

# Chapter 7: Kleene's Theorem

---

- Method of proof

Theorem: Let  $A, B, C$  be sets such that  $A \subset B$ ,  $B \subset C$ ,  $C \subset A$ .  
Then  $A = B = C$ .

Proof: Let  $x$  be an element of  $B$ .  $x \in C$  (because  $B \subset C$ ). Also  $x \in A$  (because  $C \subset A$ ). Thus  $B \subset A$ , and we can conclude  $A = B$ . By analogy,  $B = C$  and thus all the sets are equal.

Remark: Regular expressions, finite automata, and transition graphs each define a set of languages.

# Kleene's Theorem

---

Lemma 1: Every language that can be defined by a finite automaton can also be defined by a transition graph.  $FA \subset TG$

Lemma 2: Every language that can be defined by a transition graph can also be defined by a regular expression.  $TG \subset RE$

Lemma 3: Every language that can be defined by a regular expression can also be defined by a finite automaton.  $RE \subset FA$

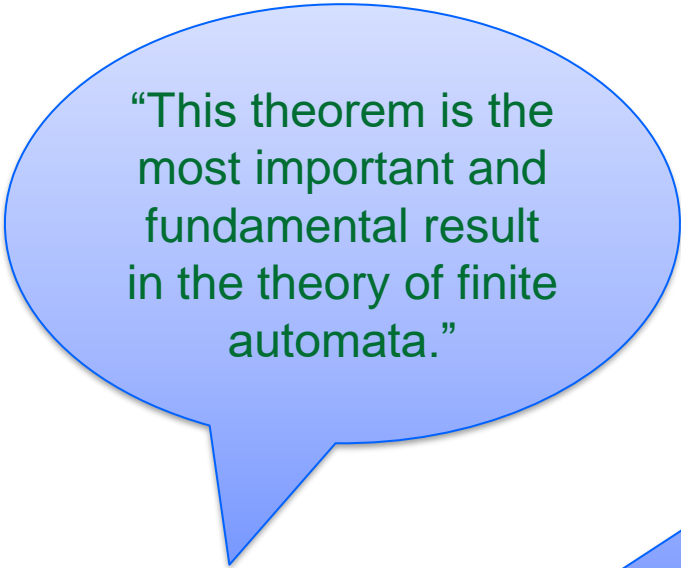
## Kleene's Theorem

Any language that can be defined by a regular expression, or finite automaton, or transition graph can be defined by all three methods.

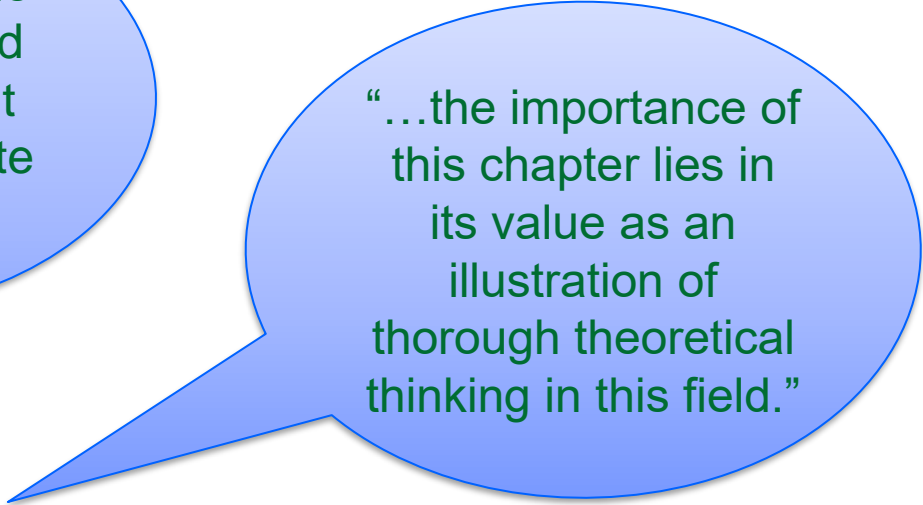
Proof: By the lemmas above, and the theorem on the previous page. (It remains to prove the lemmas.)

# Kleene's Theorem

---



“This theorem is the most important and fundamental result in the theory of finite automata.”



“...the importance of this chapter lies in its value as an illustration of thorough theoretical thinking in this field.”

## Kleene's Theorem

Any language that can be defined by a regular expression, or finite automaton, or transition graph can be defined by all three methods.

# Kleene's Theorem

---

Lemma 1: Every language that can be defined by a finite automaton can also be defined by a transition graph.

Proof: By definition, every finite automaton is a transition graph.

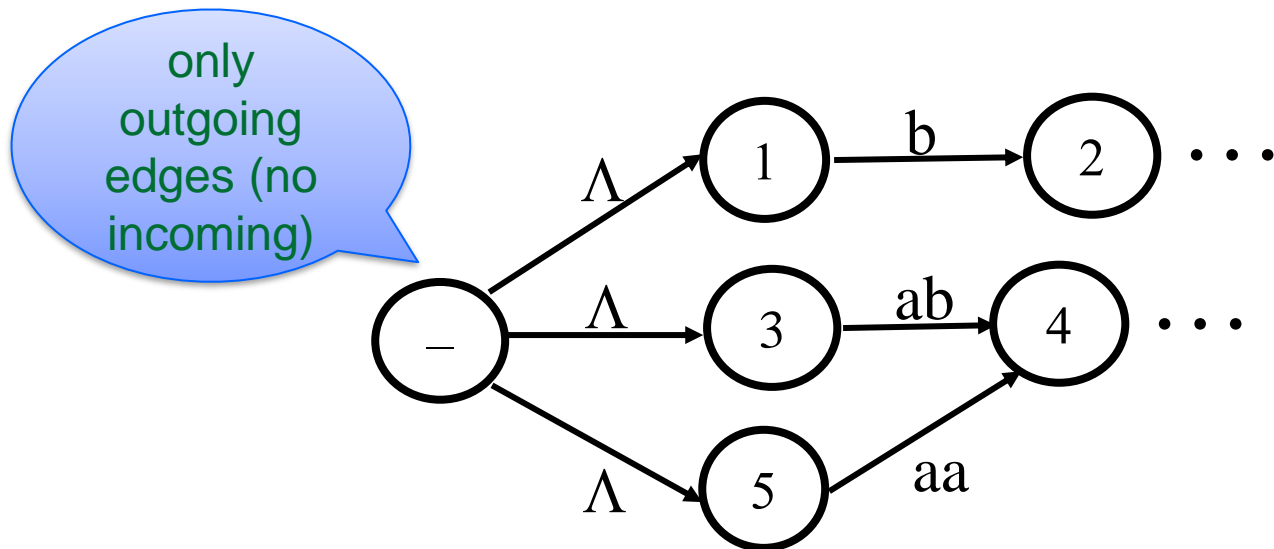
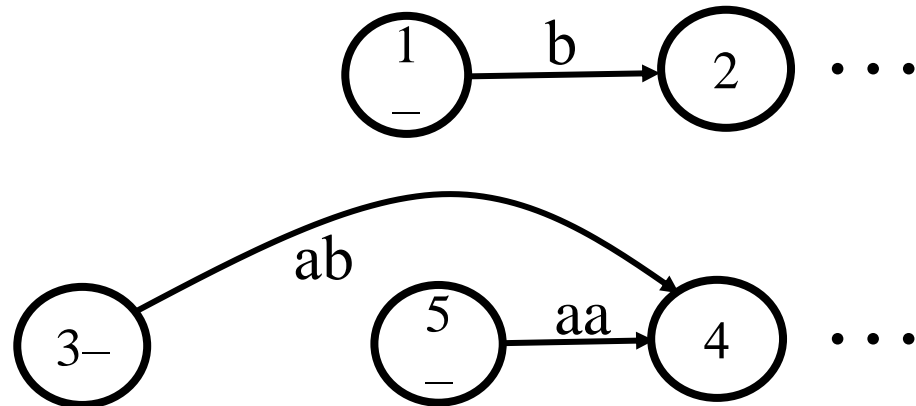
Lemma 2: Every language that can be defined by a transition graph can also be defined by a regular expression.

Proof: By **constructive algorithm**, which works

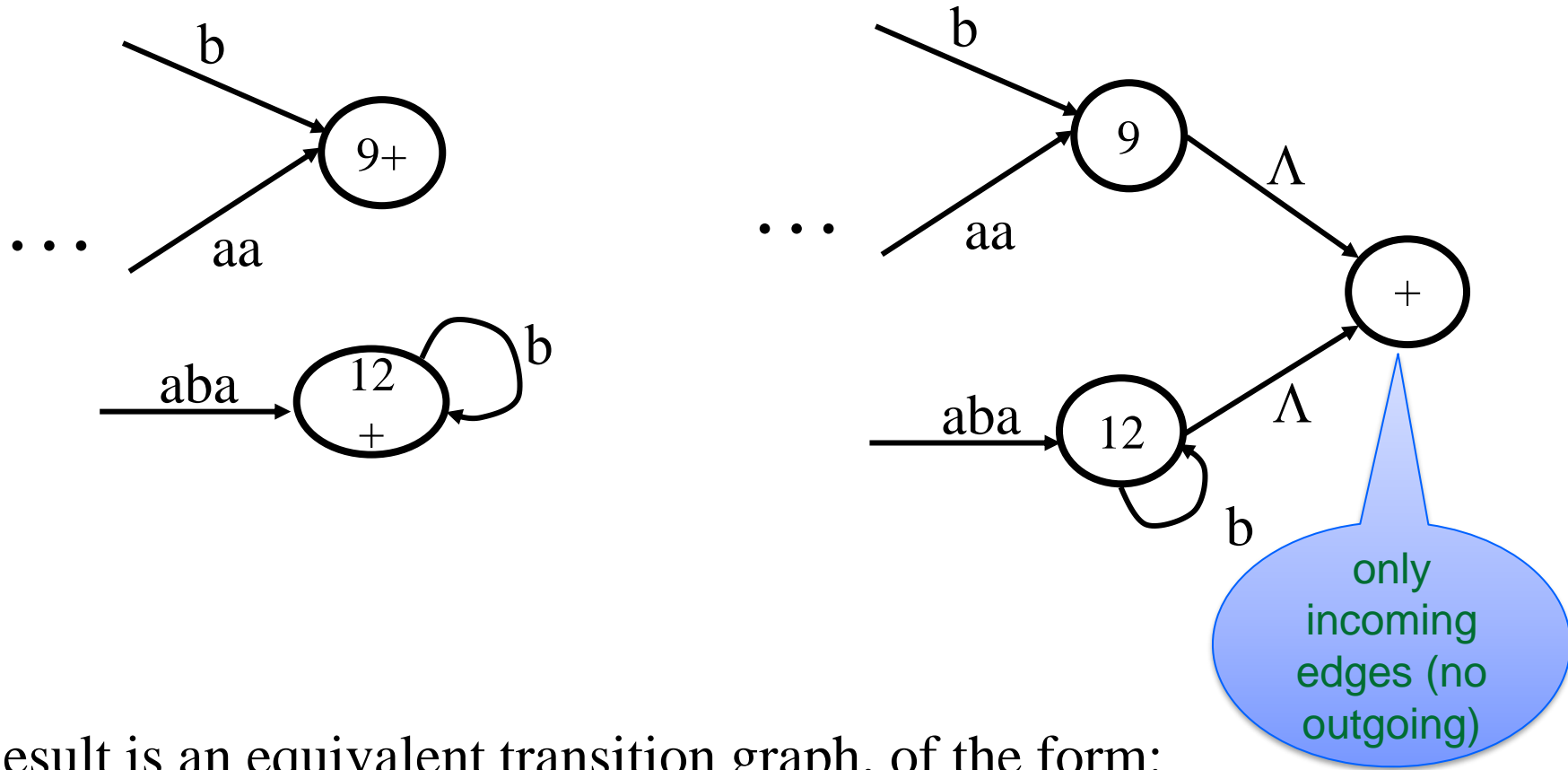
1. for every transition graph
2. in a finite number of steps

Step 1a: transform to a transition graph with a single start state.

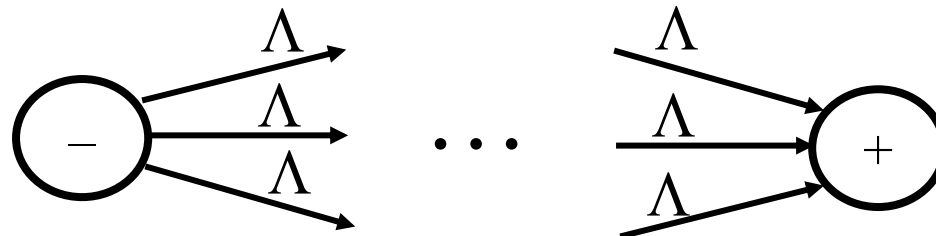
---



## Step 1b: and a single final state



Result is an equivalent transition graph, of the form:

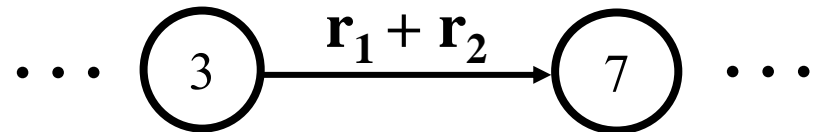
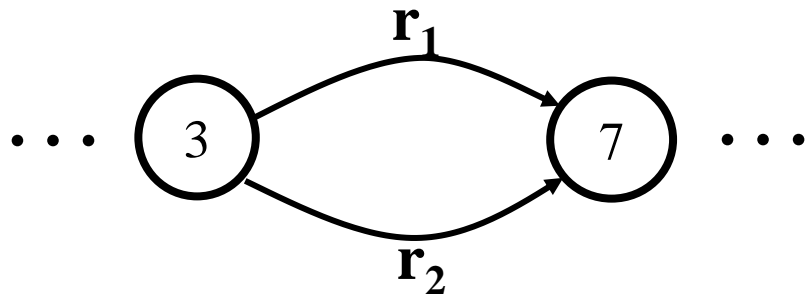
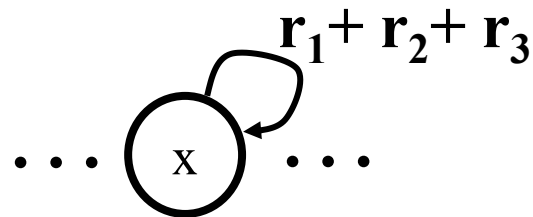
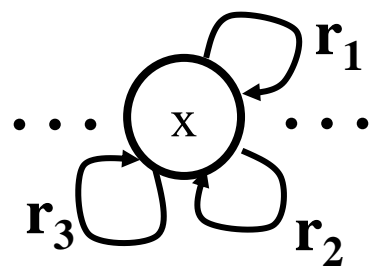


## Transform to a GTG

---

2. Transform every label (sequence of letters or  $\Lambda$ ) to bold (regular expression).

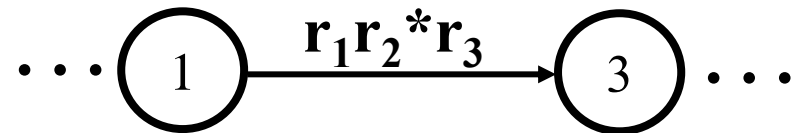
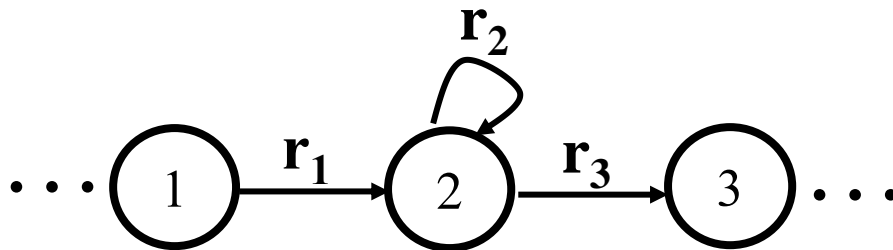
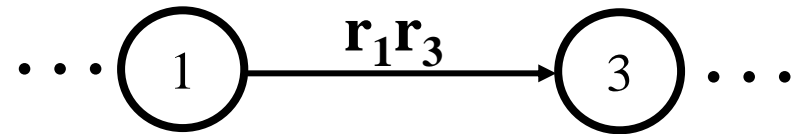
Combine edges that have the same starting and ending state.



### 3. Eliminate states one by one

---

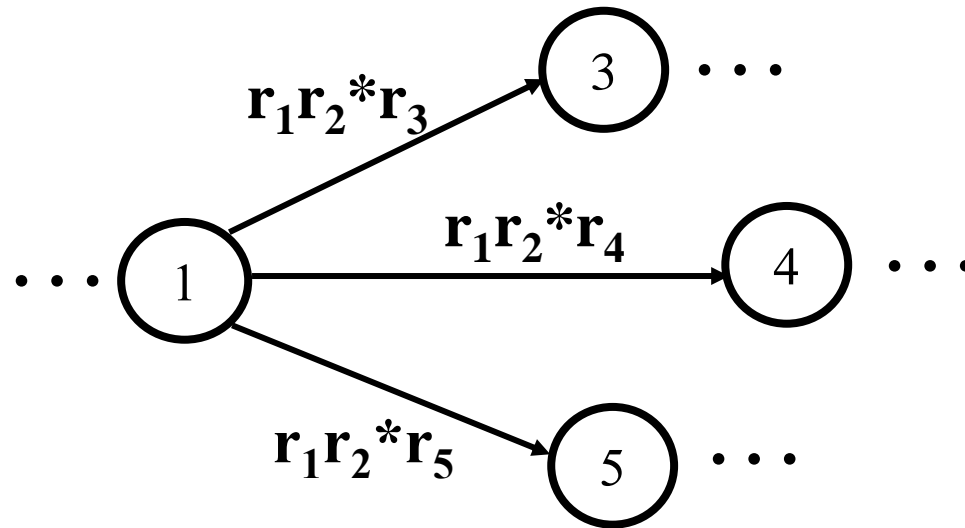
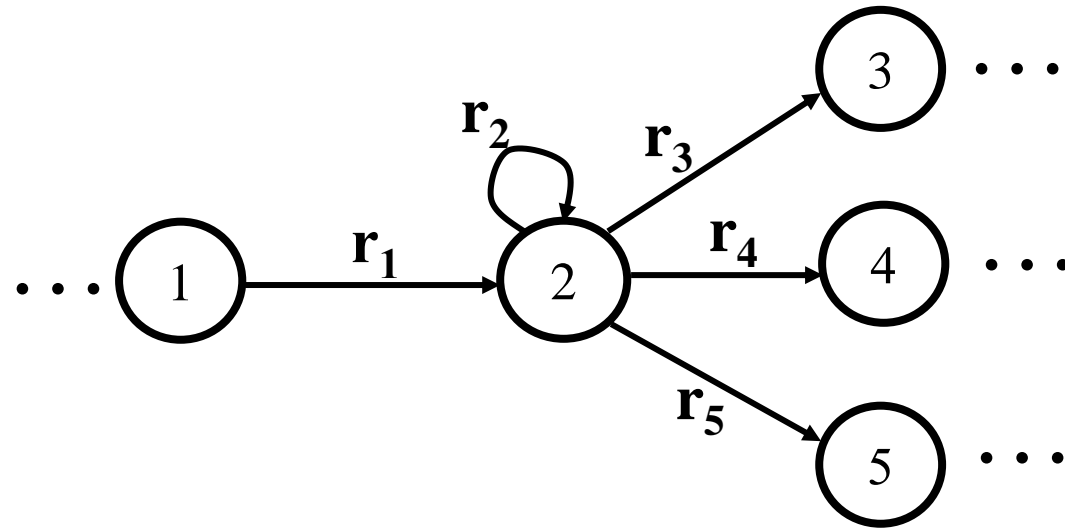
bypass and state elimination operation

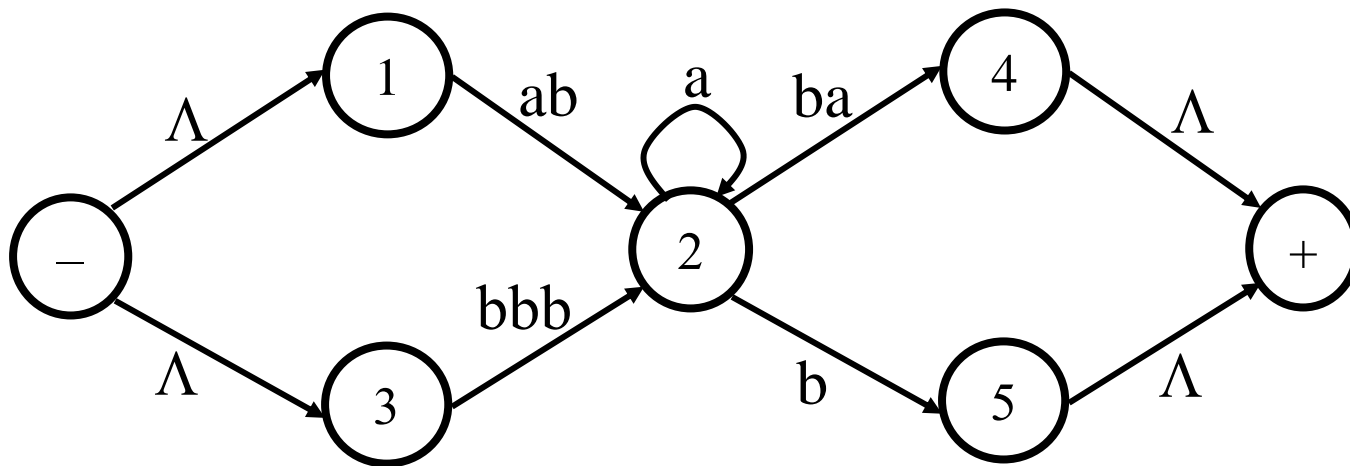




## Bypass state 2

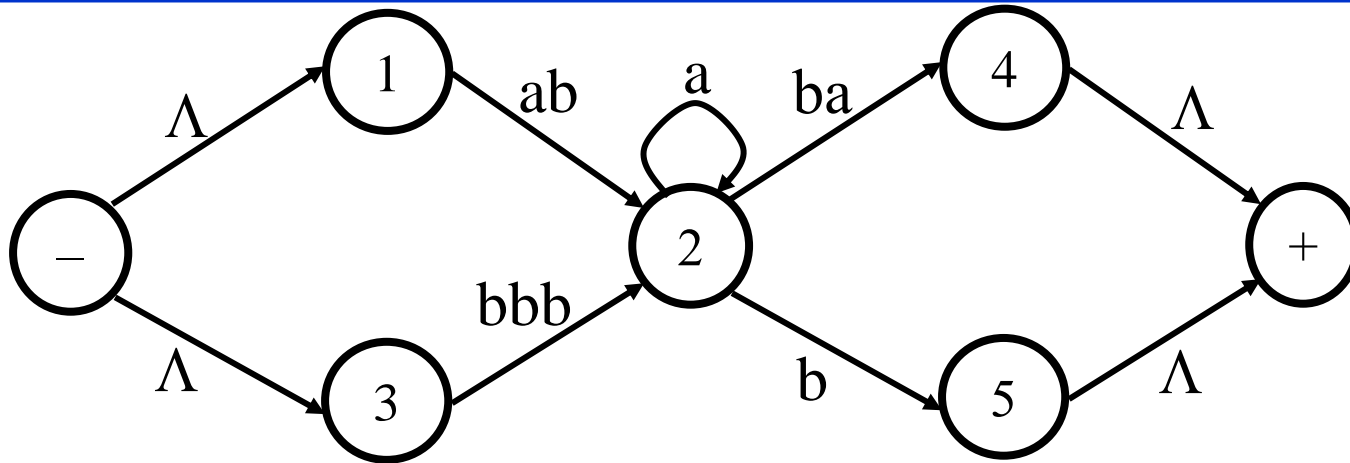
---





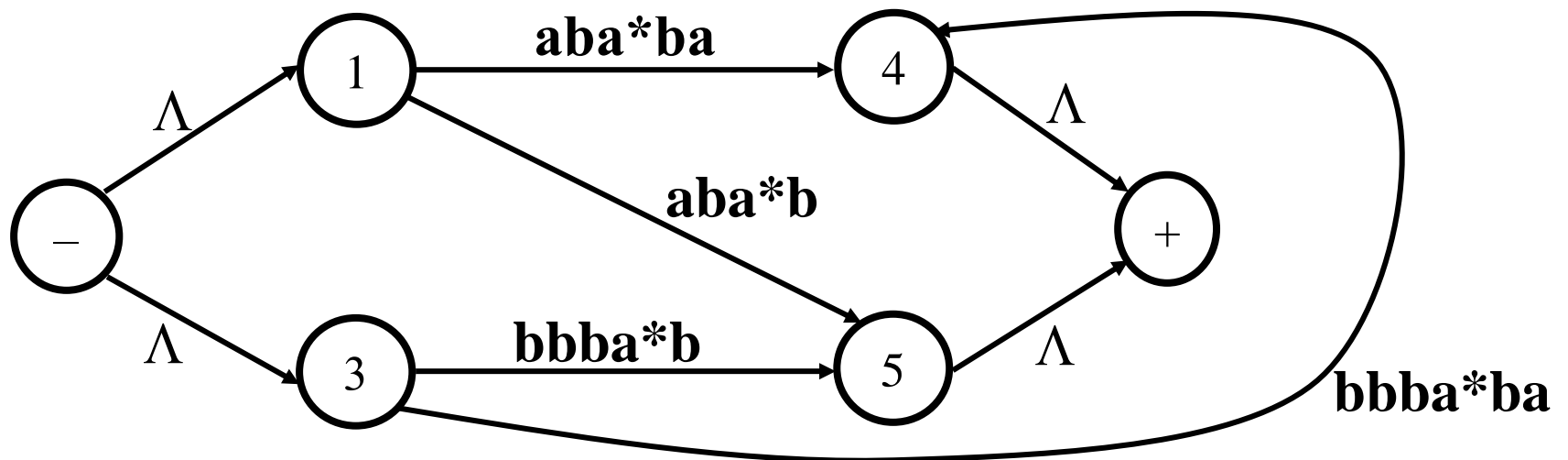
bypass state 2,

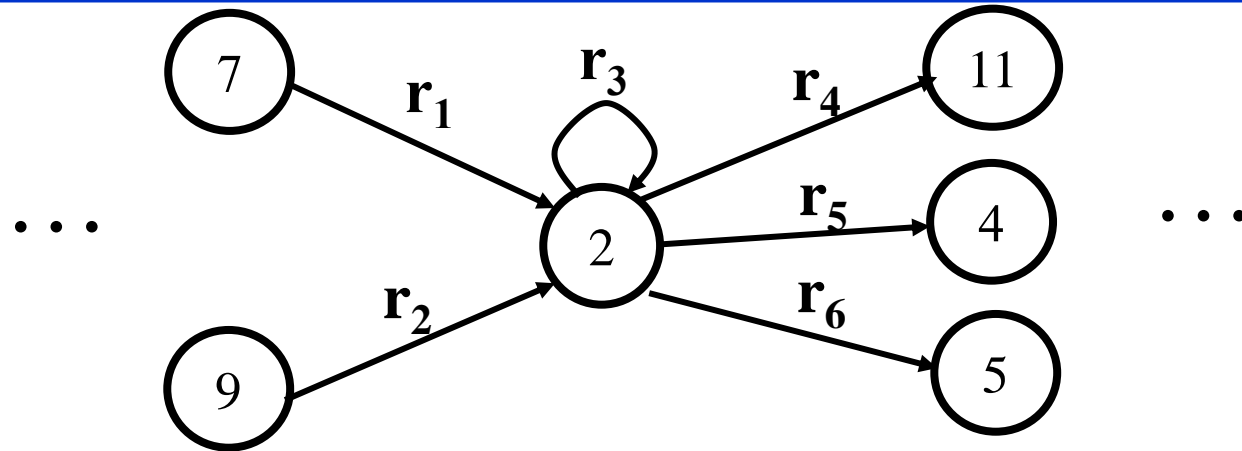
paths through state 2:  $1 \rightarrow 2 \rightarrow 4$ ,  $1 \rightarrow 2 \rightarrow 5$ ,  $3 \rightarrow 2 \rightarrow 4$ ,  $3 \rightarrow 2 \rightarrow 5$



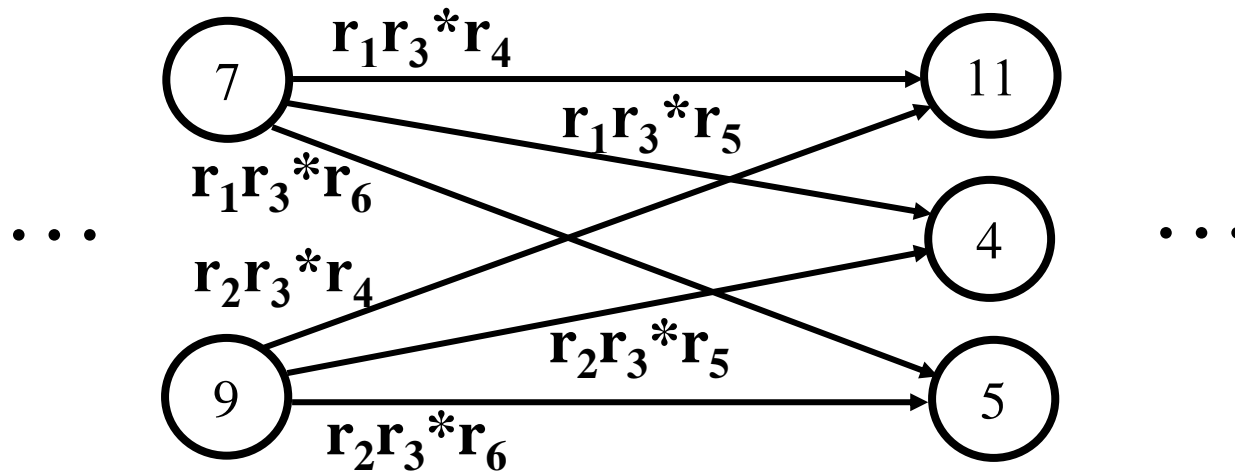
bypass state 2,

paths through state 2:  $1 \rightarrow 2 \rightarrow 4$ ,  $1 \rightarrow 2 \rightarrow 5$ ,  $3 \rightarrow 2 \rightarrow 4$ ,  $3 \rightarrow 2 \rightarrow 5$



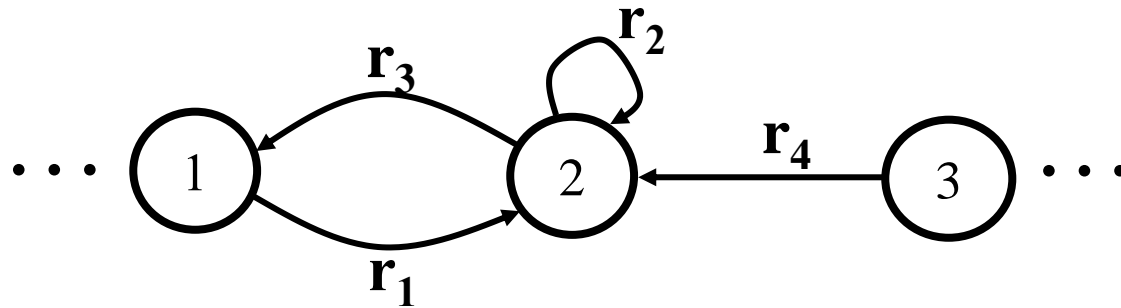


Even with many paths through state 2, always an equivalent GTG.

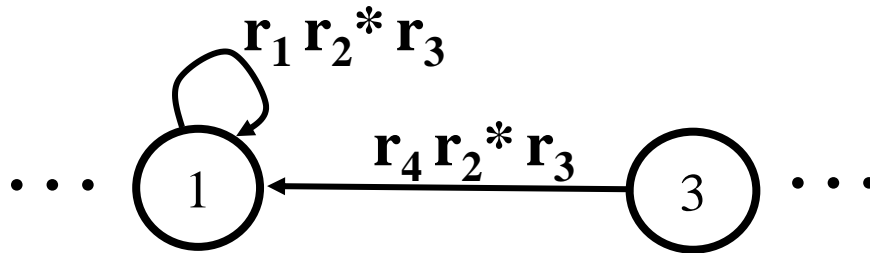


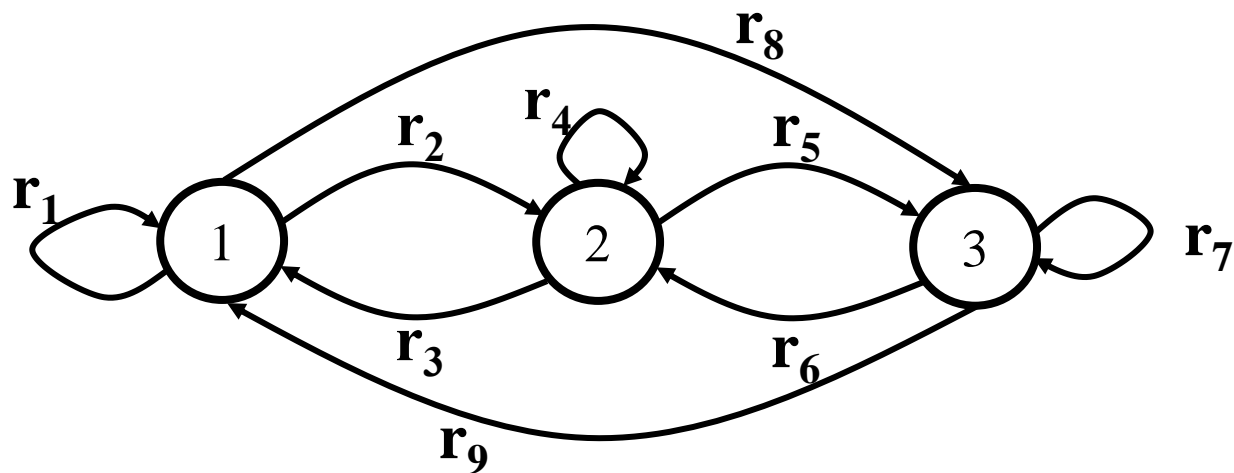
# Special cases

---

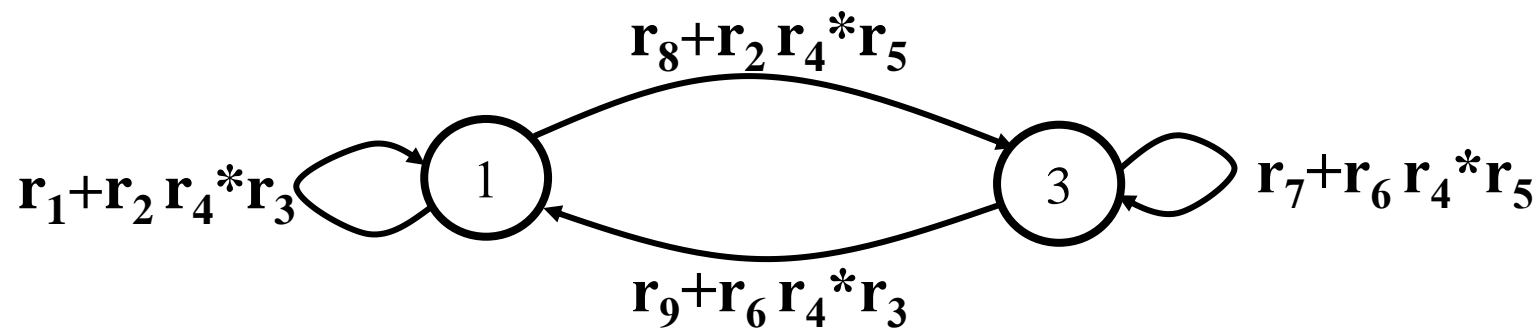


$$3 \rightarrow 2 \rightarrow 1, \quad 1 \rightarrow 2 \rightarrow 1$$



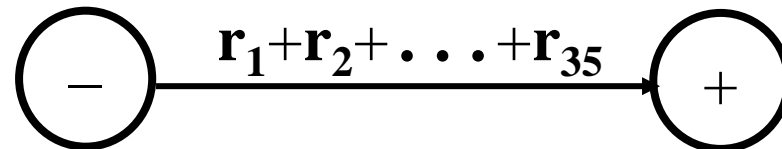
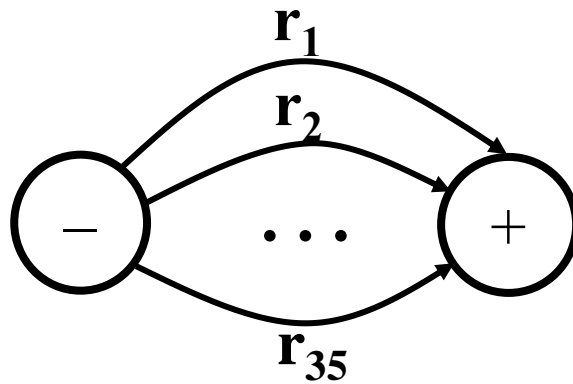


$3 \rightarrow 2 \rightarrow 1, 3 \rightarrow 2 \rightarrow 3, 1 \rightarrow 2 \rightarrow 1, 1 \rightarrow 2 \rightarrow 3$



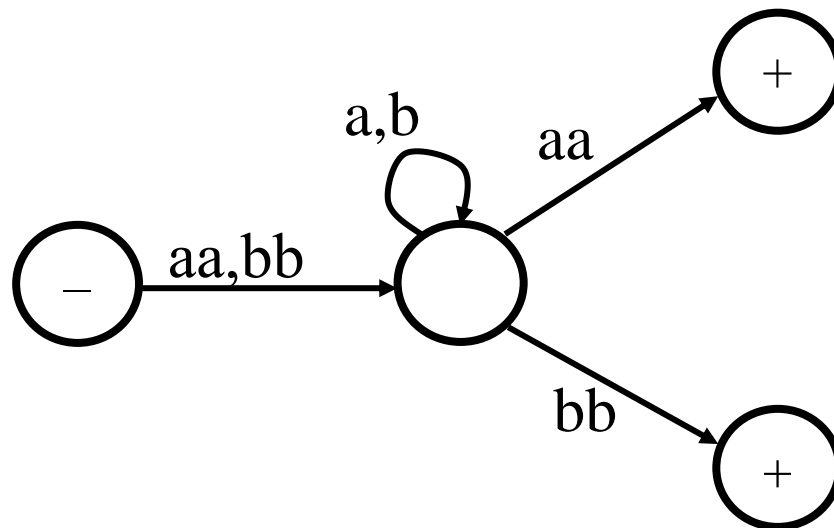
#### 4. Combine edges from start to end state

---



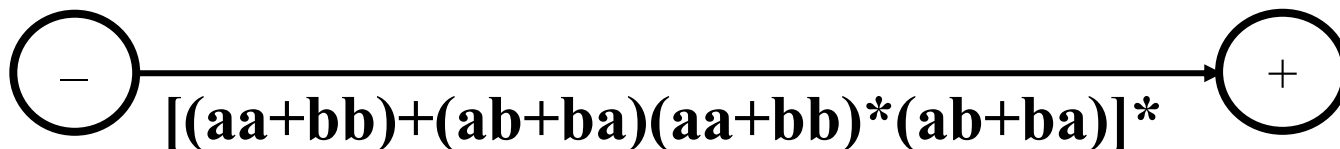
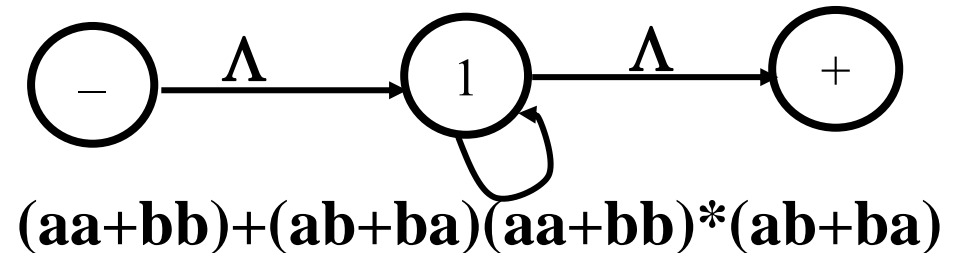
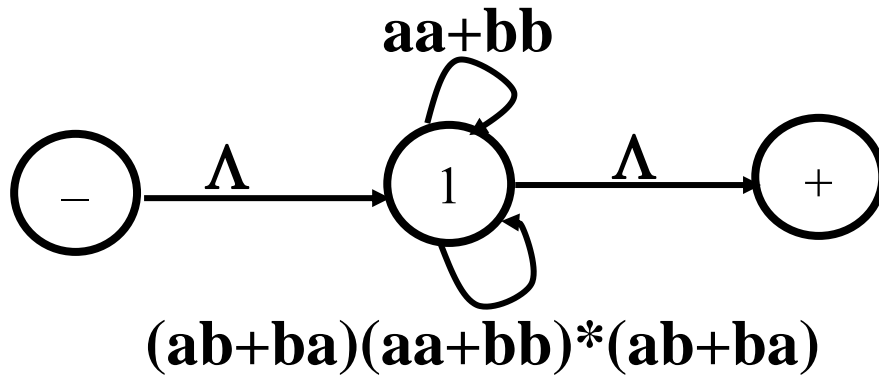
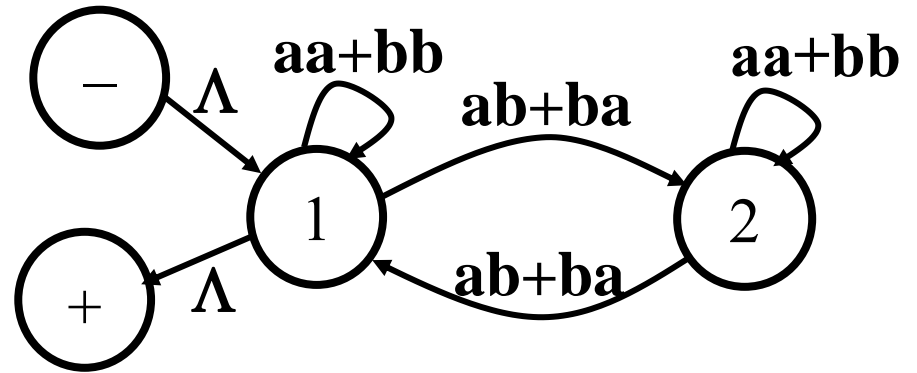
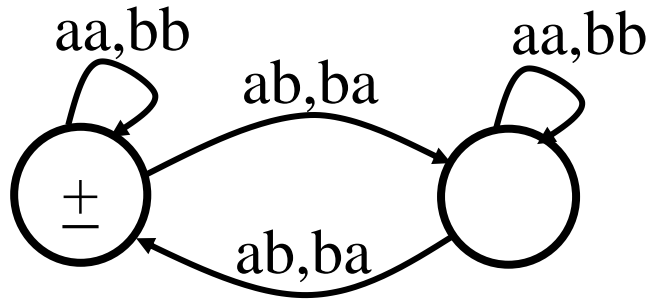
# Example 1

---



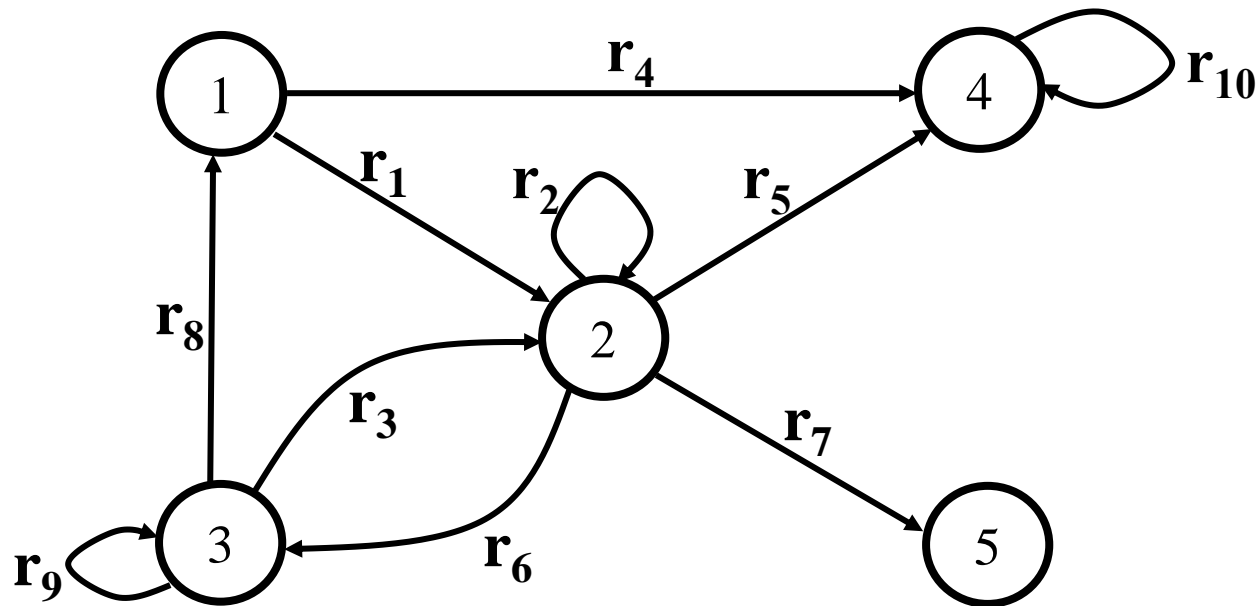


## Example 2: EVEN-EVEN



## Example 3

---



# Transition Graph $\rightarrow$ Regular Expression

---

- Algorithm (and proof)

1. Add (if necessary) a unique start state without incoming edges and a unique final state without outgoing edges.

For each state that is not a start state or a final state, repeat steps 2 and 3.

2. Do a bypass and state elimination operation.
3. Combine edges that have the same starting and ending state.
4. Combine the edges between the start state and the final state.  
The label on the only remaining edge is the regular expression result. If there is none, the language is  $\phi$ .

# Kleene's Theorem

---

Lemma 3: Every language that can be defined by a regular expression can also be defined by a finite automaton.

Proof: By constructive algorithm starting from the recursive definition of regular expressions.

- Remember: Given an alphabet  $\Sigma$ , the set of **regular expressions** is defined by the following rules.
  1. For every letter in  $\Sigma$ , the letter written in bold is a regular expression.  
 $\Lambda$  is a regular expression.
  2. If  $\mathbf{r_1}$  and  $\mathbf{r_2}$  are regular expressions, so is  $\mathbf{r_1+r_2}$ .
  3. If  $\mathbf{r_1}$  and  $\mathbf{r_2}$  are regular expressions, so is  $\mathbf{r_1 r_2}$ .
  4. If  $\mathbf{r_1}$  is a regular expression, so is  $\mathbf{r_1^*}$ .
  5. If  $\mathbf{r_1}$  is a regular expression, so is  $\mathbf{(r_1)}$ .
  6. Nothing else is a regular expression.

## Example: $(a+b)^*(aa+bb)(a+b)^*$

---

Build a finite automaton that accepts this language:	<u>Rule:</u>
1. The letter <b>a</b>	1
2. The letter <b>b</b>	1
3. The word <b>aa</b> (using 1)	3
4. The word <b>bb</b> (using 2)	3
5. The expression <b>aa+bb</b> (using 3 and 4)	2
6. The expression <b>a+b</b> (using 1 and 2)	2
7. The expression <b>(a+b)*</b> (using 6)	5,4
8. The expression <b>(a+b)*(aa+bb)</b> (using 7 and 5)	5,3
9. The expression <b>(a+b)*(aa+bb)(a+b)*</b> (using 8 and 7)	3

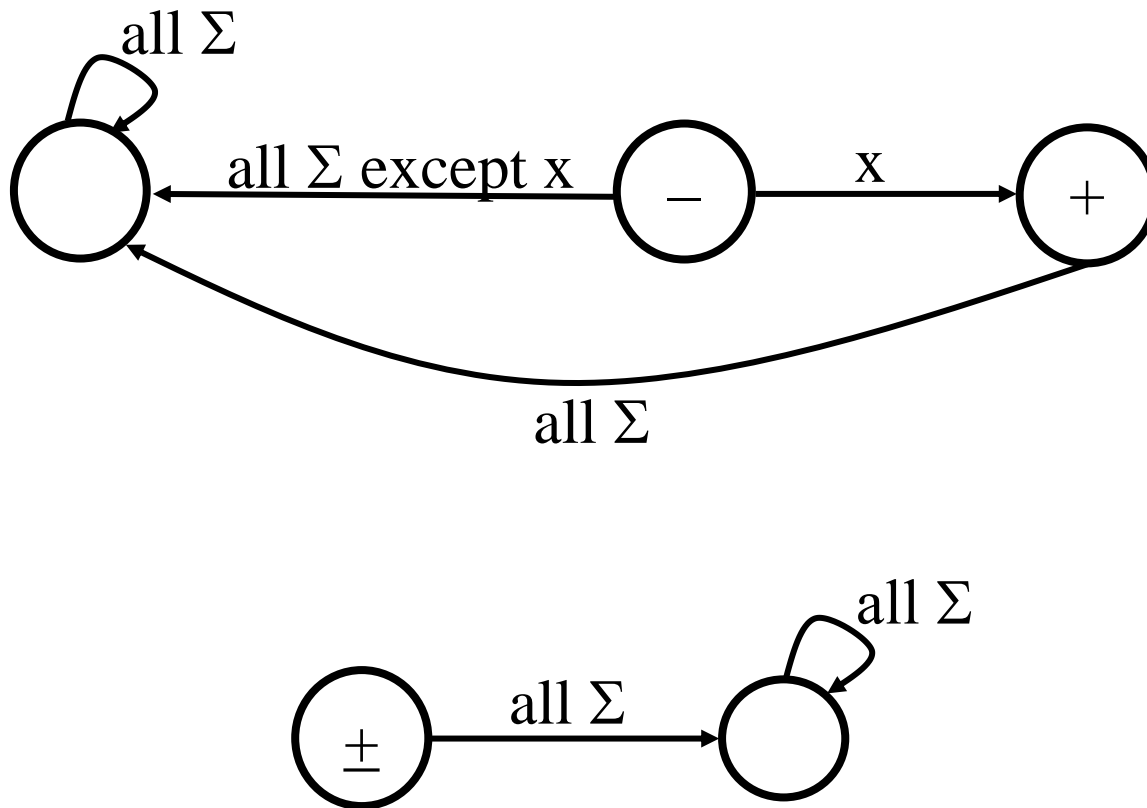
# Rule 1: $\Lambda$ and the letters in $\Sigma$

---

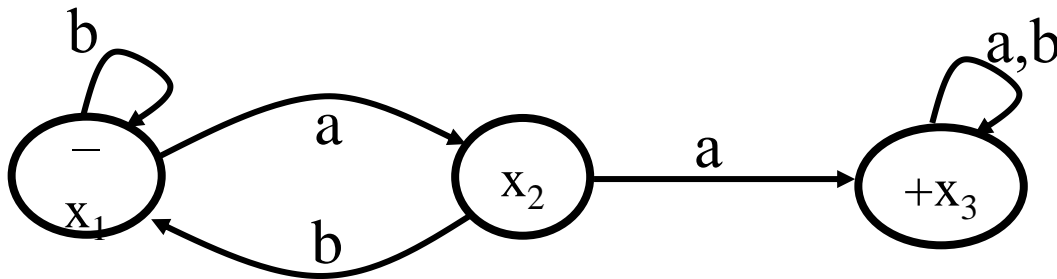
Rule 1:

There is an FA that accepts any particular letter in  $\Sigma$ .

There is an FA that accepts only the word  $\Lambda$ .



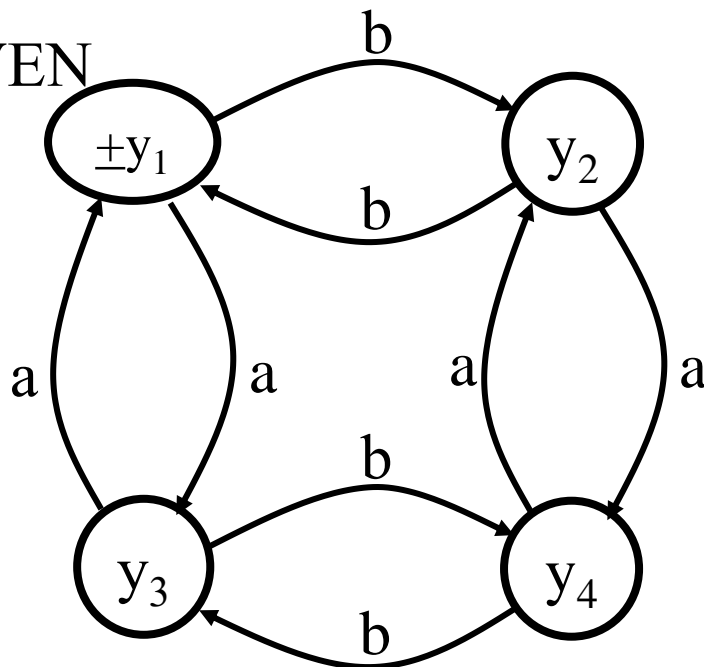
## Rule 2: $r_1 + r_2$ , Example 1



All words containing aa

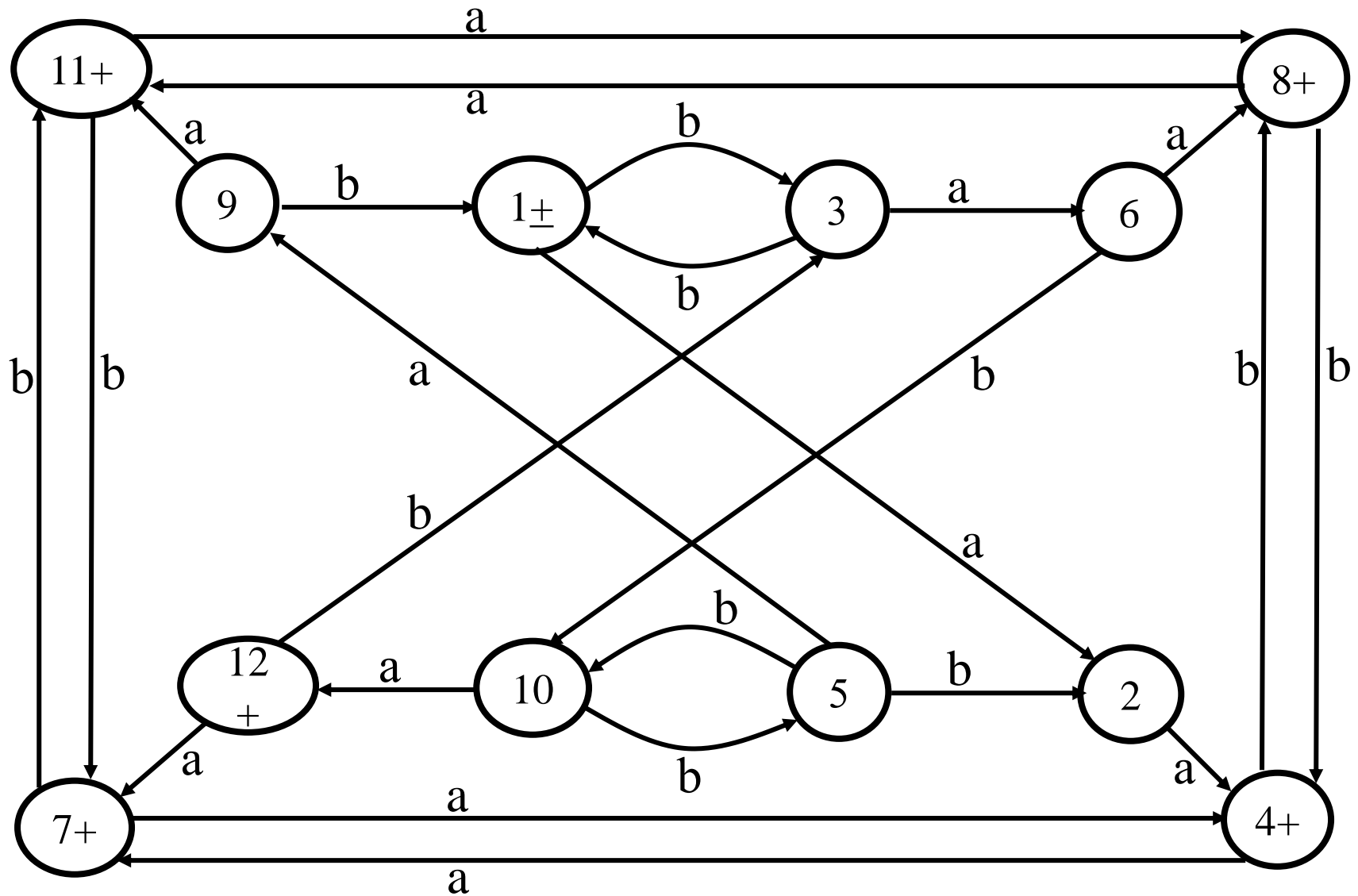
	a	b
$-x_1$	$x_2$	$x_1$
$x_2$	$x_3$	$x_1$
$+x_3$	$x_3$	$x_3$

EVEN-EVEN



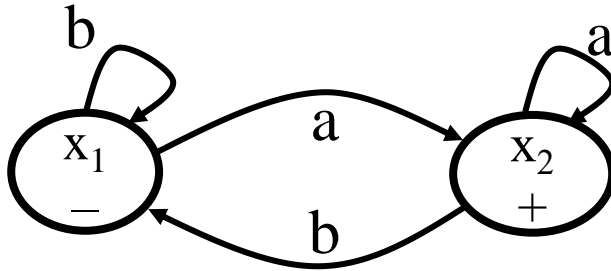
	a	b
$\pm y_1$	$y_3$	$y_2$
$y_2$	$y_4$	$y_1$
$y_3$	$y_1$	$y_4$
$y_4$	$y_2$	$y_3$

Result:  $r_1 + r_2$



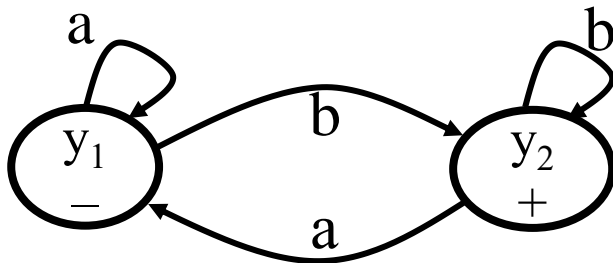


## Example 2



	a	b
$-x_1$	$x_2$	$x_1$
$+x_2$	$x_2$	$x_1$

(words ending in a)



	a	b
$-y_1$	$y_1$	$y_2$
$+y_2$	$y_1$	$y_2$

(words ending in b)

Lemma 3: Every language that can be defined by a regular expression can also be defined by a finite automaton.

---

Proof: By constructive algorithm starting from the recursive definition of regular expressions

- ✓1. There is an FA that accepts only the empty word ( $\Lambda$ ) and an FA that accepts only a single letter.
- ✓2. If there is an FA that accepts the language defined by  $\mathbf{r}_1$  and an FA that accepts the language defined by  $\mathbf{r}_2$ , then there is an FA that accepts the language  $\mathbf{r}_1 + \mathbf{r}_2$ .
- 3. If there is an FA that accepts the language defined by  $\mathbf{r}_1$  and an FA that accepts the language defined by  $\mathbf{r}_2$ , then there is an FA that accepts the language defined by their concatenation  $\mathbf{r}_1\mathbf{r}_2$ .
- 4. If there is an FA that accepts the language defined by  $\mathbf{r}$  then there is an FA that accepts the language defined by  $\mathbf{r}^*$ .
- ✓5. If there is an FA that accepts the language defined by  $\mathbf{r}$  then there is an FA that accepts the language defined by  $(\mathbf{r})$ .

Thus for every regular expression, we can construct an FA.

# Algorithm 1 for $r_1+r_2$

---

## Input:

FA 1: alphabet:  $\Sigma$     states:  $x_1, x_2, x_3, \dots$     start state:  $x_1$

FA 2: alphabet:  $\Sigma$     states:  $y_1, y_2, y_3, \dots$     start state:  $y_1$

plus final states and transitions

## The new FA:

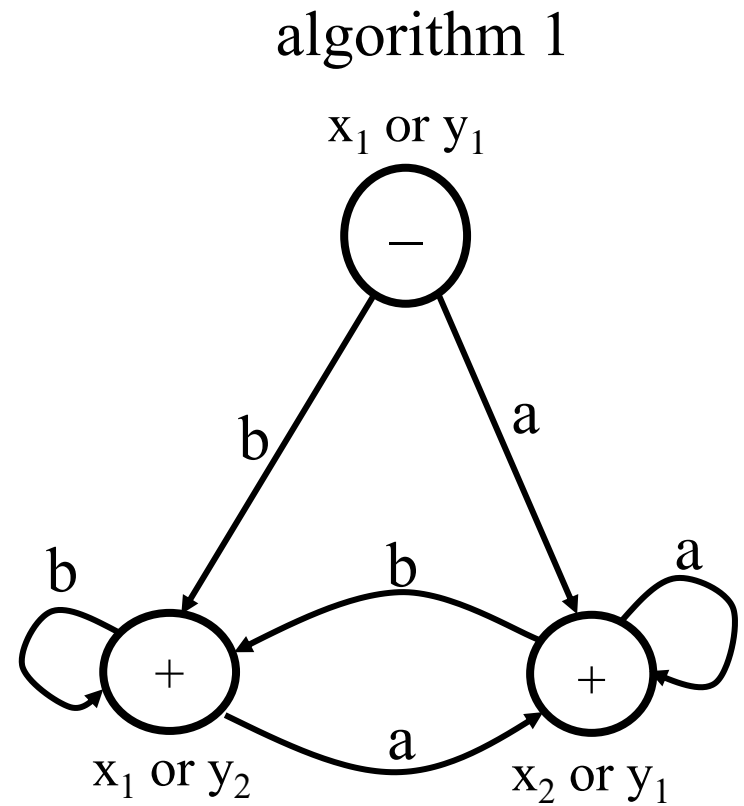
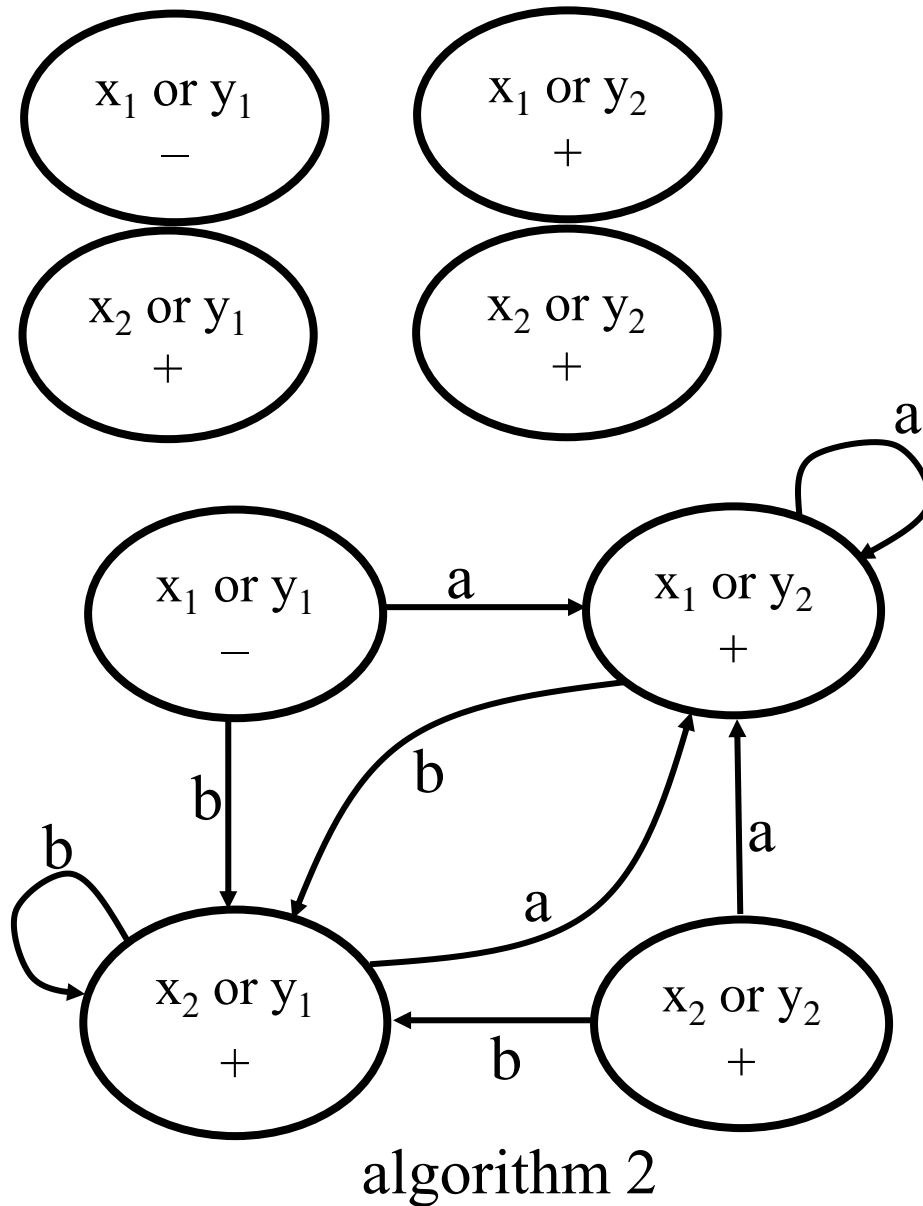
alphabet:  $\Sigma$     states:  $z_1, z_2, z_3, \dots$     start state:  $x_1$  or  $y_1$

transitions: if  $z_i = (x_j \text{ or } y_k)$  and  $x_j \xrightarrow{p} x_{\text{new}}$  and  $y_k \xrightarrow{p} y_{\text{new}}$

then  $z_{\text{new}} = (x_{\text{new}} \text{ or } y_{\text{new}})$  for input  $p$ .

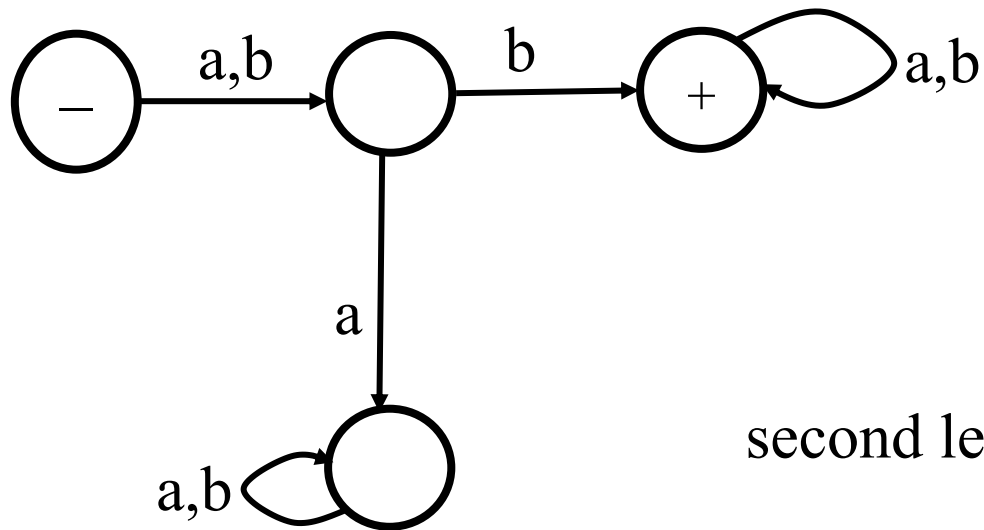
If  $x_{\text{new}}$  or  $y_{\text{new}}$  is a final state, then  $z_{\text{new}}$  is a final state.

## Algorithm 2: Put all pairs ( $x_i$ or $x_j$ ) in the transition table

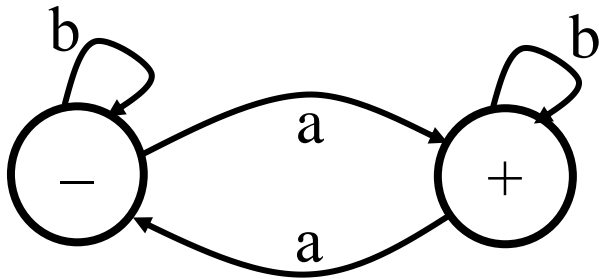


## Rule 3: $r_1 r_2$ , Example 1

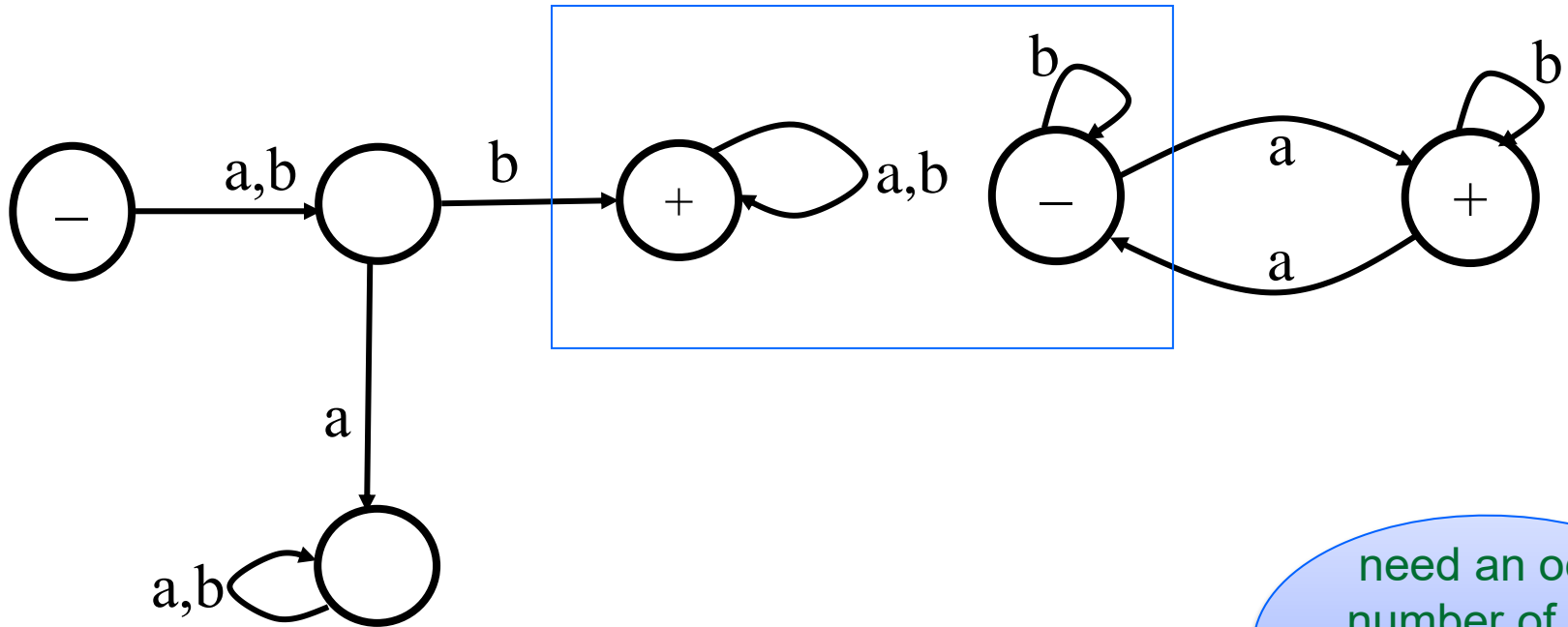
---



second letter is b



odd number of a's



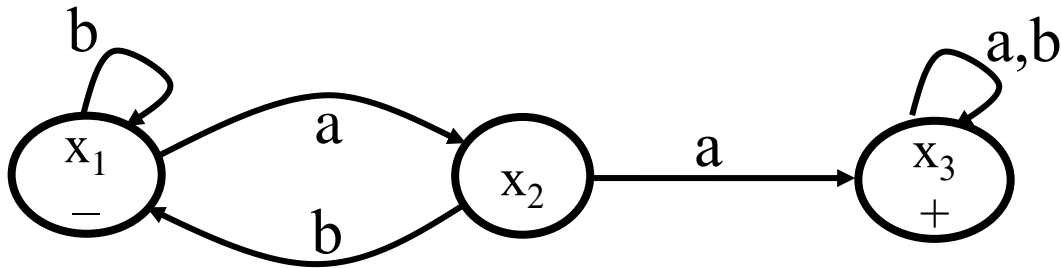
ab|abbaa  
abab|bab

~~ab|abbab~~  
aba|bbab

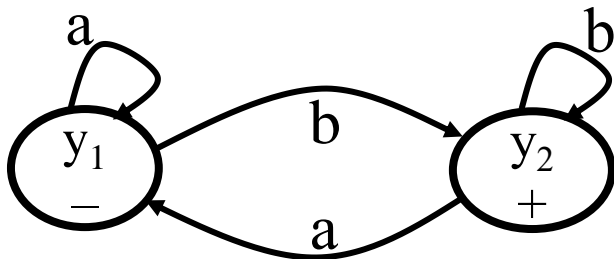
need an odd  
number of **a**'s  
in second  
factor

## Example 2

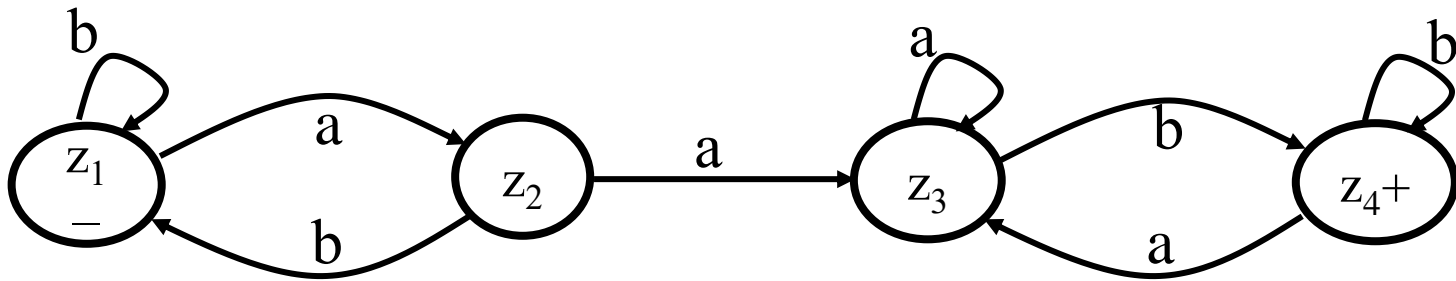
---



$\mathbf{r}_1$ : all words with aa



$\mathbf{r}_2$ : words ending in b





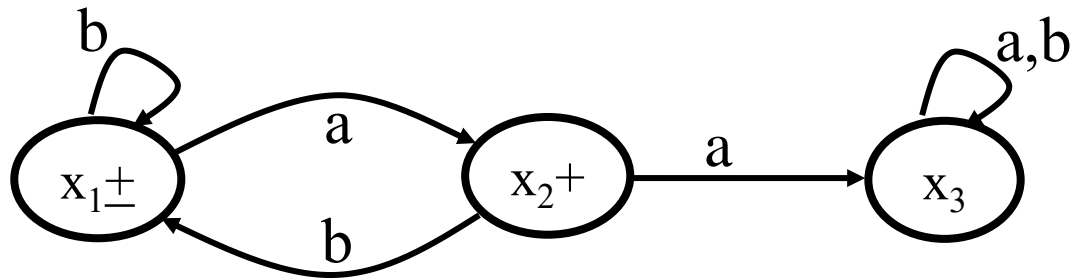
## Rule 3: Concatenation: Summary of Algorithm

---

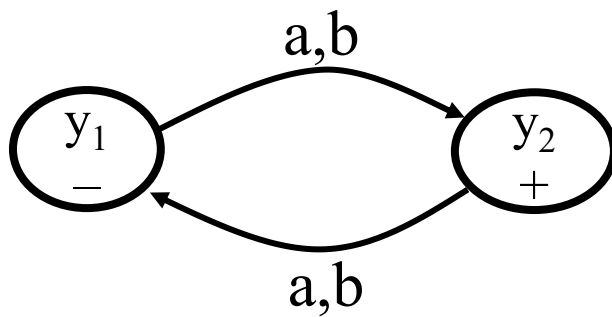
1. Add a state  $z$  for every state of the first automaton that is possible to go through before arriving at a final state.
2. For each final state, add a state  $z = (x \text{ or } y_1)$ , where  $y_1$  is the start state of the second automaton.
3. Starting from the states added at step 2, add states:
$$z = \begin{cases} x & (\text{state such that execution continues on 1}^{\text{st}} \text{ automaton}) \\ y_1 & \text{if } x \text{ a final state of the first automaton} \\ \{y \mid \text{execution continues on 2}^{\text{nd}} \text{ automaton}\} \end{cases}$$
4. Label every state that contains a final state from the second automaton as a final state.

## Example 3

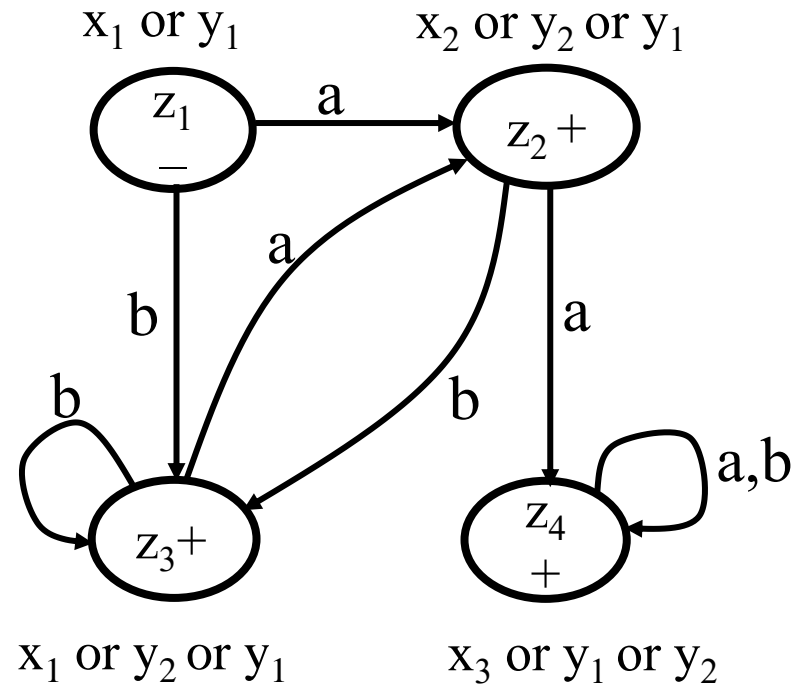
---



$r_1$ : no double a



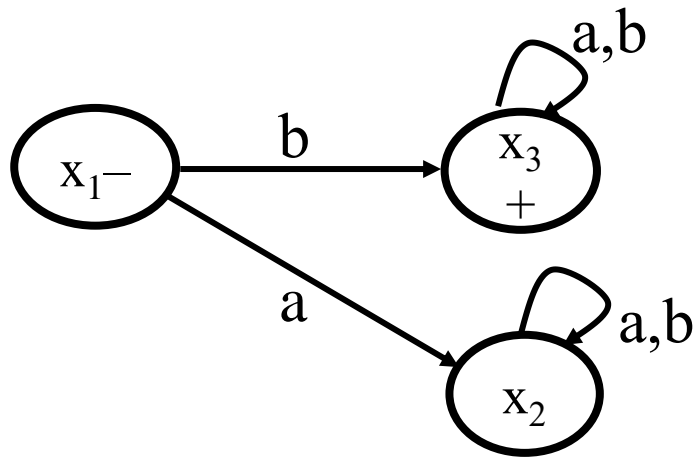
$r_2$ : odd number of letters



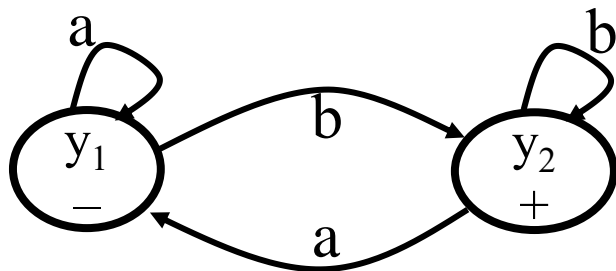
$\mathbf{r_1r_2}$ : all words except  $\Lambda$

## Example 4

---

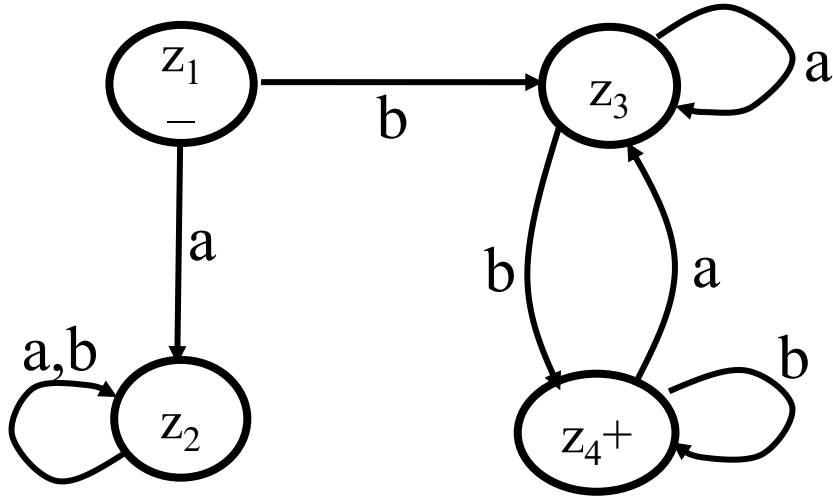


$\mathbf{r}_1$ : words starting with b

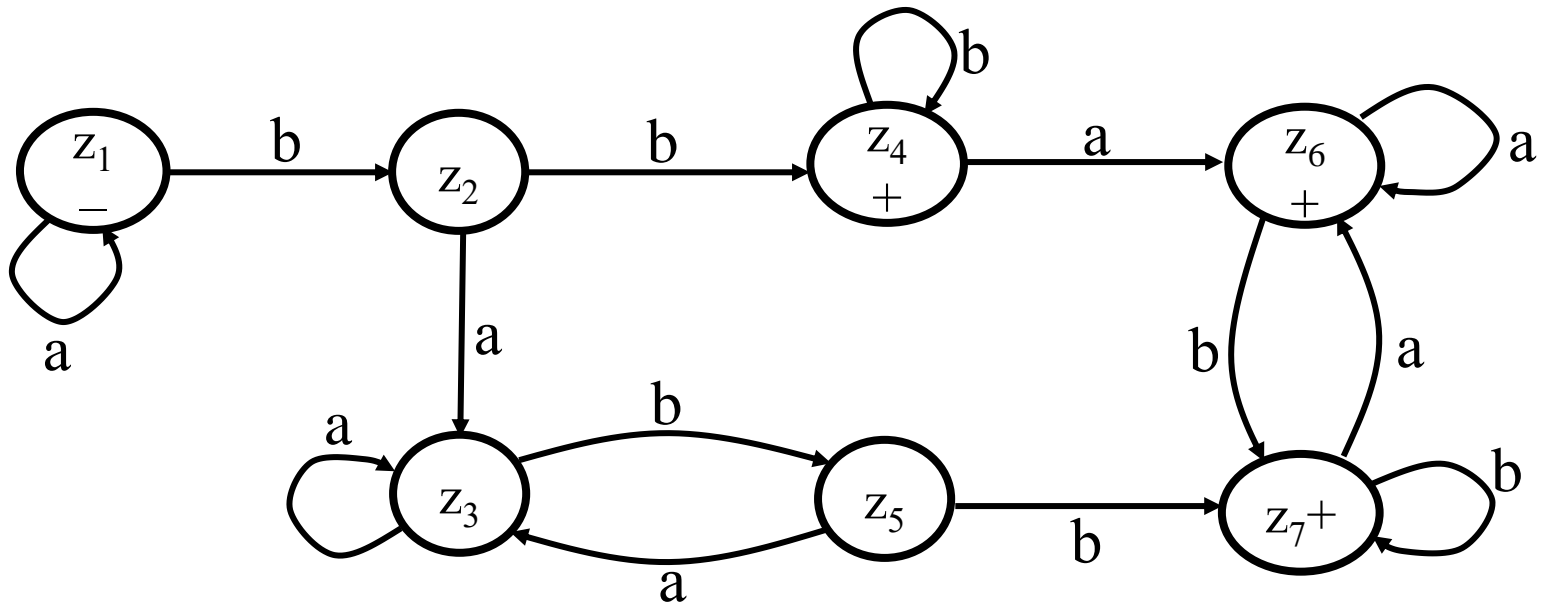


$\mathbf{r}_2$ : words ending in b

$\mathbf{r}_1\mathbf{r}_2$

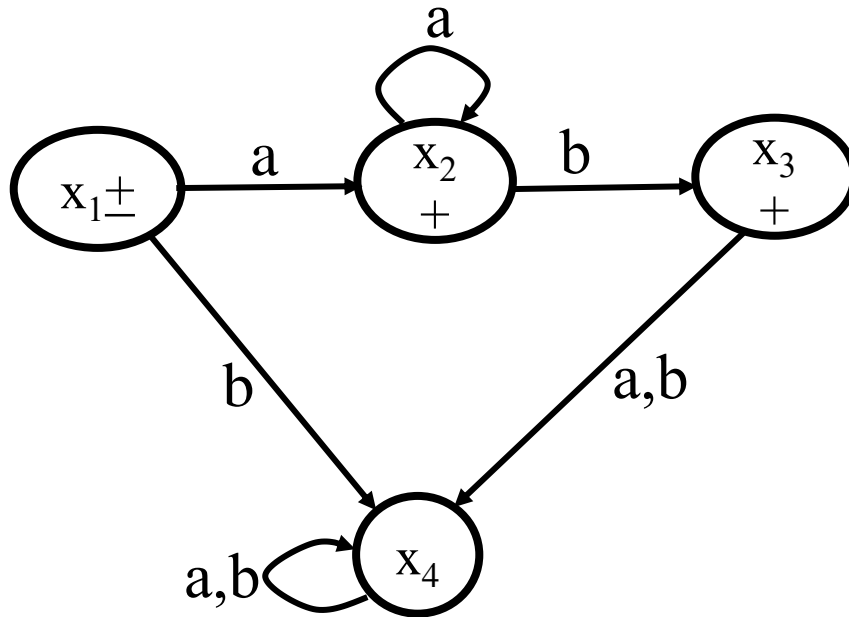


$\mathbf{r}_2\mathbf{r}_1$



## Rule 4: $r^*$ , Example 1

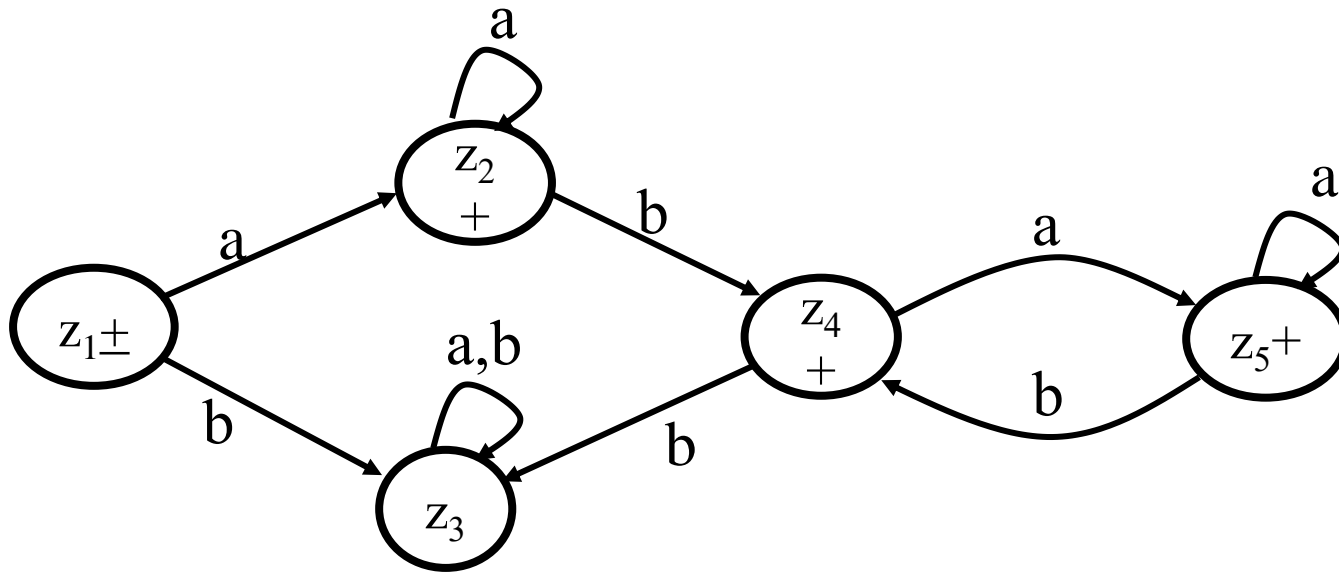
---



**$r$ :  $a^* + aa^*b$**

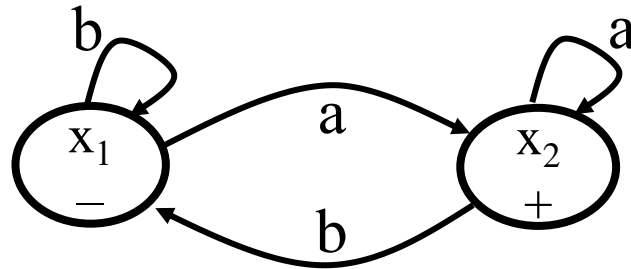
**$r^*$ :  $(a^* + aa^*b)^*$**

words without double b, and that do not start with b.



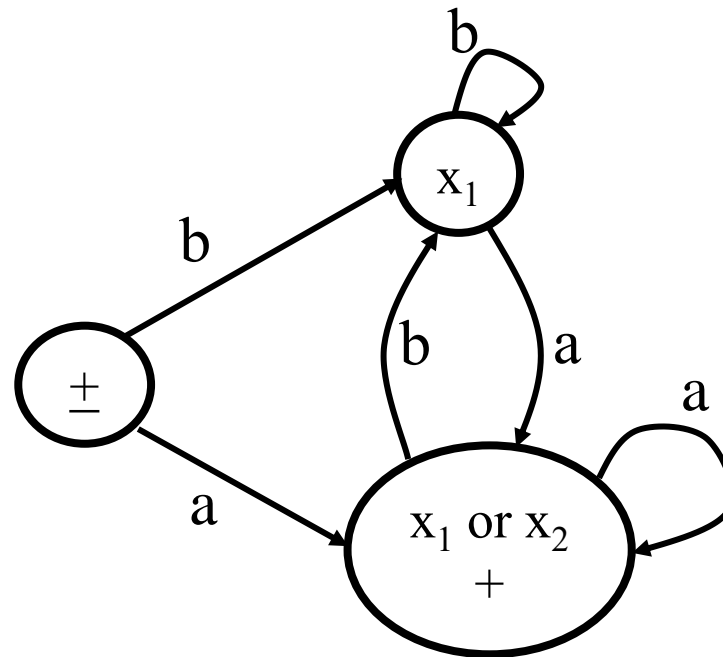
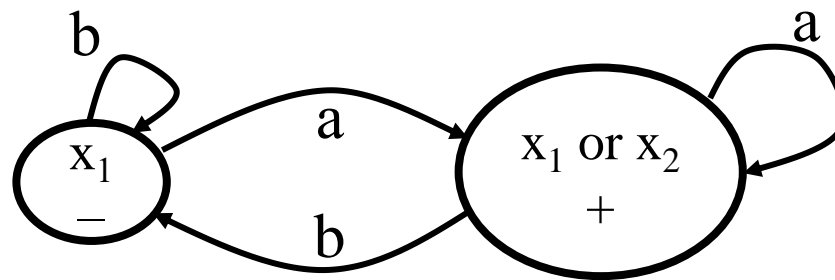
## Example 2

---



Words ending in a





## Rule 4: Kleene Star: Algorithm

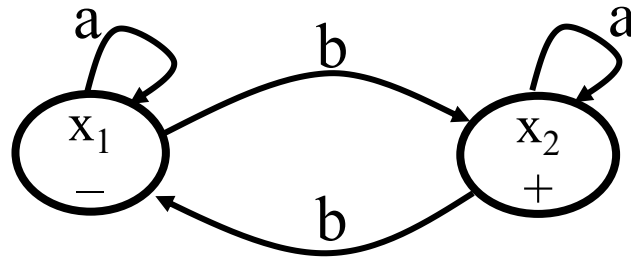
---

Given: an FA whose states are  $\{x_1, x_2, x_3, \dots\}$

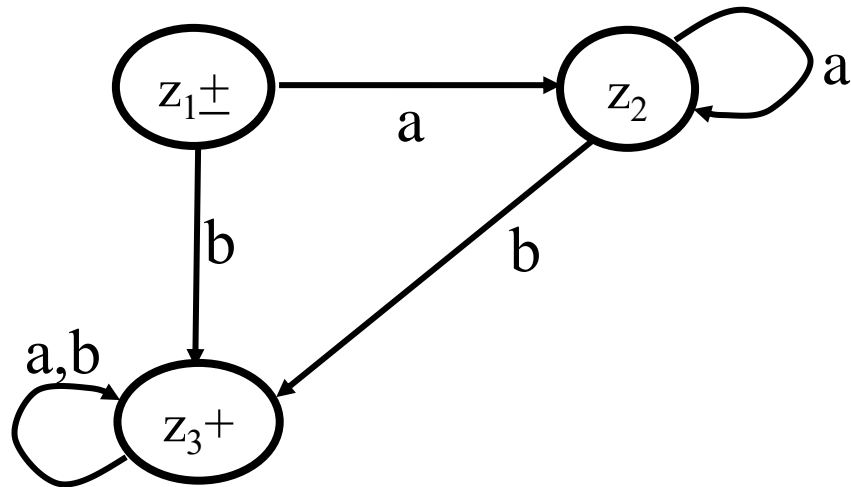
- For every nonempty subset of states, create a state of the new FA. Remove any subset that contains a final state but not the start state.
- Make the transition table for all the new states.
- Add a  $\pm$  state. Connect it to the same states as the original start state was connected to using the same transitions.
- The final states must be those that contain at least one final state from the original FA.

## Example 3

---



Words with an odd number of b's.



$\Lambda$  and words with at least one  $b$ .

# Nondeterministic Finite Automata

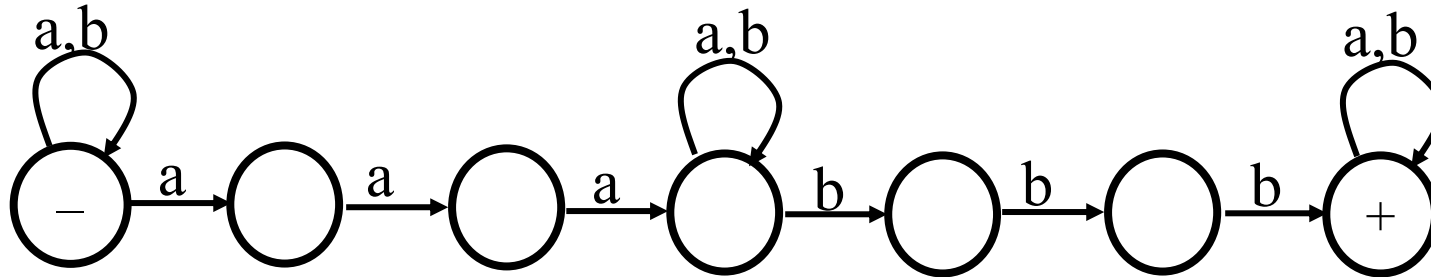
---

- A **nondeterministic finite automaton (NFA)** is:
  1. a finite set of states, one of which is designated as the start state, and some (maybe none) of which are designated as the final states
  2. an **alphabet**  $\Sigma$  of input letters
  3. a finite set of **transitions** that show how to go to a new state, for some pairs of state and letters

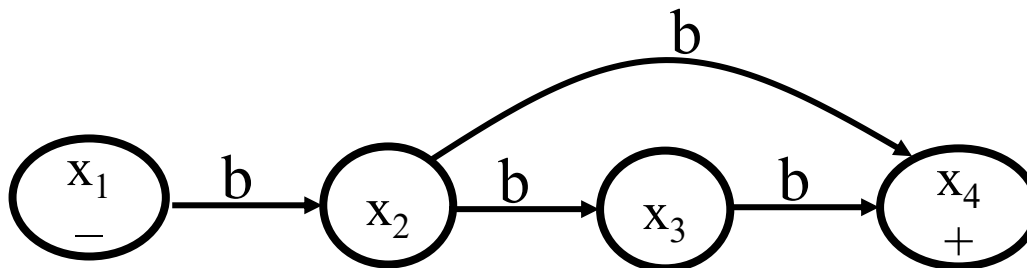
## Remarks:

- Each pair is one state and one letter (no  $\Lambda$ ).
- Can have 0, 1, or more transitions for a single pair.
- Every finite automaton is a nondeterministic finite automaton. Every nondeterministic finite automaton is a transition graph.

# Examples of Nondeterministic Finite Automata



**$(a+b)^*aaa(a+b)^*bbb(a+b)^*$**



**$bb+bbb$**

---

Theorem: Every language that can be defined by a nondeterministic finite automaton can also be defined by a deterministic finite automaton.

Proof (1): Every nondeterministic finite automaton is a transition graph.

By lemma 2: transition graph  $\rightarrow$  regular expression

By lemma 3: regular expression  $\rightarrow$  finite automaton

Proof (2): By constructive algorithm

---

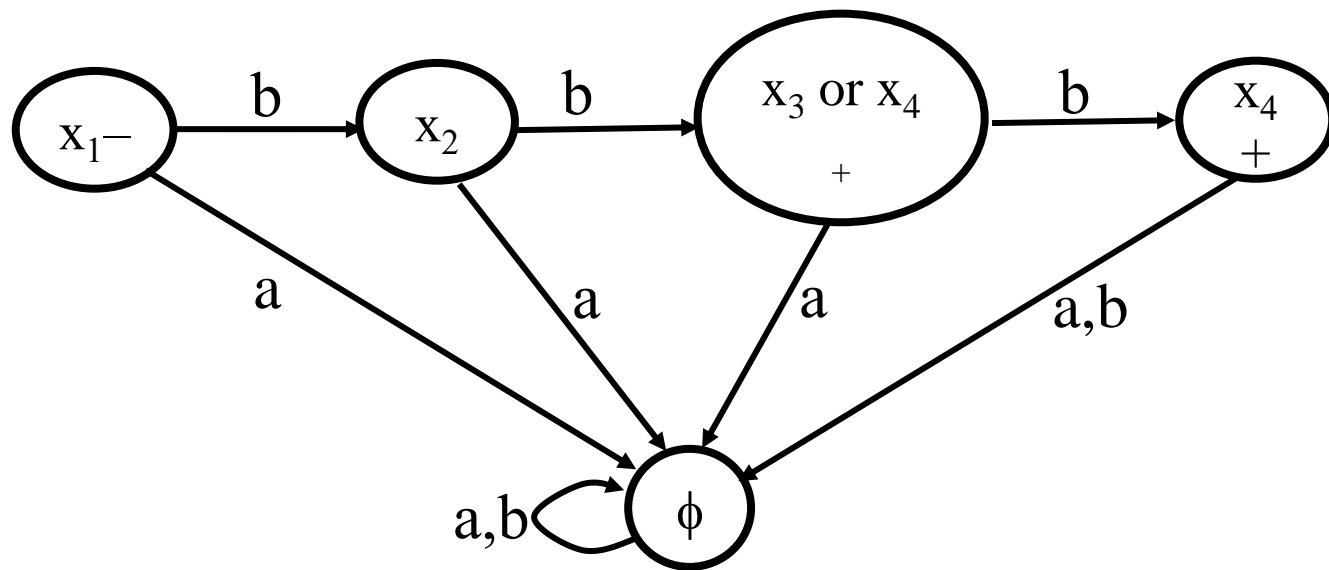
Algorithm: nondeterministic automaton  $\rightarrow$  deterministic automaton (FA)

Given: a nondeterministic automaton whose states are

$$\{x_1, x_2, x_3, \dots\}$$

1. For every subset of states, create a state of the new FA.
2. Make the transition table for all the new states (or just the new states that can be entered).
3. Add a state  $\phi$ . Add transitions that loop back to itself for all letters of the alphabet. For each new state, if there is no transition for letter  $p$ , add one that goes to the  $\phi$  state.
4. The final states must be those that contain at least one final state from the original nondeterministic finite automaton.





**bb+bbb**

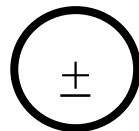
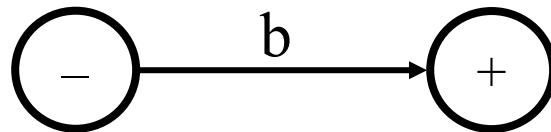
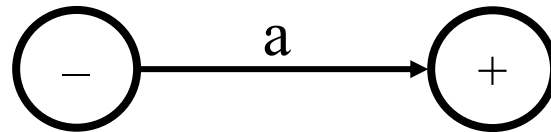
---

Lemma 3: Every language that can be defined by a regular expression can also be defined by a finite automaton.

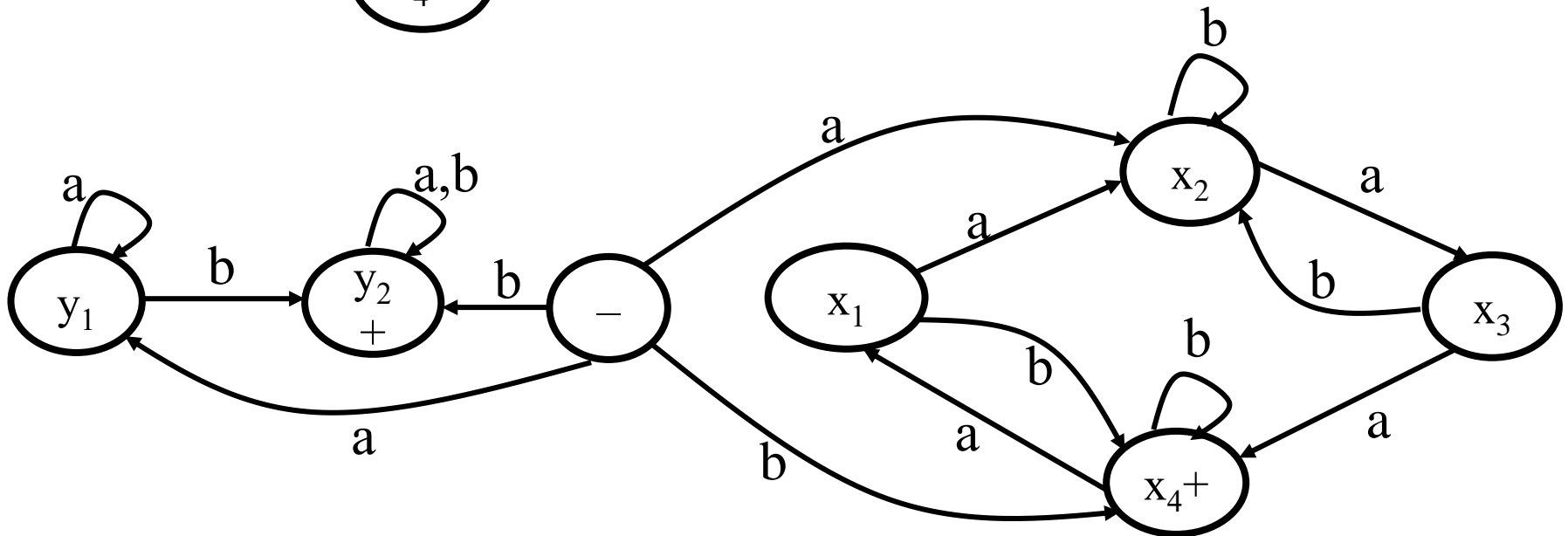
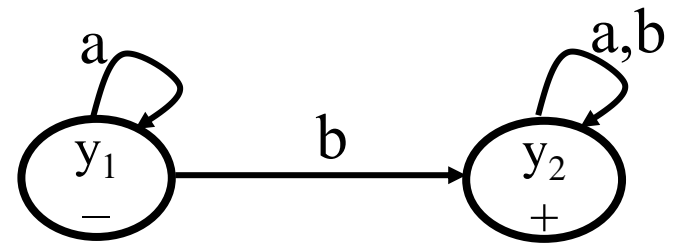
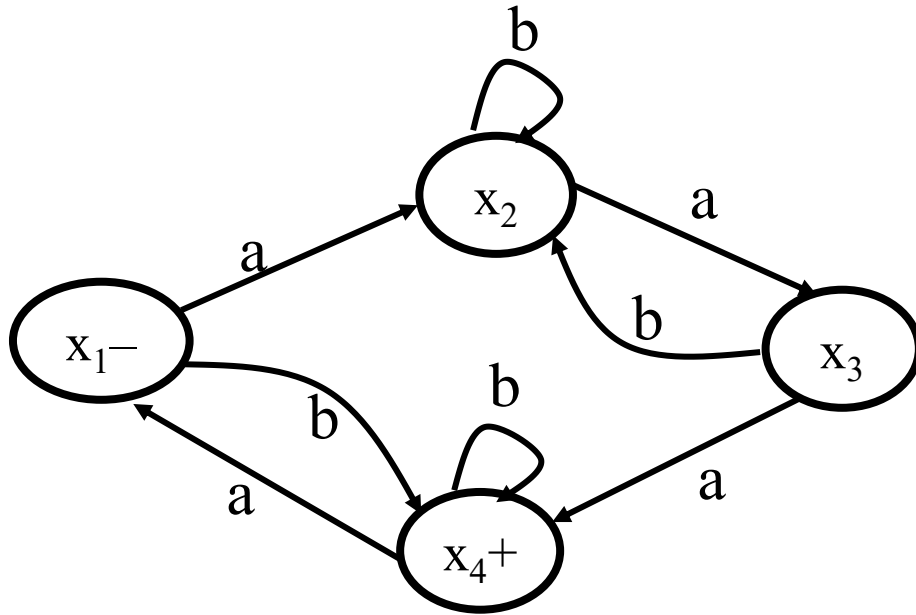
Proof: By constructive algorithm starting from the recursive definition of regular expressions, we build a nondeterministic finite automaton. Then, by the most recent theorem, it is possible to then build a finite automaton.

# Rule 1: $\Lambda$ and the letters in $\Sigma$

---



## Rule 2: $r_1 + r_2$



## Rule 3: $r_1 r_2$

---

