Conversion between Regexp and Finite State Automaton

Prof. Licia Sbattella

aa 2007-08

Translated and adapted by L. Breveglieri

FROM REGULAR EXPRESSION TO RECOGNIZER AUTOMATON

There are several algorithms, which differ as for the characteristic of the automaton (det. vs non-.det., etc), as for the size of the automaton and as for the complexity of the construction.

IN THE FOLLOWING TWO METHODS ARE PRESENTED:

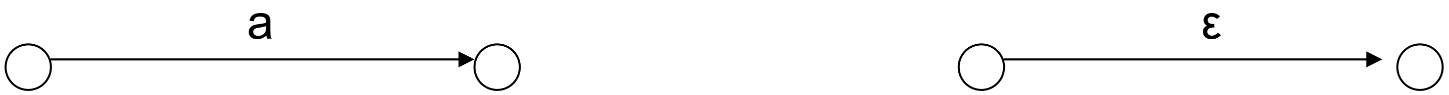
1) THOMPSON METHOD (or structural method):
    1) the regexp is decomposed into subexpressions, until the atomic constituents are reached (i.e. terminal symbols)
    2) constructs the recognizer of the various subexpressions, connects them and builds up a network of recognizers that implement the union, concatenation and star operators
2) GLUSHKOV, MCNAUGHTON AND YAMADA METHOD (GMY): directly constructs a deterministic recognizer, without spontaneous moves, but in general of size larger than the Thompson method does.

BOTH METHODS can be combined with the determinization algorithm.

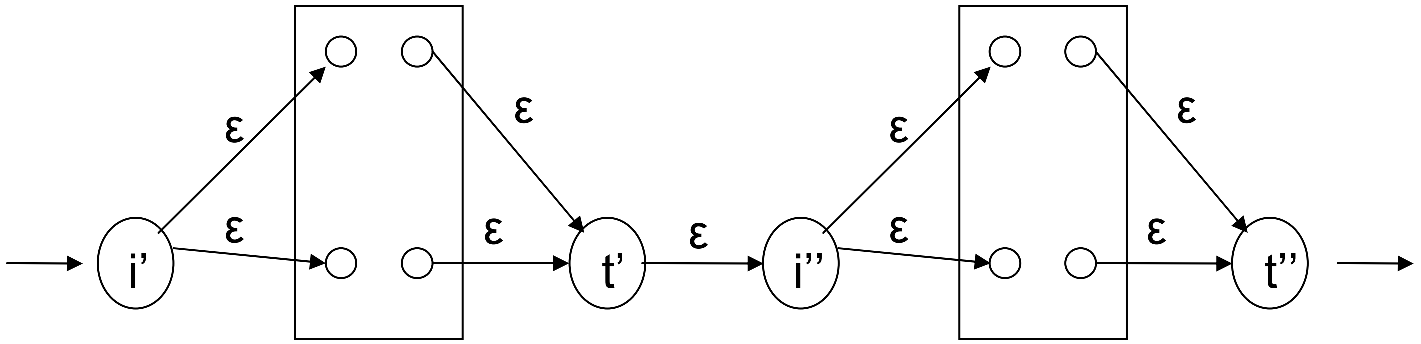
THOMPSON OR STRUCTURAL METHOD

1) modifies the original automaton so as to have unique initial and final states
2) the method is based on the correspondence between regexp and automaton

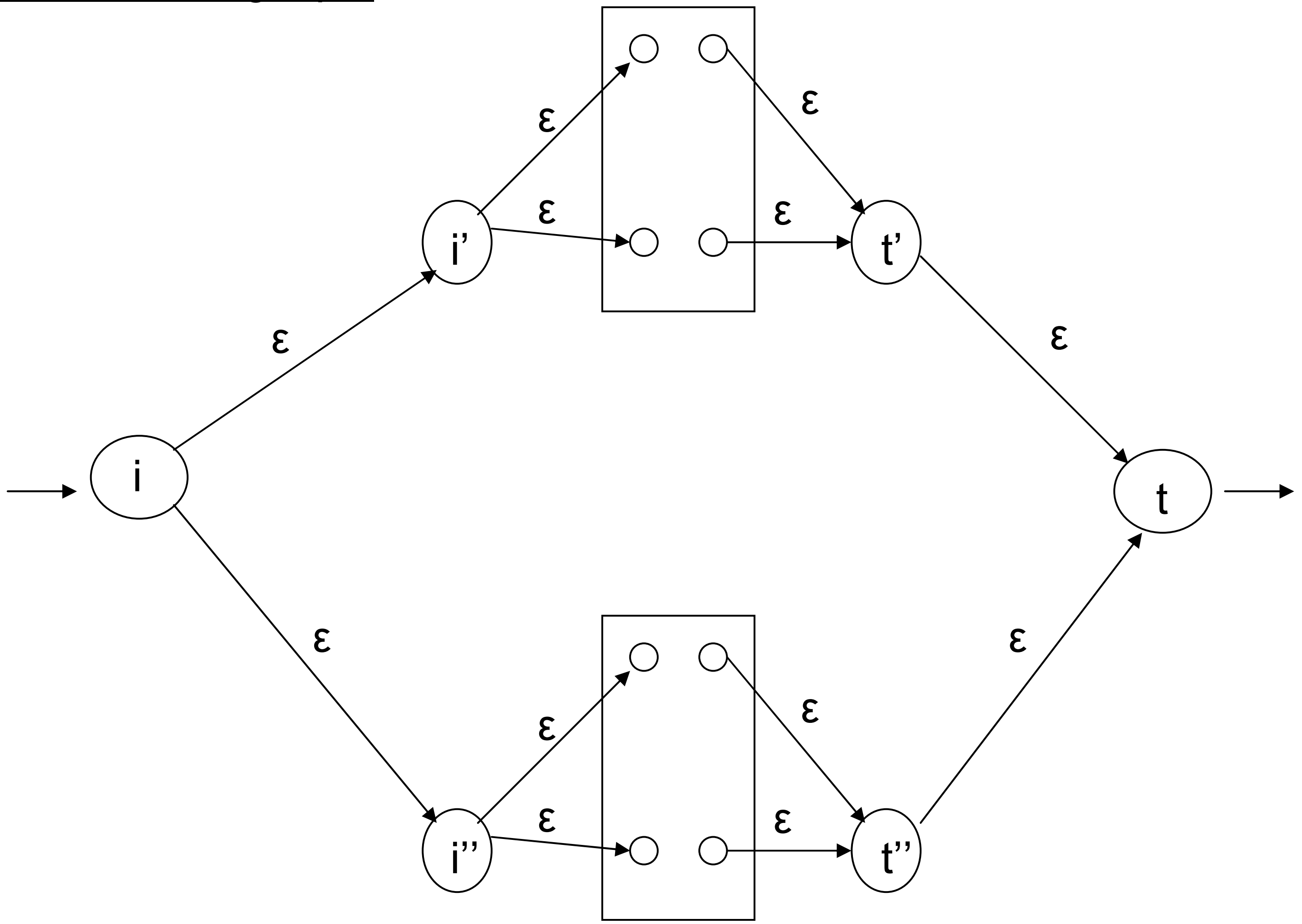
Recognizers of atomic regexps:



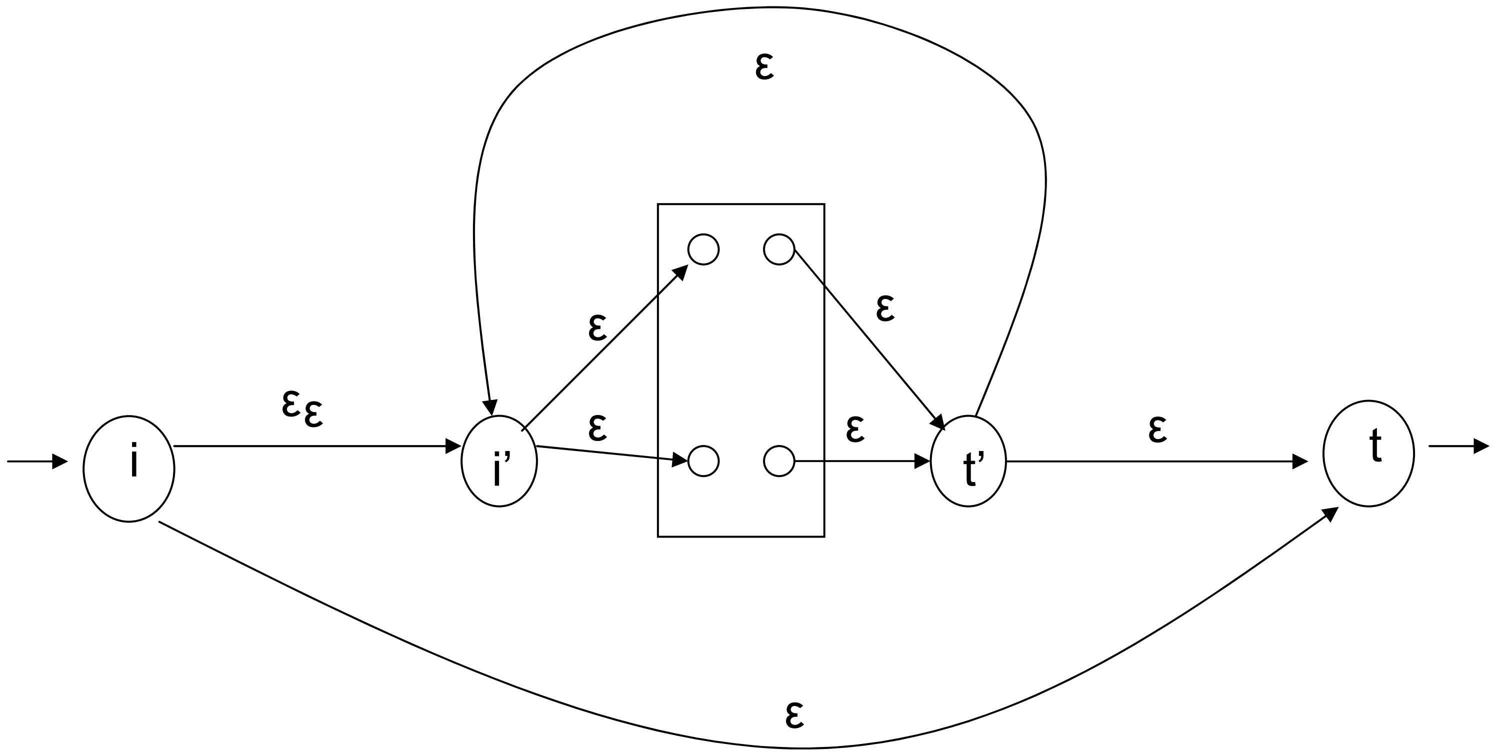
Concatenation of two regexps:



Union of two regexps:

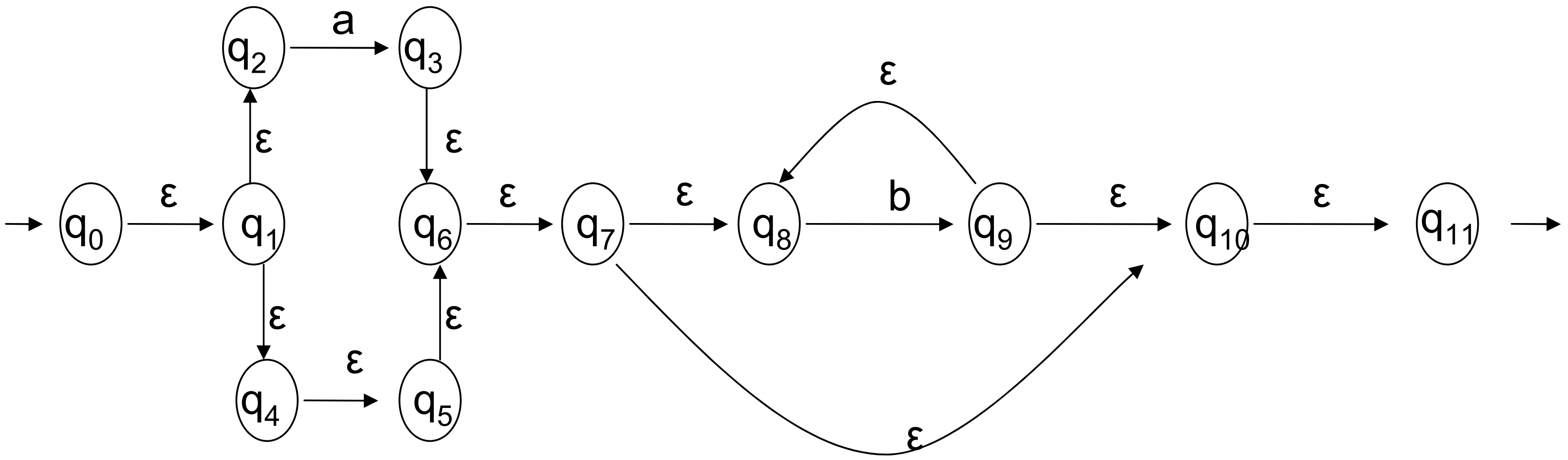


Star closure of a regexp:



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

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



There are improved versions of the Thompson method, that avoid the creation of redundant states (e.g., q_0 and q_{11}), or that eliminate soon spontaneous moves.

GLUSHKOV, McNAUGHTON and YAMADA algorithm (GMY)

It is another classical method, which constructs the automaton equivalent to a given regexp, with states that are in a direct correspondence with the terminal symbols.

LOCALLY TESTABLE LANGUAGE (a subfamily of regular languages).

These are languages very easy to be recognized, as the strings satisfy a simple set of constraints, mainly based on the occurrence and adjacency of letters (example: 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	$Ini(L) = \left\{ a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset \right\}$
end set	$Fin(L) = \left\{ a \in \Sigma \mid \Sigma^* a \cap L \neq \emptyset \right\}$
adjacency set (set of the digrams)	$Dig(L) = \left\{ x \in \Sigma^2 \mid \Sigma^* x \Sigma^* \cap L \neq \emptyset \right\}$
and the complement thereof	$\overline{Dig}(L) = \Sigma^2 \setminus Dig(L)$

EXAMPLE: a locally testable language

$$L_1 = (abc)^+$$

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

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

The three sets above characterize exactly the strings of the language:

$$L_1 \equiv \{x \mid Ini(x) \in Ini(L_1) \wedge Fin(x) \in Fin(L_1) \wedge Dig(x) \subseteq Dig(L_1)\} \quad (1)$$

or equivalently:

$$L_1 \equiv \{x \mid Ini(x) \in Ini(L_1) \wedge Fin(x) \in Fin(L_1)\} \setminus \Sigma^* \overline{Dig}(L_1) \Sigma^* \quad (2)$$

DEFINITION: a language L is said to be LOCAL (or more extensively LOCALLY TESTABLE) if it satisfies the identity 1 before (or 2).

For every language (possibly non-regular), provided it does not contain the empty string, both relations 1 and 2 before still hold when strict inclusion replaces equality. In fact every phrase starts (or ends) with a character of *Ini* (or of *Fin*) and the pairs of adjacent letters (the digrams) are included in those of the language. However, such conditions may be insufficient to exvlude invalid strings.

EXAMPLE: the non-local language L_2 is strictly contained in the set of the strings that start and end with *a* and do not contain the forbidden pairs of adjacent letters (digrams), as the latter language contains strings of even length as well.

$$L_2 = (aa)^+$$

$$Ini(L_2) = Fin(L_2) = \{a\}$$

$$\overline{Dig}(L_2) = \{ab, ba, bb\}$$

$$aaa \quad aaaaa \quad \ldots$$

$$aa \quad aaaa \quad aaaaaa \quad \ldots$$

A language L is locally testable if and only if every string belonging to L matches the constraints given by the sets *Ini*, *Fin* and *Dig*. In fact, in the previous examples the language L_1 is local, while the language L_2 is not so.

THE RECOGNIZER DEVICE OF A LOCAL LANGUAGE IS VERY SIMPLE TO DESIGN: is scans the input string one-way from left to right and checks whether:

- the initial character belongs to the set *Ini*
- every pair of adjacent characters (digrams) does not belong to $\overline{Dig}$
- the final character belongs to the set *Fin*

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

One can implement the above procedure by resorting to a sliding window of width two characters, which is shifted over the input string from left to right. At each shift step the contents of the window 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 very simple and natural to implement by means of a NON-DET. AUTOMATON.

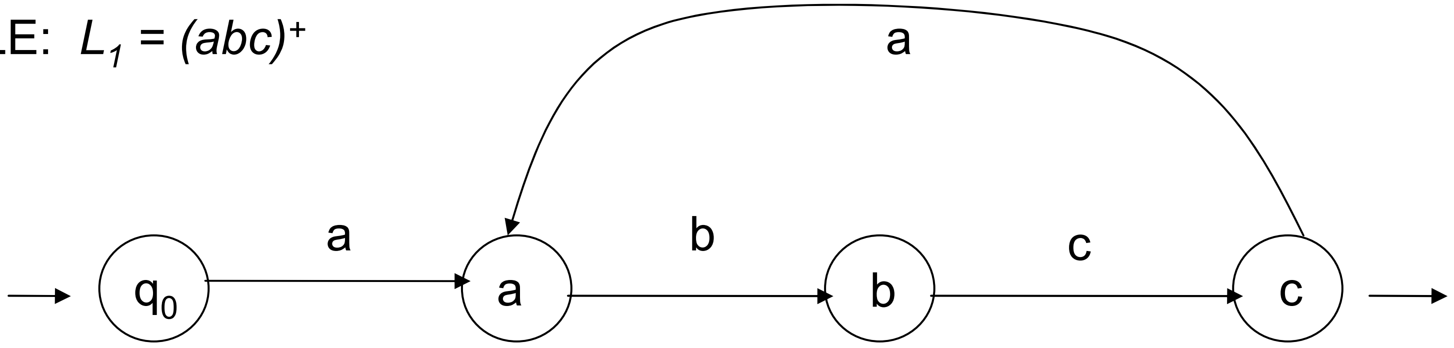
CONSTRUCTION OF THE RECOGNIZER OF THE LOCAL LANGUAGE

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

initial states	$q_0 \in Ini,$	final states	Fin
and moves	$q_0 \overset{a}{\rightarrow} a$ if $a \in Ini,$	$a \overset{b}{\rightarrow} b$ if $ab \in Dig$	

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

EXAMPLE: $L_1 = (abc)^+$



COMPOSITION OF LOCAL LANGUAGES OVER DISJOINT ALPHABETS

The basic operations on languages preserve local languages, provided that they are defined over disjoint alphabets.

PROPERTY n. 1
Given two local languages over disjoint alphabets, the concatenation, union and star (or cross) languages are local as well.

$$L', L'' \text{ with } \Sigma' \cap \Sigma'' = \varnothing$$

$$L' \cup L'', \;\; L'.L'', \;\; L'^{*}$$

Constructing the recognizer for the composite language is immediate:

UNION: - initial state q_0: is obtained by merging q_0' and q_0''
- final states: are those of the two automata, i.e. $F' \cup F''$

If either language, or both, contain the empty string, q_0 is final too.

CONCATENATION:

- initial state: q'_0
- final states: if $\varepsilon \notin L''$

  only the final states of F''

  else

  the final states F' and those of F''
- moves:
  - the moves of L'
  - the moves of L'' but those starting at q''_0
  - and the moves

    $q' \xrightarrow{a} q''$ with $q' \in F'$ and

    in L'' $q''_0 \xrightarrow{a} q''$

STAR:

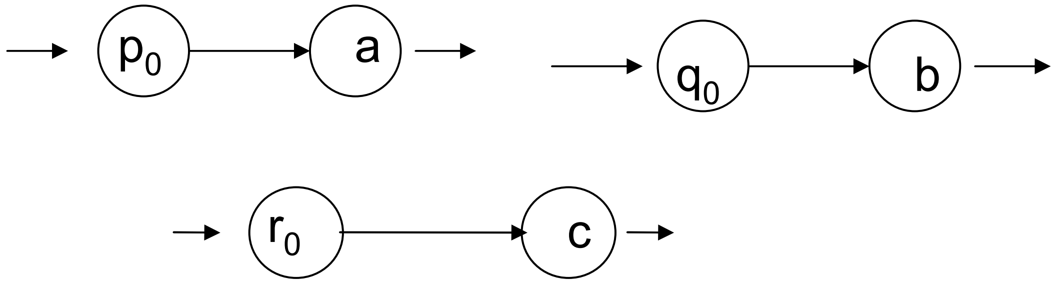
- add q'_0 to the final states
- for every other final state q in F' add

  $q \xrightarrow{a} r$ if the aut. has $q'_0 \xrightarrow{a} r$

EXAMPLE – (a b U c)*

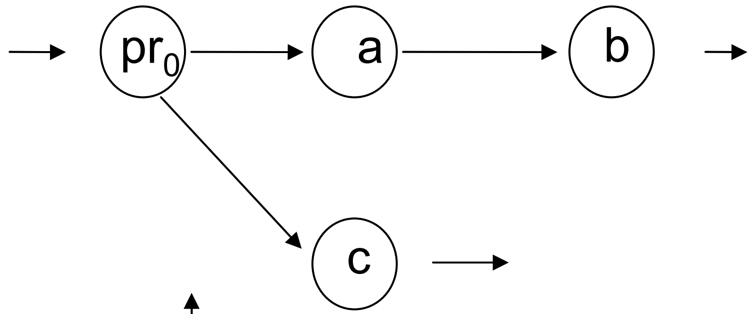
elementary automata

a, b, c



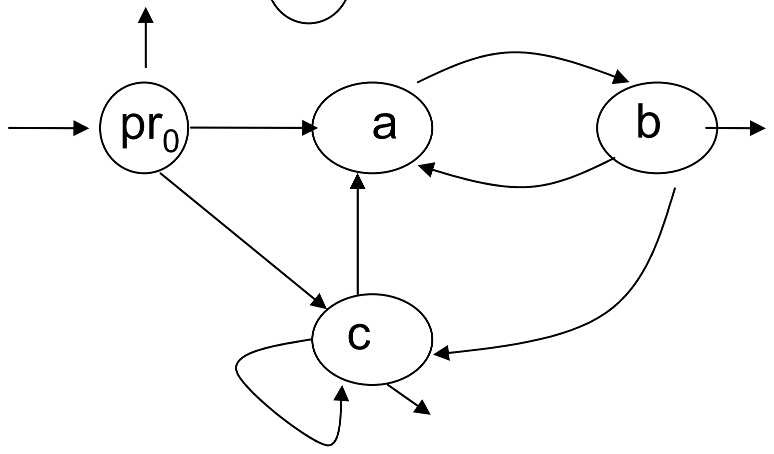
concatenation and union

a b U c



star

(a b U c)*



GMY ALGORITHM

To move from a generic regular language to a local one:.

A REGEXP is said to be linear if there is not any repeated generator

*(a b c)** is linear

*(a b)*a* is not linear (although the language is local indeed)

PROPERTY n. 2
THE LANGUAGE GENERATED BY A LINEAR REGEXP IS LOCAL (CAUTION: the opposite implication does not hold, read above) – In fact, linearity implies that the subexpressions of the regexp are defined over disjoint alphabets. But the regexp is obtained by composition of its subexpressions, and therefore the generated language is local as a consequence of property n. 1.

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 the generic language.

HOW TO COMPUTE THE LOCAL SETS OF A REGULAR LANGUAGE

Does the regexp generate the empty string ?

if $Null(e) = \varepsilon \quad \varepsilon \in L(e)$

if $Null(e) = \emptyset \quad \varepsilon \notin L(e)$

$$Null(\varepsilon) = \varepsilon \qquad Null(\emptyset) = \emptyset$$

$Null(a) = \emptyset$ for every term. $. \ a$

$$Null(e \bigcup e') = Null(e) \bigcup Null(e')$$

$$Null(e.e') = Null(e) \bigcap Null(e')$$

$$Null(e^*) = \varepsilon \qquad Null(e^+) = Null(e)$$

Example:

$$Null((a \bigcup b)^* ba) = Null((a \bigcup b)^*) \bigcap Null(ba) =$$

$$= \varepsilon \bigcap (Null(b) \bigcap Null(a)) = \varepsilon \bigcap \emptyset = \emptyset$$

Start symbols

$$Ini(\varnothing) = \varnothing$$
$$Ini(\varepsilon) = \varnothing$$
$$Ini(a) = \{a\}$$ for every term. ı. a
$$Ini(e \cup e') = Ini(e) \cup Ini(e')$$
$$Ini(e.e') = Ini(e) \cup Null(e)Ini(e')$$
$$Ini(e^*) = Ini(e^+) = Ini(e)$$

End symbols

$$Fin(\varnothing) = \varnothing$$
$$Fin(\varepsilon) = \varnothing$$
$$Fin(a) = \{a\}$$ for every term. a
$$Fin(e \cup e') = Fin(e) \cup Fin(e')$$
$$Fin(e.e') = Fin(e') \cup Fin(e)Null(e')$$
$$Fin(e^*) = Fin(e^+) = Fin(e)$$

Adjacent pairs (digrams)

$$Dig(\varnothing) = \varnothing \qquad Dig(\varepsilon) = \varnothing$$
$$Dig(a) = \varnothing$$ for every term. a
$$Dig(e \cup e') = Dig(e) \cup Dig(e')$$
$$Dig(e.e') = Dig(e) \cup Dig(e') \cup Fin(e)Ini(e')$$
$$Dig(e^*) = Dig(e^+) = Dig(e) \cup Fin(e)Ini(e)$$

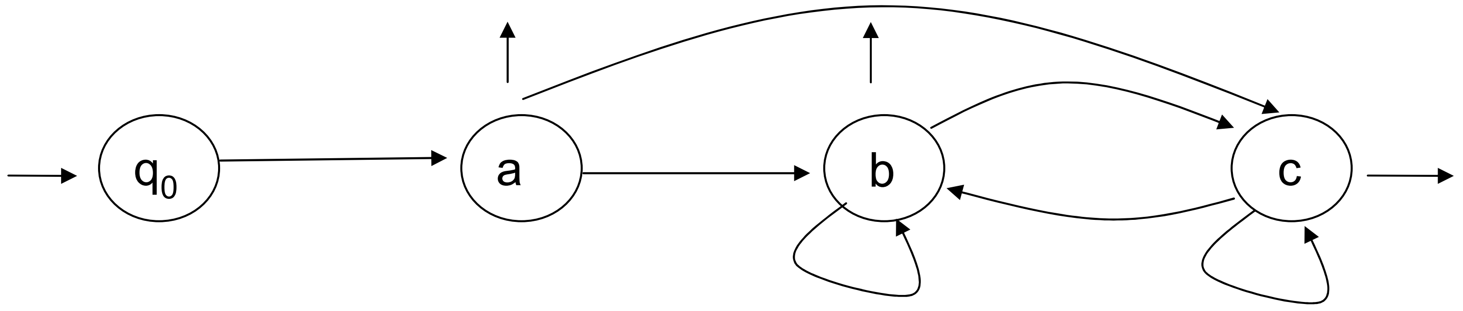
NOTE:
The computation rules of the two sets *Ini* and *Fin* are dual. They are transformed into each other if the regexp is replaced by the mirror image.

EXAMPLE – the recognizer of a linear regexp – a (b U c)*

$$Null = \emptyset \qquad Ini = a$$

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

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



HOW TO EXTEND FROM A LINEAR REGEXP TO A GENERIC ONE:

1) 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.

$(ab)^* a$ becomes $(a_1 b_2)^* a_3$

2) Construct the local recognizer of the numbered regexp (now linear).

3) 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 for *e* the three characteristic local sets *Ini*, *Fin* and *Dig*.
3) Design the recognizer of the local language generated by *e*'.
4) Cancel the indexing and thus obtain the recognizer of *e*.

CAUTION: the final recognizer, in general, is NON-DETERMINISTIC.

EXAMPLE – $(a_1\, b_2)^*\, a_3$ – $(a\, b)^*\, a$

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a_1"] \arrow[rrr, bend left=40, "a_3"] & a_1 \arrow[r, bend left, "b_2"] & b_2 \arrow[l, bend left, "a_1"] \arrow[r, "a_3"] & a_3 \arrow[r] & {}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a"] \arrow[rrr, bend left=40, "a"] & a_1 \arrow[r, bend left, "b"] & b_2 \arrow[l, bend left, "a"] \arrow[r, "a"] & a_3 \arrow[r] & {}
\end{tikzcd}$$

The outcome is an indeterministic automaton without spontaneous moves, with as many states as the occurrences of generators in the rexexp are, plus one (the initial state).

CONSTRUCTION OF THE DETERMINISTIC RECOGNIZER (BERRY & SETHI algorithm)

In order to obtain the deterministic recognizer, one could simply apply the subset construction to the non-det. recognizer built by means of the GMY algorithm.

However, there exists a more direct algorithm.

Consider the end-marked regexp *e'*⊣ instead of the original one *e'*.

let e be a regexp over Σ

and e' be a regexp over Σ_N

with $Null, Ini, Fin, Dig$

$$Seg(a_i) = \{ b_j \mid a_i b_j \in Dig(e') \}$$

$-|\ \in Seg(a_i)$ for every $a_i \in Fin(e')$

$$Seg(-|) = \varnothing$$

BS ALGORITHM

Each state is denoted by a subset of $\Sigma_N \cup \dashv$.

The algorithm examines every state to construct the outgoing moves and the destination states, and applies a rule similar to the subset construction.

The state is marked as visited to avoid a second exam.

The initial state is *Ini(e’⊣)*.
A state is final if it contains ⊣
At the beginning, the set state Q contains only the initial state.

```
$Q := \{ Ini(e' \dashv) \}$
while exists in $Q$ a state $q$ not visited
do
    mark $q$ as visited
    for every char $b \in \Sigma$
    do
        $q' := \bigcup_{\forall \text{car.numerato } b' \in q} Seg(b')$
        if $q'$ is not empty and does not
        belong to $Q$ add it as a new state,
        not visited, and pose:
        $Q := Q \cup \{ q' \}$
        add move $q \xrightarrow{b} q'$
    end do
end do
```

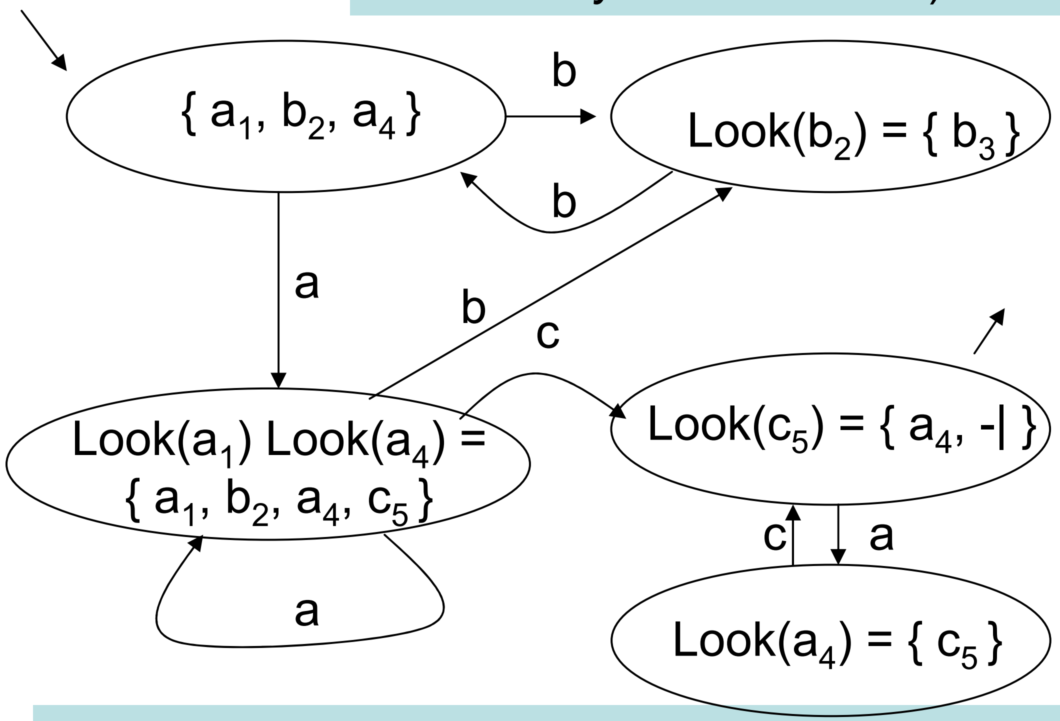
EXAMPLE

NOTE: the labelings of the nodes of a BS automaton express the question "*what is expected next in the input tape ?*" (not "*what has already been found ?*")

$$(a \mid bb)^{*}(ac)^{+}$$

$$(a_1 \mid b_2 b_3)^{*}(a_4 c_5)^{+} -|$$

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



	lookahead
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, -|

An automaton built by the BS method is usually far from being minimal, and may contain many unuseful or duplicated states. It should be reduced and minimized at the end.

REGEXP WITH COMPLEMENT AND INTERSECTION

Regexps can contain also the complement, intersection and set difference operators. Such operators may be very useful to make the regexp more coincise.

PROPERTY – REGULAR LANGUAGES ARE CLOSED WITH RESPECT TO COMPLEMENT AND INTERSECTION

$$\boxed{\begin{array}{ll}\text{if} & L, L', L'' \in REG \\ \text{then} & \neg L \in REG \quad L' \bigcap L'' \in REG\end{array}}$$

1) CONSTRUCTION OF THE DET. RECOGNIZER OF THE COMPLEMENT LANG.

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

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

ALGORITHM: deterministic recognizer $\overline{M}$ of the complement language.

Complete M by adding the error state *p* and the missing moves.

1. Create *p* (the error state) not belonging to *Q*
   The states of the complement automaton are Q U { *p* }.
2. The complement transition function is:
3. Swap the initial and final states:

$$\boxed{\begin{array}{l} a)\ \overline{\delta}(q,a) = \delta(q,a), \text{ if } \delta(q,a) \text{ is defined} \\ b)\ \overline{\delta}(q,a) = p \text{ otherwise} \\ c)\ \overline{\delta}(p,a) = p \text{ for every char. } a \in \Sigma \end{array}}$$

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

A recognizing path of M will not end into a final state in the complement automaton.

$$\boxed{\begin{array}{l} x \in L(M) \\ x \notin L(\overline{M}) \end{array}}$$

A non-recognizing path of M will end into a final state in the complement automaton.

2) THE INTERSECTION LANGUAGE OF TWO REGULAR LANGUAGES IS REGULAR AS WELL

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

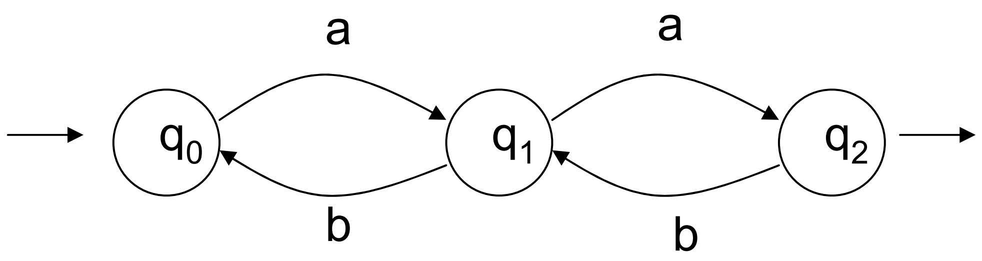
$$\boxed{L' \bigcap L'' = \neg(\neg L' \bigcup \neg L'')}$$

COROLLARY: the set difference of two regular languages is regular as well, due to the following identity:

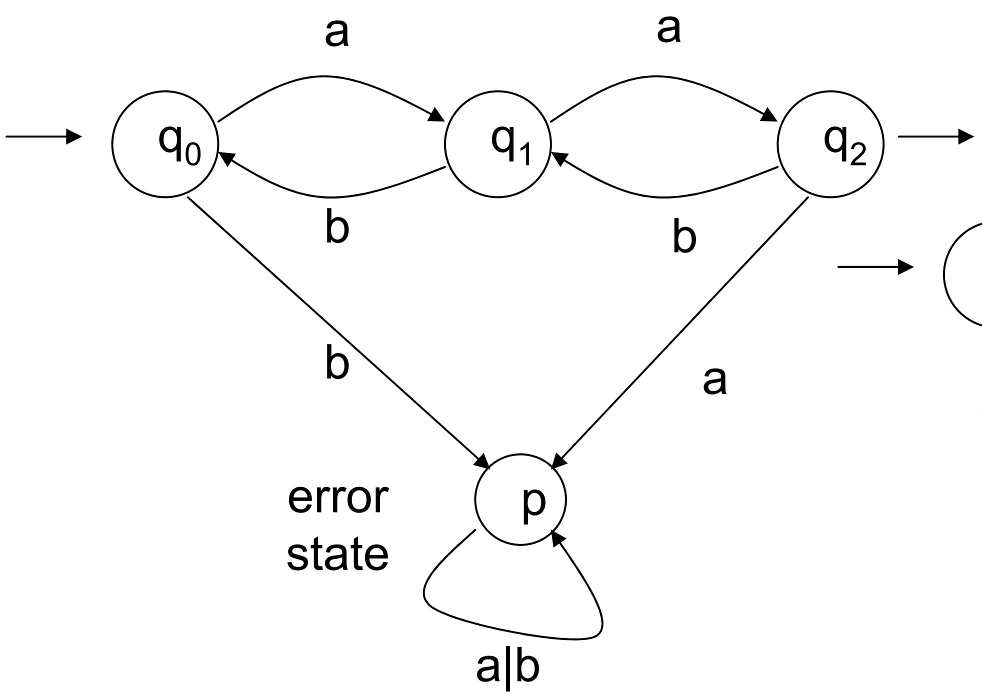
$$\boxed{L' \setminus L'' = L' \bigcap \neg L''}$$

EXAMPLE: COMPLEMENT AUTOMATON

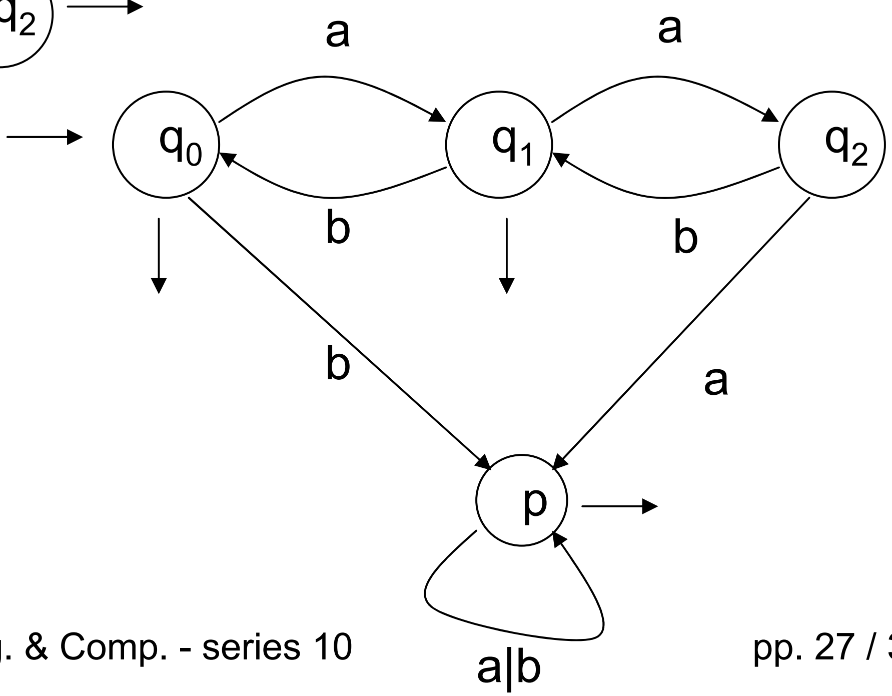
Original automaton:



Natural completion:



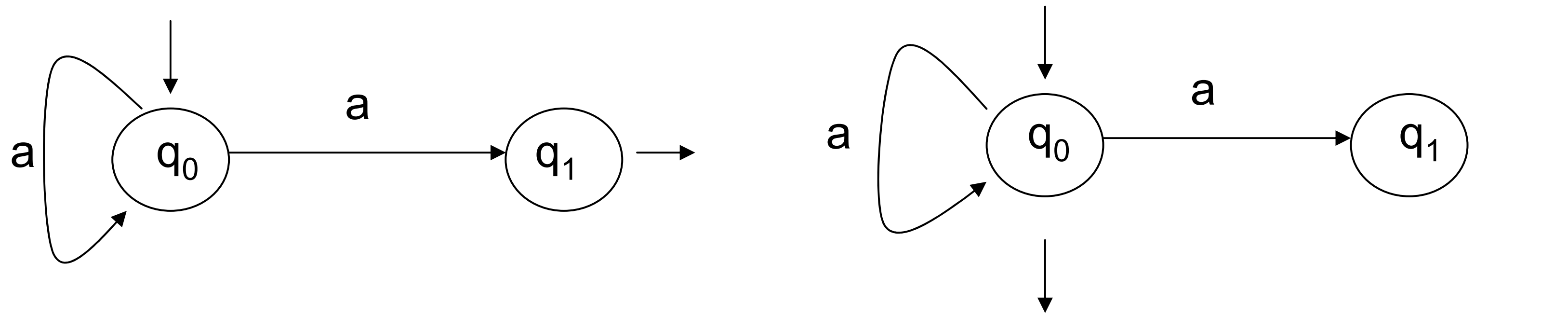
Complement automaton:



CAUTION: for the complement construction, it is essential that the original automaton is deterministic, otherwise the construction is wrong as the original and complement languages might overlap, which is in violation of the complement. See below:

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

Example:



original automaton complement pseudo-automaton

The pretended complement automaton accepts the strings *a*, which is also in the original language.

CAUTION: the complement automaton may not be in minimal form and contain unusefule states.

(CARTESIAN) PRODUCT OF AUTOMATA

A very common construction of formal languages, where a single automaton simulates the computation of two automata working in parallel on the same input string. Very useful for constructing the intersection automaton.

As said before, to construct the intersection automaton one might resort to the De Morgan theorem, as follows:

- construct the det. recognizers of the two languages
- construct the respective complement automata
- construct their union (use Thompson method)
- make deterministic the union automaton
- complement again and thus get the intersection automaton

Cartesian product allows to obtain a more direct construction.

The intersection of the two languages is recognized directly by the cartesian product of the two automata.

Suppose both automata do not contain 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 state is a pair <q', q''>, where the first (second) member is a state of the first (second) machine. The move is:

$$< q',q'' > \xrightarrow{a} < r',r'' > \quad \text{If and only if} \quad q' \xrightarrow{a} r' \wedge q'' \xrightarrow{a} r''$$

The product machine has a move if and only if the projection of such a move onto the first (second) 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 ?

1) If a string is accepted by the product machine, it is accepted simultaneously by the two automata as well.
2) If a string is not accepted by the product machine, either automaton, or both, must not accept it either, which means the string is not in the intersection.

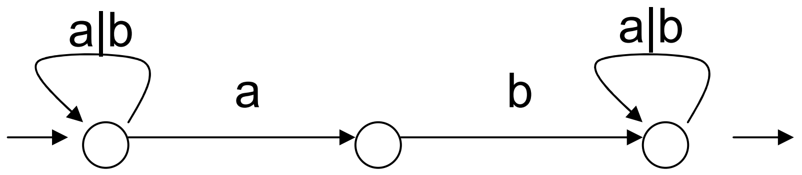
NOTE:

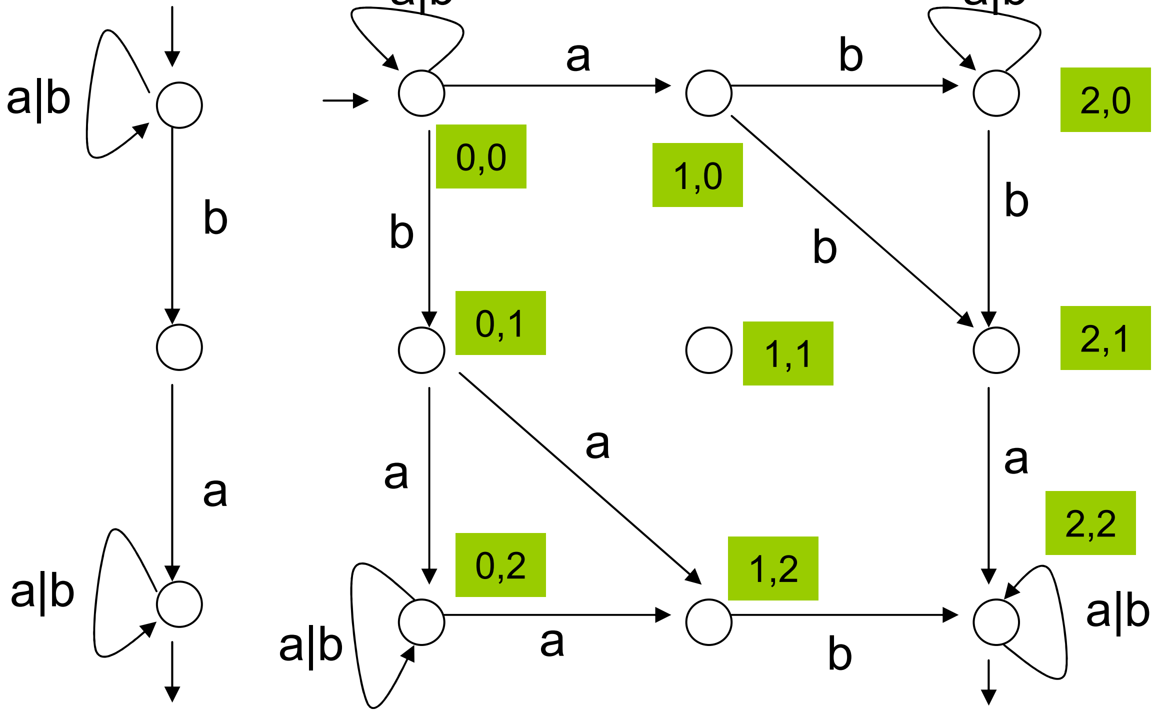
1) The product construction is equivalent to simulating in parallel both machines.
2) With some modification, the construction can be adapted to other language operators. The reader can try do so for union (not very easy) and shuffle.

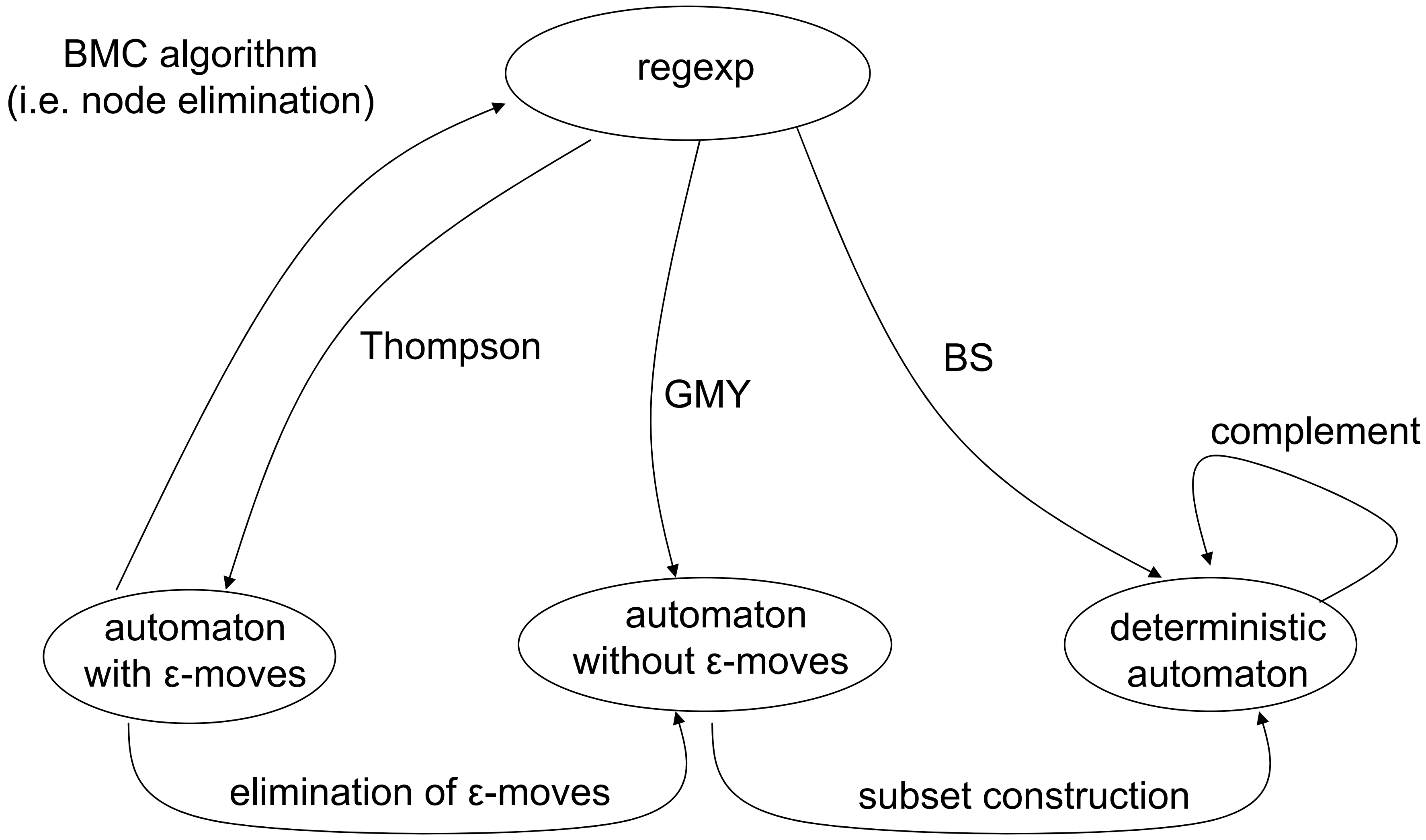
EXAMPLE – intersection & product machine

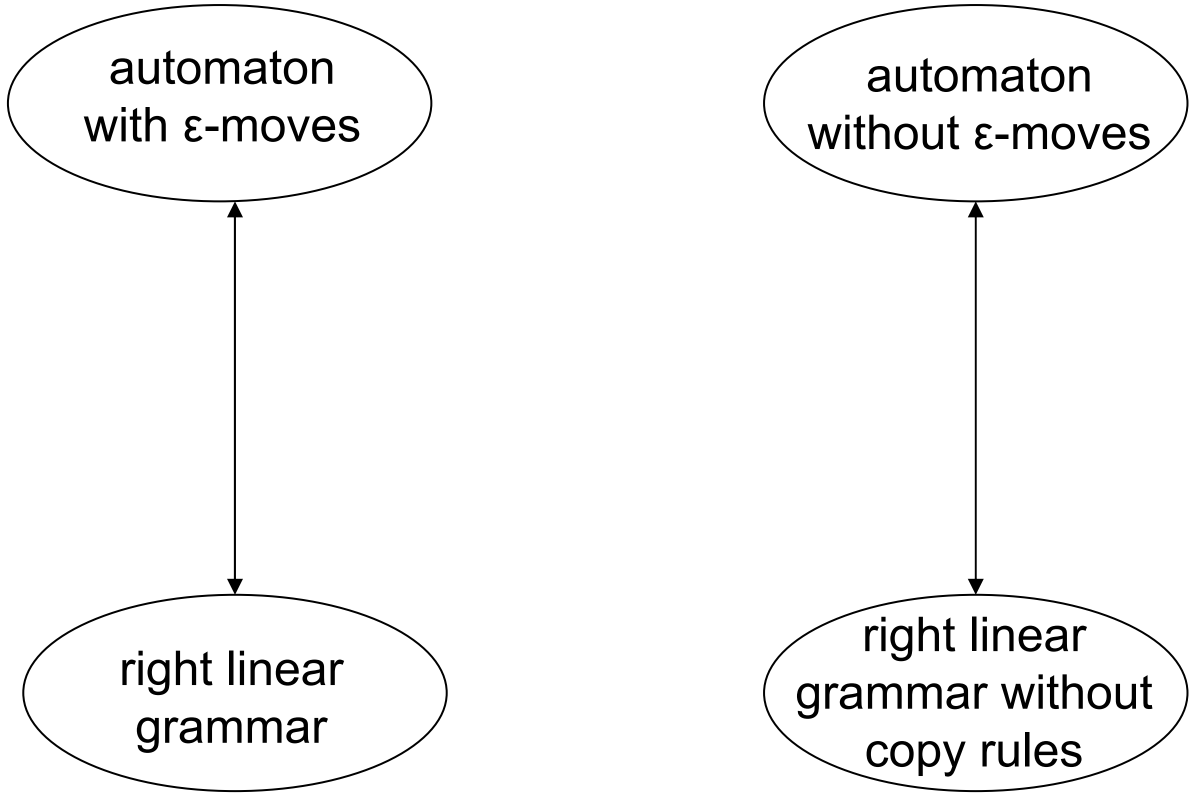
$$L' = (a\,|\,b)^* ab (a\,|\,b)^*$$

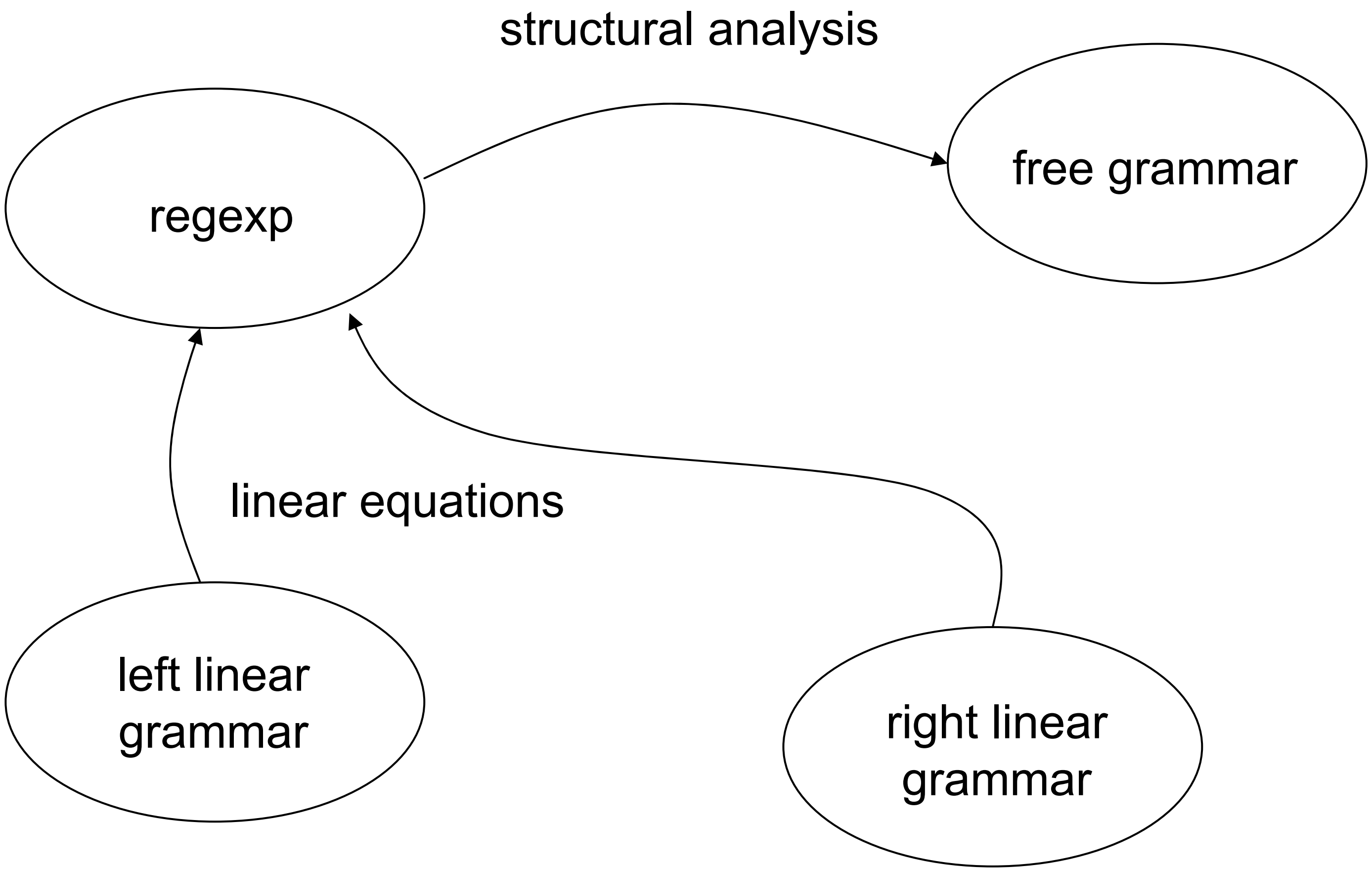
$$L'' = (a\,|\,b)^* ba (a\,|\,b)^*$$






BMC algorithm
(i.e. node elimination)
regexp
Thompson
GMY
BS
complement
automaton with ε-moves
automaton without ε-moves
deterministic automaton
elimination of ε-moves
subset construction


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


structural analysis
regexp
free grammar
linear equations
left linear grammar
right linear grammar

Bibliography

– S. Crespi Reghizzi, *Linguaggi Formali e Compilazione*, Pitagora Editrice Bologna, 2006

– Hopcroft, Ullman, *Formal Languages and their Relation to Automata*, Addison Wesley, 1969

– A. Salomaa – *Formal Languages*, Academic Press, 1973

– D. Mandrioli, C. Ghezzi – *Theoretical Foundations of Computer Science*, John Wiley & Sons, 1987

– L. Breveglieri, S. Crespi Reghizzi, *Linguaggi Formali e Compilatori: Temi d’Esame Risolti,* web site (eng + ita)