

Finite Automata

Translated and adapted by L. Breveglieri

FINITE STATE AUTOMATA AND HOW TO RECOGNIZE A REGULAR LANGUAGE

PROBLEM: recognizing whether a string belongs to a given language, before elaborating it in some way (i.e., before doing semantic analysis)

In order to describe a string recognition procedure, abstract machines (of various kinds) called AUTOMATA are used

RECOGNITION ALGORITHM

DOMAIN: a set of strings over the alphabet Σ

IMAGE: the answer *yes* or *no* (recognition is a decision problem)

Applying the recognition algorithm α to the string x is indicated as $\alpha(x)$.

The string x is *recognized (accepted)* or *unrecognized (rejected)*

by α if $\alpha(x) = \text{yes}$ or $\alpha(x) = \text{no}$, respectively

The language $L(\alpha)$ is defined as

$$L(\alpha) = \{ x \mid x \in \Sigma^* \wedge \alpha(x) = \text{yes} \}$$

If the language L is semidecidable (= recursively denumerable, but not recursive), it may happen that for some uncorrect string x the algorithm α will not terminate, i.e., $\alpha(x)$ is undefined

Here only decidable languages are considered, which have a recognition algorithm that terminates for every string, no matter if accepted or rejected

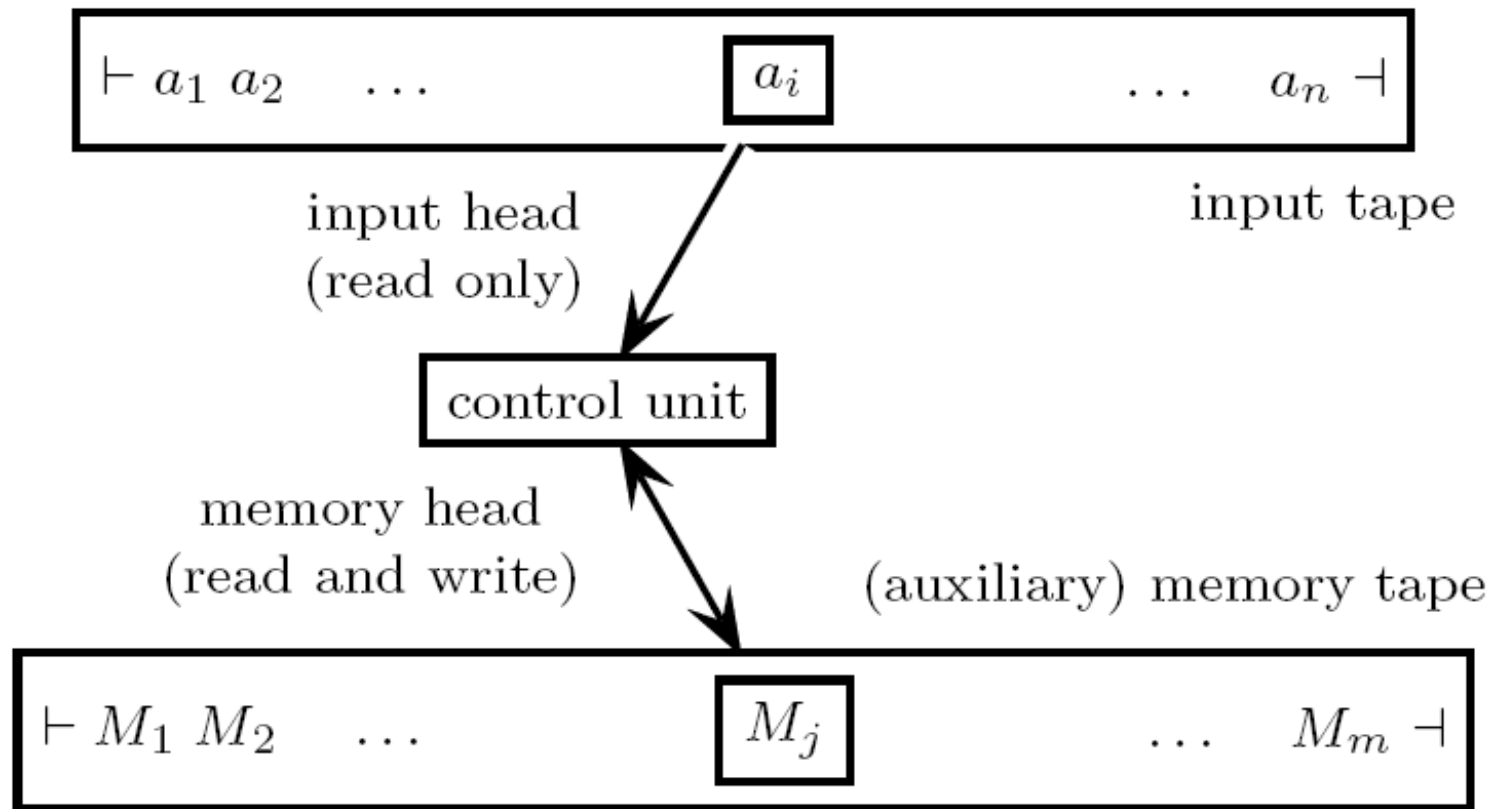
Almost all the problems of some interest here, have solutions with a relatively low time complexity degree (= the number of the computation steps executed by the recognition algorithm), i.e., a linear or at worst a polynomial time complexity, with respect to the size of the problem, which in most cases is the length of the string

In the theory and practice of formal languages, one computation step is a single atomic operation of the abstract recognition machine (automaton), which can manipulate only one symbol at a time. Therefore it is customary to present the recognition algorithm by means of an automaton of some kind, no matter if a recognizer or a transducer machine, mainly for the following reasons

1. outlining the correspondence between the various families of languages and the respective generative devices, i.e., their grammars
2. skipping any unnecessary and premature reference to the effective implementation of the algorithm in some programming language

Once the recognizer automaton has been designed, it is relatively easy to write the corresponding program, in C or similar programming languages

GENERAL MODEL OF A RECOGNIZER AUTOMATON



THE AUTOMATON ANALYZES THE INPUT STRING and executes a series of moves. Each move depends on the symbols currently pointed by the heads and also on the current state. The move may have the following effects

1. shifting the input head of one position to the left or right
2. replacing the current memory symbol by a new one and shifting the memory head of one position to the left or right
3. changing the current state

Some of the operations listed above may be absent

ONE-WAY AUTOMATON: the input head can be shifted only to the right. In the following only this case is considered. It corresponds to scanning the input string only once from left to right

NO AUXILIARY MEMORY: this is the celebrated FINITE STATE AUTOMATON. It is the well known machine model that recognizes the regular languages

AUXILIARY MEMORY: structured and handled as a PUSHDOWN STACK. This is the well known machine model that recognizes the free languages

A CONFIGURATION (instantaneous) is the set of the three components that determine the behaviour of the automaton

- the still unread part of the input tape

- the contents and position of the memory tape and head, respectively

- the current state of the control unit

INITIAL CONFIGURATION: the input head is positioned on the symbol immediately following the start-marker, the control unit is in a specific state (initial state), and the memory tape only contains a special initial symbol. The automaton configuration changes through a series of transitions, each of which is driven by a move. The whole series of transitions is the computation of the automaton

DETERMINISTIC BEHAVIOUR: in every instantaneous configuration, at most one move is possible (or none). Otherwise (i.e., there are two or more moves), the automaton is said to be non-deterministic (or indeterministic)

FINAL CONFIGURATION: the control unit is in a special state qualified as final and the input head is positioned on the end-marker of the string to be recognized. Sometimes the final configuration is characterized by a condition for the memory tape: to be empty or to contain only one special final symbol

A SOURCE STRING x IS ACCEPTED BY THE AUTOMATON if it starts from the initial input configuration $\vdash x \vdash$, executes a series of transitions (moves) and reaches a final configuration. If the automaton is non-deterministic, it may reach the same final configuration in two or more different sequences of transitions, or it may even reach two or more different final configurations

THE COMPUTATION terminates either because the automaton has reached a final configuration or because it cannot execute any more transition steps, due to the fact that in the current instantaneous configuration there is not any possible move left. In the former case the input string is accepted, in the latter case it is rejected

THE SET OF ALL THE STRINGS ACCEPTED BY THE AUTOMATON constitutes the language RECOGNIZED (or ACCEPTED or DEFINED) BY THE AUTOMATON

Two automata that accept the same language are said to be EQUIVALENT. CAUTION: two equivalent automata may be modeled differently and may not have the same computational complexity

REGULAR LANGUAGES, which are recognized by finite state automata, are a sub-family of the languages recognizable in real time by a TURING MACHINE

FREE LANGUAGES are a subfamily of the languages recognized by a TURING MACHINE that have a polynomial time complexity

FINITE STATE AUTOMATA (or simply FINITE AUTOMATA)

Many applications of computer science and engineering make use of finite state automata: digital design, theory of control, communication protocols, the study of system reliability and security, etc

STATE-TRANSITION GRAPH

A finite state automaton consists of the following three elements

- the input tape, which contains the input string $x \in \Sigma^*$

- the control unit and its finite memory, which contains the state table

- the input head, initially positioned at the start-marker of string x , which is shifted to the right at every move, as far as it reaches the end-marker of string x or an error happens

After reading an input character, the automaton updates the current state of the control unit

After scanning the input string x , the automaton recognizes string x or rejects it, depending on the current state

STATE-TRANSITION GRAPH (continued)

STATE-TRANSITION GRAPH: is a directed graph that represents the automaton and consists of the following elements

NODES: represent the states of the control unit

ARCS: represent the moves of the automaton

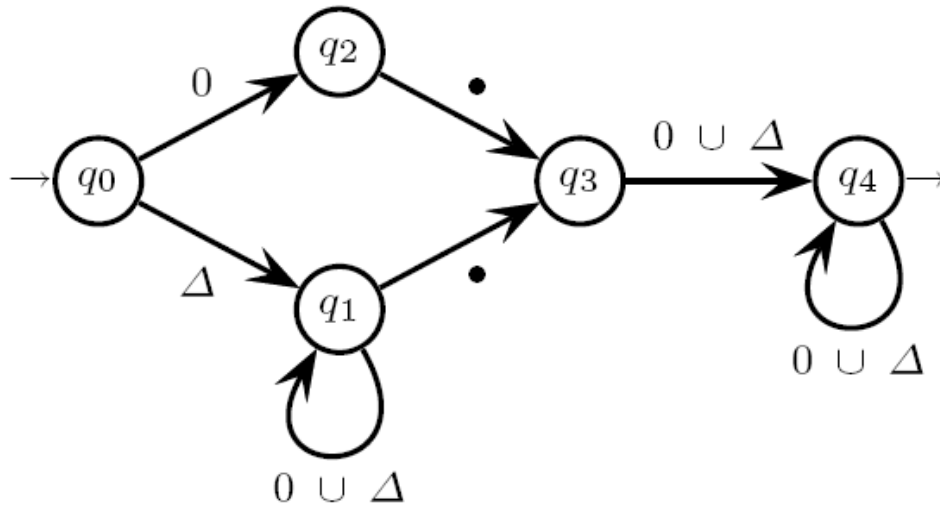
Each arc is labeled by an input symbol, and it represents the move enabled when the current state matches the source state of the arc and the current input symbol matches the arc label

The state-transition graph has a unique INITIAL STATE (if it had two or more, then it would be non-deterministic), but it may have none, one or more FINAL STATES (if it does not have any final states, then it recognizes the empty set)

The graph can be represented in the form of an INCIDENCE MATRIX. Each matrix entry is indexed by the current state and by the input symbol, and contains the next state. Such an incidence matrix is often called STATE TABLE

EXAMPLE – decimal constants in numerical form

state-transition diagram



state-transition table

<i>current state</i>	<i>current character</i>				
	0	1	...	9	•
$\rightarrow q_0$	q_2	q_1	...	q_1	—
q_1	q_1	q_1	...	q_1	q_3
q_2	—	—	...	—	q_3
q_3	q_4	q_4	...	q_4	—
$q_4 \rightarrow$	q_4	q_4	...	q_4	—

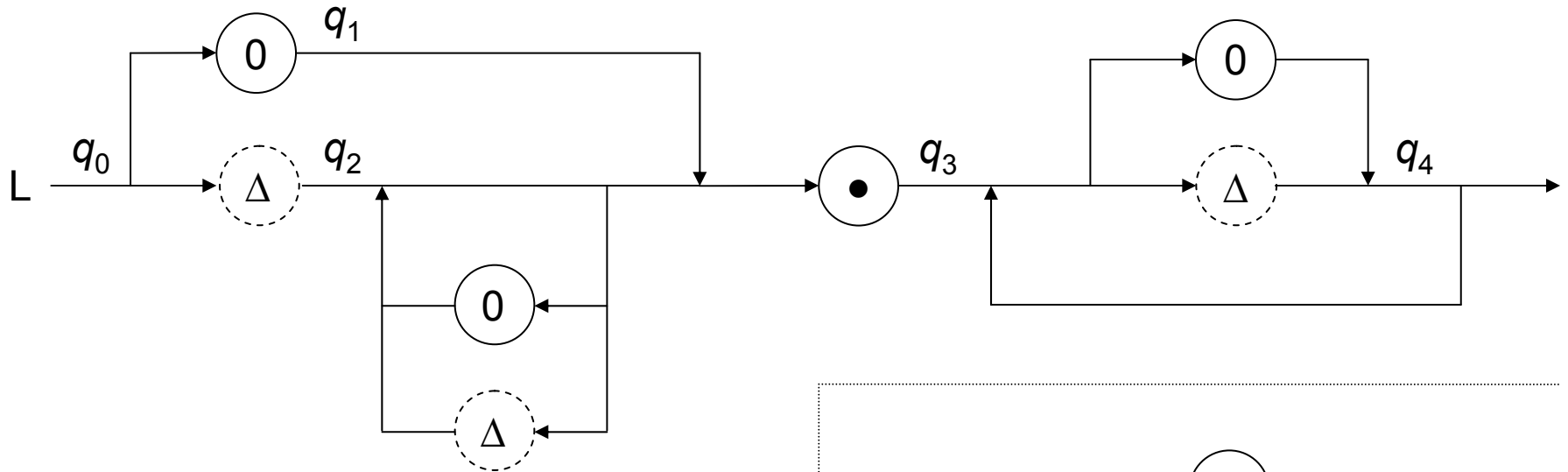
$$\Sigma = \Delta \cup \{0, \bullet\}$$

$$\Delta = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$e = (0 \cup \Delta (0 \cup \Delta)^*) \bullet (0 \cup \Delta)^+$$

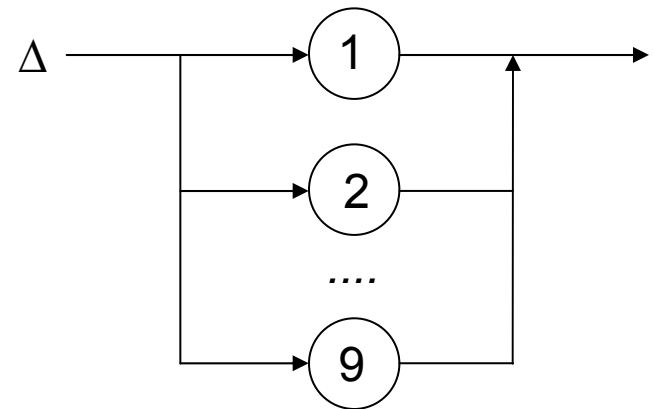
NOTE: in the drawing, a few topologically identical arcs are grouped into one (in fact from q_0 to q_1 nine arcs should be drawn)

Here follow the SYNTAX DIAGRAMS of the automaton, similar to those that can be used to represent the *EBNF* rules. They are the DUAL FORM of the state-transition graph of the automaton



syntax diagram

DUAL FORM: transform
nodes into vertices and
vertices into nodes



syntax diagram

DETERMINISTIC FINITE STATE AUTOMATON

A DETERMINISTIC FINITE AUTOMATON M consists of five elements

Q	the <i>set of states</i> (finite and not empty)
Σ	the <i>input alphabet</i> (or terminal alphabet)
$\delta: (Q \times \Sigma) \rightarrow Q$	the <i>transition function</i>
$q_0 \in Q$	the <i>initial state</i>
$F \subseteq Q$	the <i>set of final states</i> (it may be empty)

The transition function encodes the automaton moves:

$$\delta(q_i, a) = q_j$$

$$x = ab \quad \delta(q_0, a) = q_1 \quad \delta(q_1, b) = q_2$$

$$\delta(q_0, ab) = q_2$$

$$\forall q \in Q: \delta(q, \varepsilon) = q$$

When M is in the current state q_i and reads the input symbol a , it switches the current state to q_j

If $\delta(q_i, a)$ is undefined, then M stops (it enters an error state and rejects the input string)

the domain is $Q \times \Sigma^*$ and the transition function is

$$\delta(q, ya) = \delta(\delta(q, y), a) \quad \text{where } a \in \Sigma \text{ and } y \in \Sigma^*$$

EXECUTION OF A COMPUTATION (= sequence of TRANSITIONS)

The transition function δ is extended by posing as aside if, and only if, there exists a path from state q to state q' , labeled by the input string y . If the automaton moves through such a path, it recognizes the string y . The automaton executes a sequence of transitions, i.e., a computation

$$\delta(q, y) = q'$$

RECOGNITION OF A STRING

A string x is recognized (or accepted) if, and only if, when the automaton moves through a path labeled by x , it starts from the initial state and ends at one of the final states

$$\delta(q_0, x) \in F$$

The empty string is accepted if, and only if, the initial state is final as well

THE LANGUAGE RECOGNIZED BY THE AUTOMATON M

$$L(M) = \{ x \in \Sigma^* \mid x \text{ is recognized by } M \}$$

The family of the languages recognized by finite state automata is named family of *finite state languages*

Two finite automata are equivalent if they recognize the same language

The time complexity of finite state automata is optimal: the input string x is accepted or rejected in real-time, i.e., in a number of moves equal to the string length. Since it takes exactly as many steps to scan the string from left to right, the recognition time complexity could not be lower than this

EXAMPLE – decimal constants in numerical form (continued)

The automaton M is defined as follows

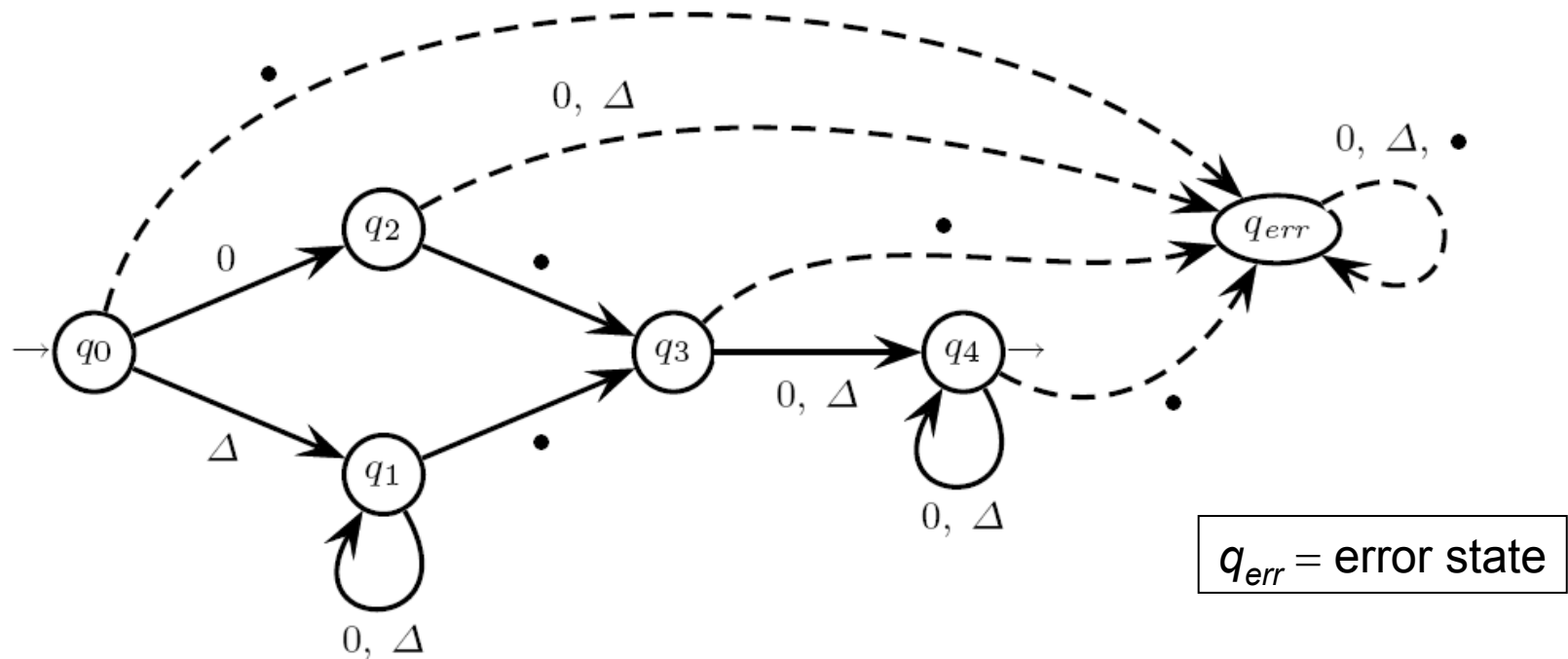
$Q = \{ q_0, q_1, q_2, q_3, q_4 \}$	state set
$\Sigma = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \bullet \}$	alphabet
$q_0 = q_0$	initial state
$F = \{ q_4 \}$	final state set

$$\begin{aligned}\delta(q_0, 3 \bullet 1) &= \delta(\delta(q_0, 3 \bullet), 1) \\ &= \delta(\delta(\delta(q_0, 3), \bullet), 1) \\ &= \delta(\delta(q_1, \bullet), 1) \\ &= \delta(q_3, 1) = q_4\end{aligned}$$

acceptance

Since it holds $q_4 \in F$, string $3 \bullet 1$ is accepted. On the contrary, since state $\delta(q_0, 3 \bullet) = q_3$ is not final, string $3 \bullet$ is rejected, as well as string $0 2$ because function $\delta(q_0, 0 2) = \delta(\delta(q_0, 0), 2) = \delta(q_2, 2)$ is undefined.

ERROR STATE AND NATURAL COMPLETION OF THE AUTOMATON



$\forall q \in Q \forall a \in \Sigma$ if $\delta(q, a)$ is undefined then set $\delta(q, a) = q_{err}$
 $\forall a \in \Sigma$ set $\delta(q_{err}, a) = q_{err}$

It is always possible to complete the deterministic automaton by adding the error state, without changing the accepted language

AUTOMATON IN CLEAN FORM (REDUCED FORM)

An automaton may contain useless parts, i.e., states, which do not give any contribution to the recognition process and which usually can be eliminated

A state q is said to be ACCESSIBLE (or REACHABLE) FROM STATE p if there is a computation that moves the automaton from state p to state q

A state q is said to be ACCESSIBLE (or REACHABLE) if it can be reached from the initial state

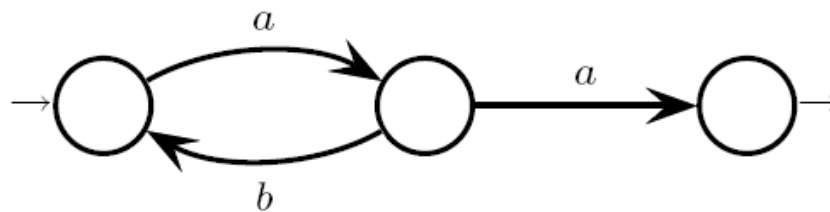
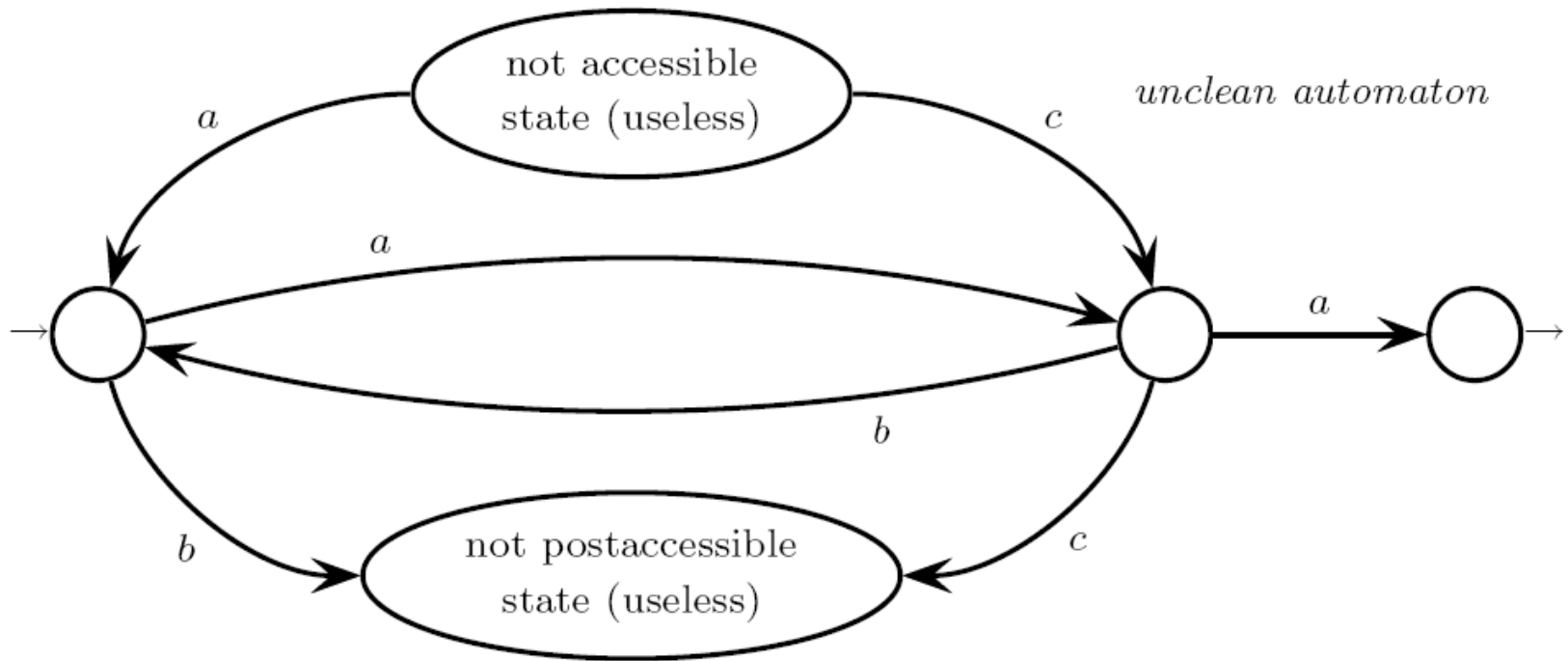
A state q said to be POSTACCESSIBLE (or DEFINED) if some final state can be reached from state q

A state q is said to be USEFUL if it is both accessible and postaccessible, i.e., if it is placed on a path that connects the initial state to a final state

An automaton A is said to be in CLEAN FORM (or in REDUCED FORM) if every state of A is useful, i.e., both accessible and postaccessible

PROPERTY – Every finite state automaton has an equivalent clean form.
To reduce an automaton: first identify all the useless states, then strip them off the automaton along with all their incoming and outgoing arcs

EXAMPLE – elimination of the useless states



clean automaton

CAUTION: it may
not be minimal yet !

MINIMAL AUTOMATON

PROPERTY – For every finite state language there exists one, and only one, deterministic finite state recognizer that has the smallest possible number of states, which is called the *minimal automaton*

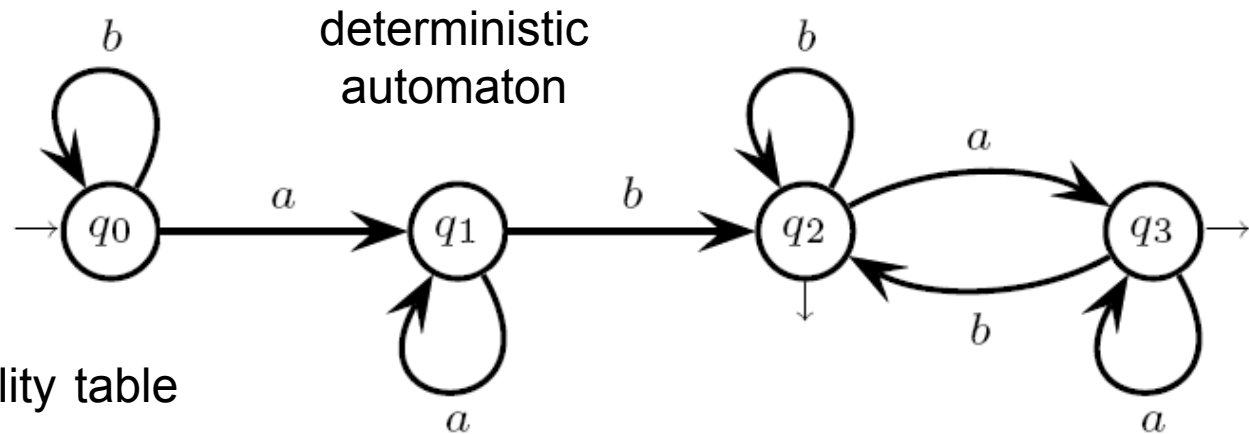
UNDISTINGUISHABLE STATE – A state p is UNDISTINGUISHABLE FROM A STATE q if, and only if, for every input string x either both the next states $\delta(p, x)$ and $\delta(q, x)$ are final, or neither one is; equivalently, if one starts from p and q , scans the string x , but does not reach a final state in either case

Two undistinguishable states can be MERGED and thus the number of states of the automaton can be reduced, without changing the recognized language

UNDISTINGUISHABILITY is a binary relation, and is reflexive, symmetric and transitive, therefore it is an *equivalence* relation

p is DISTINGUISHABLE from q if and only if (1) or (2) hold	1) p is final and q is not (or viceversa) 2) $\delta(p, a)$ is distinguishable from $\delta(q, a)$
---	---

EXAMPLE



Undistinguishability table

q_1	$(1, 1)$ $(0, 2)$		
q_2	×	×	
q_3	×	×	$(3, 3)$ $(2, 2)$
	q_0	q_1	q_2

undistinguishable state pair: $[q_2, q_3]$

q_1	×		
q_2	×	×	
q_3	×	×	$(3, 3)$ $(2, 2)$
	q_0	q_1	q_2

distinguishable state pair: $[q_0, q_1]$

Equivalence classes of the undistinguishability relation: $[q_0]$, $[q_1]$ and $[q_2, q_3]$

REDUCTION OF THE NUMBER OF STATES (MINIMIZATION)

The equivalence classes of the undistinguishability relation of the original automaton M are those of the minimal automaton M' equivalent to M .

To define the transition function of M' , it suffices to state that there is an arc from class $C_1 = [\dots, p_r, \dots]$ to class $C_2 = [\dots, q_s, \dots]$ if and only if in M there is an arc from state p_r to state q_s (with the same label)

$$p_r \xrightarrow{b} q_s$$

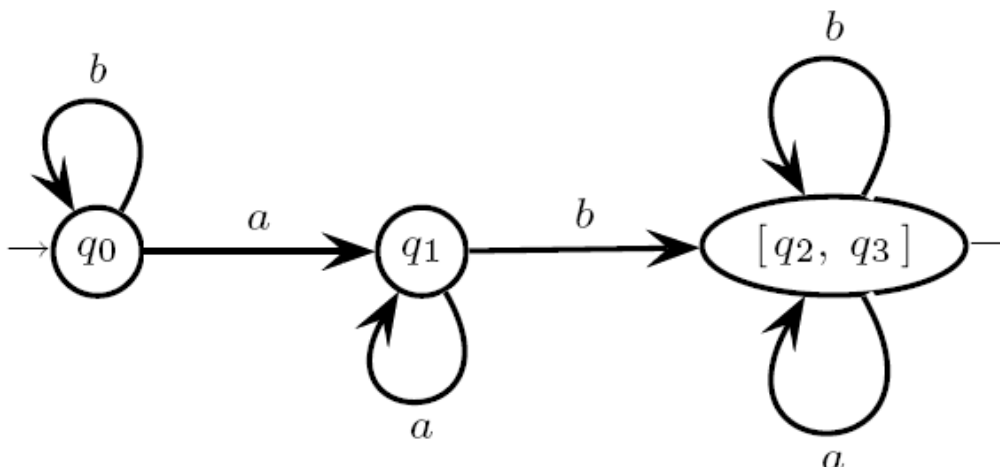
$$\overbrace{[\dots p_r \dots]}^{C_1} \xrightarrow{b} \overbrace{[\dots q_s \dots]}^{C_2}$$

That is, there is an arc between two states belonging to the two classes.

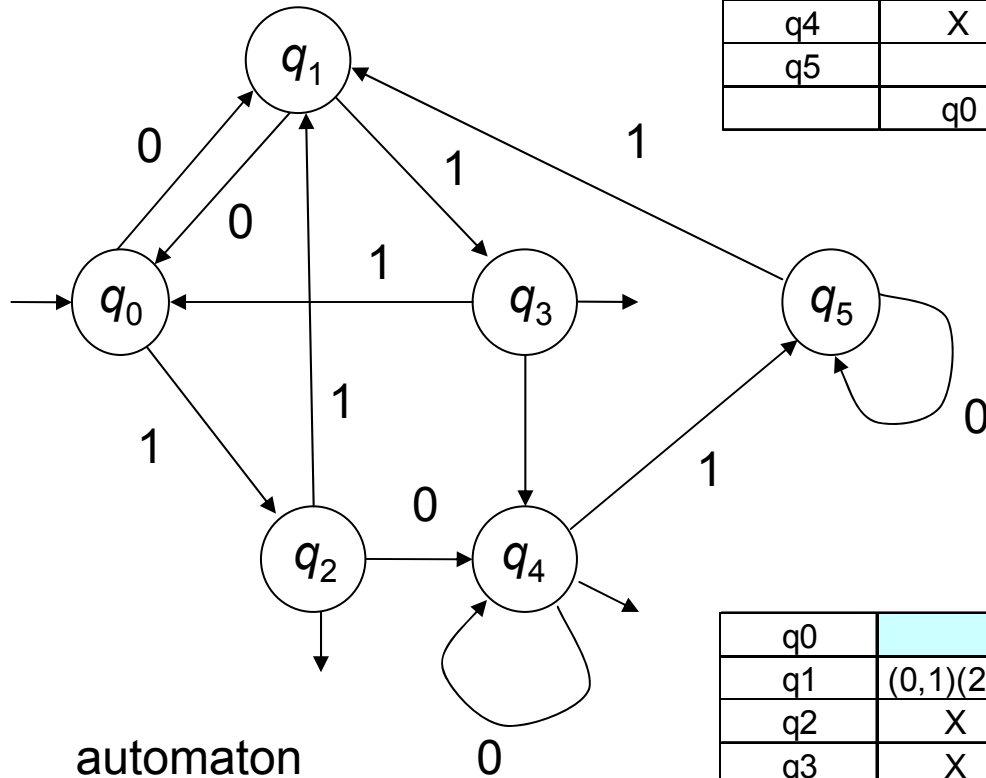
CAUTION: the same arc of M' may originate from two or more arcs of M

EXAMPLE (continued)

minimal automaton



EXAMPLE (with six states)



automaton

q_0						
q_1						
q_2	X	X				
q_3	X	X				
q_4	X	X				
q_5			X	X	X	
	q_0	q_1	q_2	q_3	q_4	q_5

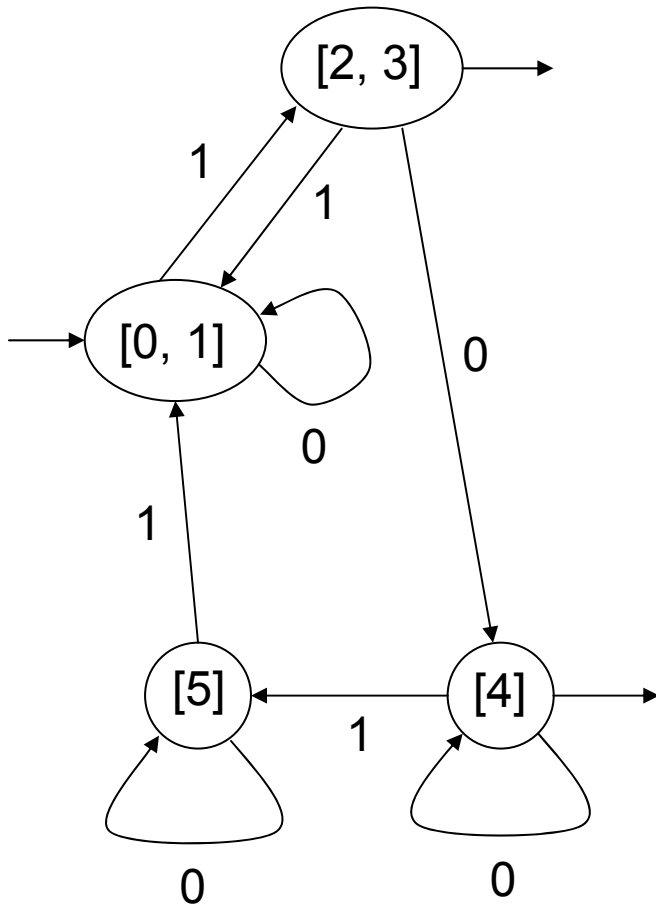
undistinguishability table

undistinguishability table

q_0						
q_1	(0,1)(2,3)					
q_2	X	X				
q_3	X	X	(4,4)(0,1)			
q_4	X	X	(4,4)(1,5)	(4,4)(0,5)		
q_5	(1,5)(2,1)	(0,5)(3,1)	X	X	X	
	q_0	q_1	q_2	q_3	q_4	q_5

EXAMPLE (continued)

minimal automaton



undistinguishability table

q0						
q1						
q2	X	X				
q3	X	X				
q4	X	X	X	X		
q5	X	X	X	X	X	
	q0	q1	q2	q3	q4	q5

equivalence classes of M
 $[q_0, q_1]$, $[q_2, q_3]$, $[q_4]$ and $[q_5]$

NON-DETERMINISTIC AUTOMATA (or INDETERMINISTIC)

A right linear grammar may contain alternative rules, i.e., rules from the same state and with the same label, which cause a non-deterministic behaviour

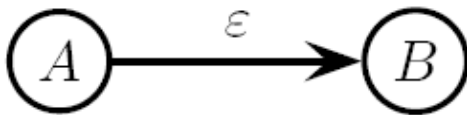


$A \rightarrow a B \mid a C$
where $a \in \Sigma$ and $A, B, C \in V$

$$\delta(A, a) = \{B, C\}$$

the transition function δ of a non-deterministic automaton may have a state set as value (not just a single state)

A non-deterministic behaviour may also originate from a spontaneous move (or ε -move), which is executed without reading any input symbol



$A \rightarrow B$ where $B \in V$

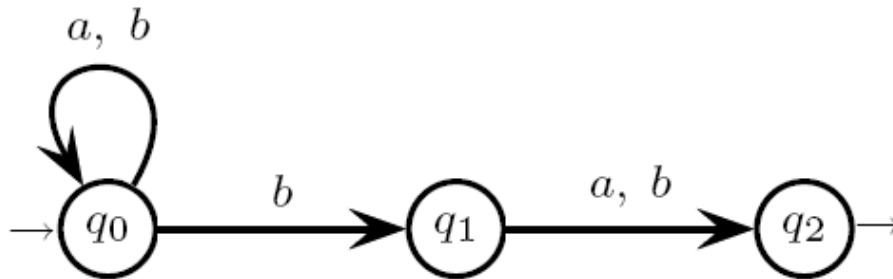


non-deterministic automaton
with a spontaneous move

FOUR MOTIVATIONS FOR INDETERMINISM

- 1) The correspondence between grammars and automata suggests to have
 - a) moves with two or more destination states
 - b) spontaneous moves (or ε -moves)
 - c) two or more initial states, i.e., the grammar has two or more axiomsPoints (a) and (b) are the two main causes of non-determinism in automata
- 2) Concision: defining a language by means of a non-deterministic automaton may be more readable and compact than using a deterministic one

EXAMPLE (all the strings must have a second last character = b)



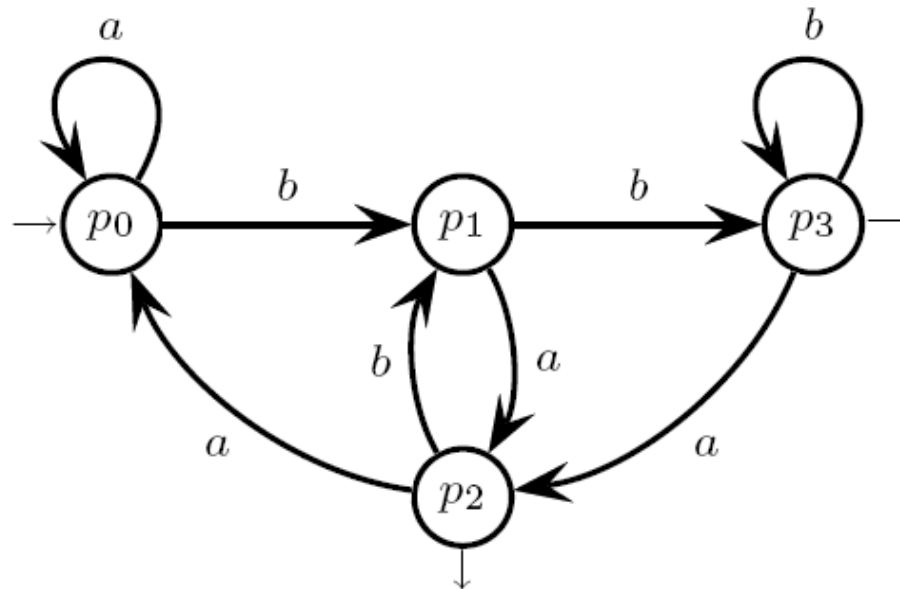
equivalent regexp

$$L = (a \mid b)^* b (a \mid b)$$

Processing string “ $b a b a$ ”. The top computation recognizes the string, the bottom one does not, as it does not go to a final state

$$\begin{array}{ccccccc} & b & & a & & b & & a \\ q_0 & \xrightarrow{\quad} & q_0 & \xrightarrow{\quad} & q_0 & \xrightarrow{\quad} & q_1 & \xrightarrow{\quad} & q_2 \\ & b & & a & & b & & a \\ q_0 & \xrightarrow{\quad} & q_0 & \xrightarrow{\quad} & q_0 & \xrightarrow{\quad} & q_0 & \xrightarrow{\quad} & q_0 \end{array}$$

The same language is accepted by the non-deterministic automaton below, which does not make as evident that the second last character of the strings has to be b



EXERCISE

If we generalize the example and we require that the k -th last element ($k \geq 2$) of the accepted string be b , then the non-deterministic automaton has $k + 1$ states, while it can be proved that the minimum number of states of a deterministic one that recognizes the same language is an exponential function of k

Allowing non-determinism may make some language definitions more coincide

NON-DETERMINISTIC RECOGNITION

At the moment ignore the existence of spontaneous moves. Here is the formal definition of non-deterministic automaton and computation

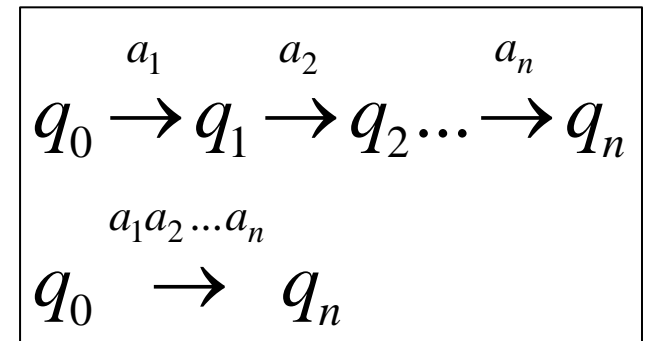
A *non-deterministic finite state automaton* N without spontaneous moves has

1. a finite set of states Q
2. a finite input (or terminal) alphabet Σ
3. two subsets of Q , namely
 - a) the set I of initial states
 - b) the set F of final states
4. a set δ of transitions, which is a subset of the set product $Q \times \Sigma \times Q$

A computation of length n originates at state q_0 and ends at state q_n , and has labeling $a_1 a_2 \dots a_n$

An input string x is accepted (recognized) by the automaton if it is the labeling of a path that starts from an initial state and ends to a final state

Language recognized by a non-det. automaton N

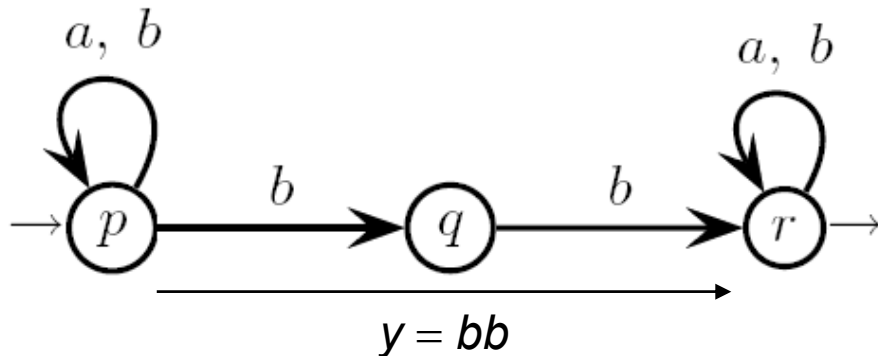


$$L(N) = \left\{ x \in \Sigma^* \mid q \xrightarrow{x} r \text{ with } q \in I \text{ and } r \in F \right\}$$

EXAMPLE – searching a word in a text

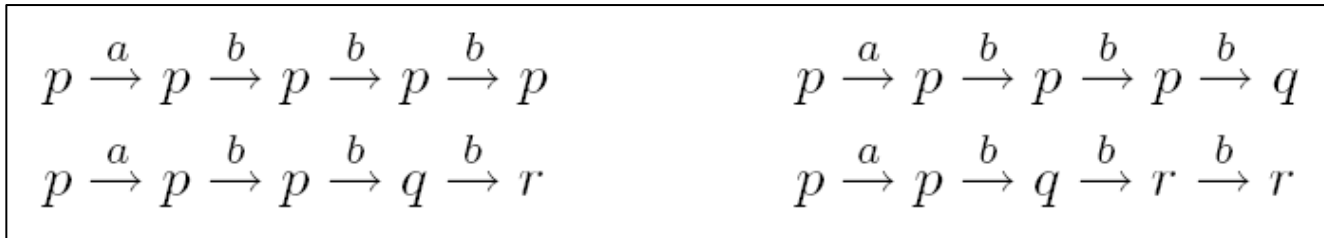
Given a word y , to recognize if a text contains that word, submit the text to the finite automaton that accepts the (regular) language $(a \mid b)^* y (a \mid b)^*$

For instance suppose to search the word $y = bb$



the automaton is
non-deterministic
in the state p

The string “ $a \ b \ b \ b$ ” labels several computations that start from the initial state



The two top computations do not find the searched word. The two bottom ones find it in these two positions, respectively

$a \ b \ \underbrace{b \ b}_y$ and $a \ \underbrace{b \ b}_y \ b$

TRANSITION FUNCTION

The moves of the non-deterministic automaton can be defined by means of a many-valued transition function

Given the non-deterministic automaton $N = (Q, \Sigma, \delta, I, F)$ without spontaneous moves, the transition function δ is defined as to have the domain and image

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(Q)$$

For any terminal char. a $\delta(q, a) = \{p_1, p_2, \dots, p_k\}$

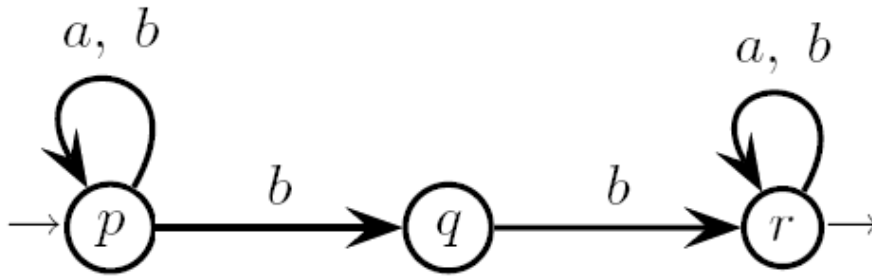
For the empty string ε $\forall q \in Q \quad \delta(q, \varepsilon) = \{q\}$

For any terminal string y $\forall q \in Q \quad \forall y \in \Sigma^* \quad \delta(q, y) = \{p \mid q \xrightarrow{y} p\}$

Here is the language accepted by N : after scanning a string x , if the current state set contains at least one final state, then the string x is accepted

$$L(N) = \{x \in \Sigma^* \mid \exists q \in I \quad \delta(q, x) \cap F \neq \emptyset\}$$

EXAMPLE – searching a word in a text (continued)



$$\begin{aligned}\delta(p, a) &= [p], \\ \delta(p, ab) &= [p, q], \\ \delta(p, abb) &= [p, q, r]\end{aligned}$$

AUTOMATA WITH SPONTANEOUS TRANSITIONS (ε -MOVES)

Spontaneous moves (or ε -moves) are represented by means of arcs labeled with the metasymbol ε (called ε -arcs) in the state-transition graph

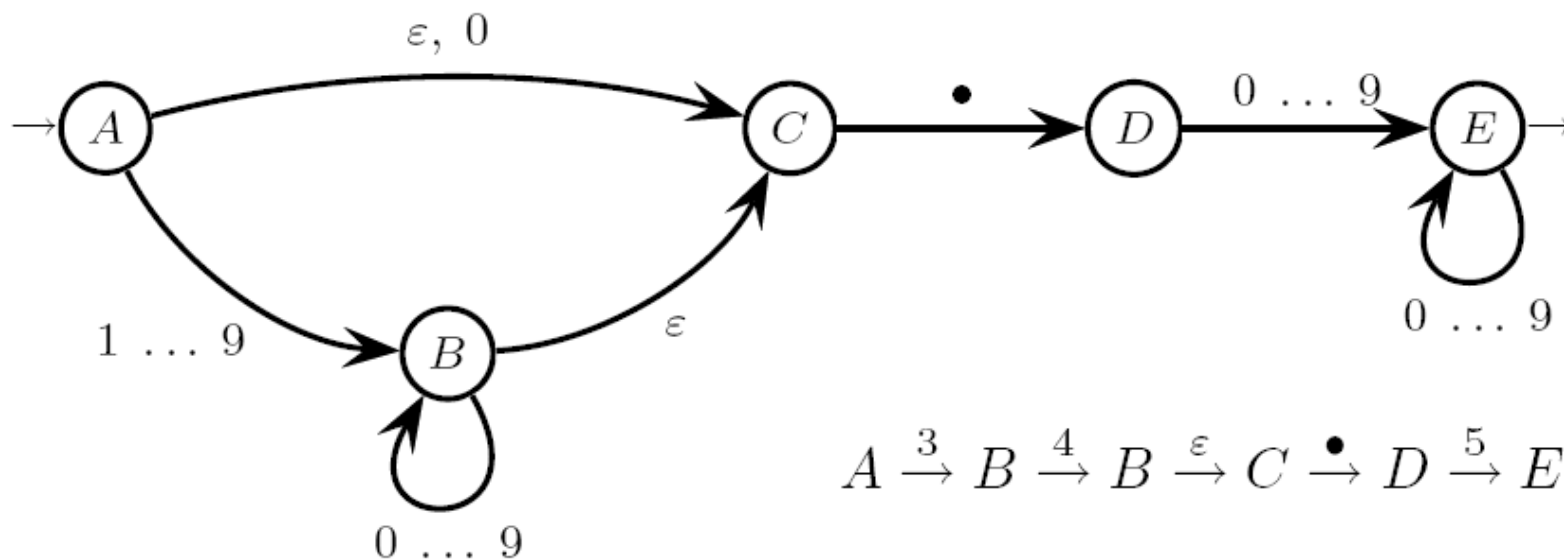
Using ε -arcs helps the modular construction of finite automata and thus allows us to reuse other automata or some parts thereof

EXAMPLE – decimal constant, where union and concatenation are used

$$L = (\varepsilon \mid 0 \mid N) \bullet (0 \dots 9)^+$$

$$\text{where } N = (1 \dots 9) (0 \dots 9)^*$$

non-deterministic
automaton
(with ε -arcs)



The spontaneous moves do not change the definition of string recognition. But the computation may be longer than the string to recognize. For instance the string “3 4 • 5” (of length 4) is accepted by the above computation of 5 steps

UNIQUENESS OF THE INITIAL STATE. A non-deterministic automaton may have two or more initial states. But it is easy to construct an equivalent non-deterministic automaton with only one initial state. Add a new initial state q_0 , connect it to the existing initial states by ε -arcs, make such states non-initial and leave only q_0

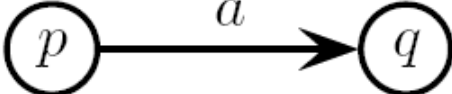
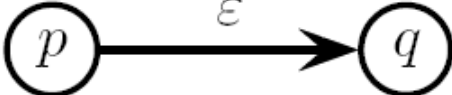
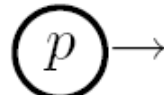
We see soon that a computation of the new automaton accepts a string if, and only if, the old automaton accepts it as well. The additional ε -arcs can be eliminated, as it will be shown next

CORRESPONDENCE BETWEEN AUTOMATON AND GRAMMAR

Suppose that

$G = (V, \Sigma, P, S)$ is a strictly right linear grammar

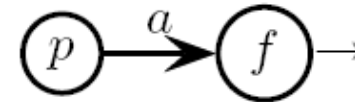
$N = (Q, \Sigma, \delta, q_0, F)$ is the automaton (suppose the initial state is unique)

#	<i>right-linear grammar</i>	<i>finite automaton</i>
1	nonterminal set $V = Q$	set of states $Q = V$
2	axiom $S = q_0$	initial state $q_0 = S$
3	$p \rightarrow a q$ where $a \in \Sigma$ and $p, q \in V$	
4	$p \rightarrow q$ where $p, q \in V$	
5	$p \rightarrow \varepsilon$	final state 

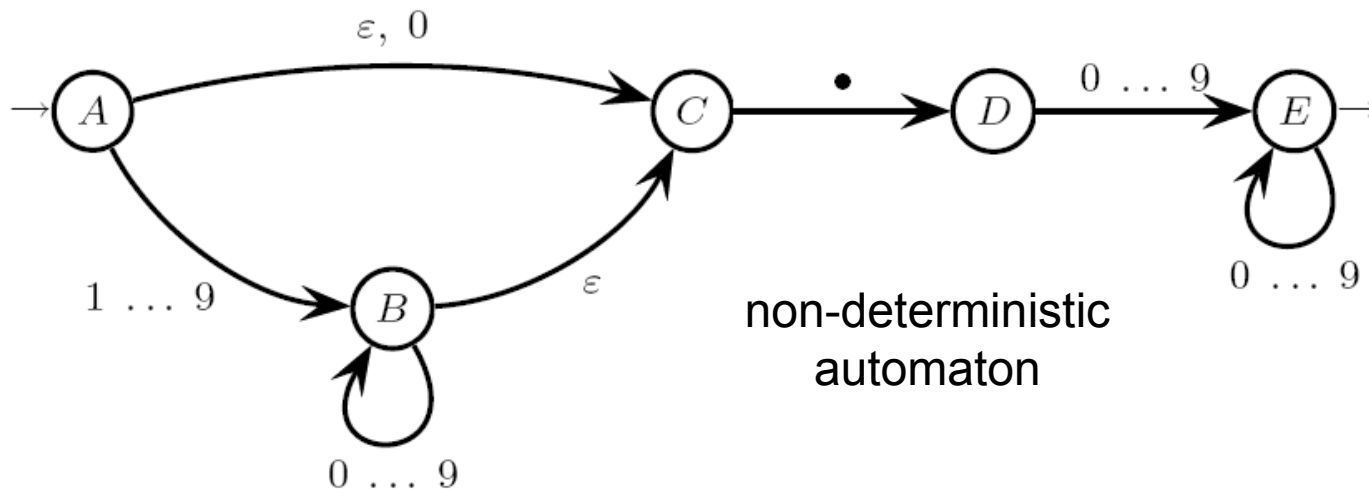
A grammar derivation corresponds to an automaton computation, and viceversa. Consequently the two models define the same language

A LANGUAGE IS GENERATED BY A RIGHT LINEAR GRAMMAR IF AND ONLY IF IT IS RECOGNIZED BY A FINITE AUTOMATON (same for a left linear grammar)

If the (right linear) grammar also has terminal rules of the type $p \rightarrow a$ with $a \in \Sigma$, then the automaton has a final state f in addition to those that correspond to the ε -rules of the grammar, and also has this move



EXAMPLE – equivalence between right linear grammar and automaton

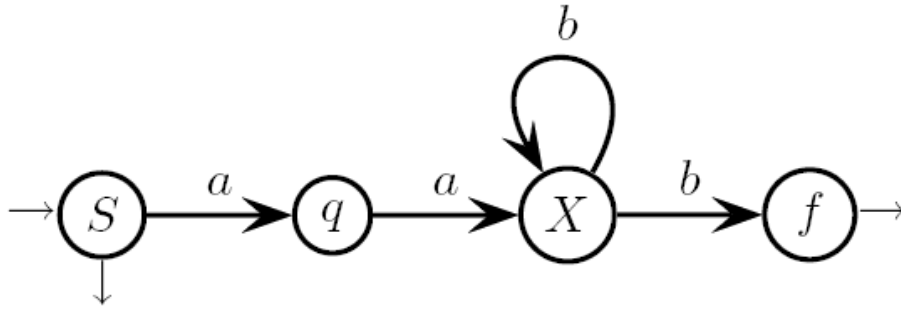


non-deterministic
automaton

$$\left\{ \begin{array}{l} A \rightarrow 0 C \mid C \mid 1 B \mid \dots \mid 9 B \\ C \rightarrow \bullet D \\ E \rightarrow 0 E \mid \dots \mid 9 E \mid \varepsilon \end{array} \right. \quad \begin{array}{l} B \rightarrow 0 B \mid \dots \mid 9 B \mid C \\ D \rightarrow 0 E \mid \dots \mid 9 E \end{array}$$

right linear grammar (axiom A)

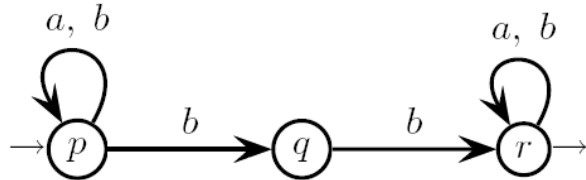
EXAMPLE – right linear grammar, but not strictly linear



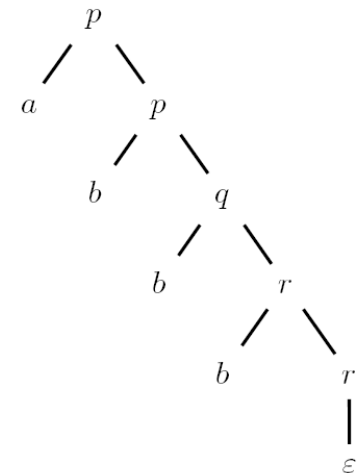
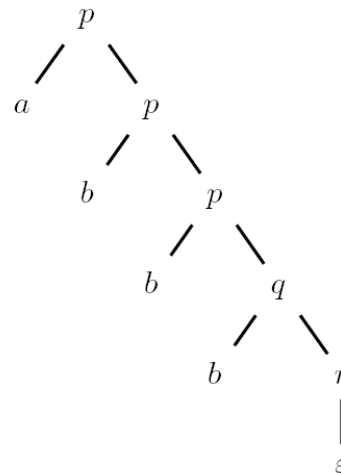
$$\begin{cases} S \rightarrow a a X \mid \varepsilon \\ X \rightarrow b X \mid b \end{cases}$$

AUTOMATON AMBIGUITY – As grammar derivations are in one-to-one correspondence with automaton computations, ambiguity is extensible to automata: an automaton is ambiguous if, and only if, the corresponding grammar is so, i.e., if a string x labels two (or more) accepting paths

EXAMPLE – searching a word in a text – recognize string “ $a b b b$ ”



$$\begin{cases} p \rightarrow a p \mid b p \mid b q \\ r \rightarrow a r \mid b r \mid \varepsilon \\ q \rightarrow b r \end{cases}$$



LEFT LINEAR GRAMMAR AND AUTOMATON

$$A \rightarrow B a$$

$$A \rightarrow B$$

$$A \rightarrow \varepsilon$$

$$L^R = (L(G))^R \text{ is generated by grammar } G_R$$

EXAMPLE – the language of the strings where the second last character is b

$$G: S \rightarrow A a \mid A b$$

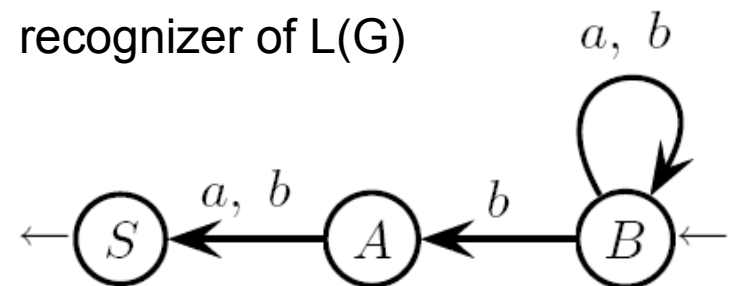
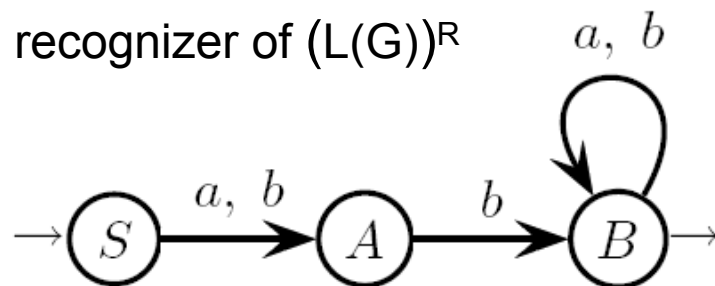
$$A \rightarrow B b$$

$$B \rightarrow B a \mid B b \mid \varepsilon$$

$$G_R: S \rightarrow a A \mid b A$$

$$A \rightarrow b B$$

$$B \rightarrow a B \mid b B \mid \varepsilon$$



FROM THE AUTOMATON TO THE REGEXP DIRECTLY
BMC METHOD (Brzozowski and McCluskey)
ALSO KNOWN AS “NODE ELIMINATION” METHOD

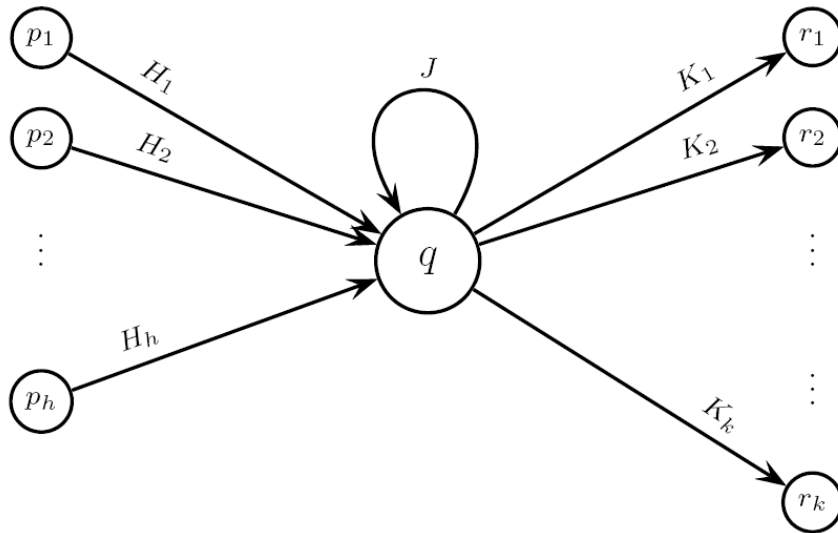
Assume the initial and final states i and t are unique and do not have incoming or outgoing arcs, respectively, i.e., they are not recirculated. If it is not so, then modify the automaton (see the next example)

INTERNAL STATES: all the other states different from i and t

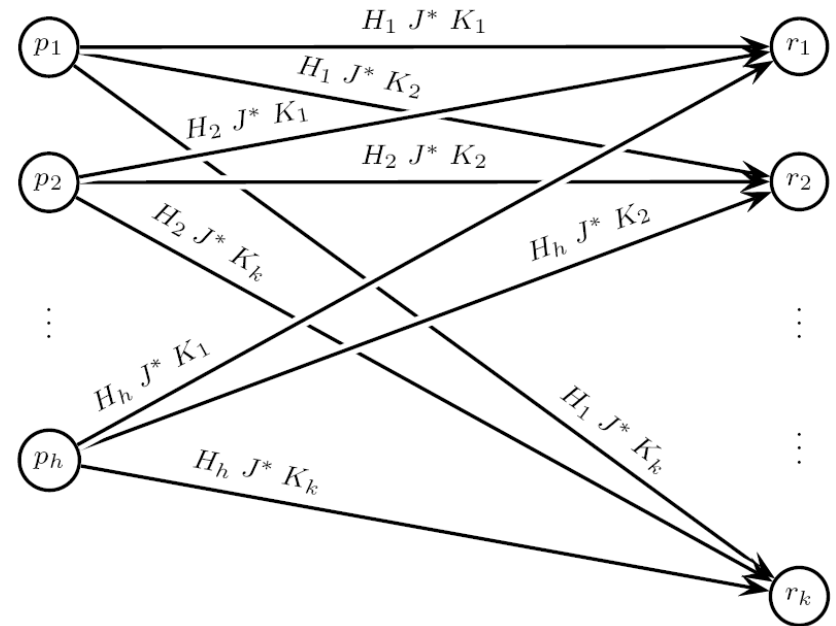
Construct the so-called GENERALIZED FINITE AUTOMATON: its arcs may be labeled with regular expressions, not only with individual terminal characters

Eliminate the internal nodes one by one, and after each elimination add one or more compensation arcs to preserve the equivalence of the automaton. Such new arcs are labeled by regexps. At the end only the nodes i and t are left, with only one arc from i to t . The regexp that labels such an arc generates the complete language recognized by the original finite automaton

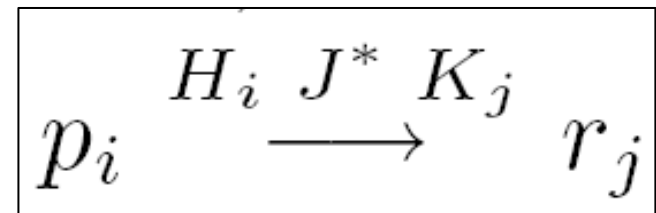
before eliminating node q



after eliminating node q

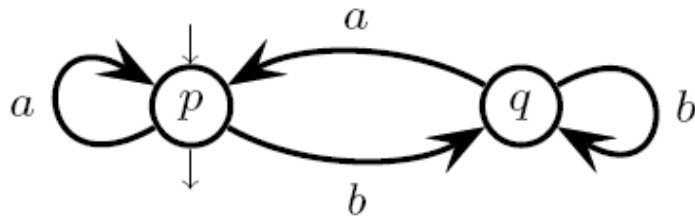


For every state pair p_i, r_j there is the arc;
sometimes p_i and r_j may coincide (self-loop)

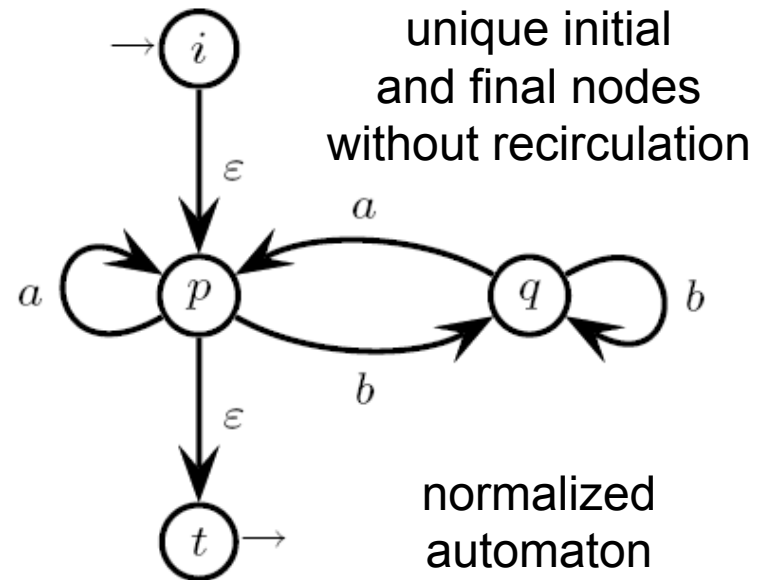


The elimination order is not relevant. However different orders may generate different regexps, all equivalent to one another but of different complexity

EXMPLE – automaton normalization and then node elimination in the order q, p

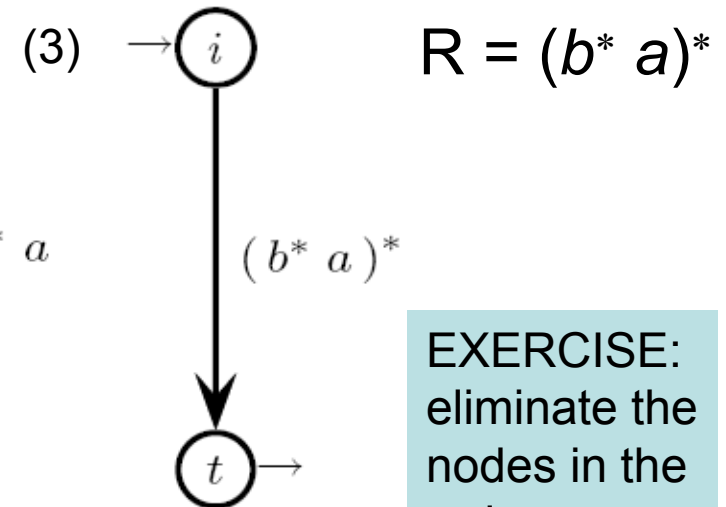
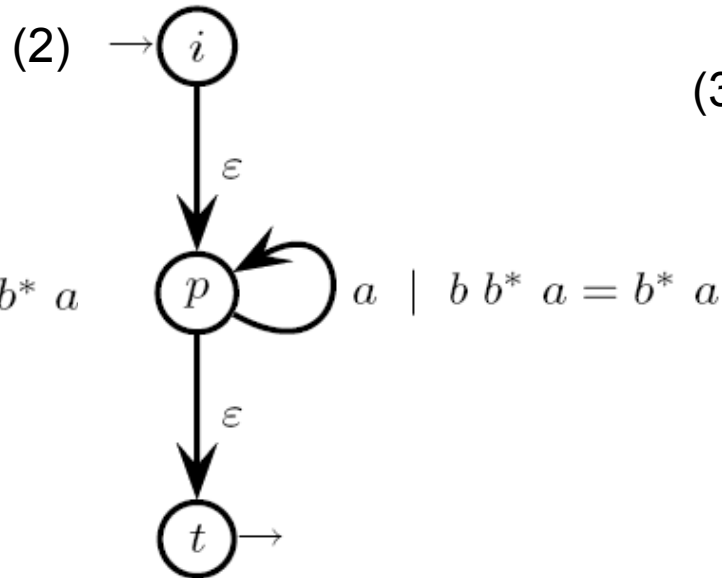
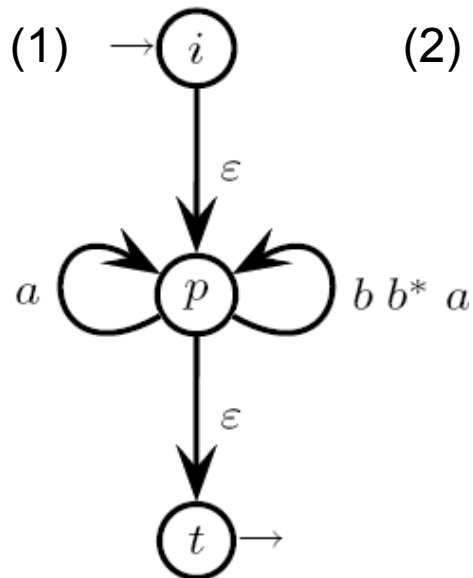


automaton



unique initial
and final nodes
without recirculation

normalized
automaton



EXERCISE:
eliminate the
nodes in the
order p, q

ELIMINATION OF INDETERMINISM – CONSTRUCTIVE PROCEDURE

For reasons of efficiency, usually the final version of a finite automaton ought to be in deterministic form

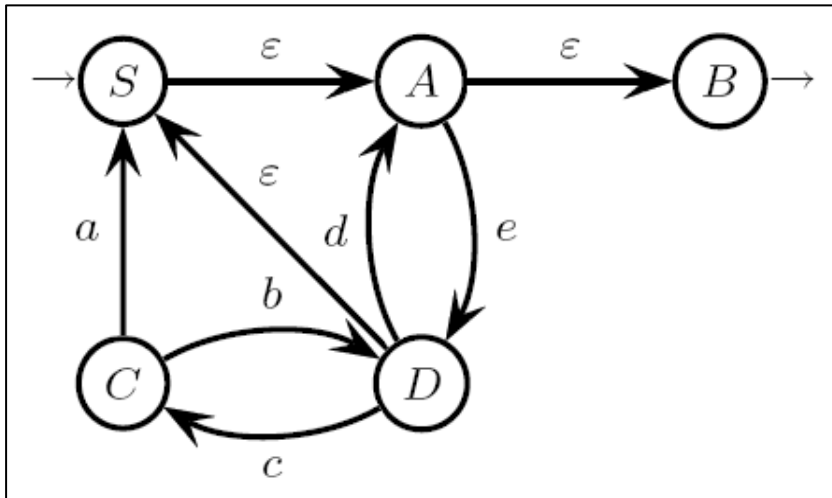
Every non-deterministic finite automaton can always be transformed into an equivalent deterministic one. Consequently every right linear grammar always admits an equivalent non-ambiguous right linear one. Thus every ambiguous regular expression can always be transformed into a non-ambiguous one

The algorithm to transform a non-deterministic automaton into a deterministic one is structured in two phases

1. Elimination of the spontaneous moves. As such moves correspond to copy rules, it suffices to apply the algorithm for removing the copy rules
2. Replacement of the non-deterministic multiple transitions by changing the automaton state set. This is the well known *subset construction*

Phase (1) can also be executed in many other ways. A practically convenient one is to cut off the ε -arcs by directly working on the state-transition graph

EXAMPLE – eliminating the ε -arcs by removing the copy rules from the grammar



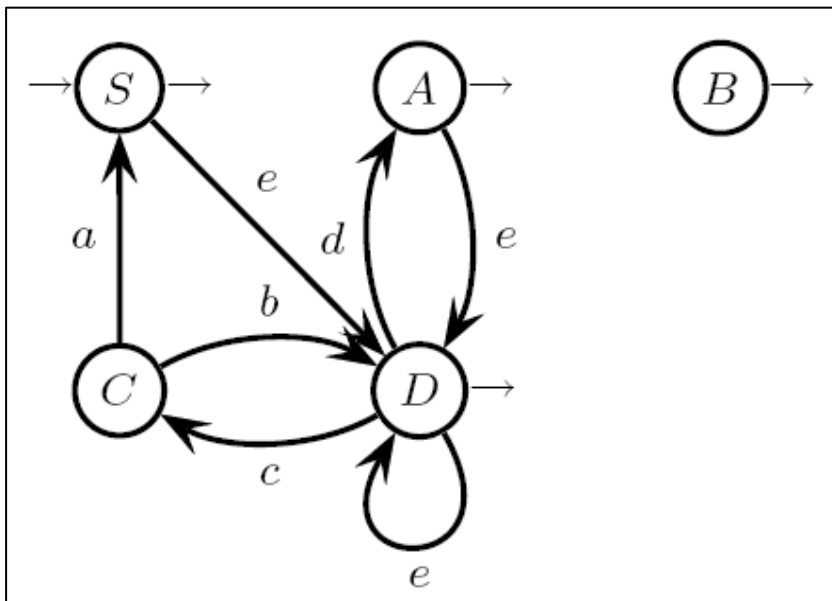
$$\left\{ \begin{array}{l} S \rightarrow A \\ C \rightarrow a S \mid b D \end{array} \right. \quad \begin{array}{l} A \rightarrow B \mid e D \\ D \rightarrow S \mid c C \mid d A \end{array} \quad B \rightarrow \varepsilon$$

grammar with copy rules

	copy
S	S, A, B
A	A, B
B	B
C	C
D	D, S, A, B

unreachable

~~$B \rightarrow \varepsilon$~~



$$\left\{ \begin{array}{l} S \rightarrow \varepsilon \mid e D \\ C \rightarrow a S \mid b D \end{array} \right. \quad \begin{array}{l} A \rightarrow \varepsilon \mid e D \\ D \rightarrow \varepsilon \mid e D \mid c C \mid d A \end{array}$$

grammar without copy rules

If after eliminating all the ε -arc the automaton is still non-deterministic, then go to the second phase

FROM A REGULAR EXPRESSION TO A FINITE STATE AUTOMATON

There are a few algorithms to transform a regexp into an automaton, which differ as for automaton characteristic, e.g., det. vs non-det., automaton size and construction complexity

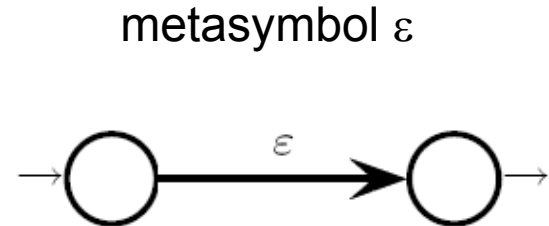
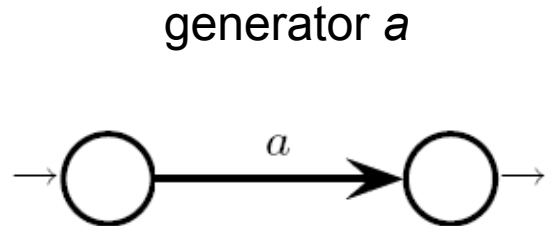
IN THE FOLLOWING THREE METHODS ARE PRESENTED

- 1) THOMPSON (or structural method)
 - 1) decomposes the regexp into subexpressions, until it reaches the atomic constituents thereof, i.e., the terminal symbols
 - 2) constructs the subexpression recognizers, connects them and builds up a network of recognizers that implement the union, concatenation and star operators
- 2) GLUSHKOV, MCNAUGHTON, YAMADA (GMY): constructs a non-det. recognizer without spontaneous moves, but with multiple transitions
- 3) BERRY-SETHI METHOD (BS): constructs a det. recognizer without spontaneous moves, but of size often larger than Thompson

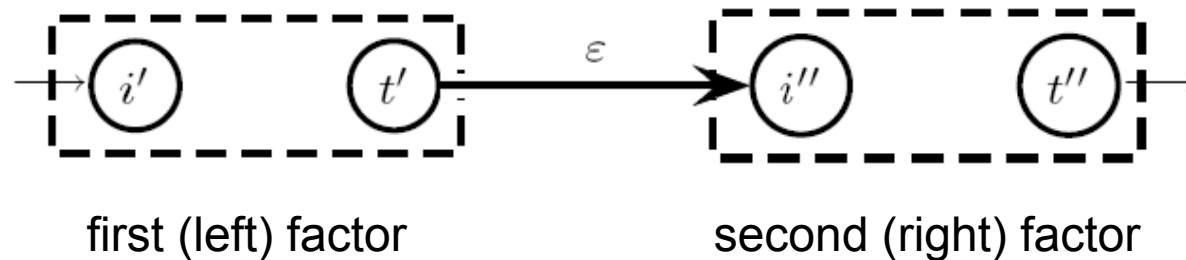
THOMPSON OR STRUCTURAL METHOD

- 1) modifies the original automaton to have unique initial and final states
- 2) is based on the correspondence of regexp and recognizer automaton

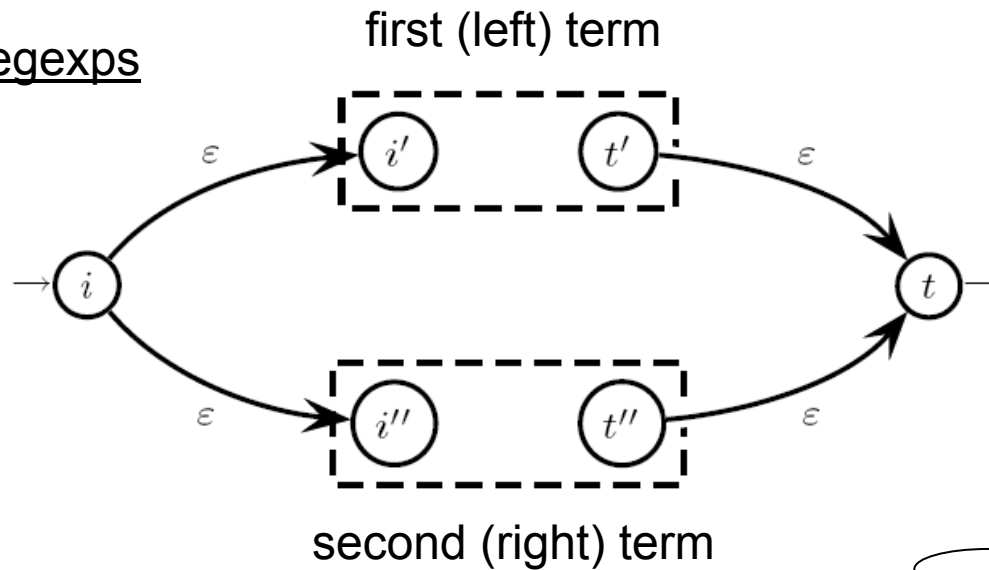
Recognizers of atomic regexps



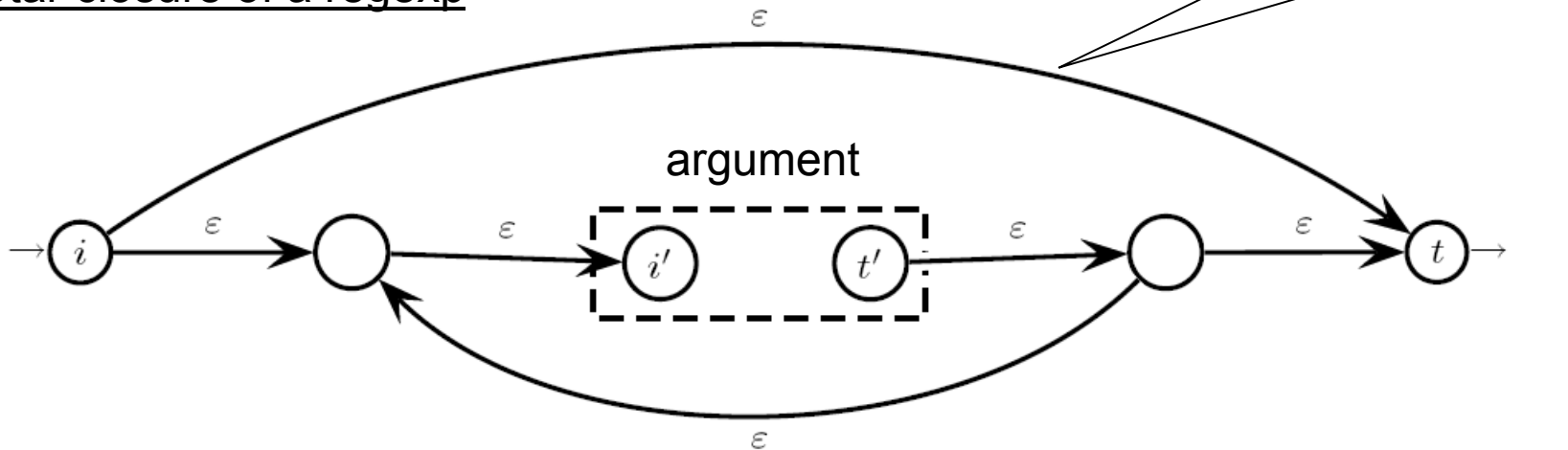
Concatenation of two regexps



Union of two regexps



Star closure of a regexp

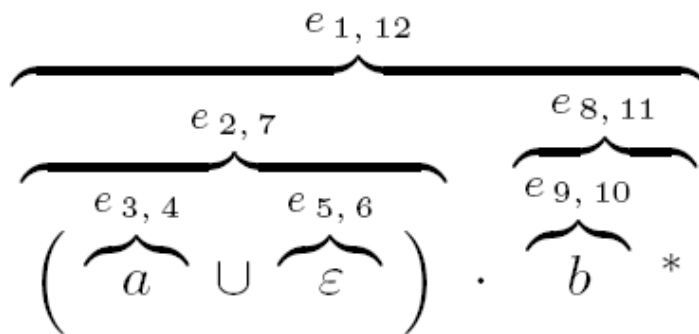


In general the outcome of the Thompson method is a non-deterministic automaton with spontaneous moves. The method is an application of the closure properties of the regular languages under the operations of union, concatenation and star

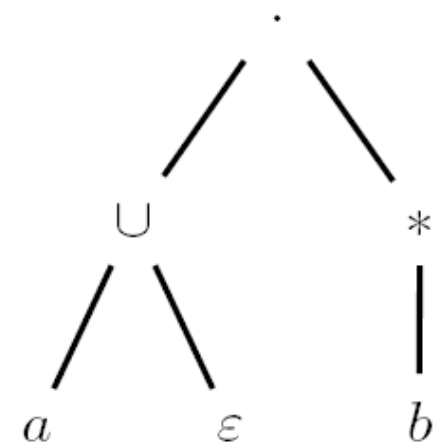
EXAMPLE $(a \cup \varepsilon) b^*$

parenthetization and
symbol / subexpression indexing

$$\left({}_1 \left({}_2 \left({}_3 a \right)_4 \cup \left({}_5 \varepsilon \right)_6 \right)_7 \cdot \left({}_8 \left({}_9 b \right)_{10} \right)^*_{11} \right)_{12}$$



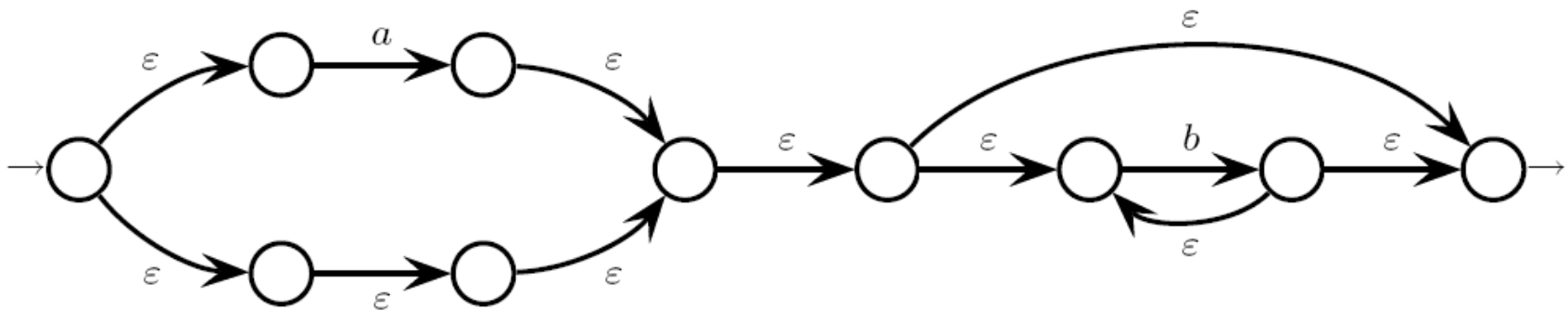
subexpression naming



structure tree

EXAMPLE (continued) $(a \cup \varepsilon) b^*$

Thompson non-deterministic automaton of the regexp,
with spontaneous moves



There are various optimizations of the Thompson method that avoid to create redundant states, e.g., the initial and final ones, or that avoid the spontaneous moves at an early stage of the construction

GLUSHKOV-MCNAUGHTON-YAMADA ALGORITHM (GMY)

It constructs the automaton equivalent to a given regexp, with states that are in a one-to-one correspondence with the generators that occur in the regexp

LOCALLY TESTABLE LANGUAGE – a subfamily of regular languages

These languages are easily recognizable as their strings satisfy a simple set of constraints, mainly based on the occurrence and adjacency of letters, e.g., all the strings that start with b , end with a or b , and contain the pairs ba , ab

DEFINITION – given a language L over the alphabet Σ

start set – *Initials*

$$Ini(L) = \{ a \in \Sigma \mid a \Sigma^* \cap L \neq \emptyset \}$$

end set – *Finals*

$$Fin(L) = \{ a \in \Sigma \mid \Sigma^* a \cap L \neq \emptyset \}$$

adjacency set – *Digrams*

$$Dig(L) = \{ x \in \Sigma^2 \mid \Sigma^* x \Sigma^* \cap L \neq \emptyset \}$$

and its complement set
(*forbidden digrams*)

$$\overline{Dig(L)} = \Sigma^2 \setminus Dig(L)$$

EXAMPLE – a locally testable language L_1

$$L_1 = (abc)^*$$

$$Ini(L_1) = \{ a \} \qquad Fin(L_1) = \{ c \} \qquad Dig(L_1) = \{ ab, bc, ca \}$$

$$\overline{Dig(L_1)} = \{ aa, ac, ba, bb, cb, cc \}$$

The three sets above characterize exactly the non-empty strings of language L_1

$$L_1 \setminus \{ \varepsilon \} = \left\{ x \mid \begin{array}{l} Ini(x) \in \{ a \} \wedge Fin(x) \in \{ c \} \\ \wedge Dig(x) \subseteq \{ ab, bc, ca \} \end{array} \right\}$$

A language L is called *local* if, and only if, it satisfies the following identity

$$L \setminus \{ \varepsilon \} = \left\{ x \mid \begin{array}{l} Ini(x) \in Ini(L) \wedge Fin(x) \in Fin(L) \\ \wedge Dig(x) \subseteq Dig(L) \end{array} \right\}$$

DEFINITION – a language L is said to be LOCAL or LOCALLY TESTABLE, if it satisfies the identity before, i.e., if it is characterized by its local sets

For every language (possibly non-regular), provided it does not contain the empty string, the definition before still holds when strict inclusion replaces equality. In fact every phrase starts or ends with a character of *Ini* or *Fin*, and the pairs of adjacent letters, i.e., the digrams, are included in those of the language. Anyway such conditions may be insufficient to exclude some invalid strings (see below)

EXAMPLE – the regular non-local language L_3 is strictly contained in the set of the strings that start and end with b , and that do not contain the forbidden digram bb ; in fact such a set also contains strings of odd length, while language L_3 does not

$$L_3 = b (a a)^+ b$$

$$Ini(L_3) = Fin(L_3) = \{ b \}$$

$$Dig(L_3) = \{ a a, a b, b a \}$$

$$\overline{Dig(L_3)} = \{ b b \}$$

A language L is local if, and only if, every string of L matches the constraints given by the local sets *Ini*, *Fin* and *Dig*. In fact, as before the regular language L_1 is local, whereas the (still regular) language L_3 is not

IT IS SIMPLE TO DESIGN THE RECOGNIZER OF A LOCAL LANGUAGE

Scan the input string from left to right and check whether

- the initial character belongs to the set *Ini*
- every digram belongs to the set *Dig*
- the final character belongs to the set *Fin*

The string is accepted if, and only if, all the above checks succeed

We can implement the above recognizer by resorting to a sliding window with a width of two characters, which is shifted over the input string from left to right. At each shift step the window contents are checked, and if the window reaches the end of the string and all the checks succeed, then the string is accepted, otherwise it is rejected. This sliding window algorithm is simple to implement by means of a NON-DETERMINISTIC AUTOMATON

CONSTRUCTION OF THE RECOGNIZER OF THE LOCAL LANGUAGE

Given the sets Ini , Fin and Dig , the corresponding recognizer has

initial states

$$q_0 \cup \Sigma$$

final states

$$Fin$$

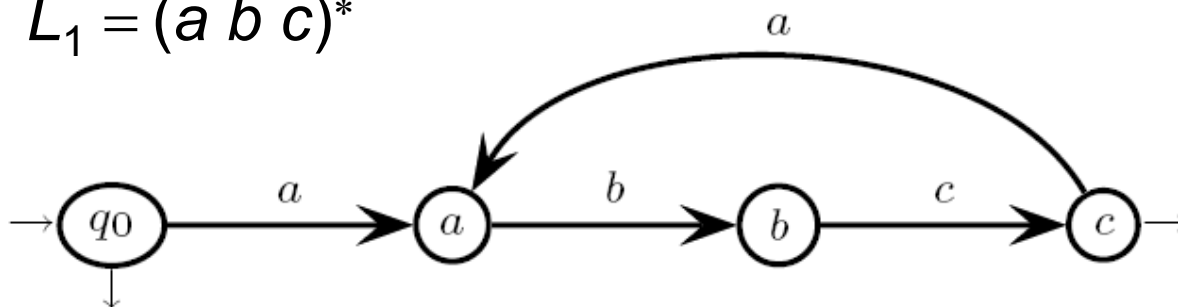
transitions

$$q_0 \xrightarrow{a} a \text{ if } a \in Ini$$

$$a \xrightarrow{b} b \text{ if } ab \in Dig$$

If the language contains the empty string, the initial state q_0 is final as well

EXAMPLE $L_1 = (a b c)^*$



deterministic recognizer of the local language L_1

LINEAR REGULAR EXPRESSION

A regexp is said to be LINEAR if there is not any repeated generator

$(a b c)^*$ is linear (all its generators differ)

$(a b)^* a$ is not linear, although its language is local

THE LANGUAGE GENERATED BY A LINEAR REGEXP IS LOCAL

Linearity implies the regexp subexpressions are defined over disjoint alphabets. But a regexp is the composition of its subexpressions, thus the language of a linear regexp is local as a consequence of the closures of the local languages over disjoint alphabets. Notice that the opposite implication does not hold

This implies that constructing the recognizer for a generic regular language reduces to the problem of finding the characteristic local sets *Ini*, *Fin*, *Dig* of such a generic language, provided the alphabet is slightly modified (see next)

HOW TO COMPUTE THE LOCAL SETS OF A REGULAR LANGUAGE

Does a regexp e generate the empty string ?

if $\text{Null}(e) = \text{true}$ **then** $\varepsilon \in L(e)$
if $\text{Null}(e) = \text{false}$ **then** $\varepsilon \notin L(e)$

$$\text{Null}(\emptyset) = \text{false}$$

$$\text{Null}(\varepsilon) = \text{true}$$

$$\text{Null}(a) = \text{false} \quad \text{for every character } a$$

$$\text{Null}(e \cup e') = \text{Null}(e) \vee \text{Null}(e')$$

$$\text{Null}(e \cdot e') = \text{Null}(e) \wedge \text{Null}(e')$$

$$\text{Null}(e^*) = \text{true}$$

$$\text{Null}(e^+) = \text{Null}(e)$$

recursive rules
that compute
the predicate *Null*
of a regexp

EXAMPLE

$$\begin{aligned} \text{Null}((a \cup b)^* b a) &= \text{Null}((a \cup b)^*) \wedge \text{Null}(b a) \\ &= \text{true} \wedge (\text{Null}(b) \wedge \text{Null}(a)) \\ &= \text{true} \wedge (\text{false} \wedge \text{false}) \\ &= \text{true} \wedge \text{false} \\ &= \text{false} \end{aligned}$$

set of initials

$$Ini(\emptyset) = \emptyset$$

$$Ini(\varepsilon) = \emptyset$$

$$Ini(a) = \{ a \} \quad \text{for every character } a$$

$$Ini(e \cup e') = Ini(e) \cup Ini(e')$$

$$Ini(e \cdot e') = \text{if } Null(e) \text{ then } Ini(e) \cup Ini(e') \text{ else } Ini(e) \text{ end if}$$

$$Ini(e^*) = Ini(e^+) = Ini(e)$$

recursive rules
that compute
the set *Ini*
of a regexp

set of finals

$$Fin(\emptyset) = \emptyset$$

$$Fin(\varepsilon) = \emptyset$$

$$Fin(a) = \{ a \} \quad \text{for every character } a$$

$$Fin(e \cup e') = Fin(e) \cup Fin(e')$$

$$Fin(e \cdot e') = \text{if } Null(e') \text{ then } Fin(e) \cup Fin(e') \text{ else } Fin(e') \text{ end if}$$

$$Fin(e^*) = Fin(e^+) = Fin(e)$$

recursive rules
that compute
the set *Fin*
of a regexp

the computation of the two sets *Ini* and *Fin* is dual: just mirror the regexp

$$Dig(\emptyset) = \emptyset$$

$$Dig(\varepsilon) = \emptyset$$

$$Dig(a) = \emptyset \quad \text{for every character } a$$

$$Dig(e \cup e') = Dig(e) \cup Dig(e')$$

$$Dig(e \cdot e') = Dig(e) \cup Dig(e') \cup Fin(e) \cdot Ini(e')$$

$$Dig(e^*) = Dig(e^+) = Dig(e) \cup Fin(e) \cdot Ini(e)$$

recursive rules
that compute
the set *Dig*
of a regexp

first compute the predicate *Null*, then the sets *Ini* and *Fin*, and finally the set *Dig*

sometimes a few passages can be quickly carried out by visual inspection

$$Null(a(b|c)^*) = Null(a) \wedge Null((b|c)^*) = false \wedge true = false$$

EXAMPLE

$$Dig(a(b|c)^*) = Dig(a) \cup Dig((b|c)^*) \cup Fin(a) \cdot Ini((b|c)^*)$$

$$= \Phi \cup Dig(b|c) \cup Fin(b|c) \cdot Ini(b|c) \cup \{a\} \cdot Ini(b|c)$$

$$= Dig(b) \cup Dig(c) \cup (Fin(b) \cup Fin(c)) \cdot (Ini(b) \cup Ini(c)) \cup \{a\} \cdot (Ini(b) \cup Ini(c))$$

$$= \Phi \cup \Phi \cup (\{b\} \cup \{c\}) \cdot (\{b\} \cup \{c\}) \cup \{a\} \cdot (\{b\} \cup \{c\})$$

$$= \{bb, bc, cb, cc\} \cup \{ab, ac\} = \{ab, ac, bb, bc, cb, cc\}$$

HOW TO EXTEND FROM A LINEAR REGEXP TO A GENERIC ONE

Transform the generic regexp into a linear one by distinguishing the generators. To do so, denumerate the generators by applying a running index to each of them

$$e = (a\ b)^* a \quad \text{becomes} \quad e_{\#} = (a_1\ b_2)^* a_3$$

PHASES OF THE GMY ALGORITHM

- 1) Construct the local recognizer of the numbered regexp (now linear)
- 2) Cancel the index from the local recognizer and thus obtain the generic recognizer of the original generic regexp

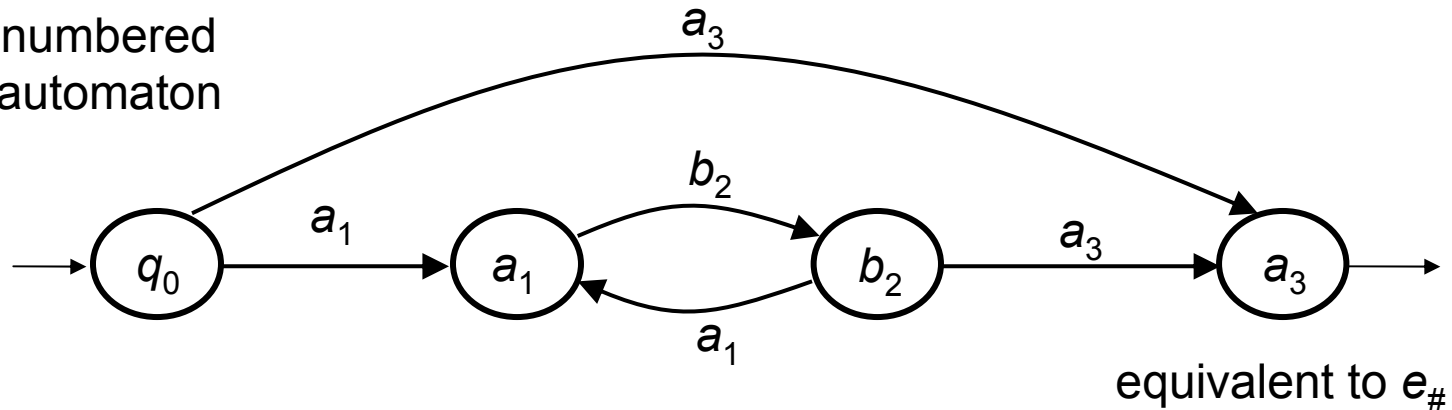
GMY ALGORITHM (in short)

- 1) denumerate the regexp e and obtain the linear regexp $e_{\#}$
- 2) compute the three characteristic local sets *Ini*, *Fin* and *Dig* of $e_{\#}$
- 3) design the recognizer of the local language generated by $e_{\#}$
- 4) cancel the indexing and thus obtain the recognizer of e

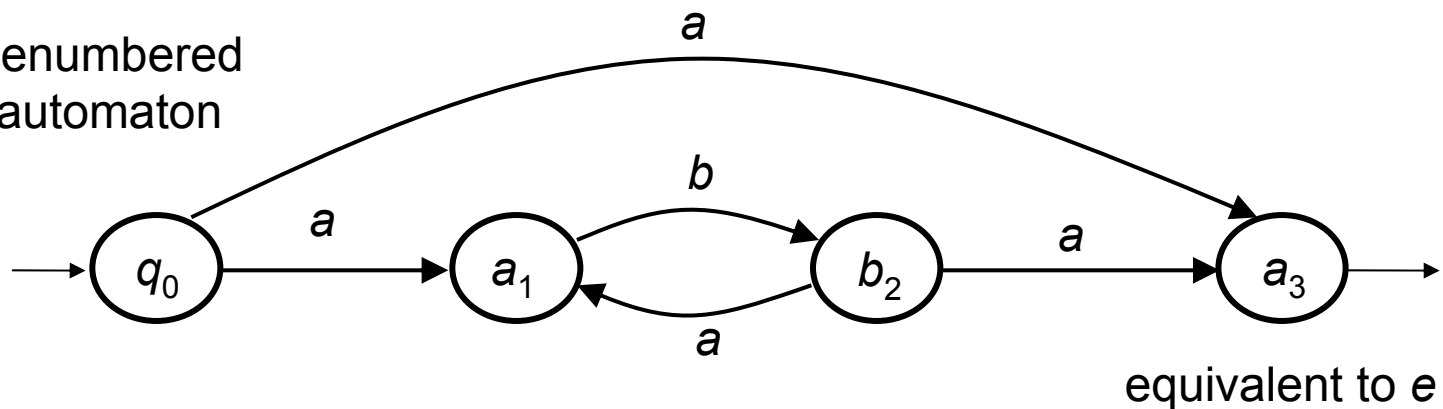
CAUTION – in general the final recognizer is non-deterministic

EXAMPLE regexp $e = (a\ b)^* a$ numbered regexp $e_{\#} = (a_1\ b_2)^* a_3$

numbered
automaton



denumbered
automaton



The result is a non-deterministic automaton without spontaneous moves, with as many states as the occurrences of generators in the regexp are, and one more state (the initial one). You may wish to rename the states.

CONSTRUCTION OF THE DETERMINISTIC RECOGNIZER (BERRY-SETHI ALGORITHM)

In order to obtain the deterministic recognizer, we can just apply the subset construction to the non-deterministic recognizer built by the GMY algorithm

However there is a more direct algorithm called BERRY-SETHI (BS)

Consider the end-marked regexp $e \dashv$ instead of the original regexp e (we still call it e and we understand it includes also the end-marker)

Let e be a regexp over the alphabet Σ , and let $e_{\#}$ be the numbered version of e over $\Sigma_{\#}$ with predicate *Null* and local sets *Ini*, *Fin* and *Dig*

Define the set $Fol(c_{\#})$ of the *followers* of $c_{\#} \in \Sigma_{\#}$ in this way

$$Fol(c_{\#}) \in \wp(\Sigma_{\#} \cup \{\dashv\})$$

$$\text{and } Fol(\dashv) = \Phi$$

$$Fol(a_i) = \{ b_j \mid a_i b_j \in Dig(e_{\#} \dashv) \} \quad a_i \text{ and } b_j \text{ may coincide}$$

In practice the set Fol is a different way to express the digram set Dig , and it is usually presented in the form of a table (see next)

BS ALGORITHM

Each state is denoted by a subset of $\Sigma_{\#} \cup \neg$

The algorithm examines the states, and constructs their destination states and outgoing moves, by the subset construction

Set $Ini(e_{\#} \neg)$ is the initial state, final states contain \neg , and the state set Q starts with the initial state

NOTICE the labelings of the nodes express the question: “*What is expected next in the input tape ?*”

$q_0 := Ini(e_{\#} \neg)$

$Q := \{ q_0 \}$

$\delta := \emptyset$

while $(\exists q \in Q$ s.t. q is unmarked) **do**

mark state q as visited

for (each character $c \in \Sigma$) **do**

$q' := \bigcup_{\substack{\forall c_{\#} \in \Sigma_{\#} \\ \text{s.t. } c_{\#} \in q}} Fol(c_{\#})$

if $(q' \neq \emptyset)$ **then**

if $(q' \notin Q)$ **then**

set q' as a new unmarked state

$Q := Q \cup \{ q' \}$

end if

$\delta := \delta \cup \left\{ q \xrightarrow{c} q' \right\}$

end if

end for

end while \square

$Ini(e_{\#} \neg) = \{ a_1, b_2, a_4 \}$

for instance

$Fol(a_1) \cup Fol(a_4) = \{ a_1, b_2, a_4, c_5 \}$

$Fol(c_5) = \{ a_4, \neg \}$

EXAMPLE

$$e = (a \mid bb)^* (ac)^+$$

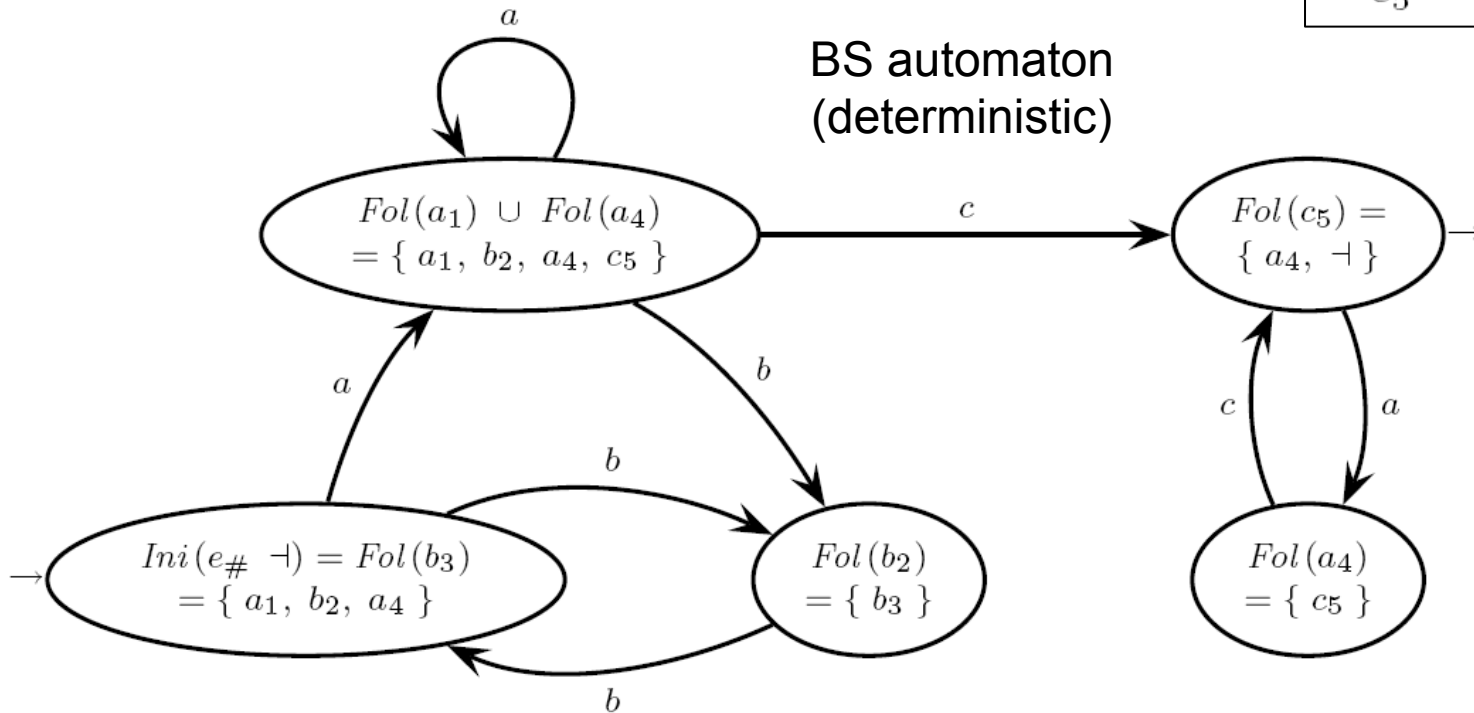
$$e_{\#} \dashv = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+ \dashv$$

$$Ini(e_{\#} \dashv) = \{ a_1, b_2, a_4 \}$$

$$Fin(e_{\#} \dashv) = \{ \dashv \}$$

$$Dig(e_{\#} \dashv) = \{ a_1 a_1, a_1 b_2, a_1 a_4, b_2 b_3, b_3 a_1, b_3 b_2, b_3 a_4, a_4 c_5, c_5 a_4, c_5 \dashv \}$$

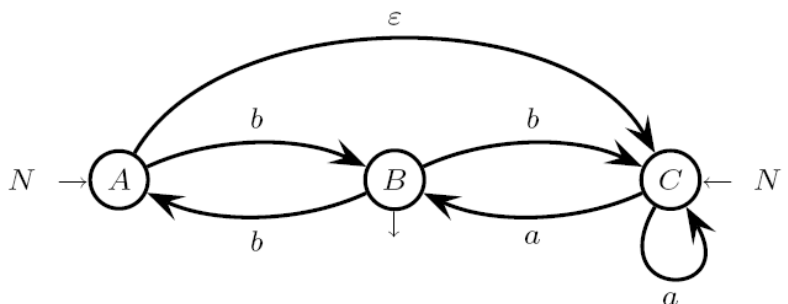
$c_{\#}$	$Fol(c_{\#})$
a_1	a_1, b_2, a_4
b_2	b_3
b_3	a_1, b_2, a_4
a_4	c_5
c_5	a_4, \dashv



It may have
equivalent
states
(minimize it)

THE BS ALGORITHM CAN BE USED TO DETERMINIZE A NON-DETERMINISTIC AUTOMATON (even with ε -arcs) – EXAMPLE

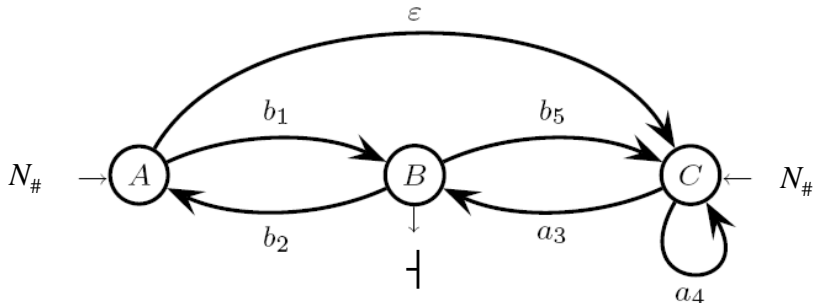
nondeterministic automaton



number the automaton, then find the initials and the followers by inspecting how the transitions are concatenated to each other (skip over the spontaneous transitions)

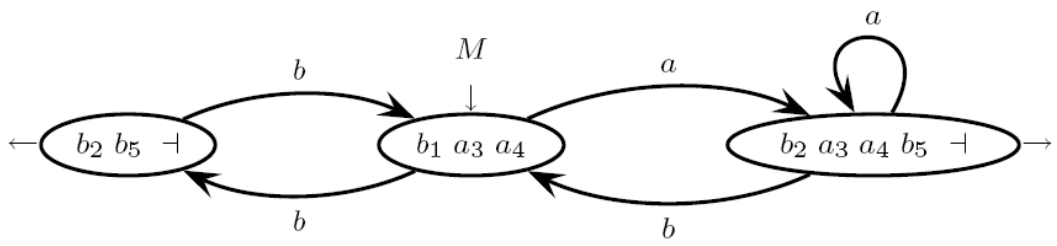
$$Ini\big(L\big(N_{\#}\vdash\big)=\{b_1,a_3,a_4\}$$

numbered automaton



$c_{\#}$	$Fol(c_{\#})$
b_1	b_2, b_5, \vdash
b_2	b_1, a_3, a_4
a_3	b_2, b_5, \vdash
a_4	a_3, a_4
b_5	a_3, a_4

deterministic automaton



REGEXP WITH COMPLEMENT AND INTERSECTION

Regexps may also contain the operators of complement, intersection and set difference, which are very useful to make the regexp more concise

THE FAMILY REG IS CLOSED UNDER COMPLEMENT AND INTERSECTION (hence also under set difference)

if $L', L'' \in REG$ then $\neg L' \in REG$ and $L' \cap L'' \in REG$

DETERMINISTIC RECOGNIZER OF THE COMPLEMENT LANGUAGE

$\neg L = \Sigma^* \setminus L$

Assume the recognizer M of L is deterministic, with initial state q_0 , state set Q , set of final states F and transition function δ

ALGORITHM – deterministic recognizer \overline{M} of the complement language

Complete the automaton M by adding the error state p and the missing moves

1. create the error state p , not in Q , so the states of \overline{M} are $Q \cup \{p\}$
2. the complement transition function is $\overline{\delta}$, see below
3. swap the non-final and final states, see below

$$\overline{F} = (Q \setminus F) \cup \{p\}$$

$$\begin{aligned}\overline{\delta}(q, a) &= \delta(q, a), \text{ if } \delta(q, a) \in Q \\ \overline{\delta}(q, a) &= p, \text{ if } \delta(q, a) \text{ is undefined} \\ \overline{\delta}(p, a) &= p, \text{ for every character } a \in \Sigma\end{aligned}$$

A recognizing path of M does not end into a final state of \overline{M}

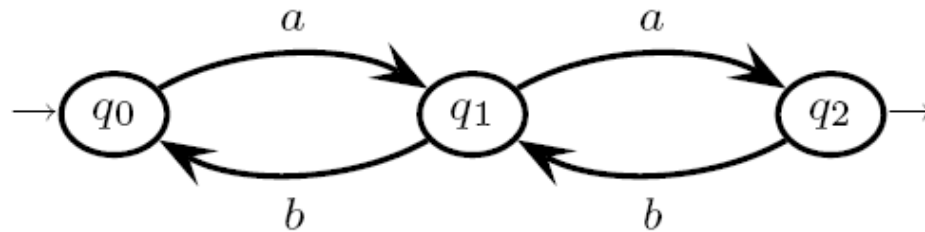
$$x \in L(M)$$

A non-recognizing path of M does not end into a final state of \overline{M}

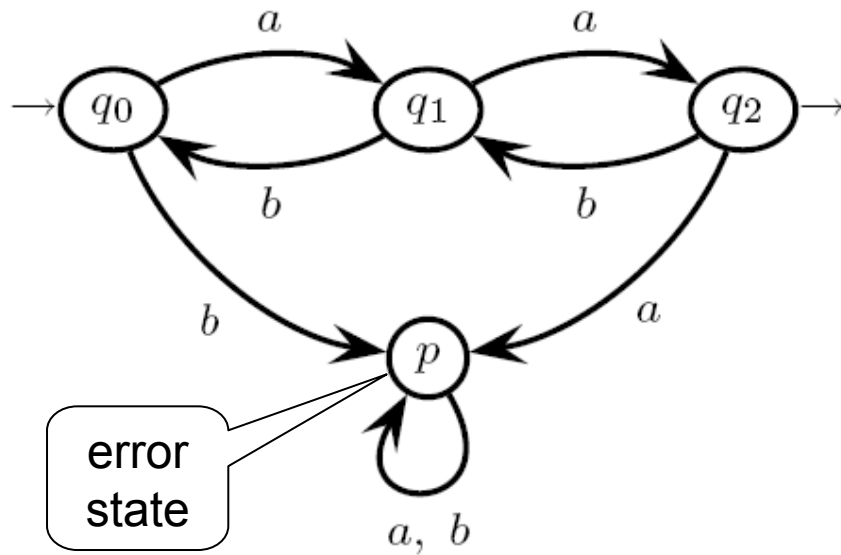
$$x \notin L(\overline{M})$$

EXAMPLE – COMPLEMENT AUTOMATON

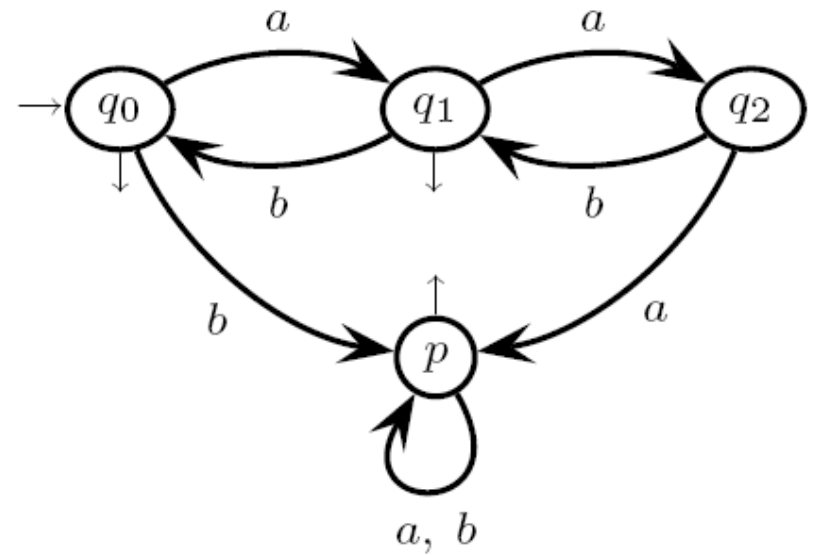
original automaton
(deterministic)



natural completion



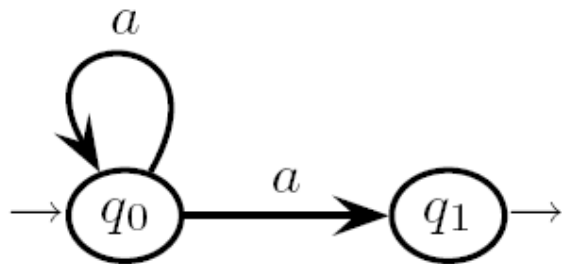
complement automaton



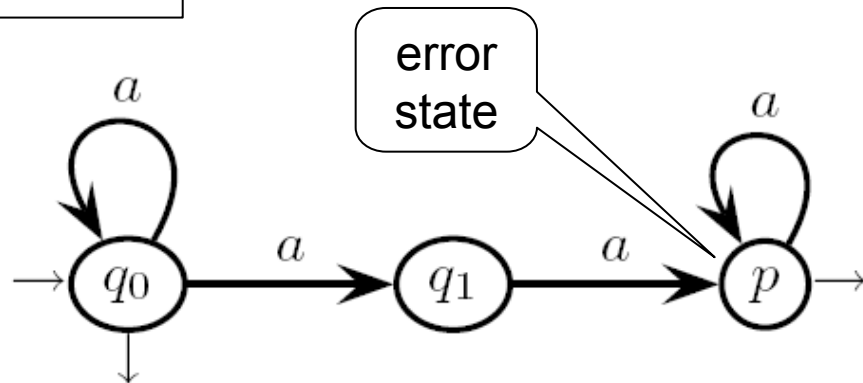
CAUTION: for the complement construction to work correctly, the original automaton must be deterministic, otherwise the original and complement languages may be not disjoint, which fact would be in violation of the complement definition (see below)

EXAMPLE

$$L \cap \neg L = \emptyset$$



original automaton
(non-deterministic)



pseudo-complement automaton

The pseudo-complement automaton accepts all the strings a^+ , which also belong to the original language, because the original automaton is non-deterministic

CAUTION – the complement automaton may contain useless states and may not be in the minimal form either; it should be reduced and minimized, if necessary

(CARTESIAN) PRODUCT OF AUTOMATA

A very common construction of formal languages, where a single automaton simulates the computation of two automata that work in parallel on the same input string. It is very useful to construct the intersection automaton

To obtain the intersection automaton we can resort to the De Morgan theorem

- construct the deterministic recognizers of the two languages
- construct the respective complement automata (as before)
- construct their union (e.g., use the Thompson method)
- make deterministic the union automaton (e.g., use BS or subset)
- complement again and thus obtain the intersection automaton

The cartesian product can also be obtained by a more direct construction

DIRECT CONSTRUCTION OF THE PRODUCT OF AUTOMATA

The intersection of the two languages is recognized directly by the cartesian product of their automata (in general non-deterministic)

Suppose both automata do not contain any spontaneous moves (they may be non-deterministic anyway)

The state set of the product machine is the cartesian product of the state sets of the two automata. Each product state is a pair $\langle q', q'' \rangle$, where the left (right) member is a state of the first (second) machine. The move is

$$\boxed{\langle q', q'' \rangle \xrightarrow{a} \langle r', r'' \rangle} \quad \text{if and only if} \quad \boxed{q' \xrightarrow{a} r' \text{ and } q'' \xrightarrow{a} r''}$$

The product machine has a move if, and only if, the projection of such a move onto the left (right) component is a move of the first (second) automaton

The initial and final state sets are the cartesian products of the initial and final state sets of the two automata, respectively

WHY DOES IT WORK ?

- if a string is accepted by the product machine, it is accepted simultaneously by the two automata as well
- if a string is rejected by the product machine, either automaton or both ones reject it as well, which means that the string is not in the intersection

NOTICE

- the product construction is equivalent to simulating both machines in parallel.
- with some modification, the construction can be adapted to other language operators; the reader may wish to try for union (not very easy) and shuffle

MORE ON THE PRODUCT

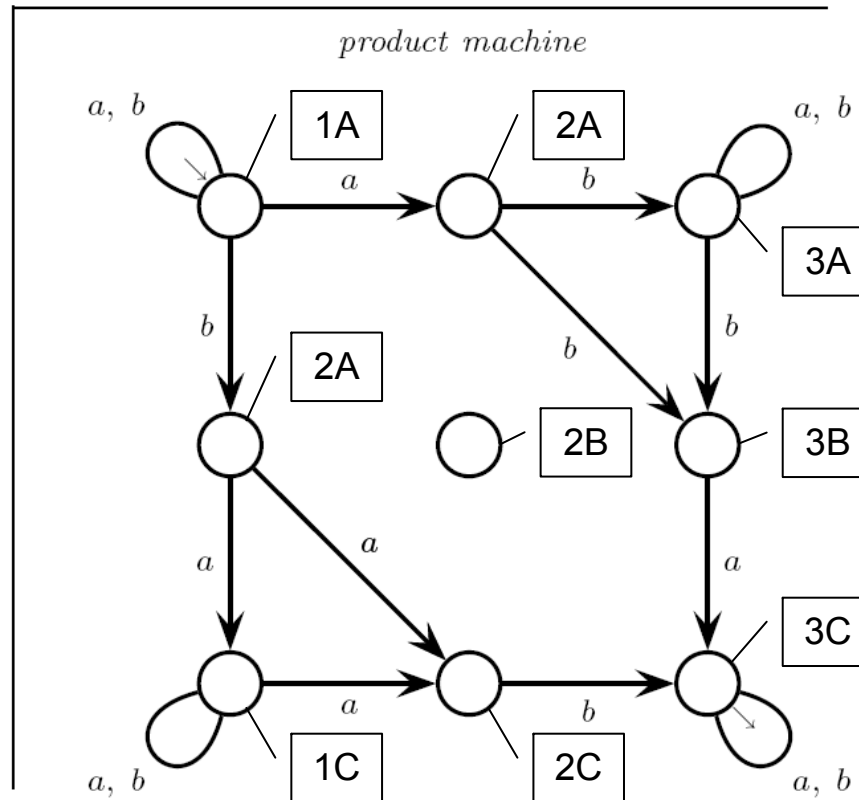
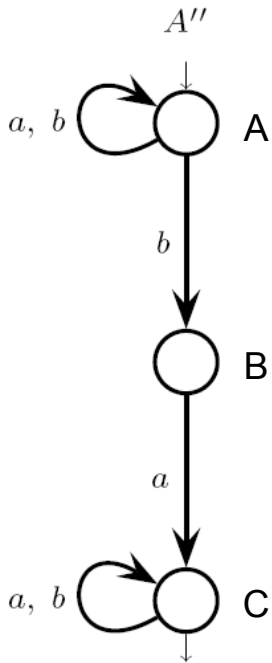
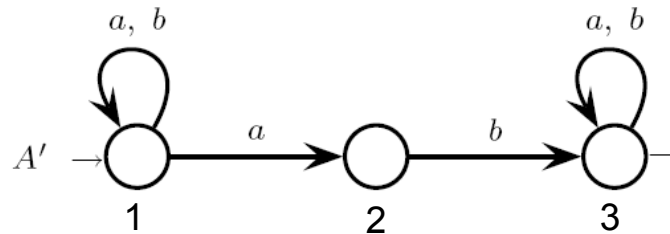
- the product automaton may have useless states and may not be minimal
- if either automaton is deterministic, their product is deterministic as well

EXAMPLE – INTERSECTION AUTOMATON

$$L' = (a \mid b)^* a b (a \mid b)^*$$

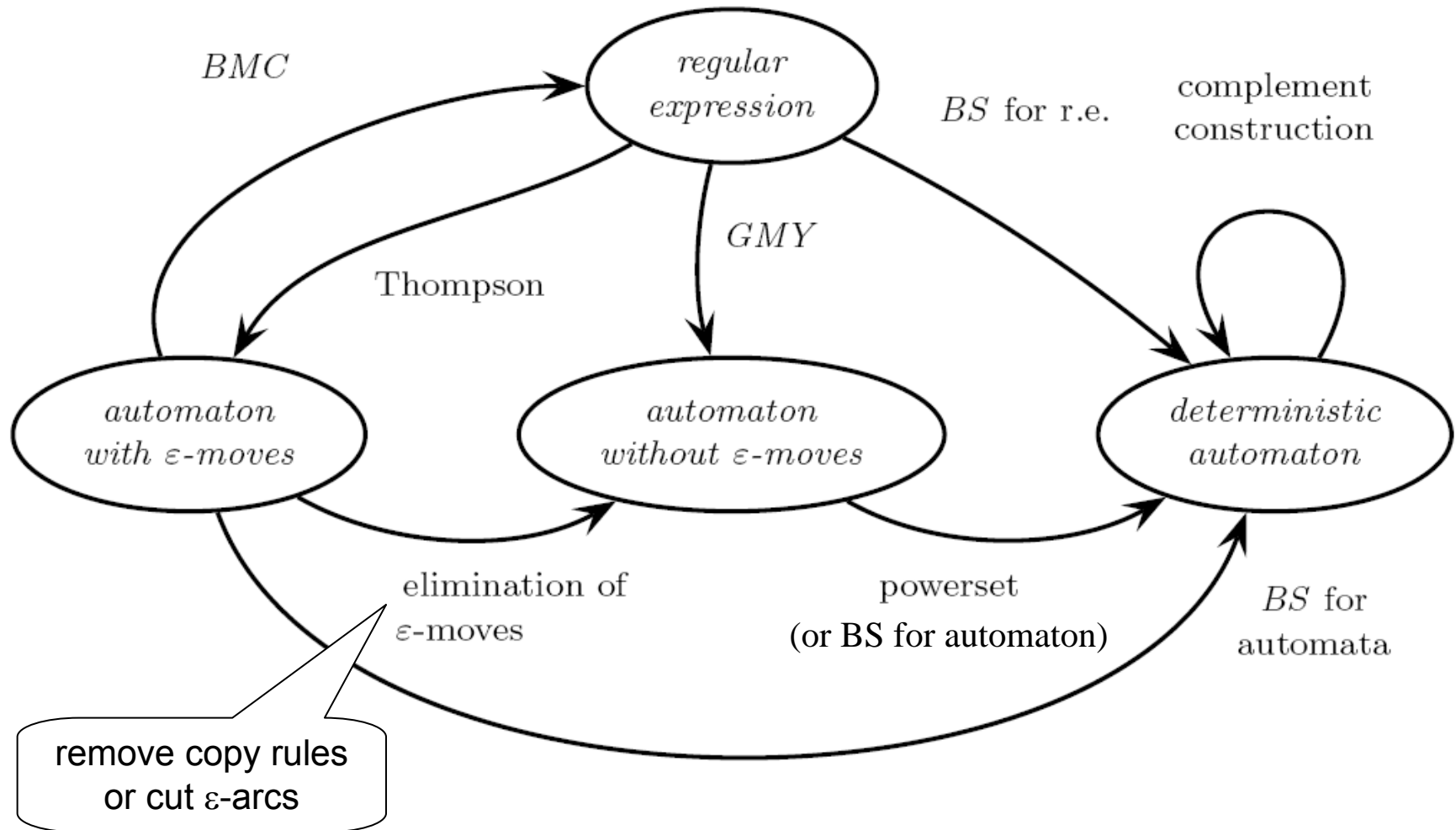
$$L'' = (a \mid b)^* b a (a \mid b)^*$$

component
machines

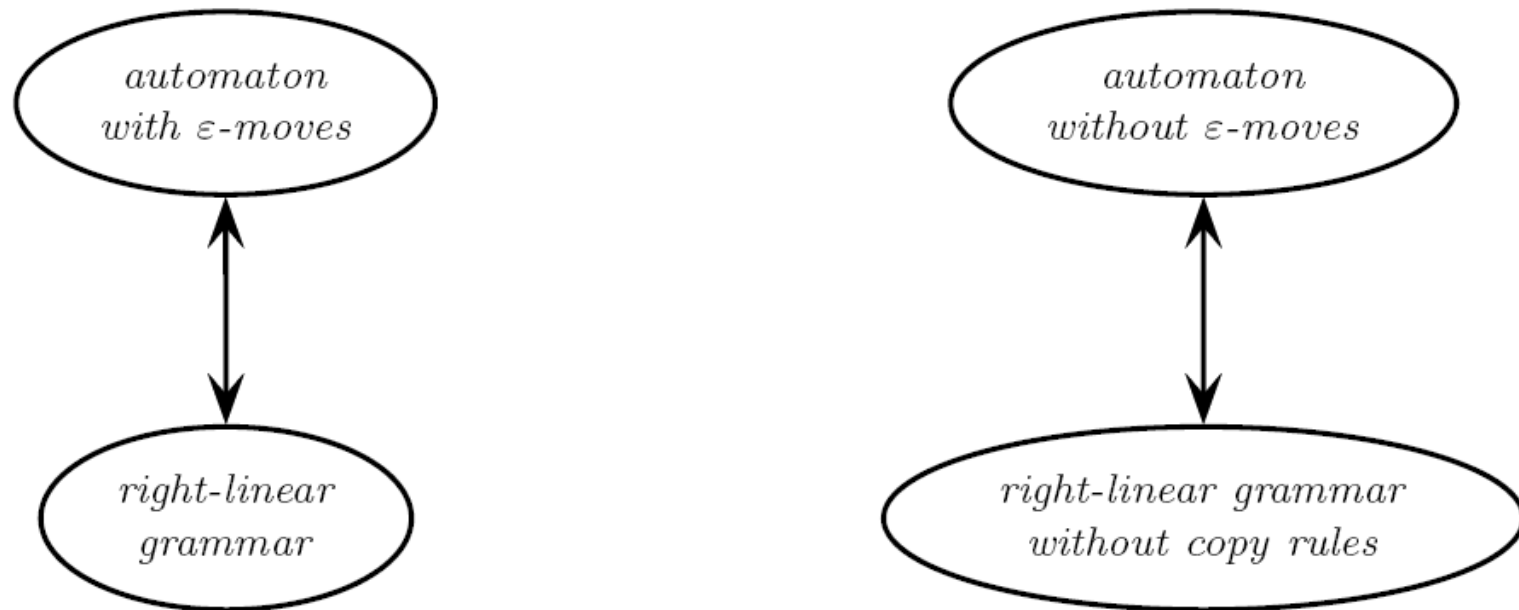


the product
automaton is
non-deterministic
and has a
useless state
(reduce it)

SUMMARY OF THE TRANSFORMATIONS OF REGEXPS AND AUTOMATA

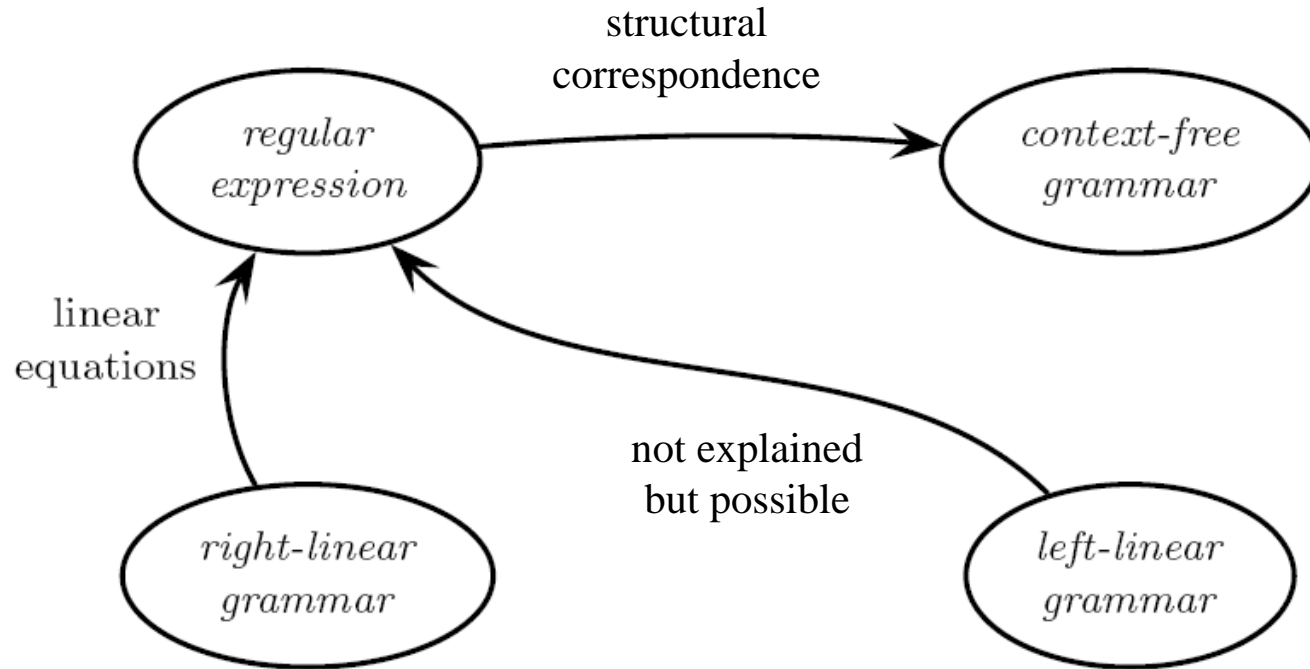


SUMMARY OF THE CORRESPONDENCE OF REGEXPS AND GRAMMARS



a similar correspondence exists for left-linear grammars

SUMMARY OF THE TRANSFORMATIONS OF REGEXPS AND GRAMMARS



to obtain a a right (left) linear grammar from a regexp, first convert the regexp into an automaton, then rewrite the automaton as a right (left) linear grammar