

# FSA and regular languages

Data Structures and Algorithms for Computational Linguistics III  
(ISCL-BA-07)

Çağrı Çöltekin

`ccoltekin@sfs.uni-tuebingen.de`

University of Tübingen  
Seminar für Sprachwissenschaft

Winter Semester 2022/23

# Languages and automata

- Recognizing strings from a language defined by a grammar is a fundamental question in computer science
- The efficiency of computation, and required properties of computing device depends on the grammar (and the language)
- A well-known hierarchy of grammars both in computer science and linguistics is the *Chomsky hierarchy*
- Each grammar in the Chomsky hierarchy corresponds to an abstract computing device (an automaton)
- The class of *regular grammars* are the class that corresponds to *finite state automata*

# How to describe a language?

## Formal grammars

A formal *grammar* is a finite specification of a (formal) language.

- Since we consider languages as sets of strings, for a finite language, we can (conceivably) list all strings
- How to define an infinite language?
- Is the definition  $\{ba, baa, baaa, baaaa, \dots\}$  ‘formal enough’?
- Using regular expressions, we can define it as  $baa^*$
- But we will introduce a more general method for defining languages

# Phrase structure grammars

- A phrase structure grammar is a generative device
- If a given string can be generated by the grammar, the string is in the language
- The grammar generates *all* and the *only* strings that are valid in the language
- A phrase structure grammar has the following components
  - $\Sigma$  A set of *terminal* symbols
  - $N$  A set of *non-terminal* symbols
  - $S \in N$  A special non-terminal, called the start symbol
  - $R$  A set of *rewrite rules* or *production rules* of the form:

$$\alpha \rightarrow \beta$$

which means that the sequence  $\alpha$  can be rewritten as  $\beta$  (both  $\alpha$  and  $\beta$  are sequences of terminal and non-terminal symbols)

# Chomsky hierarchy and automata

<i>Grammar class</i>	<i>Rules</i>	<i>Automata</i>
Unrestricted grammars	$\alpha \rightarrow \beta$	Turing machines
Context-sensitive grammars	$\alpha A \beta \rightarrow \alpha \gamma \beta$	Linear-bounded automata
Context-free grammars	$A \rightarrow \alpha$	Pushdown automata
Regular grammars	$\begin{array}{c c} \overline{A \rightarrow a} & \overline{A \rightarrow a} \\ \overline{A \rightarrow aB} & \overline{A \rightarrow B a} \end{array}$	Finite state automata

## Regular grammars: definition

A regular grammar is a tuple  $G = (\Sigma, N, S, R)$  where

$\Sigma$  is an alphabet of terminal symbols

$N$  are a set of non-terminal symbols

$S$  is a special 'start' symbol  $\in N$

$R$  is a set of rewrite rules following one of the following patterns ( $A, B \in N$ ,  $a \in \Sigma$ ,  $\epsilon$  is the empty string)

### Left regular

1.  $A \rightarrow a$
2.  $A \rightarrow Ba$
3.  $A \rightarrow \epsilon$

### Right regular

1.  $A \rightarrow a$
2.  $A \rightarrow aB$
3.  $A \rightarrow \epsilon$

# Regular languages: some properties/operations

$\mathcal{L}_1 \mathcal{L}_2$  Concatenation of two languages  $\mathcal{L}_1$  and  $\mathcal{L}_2$ : any sentence of  $\mathcal{L}_1$  followed by any sentence of  $\mathcal{L}_2$

$\mathcal{L}^*$  Kleene star of  $\mathcal{L}$ :  $\mathcal{L}$  concatenated with itself 0 or more times

$\mathcal{L}^R$  Reverse of  $\mathcal{L}$ : reverse of any string in  $\mathcal{L}$

$\overline{\mathcal{L}}$  Complement of  $\mathcal{L}$ : all strings in  $\Sigma_{\mathcal{L}}^*$  except the ones in  $\mathcal{L}$  ( $\Sigma_{\mathcal{L}}^* - \mathcal{L}$ )

$\mathcal{L}_1 \cup \mathcal{L}_2$  Union of languages  $\mathcal{L}_1$  and  $\mathcal{L}_2$ : strings that are in any of the languages

$\mathcal{L}_1 \cap \mathcal{L}_2$  Intersection of languages  $\mathcal{L}_1$  and  $\mathcal{L}_2$ : strings that are in both languages

Regular languages are closed under all of these operations.

# Three ways to define a regular language

- A language is regular if there is regular grammar that generates/recognizes it
- A language is regular if there is an FSA that generates/recognizes it
- A language is regular regular if we can define a regular expressions for the language



# Regular expressions

- Every regular language (RL) can be expressed by a regular expression (RE), and every RE defines a RL
  - A RE  $e$  defines a RL  $\mathcal{L}(e)$
  - Relations between RE and RL
    - $\mathcal{L}(\emptyset) = \emptyset$ ,
    - $\mathcal{L}(\epsilon) = \epsilon$ ,
    - $\mathcal{L}(a) = a$
    - $\mathcal{L}(ab) = \mathcal{L}(a)\mathcal{L}(b)$
    - $\mathcal{L}(a^*) = \mathcal{L}(a)^*$
    - $\mathcal{L}(a|b) = \mathcal{L}(a) \cup \mathcal{L}(b)$   
 (some author use the notation  $a+b$ , we will use  $a|b$  as in many practical implementations)
- where,  $a, b \in \Sigma$ ,  $\epsilon$  is empty string,  $\emptyset$  is the language that accepts nothing (e.g.,  $\Sigma^* - \Sigma^*$ )
- Note: no standard complement and intersection in RE

# Regular expressions

## and some extensions

- Kleene star ( $a^*$ ), concatenation ( $ab$ ) and union ( $a|b$ ) are the basic operations
- Parentheses can be used to group the sub-expressions. Otherwise, the priority of the operators are as listed above:  $a|bc^* = a|(b(c^*))$
- In practice some short-hand notations are common

$$- \cdot = (a_1 | \dots | a_n),$$

for  $\Sigma = \{a_1, \dots, a_n\}$

$$- a^+ = aa^*$$

$$- [a-c] = (a|b|c)$$

$$- [\sim a-c] = \cdot - (a|b|c)$$

$$- \backslash d = (0|1|\dots|8|9)$$

$$- \dots$$

- And some non-regular extensions, like  $(a^*)b\backslash 1$  (sometimes the term *regex* is used for expressions with non-regular extensions)

# Some properties of regular expressions

## Useful identities for simplifying regular expressions

- $u | (v | w) = (u | v) | w$
- $u | v = v | u$
- $u(v | w) = uv | uw$
- $u | \emptyset = u$
- $u\epsilon = \epsilon u = u$
- $\emptyset u = \emptyset$
- $u(vw) = (uv)w$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $(u^*)^* = u^*$
- $u | u = u$
- $(u | v)^* = (u^* | v^*)^*$
- $u^* | \epsilon = u^*$

# Some properties of regular expressions

## Useful identities for simplifying regular expressions

- $u|(v|w) = (u|v)|w$
- $u|v = v|u$
- $u(v|w) = uv|uw$
- $u|\emptyset = u$
- $u\epsilon = \epsilon u = u$
- $\emptyset u = \emptyset$
- $u(vw) = (uv)w$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $(u^*)^* = u^*$
- $u|u = u$
- $(u|v)^* = (u^*|v^*)^*$
- $u^*|\epsilon = u^*$

### An exercise

Simplify  $a|ab^*$

Note: some of these are direct statements of Kleene algebra, others can be derived from them.

# Some properties of regular expressions

## Useful identities for simplifying regular expressions

- $u|(v|w) = (u|v)|w$
- $u|v = v|u$
- $u(v|w) = uv|uw$
- $u|\emptyset = u$
- $u\epsilon = \epsilon u = u$
- $\emptyset u = \emptyset$
- $u(vw) = (uv)w$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $(u^*)^* = u^*$
- $u|u = u$
- $(u|v)^* = (u^*|v^*)^*$
- $u^*|\epsilon = u^*$

### An exercise

Simplify  $a|ab^*$

$$a|ab^* = a\epsilon|ab^*$$

Note: some of these are direct statements of Kleene algebra, others can be derived from them.

# Some properties of regular expressions

## Useful identities for simplifying regular expressions

- $u|(v|w) = (u|v)|w$
- $u|v = v|u$
- $u(v|w) = uv|uw$
- $u|\emptyset = u$
- $u\epsilon = \epsilon u = u$
- $\emptyset u = \emptyset$
- $u(vw) = (uv)w$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $(u^*)^* = u^*$
- $u|u = u$
- $(u|v)^* = (u^*|v^*)^*$
- $u^*|\epsilon = u^*$

### An exercise

Simplify  $a|ab^*$

$$\begin{aligned} a|ab^* &= a\epsilon|ab^* \\ &= a(\epsilon|b^*) \end{aligned}$$

Note: some of these are direct statements of Kleene algebra, others can be derived from them.

# Some properties of regular expressions

## Useful identities for simplifying regular expressions

- $u|(v|w) = (u|v)|w$
- $u|v = v|u$
- $u(v|w) = uv|uw$
- $u|\emptyset = u$
- $u\epsilon = \epsilon u = u$
- $\emptyset u = \emptyset$
- $u(vw) = (uv)w$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $(u^*)^* = u^*$
- $u|u = u$
- $(u|v)^* = (u^*|v^*)^*$
- $u^*|\epsilon = u^*$

### An exercise

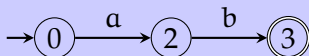
Simplify  $a|ab^*$

$$\begin{aligned} a|ab^* &= a\epsilon|ab^* \\ &= a(\epsilon|b^*) \\ &= ab^* \end{aligned}$$

Note: some of these are direct statements of Kleene algebra, others can be derived from them.

# Converting regular expressions to FSA

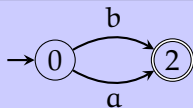
ab



$a^*$



$a|b$



- For more complex expressions, one can replace the paths for individual symbols with corresponding automata
- Using  $\epsilon$  transitions may ease the task
- The reverse conversion (from automata to regular expressions) is also easy:
  - identify the patterns on the left, collapse paths to single transitions with regular expressions

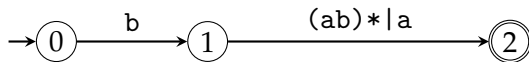


# Exercise

convert  $b((ab)^*|a)$  to an NFA

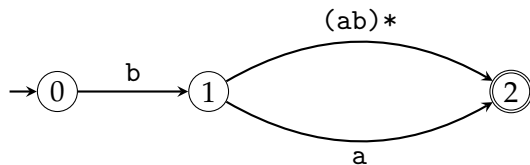
# Exercise

convert  $b((ab)^*|a)$  to an NFA



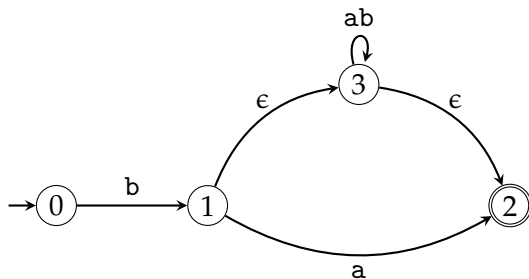
# Exercise

convert  $b((ab)^*|a)$  to an NFA



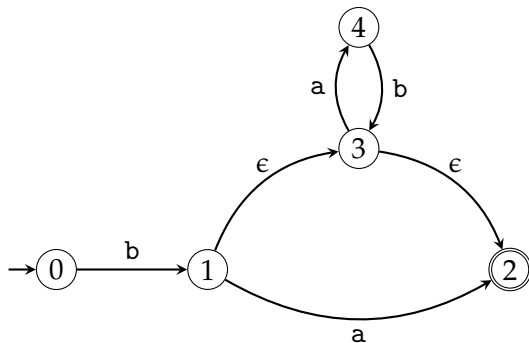
# Exercise

convert  $b((ab)^*|a)$  to an NFA

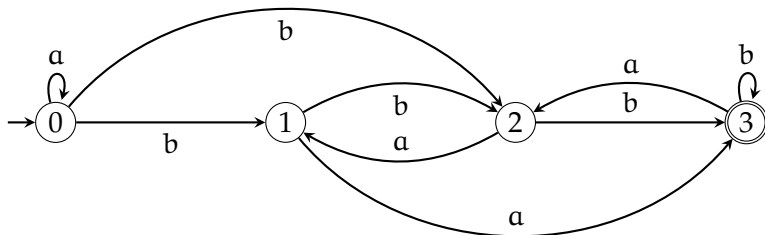


# Exercise

convert  $b((ab)^*|a)$  to an NFA

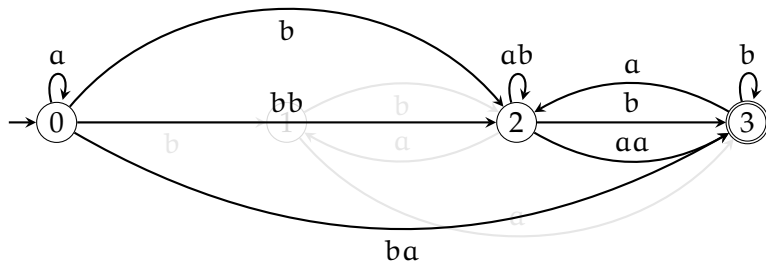


# Converting FSA to regular expressions



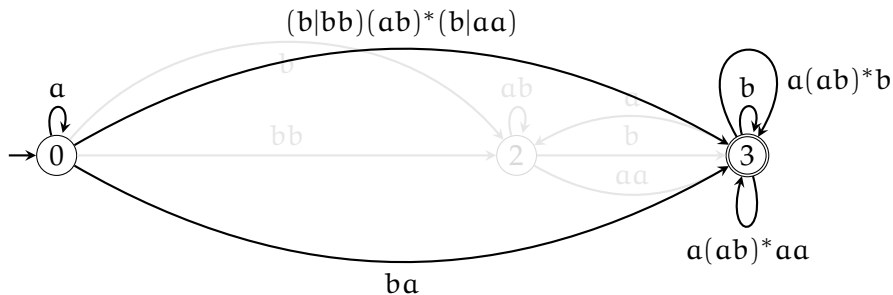
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

# Converting FSA to regular expressions



- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

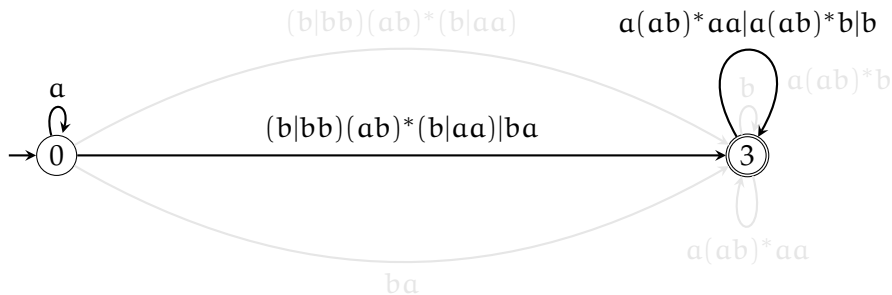
# Converting FSA to regular expressions



- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

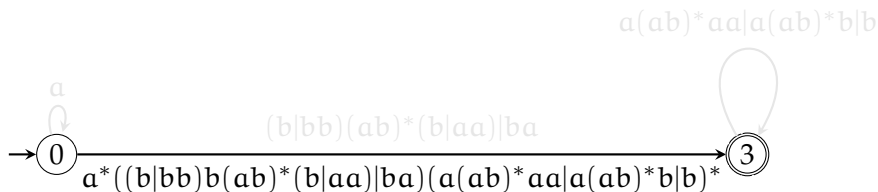


# Converting FSA to regular expressions



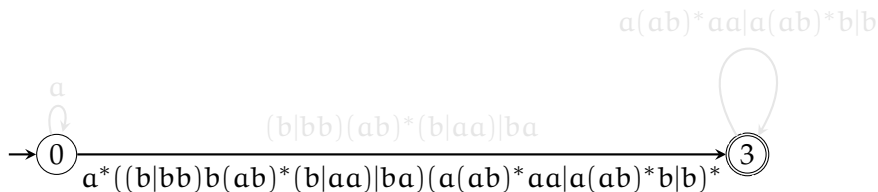
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

# Converting FSA to regular expressions



- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

# Converting FSA to regular expressions



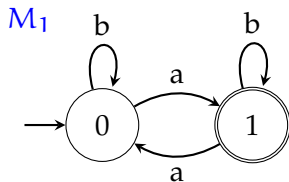
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

An exercise: simplify the resulting regular expressions

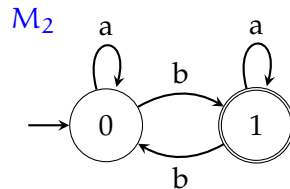
# Two example FSA

what languages do they accept?

$$L_1 = \mathcal{L}(M_1)$$



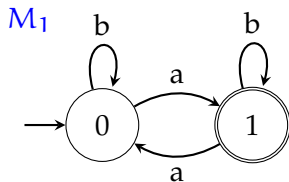
$$L_2 = \mathcal{L}(M_2)$$



## Two example FSA

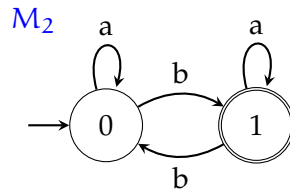
what languages do they accept?

$$L_1 = \mathcal{L}(M_1)$$



Odd number of a's over  $\{a, b\}$ .

$$L_2 = \mathcal{L}(M_2)$$

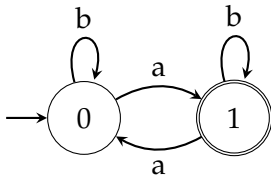


Odd number of b's over  $\{a, b\}$ .

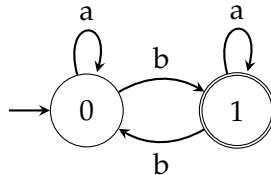
We will use these languages and automata for demonstration.

# Concatenation

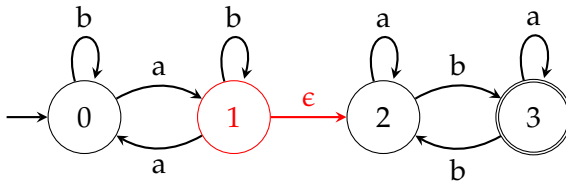
$L_1$



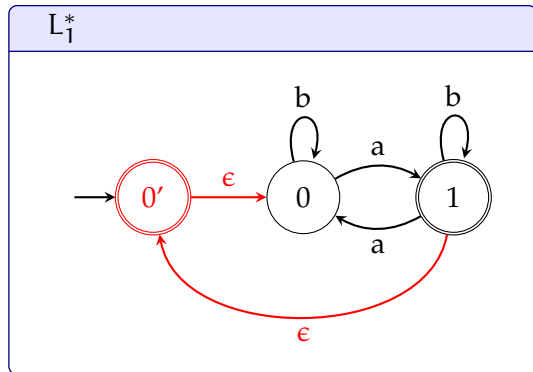
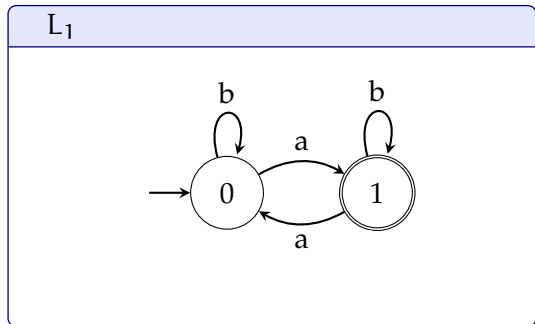
$L_2$



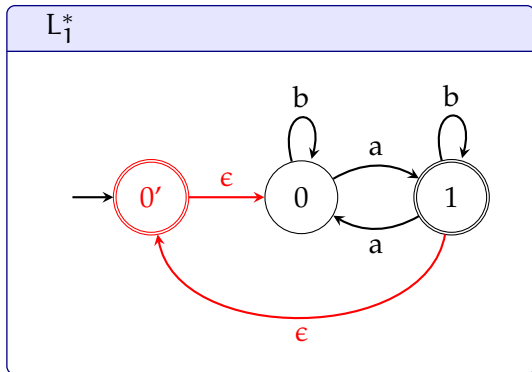
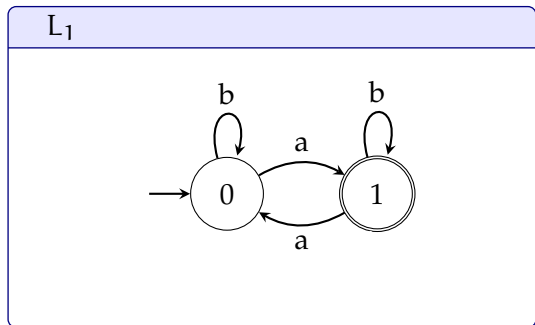
$L_1 L_2$



# Kleene star



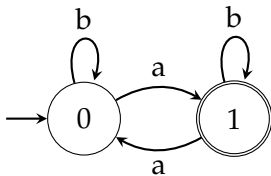
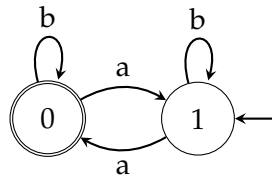
# Kleene star



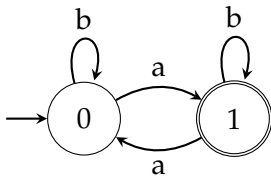
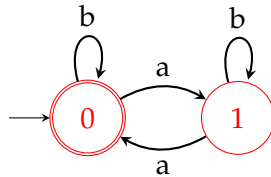
- What if there were more than one accepting states?



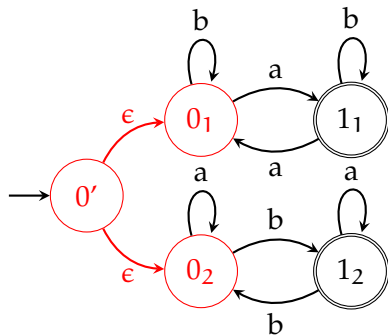
# Reversal

 $L_1$ 

 $L_1^R$ 


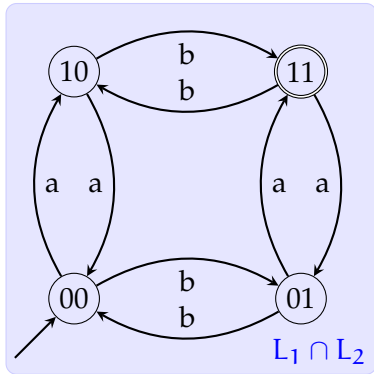
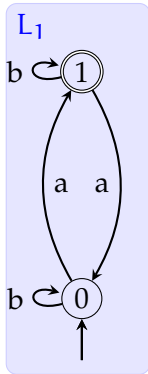
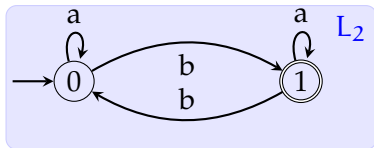
# Complement

 $L_1$ 

 $\overline{L_1}$ 


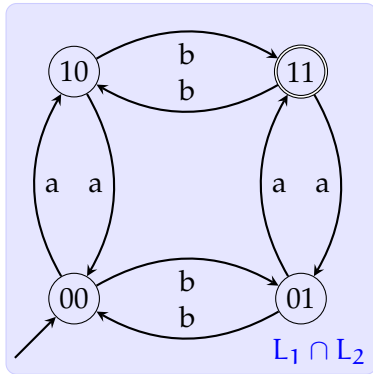
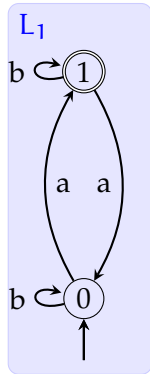
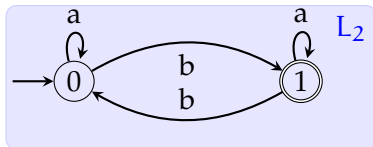
# Union

 $L_1 \cup L_2$ 


# Intersection



# Intersection



...or

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

# Closure properties of regular languages

- Since results of all the operations we studied are FSA: Regular languages are closed under
  - Concatenation
  - Kleene star
  - Reversal
  - Complement
  - Union
  - Intersection

## Wrapping up

- FSA and regular expressions express regular languages
- Regular languages and FSA are closed under
  - Concatenation
  - Kleene star
  - Complement
  - Reversal
  - Union
  - Intersection
- To prove a language is regular, it is sufficient to find a regular expression or FSA for it
- To prove a language is not regular, we can use pumping lemma (see Appendix)

# Wrapping up

- FSA and regular expressions express regular languages
- Regular languages and FSA are closed under
  - Concatenation
  - Kleene star
  - Complement
  - Reversal
  - Union
  - Intersection
- To prove a language is regular, it is sufficient to find a regular expression or FSA for it
- To prove a language is not regular, we can use pumping lemma (see Appendix)

Next:

- FSTs



## Acknowledgments, credits, references

- The classic reference for FSA, regular languages and regular grammars is Hopcroft and Ullman (1979) (there are recent editions).



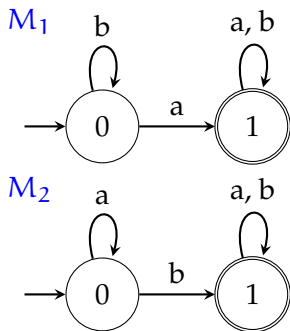
Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman (2007). *Introduction to Automata Theory, Languages, and Computation*. 3rd. Pearson/Addison Wesley. ISBN: 9780321462251.



Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley. ISBN: 9780201029888.

## Another exercise on intersection

Construct the intersection of the automata below (adapted from Hopcroft, Motwani, and Ullman (2007), Fig. 4.4)



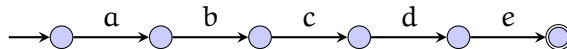
# Is a language regular?

— or not

- To show that a language is regular, it is sufficient to find an FSA that recognizes it.
- Showing that a language is *not* regular is more involved
- We will study a method based on *pumping lemma*

# Pumping lemma

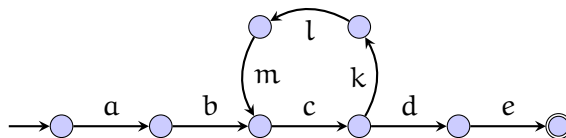
intuition



- What is the length of longest string generated by this FSA?

# Pumping lemma

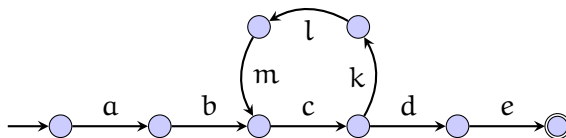
intuition



- What is the length of longest string generated by this FSA?

# Pumping lemma

## intuition



- What is the length of longest string generated by this FSA?
- Any FSA generating an infinite language has to have a loop (application of recursive rule(s) in the grammar)
- Part of every string longer than some number will include repetition of the same substring ('cklm' above)

# Pumping lemma

## definition

For every regular language  $L$ , there exist an integer  $p$  such that a string  $x \in L$  can be factored as  $x = uvw$ ,

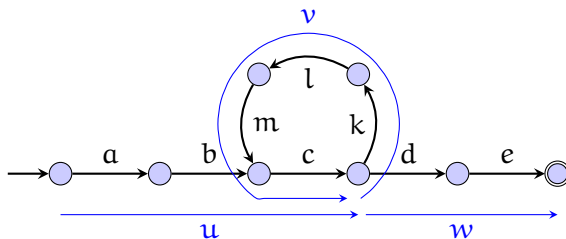
- $uv^i w \in L, \forall i \geq 0$
- $v \neq \epsilon$
- $|uv| \leq p$

# Pumping lemma

## definition

For every regular language  $L$ , there exist an integer  $p$  such that a string  $x \in L$  can be factored as  $x = uvw$ ,

- $uv^i w \in L, \forall i \geq 0$
- $v \neq \epsilon$
- $|uv| \leq p$





# How to use pumping lemma

- We use pumping lemma to prove that a language is not regular
- Proof is by contradiction:
  - Assume the language is regular
  - Find a string  $x$  in the language, for all splits of  $x = uvw$ , at least one of the pumping lemma conditions does not hold
    - $uv^i w \in L$  ( $\forall i \geq 0$ )
    - $v \neq \epsilon$
    - $|uv| \leq p$

# Pumping lemma example

prove  $L = a^n b^n$  is not regular

- Assume  $L$  is regular: there must be a  $p$  such that, if  $uvw$  is in the language
  - $uv^i w \in L$  ( $\forall i \geq 0$ )
  - $v \neq \epsilon$
  - $|uv| \leq p$
- Pick the string  $a^p b^p$
- For the sake of example, assume  $p = 5$ ,  $x = aaaaaabbbbbb$
- Three different ways to split

---

$\underbrace{a}_u \underbrace{aaa}_v \underbrace{abbbb}_w$	violates 1
$\underbrace{aaaa}_u \underbrace{ab}_v \underbrace{bbbb}_w$	violates 1 & 3
$\underbrace{aaaaab}_u \underbrace{bbb}_v \underbrace{b}_w$	violates 1 & 3

---







