Basic Notions: Sets and Functions

Sets (basics)

- A **set** is a collection of **elements**.
  $L = \{a, b, c, d\}$ is a set of four elements. Note that the four elements are distinct.
- Let L be a set. $z \in L$ denotes that z is **in** L; and $z \notin L$ denotes that z is **not it** L.
- The **empty set** $\emptyset$ contains zero elements. All other sets are **nonempty**.
- A set is **finite** if it has a finite number of elements. Otherwise, the set is **infinite**.
- A is a **subset** of B (written as $A \subseteq B$), if every element of A is an element of B.
- Two sets A and B are **equal** (written as $A = B$) if and only if $A \subseteq B$ and $B \subseteq A$.
- If $A \subseteq B$ and $A \neq B$, then A is a **proper subset** of B, written as $A \subset B$. (By this definition, $\emptyset$ is a proper subset of every nonempty set.)

Sets Operations

Let A and B be two sets:

- **Intersection** $A \cap B$: It is the set $\{x \mid x \in A$ **and** $x \in B\}$ of all elements that are both in A and B. If $A \cap B = \emptyset$, then A and B are **disjoint**
- **Union** $A \cup B$: It is the set $\{x \mid x \in A$ **or** $x \in B\}$ of all elements that are in A or in B. Note that "or" is inclusive
- **Difference** $A - B$ or $A \setminus B$: It is the set $\{x \mid x \in A$ **and** $x \notin B\}$ of all elements that are in A but are not in B
- **Power set** 2^A (or $\mathcal{P}(A)$): The set of all subsets of a set A is the power set of A. e.g.: $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$.
- **Partition:** A partition of A is any set $\{A_1, A_2, \ldots\}$ of nonempty subsets of A such that
  1. $A = A_1 \cup A_2 \cup \cdots$ and
  2. $A_i \cap A_j = \emptyset$ for all $i \neq j$ (mutual disjointness)

Cartesian Product and Functions

- The **Cartesian product** $A \times B$ of two sets A and B is the set of all possible ordered pairs (a, b) with $a \in A$ and $b \in B$. e.g.: $\{1, 2\} \times \{3, 4\} = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$.

- A binary **relation**R among two sets A and B is (any) subset of their cartesian product, $R \subseteq A \times B$.

- A **function** from A to B (a binary function), written as $f : A \to B$, is a binary relation R on A and B such that, for each $a \in A$, if $(a, b) \in R$ and $(a, c) \in R$, then $b = c$ and, for each $a \in A$, there is either exactly one $b \in B$ such that $f(a) = b$ or there is no $b \in B$ such that $f(a) = b$. Such a fuction is said to be a **partial function**.

  A function f is **total** if, for every $a \in A$, there is a $b \in B$ such that $f(a) = b$. Thus, every total function is partial but the converse does not hold.

Functions

For example, letting $f = R = \{(1,3),(2,4)\}$. Then, $f(1) = 3$ and $f(2) = 4$.

Bijections:

- $f : A \to B$ is **one-to-one** if, for any distinct $a, a' \in A$, $f(a) \neq f(a')$.
- $f : A \to B$ is **onto** if, for all $b \in B$, there is some $a \in A$ such that $f(a) = b$.
- A total function f is **bijection** or **bijective** if it is both one-to-one and onto.
- Example: Let $A = \{1,2\}$ and $B = \{3,4\}$. $f_1 = \{(1,3),(2,4)\}$ is a bijection and $f_2 = \{(1,4),(2,4)\}$ is neither one-to-one nor onto.

Basic Notions: Alphabets, Strings, and Languages

Alphabets, Strings and Languages

An **alphabet** is a finite, nonempty set of symbols denoted by Σ. Common alphabets include:

1. $\Sigma = \{0, 1\}$, the *binary* alphabet
2. $\Sigma = \{a, b, c, \ldots, z\}$, the set of all lower-case letters
3. The set of all ASCII characters, or the set of all printable ASCII characters

Alphabets, Strings and Languages

- A **string** (or else called a **word**) is a finite sequence of symbols chosen from some alphabet. e.g., 01101, 111 over $\Sigma = \{0, 1\}$
- The **empty string** is the string with zero occurrences of symbols. This string, denoted by ε (sometines denoted by λ), is a string that may be chosen from any alphabet
- The **length** of a string is the number of symbol occurrences. e.g., 01101 has lengh 5. The standard notion for the length of a string w is $|w|$. e.g., $|011| = 3$ and $|\varepsilon| = 0$
- The **occurrence** $|w|_\sigma$ of σ is the number of σ occurrences. e.g., $|01101|_0 = 2$
- The **power** Σ^k of an alphabet is the set of strings of length k, each of whose symbols is chosen from Σ. $\Sigma^0 = \{\varepsilon\}$ for any Σ. If $\Sigma = \{0, 1\}$,

  $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$, $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- The set of all strings over Σ is denoted by Σ^* (**Kleene star** or Kleene/star closure).

  e.g., $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$.

  In other words, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

Alphabets, Strings and Languages

- The set of all **nonempty** strings from Σ is denoted by Σ^+. Thus,
  - $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$.
  - $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.
- Given two strings x and y, $x \cdot y$ denotes the **concatenation** of x and y; the string formed by making a copy of x followed by y. (We often omit the concatenation operation symbol $\cdot$.)

  e.g., if $x = 01101$ and $y = 110$, then $xy = 01101110$ and $yx = 11001101$
- ε is **identity** for catenation since for any string w, $\varepsilon \cdot w = w \cdot \varepsilon = w$

Alphabets, Strings and Languages

- A **language** is a set of strings all of which are chosen from some Σ^*. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a *language over* Σ.

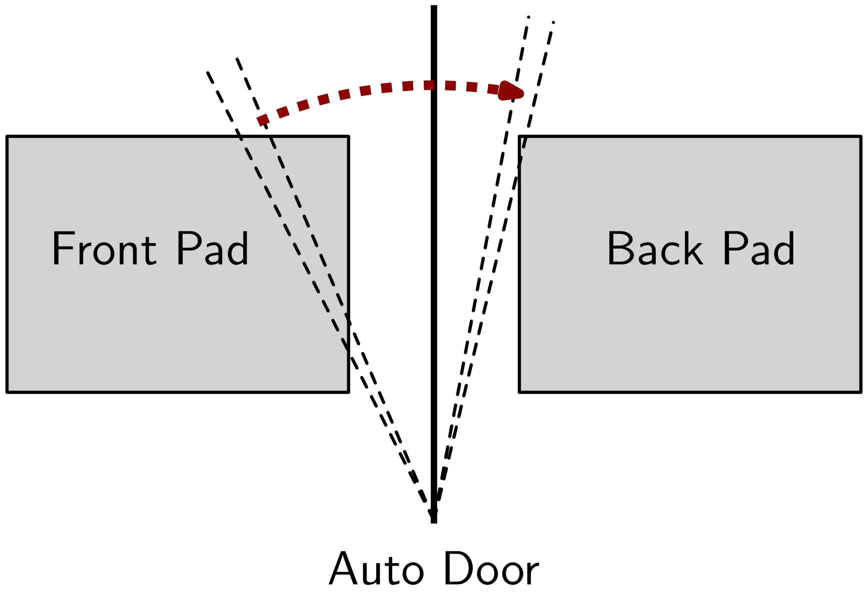
  e.g., C (the set of compilable C programs), Korean or English

  L may be *infinite* but there is some finite set of symbols of which all its strings are composed.

- Language examples
  - The set of all binary strings consisting of some number of 0's followed by an equal number of 1's; that is, $\{\varepsilon, 01, 0011, 000111, \dots\}$
  - The set of all binary strings with an equal number of 0's and 1's: $\{\varepsilon, 01, 10, 0011, 0101, 1001, \dots\}$
  - The set of binary numbers whose valuse is a prime: $\{10, 11, 101, 111, 1011, \dots\}$
  - Σ^* is a langauge for any alphabet Σ
  - $\emptyset$ is the empty language over any alphabet
  - $\{\varepsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Note that $\emptyset \neq \{\varepsilon\}$; the former has no strings but the latter has one string.

Deterministic Finite-State Machines

Finite-State Automata (FAs)



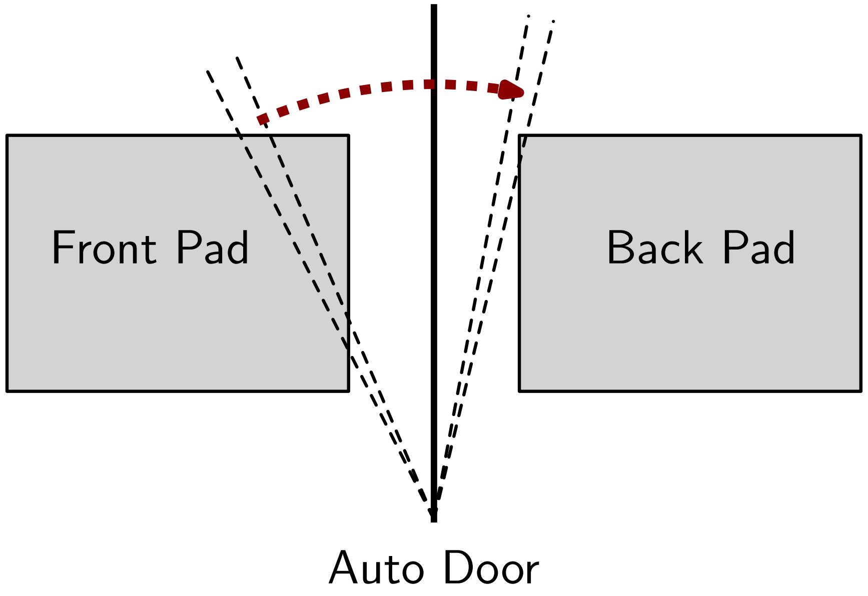
Two possible conditions for 'Auto Door'

- OPEN
- CLOSED

Four possible input conditions:

- FRONT: a person is standing on the pad in front of the doorway
- BACK: a person is standing on the pad to the rear of the doorway
- BOTH: people are standing on both pads
- NEITHER: no one is standing on either pad

Finite-State Automata (FAs)

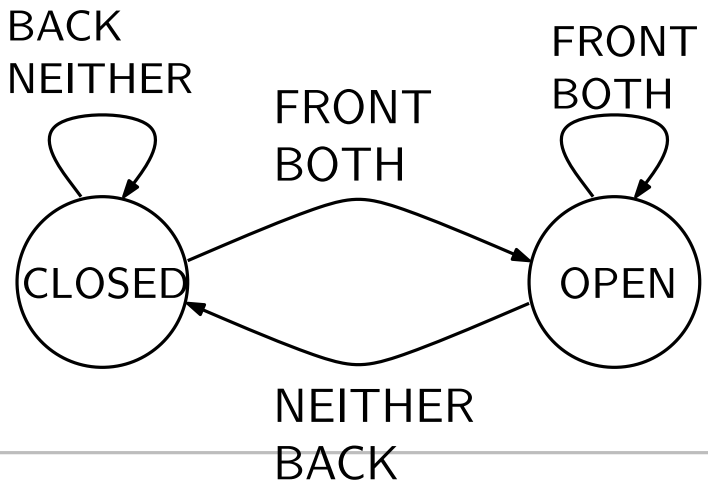


Two possible conditions for ‘Auto Door’

- OPEN
- CLOSED

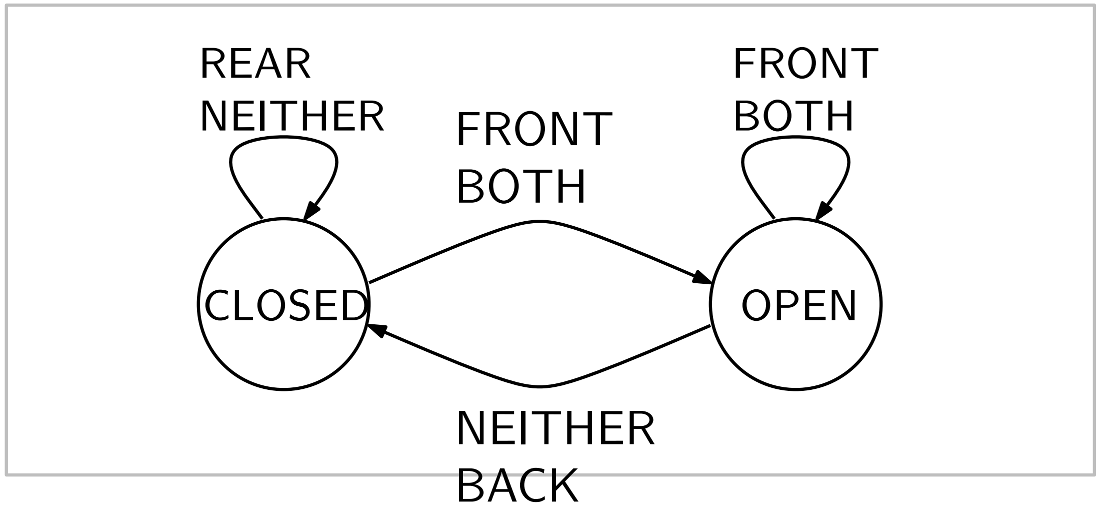
Four possible input conditions:

- FRONT
- BACK
- BOTH
- NEITHER



Deterministic Finite-State Automata (DFAs)

- Deterministic: After reading a symbol there is one and only one possible move
- Finite-State: A finite number of states
- Automata: Machines



DFA

A DFA A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where

- Q is a finite, nonempty set of states
- Σ is an input alphabet
- δ is a transition function $Q \times \Sigma \to Q$. We capture the meaning of the function schematically as follows:

Current State	Current Character	Next State
p	a	$\delta(p, a) = q$

- s is the start state
- F is a set of final states

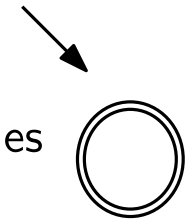
Transition Graphs

Transition graphs provide a schematic representation of a DFA.

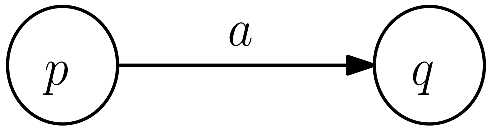
	a	b
0	0	1
1	1	0

$s = 0, F = \{0\}$

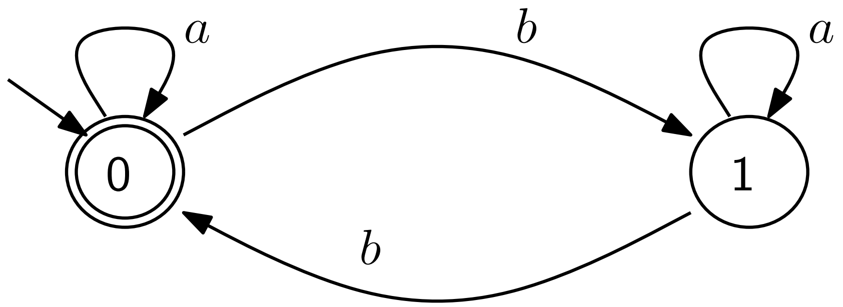
- Circles (Vertices) are states
  - The start state is indicated by a slant arrow
  - The final states are indicated by double circles



- Given a transition $\delta(p, a) = q$, there is an edge (transition) from p to q labeled with a:



Transition graph for A



Deterministic Finite-State Machines

How a DFA Works

Given a sequence w of input symobls:

$$w = a_1 a_2 a_3 \cdots a_n \in \Sigma^*.$$

1. Start from s in the DFA
2. Find the next state q_1 from $\delta(s, a_1)$
3. Process the next symbol a_2
4. Repeat the previous step and find $q_2, q_3, \ldots, q_n$
    - Accept, if $q_n \in F$
    - Reject, otherwise

A DFA A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where

- Q is a finite, nonempty set of states
- Σ is an input alphabet
- δ is a transition function $Q \times \Sigma \to Q$.

Current State	Current Character	Next State
p	a	$\delta(p, a) = q$

- s is the start state
- F is a set of final states

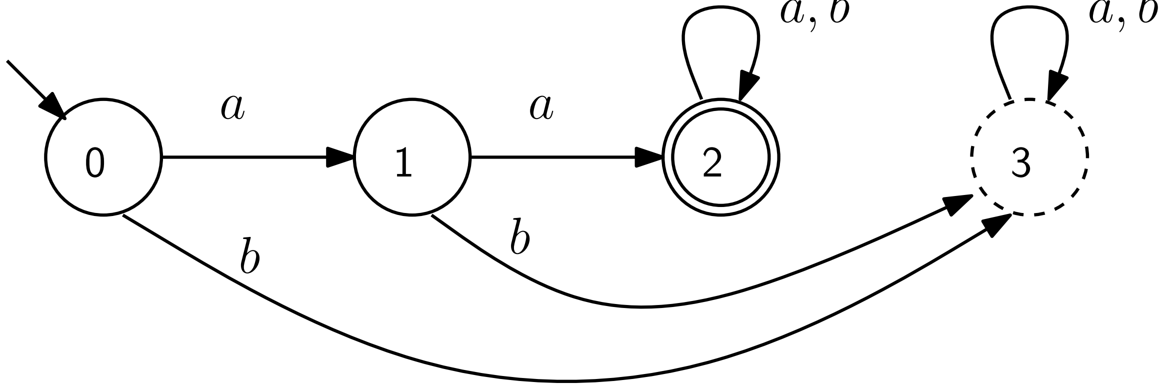
DFA Specification: Example

A DFA A is specified by $Q = \{0,1,2,3\}$, $\Sigma = \{a,b\}$, $s = 0$, $F = \{2\}$ and δ is given from the **transition table**:

	a	b
0	1	3
1	2	3
2	2	2
3	3	3

Example:

- $w_1 = aaa$
- $w_2 = abb$



Transition graph for A

DFA Configuration

- A **configuration** captures the current automaton status. If an automaton crashes and we know its current status, we can restart the automaton where it left off by using the latest configuration. For DFAs, we need only the current state and the current position of the reader in the input string.
- We use (current state, remainder of input strings), which is an element of $Q \times \Sigma^*$
- Example of DFA configurations:
    - $(q_3, abbcca)$
    - (q_7, cca)

Single-Step Computations

We now formalize the notion of a single-step computation.

1. One possible **start configuration** for A is $(0, aabba)$
2. Then, the configuration after one computational step is $(0, abba)$
3. What is the configuration after two computational steps?

	a	b
0	0	1
1	1	0

$s = 0, F = \{0\}$

We write these steps more formally as follows:

$$(0, aabba) \vdash_A (0, abba) \vdash_A (0, bba).$$

Single-step computations in a DFA A:

1. Let (p, w) be a current configuration, where $w = ax, a \in \Sigma$ and $x \in \Sigma^*$
2. If $\delta(p, a) = q$, then the next configuration is (q, x)
3. We say that (p, w) **yields** (q, x) in **one step**. Namely, $(p, w) \vdash_A (q, x)$
4. $\vdash_A$: $Q \times \Sigma^* \to Q \times \Sigma^*$

Single-Step Computations

We now formalize the notion of a single-step computation.

1. One possible **start configuration** for A is $(0, aabba)$
2. Then, the configuration after one computational step is $(0, abba)$
3. What is the configuration after two computational steps?

	a	b
0	0	1
1	1	0

$s = 0, F = \{0\}$

We write these steps more formally as follows:

Single step configuartion with respect to DFA A

$$(0, aabba) \vdash_A (0, abba) \vdash_A (0, bba).$$

Single-step computations in a DFA A:

1. Let (p, w) be a current configuration, where $w = ax, a \in \Sigma$ and $x \in \Sigma^*$
2. If $\delta(p, a) = q$, then the next configuration is (q, x)
3. We say that (p, w) **yields** (q, x) in **one step**. Namely, $(p, w) \vdash_A (q, x)$
4. $\vdash_A: Q \times \Sigma^* \to Q \times \Sigma^*$

Multiple-Step Computations

We extend the single-step computation into multi(ple)-step computation as follows:

1. If $(p, w) \vdash_A (p_1, w_1) \vdash_A \cdots \vdash_A (p_{n-1}, w_{n-1}) \vdash_A (q, x)$, we say that (p, w) **yields** (q, x) in n **steps**
$$(p, w) \vdash_A^n (q, x)$$
or, more simply,
$$(p, w) \vdash_A^* (q, x)$$
in 0 or more steps
2. Every configuration yields itself in zero steps; namely $(p, w) \vdash_A^* (p, w)$
3. $\vdash_A^*$ is the reflexive and transitive closure of $\vdash_A$
4. We can also use $\vdash_A^+ (p, w)$

Acceptance and Language by a DFA

We define the **acceptance** of a string w with respect to a DFA A as follows:

1. We say w is accepted by A if and only if

   $$(s, w) \vdash_A^* (f, \varepsilon) \text{ and } f \in F.$$

   In other words, there exists a sequence of configurations from **the start state** s to **a final state** f for the input string w with respect to A.

2. The **language** $L(A)$ of A is a set of all accepted strings;

   $$L(A) = \{w \mid w \text{ is accepted by } A\}.$$

A DFA Example

Observe that the character "b" flips the current state in A whereas "a" does not.

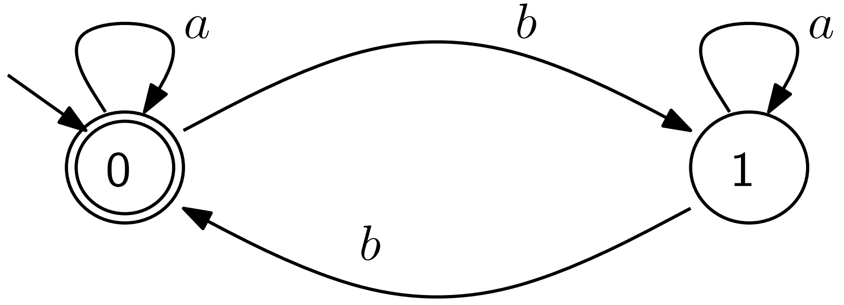
$$\begin{aligned}(0, aabba) &\vdash_A (0, abba)\\ &\vdash_A (0, bba)\\ &\vdash_A (1, ba)\\ &\vdash_A (0, a)\\ &\vdash_A (0, \varepsilon)\end{aligned}$$

	a	b
0	0	1
1	1	0

$s = 0, F = \{0\}$

So, $(0, aabba) \vdash_A^* (0, \varepsilon)$ and, therefore, A accepts $aabba$.

$L(A) = \{w \mid w$ contains an **even** number of b's$\}$.



Non-Deterministic Finite-State Machines

DFAs and NFAs

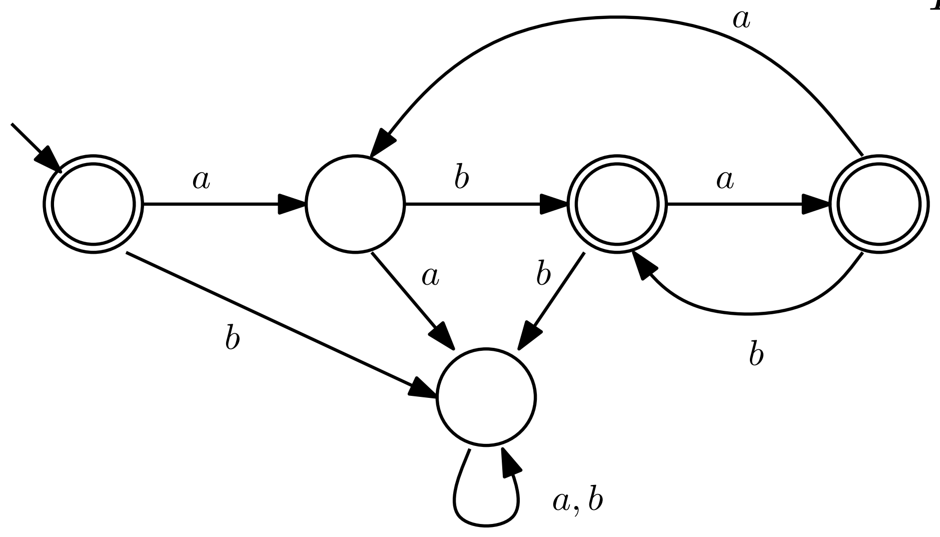
- In a DFA,
  1. The automaton has only one possible move for each character a state.
  2. The next state is completely determined by the current state and current character (there is **only one next state**).
  3. Acceptance occurs if **the computation** reads all the input string and the computation ends at a final state.

- In an NFA,
  1. The automaton may sometimes have more than one possible move.
  2. The next state is only *"partially determined"* by the current state and input character (there may be **two or more next states**).
  3. Acceptance occurs if **at least one computation** reads all the input string and the computation ends at a final state.

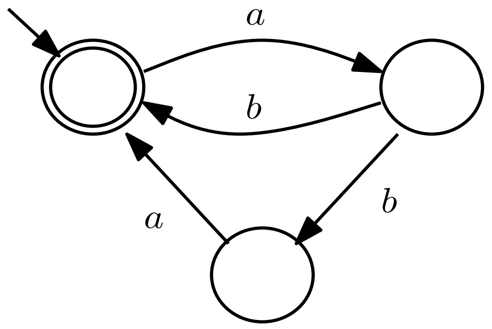
NFA Examples

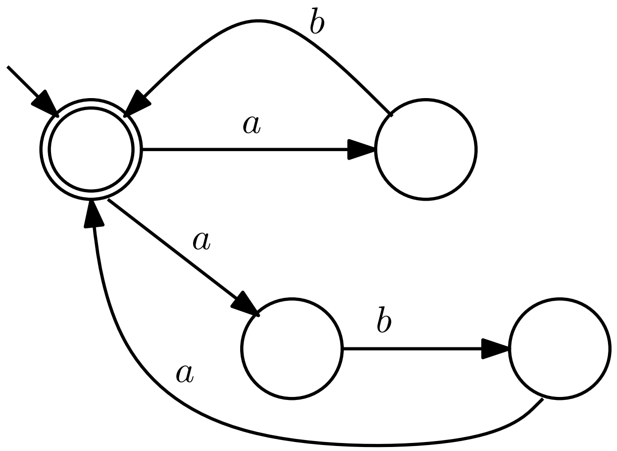
Consider the language L specified by the following DFA.

$$L = \{\varepsilon, (ab + aba)^i, ababa, abaab, \ldots\}$$



NFAs



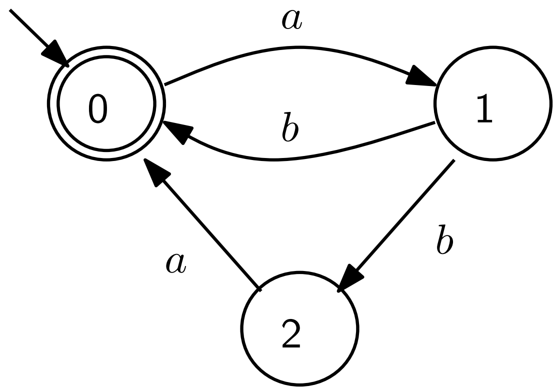


NFA Specification

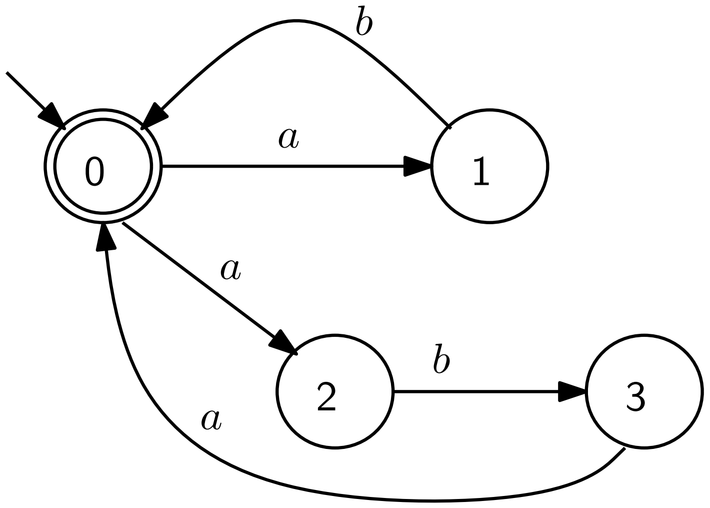
An NFA A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where

1. Q is a finite, nonempty set of states
2. Σ is an input alphabet
3. δ is a **transition relation** such that $\delta \subseteq Q \times \Sigma \times Q$.
   In other words, δ **is a transition function** $Q \times \Sigma \rightarrow 2^Q$ (c.f., for DFA, $Q \times \Sigma \rightarrow Q$)
4. s is the start state
5. F is a set of final states

NFA Examples Revisited



(I)



(II)

- NFA (I): $Q = \{0, 1, 2\}, s = 0, F = \{0\}, \Sigma = \{a, b\}$ and $\delta = \{(0, a, 1), (1, b, 0), (1, b, 2), (2, a, 0)\}$.
  Note that $(1, b, 0)$ means that when the automaton is in state 1, it **consumes** b and enters state 0
- NFA (II): $Q = \{0, 1, 2, 3\}, s = 0, F = \{0\}, \Sigma = \{a, b\}$ and $\delta = \{(0, a, 1), (0, a, 2), (1, b, 0), (2, b, 3), (3, a, 0)\}$

NFA Computations

Given an NFA $A = (Q, \Sigma, \delta, s, F)$:

1. $(q, w) \vdash_A (q', w')$ if (and only if) there is a character $\sigma \in \Sigma$ such that $w = \sigma w'$ and (q, σ, q') is in δ
2. $\vdash_A^*$ is the reflexive and transitive closure of $\vdash_A$
3. A string w is **accepted** by A if and only if there is **a computation**

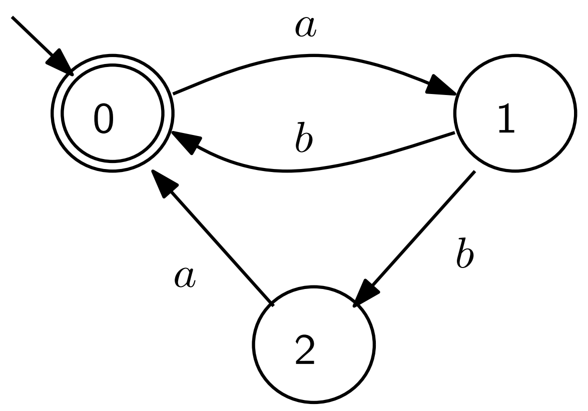
$$(s, w) \vdash_A^* (f, \varepsilon) \text{ and } f \in F.$$

4. The **language** $L(A)$ of A is defined as

$$L(A) = \{w \mid w \text{ is accepted by } A\}.$$

NFA Computations Example

Consider the NFA (I) specified as follows: $Q = \{0, 1, 2\}, s = 0, F = \{0\}, \Sigma = \{a, b\}$ and $\delta = \{(0, a, 1), (1, b, 0), (1, b, 2), (2, a, 0)\}$.



(I)

1. One computation of (I):
   $(0, aba) \vdash (1, ba) \vdash (0, a) \vdash (1, \varepsilon)$
   $(0, aba) \vdash^* (1, \varepsilon)$
2. Another computation of (I):
   $(0, aba) \vdash (1, ba) \vdash (2, a) \vdash (0, \varepsilon)$
   $(0, aba) \vdash^* (0, \varepsilon)$

NFA (I) accepts *aba* because of the second computation.
(Note that we have omitted the subscripts of $\vdash$ and $\vdash^*$ since we can assume that NFA (I) is nderstood.)

Equivalence of DFA with NFA

DFAs and NFAs

Between DFAs (**deterministic**) and NFAs (**nondeterministic**), which model have more expressive power? We know that, *by definition*, all DFAs are also NFAs. Thus, NFAs are **at least** as powerful as DFAs.

We prove that every NFA $A = (Q, \Sigma, \delta, s, F)$ can be converted into an equivalent DFA $A' = (Q', \Sigma, \delta', s', F')$. (By **equivalent**, we mean that A and A' accept the same language.)

- **Observation:** Given A, for the same input string w, all computations for w read the same character at the same computational step (except when some computations stop early)
- **Idea:** Combine all computations for w in A into one computation *that begins with the state set* $\{s\}$ and at each step computes the **next state set**, rather than the next state
- **Construction:** We compute all possible next state sets that are reachable from $\{s\}$, the **Subset Construction**.

Subset Construction

We compute all possible next state sets that are reachable from $\{s\}$ of an NFA A.

1. How many state sets are there, in the worst-case, for an NFA A with n states?
2. Which state sets are **final state sets**?
3. What is the new transition relation?

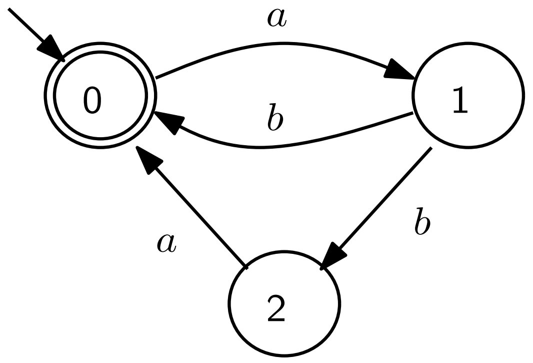
We construct δ' from δ as follows:

1. We construct transitions from $s' = \{s\}$ using the rule that $(s'\sigma, R)$ is in δ' of A' if and only if $R = \{r \mid (s, \sigma, r) \in \delta\}$
2. For each new state set P obtained in steps 1 and 2, we compute the transitions from P using the rule that (P, σ, R) is in δ' of A' if and only if $R = \{r \mid (p, \sigma, r) \in \delta \text{ and } p \in P\}$
3. When there are no new state sets generated in step 2, we have computed δ' and Q'

Now we define $F' = \{P \mid P \cap F \neq \emptyset\}$.

Subset Construction Example

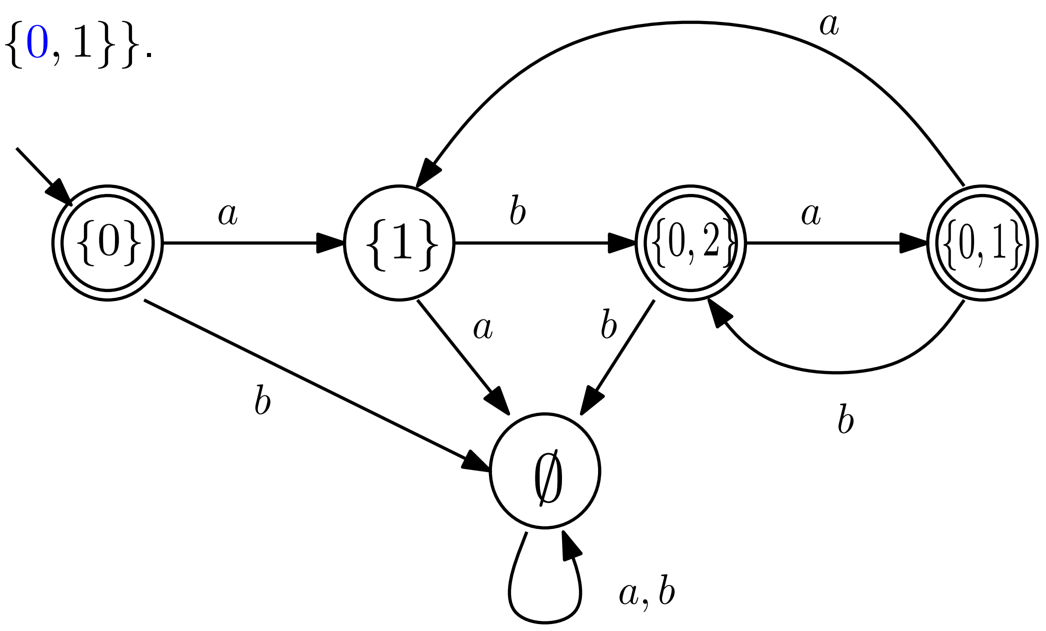
For the NFA (I), we have the following transition relation:



(I)

state set	a	b
$\{0\}$	$\{1\}$	$\emptyset$
$\{1\}$	$\emptyset$	$\{0,2\}$
$\{0,2\}$	$\{0,1\}$	$\emptyset$
$\{0,1\}$	$\{1\}$	$\{0,2\}$
$\emptyset$	$\emptyset$	$\emptyset$

Then, $F' = \{\{0\},\{0,2\},\{0,1\}\}$.



Subset Construction Correctness

Formally, we still have to prove the correctness of subset construction. However, here is only the idea of the proof. The proof of correctness has two parts:

1. Prove that A' is deterministic.
   This part is straightforward. Show that is (P, σ, R) and (P, σ, S) are both in δ', then $R = S$. Use the definition of the transitions for A'
2. Prove that $L(A') = L(A)$.
   This part is more complex. We need to prove, by induction on string length, that if $(s, w) \vdash^* (f, \varepsilon)$ in A such that $f \in F$, then there is a $P \in F'$ such that $(\{s\}, w) \vdash^* (P, \varepsilon)$ in A' and $f \in P$.

Therefore, NFAs have the **same** expressive power as DFAs.

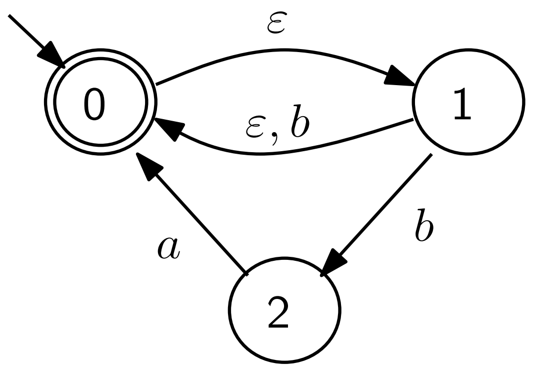
Epsilon NFA

ε-NFAs

A ε-NFA is an NFA that has a ε-transition.

A ε-transition

- allows to move between states **without reading** the input
- makes the FA **highly nondeterministic**

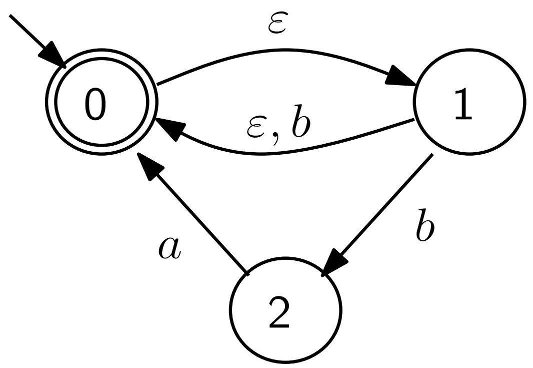


ε-NFAs

A ε-NFA is an NFA that has a ε-transition.

A ε-transition

- allows to move between states **without reading** the input
- makes the FA **highly nondeterministic**



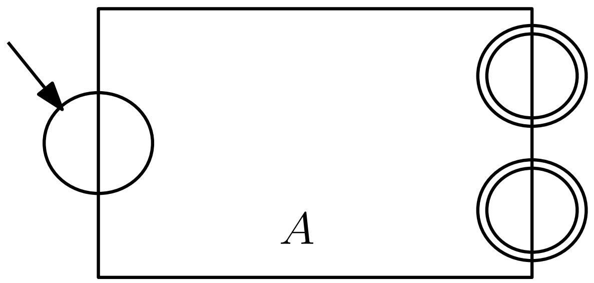
Given an input $baba$

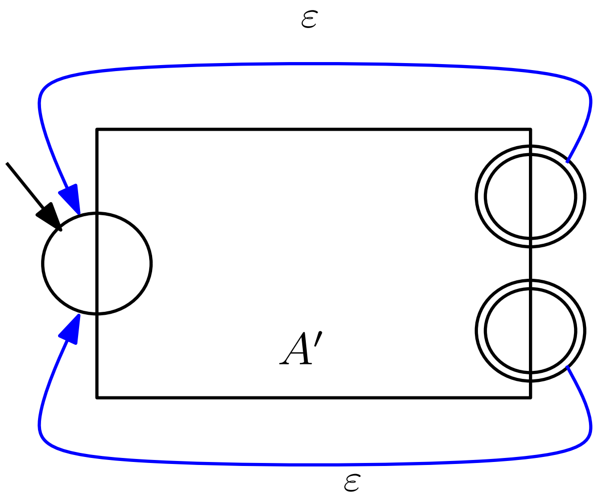
$$(1, baba) \vdash \begin{cases} (0, baba) & \text{by the } \lambda\text{-transition} \\ (0, aba) & \text{by the } b\text{-transition to 0} \\ (2, aba) & \text{by the } b\text{-transition to 2} \end{cases}$$

ε-NFAs

There are situations in which ε-transitions are useful.

- For example for a given languge L and an NFA A recognizing it, we can add ε-transitions from the final states back to the start state. The new automaton recognizes the language L^+.

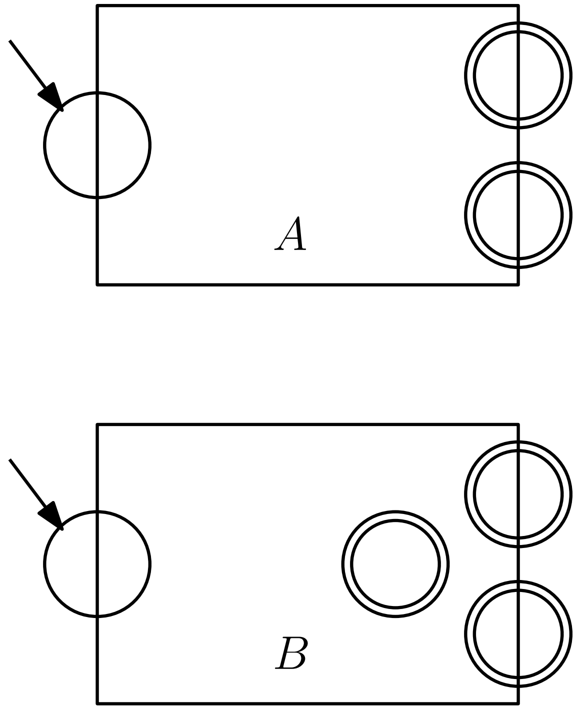


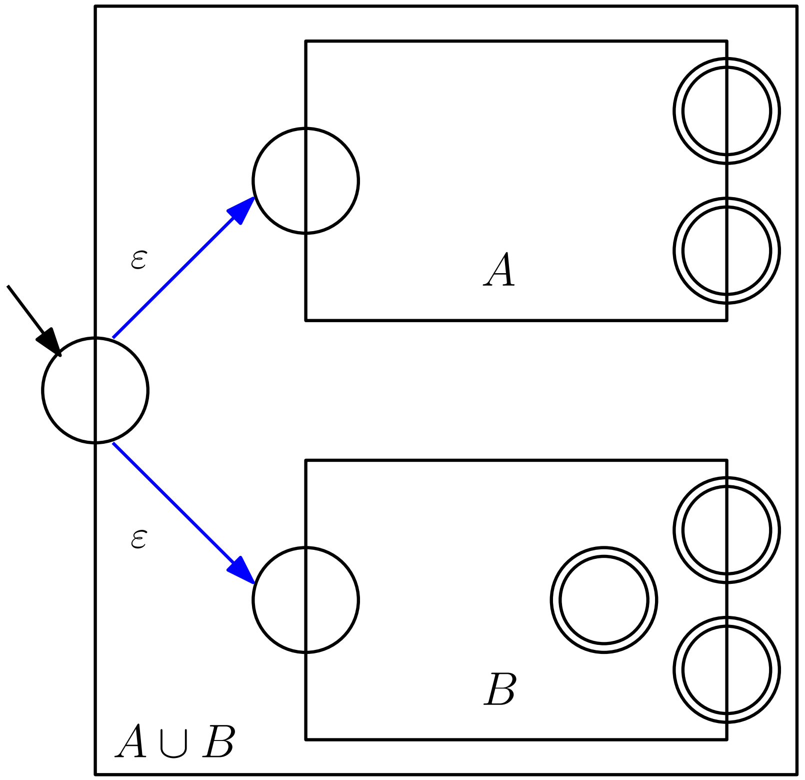


ε-NFAs

There are two situations in whcih ε-transitions are useful.

- The union of two NFAs: Given two NFAs A and B, we can make their union C that consists of A and B together with a new start state which via ε-transitions leads to the start states of A and B.



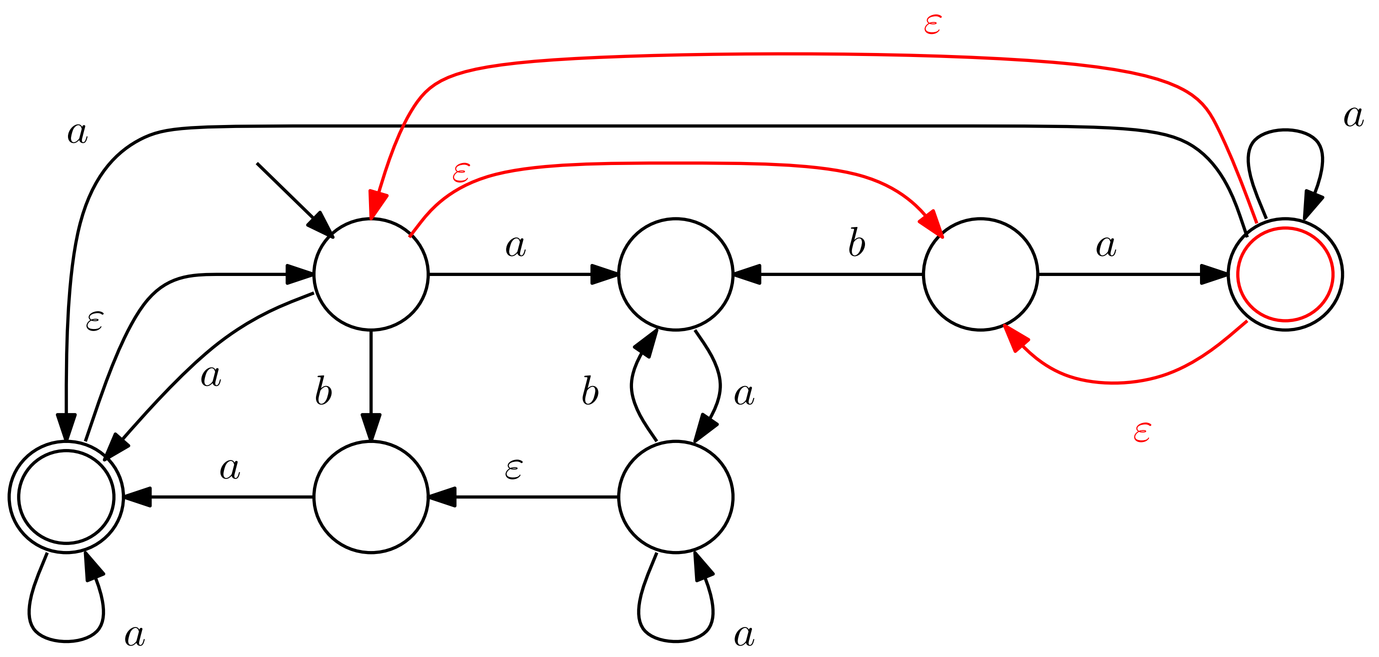


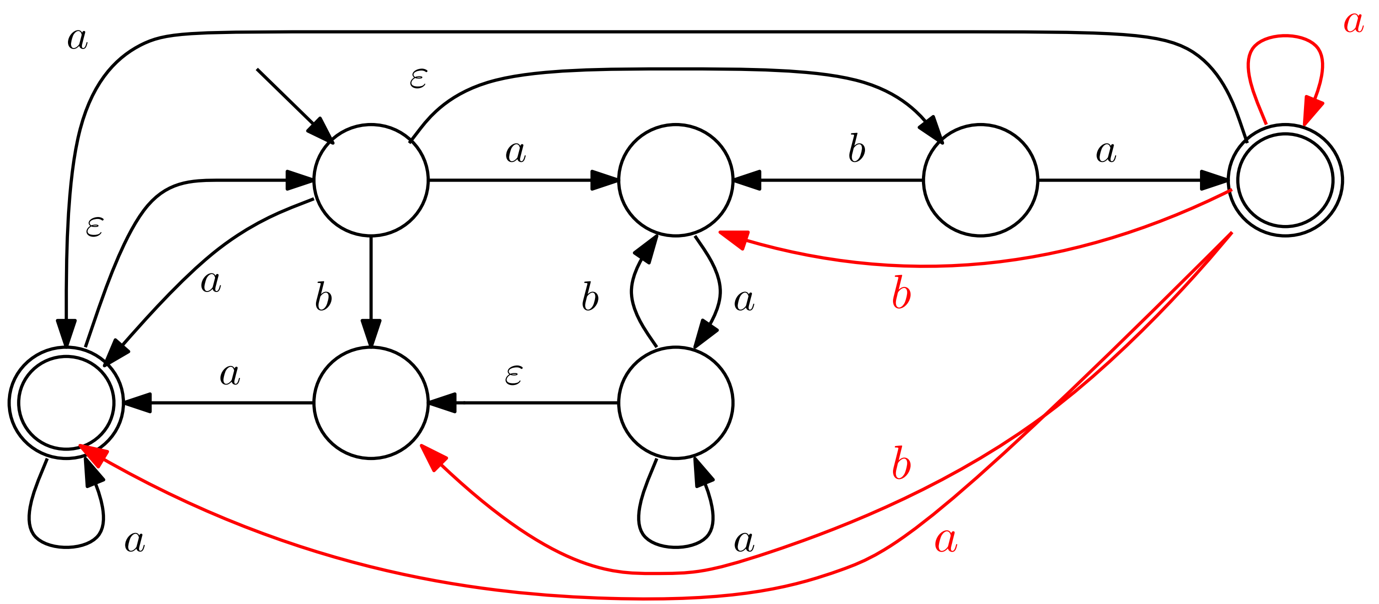
NFAs and ε-NFAs

Every NFA is an ε-NFA but *surprisingly* the expressive power of ε-NFAs is **no more** than of NFAs. In other words, we can transform an ε-NFA into an equivalent NFA.
Given an ε-NFA $A = (Q, \Sigma, \delta, s, F)$:

1. We modify A so that whenever there is a configuration $(p, w) \vdash^i (q, w)$ for $i \geq 1$, A has a transition (p, ε, q) and whenever a state can reach a final state via ε-transition, we make it to be a final state. Let A'' be the resulting ε-NFA
2. Whenever we have a configuration $(p, \sigma w) \vdash (q, \sigma w) \vdash (r, w)$ in A'', we add a transition (p, σ, r) to A''. Now we do not need to examine the configuration $(p, \sigma w) \vdash^i (q, \sigma w) \vdash (r, w)$ and make use of the new short cut. Finally, we remove all ε-transitions

The proof of correctness is left for an exercise.


a
ε
ε
ε
a
b
a
a
b
b
a
a
ε
ε
a
a

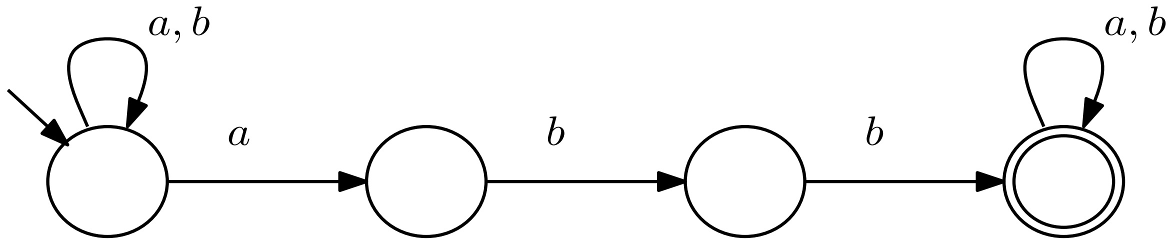

a
a
ε
ε
a
b
a
a
b
b
a
b
a
ε
b
a
a
a

Regular Expressions and Languages

Regular Expressions

Consider the language L of all strings that consist of a's and b's and have abb as a substring. We can formally define L as follows:

1. $L = \{w \mid w \in \{a,b\}^*$ and w has abb as a substring$\}$
2. $L = L(A)$, where A is an NFA given as follows:



Both definitions are lengthy. It can also be expressed by

$$L((a+b)^*abb(a+b)^*)$$

Regular Expressions

A finite method of specifying/expressing languages

The inductive definition of **regular expressions** over an alphabet Σ:

1. The $\emptyset$ character, the ε character and each character $\sigma \in \Sigma$ are regular expressions. (Note that $\emptyset$ and ε must not be in Σ.)
2. If α and β are regular expressions, then

   $$(\alpha \cdot \beta), (\alpha + \beta) \text{ and } (\alpha^*)$$

   are regular expressions (we usually omit "$\cdot$")

For example, given $\Sigma = \{a, b, c, d\}$,

- $a, ((a+b)^* \cdot d), ((c^*) \cdot (a + (b \cdot \varepsilon)))$ and $\emptyset^*$ are regular expressions
- $c+^*$ and $*$ are not regular expressions

The regular expressions $\emptyset$, ε, and σ represent the languages $\emptyset$, $\{\varepsilon\}$, and $\{\sigma\}$, respectively.
In general the regular expression α represents the language $L(\alpha)$.

Regular Expressions and Languages

Given two regular expressions E and F,

1. $E+F$ is a regular expression for the union of $L(E)$ and $L(F)$. That is,

$$L(E+F) = L(E) \cup L(F).$$

2. EF is a regular expession of the catenation of $L(E)$ and $L(F)$. That is,

$$L(EF) = L(E)L(F).$$

3. E^* is a regular expression of the closure of $L(E)$. That is

$$L(E^*) = (L(E))^*.$$

4. (E), a parenthesized E, is a regular expression for the same language. That is,

$$L((E)) = L(E).$$

Precedence of Regular Expressions

The regular expression operators have an assumed order of “precedence”; operators are associated with their operands in a particular order.

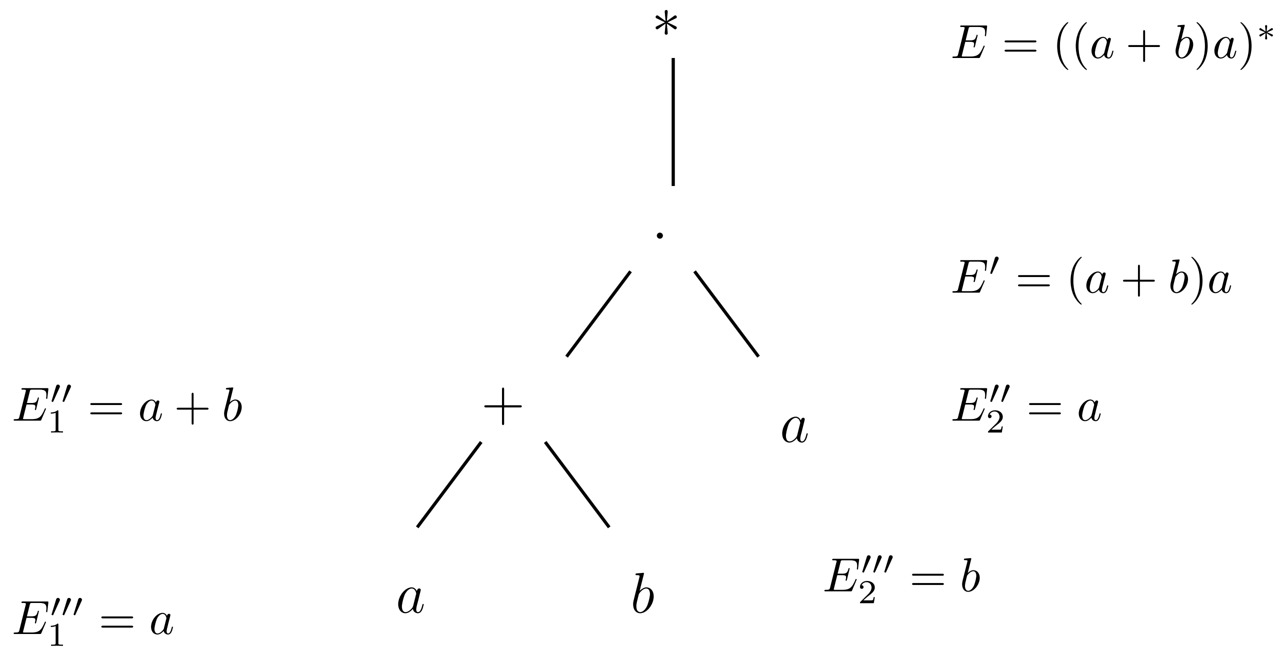
1. The star operator is of highest precedence
2. The catenation operator is next in precedence.
3. The union operator is of lowest precedence

Note that we do not always want the grouping in a regular expression to be as required by the precedence rule. If so, we can use **parentheses** to group as we want.

$$() \to {}^{*} \to \cdot \to +$$

Expression Trees

Given a regular expression E, we can display the *parsing* by an **expression tree** based on the precedence rule.



Regular Expression: Algebraic Laws

Given regular expressions E, F and G,

- Commutative law for union: $E + F = F + E$
- Commutative law for catenation: $EF \neq FE$
- Associate law for union: $(E + F) + G = E + (F + G)$
- Associative law for catenation: $(EF)G = E(FG)$
- Left distributive law of catenation over union: $E(F + G) = EF + EG$
- Right distributive law of catenation over union: $(E + F)G = EG + FG$

Regular Expression: Algebraic Laws

Given a regular expression E,

- Identities ($\emptyset$ and ε)

$$\emptyset + E = E + \emptyset = E.$$

$$\varepsilon E = E\varepsilon = E.$$

- Annihilator ($\emptyset$)

$$\emptyset E = E\emptyset = \emptyset$$

- Idempotent

$$E + E = E$$

  We define an operator to be **idempotent** if multiple applications of the operation do not change the result. Note that common arithmetic operators are not idempotent. e.g., $x+x \neq x, x \times x \neq x$

Regular Expression: Algebraic Laws

Given a regular expression E,

- $(E^*)^* = E^*$
- $\emptyset^* = \varepsilon \neq \emptyset$
- $\varepsilon^* = \varepsilon$
- $E^+ = EE^* = E^*E$ (Kleene plus or Plus closure)
- $E^* = E^+ + \varepsilon$

Regular Languages

We define a language L to be a **regular language** if and only if there is a regular expression E such that $L = L(E)$. The family of (all) regular languages is denoted by $\mathcal{L}_{REG}$.

Example: Let $E = (b^*ab^*a)^*b^*$ and
$L_{even} = \{w \mid w \in \{a,b\}^*$ and w has an even number of a's; namely, $|w|_a = 2i$ for $i \geq 0\}$.

Claim: $L(E) = L_{even}$
Proof:

1. $L(E) \subseteq L_{even}$ since every string in $L(E)$ has an even number of a's
2. Let $w \in L_{even}$. Then, we can write w as

$$w = b^{i_0}ab^{i_1}ab^{i_2}\cdots ab^{i_{2n}} \text{ for } i_0, i_1, \ldots, i_{2n} \geq 0.$$

   This implies that $w = (b^{i_0}ab^{i_1}a)(b^{i_2}ab^{i_3}a)\cdots(b^{i_{2n-2}}ab^{i_{2n-1}}a)b^{i_{2n}}$ and, therefore,

$$w \in L((b^*ab^*a)^*)L(b^*) = L(E).$$

Regular Expressions Example

Given $\Sigma = \{a, b\}$,

1. $L_1 = \{w \mid w = au \text{ and } u \in \Sigma^*\}$

2. $L_2 = \{w \mid |w|_a \equiv 0 \textbf{ mod } 3\}$

3. $L_3 = \{w \mid w$ has 2 or 3 a's with the last two appearances nonconsecutive$\}$

4. $L_4 = \{w \mid w = a^i b^j, i, j \geq 1\}$

Regular Expressions Example

Given $\Sigma = \{a, b\}$,

1. $L_1 = \{w \mid w = au \text{ and } u \in \Sigma^*\}$

   $a(a+b)^*$

2. $L_2 = \{w \mid |w|_a \equiv 0 \textbf{ mod } 3\}$

3. $L_3 = \{w \mid w \text{ has 2 or 3 } a\text{'s with the last two appearances nonconsecutive}\}$

4. $L_4 = \{w \mid w = a^i b^j, i, j \geq 1\}$

Regular Expressions Example

Given $\Sigma = \{a, b\}$,

1. $L_1 = \{w \mid w = au$ and $u \in \Sigma^*\}$

   $a(a+b)^*$

2. $L_2 = \{w \mid |w|_a \equiv 0$ **mod** $3\}$

   $(b^*ab^*ab^*a)^*b^*$

3. $L_3 = \{w \mid w$ has 2 or 3 a's with the last two appearances nonconsecutive$\}$

4. $L_4 = \{w \mid w = a^ib^j, i, j \geq 1\}$

Regular Expressions Example

Given $\Sigma = \{a, b\}$,

1. $L_1 = \{w \mid w = au \text{ and } u \in \Sigma^*\}$

   $a(a+b)^*$

2. $L_2 = \{w \mid |w|_a \equiv 0 \textbf{ mod } 3\}$

   $(b^*ab^*ab^*a)^*b^*$

3. $L_3 = \{w \mid w$ has 2 or 3 a's with the last two appearances nonconsecutive$\}$

   $b^*(a+\varepsilon)b^*ab^+ab^*$

4. $L_4 = \{w \mid w = a^ib^j, i, j \geq 1\}$

Regular Expressions Example

Given $\Sigma = \{a, b\}$,

1. $L_1 = \{w \mid w = au$ and $u \in \Sigma^*\}$

   $a(a+b)^*$

2. $L_2 = \{w \mid |w|_a \equiv 0 \textbf{ mod } 3\}$

   $(b^*ab^*ab^*a)^*b^*$

3. $L_3 = \{w \mid w$ has 2 or 3 a's with the last two appearances nonconsecutive$\}$

   $b^*(a+\varepsilon)b^*ab^+ab^*$

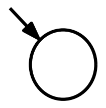
4. $L_4 = \{w \mid w = a^ib^j, i, j \geq 1\}$

   a^+b^+

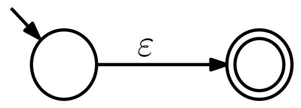
Regular Expressions into FAs

Regular Expressions into FAs

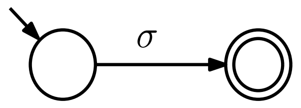
Given a regular expression E over Σ, we can construct a ε-NFA A such that $L(E) = L(A)$ using the following inductive construction:



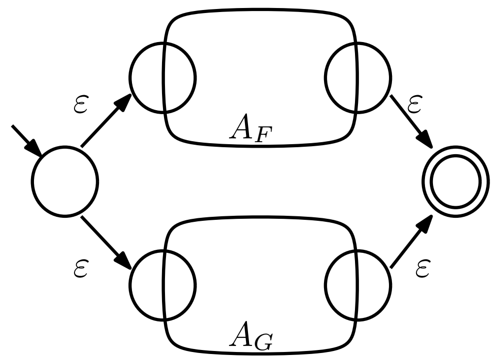
$R_1 : E = \emptyset$
$L(A) = \emptyset$



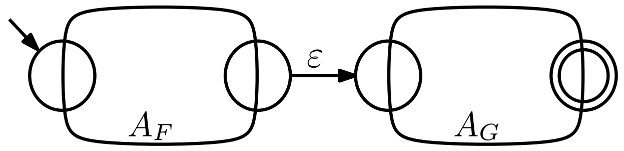
$R_2 : E = \varepsilon$
$L(A) = \{\varepsilon\}$



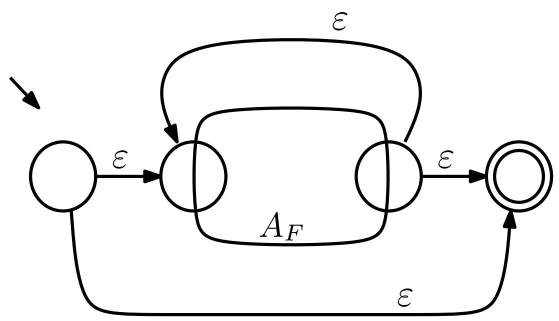
$R_3 : E = \sigma$
$L(A) = \{\sigma\}$



$R_4 : E = F + G$
$L(A) = L(F) \cup L(G)$



$R_5 : E = FG$
$L(A) = L(F)L(G)$



$R_6 : E = F^*$
$L(A) = L(E)$

Thompson Construction Example

We call this FA construction Thompson construction named after the inventor, "Ken Thompson". We can call such FAs *Thompson automata*.

Given $\Sigma = \{a, b\}$,

1. $E_1 = (a + b)^*(\varepsilon + a)$
2. $E_2 = (a + b^*)^*$
3. $E_3 = (aa + ba)(b^* + a)$
4. $E_4 = b^*(a + ab^*)^*$

Regular Expressions into FAs

Claim: Given E, the Thompson automaton A for E satisfies $L(E) = L(A)$.

Proof: Let $\mathbb{OP}(E)$ be the total number of operators($^*, \cdot, +$) in E. We prove this claim by induction on $\mathbb{OP}(E)$.

Basis: $\mathbb{OP}(E) = 0$. Then, $E = \emptyset, \varepsilon$, or $\sigma \in \Sigma$ and the claim is true by R_1, R_2 and R_3.
Hypothesis: Assume that the claim holds for all E with $\mathbb{OP}(E) \leq k$ for some $k \geq 0$.
Induction: Consider E such that $\mathbb{OP}(E) = k+1$. Since $k+1 \geq 1$, E must have at least one operator. We have three cases:

1. $E = F + G$. Note that $\mathbb{OP}(F) \leq k$ and $\mathbb{OP}(G) \leq k$. Let A_F and A_G be the corresponding FAs. Then, by the hypothesis, $L(A_F) = L(F)$ and $L(A_G) = L(G)$. Because of R_4, $L(A) = L(A_F) \cup L(A_G)$ and $L(E) = L(F) \cup L(G)$. Therefore, $L(A) = L(E)$.

2. $E = FG$.

   ....

3. $E = F^*$.

   ....

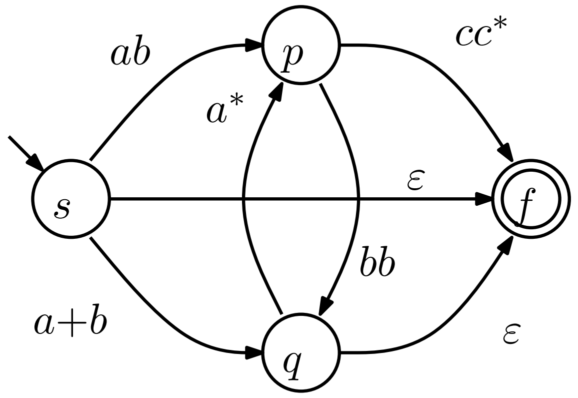
FAs into Regular Expressions

FAs into Regular Expressions

We now prove that for every FA A, there is a regular expression E such that $L(A) = L(E)$.

An expression automaton (EA): An EA A is an FA with regular expressions as transition labels. Formally, A is specified by a tuple $(Q, \Sigma, \delta, s, f^\dagger)$, where

1. Q, Σ, s are the same as in ε-NFA
2. s does not have any in-transitions
3. f is the **only** final state such that $f \neq s$ and f has no out-transitions
4. δ is a set of (Q, R_Σ, Q) (in ε-NFA, it is (Q, σ, Q))



†: To be presice, it has to be $\{f\}$. But we use f for short if not confused.

Computations for EAs

Single-step configuration in an EA A:

1. Let (p, w) be a current configuration, where $w = uv$ and $u, v \in \Sigma^*$
2. $(p, E, q) \in \delta$ and $u \in L(E)$, where E is a regular expression
3. We say that (p, w) **yields** (q, v) in **one step**. Namely, $(p, w) \vdash (q, v)$
4. We can define $\vdash^*, \vdash^+$ in a similar way: multiple-step configuration

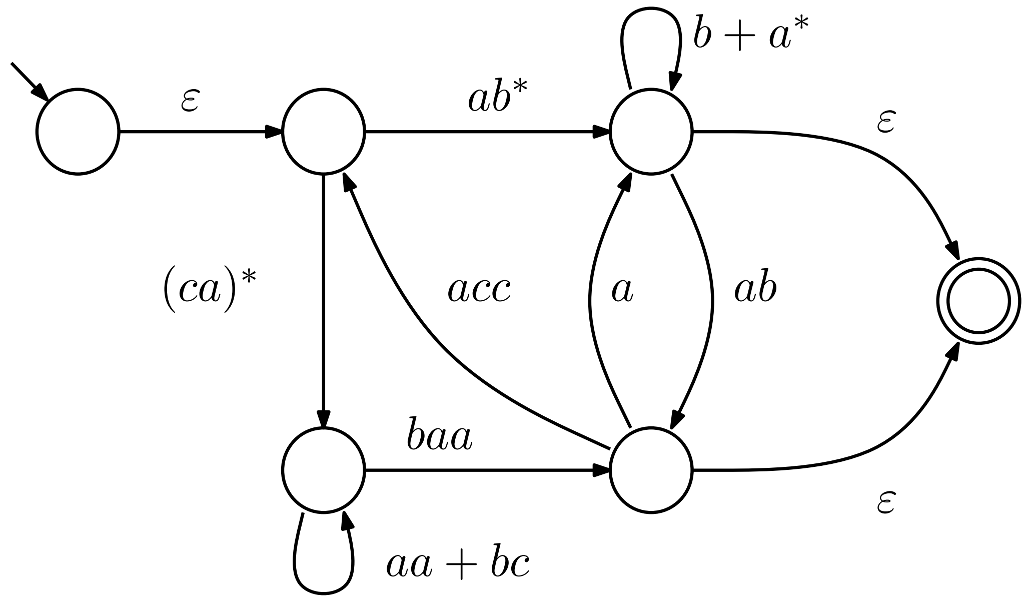
In other words, a transition (p, E, q) is applied to A if p is the current state and there is a string $u \in L(E)$ such that u is a prefix of the unread portion of w of the input string, then A moves into the next state q and the reader **consumes** u leaving v as the unread input.

Nondeterminism and Acceptance for EAs

Given an EA A and its transition (p, E, q) with the unread input $w = uv$:

1. There may be many strings that satisfy the condition, so A may be **highly** nondeterministic!
2. If $E = a^*$ and $w = a^k b$, say, where $k \gg 0$, then $u = \varepsilon, a, aa, aaa, \ldots, a^k$, are all possible choices
3. We define **acceptance** as we did for normal NFAs:
   A string w is accpetyed by an EA A if there is a computation for w that begins at the start state and ends at the final state such that w has been completely consumed.

Nondeterminism and Acceptance for EAs Example



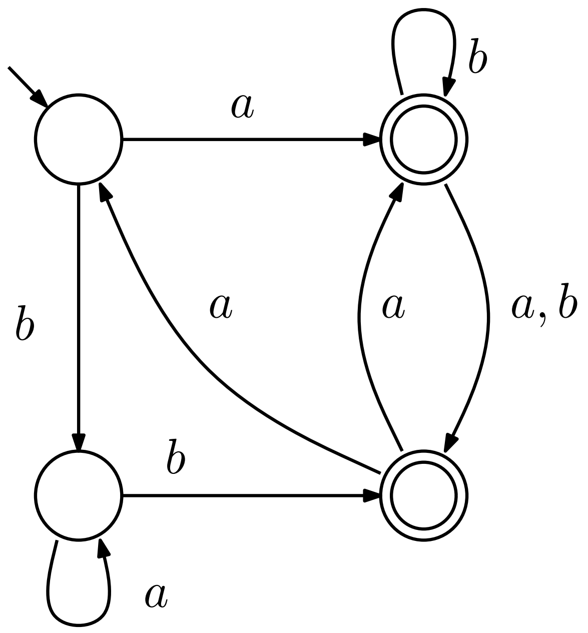
Determine if the following strings are in $L(A)$.

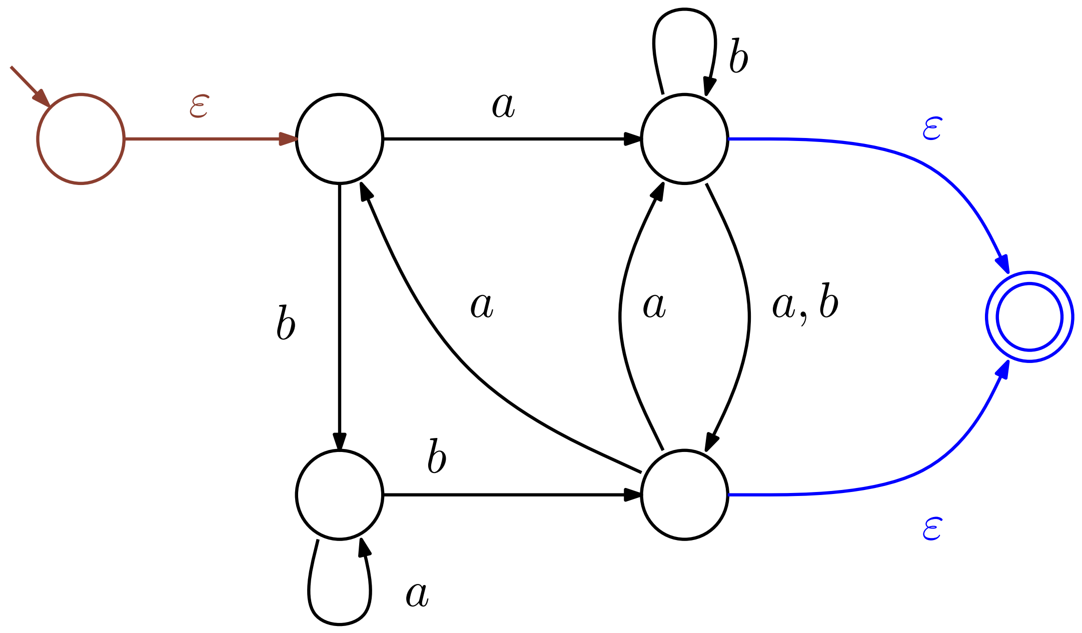
1. $caaabcbaa$
2. $aaaab$
3. baa
4. ac

FAs and EAs

Claim: Given an FA A, there is an EA A' such that $L(A) = L(A')$

Proof: It is left for an exercise. Here is an intuition:

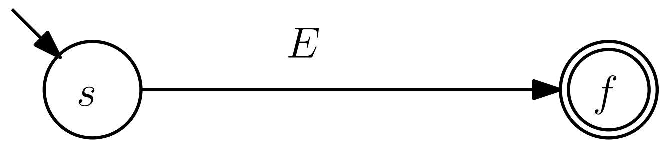




State Elimination

We are now ready to work on state elimination, a very **simple** idea for computing a regular expression from an FA. At each step, we **bypass** a nonstart, nonfinal state to give an equivalent automaton (which will be an EA) that has one less state.

Goal of the technique:

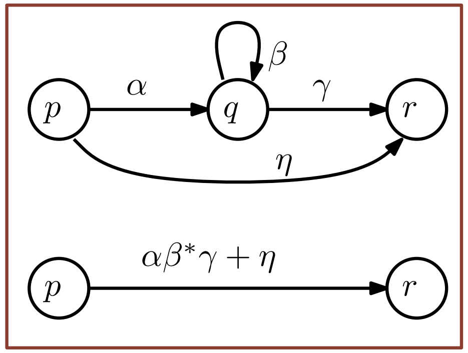


State Elimination

How does state elimination work?

1. First, consider a state q, which we wish to eliminate, that has an in-transition (p, α, q), a self-looping transition (q, β, q) and an out-transition (q, γ, r). (It may have other in/out-transitions.)

2. When we eliminate q, we replace the transition sequence



$$(p, \alpha, q), (q, \beta, q), \ldots, (q, \beta, q), (q, \gamma, r)$$

   by

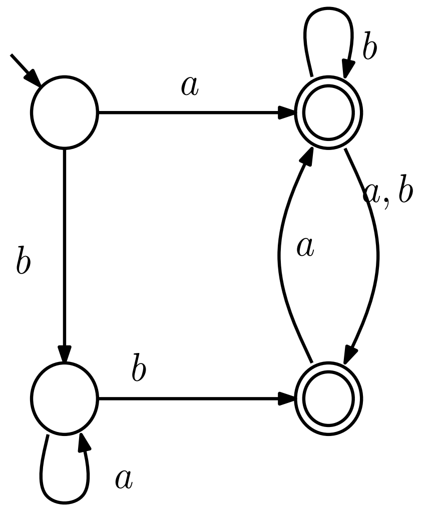
$$(p, \alpha\beta^*\gamma, r)$$

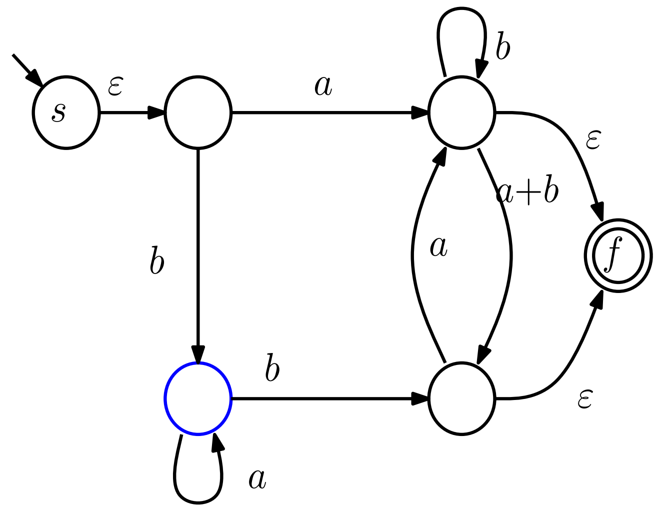
   and, since q is not final, we can see that this new transition emulates the previous transition sequence

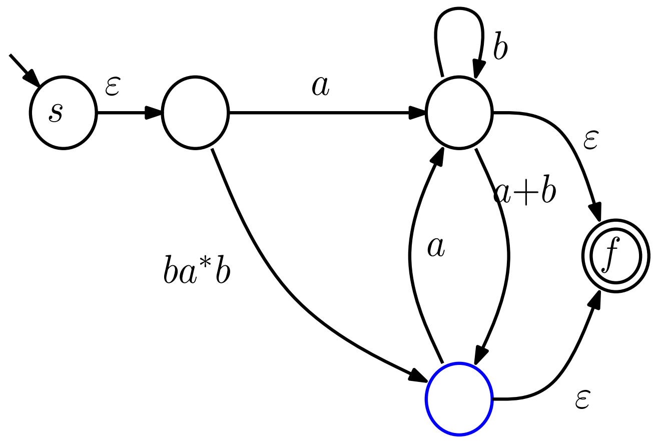
3. Finally, we union the new expression and the original expression η between p and r: $(p, \alpha\beta^*\gamma + \eta, r)$

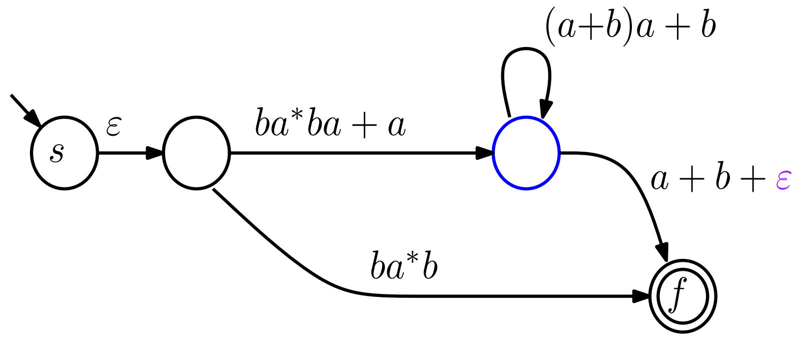
4. This observation holds for all shortcuts that we have made to avoid q, so we no longer need q after we have bypassed it.

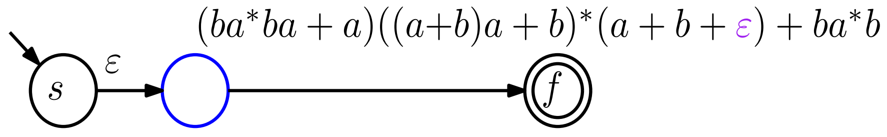
State Elimination Example

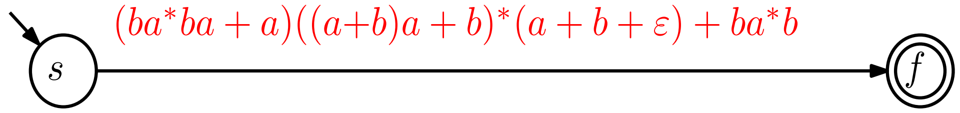












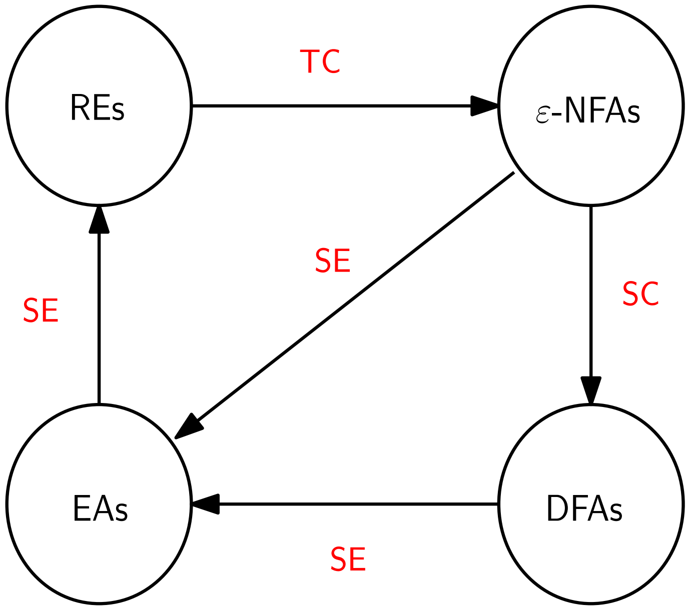
Summary of State Elimination

1. Add a **new start state** if the original one has an in-transition
2. Add a **new final state** if there are more than one final states originally or if there is a single final state but it has an out-transition. Old final states become nonfinal states.
3. Eliminate states in $Q \setminus \{s, f\}$ one by one

Summary of State Elimination

From state elimination, we know that

1. Given an FA A, we can compute a regular expression E such that $L(A) = L(E)$ using state elimination
2. EAs have the same expressive power as FAs
3. Both regular expressions and FAs define the same set of languages, **regular langauges**



Closure Properties of Regular Languages

Closure Properties of Regular Languages

Let L_1 and L_2 be regular languages. Regular languages are closed under the following operations:

- Union: $L_1 \cup L_2$
- Intersection: $L_1 \cap L_2$
- Complement: $\overline{L_1}$
- Difference: $L_1 \setminus L_2$
- Reversal: $L_1^R = \{w^R \mid w \in L_1\}$
- Kleene closure: L_1^*
- Catenation: $L_1 L_2$
- Homomorphism: $h(L_1) = \{h(w) \mid w \in L_1 \text{ and } h \text{ is a homomorphism}\}$
- Inverse homomorphism:
  $h^{-1}(L_1) = \{w \in \Delta^* \mid h(w) \in L_1, h : \Sigma \to \Delta \text{ is a homomorphism}\}$

Closure Properties of Regular Languages

Claim: Let L_1 and L_2 be regular languages. Then, the language $L_1 \cup L_2$ is regular.

Proof: Let E be a regular expression for L_1 and F be a regular expression for L_2. By definition, $L(E+F) = L_1 \cup L_2$.

Claim: Let L be a regular language. Then, the language $\overline{L}$ is regular.

Proof: Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA for L. We flip the accepting status of all states (final state $\leftrightarrow$ non-final state). Namely,

$$A' = (Q, \Sigma, \delta, s, Q \setminus F).$$

Then, $L(A') = \overline{(L)}$.

Closure Properties of Regular Languages

Claim: Let L_1 and L_2 be regular languages. Then, the language $L_1 \cap L_2$ is regular.

Proof: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ by DeMorgan's law. Since regular languages are closed under union and complement, then, the language $L_1 \cap L_2$ is regular.

Proof(2): We also look at another "direct" proof based on the Cartesian product.

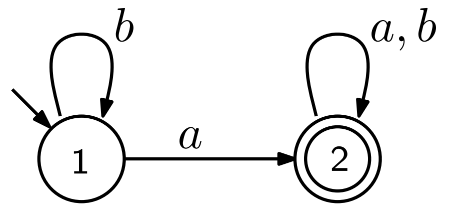
Let $A_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs for L_1 and L_2, respectively.
We construct a new FA that simulates A_1 and A_2 in parallel, and accepts a string if and only if both A_1 and A_2 accept it.
Let $A = (Q, \Sigma, \delta, s, F)$, where

1. $Q = Q_1 \times Q_2$; namely, Q is a set of state pairs
2. $\delta(q, \sigma) = (\delta(q_1, \sigma), \delta(q_2, \sigma))$, where $q = (q_1, q_2)$ and $\sigma \in \Sigma$
3. $s = (s_1, s_2)$
4. $F = F_1 \times F_2$

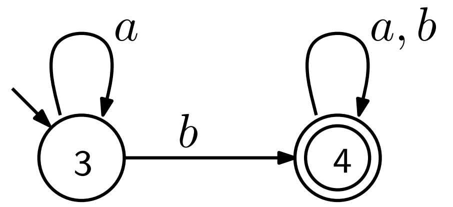
Then, $L(A) = L_1 \cap L_2$.

Cartesian Product Example

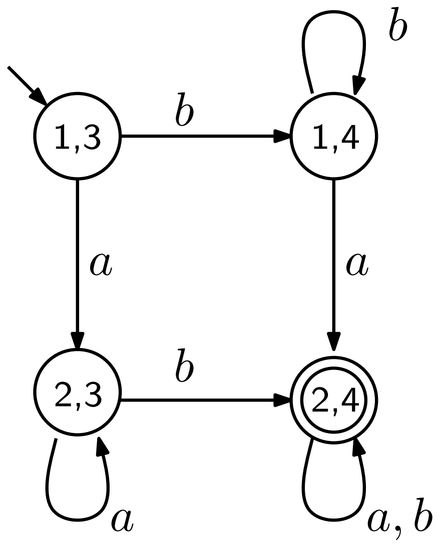
$L_1 = L(b^*a(a+b)^*)$



$L_2 = L(a^*b(a+b)^*)$



$L_1 \cap L_2$



Closure Properties of Regular Languages

Claim: Let L_1 and L_2 be regular languages. Then, the language $L_1 \setminus L_2$ is regular.

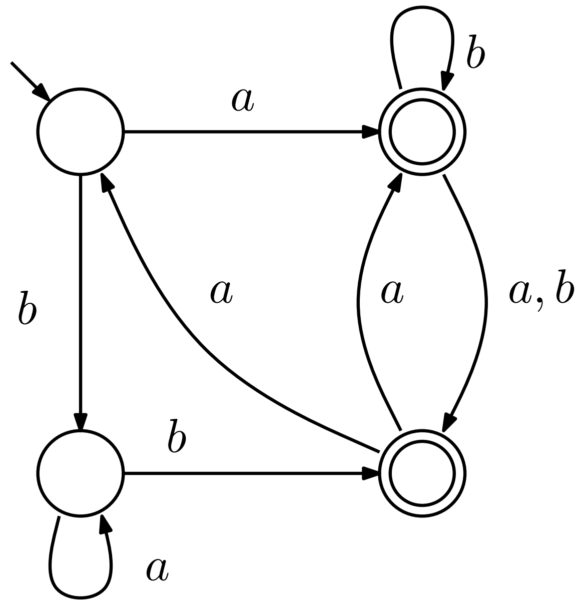
Proof: $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$.

Claim: Let L be a regular language. Then, the language L^R is regular.

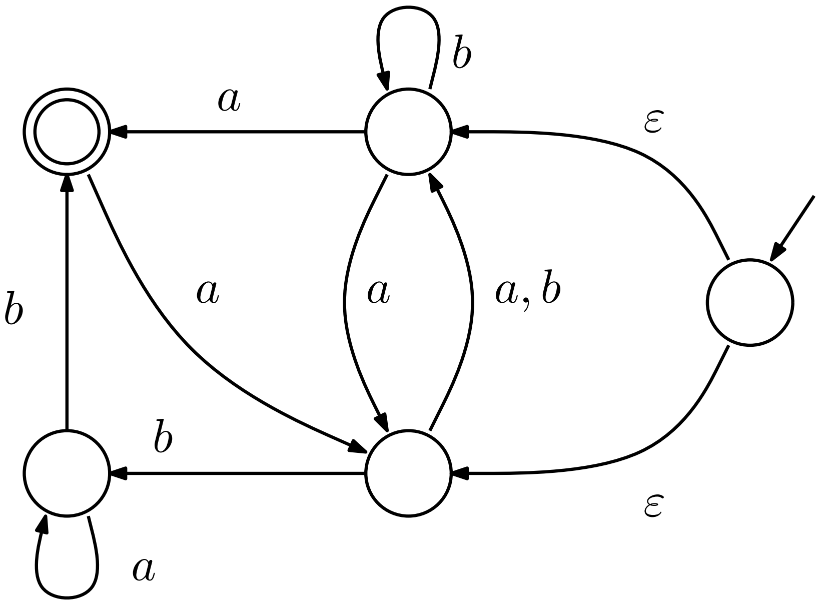
Proof: Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA for L. From A, we construct $A' = (Q, \Sigma, \delta^R, s', \{s\})$, where

1. $(q, \sigma, p) \in \delta^R$ iff $(p, \sigma, q) \in \delta$ for all states $p, q \in Q$
2. $(s', \varepsilon, f) \in \delta^R$ for all $f \in F$
3. s is the new single final state in A'

Reversal FA Construction Example



$L(A)$



$L(A)^R$

Closure Properties of Regular Languages

Let Σ and Γ be alphabets. We define homomorphism to be a function

$$h : \Sigma \to \Gamma^*.$$

Note that homomorphism is "substituting" a particular string for each symbol in Σ.

Let $w = \sigma_1\sigma_2 \cdots \sigma_n \in \Sigma^*$. Then,

1. $h(w) = h(\sigma_1)h(\sigma_2) \cdots h(\sigma_n)$ and
2. $h(L) = \{h(w) \mid w \in L\}$

Example: Let $h : \{0, 1\} \to \{a, b\}^*$ be defined by $h(0) = ab$ and $h(1) = \varepsilon$. Then,

1. $h(0011) = abab$
2. $h(L(10^*1)) = L((ab)^*)$

We can extend an homomorphism to $h : \Sigma^* \to \Gamma^*$ with the property that for every $u, v \in \Sigma^*$

$$h(uv) = h(u)h(v)$$

Closure Properties of Regular Languages

Claim: Let L be a regular language and h be a homomorphism. Then, the language $h(L)$ is regular.

Proof: Let us have the language $L = L(E)$ for a regular expression E. We prove that $h(L(E))$ is regular.

Basis:

1. If E is ε or $\emptyset$, then $h(L(E)) = L(E)$ since h does not affect the string ε and the language $\emptyset$. This implies that $h(L(E))$ is regular.
2. If $E = \sigma$, then $L(E) = \{\sigma\}$.
   This implies that $h(L(E)) = \{h(\sigma)\}$, for a string $h(\sigma) \in \Gamma^*$, which is also regular.

Induction:

1. $E = G + F$. Then, $h(L(E)) = h(L(G+F)) = h(L(G) \cup L(F)) = h(L(G)) \cup h(L(F))$ which is regular.
2. $E = GF$. Then, $h(L(E)) = h(L(GF)) = h(L(G)L(F)) = h(L(G))h(L(F))$ which is regular.
3. $E = (G^*)$. Then, $h(L(E) = h(L(G^*)) = h(L(G)^*) = h(L(G))^*$ which is regular.

Pumping Lemma for Regular Languages

Proving Regularity

A set of regular languages preserves certain regularity.

Here is a language that does not seem to be regular. Let $\Sigma = \{0, 1, \ldots, 9\}$ and L be the set of decimal representations of the natural numbers[†] that are divisible by 2 or 3.
e.g., $0, 2, 3, 843290, 578421, \ldots$.

- The set N of decimal representations of the natural numbers is
$$\{0\} \cup ((\Sigma \setminus \{0\}) \cdot \Sigma^*).$$
Clearly, N is regular
- The set E of decimal representations of the even natural numbers is
$$N \cap (\Sigma^* \cdot \{0, 2, 4, 6, 8\}).$$
Clearly, E is also regular

[†]: Here natural numbers are $0, 1, 2, \ldots$.

Proving Regularity

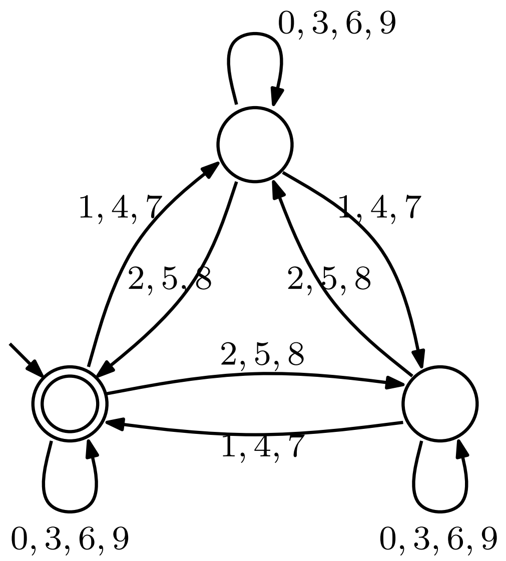
Here is a language that does not seem to be regular. Let $\Sigma = \{0, 1, \ldots, 9\}$ and L be the set of decimal representations of the natural numbers that are divisible by 2 or 3.

- The set T of decimal representations of the natural numbers that are multiples of 3 is

$$N \cap L(A),$$

  where $L(A)$ is the set of all strings accepted by the following DFA A. Since N and $L(A)$ are regular and the family of regular languages is closed under intersection, T is regular

- Hence, $L = E \cup T$ is regular



Nonregular Languages

The most famous nonregular language L is

$$L = \{a^i b^i \mid i \geq 0\}.$$

Why is it nonregular?

1. Intuitively, L is not regular because any automaton that can recongnize L must remember how many a's it has been seen so far (an unlimited number of possibilities)
2. But, this intuition is not a proof!
3. Most important, our intuition can sometimes lead us astray. Consider the two following languages:
   (a) $L_1 = \{w \mid w$ has the same number of a's and b's$\}$
   (b) $L_2 = \{w \mid w$ has the same number of the substrings ab and $ba\}$

   Are they regular?

Aside: The Pigeonhole Principle

The pigeonhole principle is: *Assume that we are given $n \geq 1$ pigeonholes and $m > n$ letters; however we assign the letters to pigeonholes, we can guarantee that some pigeonhole contains at least two letters.*

This result is very simple yet useful as we shall see. We prove the pigeonhole principle using induction on the number of pigeonholes.

Basis: $n = 1$ and $m \geq 2$. Thus, the only pigeonhole contains at least two letters.
Inductive hypothesis: Assume the claim holds for some $n \geq 1$.
Induction: We are given $n + 1$ pigeonholes and $m > n + 1$ letters. There are two cases:

1. One pigeonhole has at most one letter in it: Remove the chosen pigeonhole and its letter if it has one. Now observe that we have n pigeonholes and either m or $m-1$ letters. But, $m > n+1$ implies that $m > m - 1 > n$, so, by IH, the claim holds
2. No pigeonhole has fewer than two letters in it: This case is immediate

Proving nonregularity Example

We first prove, directly and by contradiction, that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

1. Assume that L is regular. This implies that there is a DFA $A = (Q, \Sigma, \delta, s, F)$ such that $L(A) = L$. We now derive a contradiction.

2. Let $n = |Q|$. There are infinitely many strings $w \in L$ such that $|w| = 2m$ and $m \geq n$. Consider one such string $w = a^n b^n$.

3. Now, since $w \in L(A)$, there is an accepting computation for w in A. We will write the computation as the sequence of transitions that are used in the computation:

$$(p_0 = s, \sigma_1, p_1), (p_1, \sigma_2, p_2), \ldots, (p_{2n-1}, \sigma_{2n}, p_{2n}),$$

   where $w = \sigma_1 \sigma_2 \cdots \sigma_{2n}$ and $p_{2n} \in F$.

4. The crucial observation is that there are $2n + 1$ appearances of states in the computation; namely, $p_0, p_1, \ldots, p_{2n}$. But, there are only n different states; therefore, there must be multiple appearances of some states (Pigeonhole Principle).
   (We can think of states $\equiv$ pigeonholes, and appearances of states $\equiv$ letters)

Proving nonregularity Example

We first prove, directly and by contradiction, that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

5. Moreover, there must be at least two appearances of at least one state in the first n steps of the computation for the same reason. We conclude that there are p_i and p_j such that $0 \leq i < j \leq n$ and $p_i = p_j$.

6. Consider the portion of the computation from p_i to p_j. It consumes the substring $\sigma_{i+1} \cdots \sigma_j$

7. What do we know about this string? First, it is nonempty. Since $i < j$, it consists of at least one character. Second, it is a string of *only* a's; thus, $\sigma_{i+1} \cdots \sigma_j = a^{j-i}$

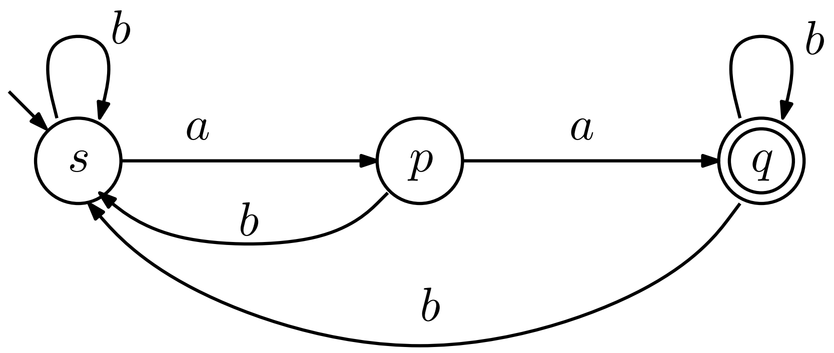
8. Finally, since this subcomputation begins and ends in the same actural state, we can omit it from the computation and *still have a valid accepting computation for a smaller string*

9. But, this omission implies that $a^{n+i-j}b^n$ is accpeted—a contradiction (Note that since $i \neq j, n + i - j < n$)

Regular Pumping Lemma

This is really a theorem!

1. It is a property of all (infinite) regular languages
2. All strings in a regular language can be pumped if they are at least as long as a given positive integer, the pumping constant.
3. Infinite regular languages must have repetitive substructures that arise from a Kleene star in a regular expression or a cycle in a state diagram



Pumping Lemma

Let L be a regular language. Then, there is a pumping constant $n \geq 1$ such that if w is any string in L of length at least n, then w can be written as the catenation of three substrings $w = xyz$ (we often call this *factoring*) that satisfy the following three conditions:

1. $|y| > 0$
2. $|xy| \leq n$
3. For all $i \geq 0$, $xy^iz \in L$

Note that $|y| > 0$ implies that y is a not the empty string. Also note that when we proved the first nonregularity result, we factored $w = a^ib^i$ into three substrings.

Proof of Pumping Lemma

1. Since L is regular, there is a DFA A for L
2. Let n be the number of states in A
3. Let w be *any string* in L of length $m \geq n$, where $w = \sigma_1\sigma_2 \cdots \sigma_m$, for $\sigma_i \in \Sigma$. Consider the accepting computation of w in A: $(p_0, \sigma_1, p_1), (p_1, \sigma_2, p_2), \ldots, (p_{n-1}, \sigma_n, p_n), \ldots$
4. Since A has n different states, by the Pigeonhole principle, there are i and j, $0 \leq i < j \leq n$ such that $p_i = p_j$
5. Hence, the substring $\sigma_{i+1}\sigma_{i+2} \cdots \sigma_j$ drives A from p_i back to $p_j = p_i$
6. Let $x = \sigma_1 \cdots \sigma_i, y = \sigma_{i+1} \cdots \sigma_j$ and $z = \sigma_{j+1} \cdots \sigma_m$. Since $i < j, |y| > 0$ and, since $j \leq n, |xy| \leq n$
7. Now y can either be removed from w or be repeated many times and A still accepts the resulting strings. Hence, A accepts xy^iz for all $i \geq 0$

Proof of Pumping Lemma

Notes:

1. If all strings in L have length less than n, then the theorem is vacuously true. L is finite
2. Even if we remove the second condition $|xy| \leq n$, the claim still holds. This condition guarantees only that the first state repetition appears within the first n characters of the string

Using Pumping Lemma

Main idea: Proof by contradiction

1. To prove that a language L is not regular, we assume that L is regular
2. By the Pumping Lemma, we know that there is a pumping constant $n \geq 1$ such that *all strings* in L of length at least n can be pumped
3. Attempt to identify *one string* w in L that has length at least n but cannot be pumped
   (a) Choose *one string* $w \in L$ with $|w| \geq n$
   (b) Consider *all valid* ways of factoring w into xyz
   (c) For each valid factoring, demonstrate *one value* i such that $xy^iz \notin L$
4. The existence of such a w contradicts the assumed regularity of L
5. Hence, L is not regular

Using PL Exmaple I

Claim: $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

Proof:

1. Suppose L is regular. Then, the Pumping Lemma applies
2. Let n be the pumping constant
3. Choose $w = a^n b^n$ (Note that $w \in L$ and $|w| \geq n$)
4. By PL, w can be factored into xyz such that $|xy| \leq n, |y| > 0$ and, for all $i \geq 0$, $xy^i z \in L$
5. Since $|xy| \leq n$ and $|y| > 0$, the string y consists only of a's and has at least one a. Let $y = a^k$, for some $k > 0$
6. Consider $i = 0$, then $xy^0 z = xz = a^{n-k} b^n \notin L$. We have obtained a contradiction of the assumed regularity of L
7. Hence, L is not regular

The Pumping Game

We can see the pumping lemma proof as a game between you and an Adversary (opponent). You want to show that L is not regular and the adversary wants to thwart you.

1. The adversary picks n
2. Given n, you pick a string w in L of length equal or greater than n
3. The adversary chooses the factoring of $w = xyz$, subject to $|xy| \leq n, |y| \geq 1$. You have to assume that the opponent makes the hardest choice for us to win the game
4. You try to pick i such that the pumped string $xy^iz \notin L$. If you can do so, you win the game. Otherwise, the adversary wins

Using PL Exmaple II

Claim: $L = \{a^i \mid i \text{ is a prime number}\}$ is not regular.

Proof:

1. Assume L is regular. Then, the Pumping Lemma applies
2. Let n be the pumping constant
3. Choose $w = a^P \in L$, where $P \geq n$ is a prime
4. By PL, w can be factored into xyz such that $|xy| \leq n, |y| > 0$ and, for all $i \geq 0$, $xy^iz \in L$
5. Since $|xy| \leq n$ and $|y| > 0$, $x = a^p, y = a^q, z = a^r$, where $p, q \geq 0$ and $p + q \leq n$
6. Consider $xy^{p+2q+r+2}z$. Then, the total number of a's is $p+q(p+2q+r+2)+r = (q+1)(p+2q+r)$. This gives a contradiction since the number is even and greater than 2. Hence, a^P is not in L—a contradiction
7. Hence, L is not regular

Proving Nonregularity

We can also prove that a language is not regular by using closure properties of regular languages to arrive at a contradiction.

Claim: $L = \{w \in \{a,b\}^* \mid w$ has the same numbers of a's and b's$\}$ is not regular.

Proof: Observe that $L \cap L(a^*b^*) = \{a^ib^i \mid i \geq 0\}$. If L were regular, then $L \cap L(a^*b^*)$ would also be regular because of closure under intersection. But, we have already proved that $\{a^ib^i \mid i \geq 0\}$ is not regular.

Example: Show that

$$L = \{a^nb^kc^{n+k} \mid n \geq 0, k \geq 0\}$$

is not regular.

Hint: You can prove this using the pumping lemma but you may want to use the closure property of regular languages under "homomorphism".

Additional Remarks

1. Is it possible to use the Pumping Lemma to prove that a language is regular?
2. If a language is not regular, is it always possible to prove that it is not regular using the Pumping Lemma?

1. WARNING: There are nonregular languages that satisfy the pumping lemma. For example, given $\Sigma = \{a, b, c\}$
$$L = (aa^*c)^n(bb^*c)^n + \Sigma^*cc\Sigma^*$$
is not regular but it satisfies the pumping lemma conditions
2. Thus, for a language L to be regular, satisfying the pumping lemma conditions is necessary but not sufficient
3. If L is finite, then it is always regular