие классические математические задачи и найти решения для таких задач (полиномиальное и матричное умножение, решение линейных уравнений и т.д.), которые требуют меньше ресурсов, чем традиционные.

7. ЛОГИЧЕСКИЙ СИНТЕЗ ВЫЧИСЛИТЕЛЬНЫХ СХЕМ

Логическая функция устанавливает соответствие между одним или несколькими высказываниями, которые называются **аргументами функции**, и высказыванием, которое называется **значением функции**. Различные комбинации значений аргументов называются **наборами**[2].

Синтез комбинационных устройств обычно начинается с табулирования значений истинности всех входных и выходных величин. Результатом рассматриваемого этапа является таблица истинности, связывающая все возможные комбинации значений аргументов и функций. Затем осуществляется переход от таблицы истинности к аналитическому выражению. Следующим этапом синтеза являются анализ и оптимизация (минимизация) логических функций, после чего строится функциональная схема синтезируемого узла. Построение схемы основано на прямом замещении элементарных произведений, сумм и отрицаний соответственно конъюнкторами, дизъюнкторами и инверторами. При построении дискретных устройств наибольшее распространение получили логические схемы, которые состоят из логических элементов и осуществляют логические операции. Для изображения логических схем используются условные графические обозначения элементов, описывающие только выполняемую элементами функцию и не зависящие от схемы устройства.

В настоящее время в мире существует несколько общепринятых стандартов условных обозначений. Наиболее распространенными являются американский стандарт MilSpec 806B и стандарт МЭК 117-15A, созданный Международной электротехнической комиссией (рис. 7.1). Часто в литературе используются также обозначения европейской системы DIN 4070. В отечественной литературе условные обозначения элементов в основном соответствуют ГОСТ 2.743–91 (табл. 7.1).

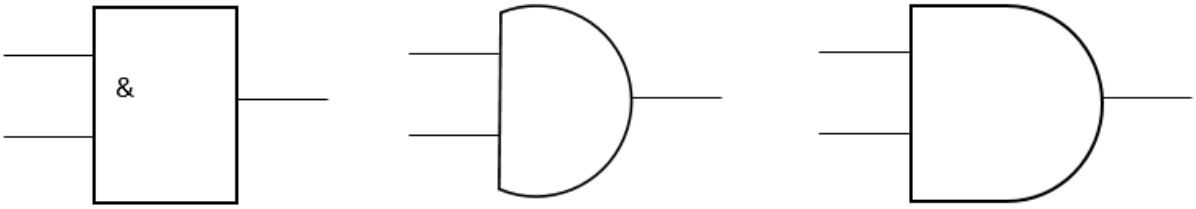


Рис. 7.1. Условные обозначения логического элемента И (конъюнктор):
а — по ГОСТ и стандарту МЭК; *б* — по стандарту DIN; *в* — по стандарту MilSpec

Равнозначность — это логическая функция от двух переменных, которая принимает единичное значение при одинаковых значениях переменных, т.е. равнозначных: $F = X \times Y + \bar{X} \times \bar{Y}$.

Отрицание равнозначности — это логическая функция от двух переменных, которая принимает единичное значение при разных по значимости переменных:

$$F = \bar{X} \times Y + X \times \bar{Y}.$$

Функция отрицания от логического умножения (штрих Шеффера) принимает значение 0, когда все аргументы равны 1, и 1 — во всех остальных случаях:

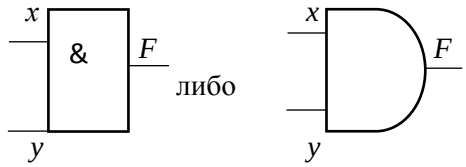
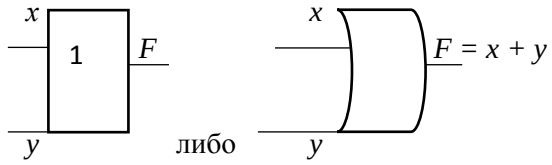
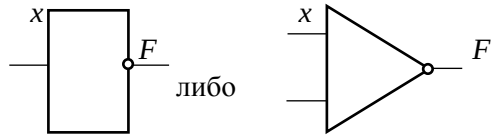
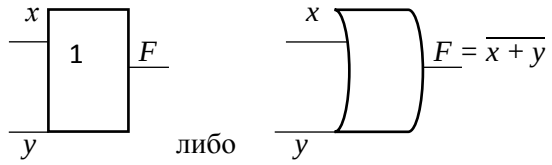
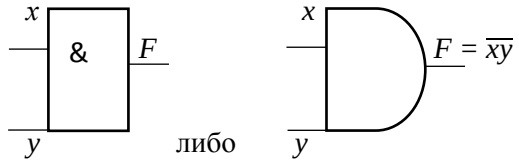
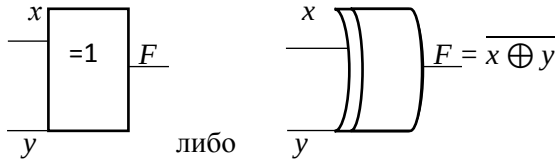
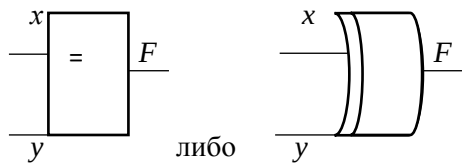
$$F = \overline{X \wedge Y} \text{ либо } F = \overline{X \,\&\, Y}, \text{ либо } F = \overline{X \times Y}.$$

Функция отрицания от логического сложения (стрелка Пирса) принимает значение 1, когда все аргументы равны 0, и значение 0 — во всех остальных случаях:

$$F = \overline{X \vee Y} \text{ либо } F = \overline{X + Y}.$$

[2] URL: <= КОНСУЛЬТАНТ => Электронный справочник по ИНФОРМАТИКЕ (Автор Панов В.А.) (narod.ru) (дата обращения: 12.11.2021).

Таблица 7.1

Условные обозначения	Описание
	Конъюнкция (конъюнктор): $F = xy$
	Дизъюнкция (дизъюнктор): $F = x + y$
	Инверсия (инвертор): $F = \bar{x}$
	Стрелка Пирса (ИЛИ–НЕ): $F = \overline{x + y}$
	Штрих Шеффера (И–НЕ): $F = \overline{xy}$
	Исключающее ИЛИ(XOR): $F = \overline{x \oplus y}$
	Неэквивалентность (неравнозначность): $F = \overline{xy} + \overline{xy}$

По формуле функции создается ее схема на логических элементах. Для упрощения создаваемой схемы вместо элементарных двухвходовых элементов по мере необходимости будем использовать трех- и четырехвходовые.

Например, произведение $X_1 \cdot X_2 \cdot X_3$ на вентилях реализовано следующим образом (рис. 7.2).

Если в произведении один из сомножителей участвует с инверсией, то для него следует ввести вентиль-инвертор.

Например, для произведения $\bar{X}_1 \cdot X_2 \cdot X_3$ схема имеет вид, показанный на рис. 7.3.

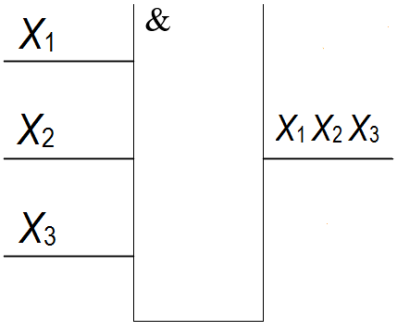


Рис. 7.2. Трехвходовые элементы

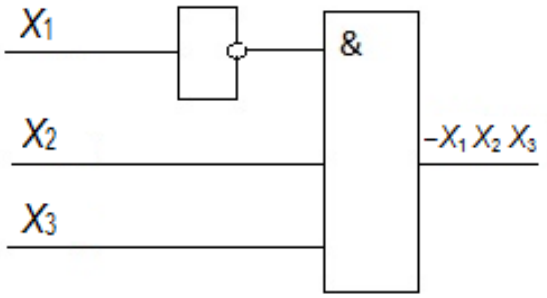


Рис. 7.3. Вентиль-инвертор

Для сложения произведений используется вентиль сложения.

Например, для суммы двух произведений $X_1 \cdot X_2 \cdot X_3$ и $\overline{X_1} \cdot X_2 \cdot X_3$ схема имеет вид, показанный на рис. 7.4.

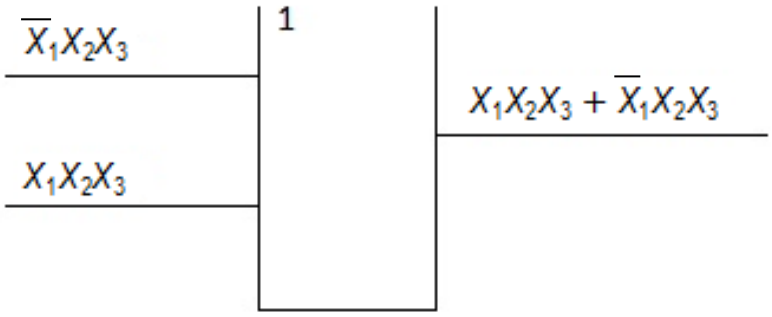


Рис. 7.4. Вентиль сложения

Под логическим элементом понимается комбинационная схема, реализующая некоторую элементарную булеву функцию. Любой логический элемент характеризуется наличием одного или нескольких входов, наличием выхода, а также функцией, которая отображает зависимость выходного сигнала от входных.

Логические схемы разделяются на два типа: последовательностные и комбинационные.

В последовательностных схемах выходные сигналы в любой момент времени зависят не только от комбинации входных сигналов в данный момент времени, но и от предыстории их изменения, т.е. от последовательности входных сигналов во времени.

Последовательностные схемы характеризуются наличием так называемых петель, по которым выход некоторого элемента соединяется со входом этого же самого элемента (через другие элементы схемы). С учетом этой задержки значение выходного сигнала по времени запаздывает на время задержки по сравнению с моментом изменения входных сигналов.

Как правило, последовательностные схемы характеризуются некоторым внутренним строением, от которого зависит значение выходного сигнала (сигналов). Внутреннее состояние такой схемы сохраняется на запоминающих элементах (триггерах), в связи с чем схемы этого типа называются схемами с памятью.

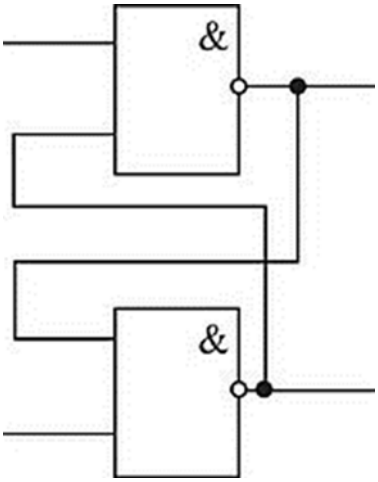


Рис. 7.5. Последовательностная схема

В общем случае последовательностная схема представляет собой некоторый цифровой автомат (рис. 7.5).

В комбинационных схемах (КС) значение выходного сигнала в любой момент времени зависит только от комбинации входных сигналов (в этот же момент времени с учетом задержки распространения сигнала по элементам схемы).

Функционирование комбинационной схемы (рис. 7.6) может быть описано булевой функцией, отражающей зависимость выходного сигнала схемы от выходных сигналов как аргументов этой функции. Для комбинационных схем с несколькими выходами такая зависимость отражается системой булевых функций.

Основные параметры комбинационных схем — стоимость и быстродействие. Быстродействие схемы оценивается задержкой распространения сигналов от входов схемы к ее выходу. Для абстрактных КС эту задержку принято выражать в виде $T = k\tau$, где τ — задержка на одном логическом элементе; k — максимальное количество логических элементов, через которые проходит сигнал от входов к выходу.

Задержка схемы сопоставляется с числом уровней этой схемы: все элементы схемы распределяются по уровням. Уровень элемента, на выходе которого формируется выходной сигнал схемы, совпадает с количеством уровней схемы и, следовательно, с ее задержкой.

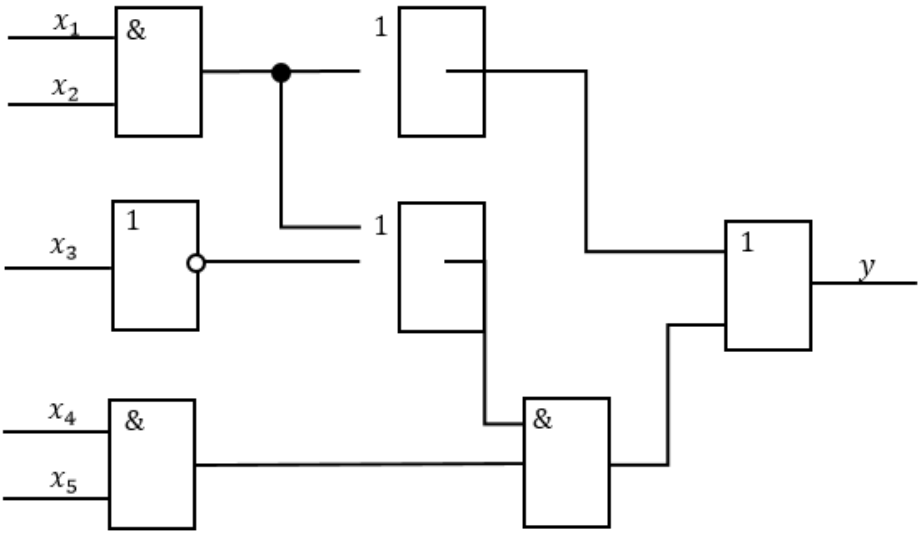


Рис. 7.6. Комбинационная схема

Примечание. В ряде случаев перед построением логического блока — схемы устройства по логической функции — последнюю, пользуясь соотношениями алгебры логики, следует преобразовать к более простому виду (минимизировать). Для логических схем ИЛИ, И и НЕ существуют типовые технические схемы, реализующие их на интегральных схемах. Для построения современных компьютеров обычно применяются системы интегральных элементов, у которых с целью большей унификации в качестве базовой логической схемы используется всего одна из схем:

– И–НЕ (штрих Шеффера);
– ИЛИ–НЕ (стрелка Пирса);
– И–ИЛИ–НЕ.

Первыми и самыми простыми комбинационными схемами были **контактные схемы**, состоящие из параллельно и последовательно соединенных электрических ключей, реализующие элементарные булевы функции и предназначенные для коммутации (замыкания или размыкания) электрических цепей. Управление такими ключами производится вручную человеком, электромагнитным реле или другими механизмами. Управляющее воздействие ключей имеет два состояния: 1 — воздействие есть, например, кнопка ключа нажата; 0 — воздействия нет, например, кнопка ключа отпущена. Если в исходном состоянии ключ разомкнут, то при нажатии кнопки он замыкается — это ключ с нормально разомкнутыми контактами (он обозначается x). Ключ с нормально замкнутыми контактами при нажатии кнопки размыкается, поэтому такие ключи обозначаются инверсией x.

Пример 7.1

Дана контактная схема (рис. 7.7), описываемая формулой

$$(\bar{X} \vee Y) \wedge X = X \wedge (\bar{X} \vee Y) = (X \wedge \bar{X}) \vee (X \wedge Y) = 0 \vee (X \wedge Y) = (X \wedge Y) \vee 0 = X \wedge Y.$$

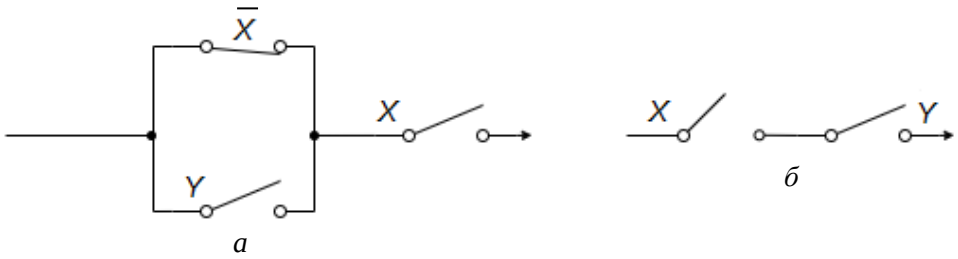


Рис. 7.7. Исходная контактная схема (*а*) и преобразованная схема (*б*)

Следовательно, логические функции, выражаемые формулами $(\bar{X} \vee Y) \wedge X$ и $X \wedge Y$, эквивалентны.

Пример 7.2

Произвести анализ и упростить контактную схему (рис. 7.8). Соответствующая схеме формула имеет вид

$$((X \vee Y) \wedge \bar{Y}) \vee (X \vee Y).$$

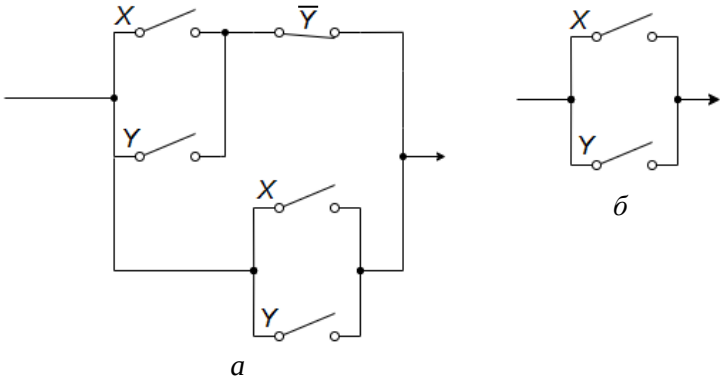


Рис. 7.8. Исходная контактная схема (*а*) и преобразованная схема (*б*)

Решение

Проведем преобразование формулы к следующему виду:

$$\left((X \vee Y) \wedge \bar{Y}\right) \vee (X \vee Y) = \left((X \vee Y) \wedge \bar{Y}\right) \vee \left((X \vee Y) \wedge 1\right) =$$

$$= (X \vee Y) \wedge (\bar{Y} \vee 1) = (X \vee Y) \wedge 1 = X \vee Y.$$

Контактные схемы на рис. 7.8 эквивалентны. Если исходная контактная схема состояла из пяти контактов, то упрощенная контактная схема, построенная на более простой формуле, состоит лишь из двух контактов.

Пример 7.3

Упростить контактную схему, изображенную на рис. 7.9. Данной контактной схеме соответствует логическая функция

$$\big((X \vee Y) \wedge (Z \vee \bar{X})\big) \vee \big(Y \wedge (\bar{Z} \vee \bar{Y})\big).$$

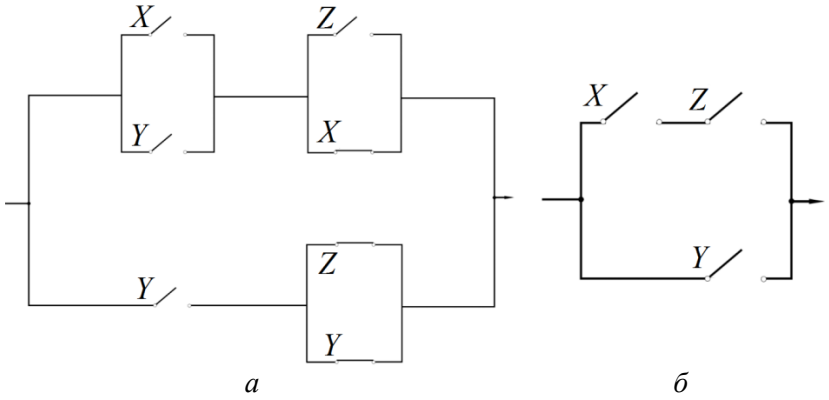


Рис. 7.9. Исходная контактная схема (*а*) и преобразованная схема (*б*)

Решение

Преобразуем исходное выражение:

$$\big((X \vee Y) \wedge (Z \vee \bar{X})\big) \vee \big(Y(\bar{Z} \vee \bar{Y})\big) = (X + Y)(Z + \bar{X}) + Y(\overline{Z + Y}) =$$
$$= XZ + X\bar{X} + YZ + Y\bar{X} + Y\bar{Z} + Y\bar{Y} = XZ + YZ + Y\bar{X} + Y\bar{Z} =$$
$$= XZ + Y\bar{X} + Y(Z + \bar{Z}) = XZ + Y\bar{X} + Y = XZ + Y(\bar{X} + 1) = XZ + Y = (X \wedge Z) \vee Y.$$

Примечание. В ходе преобразований использовались следующие обозначения: дизъюнкция — знак сложения, конъюнкция — знак умножения.

В исходной контактной схеме (рис. 7.9, *а*) было семь контактов, а в упрощенной (рис. 7.9, *б*) их стало только три.

Пример 7.4

Пусть функция задана таблицей истинности (табл. 7.2).

Таблица 7.2

A	*B*	*C*	*F*
0*	0*	0*	0*
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

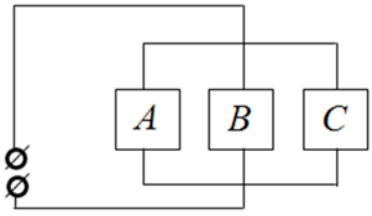


Рис. 7.10. Схема примера 7.4

Построить логическую схему для случая, когда $F = 0$. Это состояние в данной таблице помечено звездочкой, $F = (A + B + C)$. Электрическая схема для указанного случая будет иметь следующий вид (рис. 7.10).

Пример 7.5

Пусть дана электрическая цепь. Построить более простую электрическую цепь, реализующую то же самое высказывание (рис. 7.11).

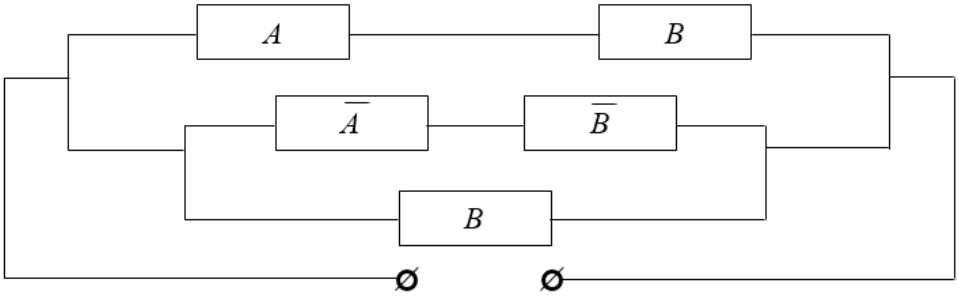


Рис. 7.11. Схема примера 7.5

Решение

Электрическая цепь реализует высказывание

$$F = AB + (\overline{AB} + B).$$

Упростим полученную логическую функцию:

$$F = (A + 1)B + \overline{AB} = B + \overline{AB} = B + \bar{A}.$$

Таким образом, упрощенная электрическая цепь, реализующая исходное высказывание, имеет следующий вид (рис. 7.12).

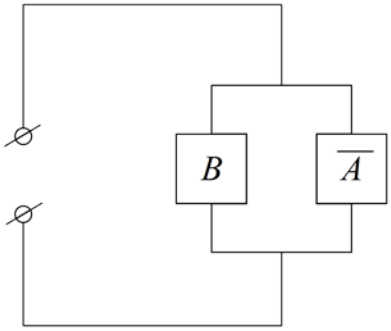


Рис. 7.12. Упрощенная электрическая схема

Пример 7.6

Для представленной электрической цепи (рис. 7.13) записать логическую функцию.

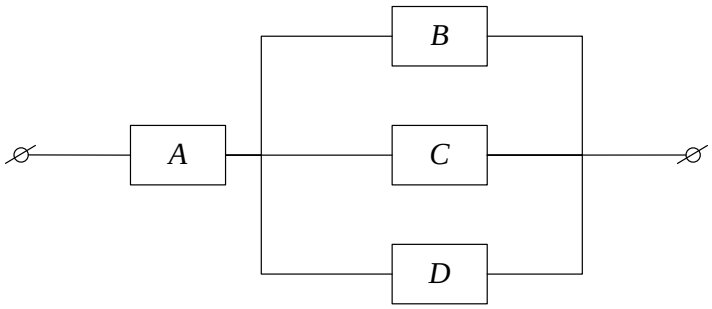


Рис. 7.13. Схема примера 7.6

Решение

$F = A(B + C + D).$

Пример 7.7

Построить одну из наиболее простых электрических цепей, реализующих высказывание $(\bar{A}+\bar{B})((\bar{C}+\bar{B})\to\bar{B})\to\bar{B}$.

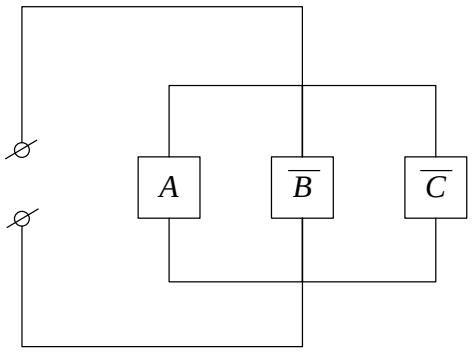


Рис. 7.14. Схема примера 7.7

Решение

Преобразуем данное высказывание:

$$(\bar{A}+\bar{B})\big((\bar{C}+\bar{B})\to\bar{B}\big)\to\bar{B}=(\bar{A}+\bar{B})\big(\overline{(\bar{C}+\bar{B})}+\bar{B}\big)\to\bar{B}=(\bar{A}+\bar{B})\big(\bar{C}\bar{B}+\bar{B}\big)\to\bar{B}=$$

$$=(\bar{A}+\bar{B})(CB+\bar{B})\to\bar{B}=(\bar{A}+\bar{B})(C+\bar{B})\to\bar{B}=(\bar{A}C+\bar{B}C+\overline{AB}+\bar{B})\to\bar{B}=$$

$$=(\bar{A}C+\bar{B}C+\bar{B})\to\bar{B}=(\bar{A}C+\bar{B})\to\bar{B}=\big(\overline{\bar{A}C+\bar{B}}\big)+\bar{B}=\overline{\bar{A}C\bar{B}}+\bar{B}=\big(\bar{\bar{A}}+\bar{C}\big)B+\bar{B}=\bar{B}+A+\bar{C}.$$

Таким образом, электрическая цепь будет иметь следующий вид (рис. 7.14).

Пример 7.8

Для данной схемы (рис. 7.15) записать логическое выражение.

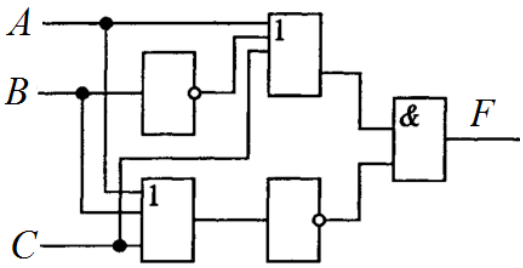


Рис. 7.15. Схема примера 7.8

Решение

$F=(A\vee\bar{B}\vee C)\wedge(\overline{B\vee A\vee C})$.

Пример 7.9

Для данной схемы (рис. 7.16) записать логическое выражение.

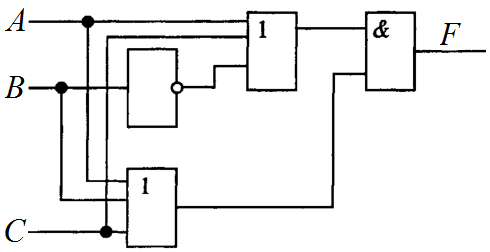


Рис. 7.16. Схема примера 7.9

Решение

$F=(A\vee B\vee C)\wedge(A\vee\bar{B}\vee C)$.

Библиографический список

1. Кнут Д.Э. Искусство программирования. Т. 1: Основные алгоритмы / Д.Э. Кнут ; пер. с англ. — 3-е изд. — Москва : Вильямс, 2002. — 720 с. — ISBN 5-8459-0080-8.

2. Поляков В.И. Основы теории алгоритмов [Электронный ресурс] : учебное пособие / В.И. Поляков, В.И. Скорубский. — Санкт-Петербург : Университет ИТМО, 2012. — 50 с. — URL: https://www.iprbookshop.ru/67504.html (дата обращения: 09.11.2021). — Режим доступа: для авторизир. пользователей.

3. Хопкрофт Дж. Введение в теорию автоматов, языков и вычислений / Дж. Хопкрофт, Р. Мотвани, Дж. Ульман ; пер. с англ. О.И. Васылык, М. Саит-Аметова, А.Б. Ставровского. — 2-е изд., испр. — Москва : Вильямс, 2008. — 527 с. — ISBN 978-5-8459-1347-0.

4. Миллер Р. Теория переключательных схем. Т. 2: Последовательные схемы и машины / Р. Миллер ; пер. с англ. О.П. Кузнецова и Ю.Л. Томфельда. — Москва : Наука, 1971. — 304 с.

5. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации автоматики / А.А. Шалыто. — Санкт-Петербург : Наука, 2000. — 780 с.

6. Трахтенброт В.А. Алгоритмы и вычислительные автоматы / В.А. Трахтенброт. — Москва : Советское радио, 1974. — 199 с.

7. Кормен Т. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — 4-е изд. — Москва : Вильямс, 2011. — 1296 с.

ПРИЛОЖЕНИЕ

Задание 1

Задание для компьютерного практикума выполняется обучающимися самостоятельно в соответствии с вариантом задания.

Выполните преобразование регулярных выражений в автоматы и обратно:

1) $(10+01)^*(11+00)(10)^*1$;
2) $(0^*((1+0)1))^*\,1(11^*01)^*11^*$;
3) $(111^*0)^*10(1(0+1))^*$;
4) $(0+10)^*11(01(0+1))^*$;
5) $(a+\ bb)^*\,b(ab^*b+(ab^*ba(\,a+b))^*)$;
6) $(bbb^*a)^*ba(b(a+b))^*$;
7) $(a^*((b+a)b))^*\,b(bb^*ab)^*bb^*$;
8) $((a+ba)^*bb(ab(a+b))^*$;
9) $a^*(b(a+b)a(b+ab)^*$;
10) $(1(\,0+1)^*\,0(0^*\,10(\,0+1)))^*$;
11) $(01+10)^*\,(10+01)0^*$;
12) $((ab\ +b)^*a(a+b)ba\,(a\ +b))^*$;
13) $(b(\,a+\ b)^*\,a(a^*\,ba(\,a+b)))^*$;
14) $(((0+\ 1)^*\,10)^*10(10)^*11)^*$;
15) $(ab+ba)^*\,(ba+ab)a^*$;
16) $((01+1)^*0(0+1)10(0\ +1))^*$;
17) $(a+ba)^*b(aa+ba)(a+ba)^*$;
18) $(0+11)^*\,1(01^*1+(01^*10(\,0+1))^*)$;
19) $(((a+b)^*\,ba)^*ba(ba)^*bb)^*$;
20) $((bbba+abbb)^*(bb+aba)(ba)^*bb.$

Задание 2

Задание для компьютерного практикума выполняется обучающимися самостоятельно в соответствии с вариантом задания.

1. Постройте граф автомата, регулярное выражение и систему уравнений, интерпретирующих КА, заданный таблицей переходов.
2. Восстановите исходное регулярное выражение.
3. Выполните преобразование в блок-схему и систему уравнений перехода.

Блок 1. Варианты 1–12

	Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j
	A	a	B		A	0	A		A	a	B		A	0	B
		b	A			1	B			b	–			1	–
	B	a	C		B	0	C		B	a	A		B	0	A
1		b	A	**2**		1	A	**3**		b	C	**4**		1	C
	C	a	D		C	0	D		C	a	D		C	0	D
		b	–			1	–			b	B			1	B
	D	a	B		D	0	B		D	a	D		D	0	A
		b	C			1	D			b	A			1	D
	Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j
	A	a	D		A	0	A		A	a	A		A	0	B
		b	B			1	B			b	B			1	C
	B	A	C		B	0	C		B	a	C		B	0	C
5		b	A	**6**		1	A	**7**		b	D	**8**		1	B
	C	A	–		C	0	B		C	A	B		C	0	–
		b	D			1	D			B	D			1	D
	D	A	B		D	0	C		D	A	C		D	0	B
		b	D			1	–			B	–			1	–
	Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j
	A	a	B		A	a	B		A	a	B		A	a	B
		b	C			b	–			b	C			b	D
	B	a	C		B	a	A		B	a	A		B	a	C
9		b	A	**10**		b	C	**11**		b	C	**12**		b	A
	C	a	–		C	a	D		C	a	D		C	a	D
		b	D			b	B			b	B			b	–
	D	a	B		D	a	D		D	a	D		D	a	B
		b	–			b	A			b	A			b	C

Блок 2. Варианты 13–24

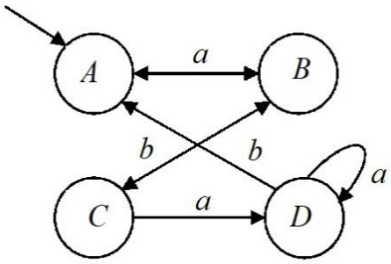
	Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j
	A	0	B		A	a	B		A	0	B		A	a	B
		1	–			b	–			1	D			b	D
	B	0	C		B	a	C		B	0	B		B	a	B
13		1	A	**14**		b	A	**15**		1	C	**16**		b	C
	C	0	–		C	a	–		C	0	A		C	a	A
		1	D			b	D			1	C			b	C
	D	0	B		D	a	D		D	0	A		D	a	–
		1	D			b	B			1	–			b	B
	Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j		Q_i	Σ	Q_j
	A	0	B		A	a	B		A	0	A		A	a	A
		1	A			b	A			1	B			b	B
	B	0	C		B	a	D		B	0	C		B	A	C
17		1	D	**18**		b	C	**19**		1	B	**20**		B	B
	C	0	–		C	a	–		C	0	A		C	a	D
		1	D			b	D			1	D			b	A
	D	0	B		D	a	B		D	0	C		D	a	–
		1	–			b	–			1	–			b	C
21	Q_i	Σ	Q_j	**22**	Q_i	Σ	Q_j	**23**	Q_i	Σ	Q_j	**24**	Q_i	Σ	Q_j
	A	0	B		A	a	B		A	a	B		A	0	B
		1	C			b	D			b	A			1	D
	B	0	D		B	a	C		B	a	B		B	0	C
		1	B			b	A			b	D			1	B
	C	0	E		C	a	E		C	a	–		C	0	E
		1	–			b	–			b	D			1	–
	D	0	A		D	a	A		D	a	C		D	0	A
		1	–			b	–			b	E			1	–
	E	0	–		E	a	–		E	a	C		E	0	–
		1	D			b	D			b	–			1	D

Пример выполнения

1. Задание.

Q_i	Σ	Q_j
A	*a*	*B*
	b	—
B	*a*	*A*
	b	*C*
C	*a*	*D*
	b	*B*
D	*a*	*D*
	b	*A*

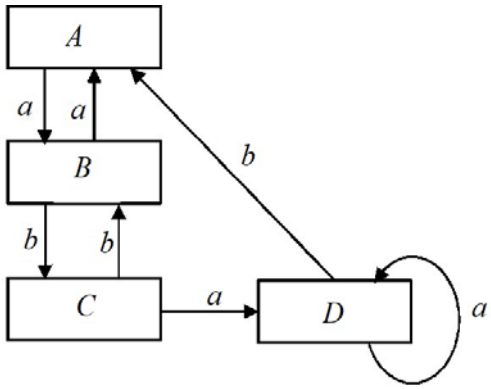
2. Граф КА.



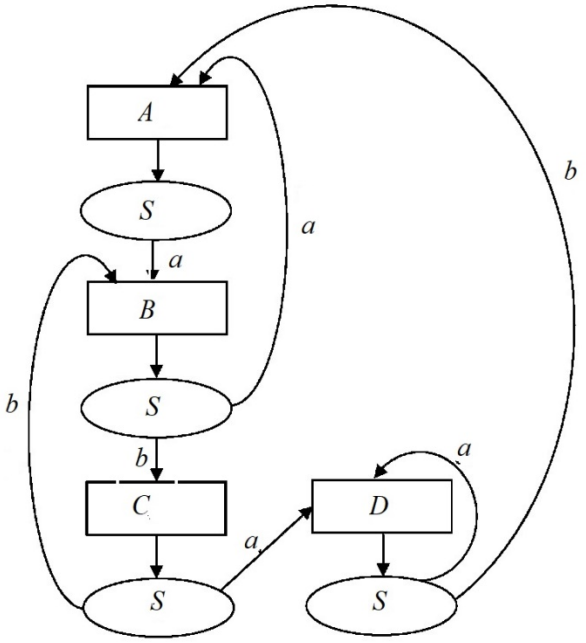
3. Регулярное выражение.

$$S = \Big(\big((AaB)(BaA)\big)^*(AaB)\big((BbC)(CbB)\big)^*(BbC)(CaD)(DaD)^*(DbA)\Big) =$$
$$= ((aa)^*a(bb)^*baa^*b) = ((aa)^*a(bb)^*baa^*b)^*ABABCBCDD.$$

4. Блок-схема.



Ребра графа представляем переключателем *Switch*.



Переход безусловный, если в условии присутствуют все символы входного алфавита.

Полагаем для общности, что в каждом состоянии выполняется некоторая одноименная команда из алфавита $\{A, B, C, D\}$, расширяем входной алфавит пустым символом $\{e\}$ и множество состояний — символом $\{F\}$, обозначающим финальное состояние (успешное завершение алгоритма и завершение идентификации слова, принадлежащего входному языку $L(M)$).

Если условие перехода по символу не определено (не полностью определенный КА), то состояние добавляется состоянием E, контролирующим ошибку.

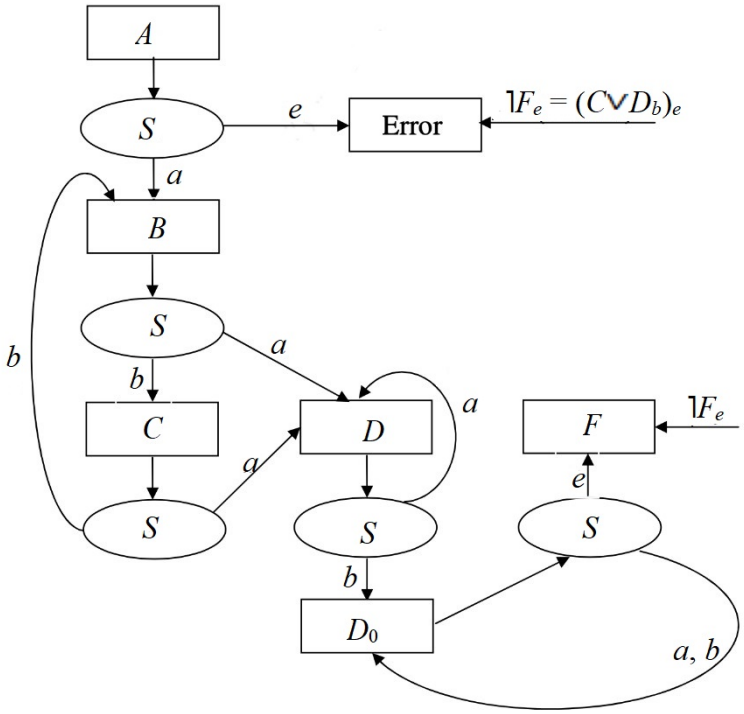
В конце непустой цепочки символов всегда (be), в данном случае это — условия C_e или D_{be}, а все другие условия ошибочные.

Правильное завершение распознавания и, соответственно, контроль логики выполнения алгоритма выполняется уравнением

$$F_e = C_e \vee D_{be} = (C \vee D_b)_e.$$

При ошибке в цепочке $E_e = \urcorner F_e$.

Для вычисления F_e можно добавить состояния F_e и операторную вершину в блок-схеме, что подразумевает недетерминированность при бесконечно больших цепочках. Команда F_e завершает алгоритм при пустой цепочке.

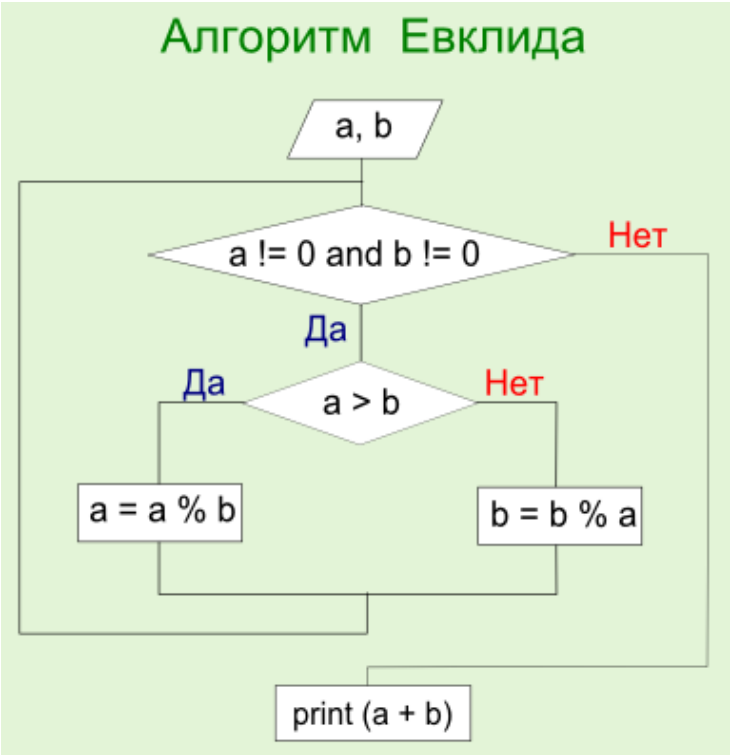


Рассмотренные преобразования из блок-схем в КА и обратно с доопределением и преобразованиями можно использовать для доопределения программы и автоматического тестирования логики программ.

ПРИМЕРЫ АЛГОРИТМОВ

1. Алгоритм Евклида (нахождение наибольшего общего делителя)

Алгоритм Евклида — это алгоритм нахождения наибольшего общего делителя (НОД) пары целых чисел. НОД — это число, которое делит без остатка два числа и делится само без остатка на любой другой делитель данных двух чисел. Проще говоря, это самое большое число, на которое можно без остатка разделить два числа, для которых ищется НОД.



Описание алгоритма нахождения НОД делением:

1) бóльшее число делим на меньшее;
2) если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла);
3) если есть остаток, то бóльшее число заменяем на остаток от деления;
4) переходим к п. 1.

Пример

Найти НОД для 30 и 18.

30 / 18 = 1 (остаток 12).

18 / 12 = 1 (остаток 6).

12 / 6 = 2 (остаток 0). Конец: НОД — это делитель. НОД (30, 18) = 6.

2. Перебор делителей («тестирование простоты»)

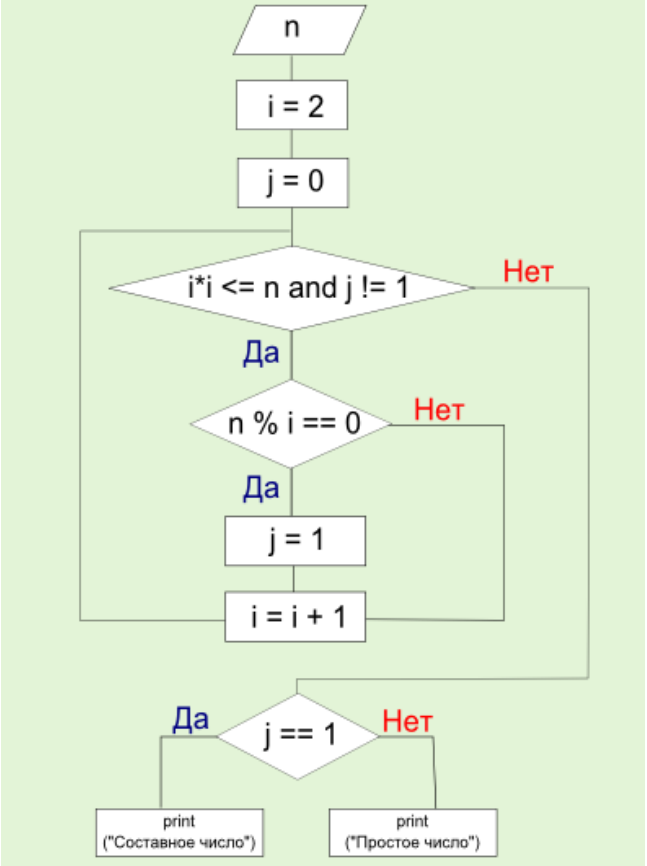
Перебор делителей — это алгоритм, предназначенный для определения, какое число перед нами: простое или составное.

Алгоритм заключается в последовательном делении заданного натурального числа на все целые числа, начиная с двойки и заканчивая значением, меньшим или равным квадратному корню тестируемого числа. Если хотя бы один делитель делит тестируемое число без остатка, то оно является составным. Если у тестируемого числа нет ни одного делителя, делящего его без остатка, то такое число является простым.

3. Решето Эратосфена — алгоритм определения простых чисел

Решето Эратосфена — это алгоритм нахождения простых чисел до заданного числа *n*. В процессе выполнения данного алгоритма постепенно отсеиваются составные числа, кратные простым, начиная с 2.

Алгоритм "Перебор делителей"



Алгоритм "Решето Эратосфена"