Conversion between Regexp and Finite State Automaton

Translated and adapted by L. Breveglieri

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*

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

end set – *Finals*

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

adjacency set – *Digrams*

$$\boxed{Dig(L) = \left\{\, x \in \Sigma^2 \mid \; \Sigma^*\, x\, \Sigma^* \;\cap\; L \neq \emptyset \,\right\}}$$

and its complement set (*forbidden digrams*)

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

EXAMPLE – a locally testable language L_1

$$\boxed{L_1 = (a\,b\,c)^*}$$

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

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

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

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

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

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

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

$$\boxed{L_3 = b\ (a\ a)^+\ b}$$

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

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

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

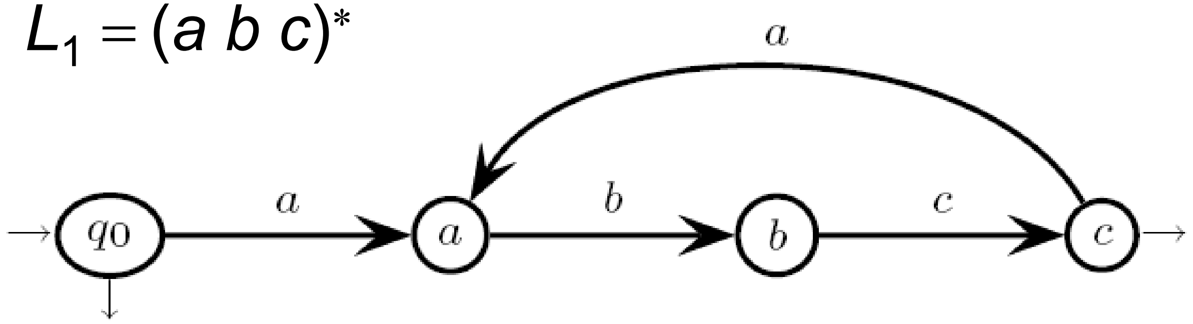
CONSTRUCTION OF THE RECOGNIZER OF THE LOCAL LANGUAGE

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

initial states	$\boxed{q_0 \cup \Sigma}$	
final states	$\boxed{Fin}$	
transitions	$\boxed{q_0 \xrightarrow{a} a \ \ if\ a \in Ini}$	$\boxed{a \xrightarrow{b} b \ \ if\ a\, b \in Dig}$

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

COMPOSITION OF LOCAL LANGUAGES OVER DISJOINT ALPHABETS

The basic language operations preserve local languages, provided that such languages are defined over disjoint alphabets

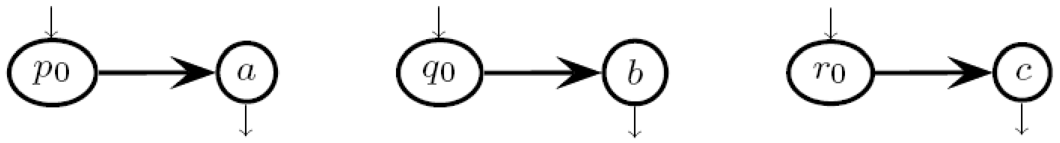
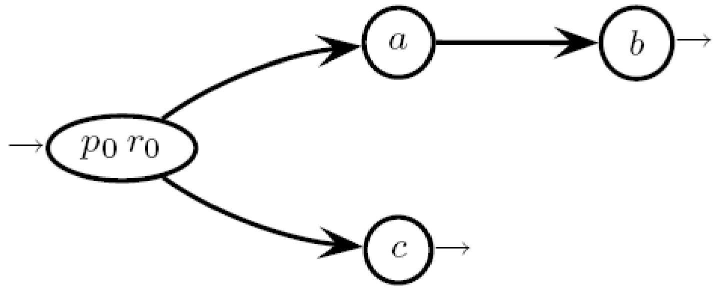
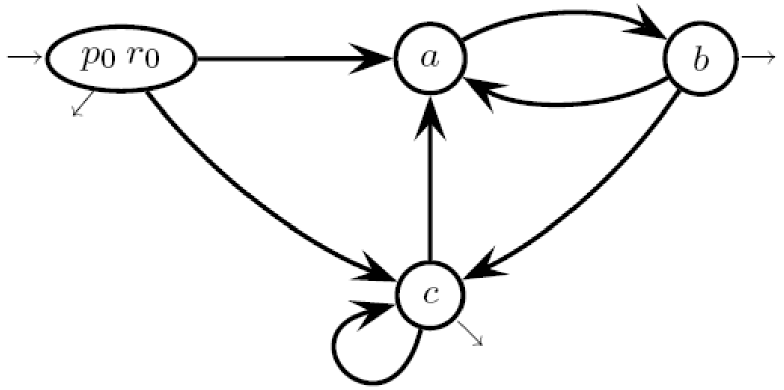
Given any two local languages *L*’ and *L*’’ over disjoint alphabets, the union, concatenation and star (or cross) languages are local as well

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

concatenate characters a and b
unite subexpression $a \cdot b$ to character c
apply star to subexpression $a \cdot b \cup c$

See the details of the construction on the text. Next the example above

EXAMPLE (a b U c)*

subexpression	*component automata*
atomic elements $a \quad b \quad c$	
concatenation and union $a \cdot b \cup c$	
(Kleene) star $(a \cdot b \cup c)^*$	

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 *Null* (*e*) = *true* **then** $\varepsilon \in L(e)$

if *Null* (*e*) = *false* **then** $\varepsilon \notin L(e)$

$$\begin{aligned}
Null(\emptyset) &= false \\
Null(\varepsilon) &= true \\
Null(a) &= false \quad \text{for every character } a \\
Null(e \cup e') &= Null(e) \vee Null(e') \\
Null(e \cdot e') &= Null(e) \wedge Null(e') \\
Null(e^*) &= true \\
Null(e^+) &= Null(e)
\end{aligned}$$

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

EXAMPLE

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

set of initials

$$Ini(\emptyset) = \emptyset$$
$$Ini(\varepsilon) = \emptyset$$
$$Ini(a) = \{ a \} \quad \text{for every character } a$$
$$Ini(e \cup e') = Ini(e) \cup Ini(e')$$
$$Ini(e \cdot e') = \textbf{if } Null(e) \textbf{ then } Ini(e) \cup Ini(e') \textbf{ else } Ini(e) \textbf{ end if}$$
$$Ini(e^*) = Ini(e^+) = Ini(e)$$

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

set of finals

$$Fin(\emptyset) = \emptyset$$
$$Fin(\varepsilon) = \emptyset$$
$$Fin(a) = \{ a \} \quad \text{for every character } a$$
$$Fin(e \cup e') = Fin(e) \cup Fin(e')$$
$$Fin(e \cdot e') = \textbf{if } Null(e') \textbf{ then } Fin(e) \cup Fin(e') \textbf{ else } Fin(e') \textbf{ end if}$$
$$Fin(e^*) = Fin(e^+) = Fin(e)$$

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

$$\begin{aligned}
Dig(\emptyset) &= \emptyset \\
Dig(\varepsilon) &= \emptyset \\
Dig(a) &= \emptyset \quad \text{for every character } a \\
Dig(e \cup e') &= Dig(e) \cup Dig(e') \\
Dig(e \cdot e') &= Dig(e) \cup Dig(e') \cup Fin(e) \cdot Ini(e') \\
Dig(e^*) = Dig(e^+) &= Dig(e) \cup Fin(e) \cdot Ini(e)
\end{aligned}$$

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

EXAMPLE

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

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

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

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

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

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

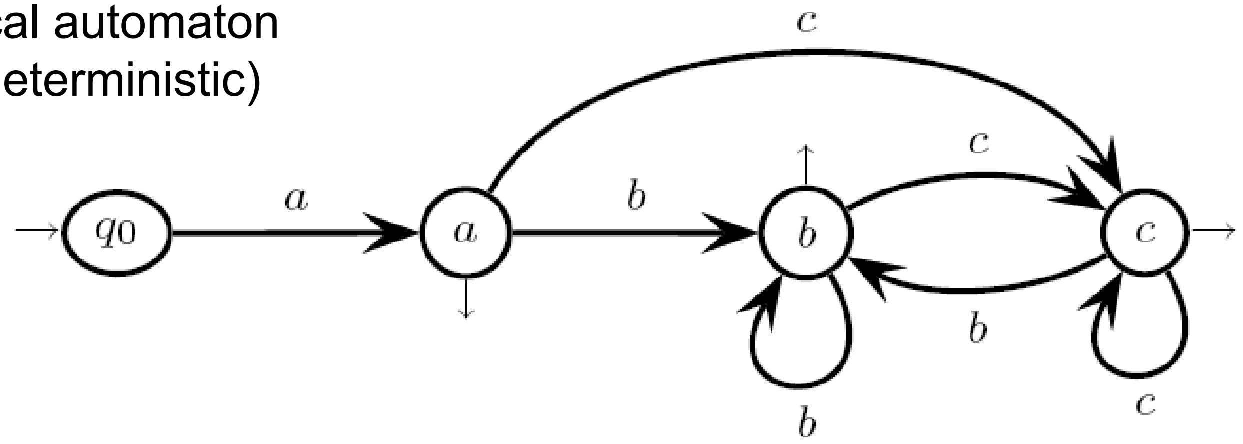
EXAMPLE – the recognizer of a linear regexp $a\,(b \cup c)^*$

$$\boxed{\mathit{Null}\,(\,a\,(b \cup c)^*\,) = \mathit{false}}$$

$$Ini = \{\, a \,\} \qquad Fin = \{\, a \,\} \cup \{\, b,\ c \,\} = \{\, a,\ b,\ c \,\}$$

$$Dig = \{\, b\,b,\ b\,c,\ c\,b,\ c\,c \,\} \cup \{\, a\,b,\ a\,c \,\} = \{\, a\,b,\ a\,c,\ b\,b,\ b\,c,\ c\,b,\ c\,c \,\}$$

local automaton
(deterministic)



$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "a"] & a \arrow[d] \arrow[r, "b"] \arrow[rr, bend left=50, "c"] & b \arrow[u] \arrow[r, bend left, "c"] \arrow[loop below, "b"] & c \arrow[r] \arrow[l, bend left, "b"] \arrow[loop below, "c"] & {} \\
& & {} & & &
\end{tikzcd}$$

HOW TO EXTEND FROM A LINEAR REGEXP TO A GENERIC ONE

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

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

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

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a_1"] \arrow[rrr, bend left, "a_3"] & a_1 \arrow[r, bend left, "b_2"] & b_2 \arrow[l, bend left, "a_1"] \arrow[r, "a_3"] & a_3 \arrow[r] & {}
\end{tikzcd}$$

equivalent to $e_\#$

denumbered automaton

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a"] \arrow[rrr, bend left, "a"] & a_1 \arrow[r, bend left, "b"] & b_2 \arrow[l, bend left, "a"] \arrow[r, "a"] & a_3 \arrow[r] & {}
\end{tikzcd}$$

equivalent to e

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

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

and *Fol* ($\dashv$) $= \Phi$

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

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

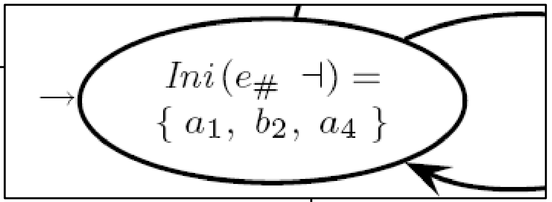
Each state is denoted by a subset of $\Sigma_{\#} \cup \dashv$

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

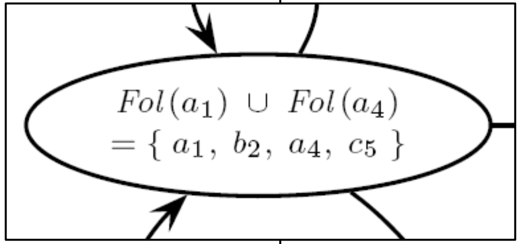
Set *Ini* ($e_{\#} \dashv$) is the initial state, final states contain $\dashv$, and the state set Q starts with the initial state

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

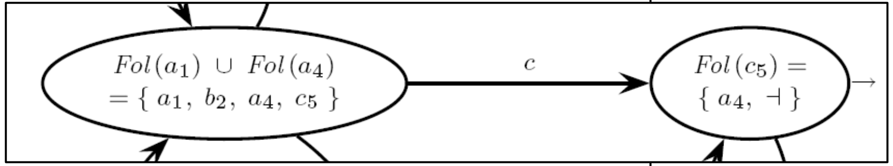
```
$q_0 := Ini(e_{\#} \dashv)$
$Q := \{ q_0 \}$
$\delta := \emptyset$
while ($\exists\, q \in Q$ s.t. $q$ is unmarked) do
    mark state $q$ as visited
    for (each character $c \in \Sigma$) do
        $q' := \bigcup_{\forall c_{\#} \in \Sigma_{\#} \text{ s.t. } c_{\#} \in q} Fol(c_{\#})$
        if ($q' \neq \emptyset$) then
            if ($q' \notin Q$) then
                set $q'$ as a new unmarked state
                $Q := Q \cup \{ q' \}$
            end if
            $\delta := \delta \cup \{ q \xrightarrow{c} q' \}$
        end if
    end for
end while □
```



for instance



for instance


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

for instance

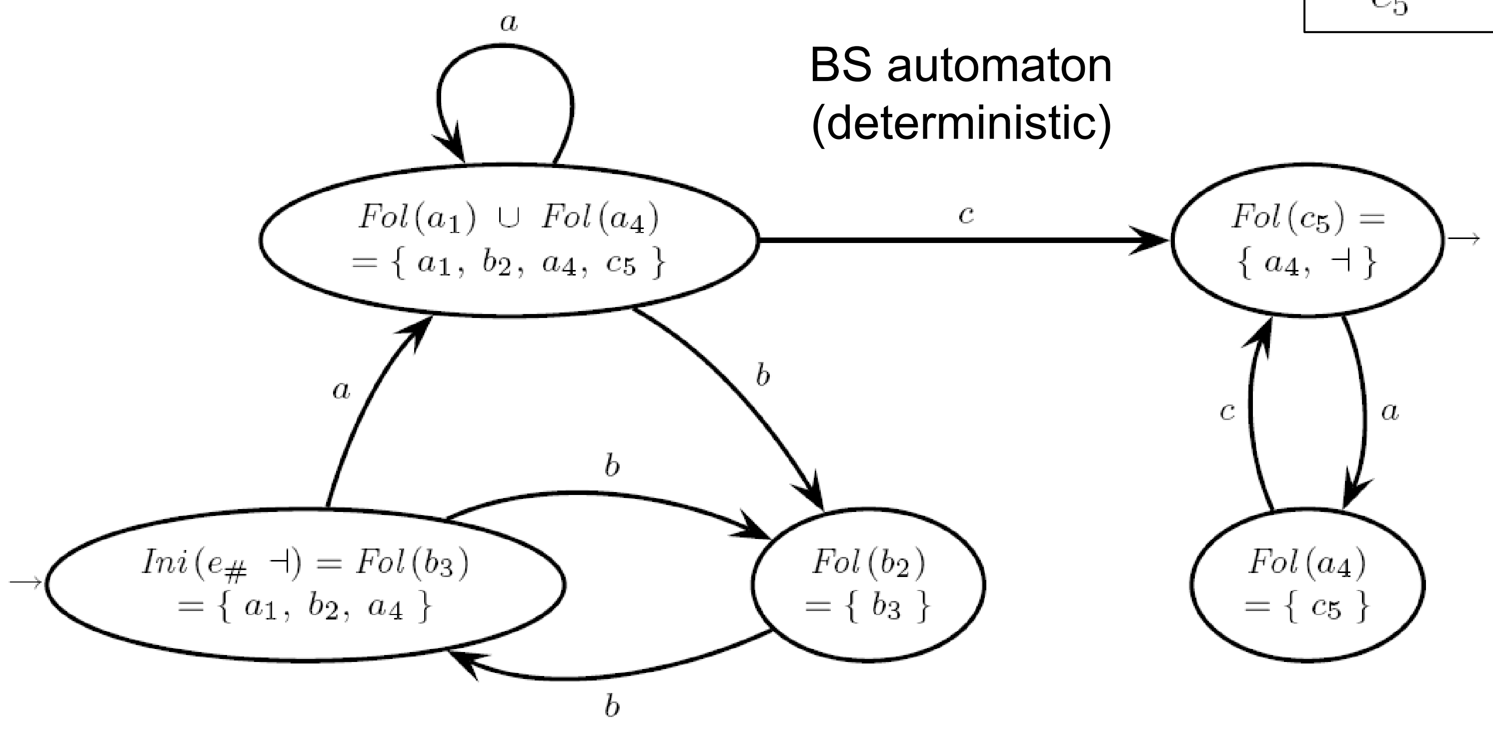
EXAMPLE $\boxed{e = (a \mid bb)^* (ac)^+}$

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

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

$c_\#$	$Fol(c_\#)$
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	$a_4, \dashv$

BS automaton (deterministic)

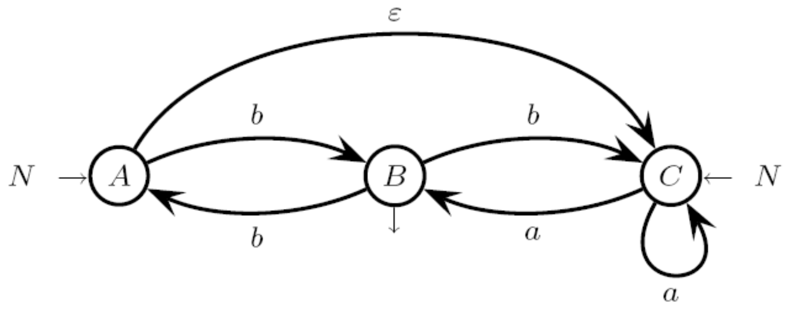


$Fol(a_1) \cup Fol(a_4) = \{ a_1, b_2, a_4, c_5 \}$
$Fol(c_5) = \{ a_4, \dashv \}$
$Ini(e_\# \dashv) = Fol(b_3) = \{ a_1, b_2, a_4 \}$
$Fol(b_2) = \{ b_3 \}$
$Fol(a_4) = \{ c_5 \}$
a
b
c

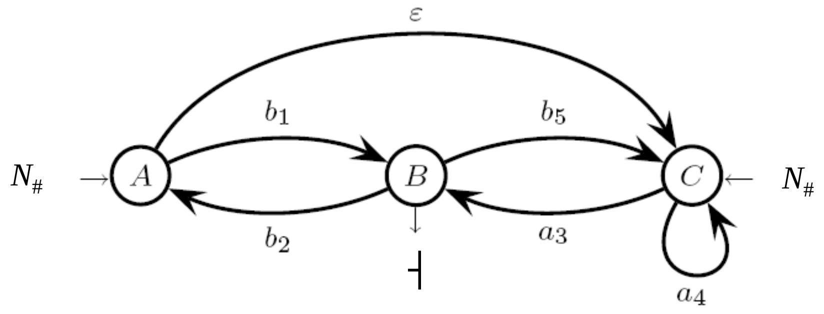
It may have equivalent states (minimize it)

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

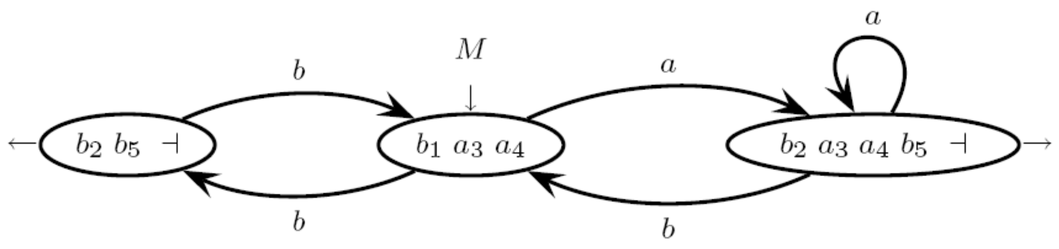
nondeterministic automaton



numbered automaton



deterministic 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)

$$Ini\left(L\left(N_\#\ \dashv\right)\right) = \{\ b_1,\ a_3,\ a_4\ \}$$

$c_\#$	$Fol(c_\#)$
b_1	$b_2,\ b_5,\ \dashv$
b_2	$b_1,\ a_3,\ a_4$
a_3	$b_2,\ b_5,\ \dashv$
a_4	$a_3,\ a_4$
b_5	$a_3,\ a_4$

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 coincise

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

$$\boxed{\textbf{if } L', L'' \in REG \textbf{ then } \neg L' \in REG \text{ and } L' \cap L'' \in REG}$$

DETERMINISTIC RECOGNIZER OF THE COMPLEMENT LANGUAGE

$$\boxed{\neg L = \Sigma^* \setminus L}$$

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 U { *p* }
2. the complement transition function is $\overline{\delta}$, see below
3. swap the non-final and final states, see below

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

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

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

$$\boxed{x \in L\,(M)}$$

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

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

THE INTERSECTION LANGUAGE OF REGULAR LANGUAGES IS REGULAR

Use the De Morgan theorem to reduce intersection to complement and union

$$\boxed{L \cap L' = \neg\,(\neg L \cup \neg L')}$$

NOTICE – there is a more diret construction (see next)

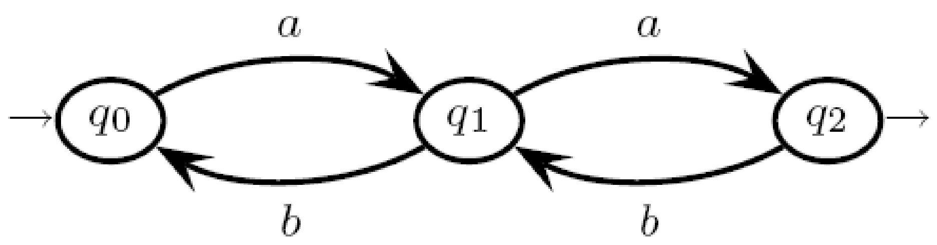
THE SET DIFFERENCE LANGUAGE OF REGULAR LANGUAGES IS REGULAR

Reduce set difference to complement and intersection

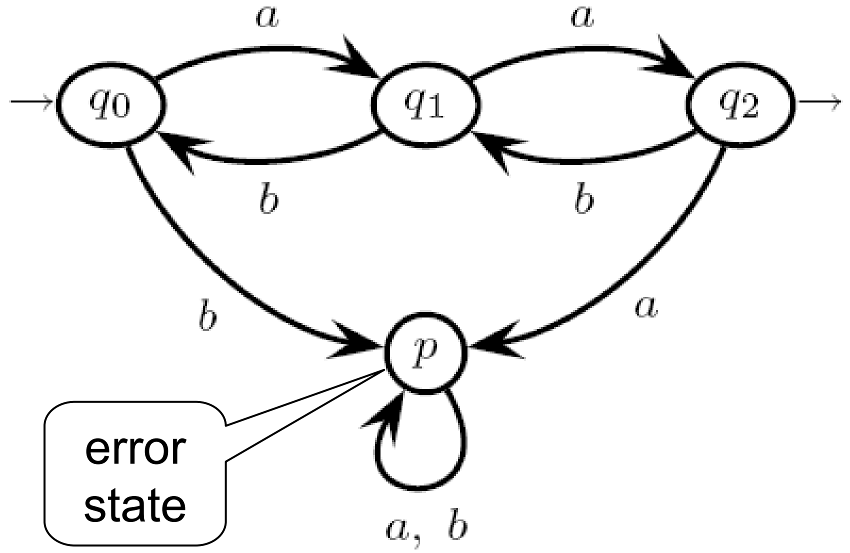
$$\boxed{L \setminus L' = L \cap \neg L'}$$

EXAMPLE – COMPLEMENT AUTOMATON

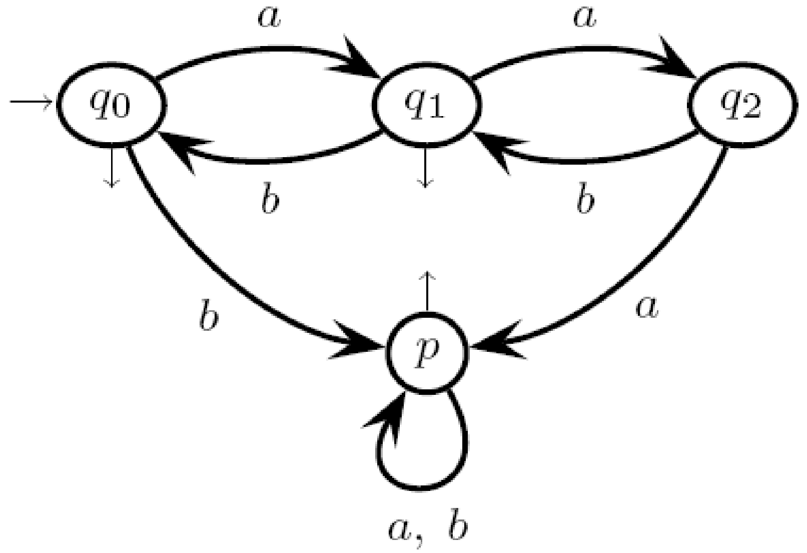
original automaton
(deterministic)



natural completion



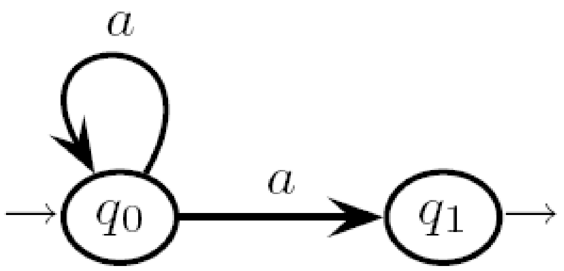
complement automaton



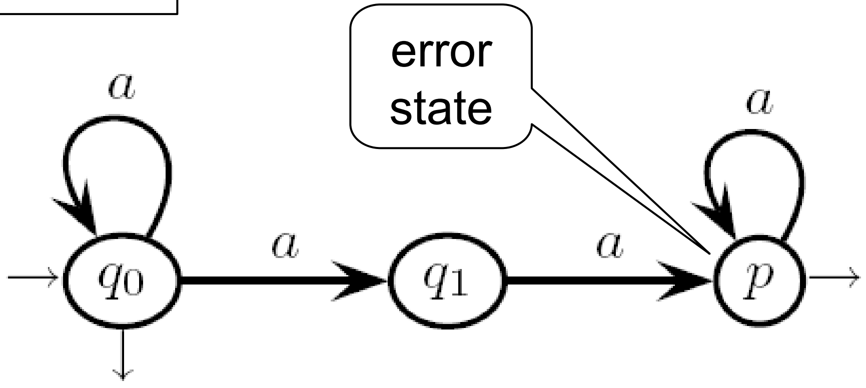
CAUTION: for the complement construction to work correctly, the original automaton must be deterministic, otherwise the original and complement languages may be not disjoint, which fact would be in violation of the complement definition (see below)

EXAMPLE

$$\boxed{L \bigcap \neg L = \varnothing}$$



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 ⟨*q*', *q*''⟩, where the left (right) member is a state of the first (second) machine. The move is

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

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

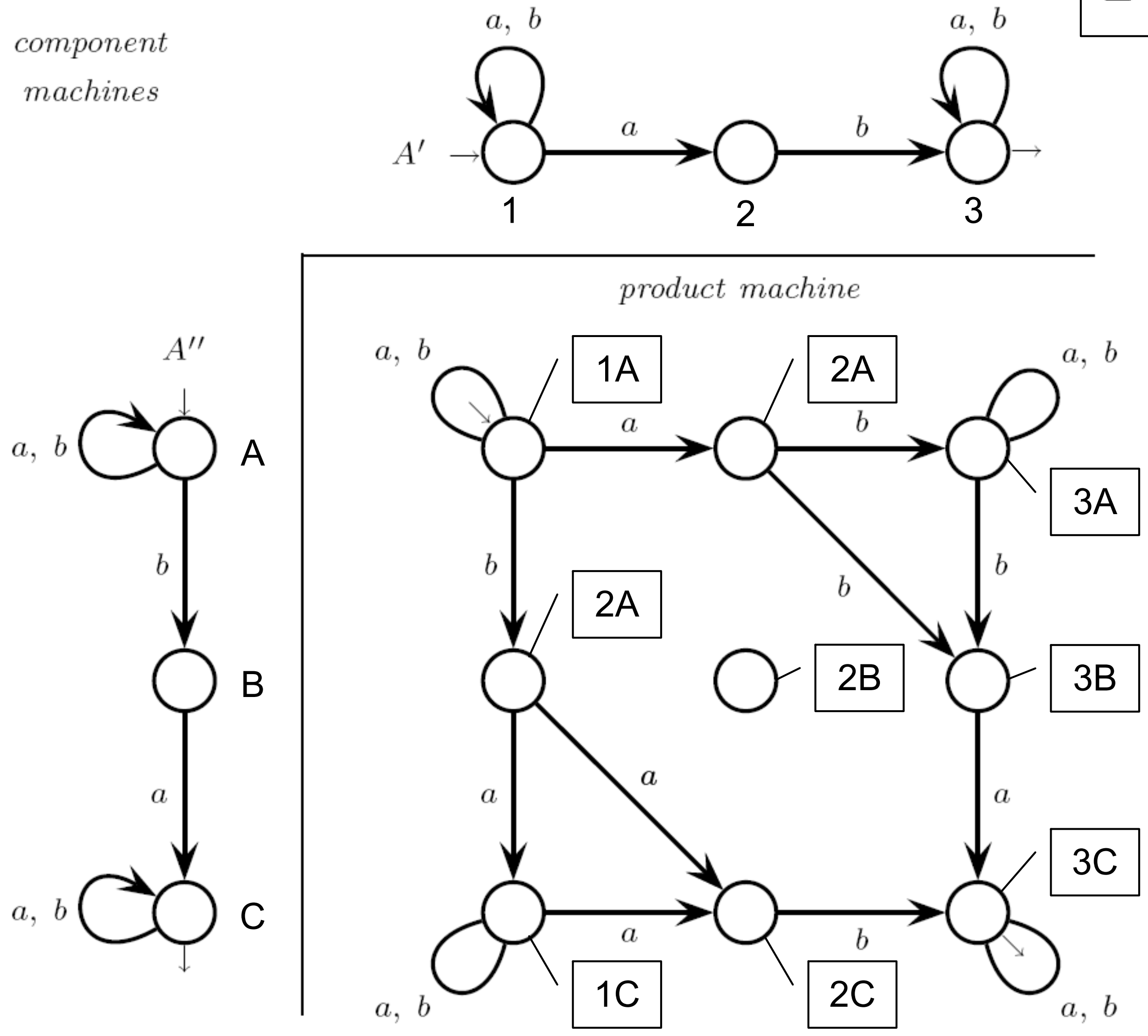
MORE ON THE PRODUCT

- 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

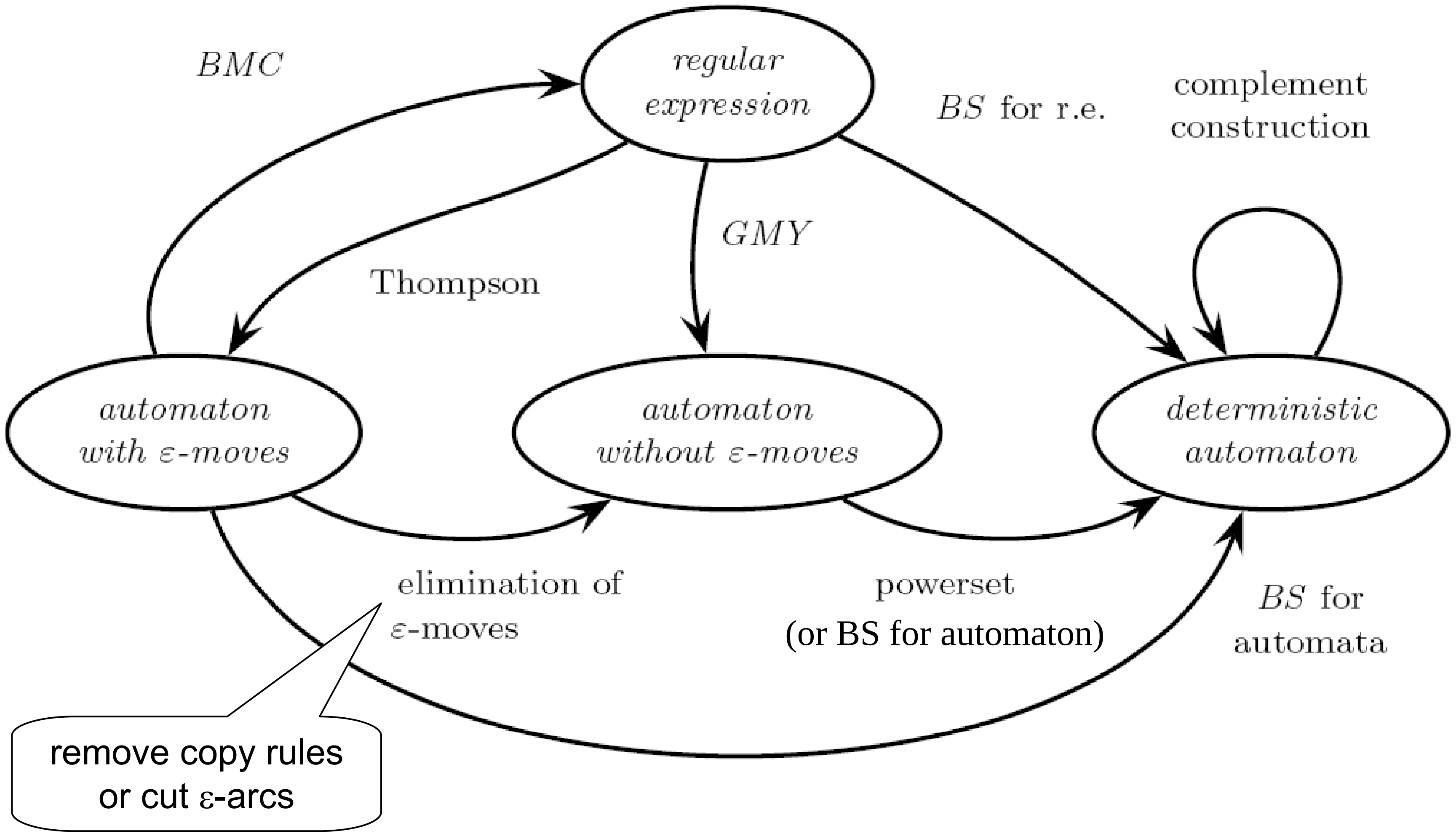
$$\boxed{L' = (a \mid b)^* \; a\, b \; (a \mid b)^*}$$

$$\boxed{L'' = (a \mid b)^* \; b\, a \; (a \mid b)^*}$$

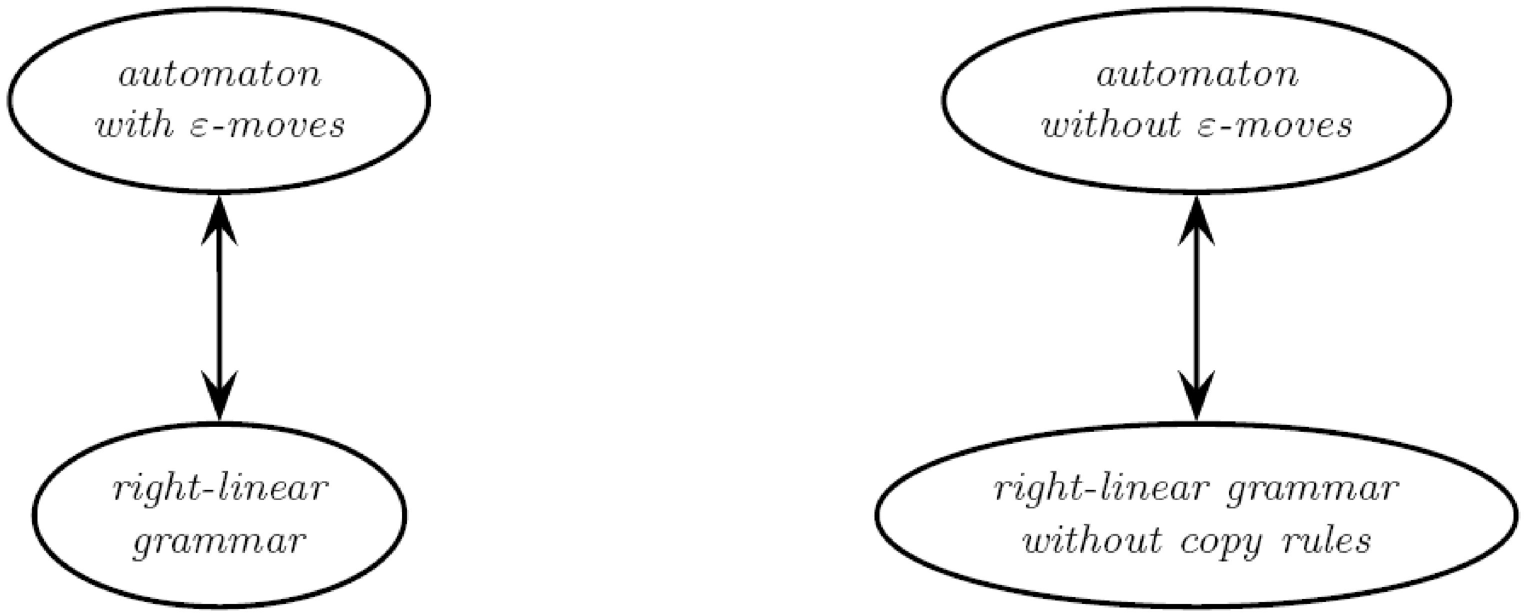


the product automaton is non-deterministic and has a useless state (reduce it)

SUMMARY OF THE TRANSFORMATIONS OF REGEXPS AND AUTOMATA

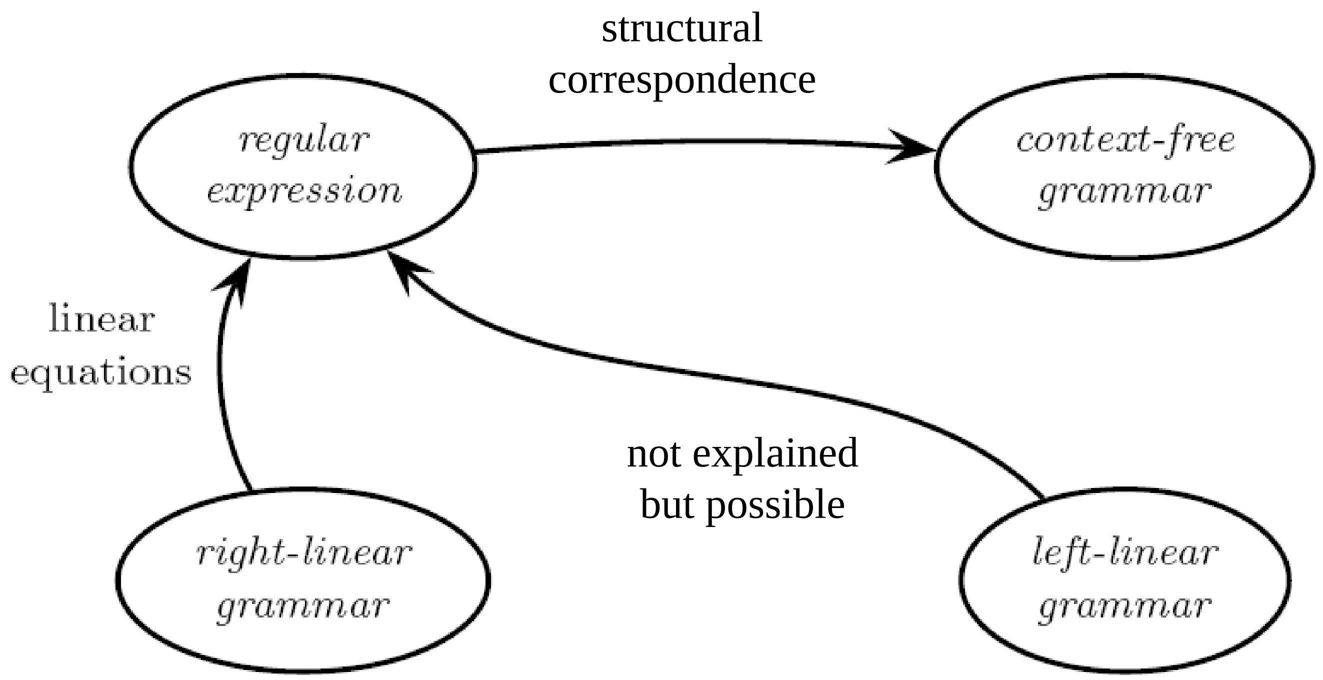


SUMMARY OF THE CORRESPONDENCE OF REGEXPS AND GRAMMARS


automaton with ε-moves
right-linear grammar
automaton without ε-moves
right-linear grammar without copy rules

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