Relations between

Finite-state automata, Grammars, Regular expressions

Prof. A. Morzenti

EQUIVALENCE OF FSA, UNILINEAR GRAMMARS and RE

Plan of coming slides:

- define ***unilinear*** grammars (UNIL-G)
- define procedures to translate a UNIL-G into a FSA and a FSA into a UNIL-G
- define procedure to translate a FSA into a RE
- define procedures to design a FSA starting from a RE (i.e., to translate a RE into a FSA)

Therefore it is proved constructively that UNIL-G $\equiv$ FSA and FSA $\equiv$ RE

By transitivity it ramains proved also that UNIL-G $\equiv$ RE

ER, UNIL-G and FSA may be considered as ***notational variants*** to denote languages

UNILINEAR GRAMMARS (called «of type 3» in the Chomsky hierarchy)
A CLASS OF GRAMMARS EQUIVALENT TO FSA

RIGHT-LINEAR RULE: $\boxed{A \to uB \quad \text{with} \quad u \in \Sigma^*, \quad B \in (V \bigcup \varepsilon)}$

LEFT-LINEAR RULE: $\boxed{A \to Bv \quad \text{with} \quad v \in \Sigma^*, \; B \in (V \bigcup \varepsilon)}$

A grammar is ***unilinear*** iff its rules are either **all right-linear** or **all left-linear**

Syntax trees grow totally unbalanced (resp. toward the right or left)

Without loss of generality one can require that a unilinear grammar has

- ***STRICTLY unilinear rules***: with at most one terminal $\boldsymbol{A \to aB}$**,** $\boldsymbol{a \in (\Sigma \cup \varepsilon)}$ $\boldsymbol{B \in (V \cup \varepsilon)}$
- all **terminal rules are empty**: e.g., rule $B \to b$ replaced by rules $B \to bB'$ and $B' \to \varepsilon$

Therefore we can assume (for the right case) just rules of type $\mathrm{A} \to aB | \varepsilon$ with $a \in \Sigma, B \in V$

(for the left case rules of type $\mathrm{A} \to Ba | \varepsilon$ with $a \in \Sigma, B \in V$)

languages generated by unilinear grammars exhibit ***inevitable repetitions***

PROPERTY: Let G be a unilinear grammar

Every sufficiently long sentence x (i.e., longer than a grammar-dependent constant k)

Can be factorized as $x = t\ u\ v$ - with **nonempty *u***

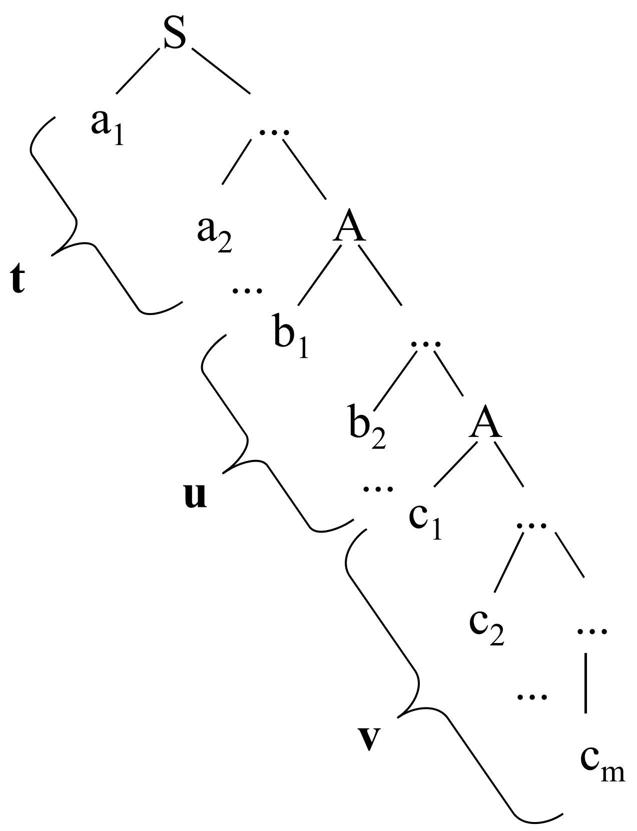
so that, $\forall\ n \geq 1,$ the string $t\ u^n\ v \in L(G)$

(the sentence can be «pumped» by injecting string u an arbitrary number of times)

NOTE the analogy with the ***pumping lemma*** of finite state automata…

in fact we'll see that unilinear gr. $\Leftrightarrow$ FSA

Proof:
Consider a strictly right-linear *G* with *k* nonterminal symbols
In the derivation of a sentence *x* whose length is *k* or more,
there is necessarily a n.t. *A* that appears at least two times



$$\boxed{\begin{array}{l} t = a_1 a_2 \ldots, \quad u = b_1 b_2 \ldots \quad v = c_1 c_2 \ldots c_m \\ S \overset{+}{\Rightarrow} tA \overset{+}{\Rightarrow} tuA \overset{+}{\Rightarrow} tuv \end{array}}$$

then it is also possible to derive *tv*, *tuuv* etc.

THE ROLE OF SELF-NESTED DERIVATIONS

$$\boxed{A \overset{+}{\Rightarrow} uAv \quad u \neq \varepsilon \wedge v \neq \varepsilon}$$

Self-nested derivations are

absent from unilinear grammars of regular languages
present in grammars of free nonregular languages
(palindromes, Dyck languages, language with equal exponents, ...)

lack of self-nested derivations allows one to transform unilinear grams. ⇒ regular expr.

This property holds in fact for any free grammar. Hence …

$$\boxed{\begin{array}{c}\text{GRAMMAR WITHOUT SELF-NESTED DERIVATIONS}\\ \Rightarrow \\ \text{REGULAR LANGUAGE}\end{array}}$$

NB: the converse does not necessarily hold:

a grammar with self-nested derivations may generate a regular language

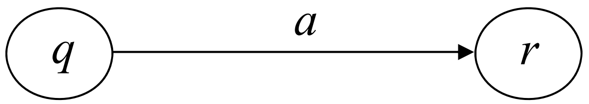
Example: G: $S \to a\, S\, a \mid \varepsilon$ $\quad L(S) = (aa)^*$

EQUIVALENCE OF FINITE STATE AUTOMATA AND UNILINEAR GRAMMARS

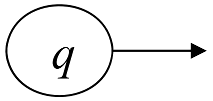
they define exactly the same languages

First we show how to derive an equivalent grammar from an automaton

The grammar: nonterminal symbols are the states $\boldsymbol{Q}$ of the automaton
axiom: the initial state q_0

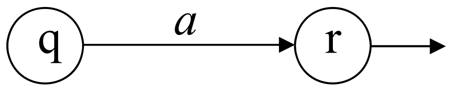


$\Longrightarrow$ $\boxed{q \to ar}$



(q final state) $\Longrightarrow$ $\boxed{\forall\ q \in F,\ q \to \varepsilon}$

If a non-nullable grammar is preferred:

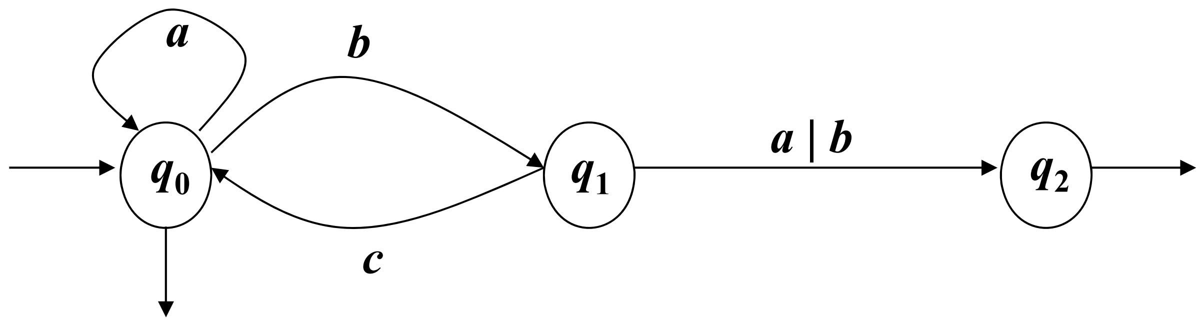


$\boxed{q \to ar \mid a}$

there is a one-to-one map: computations of the automaton $\leftrightarrow$ derivations of the grammar

string x is accepted by the automaton iff $\exists$ a derivation $q_0 \overset{*}{\Rightarrow} x$

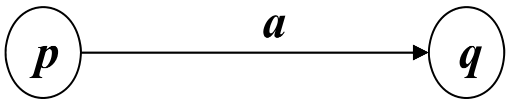
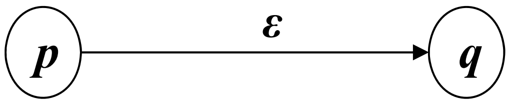
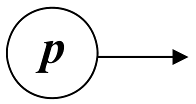
Example



$$\boxed{\begin{array}{l} q_0 \to aq_0 \mid bq_1 \mid \varepsilon \\ q_1 \to cq_0 \mid aq_2 \mid bq_2 \\ q_2 \to \varepsilon \end{array}}$$

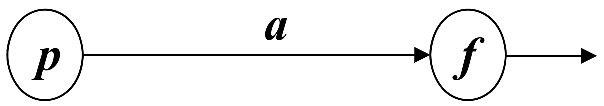
$$\boxed{\begin{array}{l} \text{derivation of string } bca \\ q_0 \Rightarrow bq_1 \Rightarrow bcq_0 \Rightarrow bcaq_0 \Rightarrow bca\varepsilon = bca \end{array}}$$

REVERSE TRANSFORMATION: GRAMMAR ⇒ AUTOMATON

right-linear grammar	*finite automaton*
1. Nonterm. alphabet ***V***	set of states $\boldsymbol{Q} = \boldsymbol{V}$
2. axiom S	initial state $q_0 = \text{S}$
3. $\boxed{p \to aq,\ a \in \Sigma \text{ and } p, q \in V}$	
4. $\boxed{p \to q \text{ where } p, q \in V}$	
5. $\boxed{p \to \varepsilon}$	final state 

notice that in general the automaton will be nondeterministic

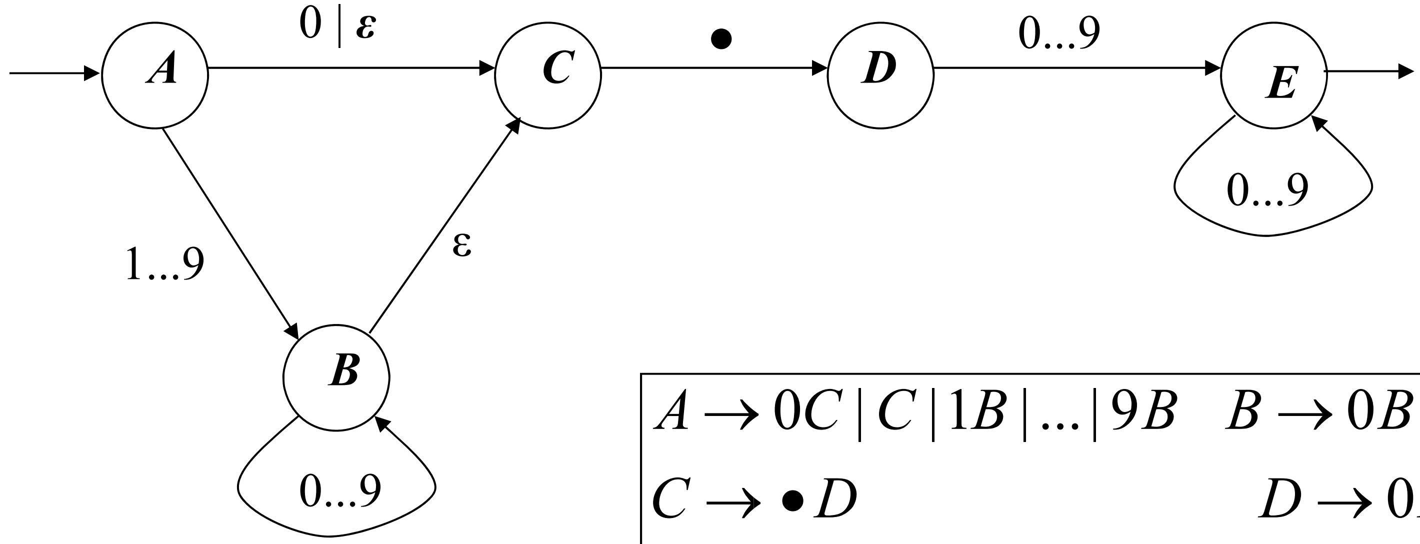
If the grammar includes terminal rules of type $\boldsymbol{p \to a,\ a \in \Sigma}$,
then the automaton will contain an additional state ***f***, with ***f*** final, and the transition:



DERIVATION OF THE GRAMMAR $\Leftrightarrow$ COMPUTATION OF THE AUTOMATON
therefore the two models define the same language

$\Rightarrow$ A LANGUAGE IS ACCEPTED BY A FINITE AUTOMATON IF AND ONLY IF IT IS GENERATED BY A UNILINEAR GRAMMAR

Example – Equivalence right-linear grammars – finite automata



$$A \to 0C \mid C \mid 1B \mid \ldots \mid 9B \quad B \to 0B \mid \ldots \mid 9B \mid C$$
$$C \to \bullet D \qquad D \to 0E \mid \ldots \mid 9E$$
$$E \to 0E \mid \ldots \mid 9E \mid \varepsilon \quad \text{axiom } A$$

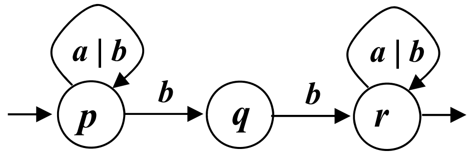
We can extend the analogy between finite state automata and unilinear grammars

DEFINITION: an automaton is ***ambiguous*** if it accepts a sentence with distinct computations

one-to-one match between computations (automata) and derivations (grammars) ⇒
an automaton is ambiguous ⇔ corresponding right-linear grammar is ambiguous

NB: for (right) linear grammars there is a 1-to-1 match between derivations and syntax trees
(because in the derivation the phrase forms include only one nonterminal)

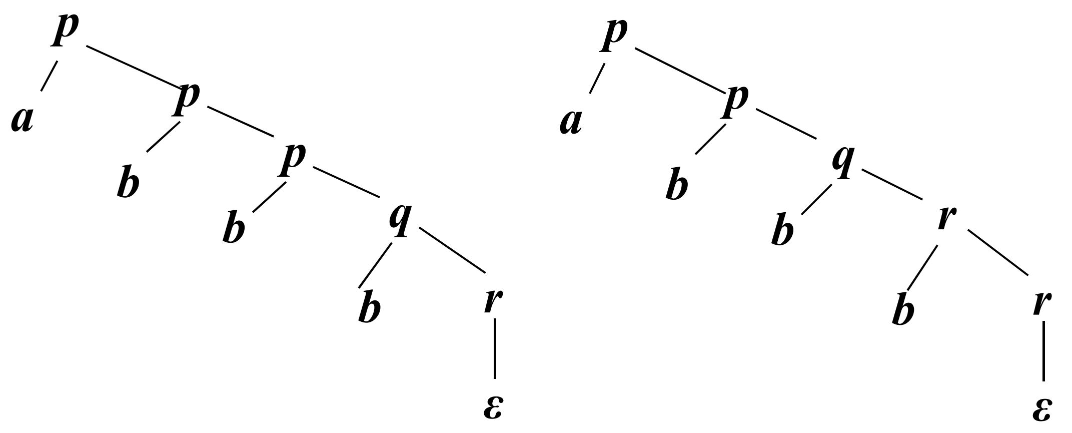
Example – A FSA modeling search of string *bb* in a text: the text is accepted iff in contains *bb*



$p \to ap \mid bp \mid bq$
$q \to br$
$r \to ar \mid br \mid \varepsilon$

(a | b)* bb (a | b)*

search of the string *bb* in the text *abbb*



FROM AUTOMATA TO REGULAR EXPRESSIONS DIRECTLY THE BMC (Brzozowski & McCluskey) METHOD

Assumptions

- unique initial state *i* (initial) without incoming arcs, and
- unique final state *t* (terminal) without outgoing arcs

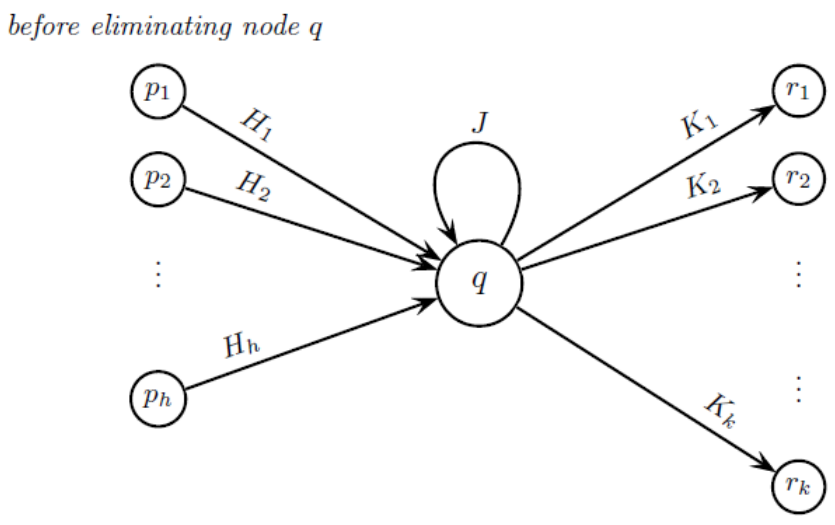
(otherwise: add initial and final states connected through suitable spontaneous moves)

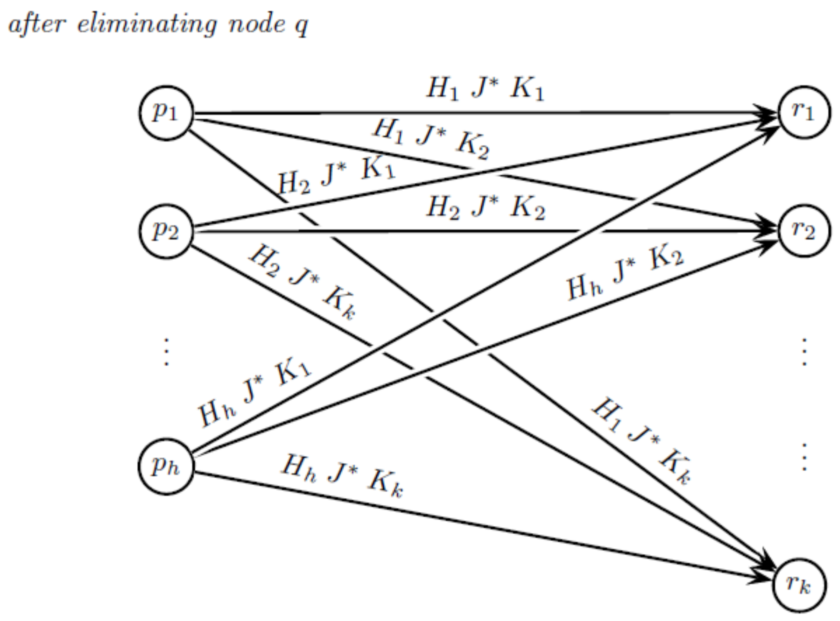
states different from *i* and *t* are called ***internal***

a new **generalized automaton** is built:
equivalent to the initial one, but with arcs labeled by regular expressions

internal states are progressively eliminated
compensating moves are added, ***labeled by r.e.*** that preserve the accepted language
until only states *i* and *t* are left

then the r.e. *e* labeling the transition $i \xrightarrow{e} t$ is the e.r. of the language



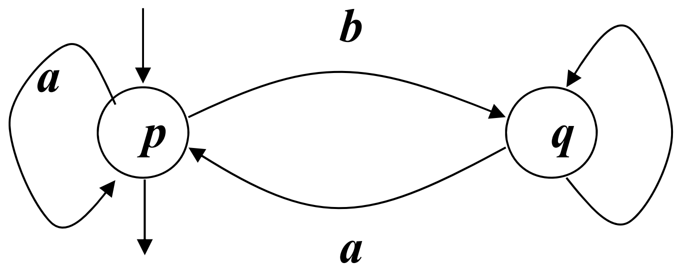


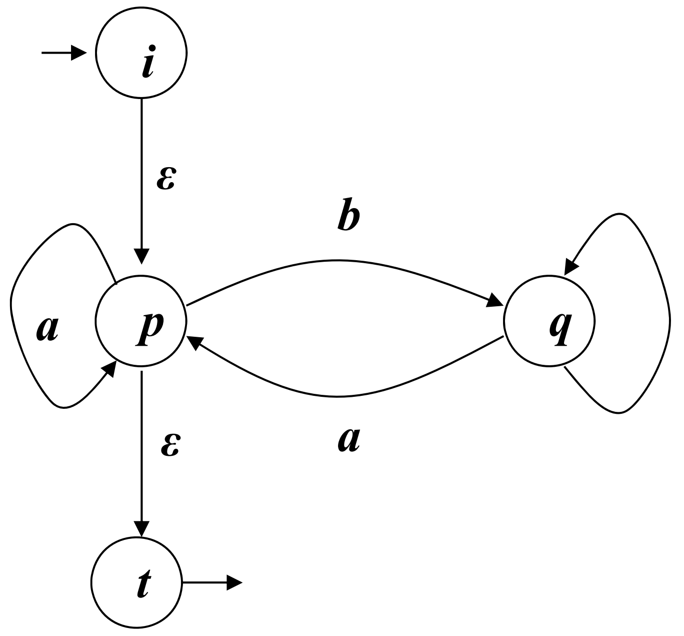
for each pair of states $\boldsymbol{p_i\ r_j}$ a compensating transition: $p_i \xrightarrow{H_i J^* K_j} r_j$ (NB: possibly, $p_i = r_j$)

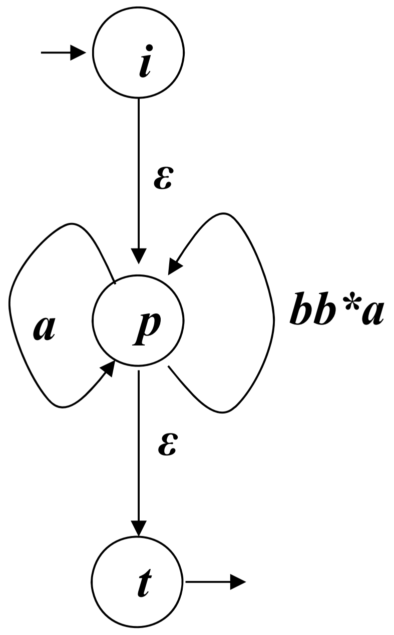
any resulting «parallel» transitions $p \overset{e_1}{\to} r$ and $p \overset{e_2}{\to} r$ «merged» into $p \xrightarrow{e_1 \mid e_2} r$

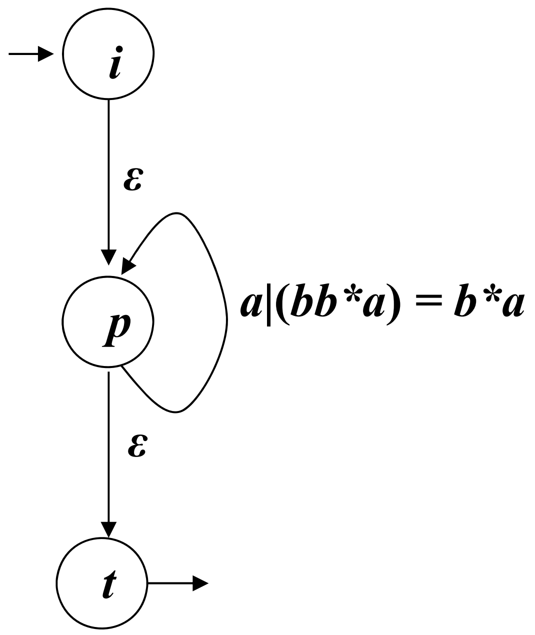
changing the order in the internal state elimination steps
leads to formally distinct, but equivalent regular expressions
pay attention when solving exercises: simplify r.e.'s whenever possible

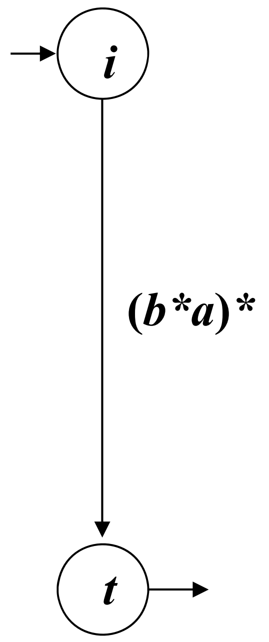
Example – application of BMC











ELIMINATION OF NONDETERMINISM

for efficiency, the final, implemented version of a finite recognizer must be deterministic

PROPERTY:
every nondeterministic automaton can be transformed into a deterministic one, and
(corollay) every unilinear grammar admits an equivalent nonambiguous grammar

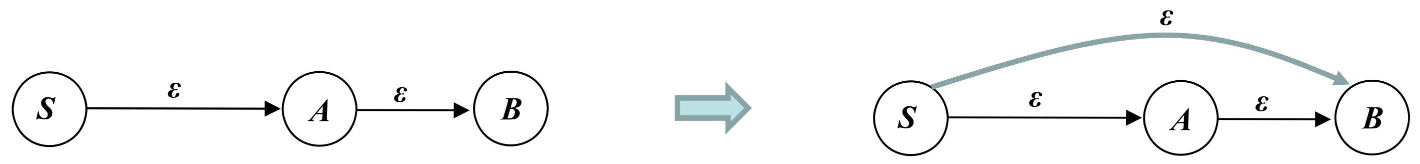
determinization of a finite automaton can be conceptually separated in two parts
(both steps can be replaced by the Berry-Sethi construction, to be presented later)

1. elimination of spontaneous moves: move sequences that include spontaneous moves are replaced by ***scanning moves*** (non-ε arcs)

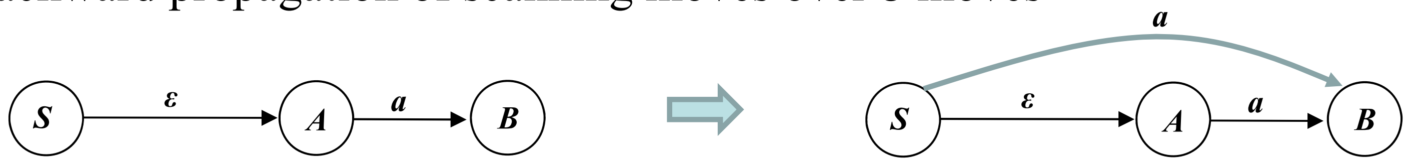
2. replacement of several nondeterministic (scanning) transitions by a single one (reachable/accessible subset construction – the new states are subsets of the initial state set): we do not cover that; see textbook §3.7.1 (also covered by previous courses)

First phase: elimination of ε-moves – a 4-steps procedure

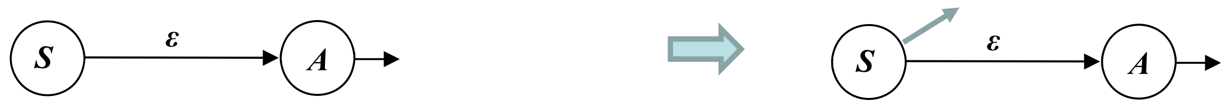
1: transitive closure of ε-moves



2: backward propagation of scanning moves over ε-moves



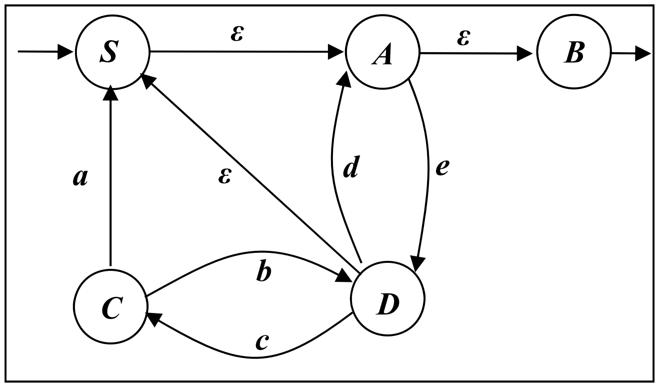
3: new final states: backward propagation of the «finality condition» for final states reached by ε-moves (antecedent states become also final)



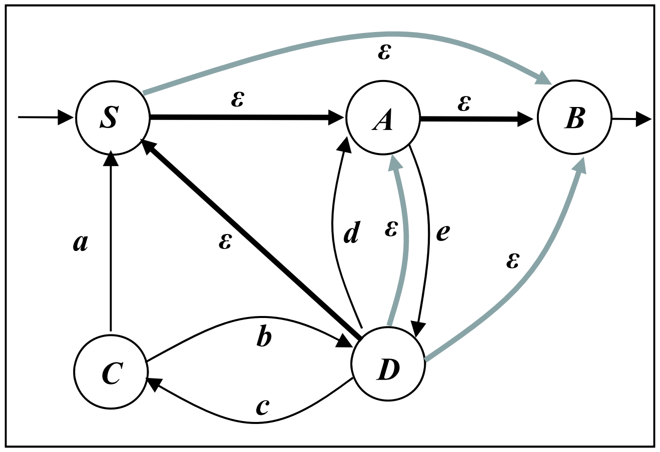
4: clean-up of ε-moves and of useless states

Comment: the purpose of steps 1-3 is to obtain an equivalent automaton where all ε-moves are redundant, so that they can be deleted without any consequence

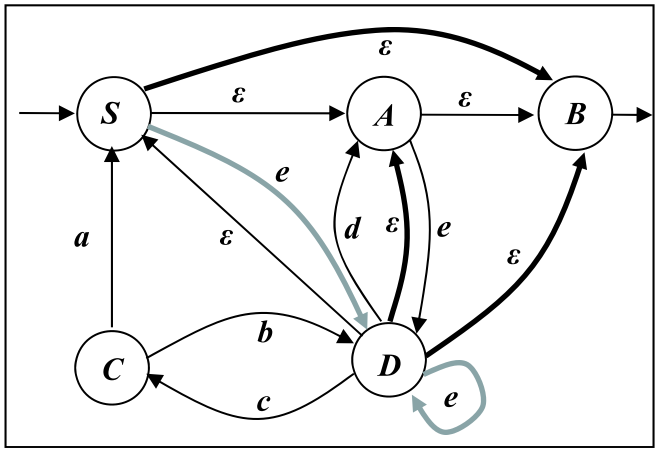
Example



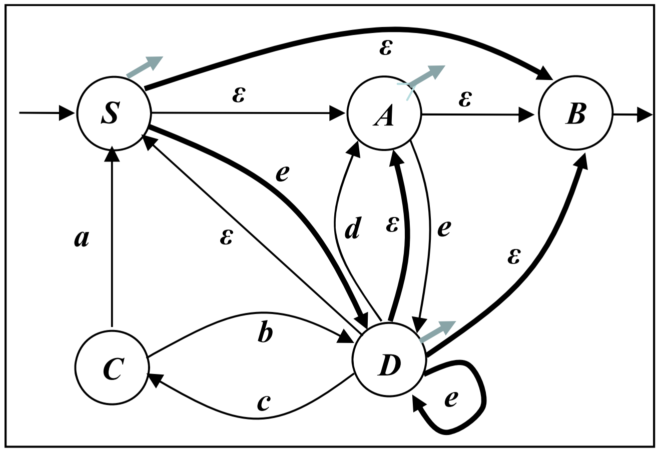
1



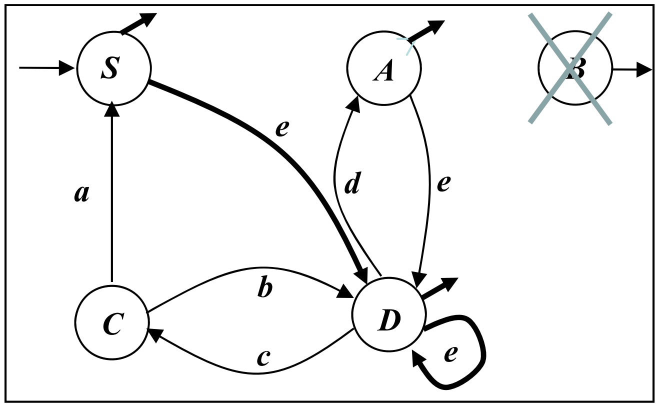
2



3



4



From Regular Expressions to Recognizing automata

Various algorithms, they differ in the kind of automaton and in the size of the result

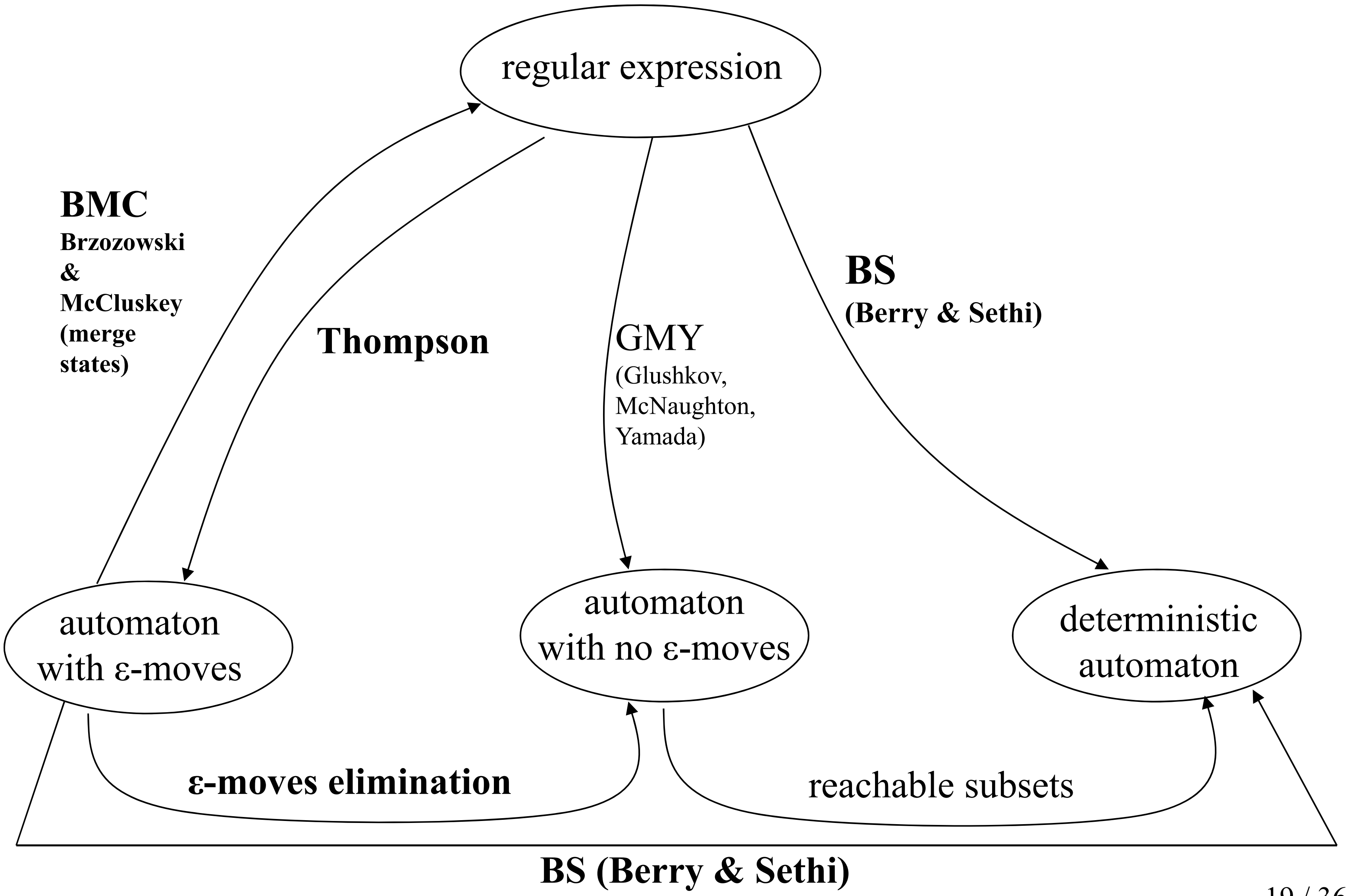
the textbook reports three methods (we discuss only (1) and (3)):

1) THOMPSON (or structural) method
   - builds the recognizers of subexpressions
   - combines them through spontaneous moves
   - resulting automata have (several) **ε-moves** and are in general **nondeterministic**

2) GLUSHKOV, MC NAUGHTON and YAMADA (GMY) method
   - builds a **nondeterministic automaton having no spontaneous moves**
   - size is less than Thompson's

3) BERRY & SETHI (BS) method
   - builds a **deterministic** automaton
   - not necessarily minimal

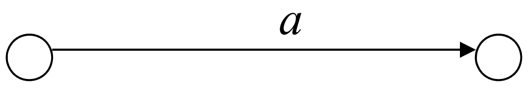
(1) and (2) can be combined with determinization algorithms,
(3) with minimization algorithms


regular expression
BMC
Brzozowski
&
McCluskey
(merge
states)
Thompson
GMY
(Glushkov,
McNaughton,
Yamada)
BS
(Berry & Sethi)
automaton
with ε-moves
automaton
with no ε-moves
deterministic
automaton
ε-moves elimination
reachable subsets
BS (Berry & Sethi)

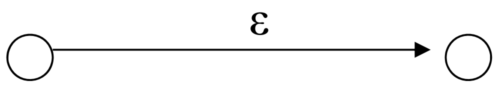
THOMPSON's STRUCTURAL METHOD

1) based on a systematic mapping between r.e. and recognizing automata (structural induction…)
2) every portion of automaton must have a unique initial and final state

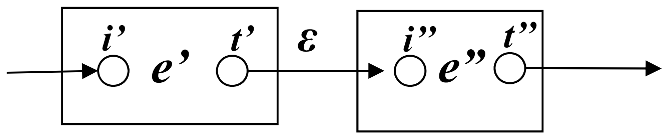
r.e. of type : a with $a \in \Sigma$



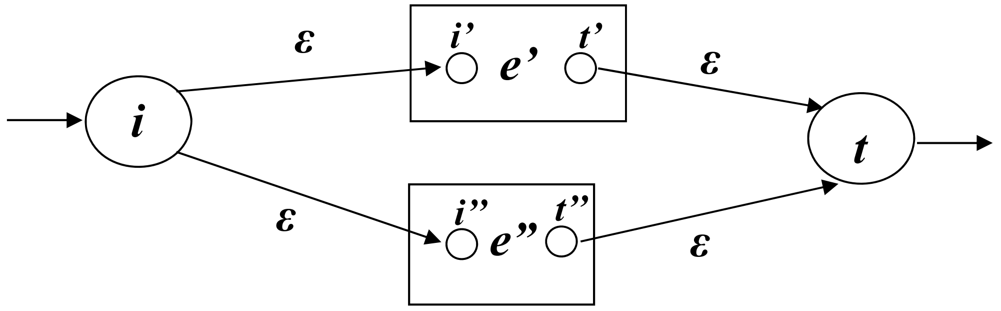
r.e. of type: ε



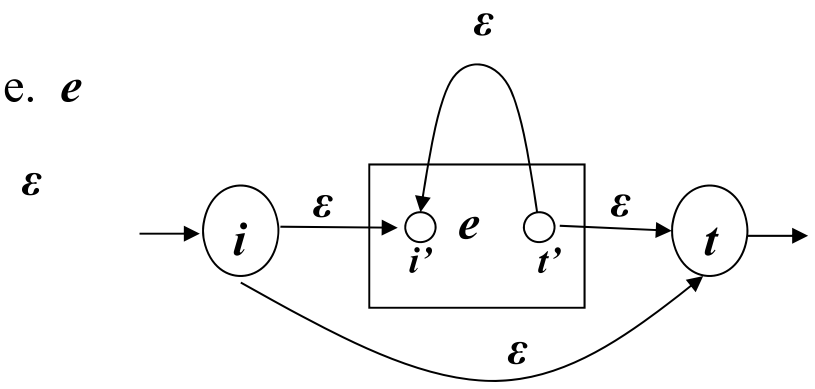
concatenation **$e' \cdot e''$** of two r.e. e' and e''



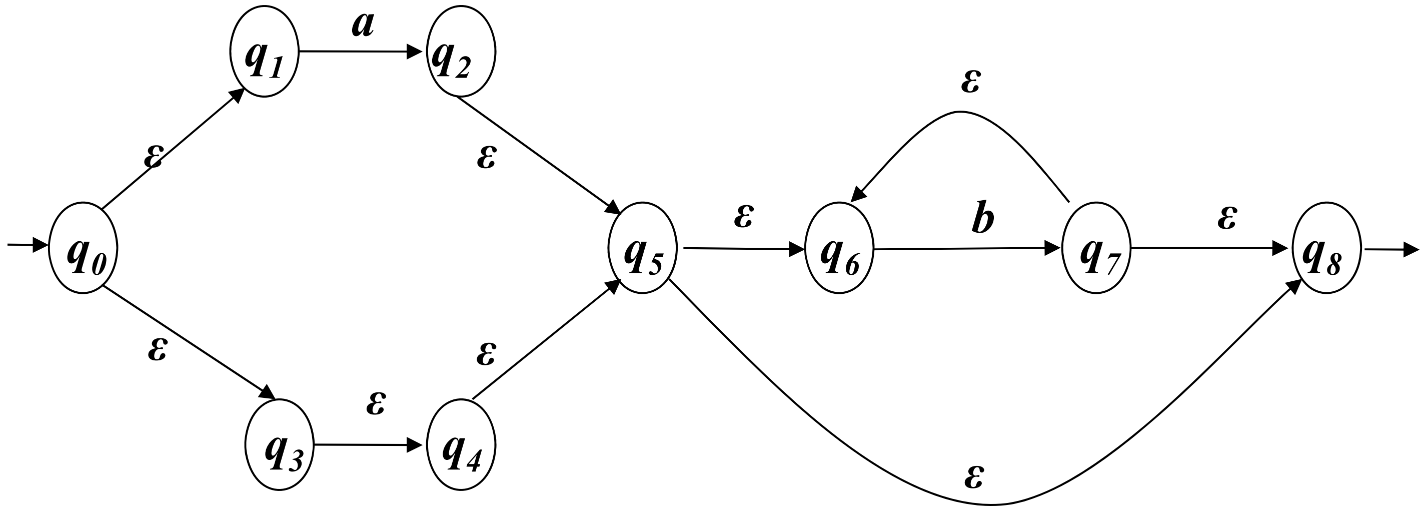
Union of two r.e. *e*’ and *e*”



Star ***e**** of a r.e. ***e***



Example $\boxed{(a \cup \varepsilon).b^*}$


$$\begin{tikzcd}
 & q_1 \arrow[r, "a"] & q_2 \arrow[dr, "\varepsilon"] & & & & \\
\arrow[r] & q_0 \arrow[u, "\varepsilon"] \arrow[d, "\varepsilon"'] & & q_5 \arrow[r, "\varepsilon"] \arrow[rrr, bend right, "\varepsilon"'] & q_6 \arrow[r, "b"] & q_7 \arrow[l, bend right, "\varepsilon"'] \arrow[r, "\varepsilon"] & q_8 \arrow[r] & {} \\
 & q_3 \arrow[r, "\varepsilon"] & q_4 \arrow[ur, "\varepsilon"'] & & & &
\end{tikzcd}$$

BERRY-SETHI DETERMINISTIC RECOGNIZER

Preliminarily, we introduce the ***local sets*** of a language L
(we focus on regular languages)
given a language L of alphabet Σ, define (assuming $\boldsymbol{a, b} \in \Sigma$ and $\boldsymbol{x, y} \in \Sigma^*$)

set of Initial chars	$Ini(L) = \{ a \mid ax \in L \}$	chars ***starting*** any string $\in L$
set of Final chars	$Fin(L) = \{ a \mid xa \in L \}$	chars ***ending*** any string $\in L$
set of Followers of char a	$Fol(a) = \{ b \mid xaby \in L \}$	chars ***following*** a in any string $\in L$

Example: $L = (a \mid bc)^* \, (de)^+ f$

Note the interaction of the r.e. operators ‘|’ ‘·’ ‘*’ ‘+’

$Ini(L) = \{ a, b, d \}$

$Fin(L) = \{ f \}$

$Fol(a) = \{ a, b, d \}$ $\quad Fol(b) = \{ c \}$ $\quad Fol(c) = \{ a, b, d \}$

$Fol(d) = \{ e \}$ $\quad Fol(e) = \{ d, f \}$ $\quad Fol(f) = \varnothing$

BERRY-SETHI DETERMINISTIC RECOGNIZER

(we only illustrate it: see textbook, §3.8.2.4 for a complete explanation/justification)

Let e the starting r.e. (of alphabet Σ): Ex. $e = (a \mid bb)^* (ac)^+$

e' its ***numbered version*** (of alphabet Σ_N): Ex. $e' = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+$

We consider expression $e' \dashv$ which includes the end-of-text mark $\dashv$

and use its local sets to build the BS automaton

Ex.: for $e' \dashv = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+ \dashv$ we have

$Ini(e' \dashv) = \{ a_1, b_2, a_4 \}$ $\quad$ $Fin(e' \dashv) = \{ \dashv \}$

$Fol(a_1)=\{a_1, b_2, a_4\}$ $\quad$ $Fol(b_2)=\{b_3\}$ $\quad$ $Fol(b_3)=\{a_1, b_2, a_4\}$

$Fol(a_4)=\{c_5\}$ $\quad$ $Fol(c_5)=\{a_4, \dashv\}$

In general, for r.e. $e' \dashv$, $\dashv \in Fol(a)$ for every $a \in Fin(e')$

Construction of the deterministic recognizer of Berry-Sethi

Every state is (corresponds to) a subset of $\Sigma_N \cup \{\dashv\}$

It contains the symbols that one can ***expect as next input***

Therefore final states are those that include (possibly among others) the end-mark $\dashv$

The ***initial state*** is the set $\boldsymbol{Ini(e' \dashv)}$

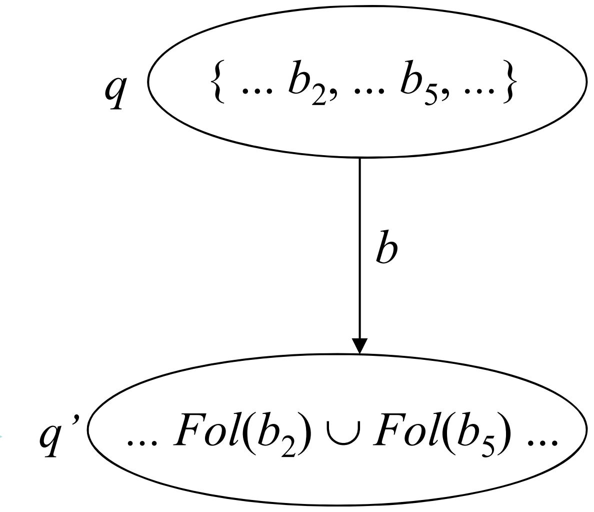
States are generated from the initial one, adding transitions and new states

State generation iterated until a fixed point is reached (nothing new can be generated)

During construction the transition function δ is viewed as a set of transitions $q \xrightarrow{a} q'$

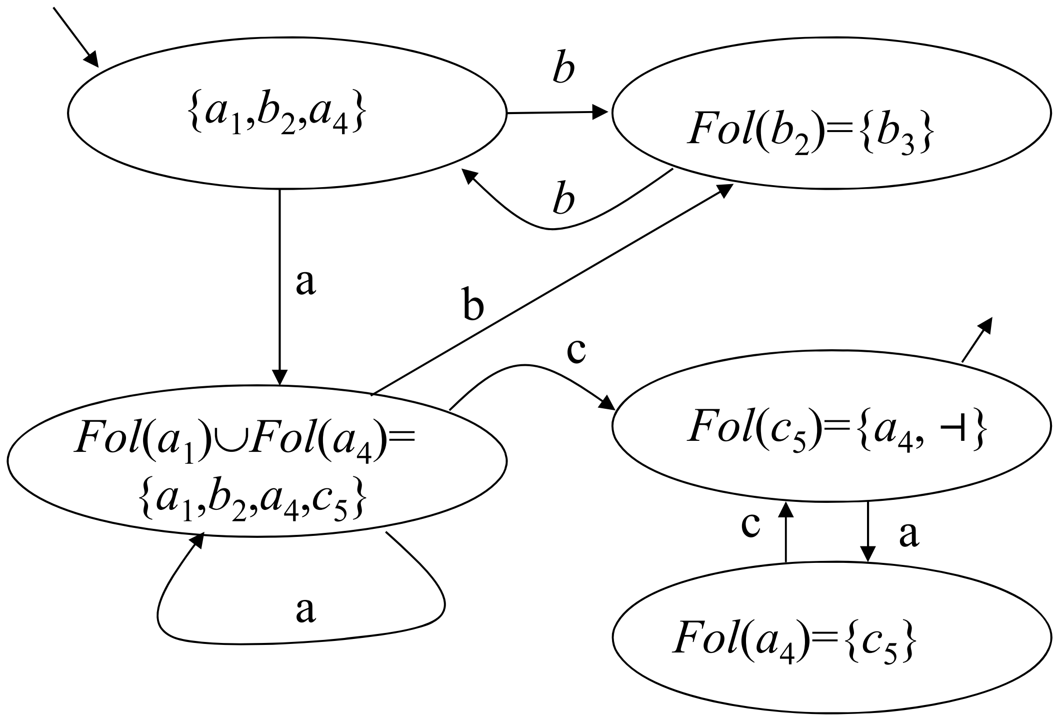
BS ALGORITHM

```
$q_0 := Ini(e' \dashv)$ ;    mark $q_0$ as *not visited*
$Q := \{q_0\}$
$\delta := \varnothing$
**while** there exists in $Q$ a non-visited state $q$ **do**
    mark $q$ as visited
    **for each** symbol $b \in \Sigma$ **do**
        $\boxed{q' := \bigcup_{\forall b_i \in q} Fol(b_i)}$
        **if** $q' \neq \varnothing$ **then**
            **if** $q' \notin Q$ **then**
                mark $q'$ as *not visited*
                $Q := Q \cup \{ q' \}$
            **end if**
            $\delta := \delta \cup \{ q \xrightarrow{b} q' \}$
        **end if**
    **end for**
**end while**
```



q
$\{ \dots b_2, \dots b_5, \dots \}$
b
q'
$\dots Fol(b_2) \cup Fol(b_5) \dots$

Example



$\{a_1,b_2,a_4\}$
$Fol(b_2)=\{b_3\}$
$Fol(a_1)\cup Fol(a_4)= \{a_1,b_2,a_4,c_5\}$
$Fol(c_5)=\{a_4, \dashv\}$
$Fol(a_4)=\{c_5\}$
b
b
a
b
c
a
c
a

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

$$e'\dashv = (a_1 \mid b_2b_3)^* \ (a_4c_5)^+ \dashv$$

$$Ini(e'\dashv) = \{ a_1, b_2, a_4 \}$$

x	$Fol(x)$
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	$a_4, \dashv$

the resulting automaton is deterministic
but it can be ***non*-minimal**
(because of the numbering of symbols)

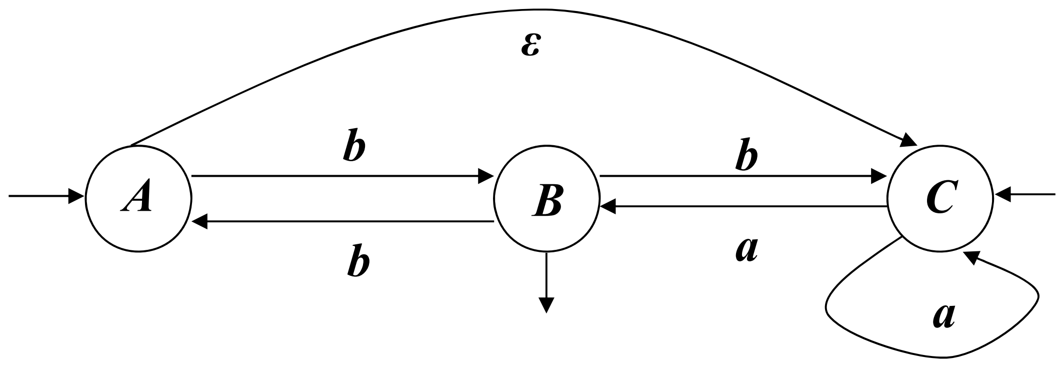
USING THE BS ALGORITHM FOR AUTOMATA DETERMINIZATION

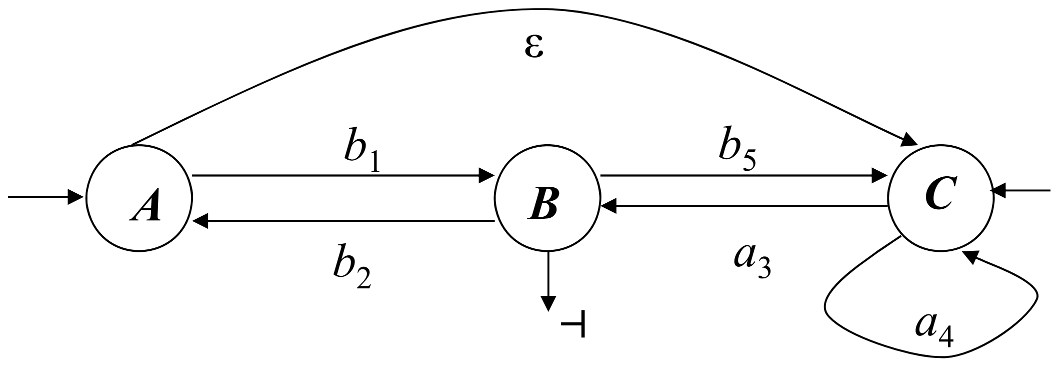
BS algorithm used for determinizing a nondeterministic automaton *N* possibly having ε-arcs

1. number the non-ε arcs of ***N***, obtaining a numbered version ***N’*** ; add an endmark ‘⊣’ on darts exiting the final states
2. compute for ***N’*** the local sets *Ini* and *Fol* (using rules similar to those for a r.e.)
3. apply the BS construction, thus obtaining an automaton ***M***

the resulting automaton is deterministic, though possibly non-minimal

Example: a nondet. FSA, with one ε-move and 2 initial states

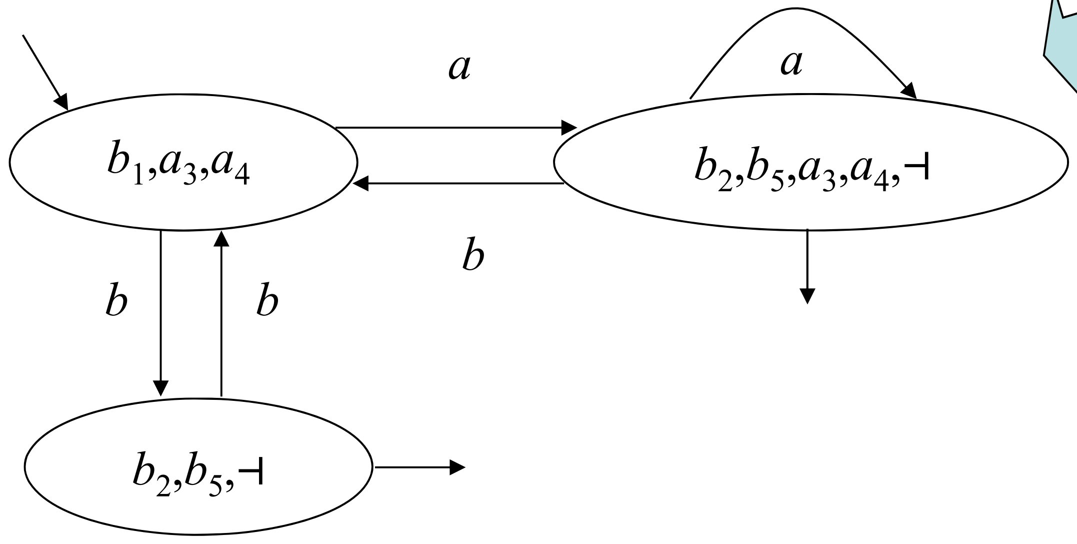




$Ini(L(N')\text{-|}) = \{b_1, a_3, a_4\}$

$\varepsilon a_3 = a_3, \varepsilon a_4 = a_4$

x	$Fol(x)$
b_1	$b_2, b_5, \text{-|}$
b_2	b_1, a_3, a_4
a_3	$b_2, b_5, \text{-|}$
a_4	a_3, a_4
b_5	a_3, a_4



REGULAR EXPRESSIONS WITH (1) COMPLEMENT AND (2) INTERSECTION

both topics covered by previous courses so we go through them quickly

(1) CLOSURE OF REG UNDER COMPLEMENT AND INTERSECTION

$$\boxed{\text{If } L, L', L'' \in REG \text{ then } \neg L \in REG \quad L' \bigcap L'' \in REG}$$

$\boldsymbol{L \in REG \;\Rightarrow\; \neg L \in REG}$

proved by constructing the recognizer for the complement language $\boldsymbol{\neg L = \Sigma^* \setminus L}$

starting from a ***deterministic*** recognizer M for L

ALGORITHM: construction of the deterministic recognizer ***M'*** for the complement

We extend $\boldsymbol{M=<Q, \Sigma, \delta, q_0, F>}$ with the ***error*** or ***sink*** state $p \notin \boldsymbol{Q}$ and the arcs to and from it

1. $Q' = Q \cup \{p\}$
2. $\delta'(q, a) = \delta(q, a)$ if $\delta(q, a)$ is defined, otherwise $\delta'(q, a)=p$; $\delta'(p, a)=p \;\; \forall a \in \Sigma$
3. Switch final and non-final states: $\boldsymbol{F' = (Q \setminus F) \cup \{ p \}}$

Example: automaton for the complement language

original automaton:

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, bend left, "a"] & q_1 \arrow[l, bend left, "b"] \arrow[r, bend left, "a"] & q_2 \arrow[l, bend left, "b"] \arrow[r] & {} \end{tikzcd}$$

automaton with the sink state added:

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, bend left, "a"] \arrow[dr, "b"'] & q_1 \arrow[l, bend left, "b"] \arrow[r, bend left, "a"] & q_2 \arrow[l, bend left, "b"] \arrow[dl, "a"] \arrow[r] & {} \\ & & p \arrow[loop below, "a|b"] & & \end{tikzcd}$$

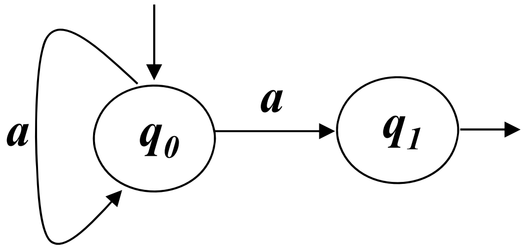
complement automaton
(final states switched):

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, bend left, "a"] \arrow[d] \arrow[ddr, "b"'] & q_1 \arrow[l, bend left, "b"] \arrow[r, bend left, "a"] \arrow[d] & q_2 \arrow[l, bend left, "b"] \arrow[ddl, "a"] \\ & {} & {} & \\ & & p \arrow[r] \arrow[loop below, "a|b"] & {} \end{tikzcd}$$

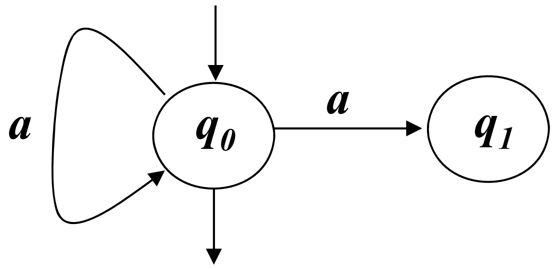
NB: the starting automaton ***M must be deterministic***
otherwise the language accepted by the complement automaton ***M'*** migh not be disjointed
and the obvious property $L \cap \neg L = \varnothing$ would be violated

If nondeterministic automaton has, for a string $x \in L$,
one accepting computation and a non-accepting one
The same holds in the complement automaton ***M'***, which then will also accept string x

Example:



original automaton ***M*** (***nondeterministic***)



(pseudo) complement automaton ***M'***

$a \in L(M)$, but M has two computations for a, one accepting and one rejecting
The pseudo complement automaton accepts string ***a***, that also belongs to the original language

(2) RECOGNIZER FOR THE INTERSECTION OF TWO REGULAR LANGUAGES

one could use the property of closure of REG under complement and union
to exploit the De Morgan identity $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$ and therefore:
- build the deterministic recognizers of L_1 and L_2
- derive those of the complement languages $\neg L_1$ and $\neg L_2$
- build the recognizer for the union (using the Thomson method)
- make the automaton deterministic
- derive the complement automaton

There is a more direct method

(CARTESIAN) PRODUCT AUTOMATON

Quite a common method:
Allows one to simulate the simultaneous execution of two automata

We assume the two automata without ε-moves but not necessarily deterministic

The state set of the product automaton ***M*** is the cartesian product of the state sets of *M'* and *M"*

A state is a pair ***< q', q">***, with ***q'*** $\in$ ***Q'*** and ***q"*** $\in$ ***Q"***

definition of transition function:

$$\boxed{< q', q" > \overset{a}{\rightarrow} < r', r" > \quad \text{if and only if} \quad q' \overset{a}{\rightarrow} r' \wedge q" \overset{a}{\rightarrow} r"}$$

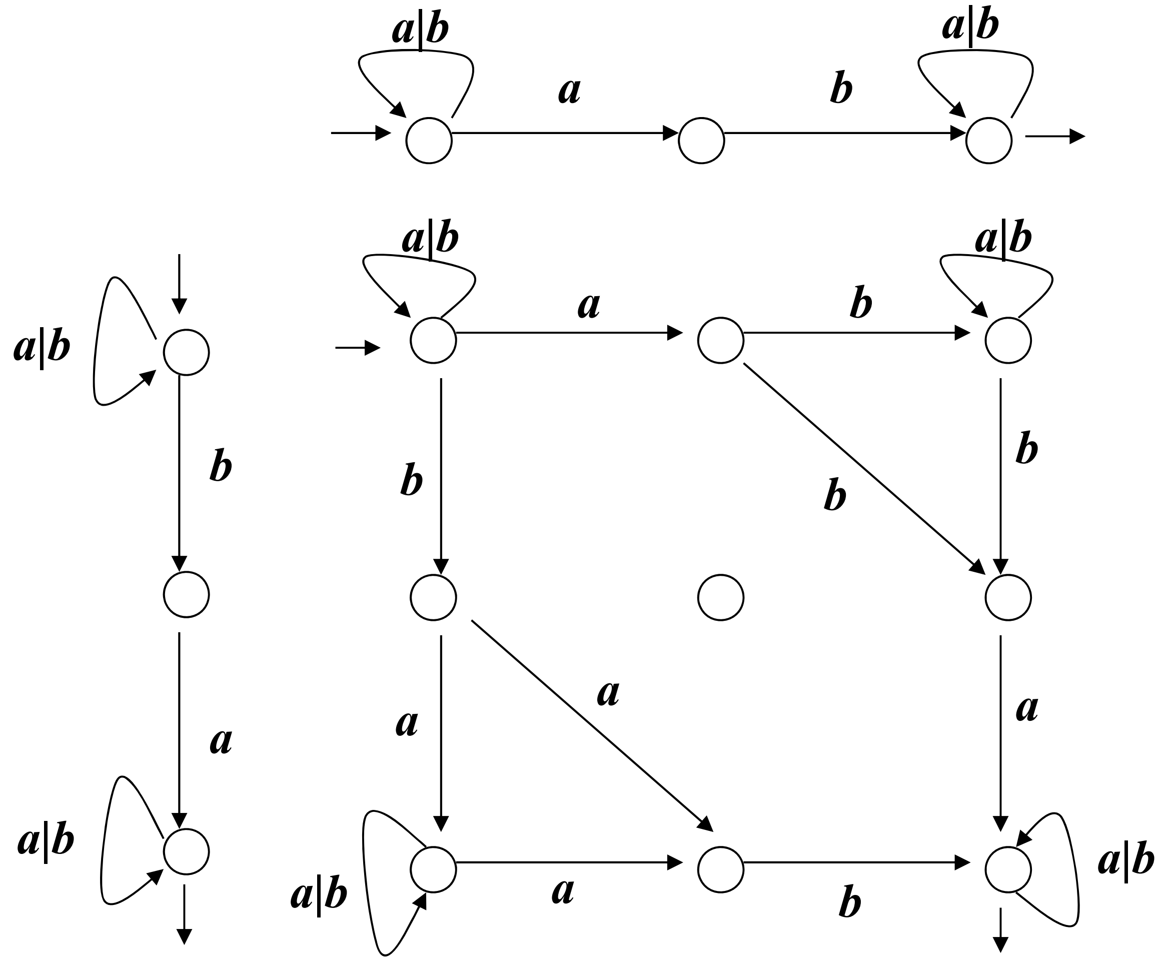
Initial states *I* of *M* are the cartesian product ***I = I'*** × ***I"***

Final states are the product ***F = F'*** × ***F"***

NOTE: The method can be applied to other set-theoretical operations (e.g. for union: final states of ***M*** are those state pairs where at least one of the two states is final)

Example – Intersection and product machine for the two languages of strings containing, respectively, substring ***ab*** and ***ba***

$$\boxed{\begin{aligned} L' &= (a\mid b)^{*}ab(a\mid b)^{*} \\ L'' &= (a\mid b)^{*}ba(a\mid b)^{*} \end{aligned}}$$



a|b
a
b
a|b
a|b
a|b
a|b
a
b
b
b
b
b
a
a
a
a
a|b
a|b
a
b
a|b