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 2025/26

version: 8fd384e @2026-01-12

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 if we can define a regular expressions for the language

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.

Regular expressions

- Every regular language can be expressed by a regular expression, and every regular expressions defines a regular language
- A regular expression `e` defines a regular language $\mathcal{L}(\texttt{e})$
- Relations between regular expressions and regular languages
  - $\mathcal{L}(\varnothing) = \varnothing$,
  - $\mathcal{L}(\epsilon) = \epsilon$,
  - $\mathcal{L}(\texttt{ab}) = \mathcal{L}(a)\mathcal{L}(b)$
  - $\mathcal{L}(\texttt{a*}) = \mathcal{L}(a)^*$
  - $\mathcal{L}(\texttt{a|b}) = \mathcal{L}(a) \cup \mathcal{L}(b)$ (some author use the notation `a+b`, we will use `a|b` as in many practical implementations)

  where, ϵ is the empty string, $\varnothing$ is the language that accepts nothing (e.g., $\Sigma^* - \Sigma^*$)
- Note: no complement and intersection operators in common regex libraries

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
  - `.` = $(a_1|\ldots|a_n)$, for $\Sigma = \{a_1, \ldots, a_n\}$
  - `a+` = `aa*`
  - `[a-c]` = `(a|b|c)`
  - `[^a-c]` = `. - (a|b|c)`
  - `\d` = `(0|1|...|8|9)`
  - …
- And some non-regular extensions, like `(a*)b\1` (sometimes the term *regexp* 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|∅ = u`
- `uϵ = ϵu = u`
- `∅u = ∅`
- `u(vw) = (uv)w`
- `∅* = ϵ`
- `ϵ* = ϵ`
- `(u*)* = u*`
- `u|u = u`
- `(u|v)* = (u*|v*)*`
- `u*|ϵ = u*`

Some properties of regular expressions

Useful identities for simplifying regular expressions

- $\texttt{u|(v|w)} = \texttt{(u|v)|w}$
- $\texttt{u|v} = \texttt{v|u}$
- $\texttt{u(v|w)} = \texttt{uv|uw}$
- $\texttt{u|}\varnothing = \texttt{u}$
- $\texttt{u}\epsilon = \epsilon\texttt{u} = \texttt{u}$
- $\varnothing\texttt{u} = \varnothing$
- $\texttt{u(vw)} = \texttt{(uv)w}$
- $\varnothing\texttt{*} = \epsilon$
- $\epsilon\texttt{*} = \epsilon$
- $\texttt{(u*)*} = \texttt{u*}$
- $\texttt{u|u} = \texttt{u}$
- $\texttt{(u|v)*} = \texttt{(u*|v*)*}$
- $\texttt{u*|}\epsilon = \texttt{u*}$

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|∅ = u
- uϵ = ϵu = u
- ∅u = ∅
- u(vw) = (uv)w
- ∅* = ϵ
- ϵ* = ϵ
- (u*)* = u*
- u|u = u
- (u|v)* = (u*|v*)*
- u*|ϵ = u*

An exercise

Simplify a|ab*

a|ab* = aϵ|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|∅ = u
- uϵ = ϵu = u
- ∅u = ∅
- u(vw) = (uv)w
- ∅* = ϵ
- ϵ* = ϵ
- (u*)* = u*
- u|u = u
- (u|v)* = (u*|v*)*
- u*|ϵ = u*

An exercise

Simplify a|ab*

a|ab* = aϵ|ab*
= a(ϵ|b*)

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

- $\texttt{u|(v|w)} = \texttt{(u|v)|w}$
- $\texttt{u|v} = \texttt{v|u}$
- $\texttt{u(v|w)} = \texttt{uv|uw}$
- $\texttt{u|}\varnothing = \texttt{u}$
- $\texttt{u}\epsilon = \epsilon\texttt{u} = \texttt{u}$
- $\varnothing\texttt{u} = \varnothing$
- $\texttt{u(vw)} = \texttt{(uv)w}$
- $\varnothing* = \epsilon$
- $\epsilon* = \epsilon$
- $\texttt{(u*)*} = \texttt{u*}$
- $\texttt{u|u} = \texttt{u}$
- $\texttt{(u|v)*} = \texttt{(u*|v*)*}$
- $\texttt{u*|}\epsilon = \texttt{u*}$

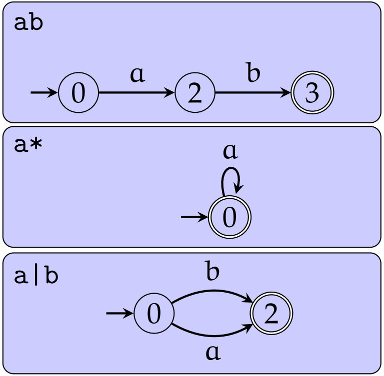
An exercise

Simplify a|ab*

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

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

Converting regular expressions to FSA



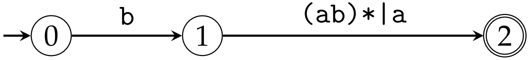
- For more complex expressions, one can replace the paths for individual symbols with corresponding automata
- Using ϵ transitions may ease the task
- The reverse conversion (from automata to regular expressions) is also easy:
  - identify the patterns on the left, collapse paths to single transitions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

Exercise

convert `b((ab)*|a)` to an NFA

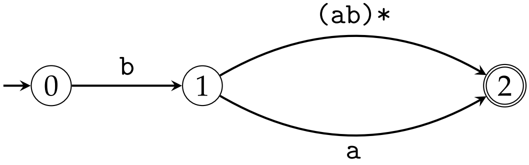
Exercise

convert `b((ab)*|a)` to an NFA



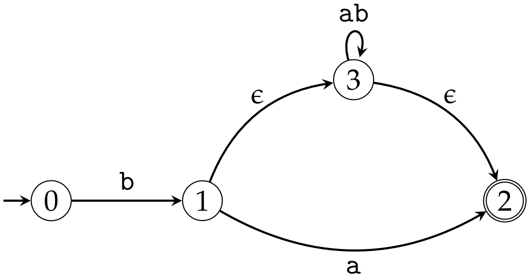
Exercise

convert `b((ab)*|a)` to an NFA



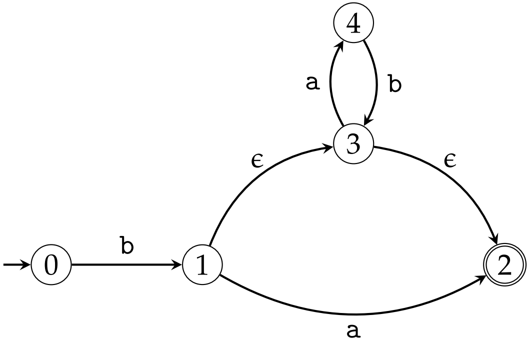
Exercise

convert `b((ab)*|a)` to an NFA

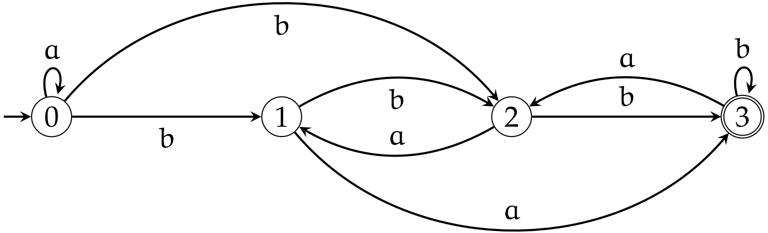


Exercise

convert `b((ab)*|a)` to an NFA

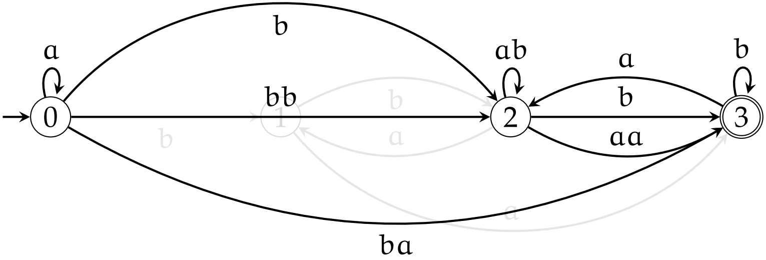


Converting FSA to regular expressions



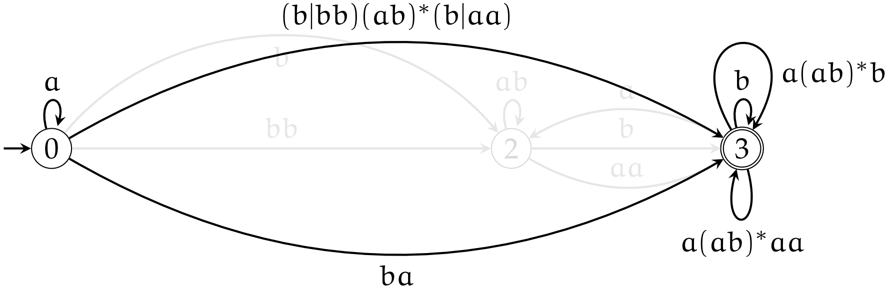
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

Converting FSA to regular expressions



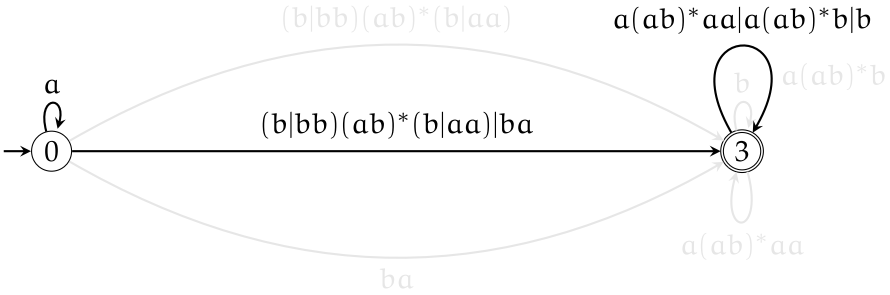
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

Converting FSA to regular expressions



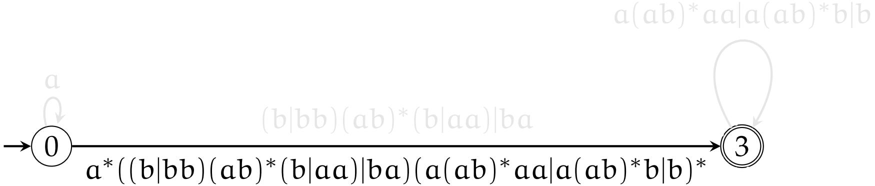
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

Converting FSA to regular expressions



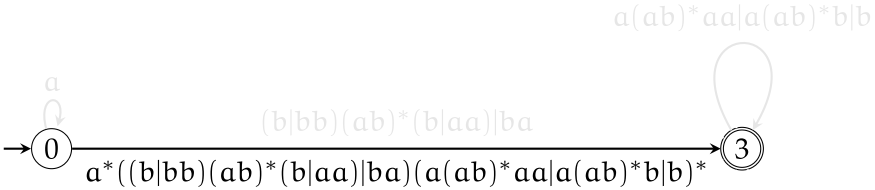
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

Converting FSA to regular expressions



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

Converting FSA to regular expressions



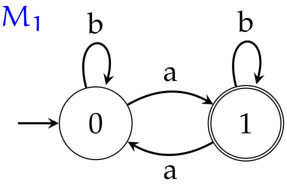
- The general idea: remove (intermediate) states, replacing edge labels with regular expressions

An exercise: simplify the resulting regular expressions

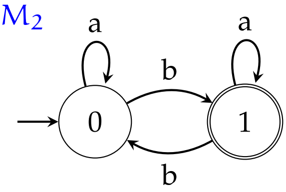
Two example FSA

what languages do they accept?

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



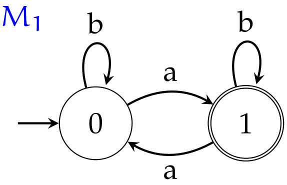
$L_2 = \mathcal{L}(M_2)$



Two example FSA

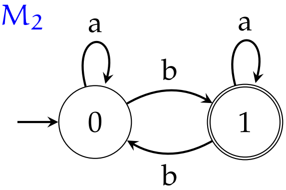
what languages do they accept?

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



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

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

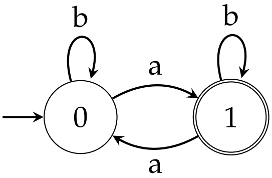


Odd number of b's over $\{a, b\}$.

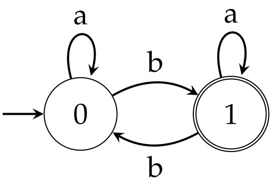
We will use these languages and automata for demonstration.

Concatenation

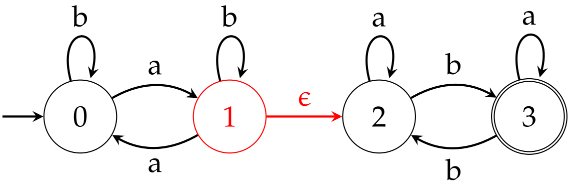
L_1



L_2

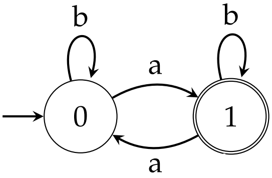


$L_1 L_2$

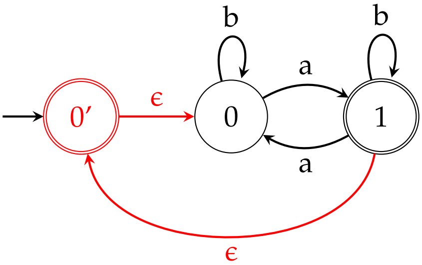


Kleene star

L_1

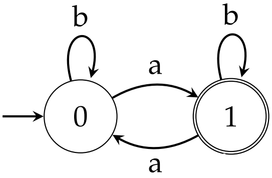


L_1^*

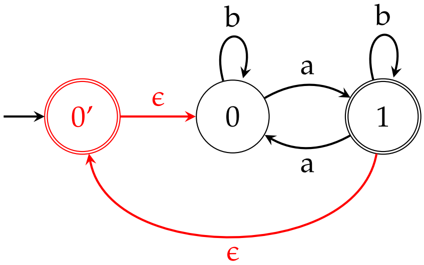


Kleene star

L_1



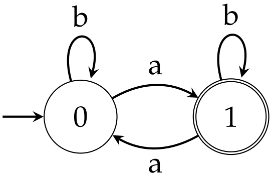
L_1^*



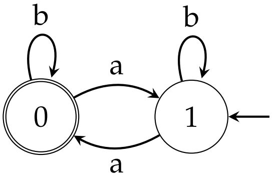
- What if there were more than one accepting states?

Reversal

L_1

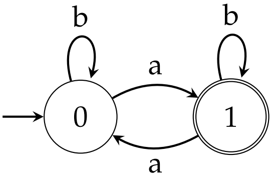


L_1^R

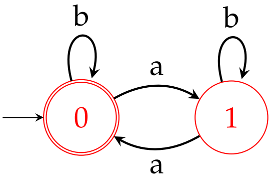


Complement

L_1

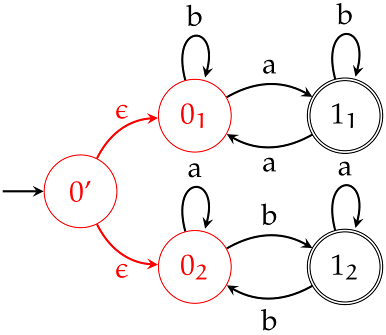


$\overline{L_1}$

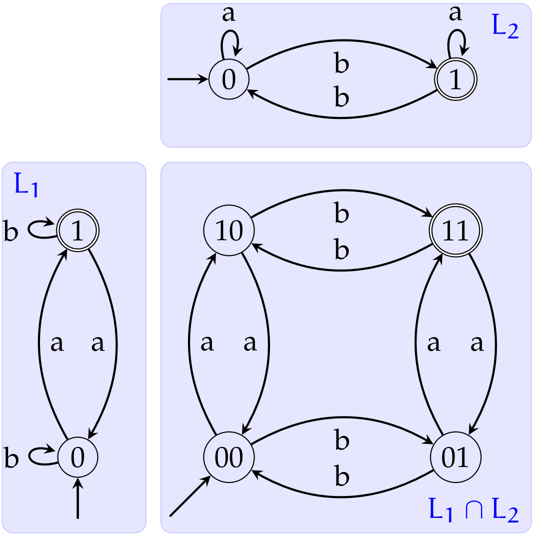


Union

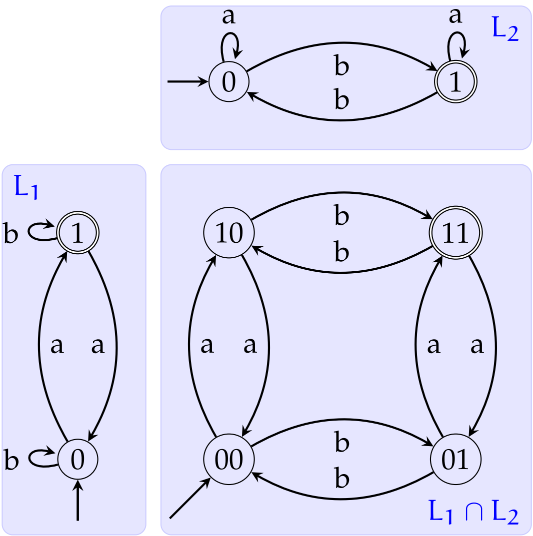
$L_1 \cup L_2$



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

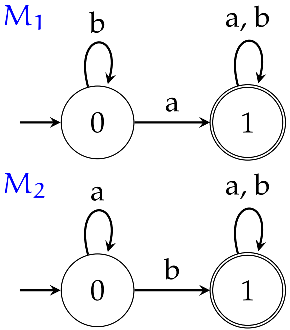
- The classic reference for FSA, regular languages and regular grammars is Hopcroft and Ullman (1979) (there are recent editions).

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)



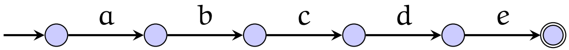
Is a language regular?

— 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*

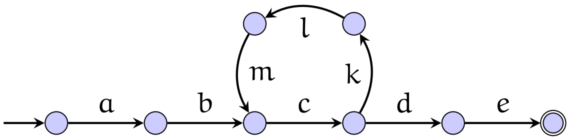
Pumping lemma

intuition



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

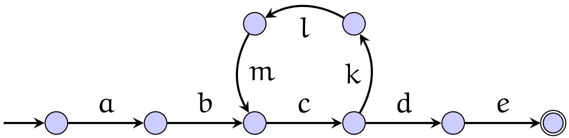
Pumping lemma
intuition



- 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

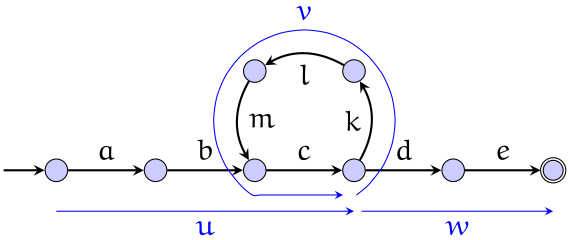
For every regular language L, there exist an integer p such that a string $x \in L$ can be factored as $x = uvw$,

- $uv^iw \in L, \forall i \geqslant 0$
- $v \neq \epsilon$
- $|uv| \leqslant p$

Pumping lemma
definition

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

- $uv^iw \in L, \forall i \geqslant 0$
- $v \neq \epsilon$
- $|uv| \leqslant p$



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
    - $uv^iw \in L \ (\forall i \geqslant 0)$
    - $v \neq \epsilon$
    - $|uv| \leqslant p$

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
  1. $uv^i w \in L$ $(\forall i \geqslant 0)$
  2. $v \neq \epsilon$
  3. $|uv| \leqslant p$
- Pick the string $a^p b^p$
- For the sake of example, assume $p = 5$, $x = aaaaabbbbb$
- Three different ways to split

$\underbrace{a}_{u}\underbrace{aaa}_{v}\underbrace{abbbbb}_{w}$	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

blank

blank

blank

blank