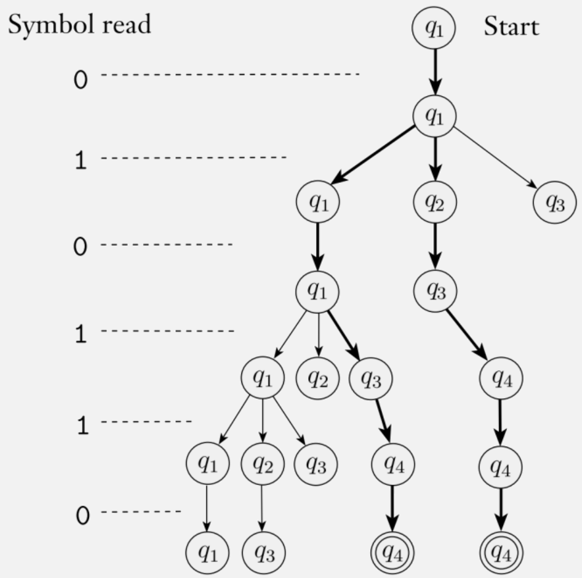
CS 622

Computing with NFAs

Wednesday, February 21, 2024

UMass Boston CS



Announcements

- HW 2 in
  - ~~Due Wed 2/21 12pm EST (noon)~~
- HW 3 out
  - Due Mon 3/4 12pm EST (noon)

HW 1 Observations

- Problems must be assigned to the correct pages
- Proof format must be a **Statements** and **Justifications** table
- Machine formal descriptions must have a tuple

How to ask for HW help

(there's **no such thing** as a **stupid question**, but …)

… there **is such thing** as a **less useful question** (gets less useful answers)

- "Is this correct?"
- "I don't get it"
- "Give me a hint?"
- "Do I need to do the thing DFA thing?"

Useful question examples (gets useful answers)**:**

- "I think string xyz and zyx is in language A but I'm not sure? Can you clarify?"
- "I'm don't understand this notation $A \otimes B \ggg C$ … and I couldn't find it in the book"
- "I couldn't this word's definition …"
- "I know I want to change the machine to add an accept state that … but I can't figure out how to write it formally. Hint?"

Previously

Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$

Concatenation of Languages

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$.

If $A = \{\mathtt{fort}, \mathtt{south}\}$ $B = \{\mathtt{point}, \mathtt{boston}\}$

$$A \circ B = \{\mathtt{fortpoint}, \mathtt{fortboston}, \mathtt{southpoint}, \mathtt{southboston}\}$$

Previously

Is Concatenation Closed?

THEOREM

The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

- Cannot? combine A_1 and A_2's machine to make a DFA because:
  - Unclear when to switch? (can only read input once)
- Need a different kind of machine!

Previously

Nondeterministic Finite Automata (NFA)

DEFINITION

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

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta \colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Transition function maps one state and label to a <u>set of states</u>

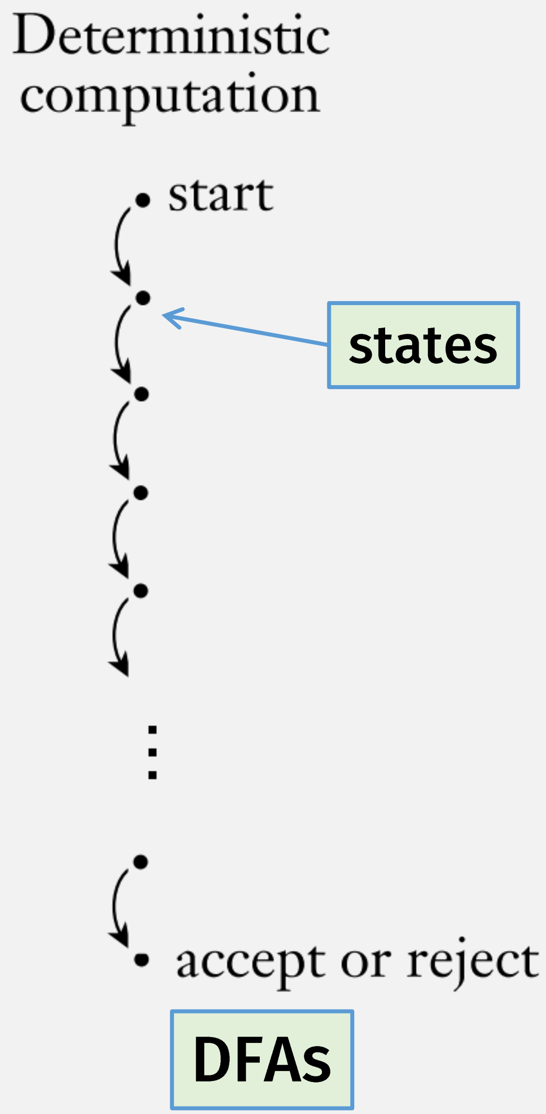
Transition label can be "empty",

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$$

<u>CAREFUL</u>:
ε symbol is <u>reused</u> here, as a **<u>transition label</u>**
(ie, an argument to δ)
- **it's not the empty string!**
- And, **it's** (still) **not a character in alphabet Σ!**

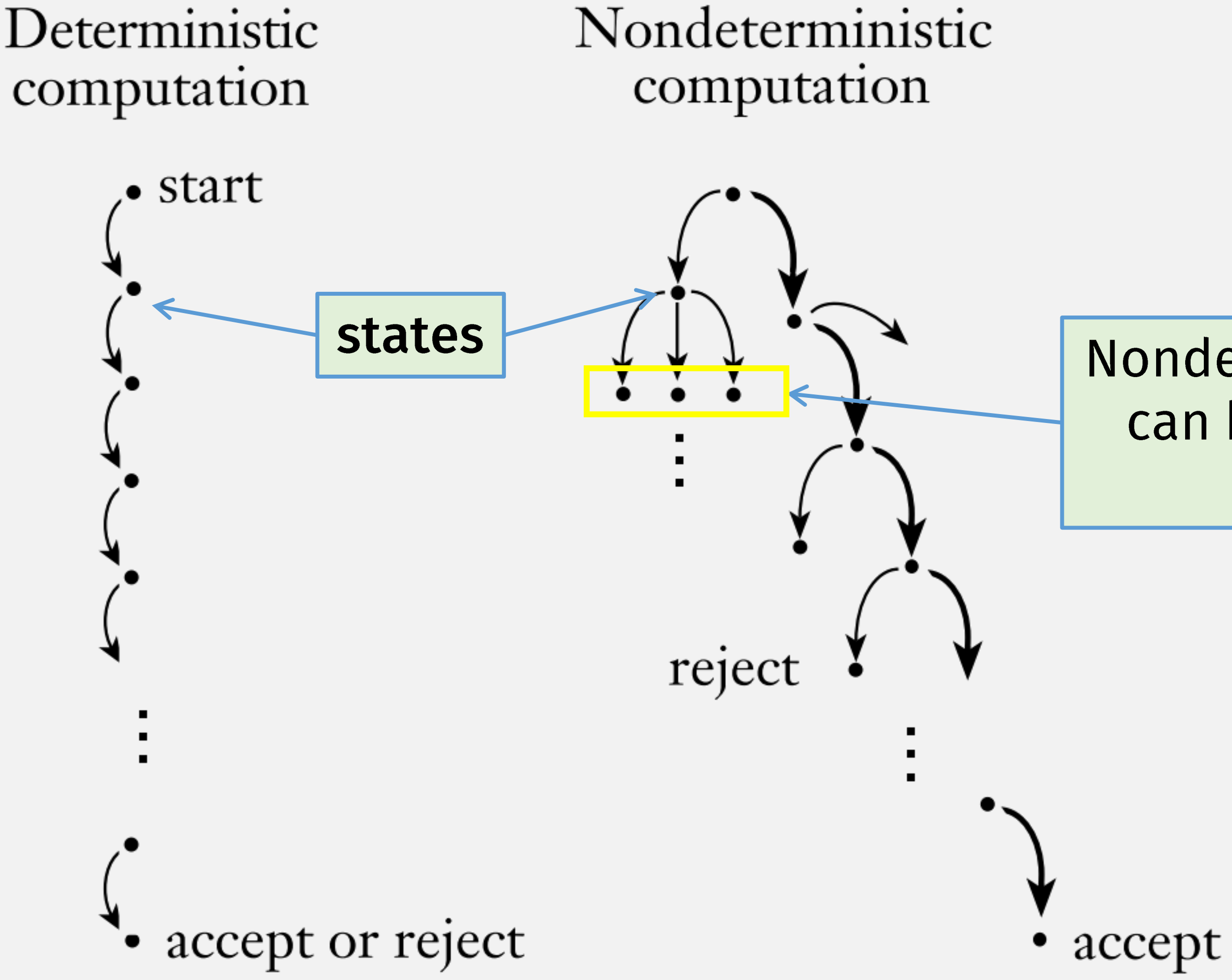
Previously

Deterministic vs Nondeterministic



Previously

Deterministic vs Nondeterministic

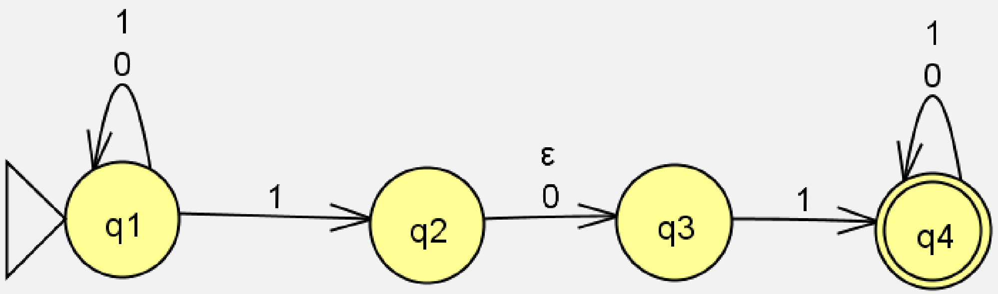


Nondeterministic computation can be in <u>multiple states at the same time</u>

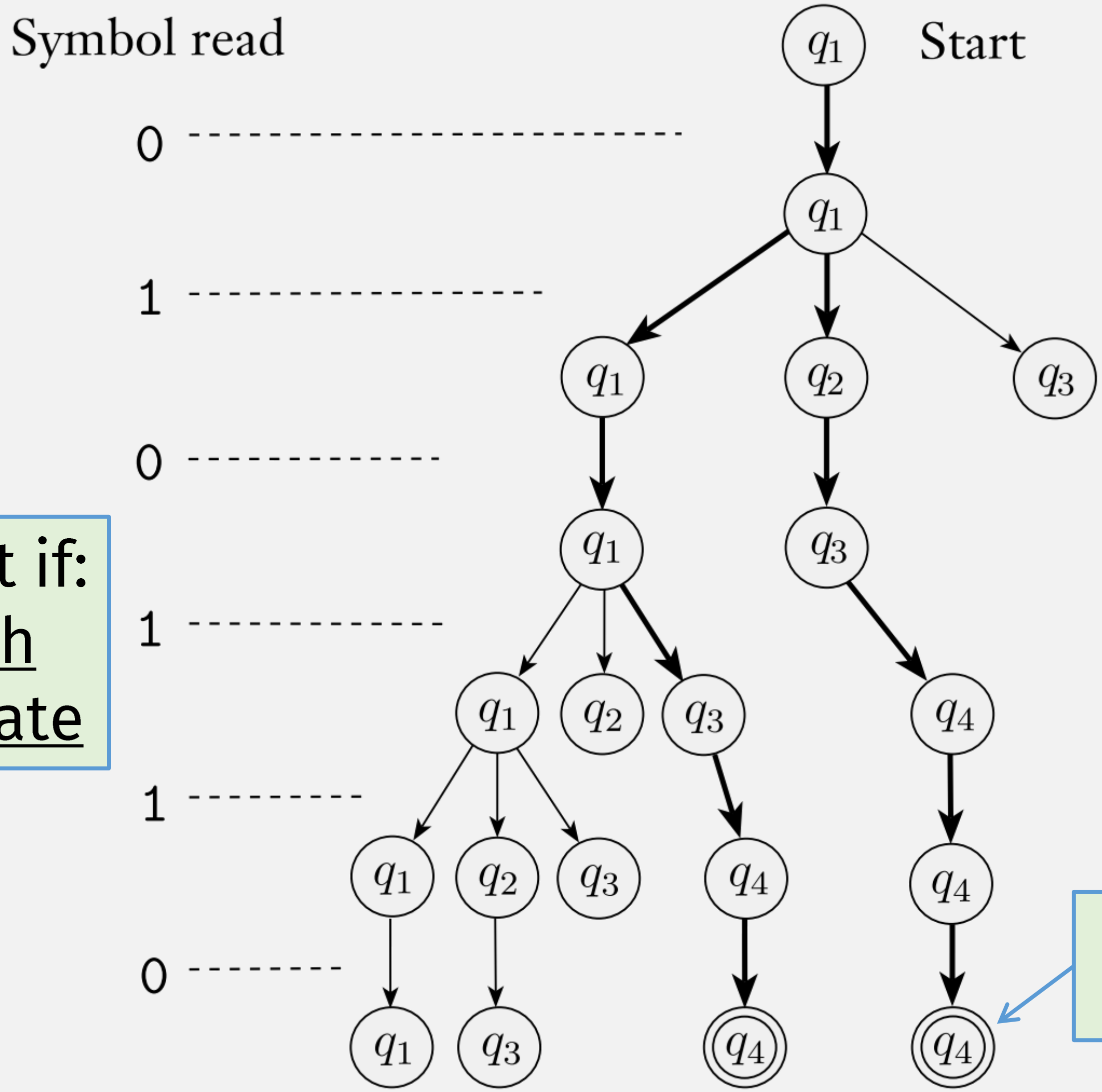
DFAs

NFA

NFA Computation (JFLAP demo): `010110`



NFA Computation Sequence (of set of states)



NFA **accepts** input if: at least <u>one path</u> <u>ends in accept state</u>

Each step can branch into <u>multiple states at the same time!</u>

So this is an **accepting computation**

Flashback

DFA Computation Rules

Informally	*Formally (i.e., mathematically)*
Given • A **DFA** (~ a "Program") • and **Input = string of chars**, e.g. "`1101`"	• $M = (Q, \Sigma, \delta, q_0, F)$ • $w = w_1 w_2 \cdots w_n$
A **DFA** <u>**computation**</u> (~ "Program run"): • <u>Start</u> in *start state*	A **DFA computation** is a <u>sequence of states:</u>

• <u>Repeat</u>:
  • <u>Read</u> **1 char** from **Input,** and
  • <u>Change state</u> according to *transition rules*
	• specified by $\hat{\delta}(q_0, w)$ where:

<u>Result</u> of **computation:**
  • <u>Accept</u> if **last state** is *Accept state*
  • <u>Reject</u> otherwise
	• M **accepts** w if $\hat{\delta}(q_0, w) \in F$
	• M **rejects** otherwise

Flashback

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input = string of chars**, e.g. "1101"

A **DFA** <u>computation</u> (~ "Program run"):

- <u>Start</u> in *start state*
- <u>Repeat</u>:
  - <u>Read</u> **1 char** from **Input**, and
  - <u>Change state</u> according to *transition rules*

<u>Result</u> of **computation**:

- <u>Accept</u> if **last state** is *Accept state*
- <u>Reject</u> otherwise

Formally *(i.e., mathematically)*

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a <u>sequence of states:</u>

- specified by $\hat{\delta}(q_0, w)$ where:

- M **accepts** w if $\hat{\delta}(q_0, w) \in F$
- M **rejects** otherwise

Ignoring ε transitions, for now!

NFA Computation Rules

Informally

Given

- An NFA (~ a "Program")
- and **Input** = **string of chars**, e.g. "1101"

An NFA computation (~ "Program run"):

- Start in *start state*
- Repeat:
  - **Read 1 char** from **Input**, and
  - For each "current" state, go to next states according to *transition rules*
  - ... then combine all "next states"

Result of computation:

- Accept if last set of states has accept state
- Reject otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

An NFA **computation** is a ...

- specified by $\hat{\delta}(q_0, w)$ where:

- *M* **accepts** *w* if ...
- *M* **rejects** ...

Ignoring ε transitions, for now!

NFA Computation Rules

Informally

Given

- An NFA (~ a "Program")
- and Input = string of chars, e.g. "1101"

A DFA computation (~ "Program run"):

- Start in *start state*
- Repeat:
  - Read 1 char from Input, and

For each "current" state, go to next states according to *transition rules*

... then combine all "next states"

Result of computation:

- Accept if last set of states has accept state
- Reject otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

An **NFA computation** is a sequence of: sets of states

- specified by $\hat{\delta}(q_0, w)$ where:

???

- M **accepts** w if ...
- M **rejects** ...

Flashback

DFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$$

- <u>Domain</u> (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u> (output):
  - state $q \in Q$ (doesn't have to be an accept state)

Recursive Input Data
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

Base case

Base case $\hat{\delta}(q, \varepsilon) =$

Flashback

DFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \to Q$$

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - state $q \in Q$ (doesn't have to be an accept state)

Recursive Input Data
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

Recursive case

Recursion on string

"smaller" argument

string

char

(Defined recursively)

Base case $\hat{\delta}(q, \varepsilon) = q$

Recursion on string

string

char

Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

"second to last" state

Flashback

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$$

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - state $q \in Q$ (doesn't have to be an accept state)

Recursive Input Data
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (**non-empty string**) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

(Defined recursively)

Base case $\hat{\delta}(q, \varepsilon) = q$

Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

Single step from "second to last" state and last char gets to last state

$\delta : Q \times \Sigma_\varepsilon \longrightarrow \boxed{\mathcal{P}(Q)}$ is the transition function

NFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$$

Result is set of states

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):

states $qs \subseteq Q$

$\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function

NFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$$

Result is set of states

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - states $qs \subseteq Q$

(Defined recursively)

Base case $\hat{\delta}(q, \varepsilon) = \{q\}$

Recursively Defined Input
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "char" in Σ

Base case

$\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function

NFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$$

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - states $qs \subseteq Q$

Recursively Defined Input
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "char" in Σ

Recursive case

Recursive part

(Defined recursively)

Base case $\hat{\delta}(q, \varepsilon) = \{q\}$

Recursive Case $\hat{\delta}(q, w' w_n) =$

where $w' = w_1 \cdots w_{n-1}$

Recursion on recursive part

"second to last" set of states

$$\hat{\delta}(q, w') = \{q_1, \ldots, q_k\}$$

$\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function

NFA Extended Transition Function

$\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - states $qs \subseteq Q$

Recursively Defined Input
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (**non-empty string**) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

Last char

(Defined recursively)

Base case $\hat{\delta}(q, \varepsilon) = \{q\}$

For each "second to last" state, take single step on last char

Recursive Case $\hat{\delta}(q, w' w_n) = \bigcup_{i=1}^{k} \delta(q_i, w_n)$

where $w' = w_1 \cdots w_{n-1}$

$\hat{\delta}(q, w') = \{q_1, \ldots, q_k\}$

$\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function

NFA Extended Transition Function

$\hat{\delta} : Q \times \Sigma^* \rightarrow$

- Domain (inpu
  - state $q \in$
  - string $w =$
- Range (outpu

states $qs \subseteq$

Given
- An **NFA** (~ a "Program")
- and **Input = string of chars**, e.g. "1101"

A **DFA computation** (~ "Program run"):
- Start in *start state*
- Repeat:
  - **Read 1 char** from **Input**, and
  - according to *transition rules*

Recursively Defined Input
needs

Still ignoring ε transitions!

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "char" in Σ

(Defined recurs

For each "current" state, go to next states

… then combine all sets of "next states"

Recursive Case

$$\hat{\delta}(q, w'w_n) = \bigcup_{i=1}^{k} \delta(q_i, w_n)$$

where $w' = w_1 \cdots w_{n-1}$

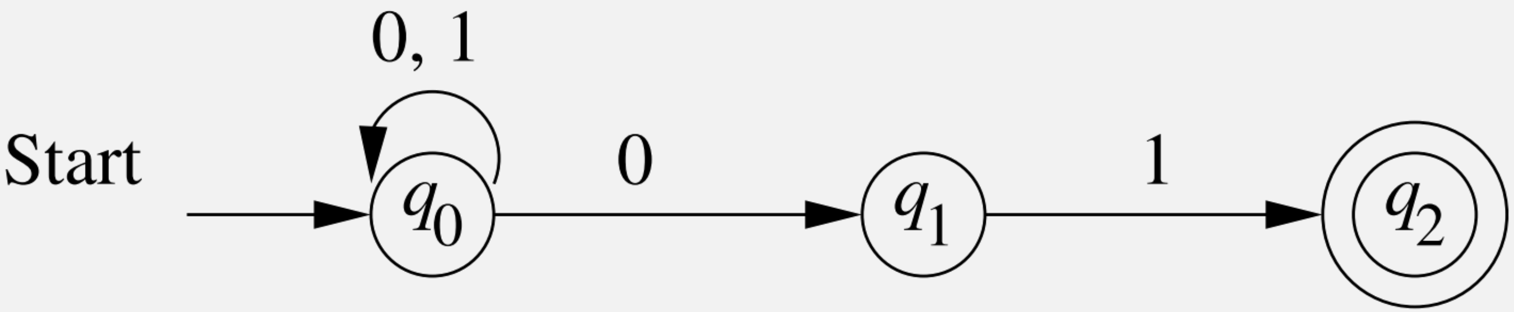
$$\hat{\delta}(q, w') = \{q_1, \ldots, q_k\}$$

NFA Extended δ Example

Base case: $\hat{\delta}(q, \epsilon) = \{q\}$

Recursive case: $\hat{\delta}(q, w) = \bigcup_{i=1}^{k} \delta(q_i, w_n)$

where: $\hat{\delta}(q, w_1 \cdots w_{n-1}) = \{q_1, \ldots, q_k\}$



- $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
- $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

We haven't considered **empty transitions!**

Combine result of recursive call with "last step"

Adding Empty Transitions

- Define **the set** $\varepsilon\text{-}\mathrm{REACHABLE}(q)$
  - ... to be all states reachable from q via <u>zero or more empty transitions</u>

(Defined recursively)

- <u>Base</u> case: $q \in \varepsilon\text{-}\mathrm{REACHABLE}(q)$

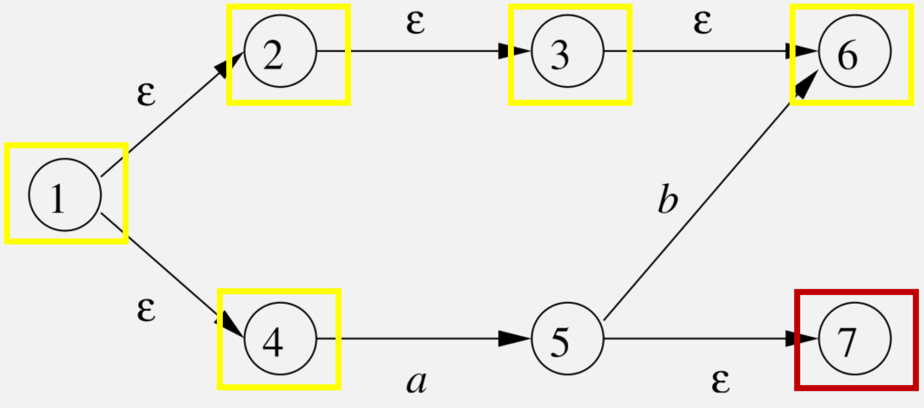
- <u>Inductive</u> case:

A state is in the reachable set if ...

$$\varepsilon\text{-}\mathrm{REACHABLE}(q) = \{r \mid p \in \varepsilon\text{-}\mathrm{REACHABLE}(q) \text{ and } r \in \delta(p, \varepsilon)\}$$

... there is an empty transition to it from another state in the reachable set

ε-REACHABLE Example



$$\varepsilon\text{-REACHABLE}(1) = \{1, 2, 3, 4, 6\}$$

Handling ε transitions now!

NFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$$

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - states $qs \subseteq Q$

(Defined recursively)

where $w' = w_1 \cdots w_{n-1}$

$\hat{\delta}(q, w') = \{q_1, \ldots, q_k\}$

Base case $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-REACHABLE}(q)$

Recursive Case $\hat{\delta}(q, w'w_n) = \bigcup_{i=1}^{k} \delta(q_i, w_n)$

Handling ε transitions now!

NFA Extended Transition Function

$$\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$$

- Domain (inputs):
  - state $q \in Q$ (doesn't have to be start state)
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range (output):
  - states $qs \subseteq Q$

(Defined recursively)

"Take single step, then follow all empty transitions"

where $w' = w_1 \cdots w_{n-1}$

$\hat{\delta}(q, w') = \{q_1, \ldots, q_k\}$

Base case $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-REACHABLE}(q)$

Recursive Case $\hat{\delta}(q, w' w_n) = \varepsilon\text{-REACHABLE}\left(\bigcup_{i=1}^{k} \delta(q_i, w_n)\right)$

Summary: NFA vs DFA Computation

DFAs

- Can only be in one state
- Transition:
  - Must read 1 char
- Acceptance:
  - If final state is accept state

NFAs

- Can be in multiple states
- Transition
  - Has empty transitions
- Acceptance:
  - If one of final states is accept state

Previously

Concatenation: $A \circ B = \{xy |\ x \in A \text{ and } y \in B\}$

Is Concatenation Closed?

THEOREM

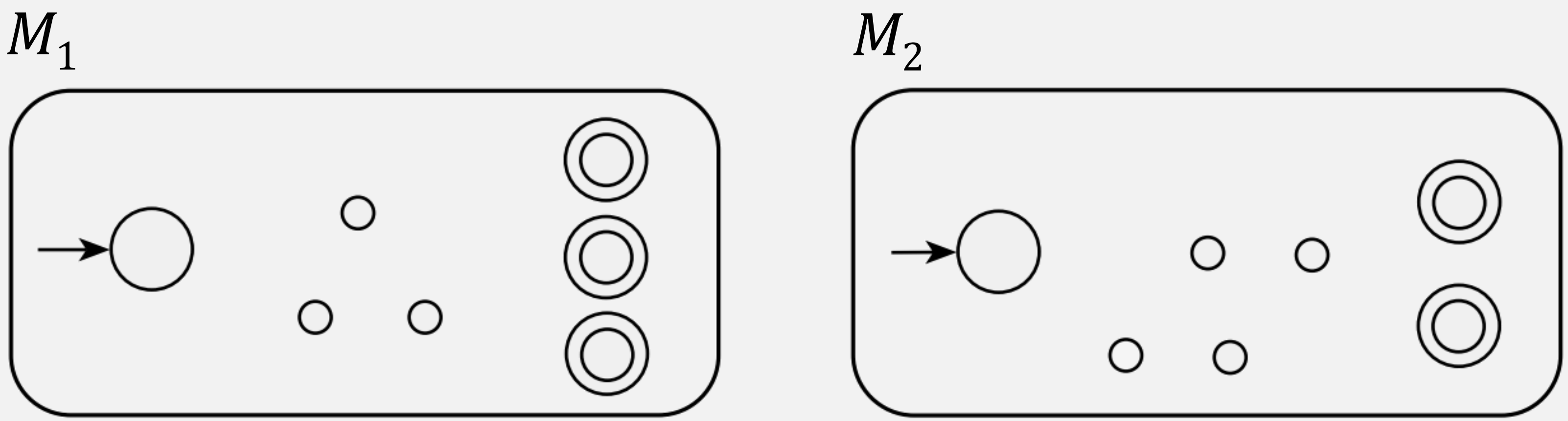
The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

Proof requires: Constructing new machine

- How does it know when to switch machines?
  - Can only read input once

Concatentation

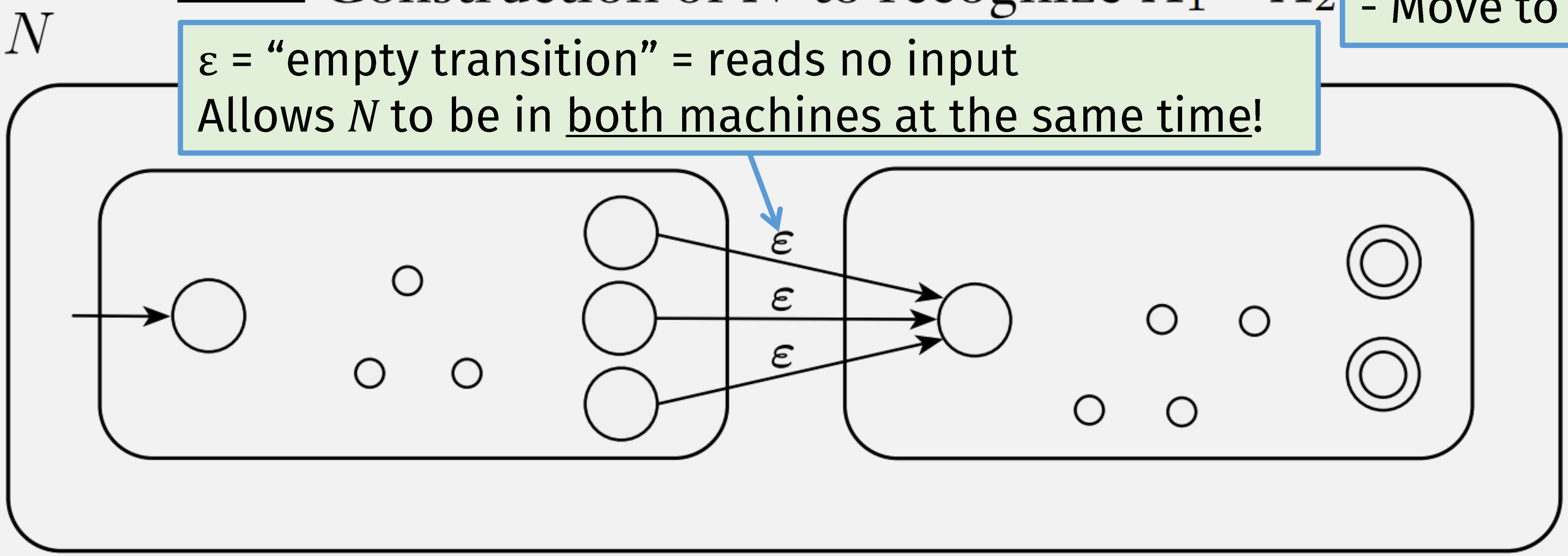


Let M_1 recognize A_1, and M_2 recognize A_2.

Want: Construction of N to recognize $A_1 \circ A_2$

N is an **NFA**! It can:
- Keep checking 1st part with M_1
  and
- Move to M_2 to check 2nd part

ε = "empty transition" = reads no input
Allows N to be in both machines at the same time!



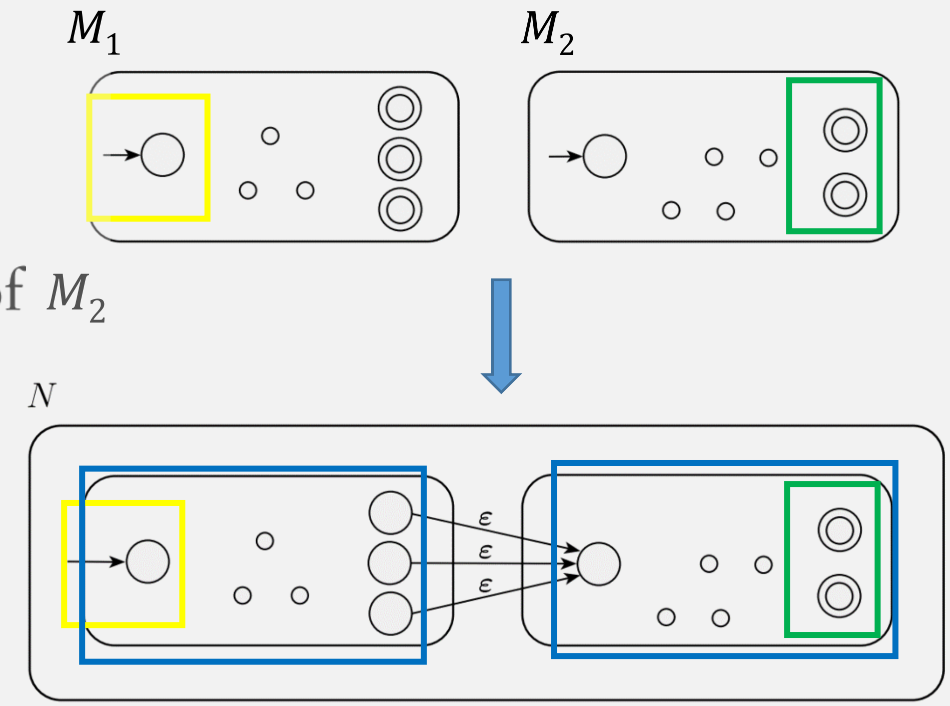
Concatenation is Closed for Regular Langs

PROOF (part of)

Let DFA $M_1 = (\boxed{Q_1}, \Sigma, \delta_1, \boxed{q_1}, F_1)$ recognize A_1
DFA $M_2 = (\boxed{Q_2}, \Sigma, \delta_2, q_2, \boxed{F_2})$ recognize A_2

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$

1. $\boxed{Q} = Q_1 \cup Q_2$
2. The state $\boxed{q_1}$ is the same as the start state of M_1
3. The accept states $\boxed{F_2}$ are the same as the accept states of M_2
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,



Concatenation is Closed for Regular Langs

Wait, is this true?

PROOF (part of)

Let DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1
DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2

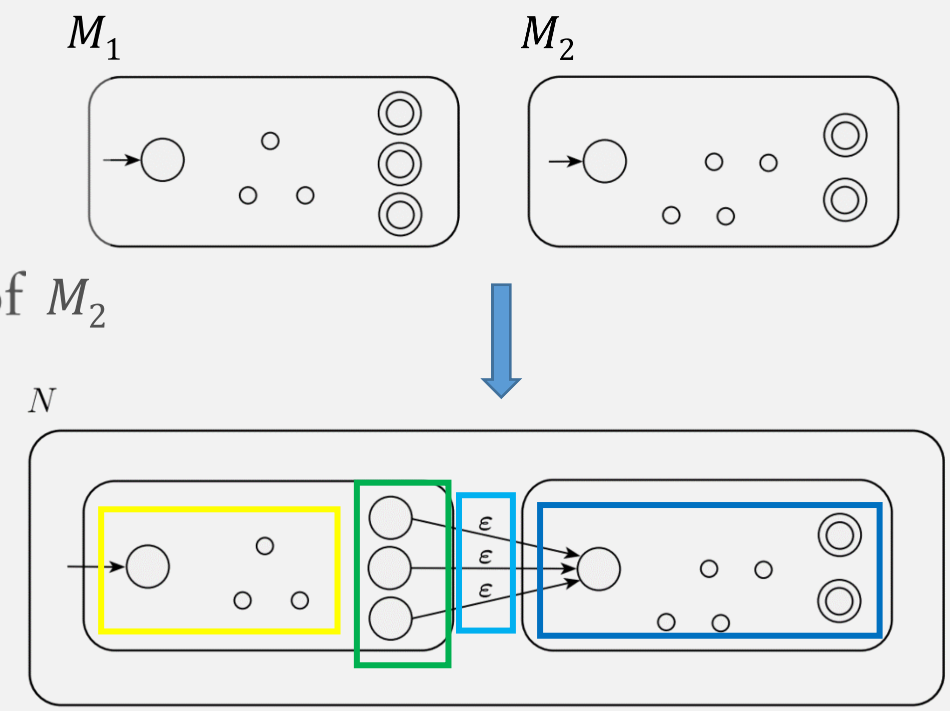
Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$

1. $Q = Q_1 \cup Q_2$
2. The state q_1 is the same as the start state of M_1
3. The accept states F_2 are the same as the accept states of M_2
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \{\delta_1(?, a)\} & \boxed{q \in Q_1 \text{ and } q \notin F_1} \\ \{\delta_1(?, a)\} & \boxed{q \in F_1} \text{ and } a \neq \varepsilon \\ ? \qquad \{q_2\} & q \in F_1 \text{ and } \boxed{a = \varepsilon} \\ \{\delta_2(?, a)\} & \boxed{q \in Q_2.} \end{cases}$$

NFA def says δ must map every state and ε to set of states

And: $\delta(q, \varepsilon) = \emptyset$, for $q \in Q, q \notin F_1$



??? ■

Previously

Is Union Closed For Regular Langs?

Proof

Statements	**Justifications**
1. A_1 and A_2 are regular languages	1. Assumption
2. A DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1	2. Def of Reg Lang (Coro)
3. A DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2	3. Def of Reg Lang (Coro)
4. Construct DFA $M = (Q, \Sigma, \delta, q_0, F)$	4. Def of DFA
5. M recognizes $A_1 \cup A_2$	5. See examples
6. $A_1 \cup A_2$ is a regular language	6. Def of Regular Language
7. The class of regular languages is closed under the union operation. In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.	7. From stmt #1 and #6

Q.E.D. ■

Is $\boxed{\text{Concat}}$ Closed For Regular Langs?

Proof?

Statements	**Justifications**
1. A_1 and A_2 are regular languages	1. Assumption
2. A DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1	2. Def of Reg Lang (Coro)
3. A DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2	3. Def of Reg Lang (Coro)
4. Construct $\boxed{\text{NFA}}$ $N = (Q, \Sigma, \delta, q_0, F)$	4. Def of $\boxed{\text{NFA}}$
5. N recognizes $\cancel{A_1 \cup A_2}$ $A_1 \circ A_2$	5. See examples
6. $\cancel{A_1 \cup A_2}$ $A_1 \circ A_2$ is a regular language	6. ??? Does NFA recognize reg langs?
7. The class of regular languages is closed under t $\boxed{\text{concatenation}}$ n operation. In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.	7. From stmt #1 and #6

Q.E.D.?

Previously

A DFA's Language

- For **DFA** $M = (Q, \Sigma, \delta, q_0, F)$

- M **accepts** w if $\hat{\delta}(q_0, w) \in F$

- M **recognizes** language $\{w \mid M \text{ accepts } w\}$

Definition: A DFA's language is a **regular language**

An NFA's Language?

- For **NFA** $N = (Q, \Sigma, \delta, q_0, F)$

intersection

accept states

- N ***accepts*** w if $\hat{\delta}(q_0, w) \cap F \neq \emptyset$ not empty
  - i.e., accept if final states contain <u>at least one</u> accept state

- Language of N = $L(N)$ = $\left\{ w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset \right\}$

<u>Q</u>: What kind of languages do NFAs recognize?

Concatenation Closed for Reg Langs?

- **Combining DFAs to recognize concatenation of languages …**

  … **produces** an **NFA**

- So **to prove concatenation is closed …**

  … **we must prove** that **NFAs *also* recognize regular languages**.

Specifically, we must prove:

NFAs ⇔ regular languages

“If and only if” Statements

$X \Leftrightarrow Y$ = “X if and only if Y” = X iff Y = X <=> Y

Represents <u>two</u> statements:

1. $\Rightarrow$ if X, then Y
   - “**forward**” direction

2. $\Leftarrow$ if Y, then X
   - “**reverse**” direction

How to Prove an “iff” Statement

$X \Leftrightarrow Y$ = “X if and only if Y” = X iff Y = X <=> Y

Proof has <u>two</u> (If-Then proof) parts:

1. $\Rightarrow$ if X, then Y
   - “**forward**” direction
   - assume X, then use it to prove Y
2. $\Leftarrow$ if Y, then X
   - “**reverse**” direction
   - assume Y, then use it to prove X