Finite State Machines

FOCS, Fall 2020

Last week

- Decision functions
- Formal languages = sets of strings
- Regular languages, regular expressions

Let's dig a little bit deeper into regular languages

How to check if a string is in a regular language?

We gave a definition of what it means for a string to match a regular expression

You can turn that into an algorithm directly, but it's not a very efficient algorithm - in part because of this rule:

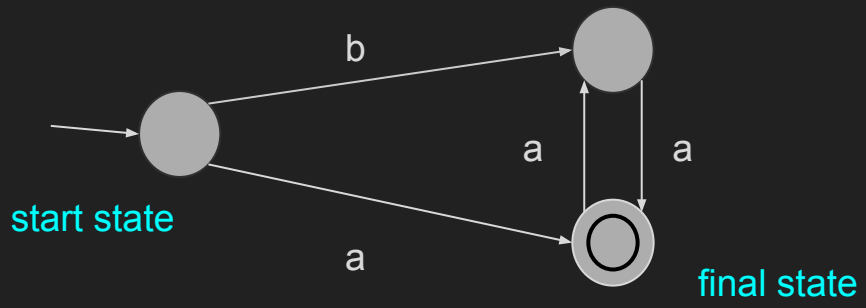
> A string matches $r_1 + r_2$ if it matches r_1 or it matches r_2

Intuitively, this requires first checking that the string matches r_1, and if that check fails, then check if matches r_2.

(This is called a backtracking regular expression matcher, by the way)

Finite state machine

A more efficient way of checking matching for regular expressions is to use a finite state machine.



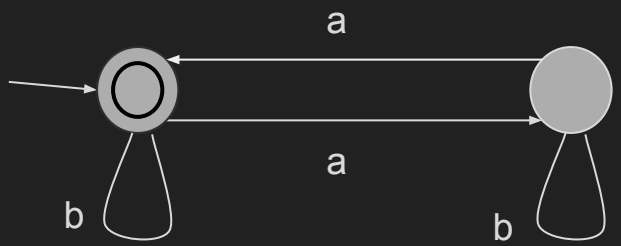
Finite set of states with labeled transitions between them (labels over alphabet Σ)

Given an input string, follow the symbols in the input strings starting from the start state — if you reach a final state, accept the input string; otherwise, reject

The above machine accepts ba, baaa, baaaaa, …, a, aaa, aaaaaa, ...

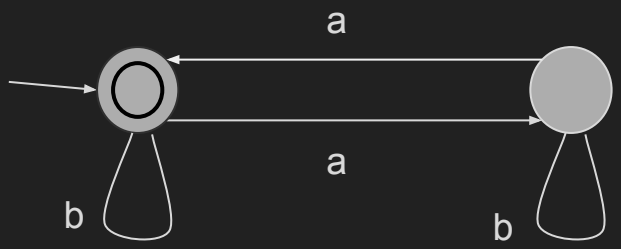
Examples

Σ = { a, b }



Examples

Σ = { a, b }

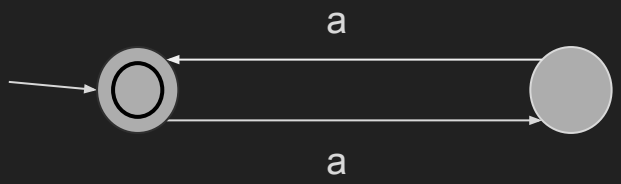


Accepts ε, aa, aaaa, aaaaaa, baa, baba, babbbab, …

Accepts all strings with an even number of a's and any number of b's

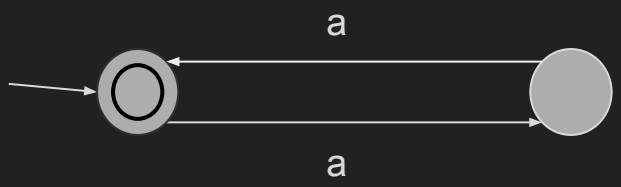
Examples

Σ = { a, b }



Examples

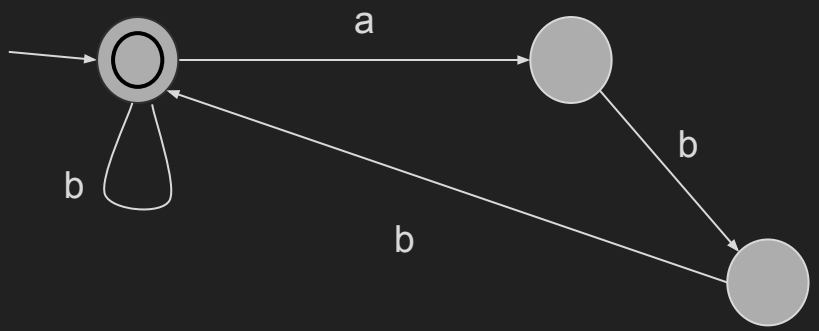
Σ = { a, b }



Accepts all strings with an even number of a's and no b's

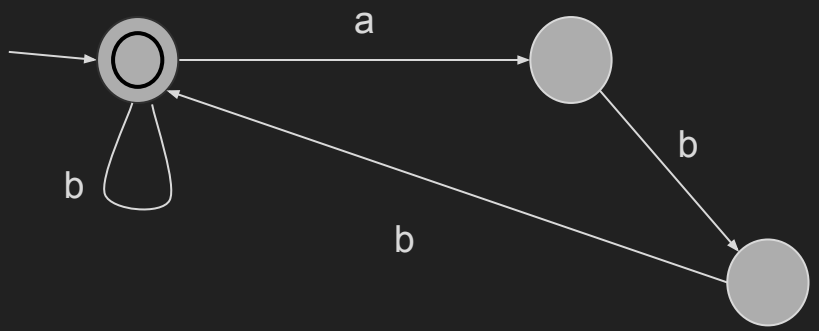
Examples

Σ = { a, b }



Examples

Σ = { a, b }

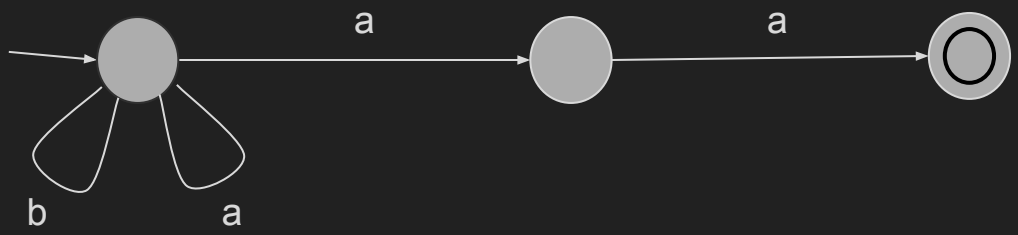


Accepts all strings in which every a is followed by at least 2 b's

Examples

A non-deterministic example (multiple choices)

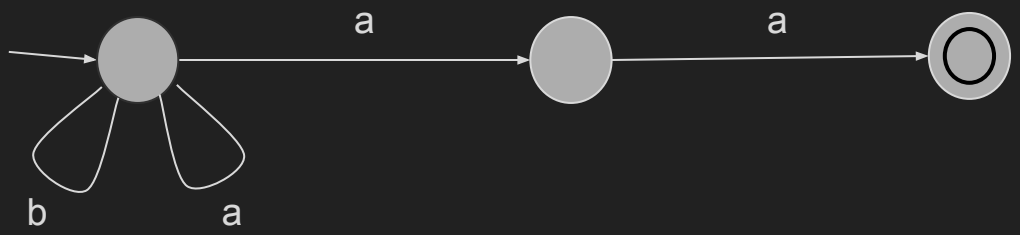
Σ = { a, b }



Examples

A non-deterministic example (multiple choices)

Σ = { a, b }



Accepts all strings ending with two a'

Formal definition

A finite state machine (or finite automaton) is a structure M = (Q, Σ, Δ, s, F) where

- Q is a finite set of states
- Σ is a finite alphabet
- Δ is a relation over Q x Σ x Q
  - (q, a, p) $\in \Delta$ when you can go from state q to state p by reading symbol a
- s is the start state ($\in$ Q)
- F is a set of final states (each $\in$ Q)

Accepting a string

A finite state machine M = (Q, Σ, Δ, s, F) accepts string u = a_1 a_2 … a_k if there exists a sequence of states q_0, q_1, q_2, …, q_k such that

- q_0 = s
- (q_{i-1}, a_i, qi) $\in$ Δ for 1 ≤ i ≤ k
- q_k $\in$ F

Language of a finite state machine

The language L(M) accepted by finite state machine M is

$$L(M) = \{ u \in \Sigma^* \mid M \text{ accepts } u \}$$

Theorem: *A language can be accepted by some finite state machine if and only if it is regular*

We show one direction of the proof: if a language is regular, then there is a finite state machine that accepts it.

Regular expressions to finite state machines

Assume a language is regular. Then there is a regular expression that denotes it. If we can figure out how to transform any regular expression into a finite state machine that accepts the language denoted by the regular expression, we can construct a finite state machine to accept that regular language.

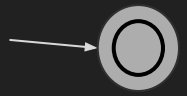
We define a recursive process that takes a regular expression and transforms it into a finite state machine that accepts the same language.

$$\text{build} : \text{RegularExpressions} \rightarrow \text{FiniteStateMachines}$$

Regular expression 1

Regular expression 1 denotes language {ε}

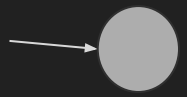
We can easily build a finite state machines accepting {ε}



Regular expression 0

Regular expression 0 denotes language $\varnothing$

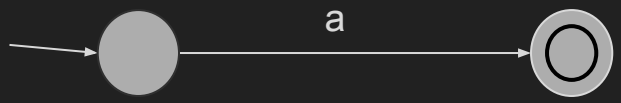
We can easily build a finite state machines accepting $\varnothing$



Regular expression a

Regular expression a denotes language { a }

We can easily build a finite state machines accepting { a }

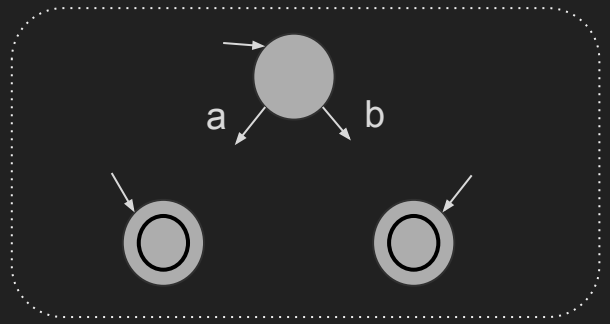


Regular expression $r_1 + r_2$

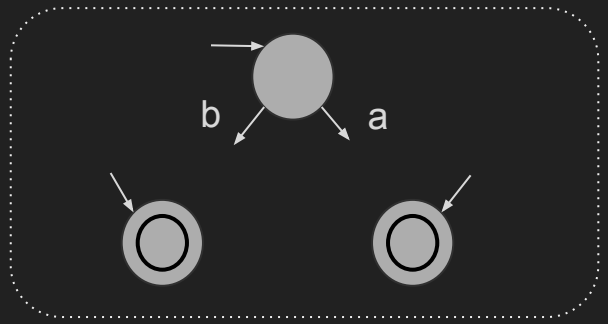
Regular expression $r_1 + r_2$ denotes language $\text{lang}(r_1) \cup \text{lang}(r_2)$

Recursively build M_1 that accepts $\text{lang}(r_1)$ and M_2 that accepts $\text{lang}(r_2)$

Perform some surgery to build M that accepts $\text{lang}(r_1) \cup \text{lang}(r_2)$



M_1



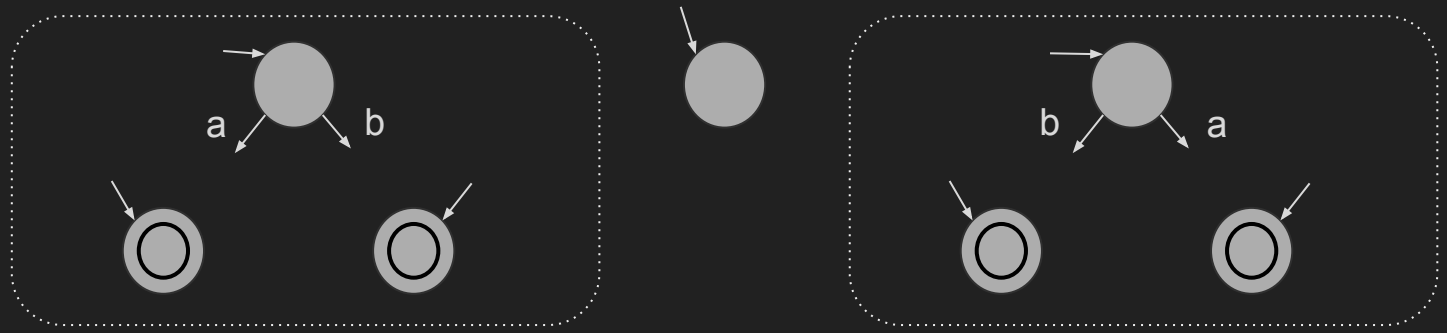
M_2

Regular expression $r_1 + r_2$

Regular expression $r_1 + r_2$ denotes language $\text{lang}(r_1) \cup \text{lang}(r_2)$

Recursively build M_1 that accepts $\text{lang}(r_1)$ and M_2 that accepts $\text{lang}(r_2)$

Perform some surgery to build M that accepts $\text{lang}(r_1) \cup \text{lang}(r_2)$

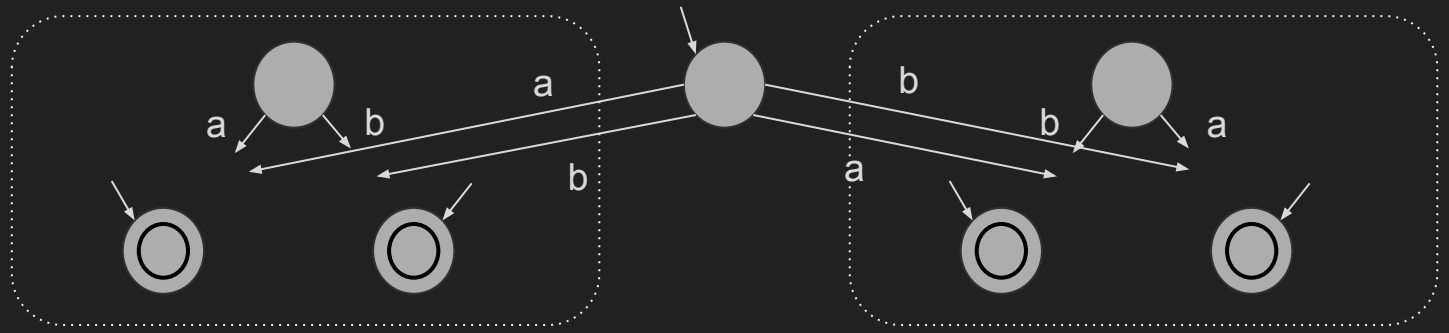


Regular expression $r_1 + r_2$

Regular expression $r_1 + r_2$ denotes language $lang(r_1) \cup lang(r_2)$

Recursively build M_1 that accepts $lang(r_1)$ and M_2 that accepts $lang(r_2)$

Perform some surgery to build M that accepts $lang(r_1) \cup lang(r_2)$

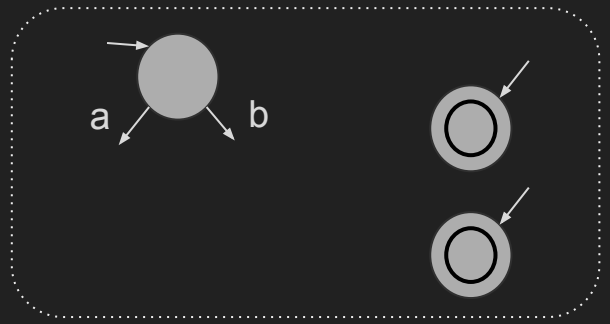


Regular expression r_1 r_2

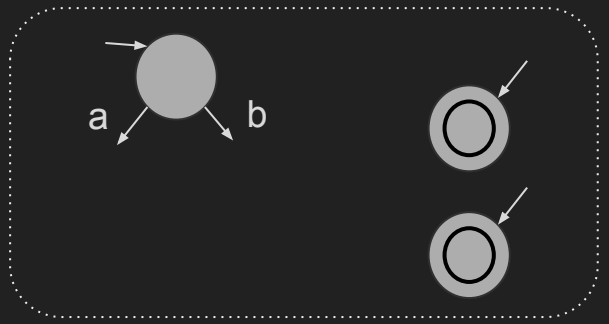
Regular expression r_1 r_2 denotes language lang(r_1) · lang(r_2)

Recursively build M_1 that accepts lang(r_1) and M_2 that accepts lang(r_2)

Perform some surgery to build M that accepts lang(r_1) · lang(r_2)



M_1



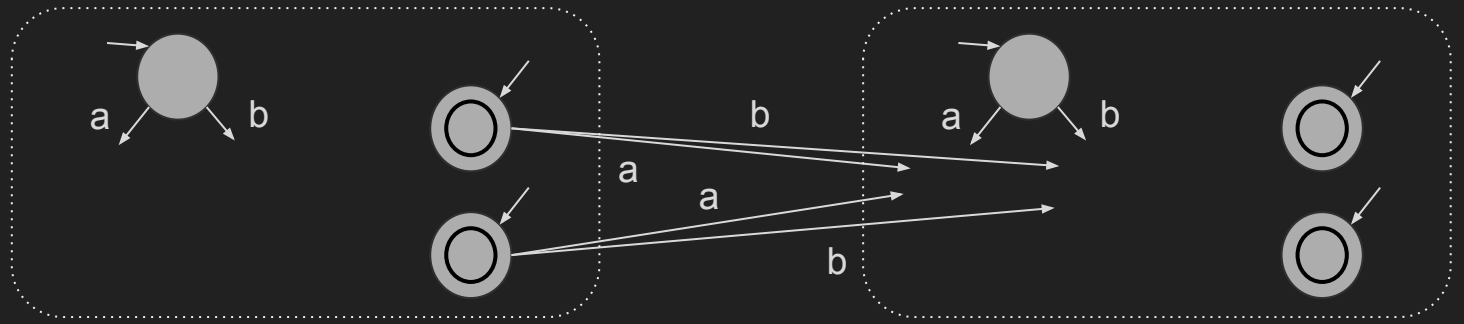
M_2

Regular expression r_1 r_2

Regular expression r_1 r_2 denotes language $lang(r_1) \cdot lang(r_2)$

Recursively build M_1 that accepts $lang(r_1)$ and M_2 that accepts $lang(r_2)$

Perform some surgery to build M that accepts $lang(r_1) \cdot lang(r_2)$

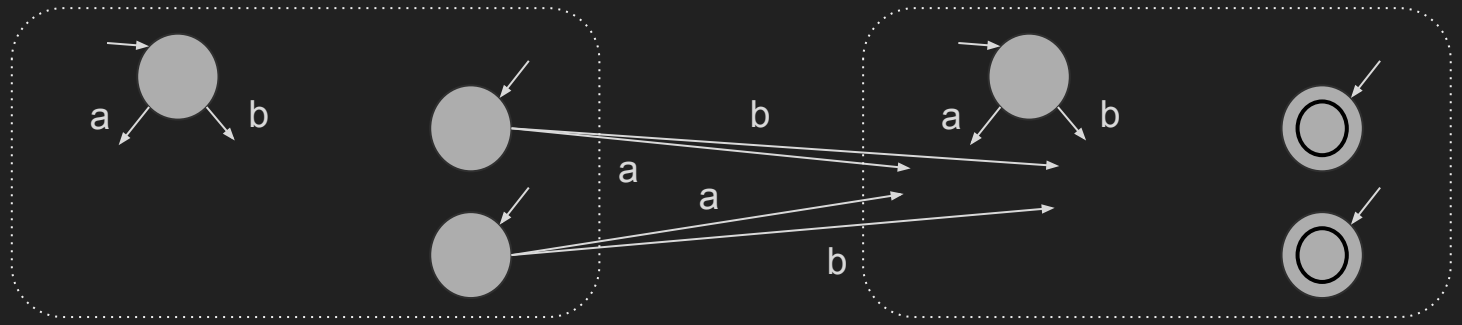


Regular expression r_1 r_2

Regular expression r_1 r_2 denotes language lang(r_1) · lang(r_2)

Recursively build M_1 that accepts lang(r_1) and M_2 that accepts lang(r_2)

Perform some surgery to build M that accepts lang(r_1) · lang(r_2)



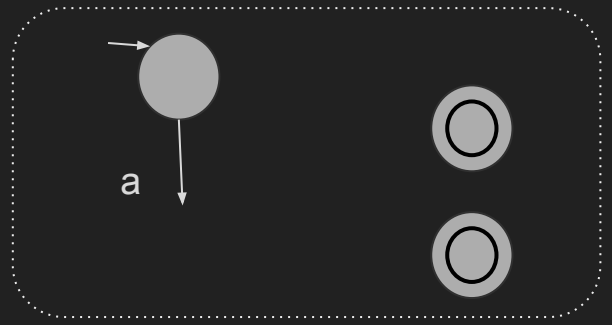
"Unfinal" these is s_2 is not final

Regular expression r_1*

Regular expression r* denotes language lang(r_1)*

Recursively build M_1 that accepts lang(r_1)

Perform some surgery to build M that accepts lang(r_1)*



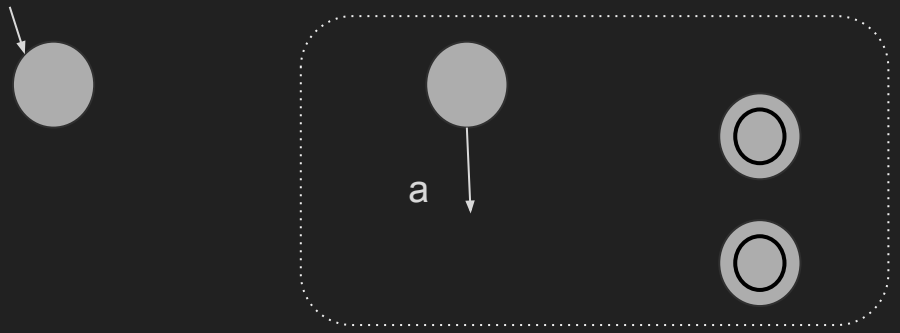
M_1

Regular expression r_1*

Regular expression r* denotes language lang(r_1)*

Recursively build M_1 that accepts lang(r_1)

Perform some surgery to build M that accepts lang(r_1)*

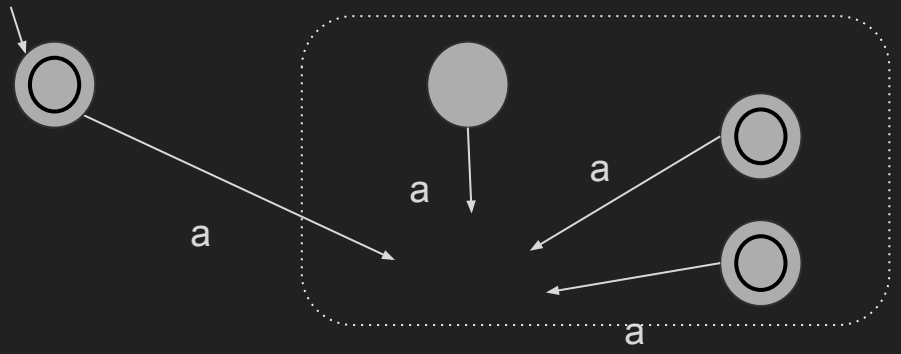


Regular expression r_1*

Regular expression r* denotes language lang(r_1)*

Recursively build M_1 that accepts lang(r_1)

Perform some surgery to build M that accepts lang(r_1)*



Deterministic finite state machines

Deterministic finite state machines are finite state machines where there is at most one transition per symbol out of every state.

Finite state machines that are not deterministic sometimes called nondeterministic finite state machines for emphasis.

Sounds like a restricted form of finite state machine — but every regular language can be accepted by some deterministic finite state machine.

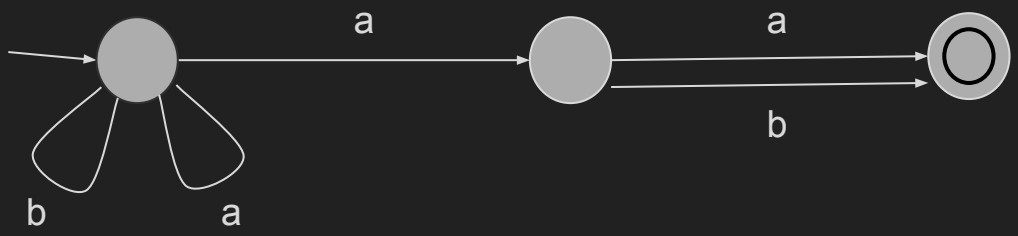
This is because we can always transform a finite state machine accepting language A into a deterministic finite state machine accepting language A.

The subset construction

Idea: given a finite state machine M, create a new finite state machine M' where

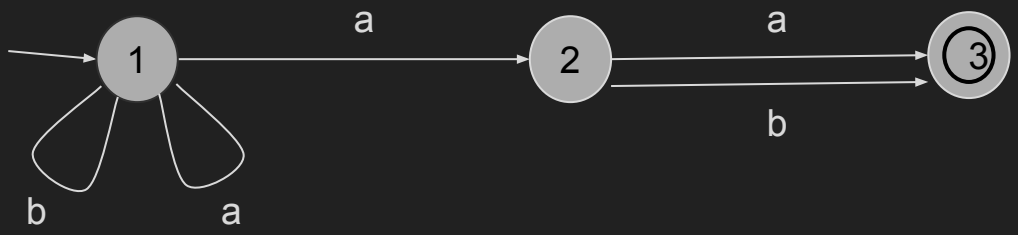
- states of M' are *sets of states of M*
- there is an a-transition from P_1 to P_2 in M' if P_2 is the set of states of M that are reachable via an a-transition in M from a state in P_1
- the start state of M' is {s} where s is the start state of M
- the final states of M' are those containing a final state of M

Example

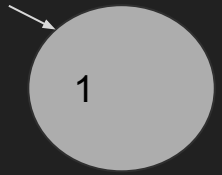


Σ = {a,b}

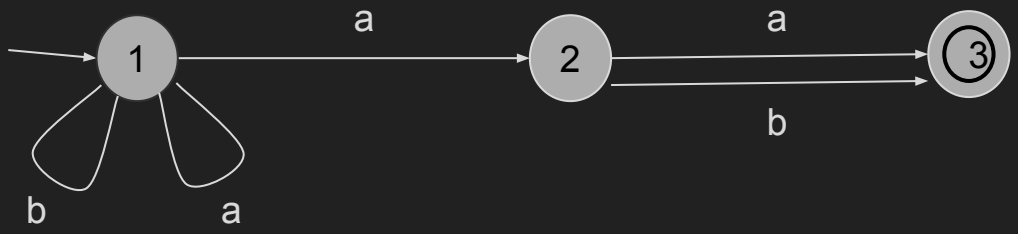
Example



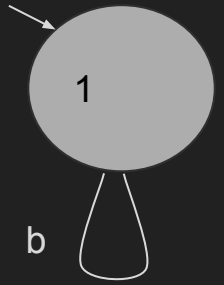
Σ = {a,b}



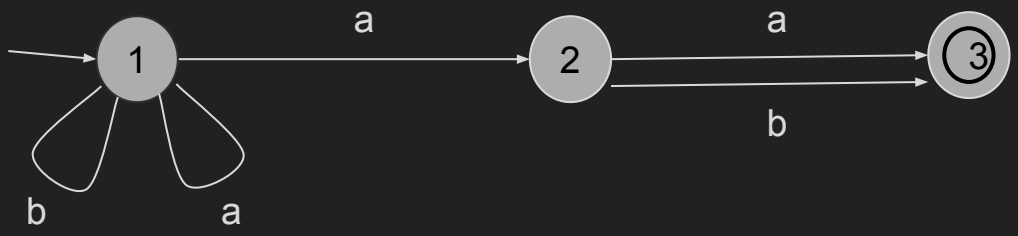
Example



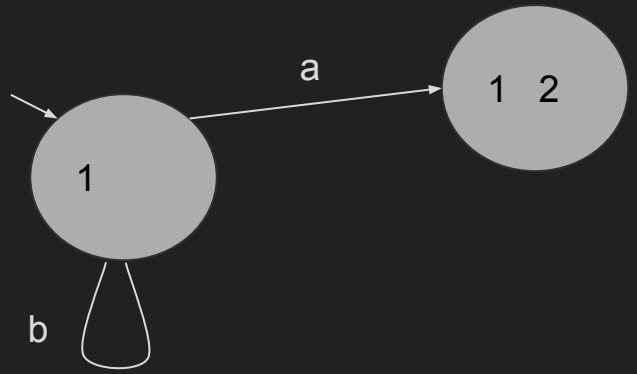
Σ = {a,b}



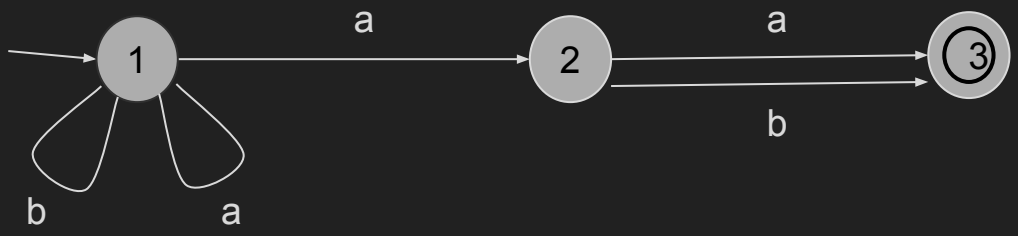
Example



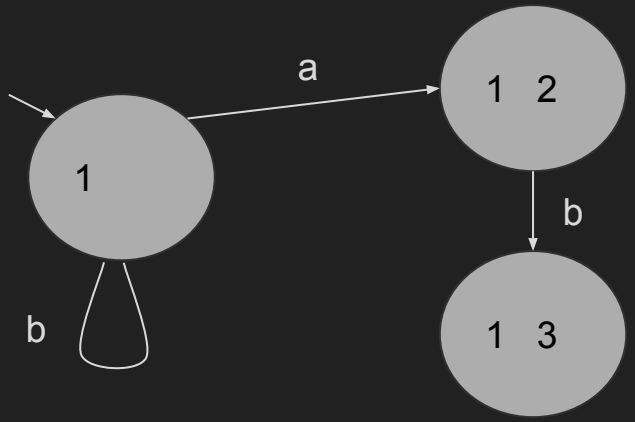
Σ = {a,b}



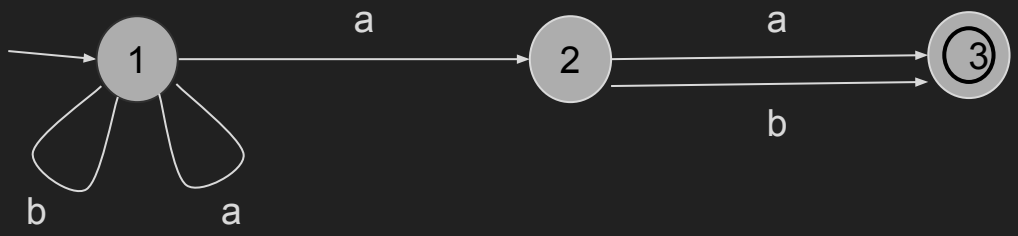
Example



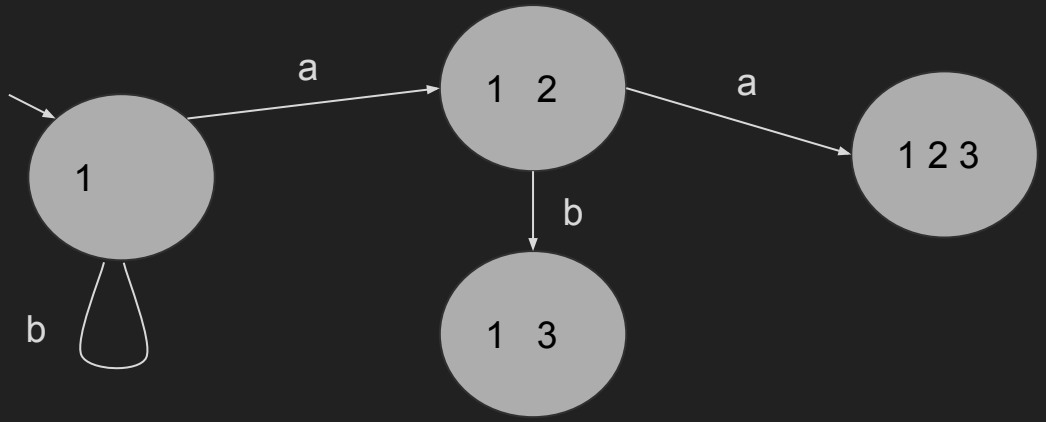
Σ = {a,b}



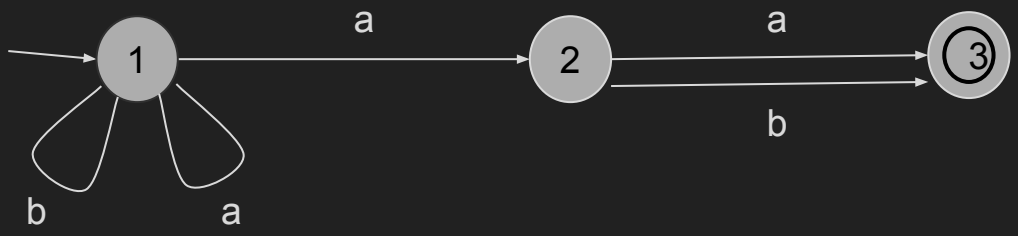
Example



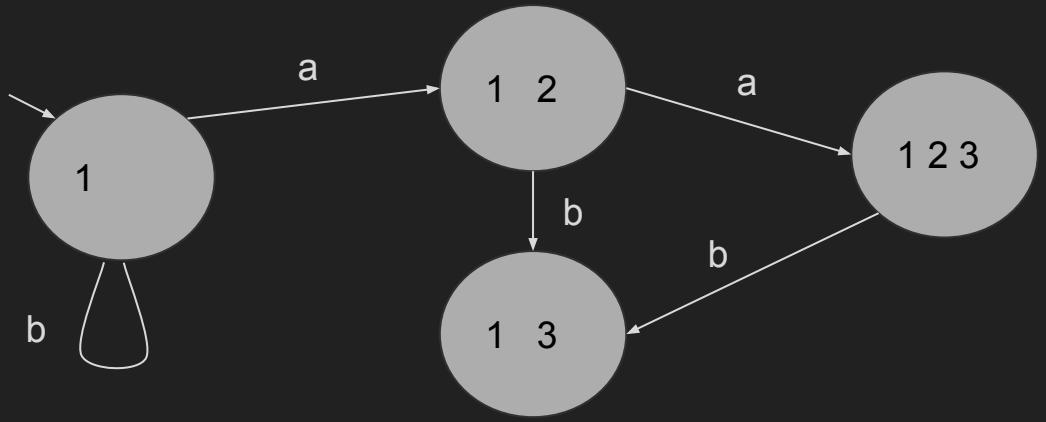
Σ = {a,b}



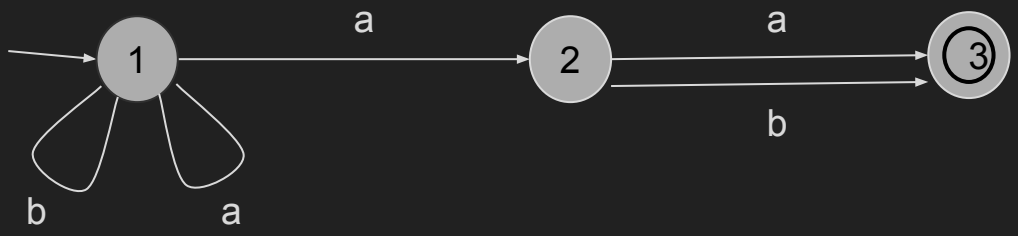
Example



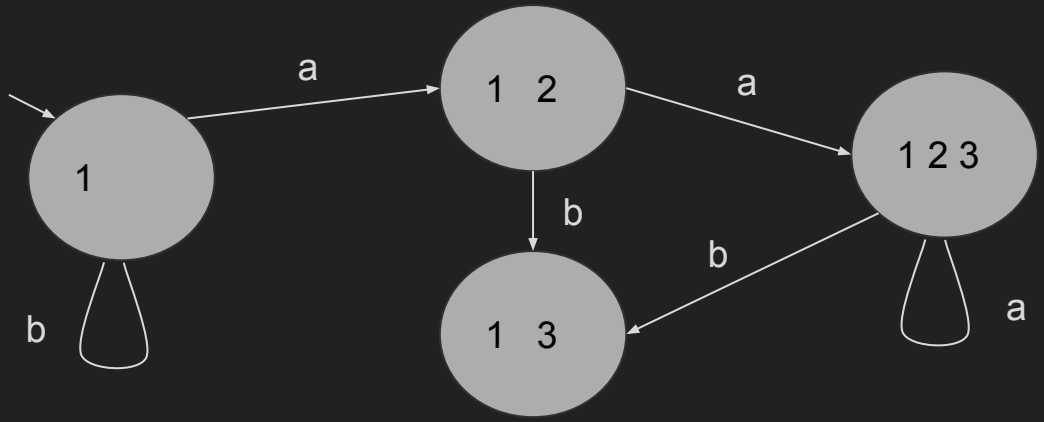
Σ = {a,b}



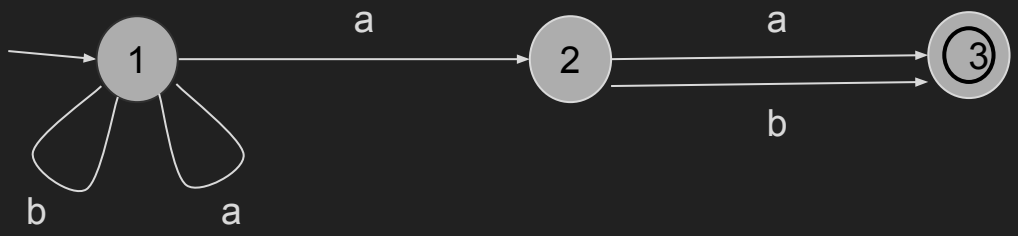
Example



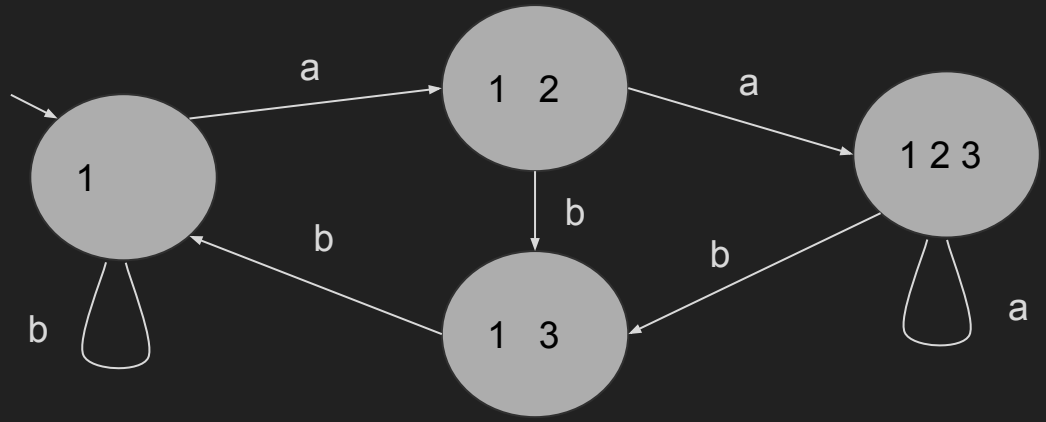
Σ = {a,b}



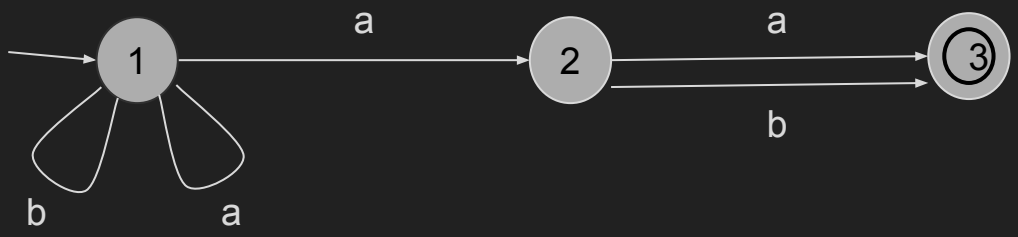
Example



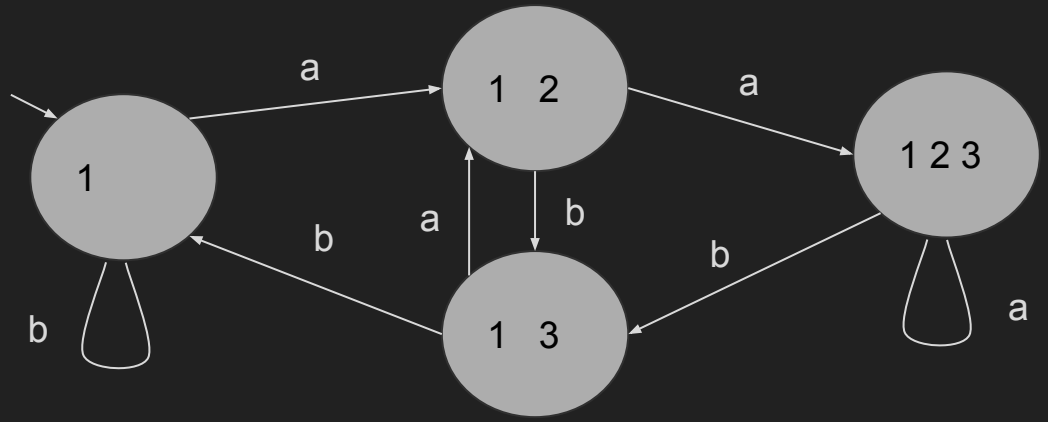
Σ = {a,b}



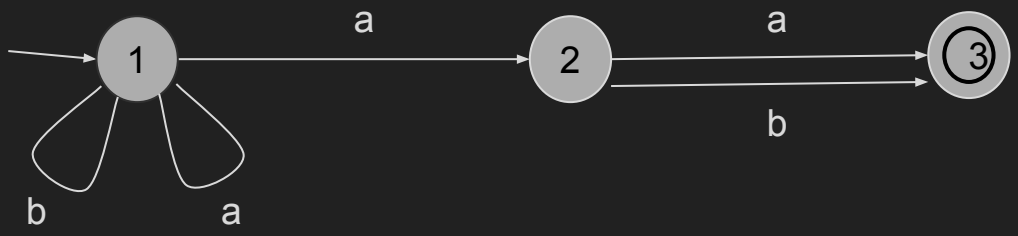
Example



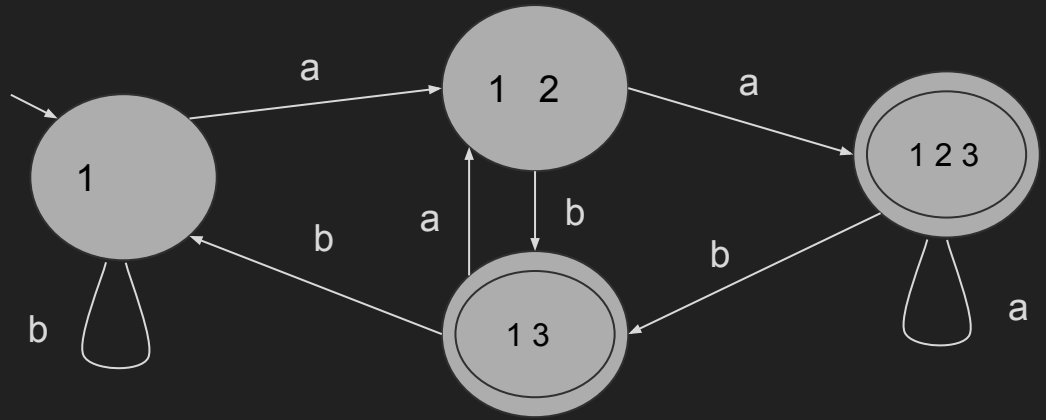
Σ = {a,b}



Example



Σ = {a,b}



Deterministic versus nondeterministic machines

Deterministic finite state machines are easier to implement

- never any choice of which transition to take, so simpler implementation

Nondeterministic finite state machines are easier to manipulate

- when constructing finite state machines out of other finite state machines, don't have to worry about breaking determinism
- the union and concatenation constructions do not preserve determinism

Nondeterministic finite state machines may also have fewer states

Equivalences

A language is regular if:

- there is a regular expression that denotes it
- there is a finite state machine that accepts it
- there is a deterministic finite state machine that accepts it

We can use this to show that if some languages are regular, other languages are also regular

Examples

If A and B are regular, then $A \cup B$, $A \cdot B$, and A* are regular

- pretty much by definition

If A is regular, reverse(A) = { u | the reverse of u is in A } is regular

- construct a finite state machine for A, reverse all the transitions, create a new start state

If A is regular, the complement of A is regular

- construct a deterministic finite state machine for A with exactly one transition per symbol in every state, and flip final and non-final states

Finite state machines = our first computation model!

A finite state machine is an implementation for a function f : $\Sigma^* \rightarrow$ { true, false}

- f(u) = true when the finite state machine accepts u, and = false otherwise

A function f : $\Sigma^* \rightarrow$ { true, false} is FA-computable if it there exists a finite automaton M such that f(s) = true exactly when M accepts s

Theorem: *f : $\Sigma^* \rightarrow$ { true, false} is FA-computable if and only if its characteristic language A_f is regular*

There are many functions that are not FA-computable but intuitively should still be computable: f(u) = true if u has the same number of a's and b's