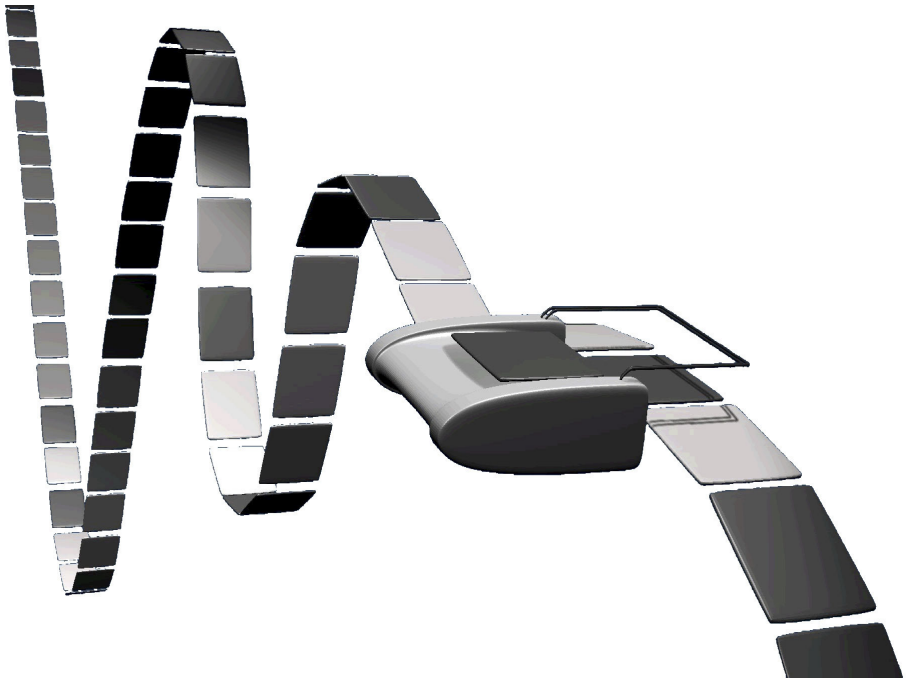
В.И. Поляков, В.И. Скорубский

Основы теории алгоритмов

Учебное пособие по дисциплине «Математическая логика и теория алгоритмов»



Санкт-Петербург

2012

Министерство образования и науки
Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

В.И. Поляков, В.И. Скорубский

Основы теории алгоритмов

Учебное пособие по дисциплине «Математическая логика и теория алгоритмов»

Санкт-Петербург

2012

Поляков В.И., Скорубский В.И. Основы теории алгоритмов. – СПб: СПб НИУ ИТМО, 2012. – 51 с.

Пособие содержит обзор моделей алгоритма:

- алгоритмы распознавания регулярных языков конечными автоматами;
- свойства читающих, записывающих конечных автоматов и автоматов с выходом;
- преобразования блок-схем в конечные автоматы и регулярные выражения;
- машины Тьюринга и Поста;
- ассоциативные вычисления;
- рекурсивные функции.

Приводятся задания для преобразования регулярных выражений в конечные автоматы и блок-схемы.

Пособие предназначено для студентов, обучающихся по направлениям 230100 «Информатика и вычислительная техника» и 231000 «Программная инженерия».

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики».

Содержание

ВВЕДЕНИЕ

Современное формальное определение **алгоритма** было дано в 30 - 50-х гг. XX века в работах А. Тьюринга[1], Э. Поста[2], А. Чёрча[3], Н. Винера[4], А. А. Маркова[5].

Само слово «алгоритм» происходит от имени учёного Абу Абдуллах Мухаммеда ибн Муса аль-Хорезми. Около 825 г. он написал сочинение, в котором впервые дал описание придуманной в Индии позиционной десятичной системы счисления. К сожалению, арабский оригинал книги не сохранился. Аль-Хорезми сформулировал правила вычислений в новой системе и, вероятно, впервые использовал цифру 0 для обозначения пропущенной позиции в записи числа (её индийское название арабы перевели как as-sifr или просто sifr, отсюда такие слова, как «цифра» и «шифр»). Приблизительно в это же время индийские цифры начали применять и другие арабские учёные. В первой половине XII века книга аль-Хорезми в латинском переводе проникла в Европу. Переводчик, имя которого до нас не дошло, дал ей название Algoritmi de numero Indorum («Алгоритми о счёте индийском»). По-арабски же книга именовалась Китаб аль-джебр валь-мукабала («Книга о сложении и вычитании»). Из оригинального названия книги происходит слово Алгебра.

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” [2].

Функция алгоритмически эффективно вычислима, если существует механическая процедура, следуя которой для конкретных значений ее аргументов можно найти значение этой функции [3].

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

Конечность – выполнение алгоритма при конкретных исходных данных за конечное число шагов.

Для демонстрации алгоритмов в теории используются алгоритмические преобразования слов и предложений формального языка [1, 3].

В **формальных** описаниях алгоритм конструктивно связывают с по-

нятием **машины**, предназначенной для автоматизированных преобразований символьной информации.

Для автоматических вычислений разрабатываются модели алгоритмов распознавания языков и машина, работающая с этими моделями. Таким образом, соединяют математическое, формальное определение алгоритма и конструктивное, позволяющее реализовать модели вычислительными машинами.

Алгоритмические вычисления применяются во всех областях науки и техники – монография Д. Кнута [1] рассматривает разнообразные проблемно-ориентированные алгоритмические решения.

Графический интуитивный метод построения алгоритмов в виде **схем** и **диаграмм** сохраняет актуальность, поддерживается стандартами на языки редактирования.

Объектно-ориентированное обобщение алгоритмических языков позволяет активизировать алгоритмическое мышление и организовать масштабное программирование, опираясь на огромные ресурсы производительности и объемы памяти ЭВМ, постоянно наращиваемые стандартные библиотеки.

Теория алгоритмических языков и компиляторов, безусловно, имеет отношение к алгоритмизации, но является самостоятельной областью знаний и применения алгоритмов.

Общая Теория алгоритмов занимается проблемой эффективной **вычислимости.** Разработано несколько формальных определений алгоритма, в которых эффективность и конечность вычислений может быть определена количественно – числом элементарных шагов и объемом требуемой памяти.

Подобными моделями алгоритмических преобразований символьной информации являются:

- конечные автоматы;
- машина Тьюринга;
- машина Поста;
- ассоциативное исчисление или нормальные алгоритмы Маркова;
- рекурсивные функции.

Некоторые из этих моделей лежат в основе методов программирования и используются в алгоритмических языках.

В современной **программной инженерии** алгоритмы, как методы решения задач, занимают ведущее место по сравнению с традиционной математикой. Причем не важно, существует или нет чистое алгоритмическое решение в абстрактных моделях алгоритмов. Если решение задачи необходимо, широко используется эвристика, а “**доказательством”** работоспособности алгоритма является успешное его **тестирование.**

1. РЕГУЛЯРНЫЕ ЯЗЫКИ

Для того, чтобы представить формальное описание алгоритма необходимо формальное описание решаемой задачи. В большинстве случаев описание задачи неформальное (вербальное) и переход к алгоритму неформальный и требует верификации и тестирования и многократных итераций для приближения к решению.

Верификация (от лат. *verus* – «истинный» и *facere* – «делать») – проверка, способ подтверждения каких-либо теоретических положений, алгоритмов, программ и процедур путем их сопоставления с опытными (эталонными или эмпирическими) данными, алгоритмами и программами [4].

Тестирование применяется для определения соответствия предмета испытания заданным спецификациям [4].

К сожалению, теория алгоритмов не дает и не может дать как универсального, формального способа описания задачи, так и ее алгоритмического решения. Однако ценность теории состоит в том, что она дает примеры таких описаний и дает определение алгоритмически неразрешимых задач.

Один из примеров представлен регулярным языком, и задача формулируется как разработка алгоритма для распознавания принадлежности любого предложения к конкретному регулярному языку. Доказывается, что регулярный язык может быть формально преобразован в **модель алгоритма** решения этой задачи за конечное число шагов. Этой моделью является **конечный автомат.**

Расширения регулярных языков находят применение при описании лексики формальных алгоритмических языков, при описании алгоритмов, редактирования поиска по шаблону, в современном программировании (языки Perl, Java, Phyton, C#, PHP), в компиляторах и системах управления обработкой данных, как интерактивный язык в операционных системах. Следовательно, для алгоритмического решения задач из этих областей может быть использован конечный автомат.

Алфавит языка обозначается как конечное множество символов. Например: **Σ={a,b,c,d}, Σ={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(**Σ**).

Языки S=L(**Σ**) могут содержать неограниченное число слов, для их определения используют различные формальные правила. В простейшем случае это **алгебраическая формула**, которая содержит операции формирования слов из символов алфавита и ранее полученных слов.

Рассмотрим следующие операции формирования новых множеств из существующих множеств слов:

1) Символы алфавита могут соединяться **конкатенацией** (сцепление, соединение) в цепочки символов-слов, которые соединяются в новые слова.

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

Произведение S1|S2=S1S2 множеств слов S1 и S2 - это множество всех различимых слов, построенных конкатенацией соответствующих слов из S1 и S2.

Если S1={a, aa, ba}, S2={e, bb, ab}, то S1S2={a, aa, ba, abb, aabb, baab, …}.

Для конкатенации выполняется ассоциативность, но коммутативность и идемпотентность не выполняются:

$S1S2 \neq S2S1$;

$SS \neq S$.

2) **объединение** $(S1 \cup S2)$ или $(S1+ S2)$ множеств

S1={a, aa, ba}, S2={e, bb, ab}, $S1 \cup S2$={a, aa, ba, e, bb, ab}.

Для операции объединения выполняются следующие законы:

Коммутативность объединения $S1 \cup S2=S2 \cup S1$.

Идемпотентность объединения $S \cup S=S$.

Ассоциативность объединения $S1 \cup (S2 \cup S3) =(S1 \cup S2) \cup S3$.

Дистрибутивность конкатенации (умножения) и объединения

$S1(S2 \cup S3) = S1S2 \cup S1S3$.

3) **Итерация множества** {S}* состоит из пустого слова и всех слов вида $S^0=e$, $S^1=S$, $S^2=SS$, $S^3=SSS$.

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

Ассоциативность итерации $S1 * (S2 * S3) =(S1 * S2)* S3$.

Дистрибутивность объединения с итерацией

$S1 *(S2 \cup S3) = S1*S2 \cup S1*S3$.

Если a, b – любые регулярные выражения, то

$(a \cup b)^* = (a^* \cup b^*)^* = (a^*b^*)^* =(a^*b)^*a^*$;

$a^*=a^*a^*=(a^*)^*=(a \cup a^2 \cup .. \cup a^k)^*$;

$(a^*b)^*=(a \cup b)^*b$.

Таким образом, формулы могут содержать скобки и могут быть преобразованы с использованием этих законов.

Регулярные выражения допускают формальные алгебраические преобразования,

Языки, определяемые регулярными выражениями, называются **регулярными языками**, а множество 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.

Регулярное выражение, определяющее правильное арифметическое выражение [5].

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,** … и регулярное выражение, определяющее регулярный язык,

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

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

2. КОНЕЧНЫЕ АВТОМАТЫ

2.1. Читающие конечные автоматы

2.1. 1. Конечные автоматы - модель алгоритма распознавания предложений регулярного языка

Алгоритм распознавания предложений регулярного языка называют **конечным автоматом (КА)** [5].

Определение. Конечный автомат определяется символами

M=(Q, Σ, δ, q0, F), где

Q={q0, q1, …, qn} – конечное множество состояний;
Σ={a, b, c, ...} – входной алфавит (конечное множество);
δ: Q* Σ → {Pj } – функция переходов, Pj - подмножество Q.

Конечное множество значений для этого функционального отношения может быть определено перечислением в **таблице переходов:**

Q	Σ	Pj
qi	a	qj

q0 - начальное состояние;
F - множество заключительных состояний.

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

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

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

Работа КА или выполнение алгоритма распознавания слов регулярного языка могут быть представлены последовательностью шагов, которые определяются текущим состояние Q, входным символом **Σ** и следующим состоянием Pj.

Используется конструктивное описание принципа работы КА как машины M, имеющей следующую организацию:

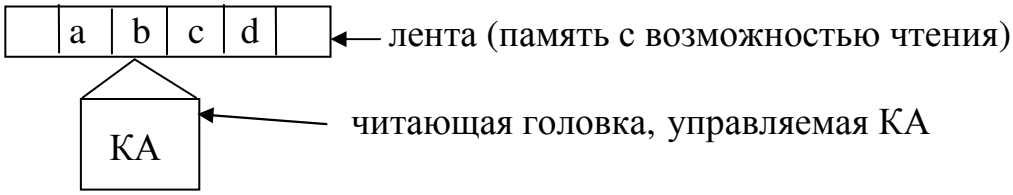


Рис 2.1. Машина, исполняющая Конечный автомат

КА читает входной символ в текущем состоянии qi ∈ Q, переходит в следующее состояние qj ∈ Q и сдвигает читающую головку к следующему символу.

Автомат допускает входное слово, если приходит в заключительное состояние из F, последовательно считывая символы из памяти и переходя в следующие состояния в соответствии с таблицей переходов. При этом входное слово исчерпано и автомат останавливается.

Конфигурация КА k=(q, ω), где q-текущее состояние КА, ω - непрочитанная цепочка символов слова на ленте, включая символ под читающей головкой.

k = (q, ω) текущая конфигурация;

$k_0 = (q_0, \omega 0)$ начальная конфигурация;

k_f = (q, e), где q ∈ F, - заключительная конфигурация и (e) – символ, обозначающий конец строки.

Шаг алгоритма - переход из одной конфигурации КА в другую

Ki →Kj или (qi, ωi) → (qj, ωj).

Функция переходов, заданная в табличной форме, может быть представлена графом переходов G=(Q, R), где Q - вершины графа, R - бинарное отношение между парой вершин, которое представлено множеством дуг (qi, qj).

(qi, qj) ∈ Q*Q, если существует символ **a** ∈ **Σ** и **δ** (qi, a) = qj.

На дугах графа (qi, qj) отмечаются соответствующие символы алфавита.

Пример 2.1.

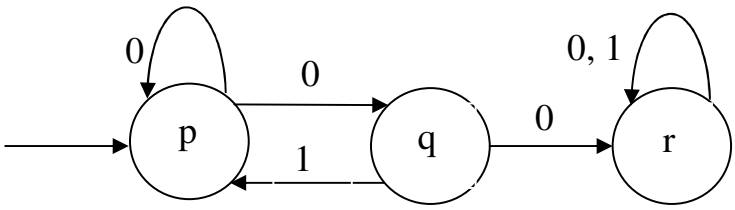


Рис.2.2. Детерминированный читающий КА

Для ДКА, приведенного на рис. 2.2 состояния Q={p, q, r}, входной алфавит **Σ**={0,1}, начальное состояние p, конечное - r.

Таблица переходов ДКА

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) → (q, 1001) → (p, 001) → (q, 01) → (r, 1) → (r, e);

(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 ω0 , если существует

(q0, ω0) →*(qf, e),

где →*( ), обозначает **транзитивное замыкание** и существует путь, соединяющий q0 и qf для входного слова ω0.

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

2.1.2. 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 [5, 6]:

1) Разметка состояний устанавливает позиции символов в регулярном выражении (п.1, пример 1.2)

$$L(M)=(mi + i)((p + m) i)^*$$
$$0\ 12\ 3\quad 4\quad 5\ 6$$

2) Рассматриваются 2^6+1 подмножеств мест предполагаемых состояний и формируются по регулярному выражению возможные переходы между состояниями. Очевидная избыточность состояний устраняется, так как в некоторые состояния переходы отсутствуют.

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 ≤ n+1** – количество позиций в регулярном выражении (в данном примере m=7).

Для рассматриваемого примера можно построить КА, используя верхнюю оценку

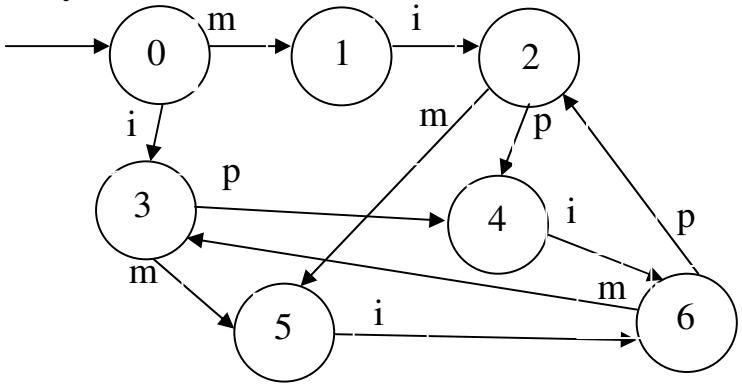


Рис.2.3. Преобразование регулярного выражения в КА

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

Слово *различает* состояния qi и qj, если

((qi, ω) →*(qm, e) (qj, ω) →* (qn, e) и **одно** из состояний qm или qn принадлежит F,

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

Состояния не различимы (*эквивалентны* qi=qj), если они не различимы при k → ∞**.**

В автомате рис.2.3 каждую пару эквивалентных состояний {2,3}, {4,5} заменяем одним состоянием {2,3}→ 2, {4,5}→3,

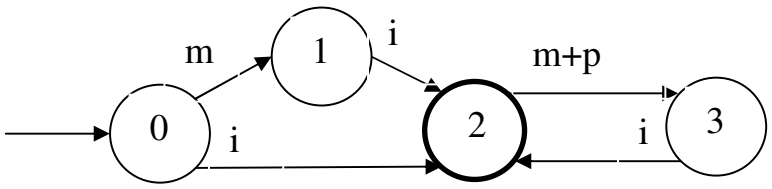


Рис.2.4. ДКА распознавания L(M)

Рассмотренная методика преобразования может быть упрощена с учетом очевидных алгебраических свойств операций регулярного выражения:

- последовательности символов в скобках с операцией (+) имеют общее начальное состояние перед открывающей скобкой и конечное после закрывающей скобки;

- итерация обозначает цикл с произвольным числом повторений, при пустом числе циклов сохраняется состояние, с которым входим в цикл;

- в конкатенации нескольких символов различимые состояния заменяют конкатенацию между парой символов;

$$L(M)=(mi + i)((p + m)i)^*$$
$$0\ 1 \quad 2 \qquad 3\ 2$$

Состояния размешаем в найденные для них позиции, исключая повторения:

$$L(M)=(mi + i)(\ (\ p + m\ )i\ )^* = 0(m1i + i)2((p + m)3i2)^*$$
$$0\ 1 \quad 2 \qquad 3\ 2$$

Заменяем линейную помеченную строку графом КА рис.2.4. Начальное состояние – 0. Конечное состояние – 2.

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

Пример 2.2.

$$L(M)=aa^*bd^* + ad^* = (aa^*b + a)d^* = (0a1(a1)^*b + 0a)2(d2)^*$$
$$0\ 1 \qquad 2$$

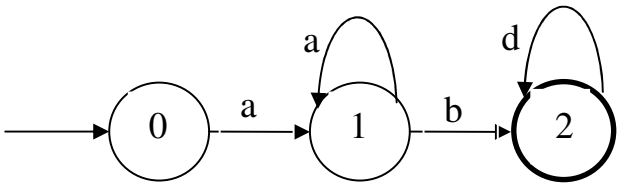


Рис.2.5. НДКА распознавания L(M)

При исполнении алгоритма (НДКА) в соответствии с методом поиска по шаблону [7] сохраняются номера позиций, в которых возможен выбор нескольких переходов в КА. Если поиск неудачный, то осуществляется возврат к ближайшей позиции **(бектрекинг)**. Шаги по различным ветвям выполняются параллельно:

$$(0, add) \rightarrow (1, dd)$$
$$\rightarrow (2, dd) \rightarrow (2, d) \rightarrow (2, e).$$

Пример 2.3. [5]

Всякая подцепочка из 3 символов содержит 2 нуля -

$$((001)^*(100)^*(010)^*)^*.$$

Число 0 делится на 2, а число 1 на три -

$$((00)^*(111)^*)^* = (00 + 111).$$

Множество цепочек, в которых каждая пара 00 находится перед парой:
11 $(((00)(11)^*01^*)^*$.

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 по методике [5] представим следующей процедурой:

- на i-ом шаге множество состояний ДКА обозначим Qi;
- 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 $Q_i \neq Q_{i+1}$.

Лемма. Число состояний в ДКА не превышает $\mathbf{2^n-1}$, где n **–** число состояний в НДКА.

Число $\mathbf{2^n-1}$ **–** количество собственных подмножеств для множества, состоящего из n различимых элементов.

Следовательно, процедура конечна.

Пример 2.4. для НДКА рис.2.5.

$Q_0 = \{0, 1, 2\}$
$Q_1 = \{0, 1, 2, \{1, 2\}\}$

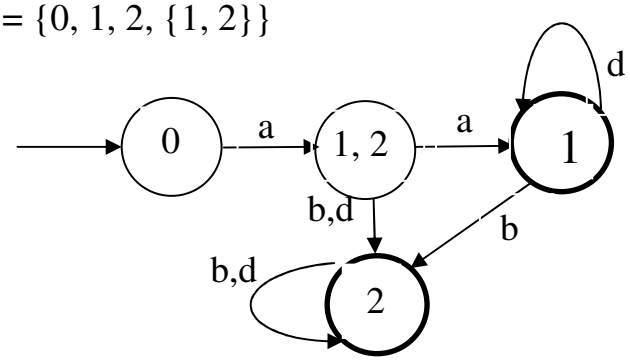


Рис.2.6. ДКА получен из НДКА

Эквивалентный детерминированный автомат (рис. 6) содержит 4 состояния и определен тем же регулярным выражением L(M') = L(M).

Пример 2.5. НДКА задан таблицей переходов

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

Преобразование в ДКА:
Q0 = {p, q, r, s};
Q1 = {p, q, r, s, {q, s}, {r, q}};
Q2 = {p, q, r, s, {q, s}, {r, q}, {r, s}};
Q3 = {p, q, r, s, {q, s}, {r, q}, {q, r, p},{r, s}};
Q4 = {p, q, r, s, {q, s}, {r, q}, {q, r, p},{r, s}, {q, r, s}};
Q5 = Q4;
Q4 = {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

	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 9 состояний. Если S-конечное состояние в НДКА, то в ДКА выделены конечные состояния {S, X, M, F}, содержащие S.

2.1.3. Преобразование конечного автомата в регулярное выражение

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

Метод исключения состояний [5]:

Для допускающего (финального) состояния исключить промежуточное состояние (кроме начального q0), следующим образом:

пусть qi – предшествующее состояние, qs – промежуточное и qj – следующее, таким образом:

1) над каждой дугой (qi, qs) записать регулярное выражение Qis;

2) на дуге (qs, pj) – выражение Qsj,

3) петля в s обозначается регулярным выражением S*;

5) дугам (qi, qj) после исключения промежуточного состояния qs приписываются регулярные выражения Qis S*Qsj.

После удаления всех промежуточных вершин остаются начальное состояние и финальные. Финальные состояния объединяются суммированием соответствующих регулярных выражений. В частном случае, если q0 ∈ F, останется одно состояние q0 и регулярное выражение приписано дуге.

Когда останутся два состояния - начальное и финальное, то промежуточное регулярное выражение может быть записано в виде:

(R + SU*T)*SU*.

Полученный КА представлен на рис. 2.7.

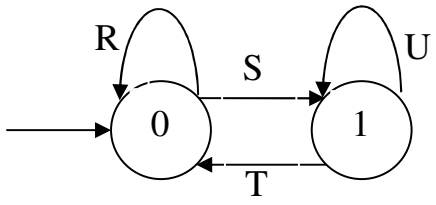


Рис.2.7.Формирование регулярного выражения

Пример 2.6. Преобразуем ДКА (рис.2.2) в регулярное выражение:

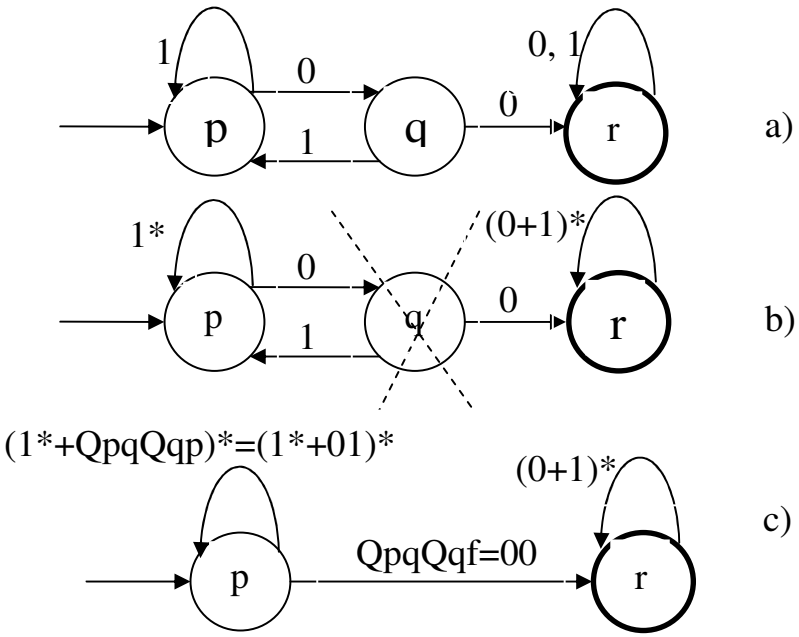


Рис.2.8. Три стадии преобразование КА в регулярное выражение:
a) исходный КА, b) исключение состояния, c) свертка для двух состояний

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

R=(1* + QpqQqp)*=(1*+01)*
U=(0 + 1)*
S=QpqQqf=00
T отсутствует
(R + SU*T)*SU*= (1*+01)* 00(0 + 1)*
L(M)=(1*+01)*00(0+1)*
0 0 10 23

Состояния qi и qj связаны дугой (qi,qj), если существует смежный символ в регулярном выражении, в этом случае дуга помечается соответствующим символом. Получен НДКА.

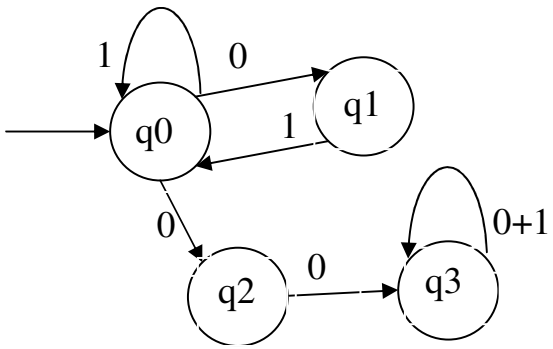


Рис. 2.9. Преобразование КА в регулярное выражение и обратно в КА

Преобразование в ДКА производится следующим образом

Q0 = {q0, q1, q2, q3}
Q1 = {q0, q1, q2, q3,{q1, 2}}.

Состояния q1 и q2 не достижимы из начального состояния q0 и могут быть исключены. Получим исходный КА рис.2.2. с состояниями

Q1= {q0, q3,{q1, 2}}, что и подтверждает корректность преобразования КА в регулярные выражения.

2.2. Распознавание нерегулярных языков

Нерегулярные языки отличаются тем, что на повторение символов задаются ограничения.

Например, $L(M)=\{0^n1^n$, 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 (pumping):

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 $\mathbf{xy}\leq\mathbf{n}$ и для любого $k\geq 0$ слово $\mathbf{xy^kz}$ принадлежит L.

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

Так, например, для $\mathbf{L(M)=\{0^n1^n}$, где $n\geq 0\}$, если k=n=5, то $\mathbf{0^51}$ не принадлежит **L(M).**

Примерами нерегулярных языков являются:

- все двоичные коды, длины которых простые числа (при записи регулярного выражения появляется затруднение, поэтому можно применить лемму для проверки регулярности [5];

- множество двоичных кодов, которые начинаются с единицы и являются простыми целыми числами,

2.2.1. 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 2.7. Пусть система состоит из однотипных автоматов [8]:

L(Σ), Σ={a, b, L}.

L(A)=ab + aLb, где L запускает новый образец автомата.

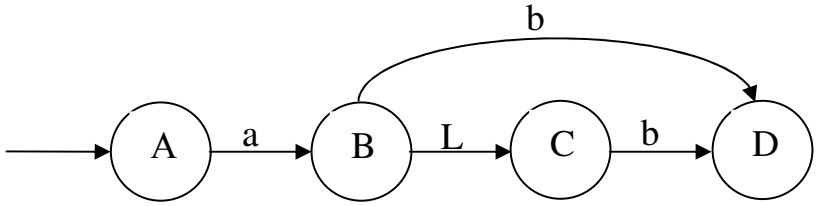


Рис.2.10. Рекурсивный КА запускает себя в L

Нерегулярный язык, распознаваемый рекурсивным вызовом автомата L,

$L(S) = ab + aabb + aaabbb + = \{a^n b^n\}$, где n>0.

Работа КА при исполнении алгоритма распознавания нерегулярного язык происходит следующим образом:

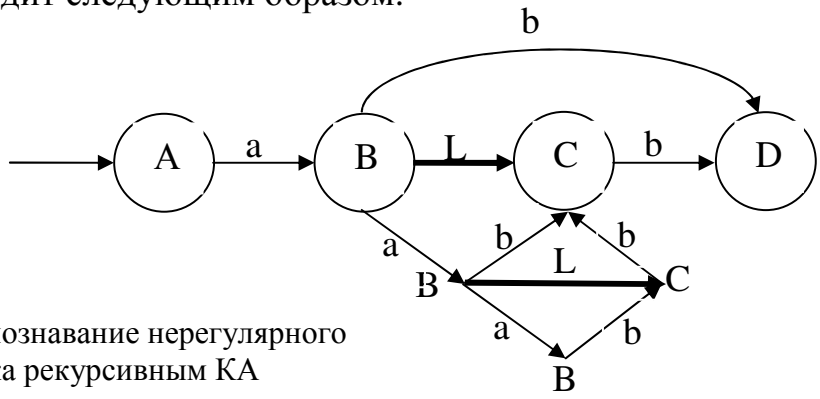


Рис.2.11. Распознавание нерегулярного языка рекурсивным КА

2.3. Конечные автоматы с выходом

Для выполнения автоматического распознавания слов регулярного языка необходимы внешние признаки состояний, в которых пребывает автомат после завершения чтения слова.

При этом:

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

- входной алфавит Σ* включает символ e, который заканчивает любое

слово;

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

Рассмотрим КА рис.2.5. При доопределении Q={A, B, F, D, Z}, Σ*={a, b, d, e}, W={α, β, γ}.

W={α, β, γ} ={начальное состояние - α, завершение ввода (слово не принадлежит регулярному языку) - β, слово принадлежит регулярному языку - γ}

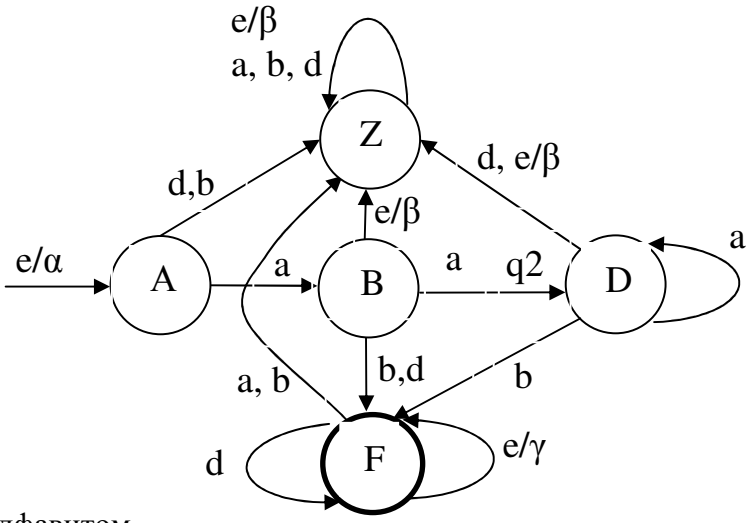


Рис.2.12. КА с выходным алфавитом

2.4. Событийная интерпретация конечного автомата с выходом

Схема машины для выполнения алгоритма распознавания может быть определена интерпретацией табличного описания КА событиями в форме логических уравнений.

Событиями являются предикаты:

- идентификации и выбора состояний A=(is A);
- идентификации входов a=(is a);
- выбора выходов α =(is α).

Уравнения перехода в состояния:

A’=Ae

B’=Aa ∨ Ba

D’=Ba ∨ Da

Z’=a(b ∨ d) ∨ Be ∨ D(d ∨ e) ∨ Z(a ∨v b ∨ d ∨ e)

F’= B(b ∨ d) ∨ F(d ∨v e).

Уравнения выхода:

α =Ae

β =Be ∨ Ze ∨ De

γ =Fe.

При этом, для выполнения КА решается система уравнений для заданной входного слова, работа завершается формированием выхода из W.

В итерационных вычислениях с использованием уравнений необходимо различать переменные, обозначающие текущие и следующие состояния, например, в B'=Aa $\vee$ Ba, справа текущие состояния, а слева – следующие.

Более простая схема вычислений в виде Switch-программы рассмотрена в [9], 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.

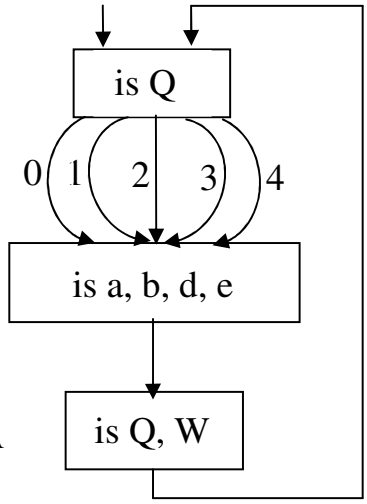


Рис.2.13. Switch-программа интерпретации КА системой уравнений в событиях

2.5. Записывающие конечные автоматы

Конечные автоматы с выходом позволяют определять не только алгоритмы - распознаватели, но и формировать простые операции преобразования (редактирования) слов языка. Символы этих операций являются командами для ручных преобразований или являются входными для формирования выходных слов в памяти с записью.

Конструкция КА может быть усовершенствована для этой цели добавлением записывающей ленты (записывающей памяти)

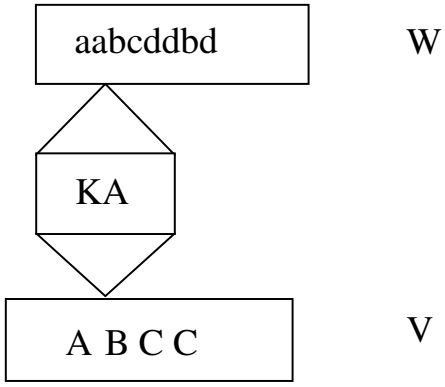


Рис.2.14. Записывающий КА, где W- цепочка символов, считываемая с входной памяти (ленты), V-цепочка символов, записываемых в выходную память (ленту)

Конфигурация автомата K(qi, ω, w) включает текущее состояние, оставшуюся входную цепочку символов и выходную строку символов

Пример 2.8. [5].

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

Входной алфавит Σ = {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, a** регулярное выражение, для которого требуется редактирование

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)((p + m)i)*.

КА - преобразователь исходного предложения представлен на рис. 2.15.

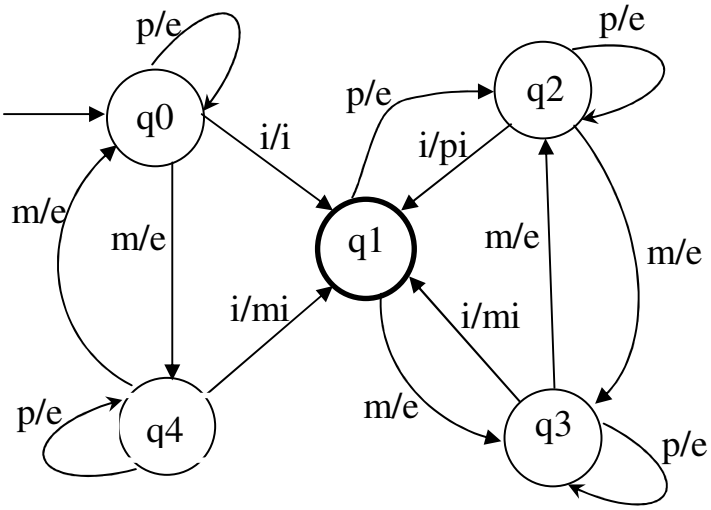


Рис.2.15. Записывающий КА

3. ПРИМЕНЕНИЕ КОНЕЧНЫХ АВТОМАТОВ В ПРОГРАММИРОВАНИИ

Конечные автоматы применимы как модели алгоритмов для непосредственного программирования задач управления объектами [9].

В разделе 2.4. эта задача рассматривалась как событийная интерпретация КА. Основная проблема, которая возникает в случае прямого программирования в автоматной модели, - определение множества состояний КА и входного алфавита. Эта проблема легко решается в преобразовании блок-схемы программы в КА.

Утверждение. Существует конструктивный метод преобразования блок-схем алгоритмов в КА [10].

Рассмотрим преобразование блок-схемы алгоритма умножения S=A*B, построенной по школьному методу.

Содержательное описание:
сомножители $A=(a_{n-1}a_{n-2}\ldots a_1a_0)$ и $B=(b_{n-1}b_{n-2}\ldots b_1b_0)$ – n-разрядные десятичные числа, A - множимое, B - множитель.

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

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

Предполагается, что вычисление частичных произведений, сдвиг и суммирование – эффективные операции.

Схема алгоритма умножения и ее преобразование в КА приведены на рис.3.1.

Операторные вершины, а также начальная (ввод) и конечная (вывод) обозначаются как состояния КА и имеют смысл **задержки,** необходимой для выполнения соответствующих операций. В алгоритме можно выполнять указанные операции или формировать команды для их автоматического исполнения в машине. Таким образом, множество состояний Q={q0, q1,..., q5}. Входной алфавит определяет множество символов, кодирующих обозначенные в условных вершинах предикаты и принимающие двоичные значения {true, false}. Переходы из одного состояние в другое могут быть представлены конъюнкциями значений входных переменных {$x1$, $x2$}.

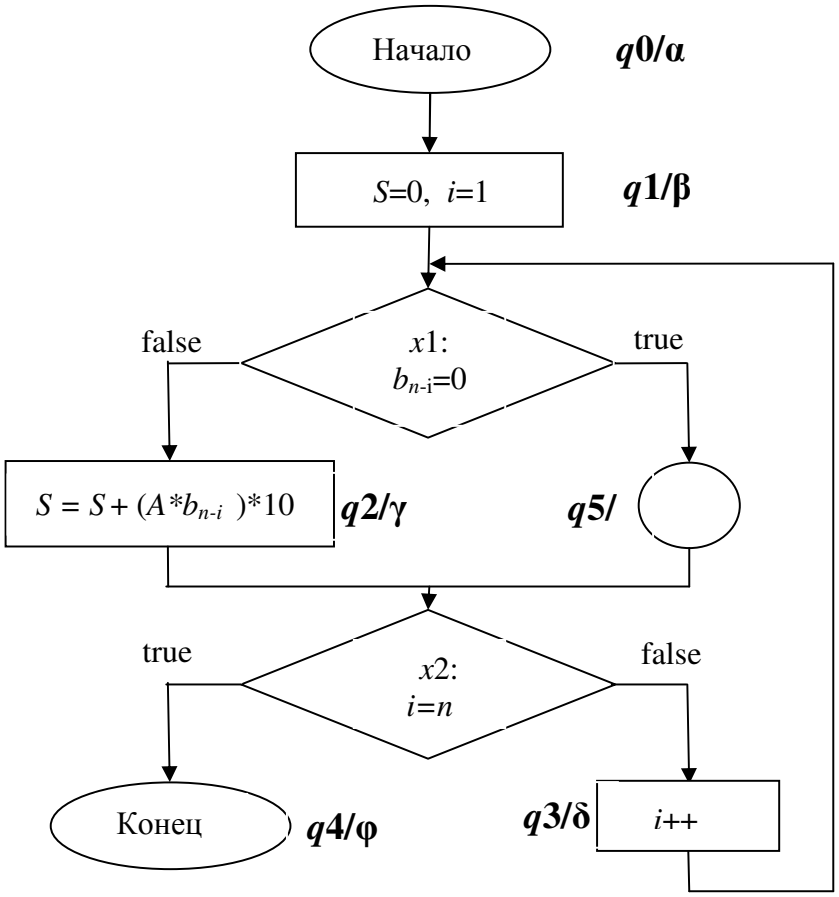


Рис.3.1. Преобразование блок-схемы в КА

Функция выходов, формируемых в состояниях, обозначим символами-командами выходного алфавита W={α, β, δ, γ, φ} и поставим им в соответствие состояния КА.

Функция переходов КА определяется в следующей таблице:

Таблица переходов КА

	q0	q1	q2	q3	q4	q5
q0	-	***T***	-	-	-	-
q1	-	-	$\bar{x}1$	-	-	***x*1**
q2	-	-	-	$\bar{x}2$	***x*2**	-
q3	-	-	$\bar{x}1$	-	-	***x*1**
q4	-	-	-	-	-	-
q5	-	-	-	$\bar{x}2$	***x*2**	-

Метод преобразования блок-схем в КА хорошо известен и апробирован, в основном, для дальнейших преобразований алгоритмов в схемы [10].

Очевидно, что модель алгоритма в виде КА легко преобразуется в блок-схему и, следовательно, всегда может быть выполнен переход к

формальному описанию алгоритма на алгоритмическом языке и в виде программы ВМ.

Методы оптимизации преобразования блок-схем можно найти в [12].

В преобразованиях алгоритмов, представленных блок-схемами, в КА могут быть использованы **частичные КА**.

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

- для распознавателей языка это ограничение на регулярный язык;
- для алгоритмов преобразования данных – ограничение на область значений данных - результатов выполнения преобразований;
- для алгоритмов управления – ограничение на входные данные.

В случае распознавателей – признаки частичных автоматов могут быть использованы для контроля и выявления предложений, не принадлежащих языку **при тестировании**.

В остальных случаях частичные автоматы также используются для тестирования. В реальных программах возможно доопределение переходов любым способом, так как соответствующие переходы в правильно работающей программе не возможны. Доопределение можно использовать для контроля и тестирования программы в процессе исполнения.

Выполним замену логических условий символами входного алфавита $\Sigma^*=\{T, a, b, c, d\}$, где $a=\overline{x}1$, $b=x1$, $c=\overline{x}2$, $d=x2$, $q0$ - начальное состояние, $q4$ - финальное состояние.

Таблица переходов КА

	$q1$	$q2$	$q3$	$q4$	$q5$
$q0$	T	-	-	-	-
$q1$	-	a	-	-	b
$q2$	-	-	c	d	-
$q3$	-	a	-	-	b
$q4$	-	-	-	-	-
$q5$	-	-	c	d	-

Модель алгоритма умножения в символах входного алфавита приведена на рис.3.2.

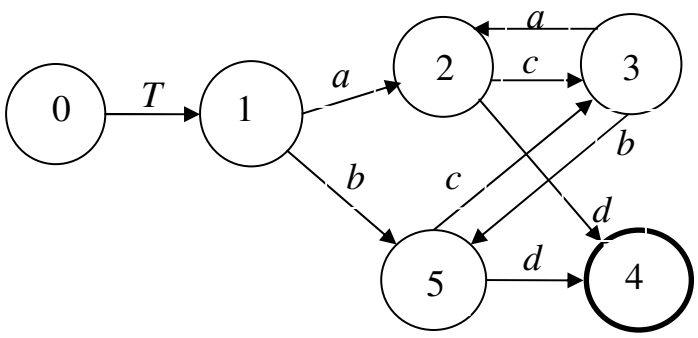


Рис. 3.2. Модель алгоритма умножения

$L(M) = (0T1)\{[(1a2)(((2c3)(3a2))^*(2d4)+(2c3)((3b5)(5c3))^*(3b5)(5d4)\ ] +$

$+(1b5)[((5c3)(3b5))^*(5d4)+(5c3)((3b5)(5c3))^*((3a2)(2c3))^*(3a2)(2d4)]\}=$

$=T[a((ca)^*d+c(bc)^*bd)+b((cb)^*d+c(bc)^*(ac)^*ad]=$

$=T[a((ca)^*+c(bc)^*b)+b((cb)^*+c(bc)^*(ac)^*a]d=$

$=T[a((ca)^* +c(bc)^*b) +b((cb)^* +c(bc)^*(ac)^*a]d=$

$= T[a((ca)^* +tb) +b((cb)^* +t(ac)^*a]d,$

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

4. МАШИНА ТЬЮРИНГА

Универсальный КА, применяемый для решения любой алгоритмически разрешимой задачи, в теории алгоритмов и вычислений называется **машиной Тьюринга** [5].

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

Множество входных и выходных символов не различаются - единый алфавит на входе (чтение) и на выходе (запись) $\Sigma=W$.

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

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

Принципы работы машины:

Начальное слово - в алфавите Σ размещается на ленте.

При чтении очередного символа с ленты выполняется определенная команда K, автомат переходит в следующее состояние и в ту же позицию на ленте записывается символ;

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

Машина Тьюринга реализует алгоритм, если она всегда применима к начальной информации (слову), изображающей условия задачи, и перерабатывает входное слово в результирующую информацию (выходное слово).

Конфигурация машины при исполнении алгоритма – K(qi,s), где qi - состояние автомата, s - текущая строка в памяти. Шаг алгоритма – переход от одной конфигурации к другой после чтения символа.

Машина Тьюринга имеет фундаментальное значение в теории алгоритмов как формальное определение алгоритма в строгой и точной форме - в виде **схемы** машины.

Основная **гипотеза** теории алгоритмов: Машина Тьюринга решает любую алгоритмически разрешимую задачу.

Вопрос **алгоритмической разрешимости** (существования алгоритма решения задачи) сводится к доказательству существования машины T, решающей задачу за конечное число шагов. Если число шагов бесконечно, то задача трудно разрешимая или алгоритмически неразрешимая проблема.

Однако в такой постановке проблема доказательства разрешимости задачи сама является неразрешимой, так как отсутствует алгоритм построения такой машины (аналогия с формальным выводом в логике).

Поэтому задача сводится (**приводится**) к частной и более простой задаче, для которой можно доказать, что соответствующей машины Тьюринга построить нельзя и тогда, и более общая задача алгоритмически не

разрешима.

Машина Тьюринга позволяет выполнить алгоритм распознавания для регулярных и нерегулярных языков.

Пример 4.1.

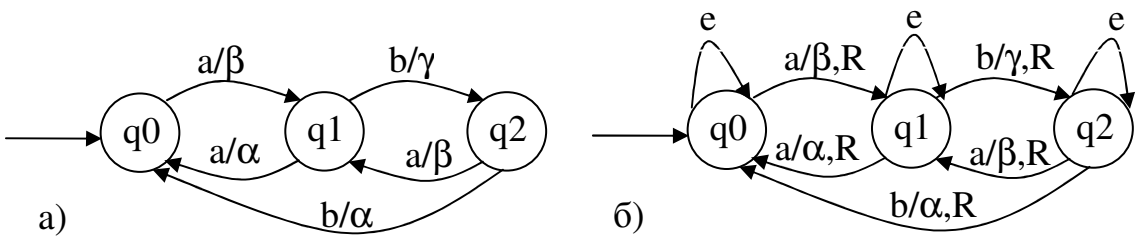


Рис.4.1. Вычисление КА в машине Тьюринга
а) конечный автомат, б) машина Тьюринга

Практического применения такая конструкция не имеет - любая ЭВМ является универсальной машиной Тьюринга с физически ограниченной памятью и может интерпретировать любую машину, если принять как в машине Тьюринга, что память бесконечна.

В работе [11] приводится модель машины Тьюринга на языке Пролог.

Следующая формулировка отечественного математика Крупского В.Н. гипотезы Тьюринга связывают ее с программированием и алгоритмическими языками.

Тезис Тьюринга (неформальный): Каждую вычислимую (программой какого угодно языка программирования) функцию типа

$\sum^* \to \sum^*$ можно вычислить на подходящей машине Тьюринга.

Неформальность Тезиса Тьюринга состоит в том, что он не может быть полностью обоснован математическими средствами (доказан как математическая теорема). В то же время, все многочисленные попытки его опровергнуть, т.е. предложить язык программирования с большими вычислительными возможностями, оказались безуспешными. Причина невозможности полного математического доказательства в том, что не определяются семейства языков, о которых идет речь. Если их заменить на достаточно информативное описание семейства языков программирования, то соответствующий частный случай Тезиса станет обычным, “поддающимся доказательству” математическим утверждением.

Пример 4.2.

Пусть семейство состоит из языков C и PASCAL, языки имеют полное описание их синтаксиса и операционной семантики. Языки обладают компиляторами в ASSEMBLER, поэтому для программирования

соответствующего частного случая Тезиса Тьюринга требуется построить компилятор, преобразующий ассемблерный код в программу для машины Тьюринга. Последнее является весьма трудоемкой, но реалистичной задачей по программированию. Для полного математического доказательства потребуется еще верифицировать программу на языке.

5. МАШИНА ПОСТА

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

В машине Тьюринга следует сконструировать управляющий автомат, а в машине Поста – программу решения задачи.

Следствие – универсальность и простота машины Поста.

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 – останов.

Таким образом, алгоритмическое описание и решение проблемы – машинная **программа.**

Упорядоченный набор команд (программа) применим к данной проблеме и является алгоритмом ее решения, если применение программы завершится командой Stop. Если произойдет останов по неприменимости, то программа не применима к данной проблеме и проблема алгоритмически не разрешима.

Практического значения машина Поста не имеет, в теории вычислительных машин она представляет минимальную универсальную и полную систему команд и элементарную конструкцию.

6. 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 обратная.

Ассоциативное исчисление – совокупность слов в алфавите и **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 Маркова А.А.

Алфавит $\sum=\{a, b, c\}$.

Схема алгоритма – упорядоченная система ориентированных подстановок

$b \to acc$

$ca \to accc$

$aa \to \varnothing$

$bb \to \varnothing$

$cccc \to \varnothing$

- используется первая по порядку подстановка;
- рассматривается самое левое вхождение

$\underline{b}b \to acc\underline{b} \to ac\underline{ca}cc \to ac\underline{a}ccccc \to a\underline{a}cccccccc \to cccc\ cccc \to cccc \to \varnothing$

2. Исчисления Цейтина Г.С.[6]

Алфавит $\sum=\{a, b, c, d, e\}$

схема алгоритма

$ac \to ca$

$ad \to da$

$bc \to cb$

$bd \to db$

$abac \to abace$

$eca \to ae$

$edb \to be$

abcde → acbde → cabde → cadbe →

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 → S.

Теорема Черча [12] доказывает неразрешимость выводимости в виде конечного алгоритма.

4. Синтаксический разбор предложений алгоритмического языка [13,14].

Синтаксис определяется множеством подстановок, применимых к цепочке символов алфавита языка, расширенного промежуточными – терминальными символами. Если цепочка символов прочитана и распознана применением подстановок, то цепочка принадлежит языку. Схему применения подстановок, выполненную в виде читающего и записывающего автомата, которая формирует символическую строку команд, называют **компилятором**.

7. РЕКУРСИВНЫЕ ФУНКЦИИ

Если зафиксировать алфавит A из r букв, то всякое слово R в этом алфавите можно рассматривать как запись некоторого натурального числа в **r-ичной** системе счисления.

Таким образом, исходные данные – слова R в алфавите A можно интерпретировать как натуральные числа. Также можно интерпретировать результат выполнения алгоритма как вычисление значения числовой функции по заданному значению аргумента.

Рекурсивные (возвратные) функции позволяют определить значение от неизвестного к известному (от сложного к простому).

Функция **вычислима**, если существует алгоритм – **эффективная** последовательная процедура вычисления от простого к сложному.

Выделяется **базис элементарных функций**, интуитивно вычислимых, и средства-**операторы** получения из них более сложных функций.

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

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

К элементарным вычислимым функциям относятся:

1) $S(x) = x+1$ - следование – вычисление следующего натурального числа;

2) $0(x) = 0$ – константа нуля;

3) $I_m(x_1 x_2 .. x_n) = x_m$ - 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 $\Phi(N)$, $\Phi(N-1)$, …, $\Phi(0)$.

Возврат-вычисление функций по трассе проводится:

- если трасса известна, то строится циклическая программа;
- прямое вычисление по формуле $\Phi(N)$, если она известна.

Например, сумма членов арифметической прогрессии, или чисел Фибоначчи.

Примеры построения сложных рекурсивных функций на основе элементарных и рекурсии:

1. **Сумма** - вычисляется Ф(x, C) = x+C для заданного x
   Ф(0, C) = C базис,
   итерация Ф(x+1, C) = Ф(x, C) +1.
2. **Произведение** - вычисляется Ф(x, C) = x * C для заданного множителя x
   Ф(0, C) = 0 базис
   итерация Ф(x+1, C) = Ф(x, C) +C.
3. **Факториал -** вычисляется Ф(x) = x!
   Ф(0) = 1
   итерация Ф(x+1) = (x+1)*Ф(x).
4. **Вычисляется** $\Phi(x, C) = x!C^{x+1}$
   Ф(0, C)=C
   итерация Ф(x+1, C) = (x+1)*Ф(x, C)*C.
5. **Вычисляется** $\Phi(x, C) = C^x$
   Ф(0, C) =1
   итерация Ф(x+1, C) = Ф(x, C)*C.

4) **μ-оператор** для определения функции, число повторений которой неизвестно, но задается предикатом.

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

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

В теории алгоритмов показано, что все эффективно вычислимые рекурсивные функции могут быть вычислены машинами Тьюринга.

На практике рекурсивное описание задач является трудоемким и интуитивным, вместе с тем уровень сложности решаемых задач практически доступен, применяется в высокоуровневых алгоритмических языках и компилируется в эффективные вычислительные процедуры.

Язык рекурсивных вычислений РЕФАЛ (рекурсивных функций алгоритмы) предложен **Турчиным В.Ф.**[7].

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

Примеры рекурсивных вычислений

1. ***Факториал:***

$\Phi(0) = 1$

$\Phi(x+1) = (x+1)*\Phi(x)$.

Формируется и сохраняется трасса

$\Phi(10) = 10*\Phi(9) = 10*(9*\Phi(8)) = 10*(9*(8* \ldots\ldots 1*\Phi(0)$

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 $F_n = F_n\text{-}1 + F_n\text{-}2$, $n \geq 2$;

Общая формула Бине

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

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***.

Широко используются в виде производящих функций, в приближениях функций рядами Тейлора, в позиционных системах счисления.

Вычисления рядов во многих случаев можно представить рекуррентными формулами и привести их к итерационным алгоритмам.

Пример 7.1.

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

$$\mathbf{N} = \sum_{i=0}^{m-1} \mathbf{a_i d^i} = a_{m-1}10^{m-1} + a_{m-2}10^{m-2} + \ldots + a_1 10 + a_0 =$$

$$= (\ldots((0 + a_{m-1})10 + a_{m-2})10 + \ldots + a_1)10 + a_0.$$

С использованием обобщенной схемы Горнера[8] [15], полином

преобразуется в скобочную форму, которую можно определить рекуррентной формулой, применяемой для преобразования числа в любую другую позиционную однородную систему счисления.

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

8. КЛАССЫ СЛОЖНОСТИ [4]

В рамках классической теории алгоритмические задачи различаются по классам сложности (P-сложные, NP-сложные, экспоненциально сложные и др.).

Классы сложности - множества вычислительных задач, примерно одинаковых по сложности вычисления. Более узко, классы сложности — это множества предикатов (функций, получающих на вход слово и возвращающих ответ 0 или 1), использующих для вычисления примерно одинаковые количества ресурсов. Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих некоторыми свойствами.

Классом сложности X называется множество предикатов P(x), вычислимых на машинах Тьюринга и использующих для вычисления O(f(n)) ресурса, где n — длина слова x.

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов машины Тьюринга) или рабочая зона (количество использованных ячеек на ленте во время работы).

Класс P – задачи, которые могут быть решены за время, полиномиально зависящее от объёма исходных данных, с помощью детерминированной вычислительной машины (например, машины Тьюринга),

Класс NP — задачи, которые могут быть решены за полиномиально выраженное время с помощью недетерминированной вычислительной машины, то есть машины, следующее состояние которой не всегда однозначно определяется предыдущими. К классу NP относятся задачи, решение которых с помощью дополнительной информации полиномиальной длины, данной нам свыше, мы можем проверить за полиномиальное время. В частности, к классу NP относятся все задачи, решение которых можно проверить за полиномиальное время. Класс P содержится в классе NP. Классическими примерами NP-задач являются задачи о коммивояжёре, нахождение гамильтонова цикла, раскраска вершин графа.

Работу такой машины можно представить как разветвляющийся на каждой неоднозначности процесс: задача считается решённой, если хотя бы одна ветвь процесса пришла к ответу.

Поскольку класс P содержится в классе NP, принадлежность той или иной задачи к классу NP зачастую отражает наше текущее представление о способах решения данной задачи и носит неокончательный характер. В общем случае нет оснований полагать, что для той или иной NP-задачи не может быть найдено P-решение. Вопрос о возможной эквивалентности

классов P и NP (то есть о возможности нахождения P-решения для любой NP-задачи) считается многими одним из основных вопросов современной теории сложности алгоритмов. Ответа на этот вопрос нет. Сама постановка вопроса об эквивалентности классов P и NP возможна благодаря введению понятия NP-полных задач. NP-полные задачи составляют подмножество NP-задач и отличаются тем свойством, что все NP-задачи могут быть тем или иным способом сведены к ним. Из этого следует, что если для NP-полной задачи будет найдено P-решение, то P-решение будет найдено для всех задач класса NP. Примером NP-полной задачи является задача о конъюнктивной форме.

Наиболее часто встречающиеся классы сложности в зависимости от числа входных данных n таковы (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 определённую часть данных входного потока и игнорирующие все остальные данные. Например, чистка 1 квадратного метра ковра вне зависимости от его размеров.

Ряд алгоритмов имеют порядок, включающий $\log_2 n$, и называются логарифмическими (logarithmic). Эта сложность возникает, когда алгоритм неоднократно подразделяет данные на подсписки, длиной 1/2, 1/4, 1/8, и так далее 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, алгоритм Краскала[9] - построение минимального связывающего дерева, n - число ребер графа.

Алгоритм со сложностью O(n) - алгоритм линейной сложности. Например, просмотр обложки каждой поступающей книги - то есть для каждого входного объекта выполняется только одно действие;

Алгоритмы, имеющие порядок $O(n^2)$, являются квадратичными (quadratic). К ним относятся наиболее простые алгоритмы сортировки; алгоритм Дейкстры[10] - нахождение кратчайших путей в графе, n – число вершин графа; алгоритм Прима[11] - построение минимального связывающего дерева, n - число вершин графа [16]. Квадратичные алгоритмы используются на практике только для относительно небольших значений n. Всякий раз, когда n удваивается, время выполнения такого алгоритма увеличивается на множитель четыре.

Алгоритм показывает кубическое (cubic) 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. Алгоритм Флойда –Уоршелла (динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. Разработан в 1962 году Робертом Флойдом[12] и Стивеном Уоршеллом[13]) — это алгоритм со сложностью порядка $O(n^3)$.

Алгоритм со сложностью $O(2^n)$ имеет экспоненциальную сложность (exponential complexity). 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 таблице 8.1 сравниваются значения n^2 и $n\log_2 n$. Заметьте, насколько 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 8.1

n	n^2	$n \log_2 n$
5	25	11,6
10	100	33,2
100	10000	664,3
1000	1000000	9965,7
10000	100000000	132877,1

Таблица 8.2

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	65536
32	5	160	1024	32768	4294967296
128	7	896	16384	2097152	3.4×10^{38}
1024	10	10240	1048576	1073741824	1.8×10^{308}
65536	16	1048576	4294967296	2.8×10^{14}	Избегайте!

В таблице 8.2 приводятся линейный, квадратичный, кубический, экспоненциальный и логарифмический порядки величины для выбранных значений n. Из таблицы 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. 8.3, 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. [17].

Таблица 8.3.

$Q(n)$	$Б_1$	$Б_2 = 100\ Б_1$	$Б_з = 1000\ Б_1$
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 $lgK / lg2$ единиц. Следовательно, переходя от ЭВМ с $Б_1$ = 1 Гфлопс к суперЭВМ 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 БИС, число входных двоичных переменных может составлять более 100 и поэтому полный перебор всех возможных проверяющих кодов потребует выполнения более 2^{100} вариантов моделирования схемы.

Исследования сложности алгоритмов позволили по-новому взглянуть на решение многих классических математических задач и найти для ряда таких задач (умножение многочленов и матриц, решение линейных систем уравнений и др.) решения, требующие меньше ресурсов, нежели традиционные.

ЗАКЛЮЧЕНИЕ

В основах теории алгоритмов рассматриваем задачи, для которых возможен переход от содержательной (вербальной) формулировки к алгоритмическому решению.

Алгоритм представлен в формальной записи – конечными автоматами, синтаксическим разбором, рекурсивными функциями, которые очевидно преобразуются в программы вычислительных машин. Скомпилированные по этим моделям программы ВМ не требуют верификации и, по существу, построение этих моделей является доказательством существования алгоритма.

Модели алгоритмов используются в алгоритмических языках, и возможность их реализации является признаком универсальности языка.

В практическом программировании чаще эти модели не применяются, используется процедурное, объектное и функциональное расширения, которые уже относятся к инженерии программирования.

ИМЕННОЙ УКАЗАТЕЛЬ

1. Алан Мэтисон Тьюринг (англ. Alan Mathison Turing; 23.06.1912 — 7.06.1954) — выдающийся английский математик, логик, криптограф, оказавший существенное влияние на развитие информатики. Его именем названа самая престижная премия в информатике, вручаемая Ассоциацией вычислительной техники за выдающийся научно-технический вклад в этой области — премия Тьюринга.
2. Пост Эмиль Леон (англ. Post Emil Leon; 11.02.1897 — 21.04.1954) — американский математик и логик.
3. Алонзо Чёрч (англ. Alonzo Church; 14.06.1903— 11.08.1995) — выдающийся американский математик и логик, внесший значительный вклад в основы информатики.
4. Норберт Винер (англ. Norbert Wiener; 26.11.1894 —18.03.1964)— выдающийся американский учёный, математик и философ, основоположник кибернетики и теории искусственного интеллекта.
5. Марков Андрей Андреевич (2.06.1856—20.07.1922) — русский математик, академик, внёсший большой вклад в теорию вероятностей, математический анализ и теорию чисел.
6. Цейтин Григорий Самуилович (р. 1936) — заведующий лабораторией интеллектуальных систем НИИ математики и механики СПбГУ.
7. Турчин Валентин Фёдорович (1931—2010) — советский и американский физик и кибернетик.
8. Горнер Уильям Джордж (англ. Horner William George, 1786 - 22.9.1837) - английский математик.
9. Джозеф Бернард Краскал – младший (англ. Joseph Bernard Kruskal; 29.01.1928—19.09.2010) — американский математик, статистик.
10. Эдсгер Вибе Дейкстра (нидерл. Edsger Wybe Dijkstra, 11.05.1930, — 6.08.2002) — выдающийся нидерландский учёный, идеи которого оказали огромное влияние на развитие компьютерной индустрии, один из авторов концепции структурного программирования.
11. Роберт Прим (англ. Robert Clay Prim, род. 1921) – американский математик и информатик.
12. Роберт Флойд (англ. Robert Floyd; 8.06.1936 — 25.09.2001) — американский учёный в области теории вычислительных систем. Лауреат премии Тьюринга.
13. Стивен Уоршелл (англ. Stephen Warshall; 1935 –11.12.2006) — американский учёный в области разработки операционных систем, проектирования компиляторов, языков и исследования операций.

ЛИТЕРАТУРА

1. Д. Кнут. Искусство программирования. Том 1. Основные алгоритмы. — 3-е изд. — М.: «Вильямс», 2006 —720 с.

2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т1. М:Мир, 1978 — 612 с.

3. Мендельсон Э. Введение в математическую логику М:Наука, 1976 — 320 с.

4. Википедия. http://ru.wikipedia.org/wiki/

5. Хопрокфт Дж., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. 2-е издание. - М.: Издательский дом "Вильямс", 2002 – 528 с.

6. Миллер Р. Теория переключательных схем / Р. Миллер. – М.: Наука, 1971. – Том 2: Последовательностные схемы и машины. – 304 с.

7. Стаблибайн Т. Регулярные выражения. Карманный справочник. - СПб.: Питер, 2004 —160 с.

8. Оллонгрен А. Определение языков программирования интерпретирующими автоматами. М.:Мир, 1977 — 288 с.

9. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации алгоритмов. СПб.: Наука, 2000 — 780 с.

10. Проектирование цифровых вычислительных машин / С.А. Майоров, Г.И. Новиков, О.Ф. Немолочнов и др. Под ред. С.А. Майорова. М.: Высш.шк., 1972. 344 с.

11. Гринченков Д.В., Потоцкий С.И. Математическая логика и теория алгоритмов для программистов. Учебное пособие. М: Кнорус, 2010 – 208 с.

12. Трахтенброт В. А. Алгоритмы и вычислительные автоматы. М.: Сов.радио, 1974 – 200 с.

13. Гавриков М.М., Иванченко А.Н., Гринченков Д.В. Теоретические основы разработки и реализации языков программирования, Учебное пособие. М:Кнорус, 2010 - 184 с.

14. Ахо А., Хопкровт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов, М: Мир , 1979 – 536 с.

15. Ананий В. Левитин. Алгоритмы: введение в разработку и анализ. — М.: «Вильямс», 2006 — 576 с.

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

17. Норенков И. П. Основы автоматизированного проектирования: учеб. для вузов. — 4-е изд., перераб. и доп. — М.: Изд-во МГТУ им. Н. Э. Баумана, 2009. — 430 с.

ПРИЛОЖЕНИЕ 1

Преобразование регулярных выражений в автоматы и обратно.

1. ((0+10)*11(01(0+1))*
2. (1(0 + 1)* 0(0* 10(0 +1)))*
3. (01 + 10)* (10+01)0*
4. (((0 + 1)* 10)*10(10)*11)*
5. (0+10)*1(00+10)(0+10)*
6. ((01 +1)*0(0+1)10(0 +1))*
7. 0*(1(0+1)0(1+01)*
8. (0+ 11)* 1(01*1+(01*10(0 +1))*)
9. (111*0)*10(1(0+1))*
10. (0*((1+0)1))* 1(11*01)*11*
11. ((10 +01)*(11+00)(10)*1
12. ((a+ba)*bb(ab(a+b))*
13. (b(a + b)* a(a* ba(a +b)))*
14. (ab + ba)* (ba+ab)a*
15. (((a + b)* ba)*ba(ba)*bb)*
16. (a+ba)*b(aa+ba)(a+ba)*
17. ((ab +b)*a(a+b)ba (a +b))*
18. a*(b(a+b)a(b+ab)*
19. (a+ bb)* b(ab*b+(ab*ba(a +b))*)
20. (bbb*a)*ba(b(a+b))*
21. (a*((b+a)b))* b(bb*ab)*bb*
22. ((ba +ab)*(bb+aa)(ba)*b

ПРИЛОЖЕНИЕ 2

Преобразования КА в блок-схемы.

Построить граф автомата, регулярное выражение, систему уравнений, интерпретирующих КА, заданный таблицей переходов и восстановить исходное регулярное выражение, выполнить преобразование в блок-схему и систему уравнений перехода.

1

Qi	Σ	Qj
A	0 1	B ---
B	0 1	C A
C	0 1	-- D
D	0 1	B D

2

Qi	Σ	Qj
A	a b	B ---
B	a b	C A
C	a b	-- D
D	a b	D B

3

Qi	Σ	Qj
A	0 1	B D
B	0 1	B C
C	0 1	A C
D	0 1	A --

4

Qi	Σ	Qj
A	a b	B D
B	a b	B C
C	a b	A C
D	a b	-- B

5

Qi	Σ	Qj
A	0 1	B A
B	0 1	C D
C	0 1	-- D
D	0 1	B --

6

Qi	Σ	Qj
A	a b	B A
B	a b	D C
C	a b	-- D
D	a b	B --

7

Qi	Σ	Qj
A	0 1	A B
B	0 1	C B
C	0 1	A D
D	0 1	C --

8

Qi	Σ	Qj
A	a b	A B
B	A B	C B
C	a b	D A
D	a b	-- C

9

Qi	Σ	Qj
A	0 1	B C
B	0 1	D B
C	0 1	E ---
D	0 1	A --
E	0 1	-- D

10

Qi	Σ	Qj
A	a b	B D
B	a b	C A
C	a b	E ---
D	a b	A --
E	a b	-- D

11

Qi	Σ	Qj
A	a b	B A
B	a b	B D
C	a b	-- D
D	a b	C E
E	a b	C --

12

Qi	Σ	Qj
A	0 1	B D
B	0 1	C B
C	0 1	E ---
D	0 1	A --
E	0 1	-- D

13

Qi	Σ	Qj
A	a	B
	b	A
B	a	C
	b	A
C	a	D
	b	--
D	a	B
	b	C

14

Qi	Σ	Qj
A	0	A
	1	B
B	0	C
	1	A
C	0	D
	1	--
D	0	B
	1	D

15

Qi	Σ	Qj
A	a	B
	b	--
B	a	A
	b	C
C	a	D
	b	B
D	a	D
	b	A

16

Qi	Σ	Qj
A	0	B
	1	--
B	0	A
	1	C
C	0	D
	1	B
D	0	A
	1	D

17

Qi	Σ	Qj
A	a	D
	b	B
B	A	C
	b	A
C	A	--
	b	D
D	A	B
	b	D

18

Qi	Σ	Qj
A	0	A
	1	B
B	0	C
	1	A
C	0	B
	1	D
D	0	C
	1	--

19

Qi	Σ	Qj
A	a	A
	b	B
B	a	C
	b	D
C	A	B
	B	D
D	A	C
	B	--

20

Qi	Σ	Qj
A	0	B
	1	C
B	0	C
	1	B
C	0	--
	1	D
D	0	B
	1	--

21

Qi	Σ	Qj
A	a	B
	b	C
B	a	C
	b	A
C	a	--
	b	D
D	a	B
	b	--

22

Qi	Σ	Qj
A	a	B
	b	--
B	a	A
	b	C
C	a	D
	b	B
D	a	D
	b	A

23

Qi	Σ	Qj
A	a	B
	b	C
B	a	A
	b	C
C	a	D
	b	B
D	a	D
	b	A

24

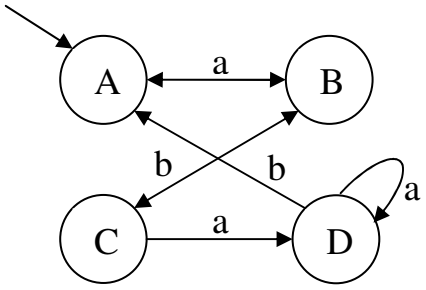
Qi	Σ	Qj
A	a	B
	b	D
B	a	C
	b	A
C	a	D
	b	--
D	a	B
	b	C

Пример

1) **Задание.**

Qi	Σ	Qj
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.

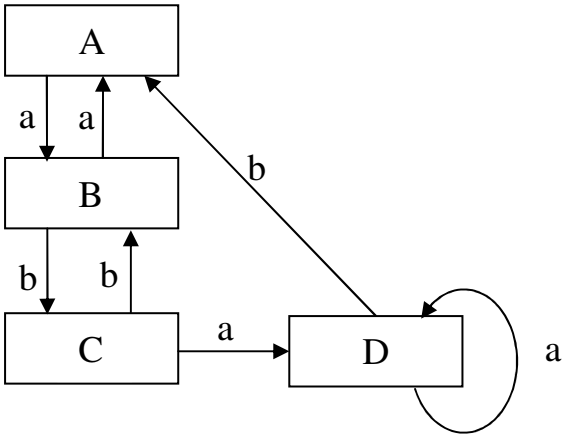
S=(((A aB)(BaA))*(AaB)((BbC)(CbB))*(BbC)(CaD)(DaD)*(DbA))*

= ((a a)* a (b b)* b a a* b)* = ((a a)* a(bb)* baa*b)*

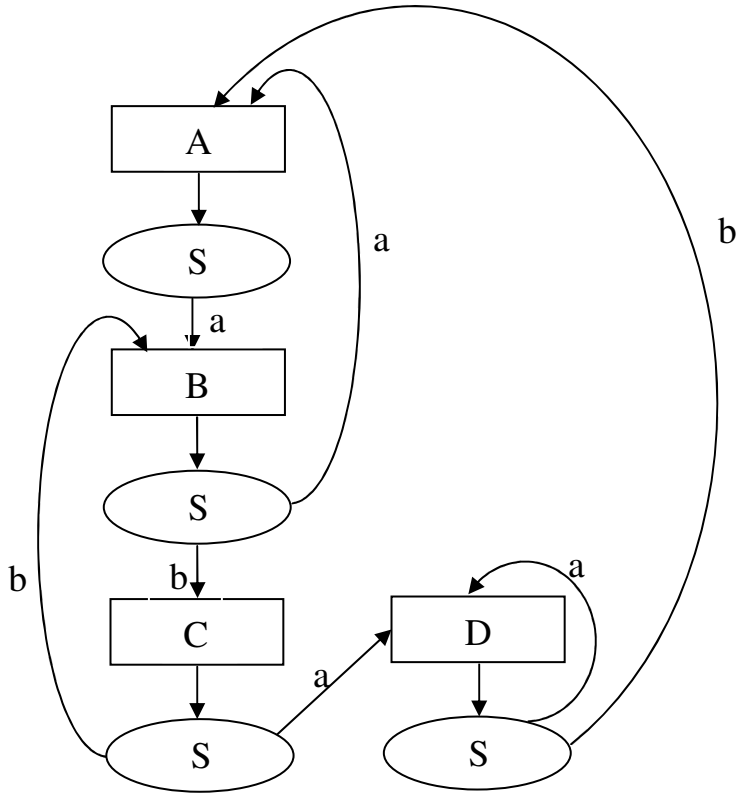
A B A B C B C D D A

4) Блок-схема.

а) состояние представляем операторной вершиной



б) ребра графа представляем переключателем **(Switch)**,



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

c) 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 состояние E, контролирующее ошибку.

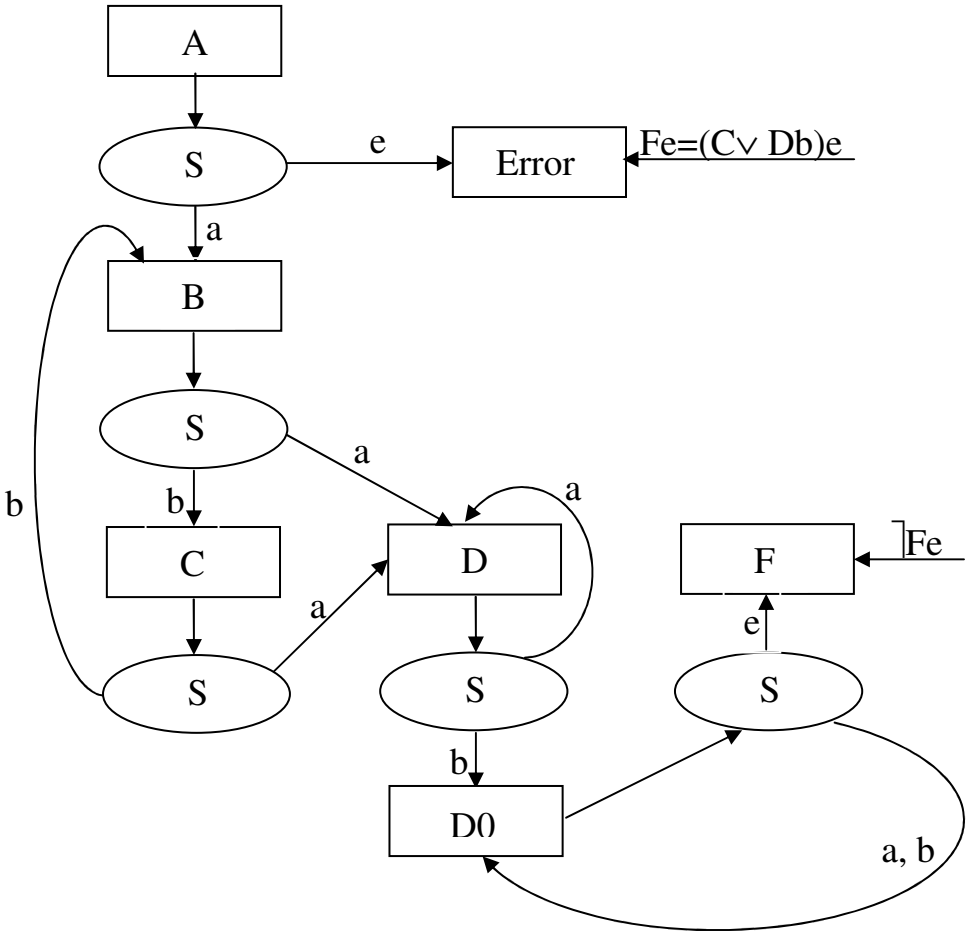
В конце непустой цепочки символов всегда (be) – в данном случае, это условия Ce или Dbe, а все другие условия - ошибочные

В данном случае можно определить правильное завершение распознавания и, соответственно, контролировать логику выполнения алгоритма уравнением

$$\mathbf{Fe= Ce \vee Dbe=(C \vee Db)e.}$$

При ошибке в цепочке Ee= ⌉Fe.

Для вычисления Fe можно добавить состояния (Fe) и операторную вершину в блок-схеме, что подразумевает недетерминированность при бесконечно больших цепочках. Команда (Fe) 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.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра вычислительной техники СПбГУ ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России. Заведующими кафедрой в разное время были выдающиеся деятели науки и техники М.Ф. Маликов, С.А. Изенбек, С.А. Майоров, Г.И. Новиков. Многие поколения студентов и инженеров в Советском Союзе и за его пределами изучали вычислительную технику по учебникам С.А. Майорова и Г.И. Новикова, О.Ф. Немолочного, С.И. Баранова, В.В. Кириллова и А.А. Приблуды, Б.Д.Тимченко и др.

Основные направления учебной и научной деятельности кафедры в настоящее время включают в себя встроенные управляющие и вычислительные системы на базе микропроцессорной техники, информационные системы и базы данных, сети и телекоммуникации, моделирование вычислительных систем и процессов, обработка сигналов, защита информации и информационная безопасность..

Выпускники кафедры успешно работают не только в разных регионах России, но и во многих странах мира: Австралии, Германии, США, Канаде, Германии, Индии, Китае, Монголии, Польше, Болгарии, Кубе, Израиле, Камеруне, Нигерии, Иордании и др. Выпускник, аспирант и докторант кафедры ВТ Аскар Акаев был первым президентом Киргизии.

Владимир Иванович Поляков
Владимир Иванович Скорубский

Основы теории алгоритмов

Учебное пособие по дисциплине
«Математическая логика и теория алгоритмов»

В авторской редакции
Дизайн В.И. Поляков
Верстка В.И. Поляков
Редакционно-издательский отдел Санкт-Петербургского национального исследовательского университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ №
Тираж 250 экз.
Отпечатано на ризографе