CS420

Introduction to the Theory of Computation

Lecture 14: Deterministic Finite Automata

Tiago Cogumbreiro

Today we will learn...

- Deterministic Finite Automata (DFA)
- Implementing a DFA
- Converting NFAs into DFAs
- Practical applications of DFAs and NFAs

Finite Automata

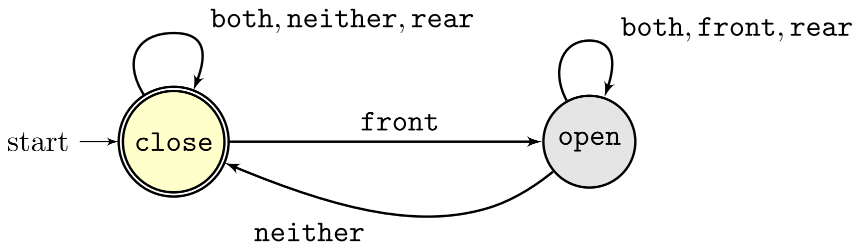
a.k.a. finite state machine

A turnstile controller

> Allows one-directional passage. Opens when the front sensor is triggered. It should remain open while any sensor is triggered, and then close once neither is triggered.

- **States:** open, close
- **Inputs:** front, rear, both, neither

State Diagram



Each state must have exactly one transition per element of the alphabet (all states must have same transition count)

Definition

- Graph-based diagram
- **Nodes:** called states; annotated with a name
  (Distinct names!)
- **Edges:** called transitions; annotated with inputs
- Initial state has an incoming edge (only one)
- Accepted nodes have a double circle (zero or more)
- Multiple inputs are comma separated

In the example: Two states: `open`, `close`. State `close` is an ***accepting*** state. State `close` is also the ***initial*** state

The controller of a turnstile

State transition

(prev. state)	*front*	*rear*	*both*	*neither*
close	**open**	close	close	close
open	open	open	open	**close**

```
from enum import *

class State(Enum): Open = 0; Close = 1

class Input(Enum): Neither = 0; Front = 1; Rear = 2; Both = 3

def state_transition(old_st, i):
  if old_st == State.Close and i == Input.Front: return State.Open
  if old_st == State.Open and i == Input.Neither: return State.Close
  return old_st
```

An automaton

An automaton receives a sequence of inputs, processes them, and outputs whether it accepts the sequence.

- ***Input:*** a string of inputs, and an initial state
- ***Output:*** `accept` or `reject`

Implementation example

```
def automaton_accepts(inputs):
  st = State.Close
  for i in inputs:
    st = state_transition(st, i)
  return st is State.Close
```

An automaton acceptance examples

```
>>> automaton_accepts([])
True
>>> automaton_accepts([Input.Front, Input.Neither])
True
>>> automaton_accepts([Input.Rear, Input.Front, Input.Front])
False
>>> automaton_accepts([Input.Rear, Input.Front, Input.Rear, Input.Neither, Input.Rear])
True
```

Formal definition of a Finite Automaton

Definition 1.5

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set called states
2. Σ is a finite set called alphabet
3. $\delta\colon Q \times \Sigma \to Q$ is the transition function
   (δ takes a state and an alphabet and produces a state)
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accepted states

> A formal definition is a precise mathematical language. In this example, item declares a name and possibly some constraint, e.g., $q_0 \in Q$ is saying that q_0 **must** be in set Q. These constraints are visible in the code in the form of assertions.

Formal declaration of our running example

Let the running example be the following finite automaton $M_{turnstile}$

$$(\{\texttt{Open}, \texttt{Close}\}, \{\texttt{Neither}, \texttt{Front}, \texttt{Rear}, \texttt{Both}\}, \delta, \texttt{Close}, \{\texttt{Close}\})$$

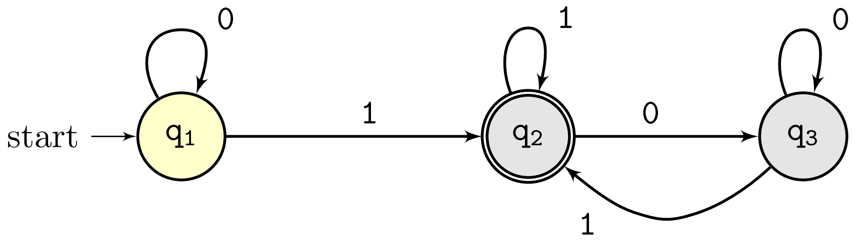
where

$$\delta(\texttt{Close}, \texttt{Front}) = \texttt{Open}$$
$$\delta(\texttt{Open}, \texttt{Neither}) = \texttt{Close}$$
$$\delta(q, i) = q$$

Facts

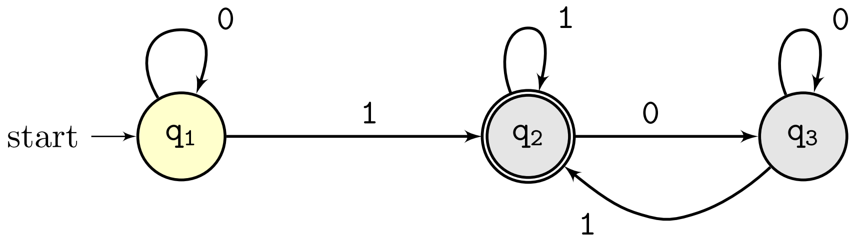
- $M_{turnstile}$ accepts [Front, Neither]
- $M_{turnstile}$ rejects [Rear, Front, Front]
- $M_{turnstile}$ accepts [Rear, Front, Rear, Neither, Rear]

Example



States?

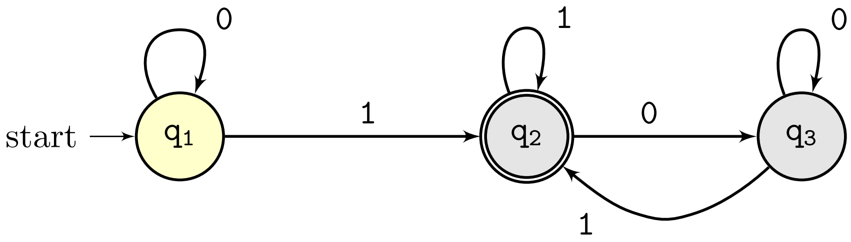
Example



States? $Q = \{q_1, q_2, q_3\}$

Alphabet?

Example

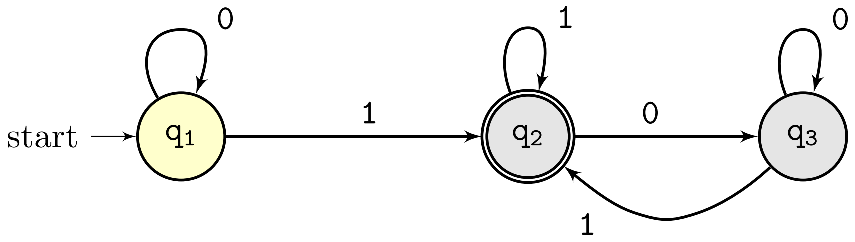


States? $Q = \{q_1, q_2, q_3\}$

Alphabet? $\Sigma = \{0, 1\}$

Transition table δ?

Example



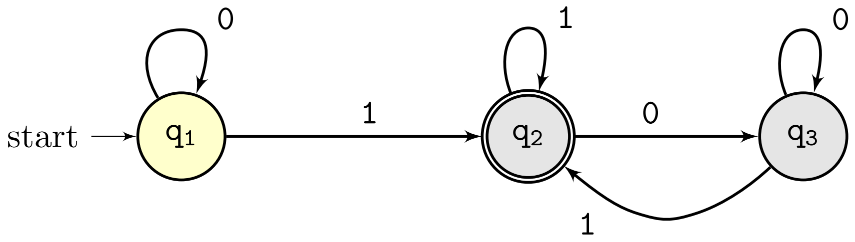
States? $Q = \{q_1, q_2, q_3\}$

Alphabet? $\Sigma = \{0, 1\}$

Transition table δ?

(prev)	***0***	***1***
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_3	q_2

Example



States? $Q = \{q_1, q_2, q_3\}$

Alphabet? $\Sigma = \{0, 1\}$

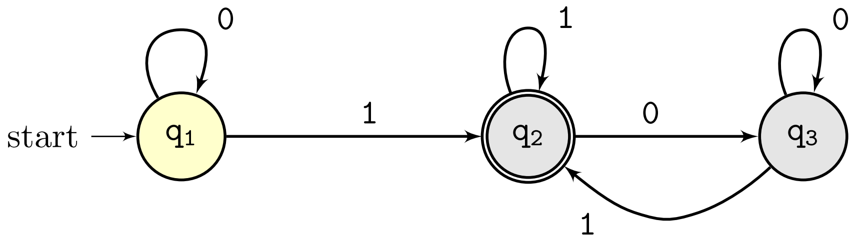
Transition table δ?

(prev)	*0*	*1*
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_3	q_2

Finite Automaton:

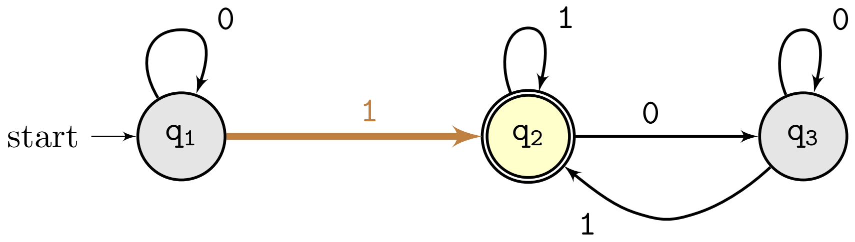
$$(\{q_1, q_2, q_3\}, \{0, 1\}, q_1, \{q_2\})$$

Example



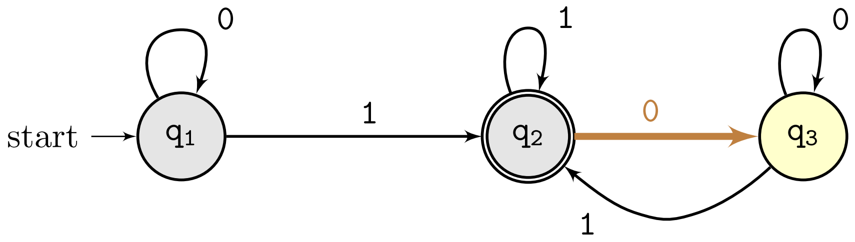
[1, 0, 1, 1]

Example



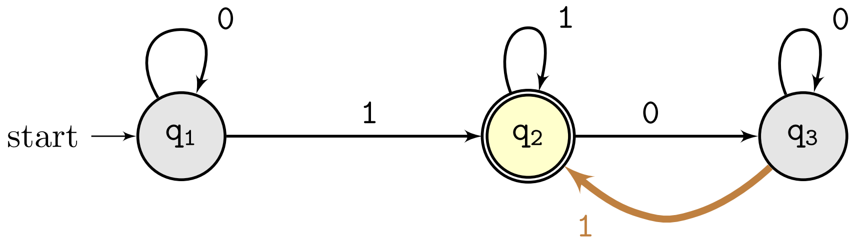
[**1**, 0, 1, 1]

Example



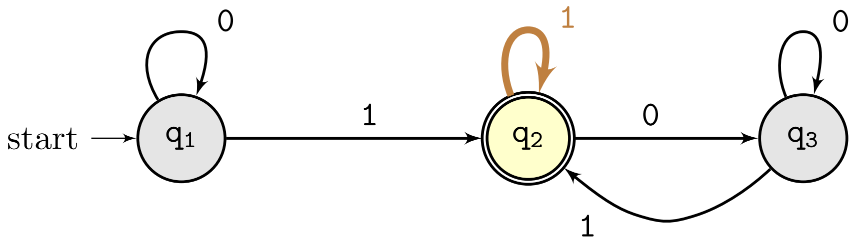
[1, **0**, 1, 1]

Example



[1, 0, **1**, 1]

Example



[1, 0, 1, **1**]

What are the set of inputs accepted by this automaton?

What are the set of inputs accepted by this automaton?

Answer: Strings terminating in 1

The language of a machine

Definition: language of a machine

1. We define $\mathrm{L}(M)$ to be the set of all strings accepted by finite automaton M.
2. Let $A = \mathrm{L}(M)$, we say that the finite automaton M **recognizes** the set of strings A.

The language of a machine

Definition: language of a machine

1. We define $\mathrm{L}(M)$ to be the set of all strings accepted by finite automaton M.
2. Let $A = \mathrm{L}(M)$, we say that the finite automaton M **recognizes** the set of strings A.

Notes

- The language is the ***set*** of all possible alphabet-sequences recognized by a finite automaton
- Since $\mathrm{L}(M)$ is a **total** function, then the language recognized by a machine always exists and is unique
- A language may be empty
- We **cannot** write a program that returns the language of an arbitrary finite automaton. Why? **Because the language set may be infinite. How could a program return $\Sigma^\star$?**

Are all DFAs also NFAs?

Are all DFAs also NFAs?

- **Yes,** DFAs can be trivially converted into NFAs.
  The state diagram of a DFA is equivalent to the same state diagram as an NFA.
- We only need to slightly change the transition function to handle ϵ inputs.

Are all NFAs also DFAs?

Are all NFAs also DFAs?

Yes!

Theorem 1.39

Every NFA has an equivalent DFA

- We study the algorithm that converts an NFA into a DFA
- **Tip:** understanding the implementation of the acceptance algorithm, helps understanding the conversion and vice-versa

Intuition

- **States:**

Theorem 1.39

Every NFA has an equivalent DFA

- We study the algorithm that converts an NFA into a DFA
- **Tip:** understanding the implementation of the acceptance algorithm, helps understanding the conversion and vice-versa

Intuition

- **States:** Each state becomes a set of all possible concurrent states of the NFA
- **Alphabet:**

Theorem 1.39

Every NFA has an equivalent DFA

- We study the algorithm that converts an NFA into a DFA
- **Tip:** understanding the implementation of the acceptance algorithm, helps understanding the conversion and vice-versa

Intuition

- **States:** Each state becomes a set of all possible concurrent states of the NFA
- **Alphabet:** same alphabet
- **Initial state:**

Theorem 1.39

Every NFA has an equivalent DFA

- We study the algorithm that converts an NFA into a DFA
- **Tip:** understanding the implementation of the acceptance algorithm, helps understanding the conversion and vice-versa

Intuition

- **States:** Each state becomes a set of all possible concurrent states of the NFA
- **Alphabet:** same alphabet
- **Initial state:** The state that consists of an epsilon-step on the initial state.
- **Transition:**

Theorem 1.39

Every NFA has an equivalent DFA

- We study the algorithm that converts an NFA into a DFA
- **Tip:** understanding the implementation of the acceptance algorithm, helps understanding the conversion and vice-versa

Intuition

- **States:** Each state becomes a set of all possible concurrent states of the NFA
- **Alphabet:** same alphabet
- **Initial state:** The state that consists of an epsilon-step on the initial state.
- **Transition:** One input-step followed by one epsilon-step

Are all NFAs also DFAs?

```
def nfa_to_dfa(nfa):
  def transition(q, c):
    return nfa.epsilon(nfa.multi_transition(q, c))

  def accept_state(qs):
    for q in qs:
      if nfa.accepted_states(q):
        return True
    return False

  return DFA(
    nfa.alphabet,
    transition,
    nfa.epsilon({nfa.start_state}),
    accept_state)
```

Nondeterministic transition $\delta_\cup$

$$\delta_\cup(R, a) = \bigcup_{q \in R} \delta(r, a)$$

```
def multi_transition(self, states, input):
  new_states = set()
  for st in states:
    new_states.update(self.transition_func(st, input))
  return set(new_states)
```

(See Theorem 1.39; in the book $\delta_\cup$ is δ')

Epsilon transition

$$E(R) = \{q \mid q \text{ can be reached from R by travelling along 0 or more } \epsilon \text{ arrows}\}$$

```
def epsilon(self, states):
  states = set(states)
  while True:
    count = len(states)
    states.update(self.transition(states, None))
    if count == len(states):
      return states
```

(See Theorem 1.39)

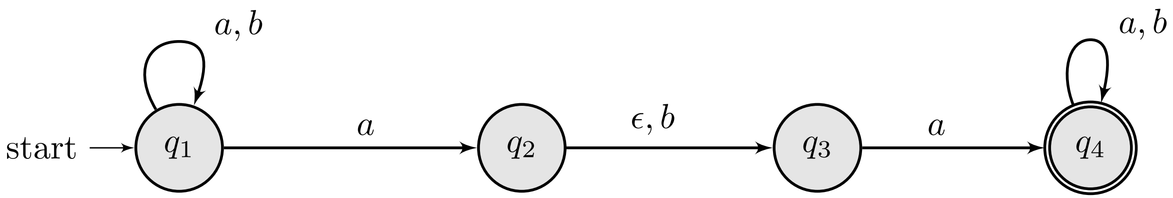
Theorem 1.39

Every NFA has an equivalent DFA

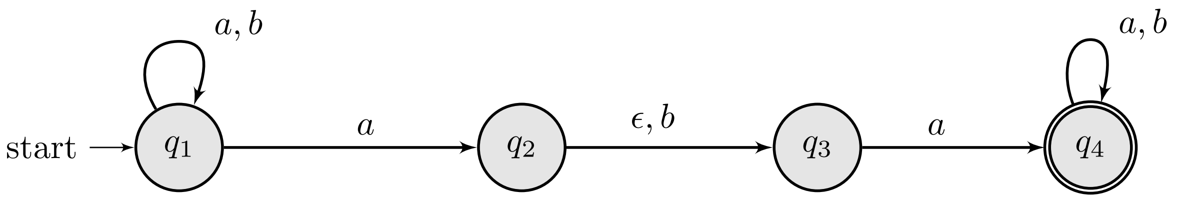
Formally, we introduce function $\mathrm{nfa2dfa}$ that converts an NFA into a DFA.

$\mathrm{nfa2dfa}((Q, \Gamma, \delta, q_1, F)) = (\mathcal{P}(Q), \Gamma, \delta_D, E(q_1), F_D)$ where

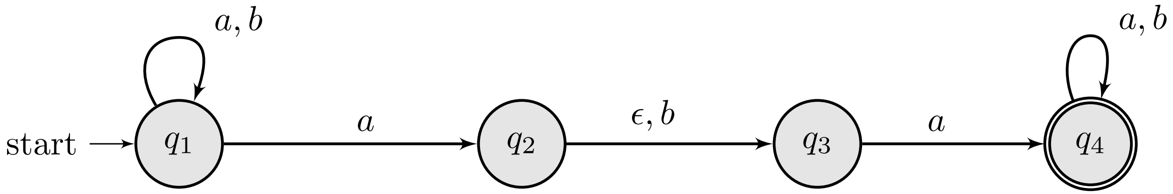
- $\delta_D(Q, c) = E(\delta_\cup(Q, c))$
- $F_D = \{Q \mid Q \cap F \neq \emptyset\}$



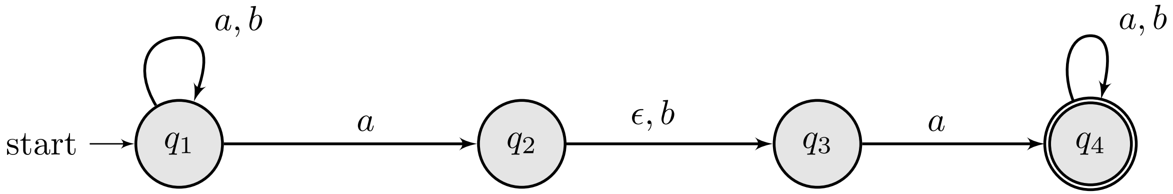
States	***Input***	***States***	***Done***
$\{q_1\}$	a	$\{q_1, q_2, q_3\}$	
$\{q_1\}$	b	$\{q_1\}$	x
$\{q_1, q_2, q_3\}$	a		
$\{q_1, q_2, q_3\}$	b		



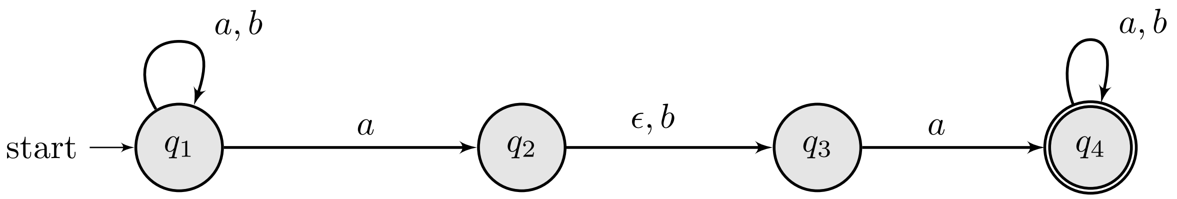
States	Input	States	Done
$\{q_1\}$	a	$\{q_1, q_2, q_3\}$	x
$\{q_1\}$	b	$\{q_1\}$	x
$\{q_1, q_2, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	
$\{q_1, q_2, q_3\}$	b	$\{q_1, q_3\}$	
$\{q_1, q_2, q_3, q_4\}$	a		
$\{q_1, q_2, q_3, q_4\}$	b		
$\{q_1, q_3\}$	a		
$\{q_1, q_3\}$	b		



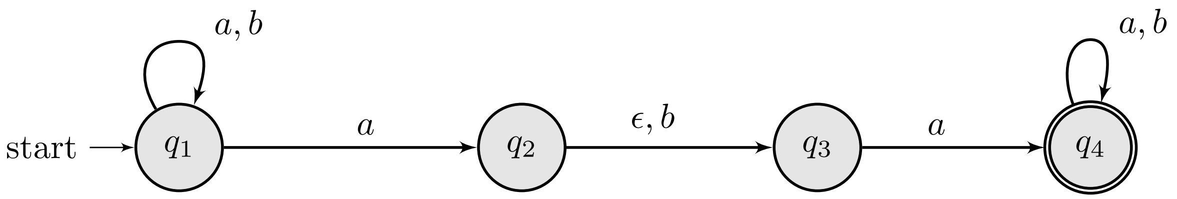
States	Input	States	Done
$\{q_1\}$	a	$\{q_1, q_2, q_3\}$	x
$\{q_1\}$	b	$\{q_1\}$	x
$\{q_1, q_2, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3\}$	b	$\{q_1, q_3\}$	
$\{q_1, q_2, q_3, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3, q_4\}$	b	$\{q_1, q_3, q_4\}$	
$\{q_1, q_3\}$	a		
$\{q_1, q_3\}$	b		
$\{q_1, q_3, q_4\}$	a		
$\{q_1, q_3, q_4\}$	b		



States	Input	States	Done
$\{q_1\}$	a	$\{q_1, q_2, q_3\}$	x
$\{q_1\}$	b	$\{q_1\}$	x
$\{q_1, q_2, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3\}$	b	$\{q_1, q_3\}$	x
$\{q_1, q_2, q_3, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3, q_4\}$	b	$\{q_1, q_3, q_4\}$	x
$\{q_1, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_3\}$	b	$\{q_1\}$	x
$\{q_1, q_3, q_4\}$	a		
$\{q_1, q_3, q_4\}$	b		

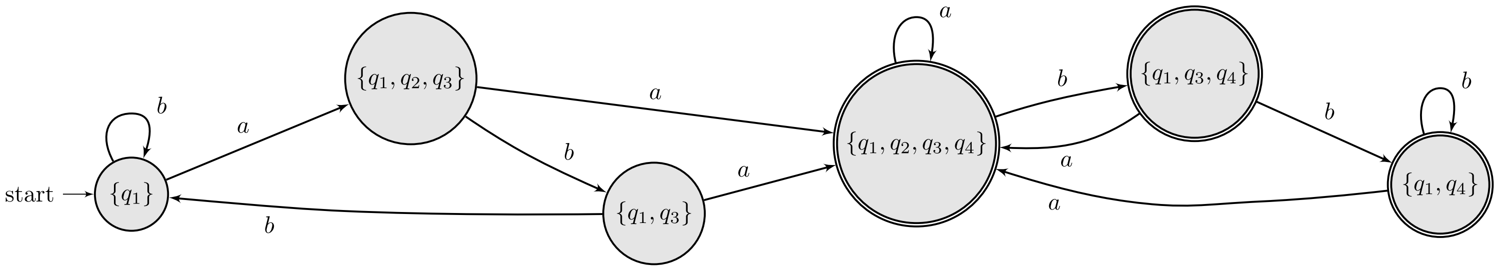


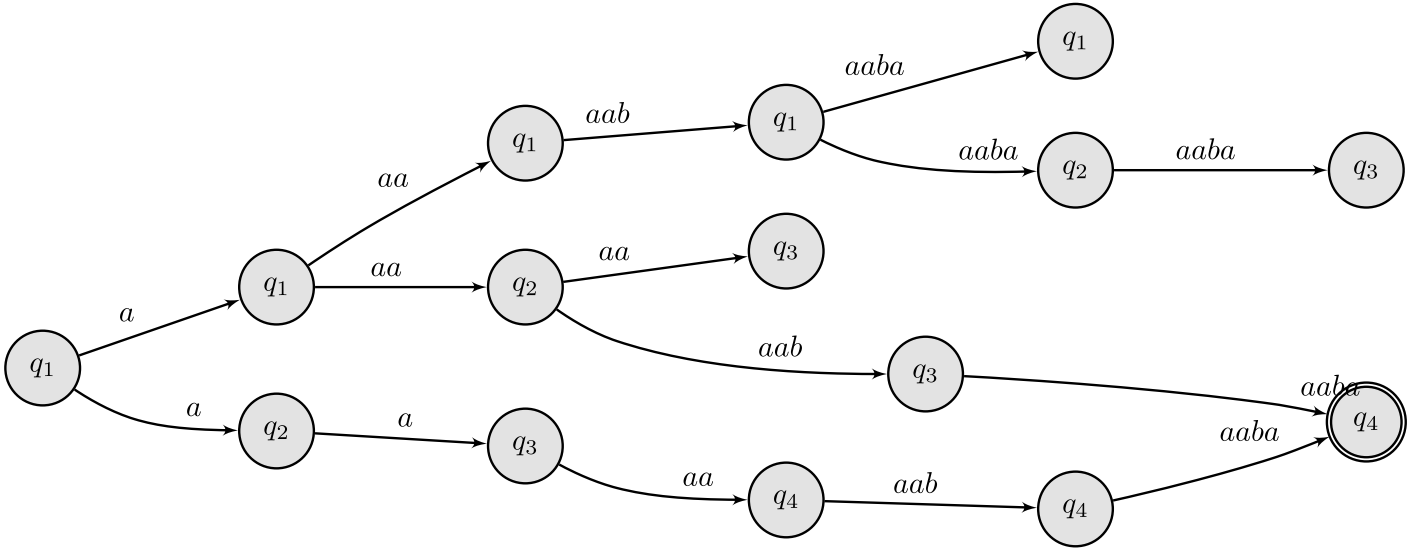
States	***Input***	***States***	***Done***
$\{q_1\}$	a	$\{q_1, q_2, q_3\}$	x
$\{q_1\}$	b	$\{q_1\}$	x
$\{q_1, q_2, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3\}$	b	$\{q_1, q_3\}$	x
$\{q_1, q_2, q_3, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3, q_4\}$	b	$\{q_1, q_3, q_4\}$	x
$\{q_1, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_3\}$	b	$\{q_1\}$	x
$\{q_1, q_3, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_3, q_4\}$	b	$\{q_1, q_4\}$	
$\{q_1, q_4\}$	a		
$\{q_1, q_4\}$	b		



States	Input	States	Done
$\{q_1\}$	a	$\{q_1, q_2, q_3\}$	x
$\{q_1\}$	b	$\{q_1\}$	x
$\{q_1, q_2, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3\}$	b	$\{q_1, q_3\}$	x
$\{q_1, q_2, q_3, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_2, q_3, q_4\}$	b	$\{q_1, q_3, q_4\}$	x
$\{q_1, q_3\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_3\}$	b	$\{q_1\}$	x
$\{q_1, q_3, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_3, q_4\}$	b	$\{q_1, q_4\}$	x
$\{q_1, q_4\}$	a	$\{q_1, q_2, q_3, q_4\}$	x
$\{q_1, q_4\}$	b	$\{q_1, q_4\}$	x

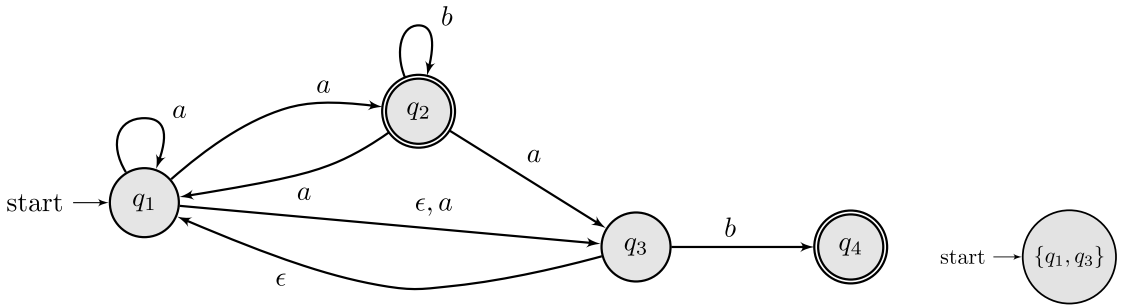
Exercise


start
{q1}
{q1, q2, q3}
{q1, q3}
{q1, q2, q3, q4}
{q1, q3, q4}
{q1, q4}
a
b


q1
q2
q3
q4
a
aa
aab
aaba

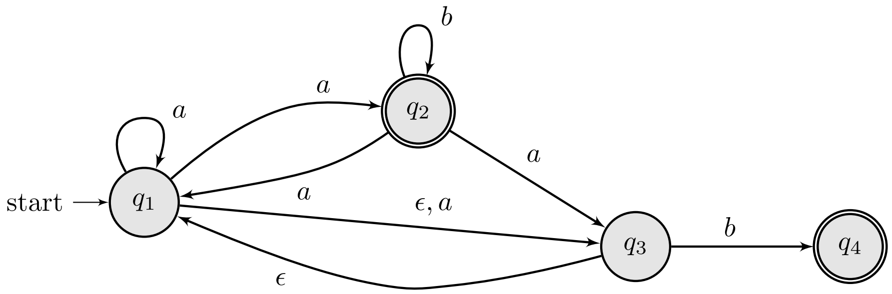
Producing a DFA from an NFA

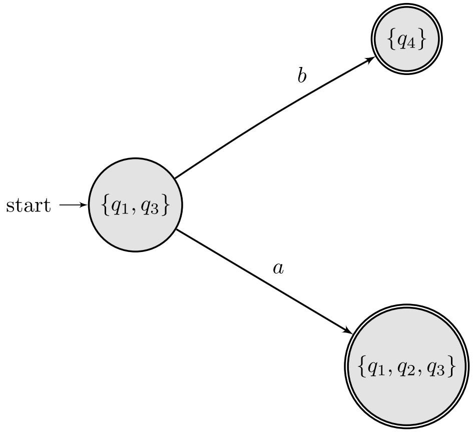
Producing a DFA from an NFA



The initial state is the set of all states in the NFA that are reachable from q_1 via ϵ transitions plus q_1.

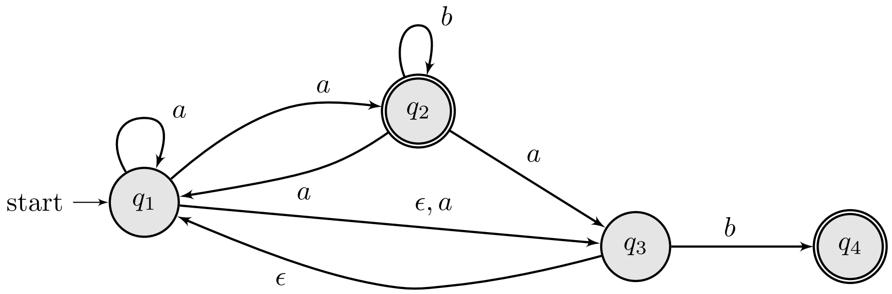
Producing a DFA from an NFA

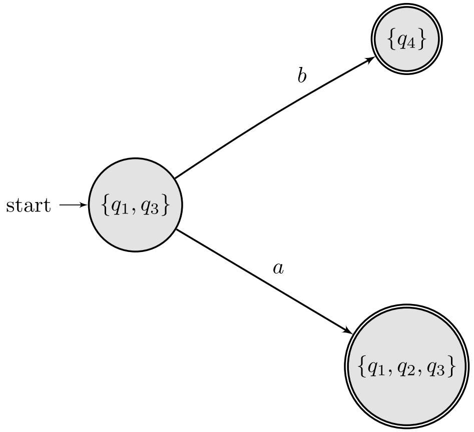




- For each input in Σ range we must draw a transition to a target state.
- A target state is found by taking an input, say a, and doing an input+epsilon step on each sub-state.

Producing a DFA from an NFA

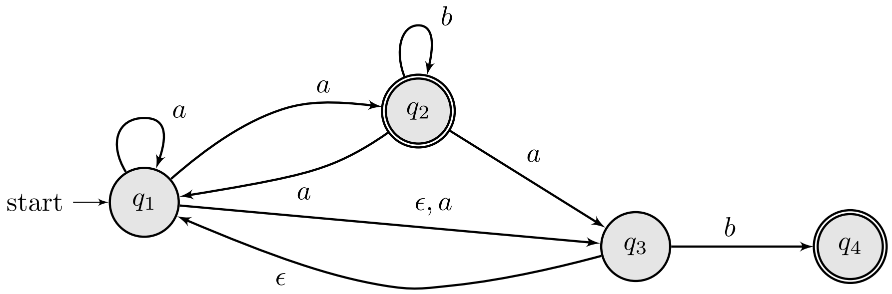


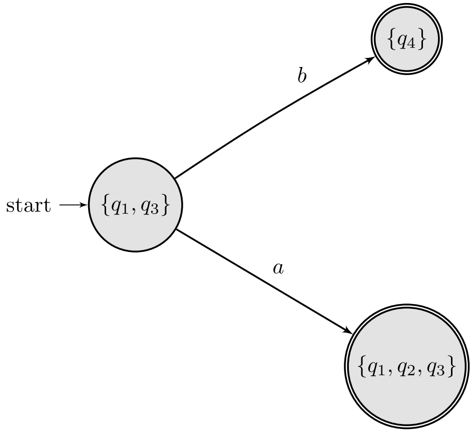


First, input a we find all reachable states (via input+epsilon state) that start from either q_1 or q_3.

- From q_1 via a we get $\{q_1, q_2, q_3\}$
- From q_3 via a we get $\emptyset$
- Result state is $\{q_1, q_2, q_3\} \cup \emptyset = \{q_1, q_2, q_3\}$

Producing a DFA from an NFA

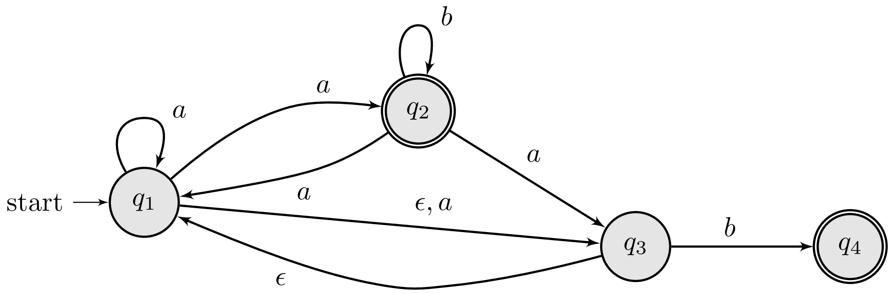


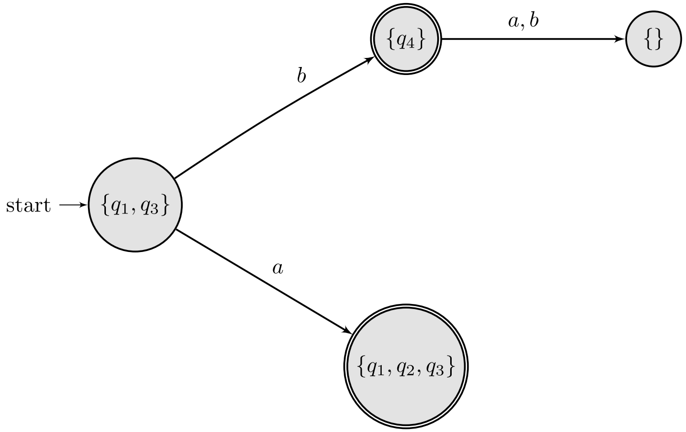


Second, input b we find all reachable states (via input+epsilon state) that start from either q_1 or q_3.

- From q_1 via b we get $\emptyset$
- From q_3 via b we get $\{q_4\}$
- Result state is $\emptyset \cup \{q_4\} = \{q_4\}$

Producing a DFA from an NFA

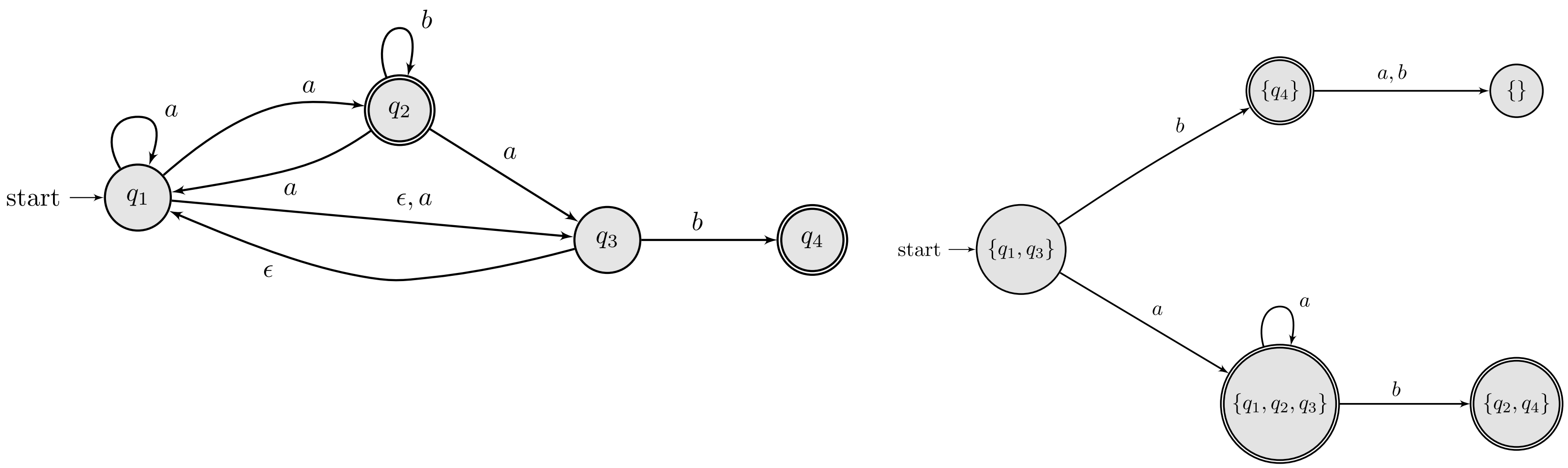




For inputs a and b we find all reachable states (via input+epsilon state) that start from q_4:

- From q_4 via a we get $\emptyset$, so the result state is $\emptyset$
- From q_4 via b we get $\emptyset$, so the result state is $\emptyset$

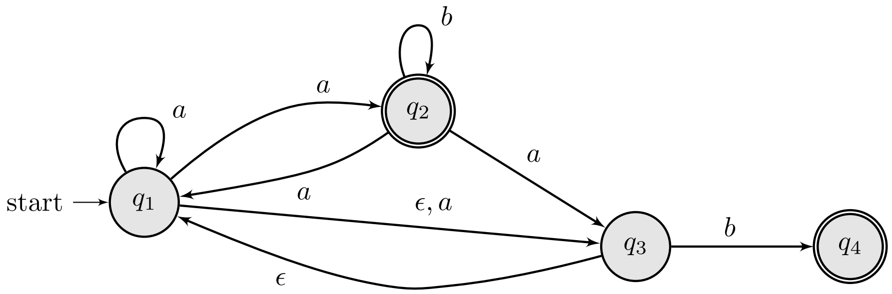
Producing a DFA from an NFA

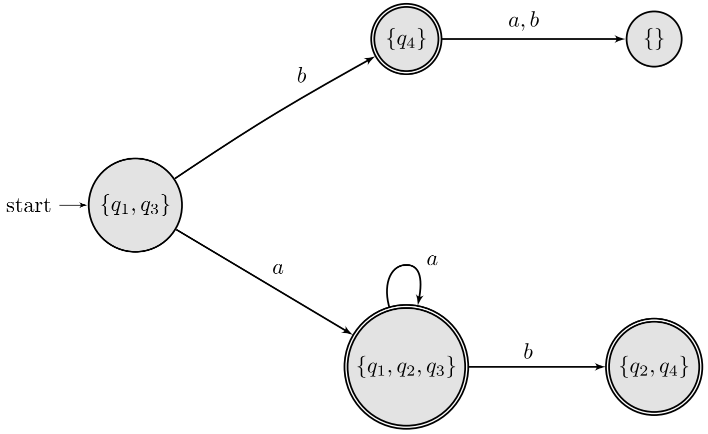


Transition from $\{q_1, q_2, q_3\}$ via a?

- We know with $\{q_1, q_3\}$ with a we reach $\{q_1, q_2, q_3\}$
- From q_2 with a we reach $\{q_3\}$
- Thus, result state is $\{q_1, q_2, q_3\} \cup \{q_3\} = \{q_1, q_2, q_3\}$ (self-loop)

Producing a DFA from an NFA

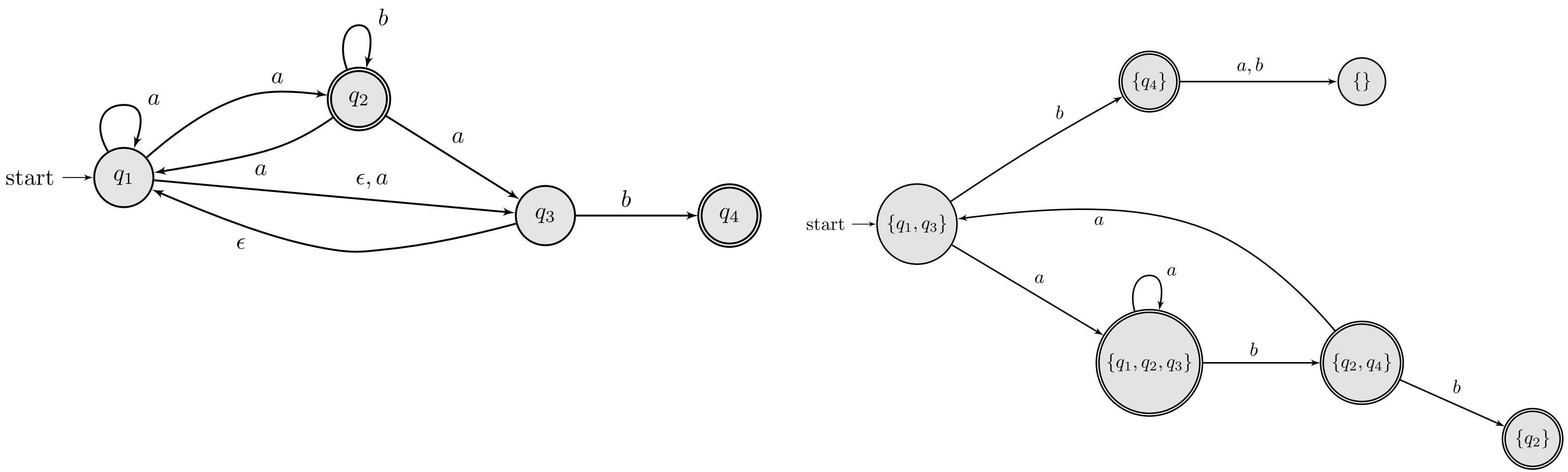




Transition from $\{q_1, q_2, q_3\}$ via b?

- We know with $\{q_1, q_3\}$ with b we reach $\{q_4\}$
- From q_2 with b we reach $\{q_2\}$
- Thus, result state is $\{q_4\} \cup \{q_2\} = \{q_2, q_4\}$

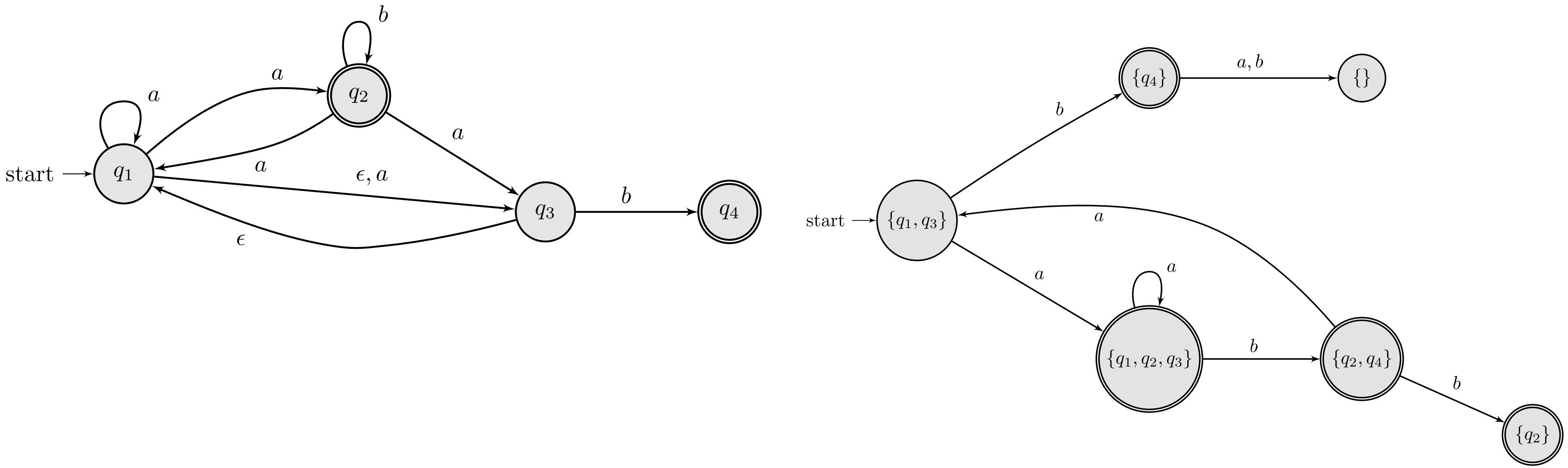
Producing a DFA from an NFA



Transition from $\{q_2, q_4\}$ via a?

- From q_2 with a we reach $\{q_1, q_3\}$
- From q_4 with a we reach $\emptyset$
- Thus, result state is $\{q_1, q_3\} \cup \emptyset = \{q_1, q_3\}$

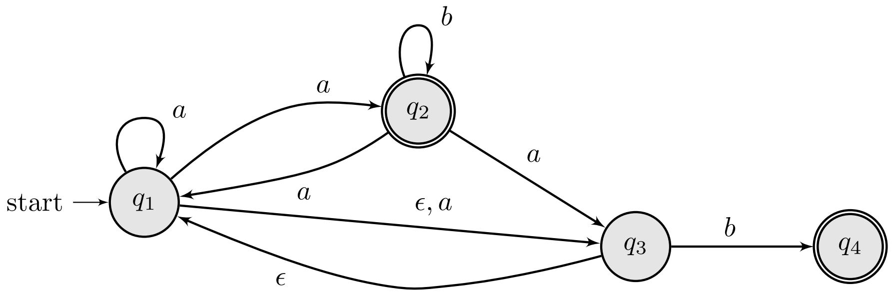
Producing a DFA from an NFA

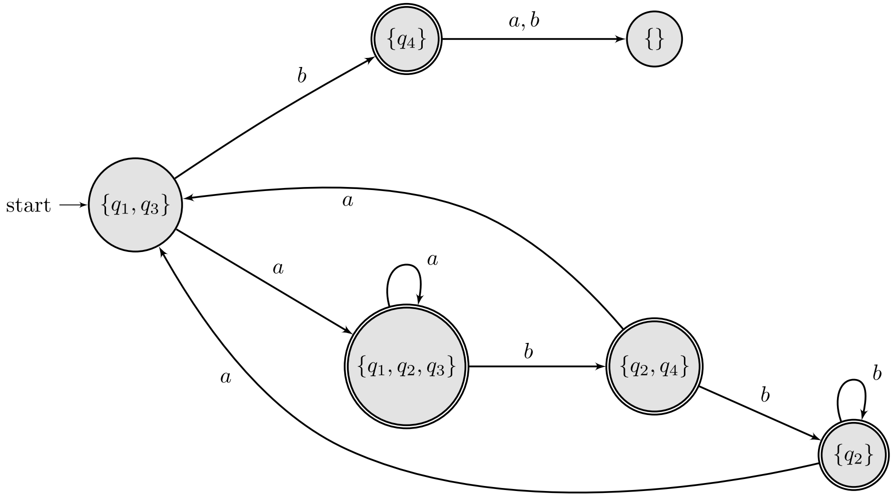


Transition from $\{q_2, q_4\}$ via b?

- From q_2 with b we reach $\{q_2\}$
- From q_4 with b we reach $\emptyset$
- Thus, result state is $\{q_2\} \cup \emptyset = \{q_2\}$

Producing a DFA from an NFA

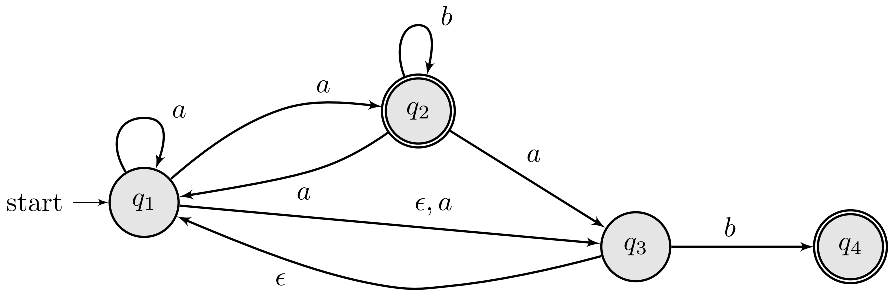


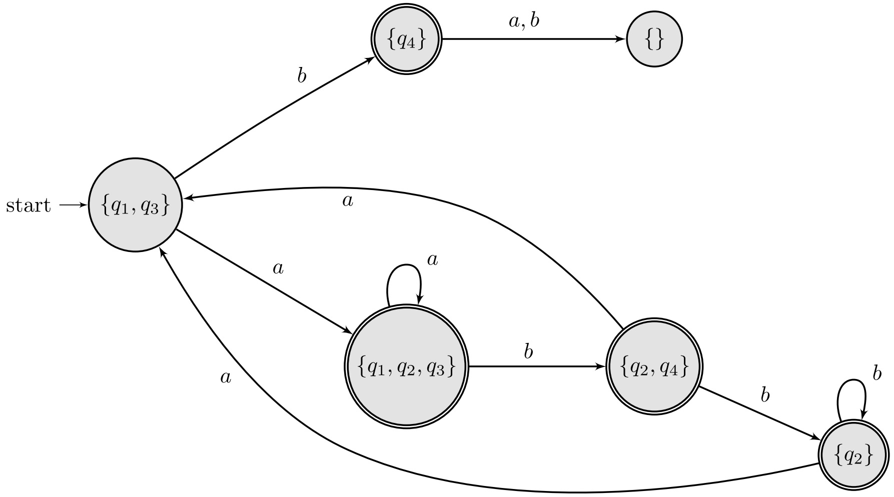


Transition from $\{q_2\}$ via a?

- From q_2 with a we reach $\{q_1, q_3\}$ (result state)

Producing a DFA from an NFA

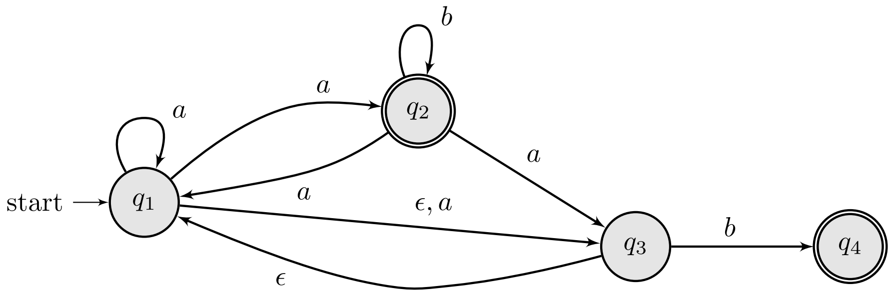


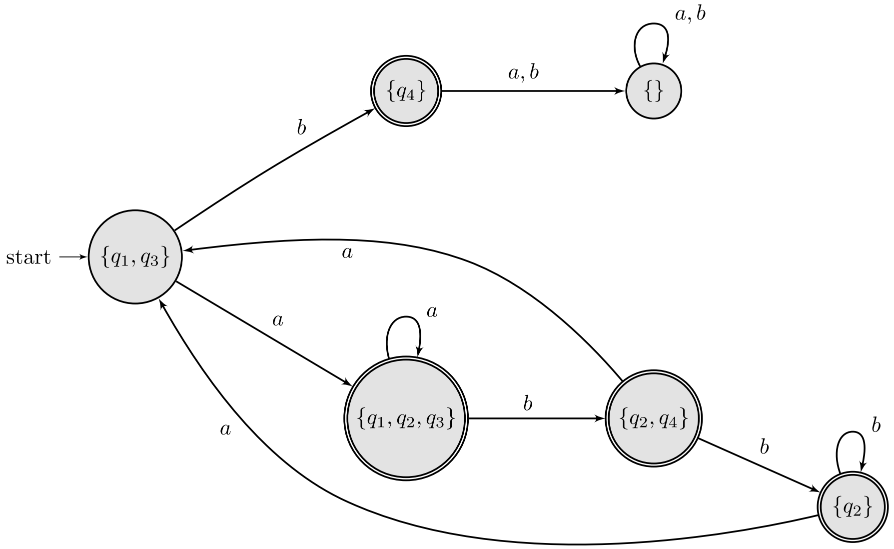


Transition from $\{q_2\}$ via b?

- From q_2 with b we reach $\{q_2\}$ (result state; self loop)

Producing a DFA from an NFA





State $\{\}$ (also known as $\emptyset$) is a **sink state**, so we draw a self loop for every input in Σ.

Applications of automaton

- DFAs are crucial to implement regular expression matching by converting
  `REGEX → NFA → DFA`
- DFAs are simple to implement and fast to run
- DFAs can be **minimized**
  Any regular language has a minimal DFA, which is defined as a DFA with the smallest number of states that recognizes that language.

Use Case 1: implementing regex

- Rust standard library's regular expresson implementation (source)

```
struct ExecReadOnly {
    /// The original regular expressions given by the caller to compile.
    res: Vec<String>,
    /// A compiled program that is used in the NFA simulation and backtracking.
    /// It can be byte-based or Unicode codepoint based.
    ///
    /// N.B. It is not possibly to make this byte-based from the public API.
    /// It is only used for testing byte based programs in the NFA simulations.
    nfa: Program,
    /// A compiled byte based program for DFA execution. This is only used
    /// if a DFA can be executed. (Currently, only word boundary assertions are
    /// not supported.) Note that this program contains an embedded .*?
    /// preceding the first capture group, unless the regex is anchored at the
    /// beginning.
    dfa: Program,
```

Use Case 2: DFA/NFA

Using a DFA/NFA to structure hardware usage

Use Case 2: DFA/NFA

Using a DFA/NFA to structure hardware usage

- Arduino is an open-source hardware to design **microcontrollers**
- Programming can be difficult, because it is highly concurrent
- Finite-state-machines structures the logical states of the hardware
- **Input:** a string of hardware events
- String acceptance is not interesting in this domain

Example

> The FSM represents the logical view of a micro-controller with a light switch

Use Case 2

Declare states

```
#include "Fsm.h"
// Connect functions to a state
State state_light_on(on_light_on_enter, NULL, &on_light_on_exit);
// Connect functions to a state
State state_light_off(on_light_off_enter, NULL, &on_light_off_exit);
// Initial state
Fsm fsm(&state_light_off);
```

Source: platformio.org/lib/show/664/arduino-fsm

Use Case 2

Declare transitions

```
// standard arduino functions
void setup() {
  Serial.begin(9600);

  fsm.add_transition(&state_light_on, &state_light_off,
                     FLIP_LIGHT_SWITCH,
                     &on_trans_light_on_light_off);
  fsm.add_transition(&state_light_off, &state_light_on,
                     FLIP_LIGHT_SWITCH,
                     &on_trans_light_off_light_on);
}
```

Source: platformio.org/lib/show/664/arduino-fsm

Use Case 2

Code that runs on before/after states

```
// Transition callback functions
void on_light_on_enter() {
  Serial.println("Entering LIGHT_ON");
}

void on_light_on_exit() {
  Serial.println("Exiting LIGHT_ON");
}

void on_light_off_enter() {
  Serial.println("Entering LIGHT_OFF");
}
// ...
```

Source: platformio.org/lib/show/664/arduino-fsm