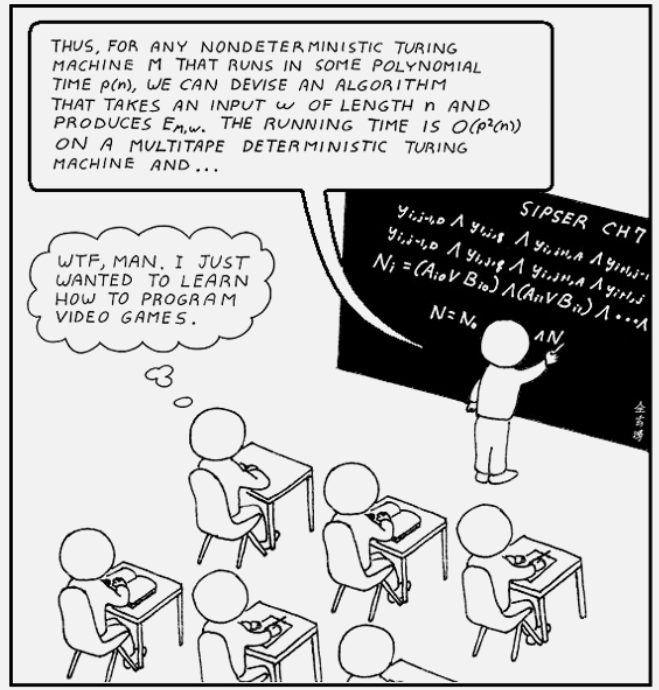
CS 420

Turing Machine Variants

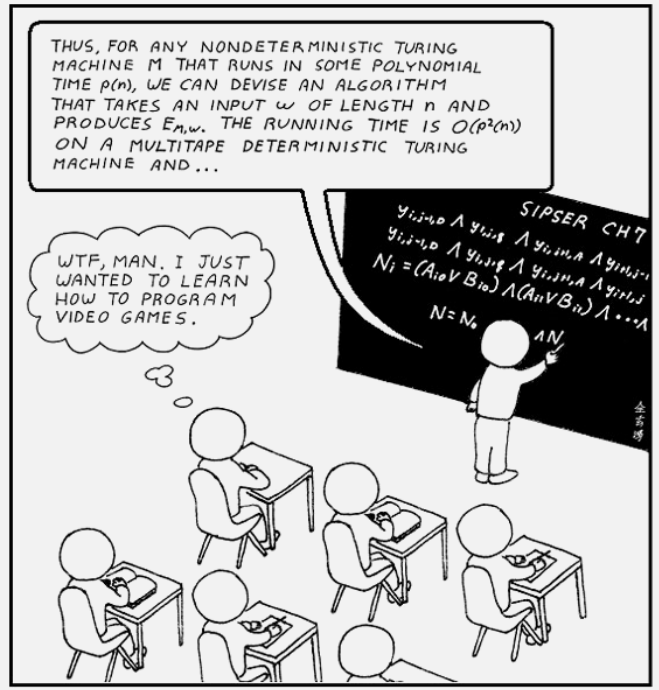
Thursday, November 5, 2026

UMass Boston Computer Science



Announcements

- HW 7
  - ~~Due: 11/5~~
- HW 8
  - Out: Thu 11/5
  - Due: Thu 11/12 12:30pm EST

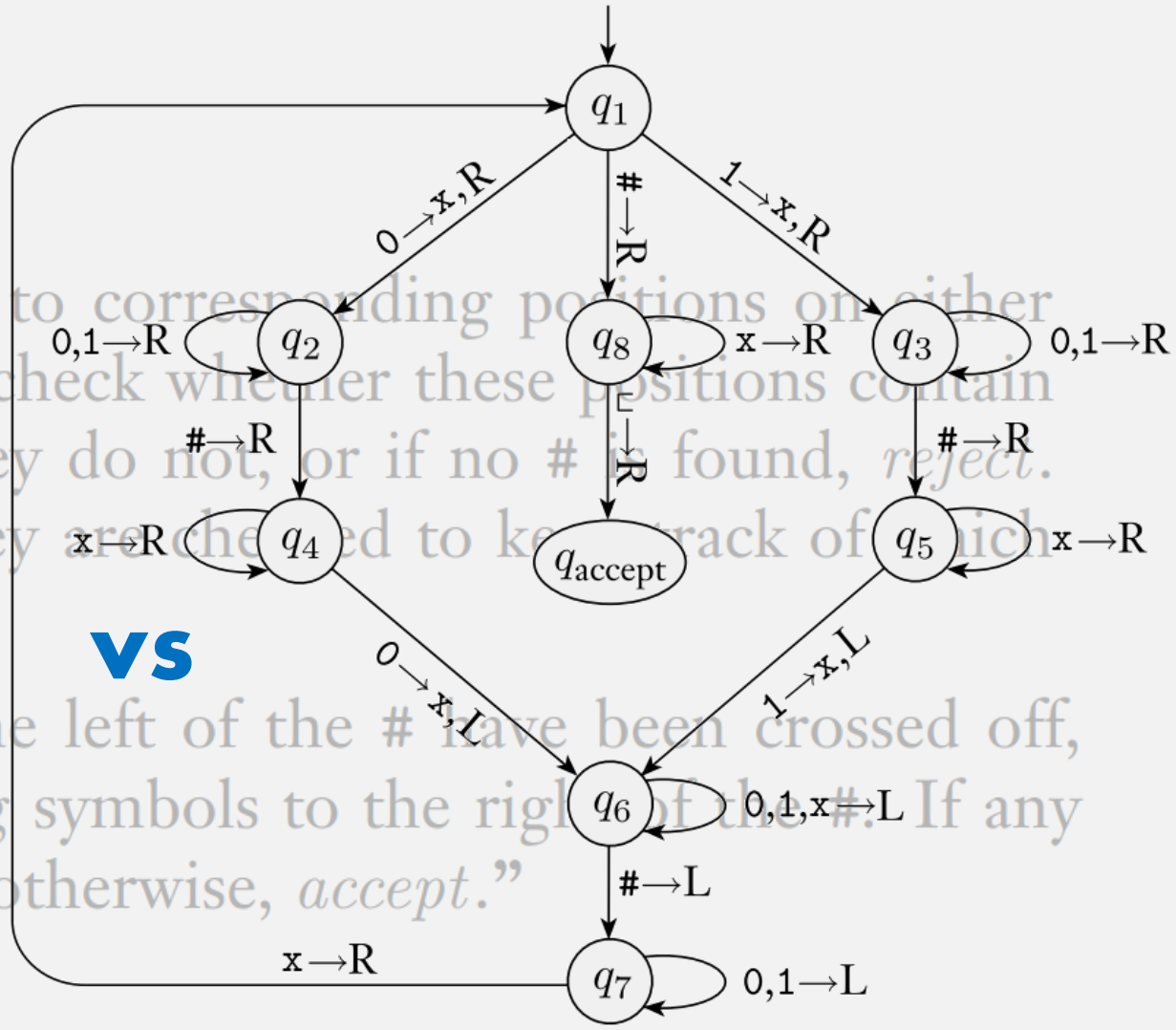


Last Time

TMs: High-level vs Low-level?

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, *reject*; otherwise, *accept*."



Turing Machine: High-Level Description

- M_1 accepts if input is in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

M_1 = "On input string w:

1. Zig-zag across th ding positions on either side of the # s ne the same symb n Cross off symbo d to keep track of which symbols correspond.
2. When all symbols to the check for any remaining s symbols remain, *reject*; ot

We will (mostly) define TMs using **high-level descriptions,** like this one

(But it must always **correspond to** some formal **low-level tuple** description)

Analogy:
High-level (e.g., Python) function definitions
vs
Low-level assembly language

TM High-level Description Tips

Analogy:

- **High-level** TM description ~ function definition in **"high level" language,** e.g. **Python**
- Low-level TM tuple ~ function definition in **bytecode** or **assembly**

TM high-level descriptions must still follow **precise rules!**

1. TMs and input strings must be named (like **function definitions**)
2. Steps must be numbered
3. TMs can "Call" or "Simulate" **other TMs** (if they "**pass**" appropriate **arguments**!)
   - e.g., **step for a TM** M can **say: "call TM** M_2 **with argument string** w**, if** M_2 **accepts** w **then ..., e**
   - Can **split input into substrings** and **pass to different TMs**
4. Follow typical programming "scoping" rules
   - can **assume functions already defined** are in "global" scope, "CONVERT" ...
5. Other variables must also be defined before use
   - e.g., can define a TM inside another TM
6. must be **equivalent** to a low-level formal tuple
   - high-level "step" represents a finite # of low-level δ transitions
   - So one step cannot run forever
   - E.g., can't say "count to infinity" as a "step"

let M_1 = "On input string w:

Too vague: "**convert NFA** N to **DFA** D where $LangOf(N) = LangOf(D)$", must give actual conversion steps!

let M = "On input w

1. Simulate B on input w.
2. If simulation ends in accept state...

let N = "On input $\langle B, w \rangle$, where B IS-A NFA AND w IS-A string:

1. let C = $\text{CONVERT}_{\text{NFA-DFA}}$ B
2. Run TM M from Theorem 4.1 on input $\langle C, w \rangle$.

let S = "On input w

1. let M_2 = "On input x:

Non-halting Turing Machines (TMs)

IMPORTANT SLIDE

- **A Turing Machine** can <u>run forever</u>
  - E.g., **head** can **move back and forth in a loop**

So: **TM computation** has <u>3 possible results</u>:
- Accept
- Reject
- Loop forever

- We will work with <u>two classes of Turing Machines</u>:
  - A **recognizer** is a **Turing Machine** that **may run forever** (all possible TMs)
  - A **decider** is a **Turing Machine** that **always halts.**

DefRecognizeLang

IF M IS-A **Turing** Machine AND M **recognizes** L
THEN L IS-A **Turing-recognizable** Language

IF M IS-A **TM** AND M **recognizes** L
THEN L IS-A **Recognizable** Language

(abbrev)

(3 possible computation results)

DefDecideLang

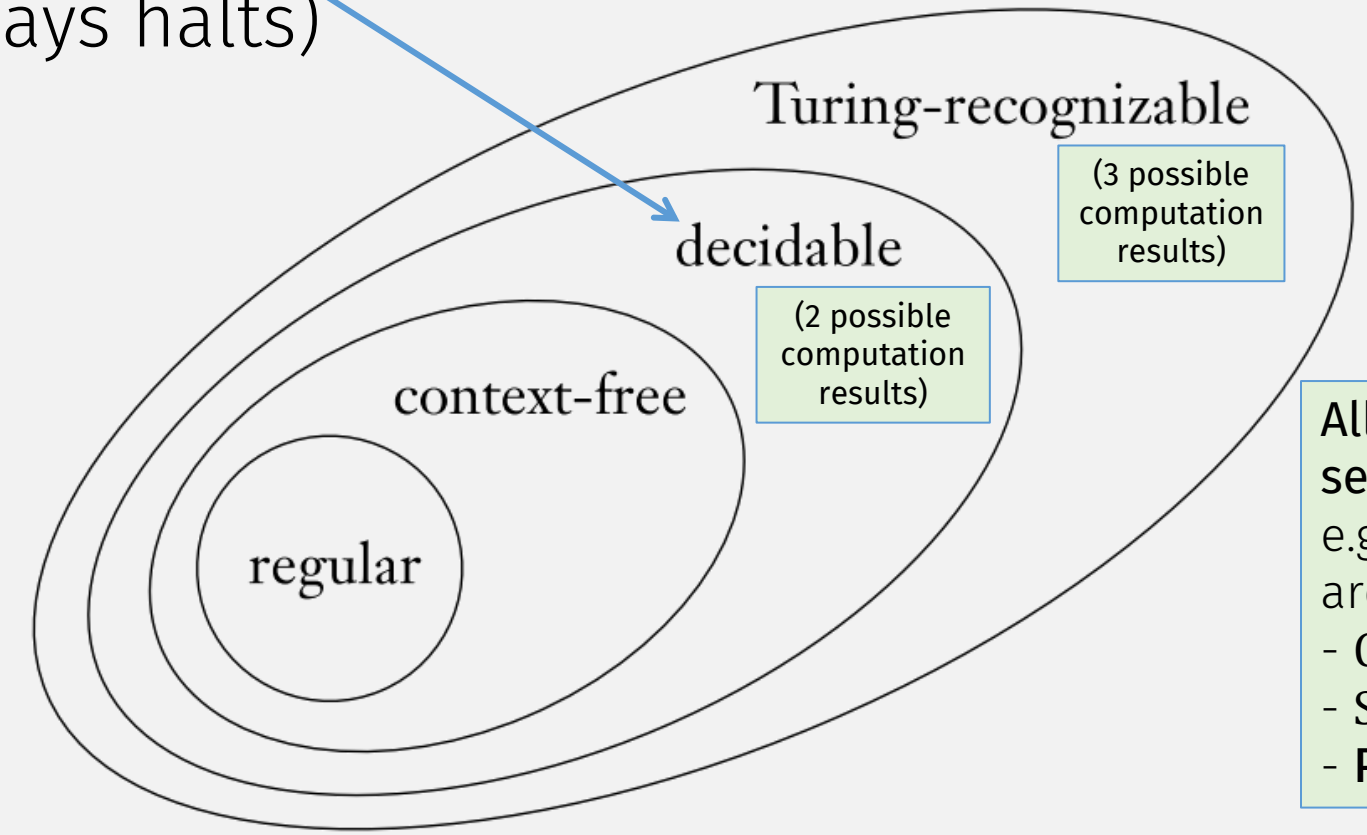
IF M IS-A **Decider** Machine AND M **decides** L
THEN L IS-A **Turing-decidable** Language

IF M IS-A **Decider** AND M **decides** L
THEN L IS-A **Decidable** Language

(2 possible computation results)

Formal Definition of an “Algorithm”

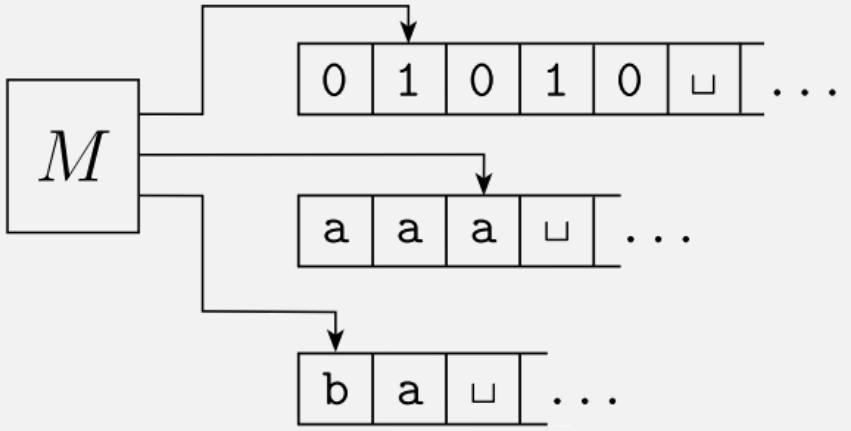
- An **algorithm** is **equivalent** to a **Turing-decidable** Language (always halts)



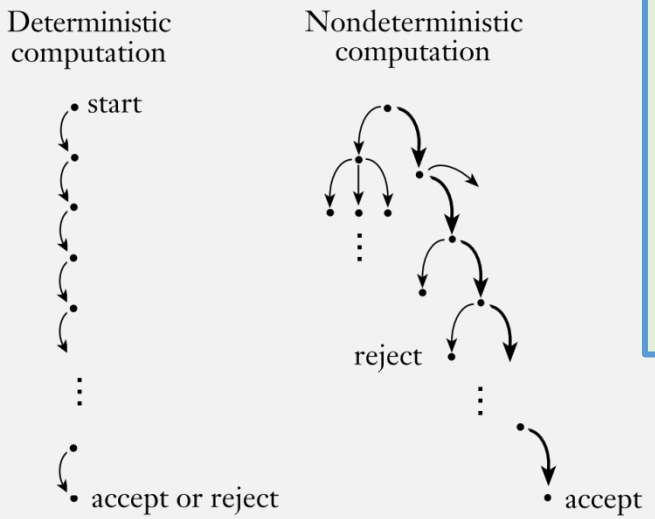
All <u>functions</u> we have **defined this semester** are **algorithms!**
e.g., all our <u>conversion</u> functions are **decider** TMs!!
- $\text{CONVERT}_{\text{DFA-NFA}}$
- STAR_{NFA}
- **PDA→CFG**

Turing Machine Variations

1. **Multi-tape TMs**



2. **Non-deterministic TMs**



Want to **prove:** these **TM variations** are **equivalent to** deterministic, single-tape machines

Reminder: Equivalence of Machines

- Two machines are **equivalent** when …
- … they recognize the same language

Theorem: Single-tape TM ⇔ Multi-tape TM

⇒ IF a single-tape TM recognizes a language,
THEN a multi-tape TM recognizes the language

- Single-tape TM is equivalent to …
- … multi-tape TM that only uses one of its tapes
- (could you write out the formal conversion?)

⇐ IF a multi-tape TM recognizes a language,
THEN a single-tape TM recognizes the language

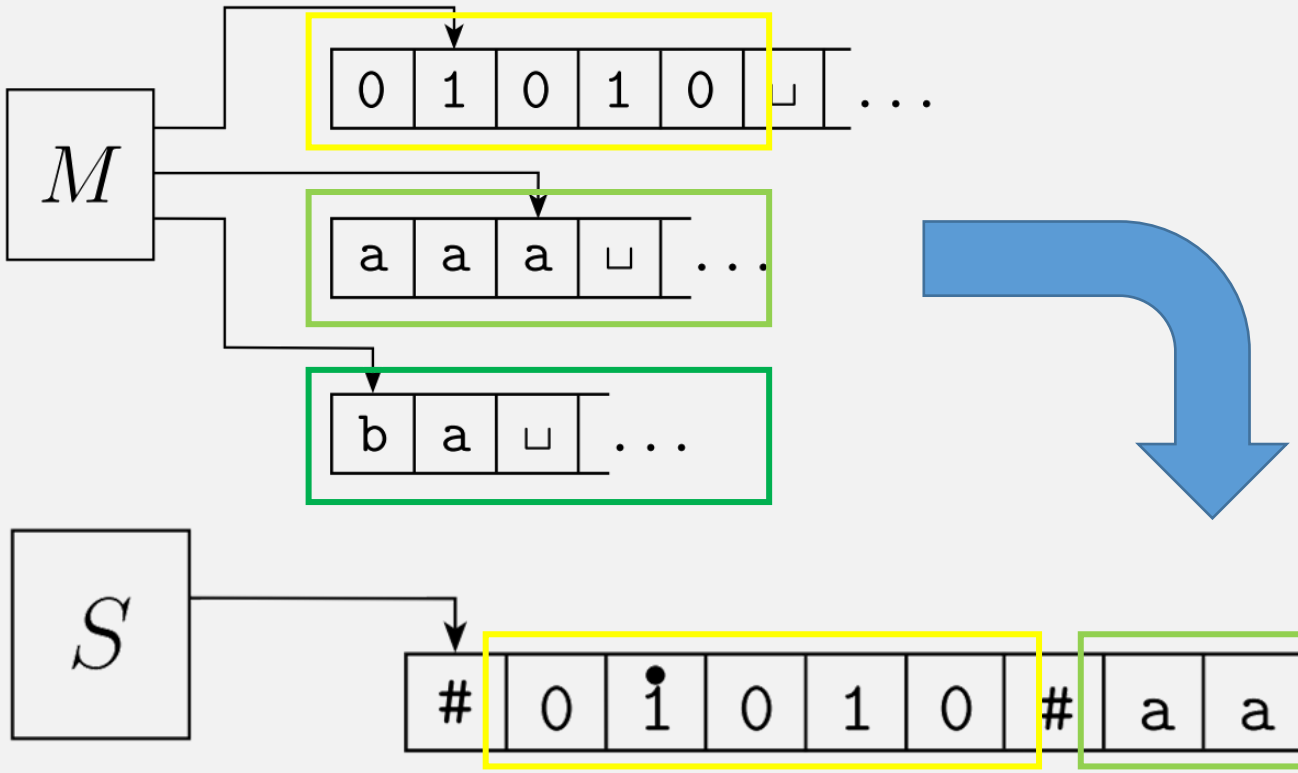
- Convert: multi-tape TM → single-tape TM

Key insight: single-tape is infinite in length!

Multi-tape TM ➔ Single-tape TM

Idea: **Use delimiter (#)** on **single-tape** to **simulate** multiple **tapes**

- **Add "dotted" version of every char** to **simulate** multiple **heads**



Single-tape machine will take **more steps** to **accept/reject,** but the **only requirement for "equivalence"** is …

… the **machines recognize** the **same language!** i.e., they
- **accept** the **same strings**
- **don't accept** the **same strings**

Theorem: Single-tape TM ⇔ Multi-tape TM

☑ ⇒ IF a **single-tape TM recognizes** a **language**,
THEN a **multi-tape TM recognizes** the **language**

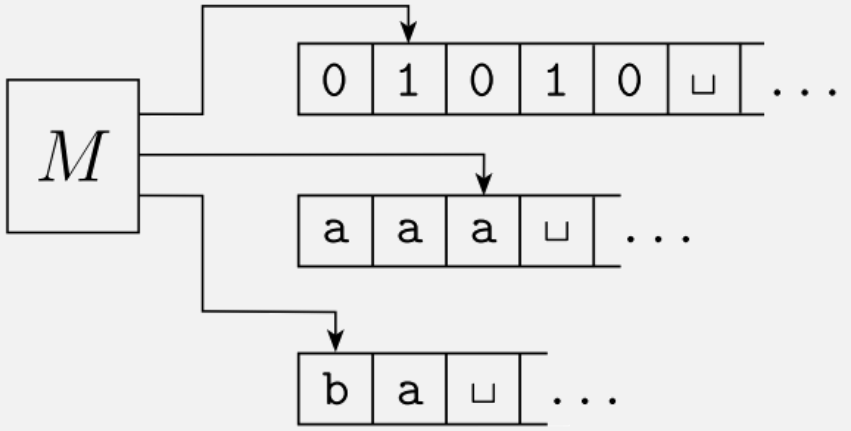
- Single-tape TM is equivalent to …
- … multi-tape TM that only uses one of its tapes

☑ ⇐ IF a **multi-tape TM recognizes** a **language**,
THEN a **single-tape TM recognizes** the **language**

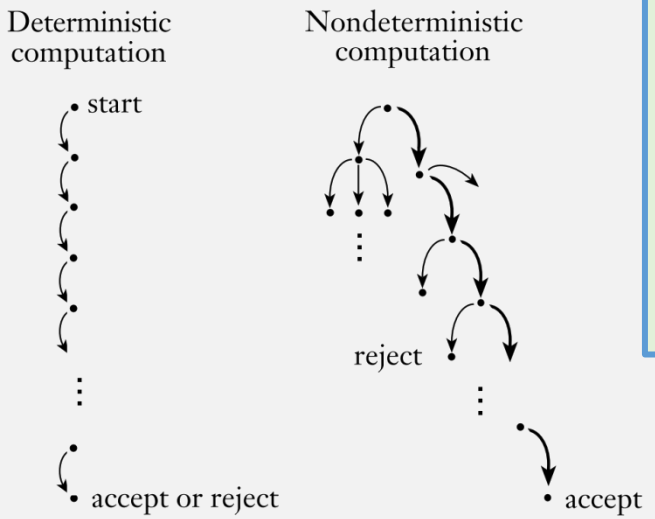
- Convert: multi-tape TM → single-tape TM

■

☑ **1. Multi-tape TMs**



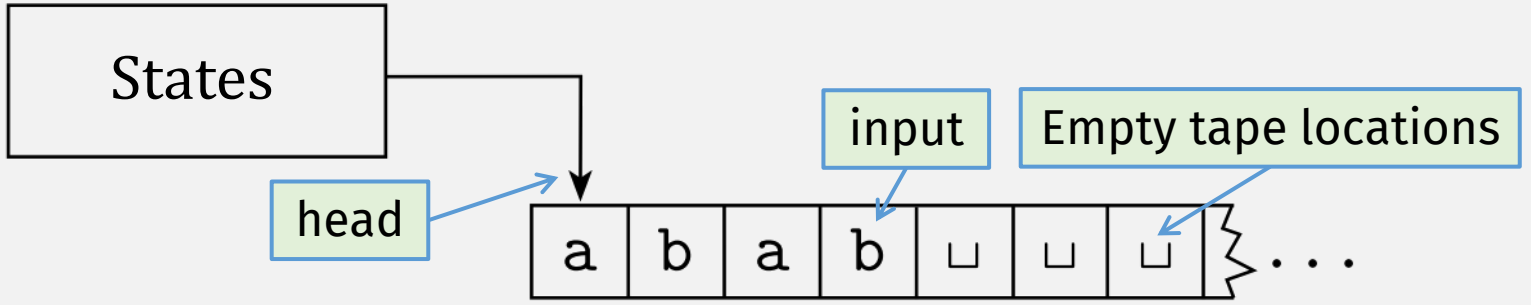
➡ **2. Non-deterministic TMs**



Want to **prove:** these **TM variations** are **equivalent to** deterministic, single-tape machines

Previously: Turing Machines

- **Turing Machines** can read and write to arbitrary "tape" cells
  - Tape initially contains input string
- The tape is infinite
  - (to the right)



- On a transition, "head" can move left or right 1 step

IF M IS-A **TM** AND M **recognizes** L
THEN L IS-A **Recognizable** Language

Turing Machine: High-Level Description

M_1 accepts if input is in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

let M_1 = "On input string w:

1. Zig-zag across th ding positions on either side of the # s the same symb Cross off symbo to keep track of which symbols correspond.
2. When all symbols to the check for any remaining s symbols remain, *reject*; ot

We will (mostly) define TMs using **high-level descriptions,** like this one

(But it must always **correspond to** some formal **low-level tuple** description)

Analogy:
High-level (e.g., Python) function definitions
vs
Low-level assembly language

Turing Machines: Formal Tuple Definition

DefTMLow

A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

read write move

Flashback: DFAs vs NFAs

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

vs

A ***nondeterministic finite automaton*** is a `mk-NFA` $(Q, \Sigma, \delta, q_0, F)$, where

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

Nondeterministic transition produces set of possible next states

Remember: Turing Machine Formal Definition

DefTMLow

A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Nondeterministic

Remember:

Turing Machine Formal Definition

DefNTMLow

Nondeterministic Turing Machine

A is a `mk-NTM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. ~~$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$~~ $\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\text{L}, \text{R}\})$
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Thm: Deterministic TM ⇔ Non-det. TM

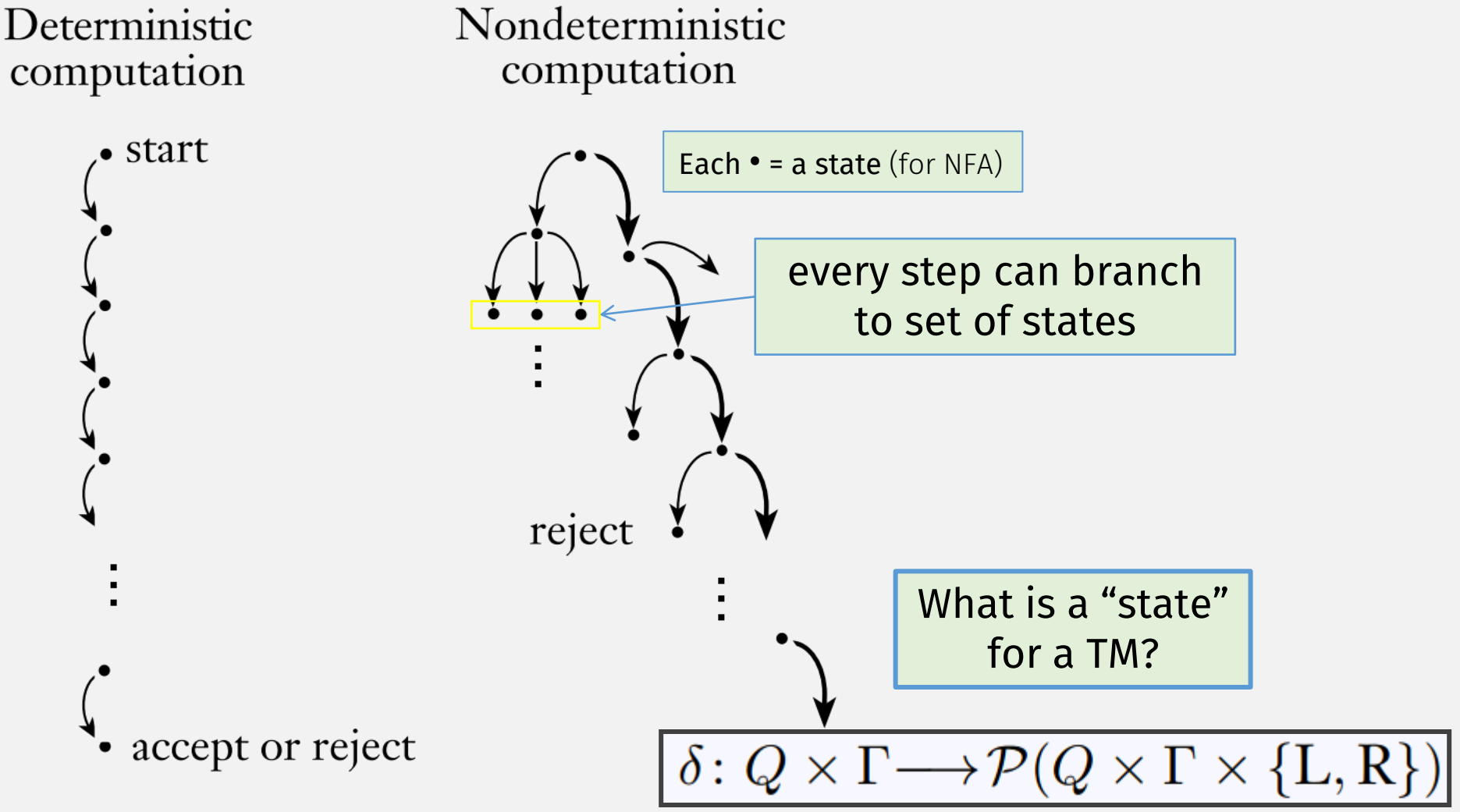
⇒ IF a **deterministic TM** recognizes a language,
THEN a **non-deterministic TM** recognizes the language

- Convert: Deterministic TM → Non-deterministic TM …
- … change Deterministic TM δ output to: one-element set
  - $\delta_{\text{NTM}}(q, a) = \{\delta_{\text{DTM}}(q, a)\}$
  - (just like conversion of DFA to NFA --- from previous hws)
- **DONE!**

⇐ IF a **non-deterministic TM** recognizes a language,
THEN a **deterministic TM** recognizes the language

- Convert: Non-deterministic TM → Deterministic TM …
- … ???

Review: Nondeterminism



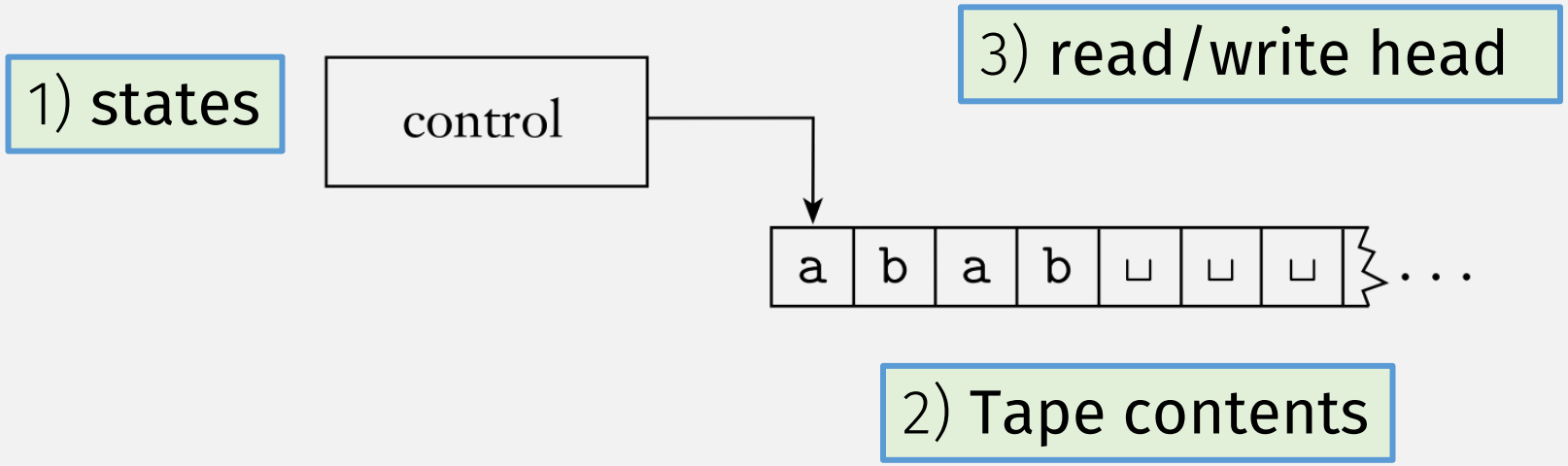
Flashback: PDA Configurations (IDs)

- A **configuration** (or **ID**) is a "snapshot" of a PDA's computation

- 3 components (q, w, γ) :
  q = the **current state**
  w = the **remaining input string**
  γ = the **stack contents**

A **sequence of configurations** represents a **PDA** computation

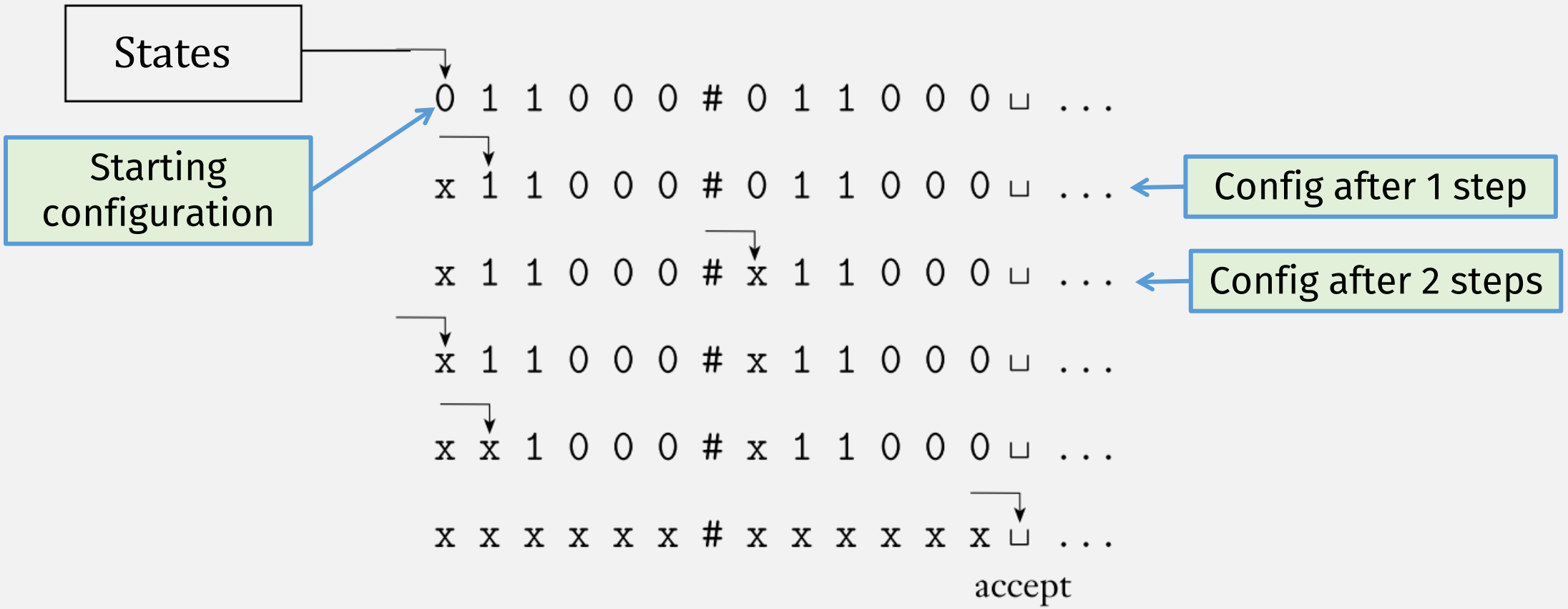
TM Configuration (ID) = ???



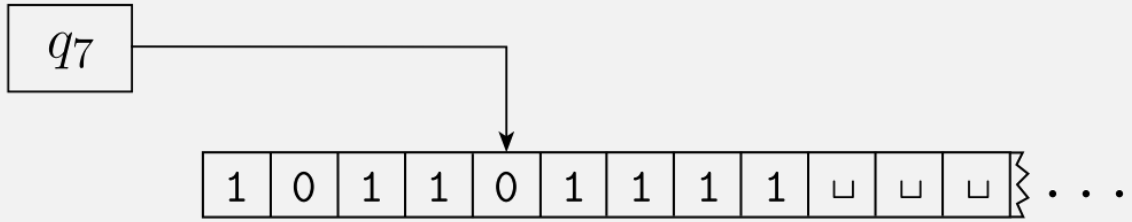
A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

TM Configuration = State + Head + Tape



TM Configuration = State + Head + Tape



$$1011q_701111$$

Textual representation of "configuration" (use this in HW)

1st char after state is current head position

TM Computation, Formally

Sipser says:
"For completeness, we say that the **head moves Right** in … transitions to the reject state"

let $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ = **`get-TM-parts`** …

Single-step

(Right) $\alpha q_1 \mathbf{a} \beta \vdash \alpha \mathbf{x} q_2 \beta$

head — Next config (head moved past written char)

write

if $q_1, q_2 \in Q$

$\delta(q_1, \mathbf{a}) = (q_2, \mathbf{x}, \mathrm{R})$

read

$\mathbf{a}, \mathbf{x} \in \Gamma \quad \alpha, \beta \in \Gamma^*$

(Left) $\alpha b q_1 \mathbf{a} \beta \vdash \alpha q_2 b \mathbf{x} \beta$

head

if $\delta(q_1, \mathbf{a}) = (q_2, \mathbf{x}, \mathrm{L})$

(wrote **x** and) head moved left

Multi-step

- Base Case

$I \vdash^* I$ for any ID I

- Recursive Case

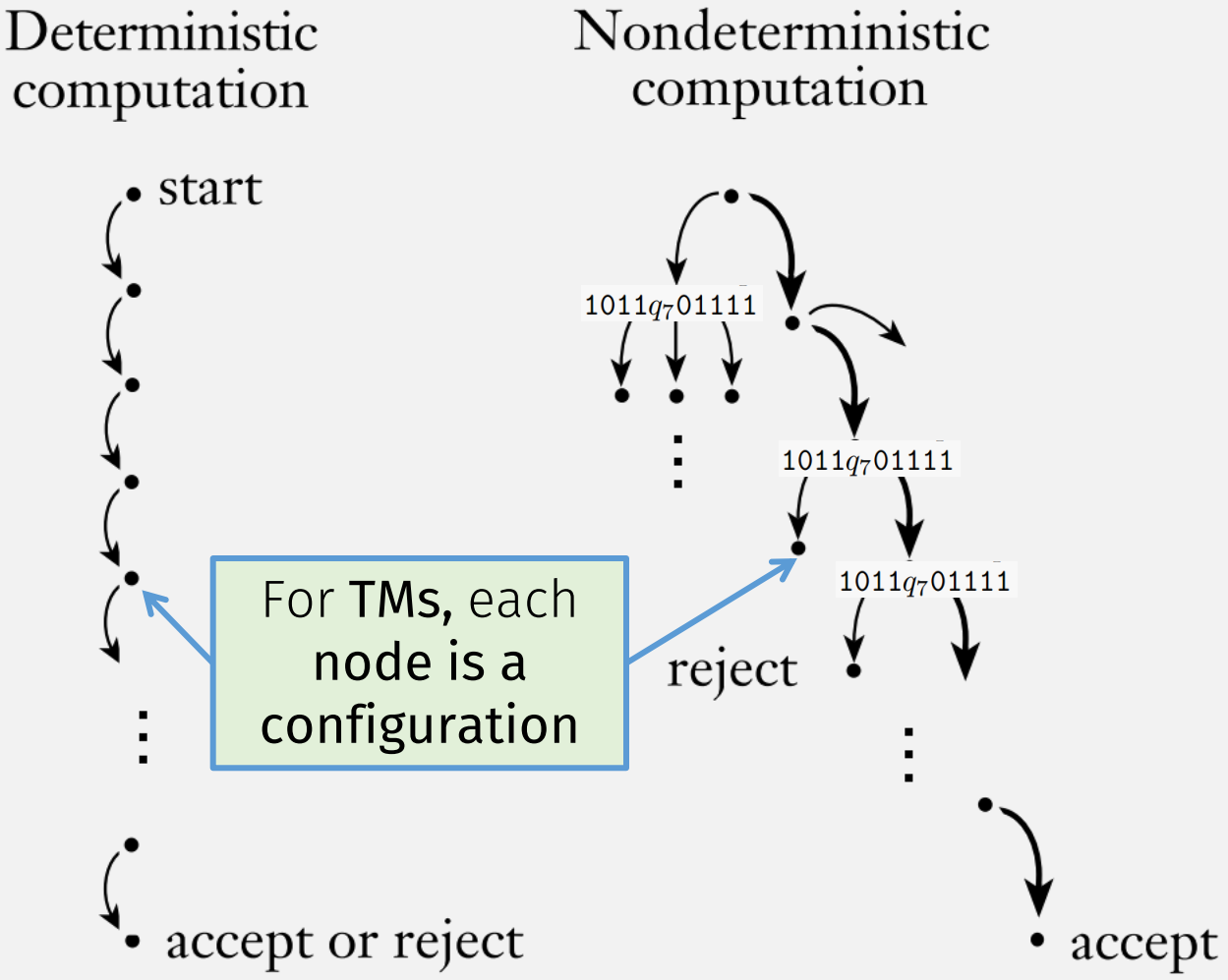
$I \vdash^* J$ if there exists some ID K such that $I \vdash K$ and $K \vdash^* J$

Edge cases:

Head stays at leftmost cell — $q_1 \mathbf{a} \beta \vdash q_2 \mathbf{x} \beta$ if $\delta(q_1, \mathbf{a}) = (q_2, \mathbf{x}, \mathrm{L})$ (L move, when already at leftmost cell)

Add blank symbol to config — $\alpha q_1 \vdash \alpha \sqcup q_2$ if $\delta(q_1, \sqcup) = (q_2, \sqcup, \mathrm{R})$ (R move, when at rightmost filled cell)

Nondeterminism in TMs

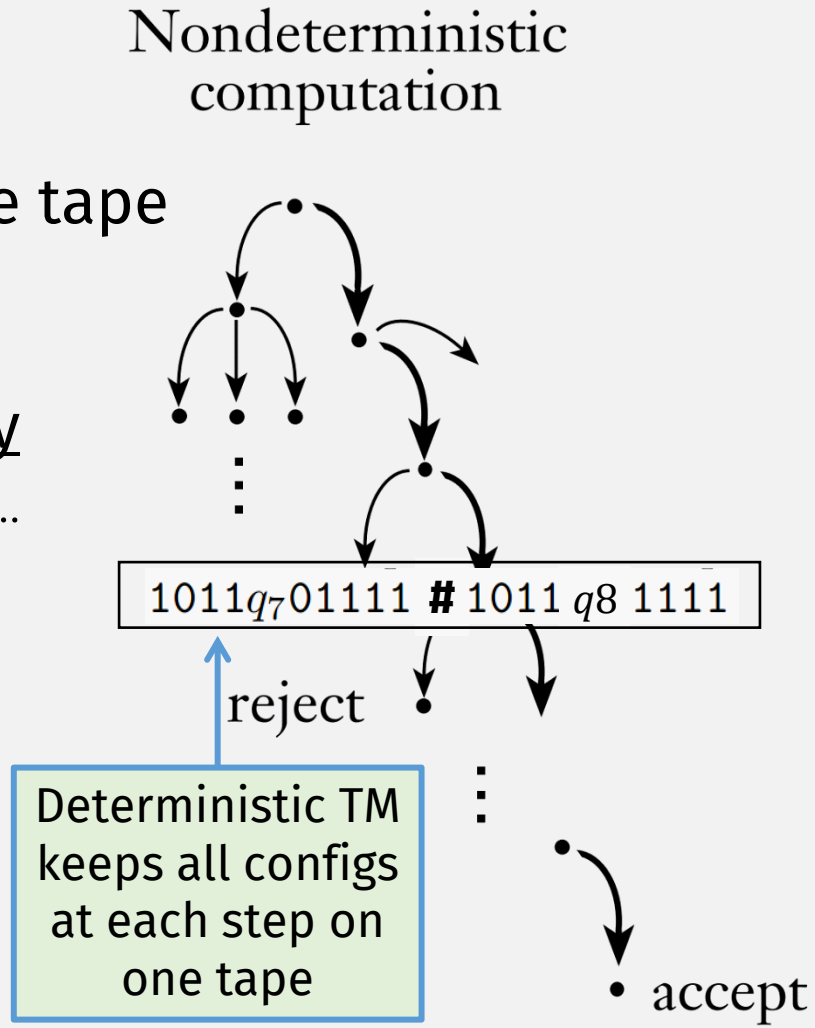


Nondeterministic TM → Deterministic

1st way

- Simulate NTM with Det. TM:
  - Det. TM keeps multiple configs on single tape
    - Like how single-tape TM simulates multi-tape
  - Then run all computations, <u>concurrently</u>
    - I.e., 1 step on one config, 1 step on the next, …
  - Accept if any accepting config is found
  - **Important**:
    - Why must we step configs <u>concurrently</u>?

Because any one path can go on forever!



Interlude: Running TMs inside other TMs

Remember analogy: **TMs are like function definitions, they can be "*called*" like functions …**

Exercise:

- Given: TMs M_1 and M_2
- Create: TM $\boldsymbol{M}$ that **accepts** if either M_1 or M_2 accept

Possible solution #1:

let $\boldsymbol{M}$ = on input x,

1. `Call` M_1 with arg x; **accept** x if M_1 accepts
2. `Call` M_2 with arg x; **accept** x if M_2 accepts

Possible Results for $\boldsymbol{M}$

M_1	M_2	$\boldsymbol{M}$	"in the lang" that we want M to recognize: $\boldsymbol{M}$ Expected?
reject	accept	accept	accept
accept	reject	accept	accept
accept	loops	accept	accept
loops	accept	loops	accept

Note: This solution would be ok if we knew M_1 and M_2 were **deciders** (which halt on all inputs)

"loop" means input string not accepted (but it should be)

Interlude: Running TMs inside other TMs

$\alpha q_1 \mathbf{a} \beta \vdash \alpha \mathbf{x} q_2 \beta$

Just an analogy: "*calling*" a TM actually requires "computing" how it computes …

… with concurrency!

Exercise:

- Given: TMs M_1 and M_2
- Create: TM $\boldsymbol{M}$ that **accepts** if either M_1 or M_2 accept

Possible solution #1:

let M = on input x,

1. Call M_1 with arg x; accept x if M_1 accepts
2. Call M_2 with arg x; accept x if M_2 accepts

M_1	M_2	M	
reject	accept	accept	☑
accept	reject	accept	
accept	loops	accept	
loops	accept	loops	✖

Possible solution #2:

let $\boldsymbol{M}$ = on input x,

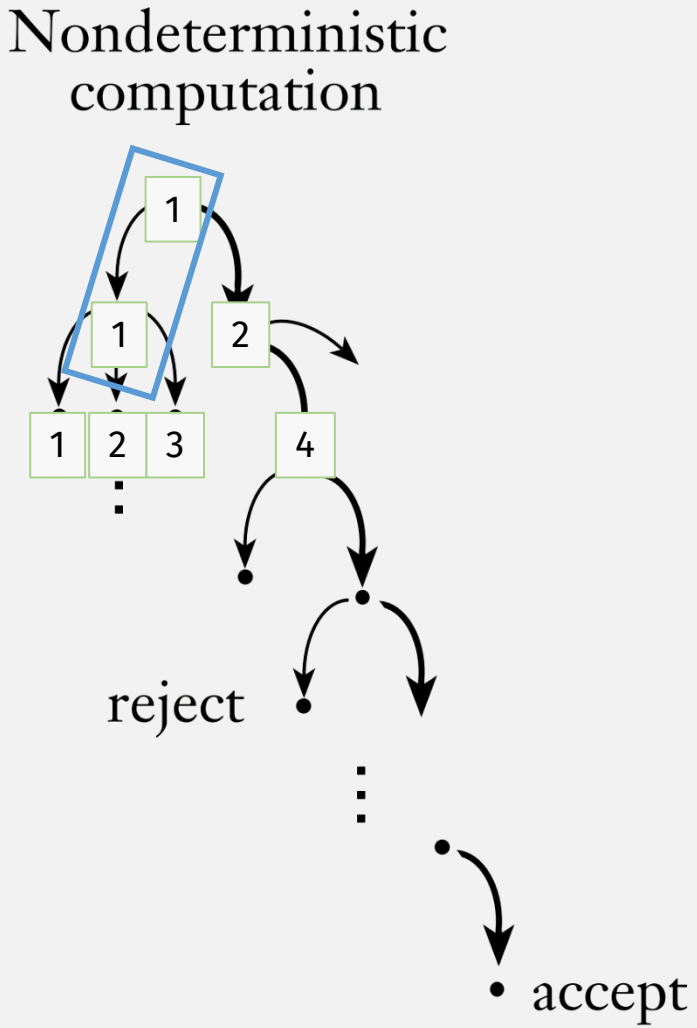
1. Call M_1 and M_2, each with x, concurrently, i.e.,
   a) Run M_1 with x for 1 step; **accept** x if M_1 accepts
   b) Run M_2 with x for 1 step; **accept** x if M_2 accepts
   c) Repeat

M_1	M_2	$\boldsymbol{M}$	$\boldsymbol{M}$ Expected?
reject	accept	**accept**	**accept**
accept	reject	**accept** ☑	**accept**
accept	loops	**accept**	**accept**
loops	accept	**accept** ☑	**accept**

Nondeterministic TM → Deterministic

2nd way (Sipser)

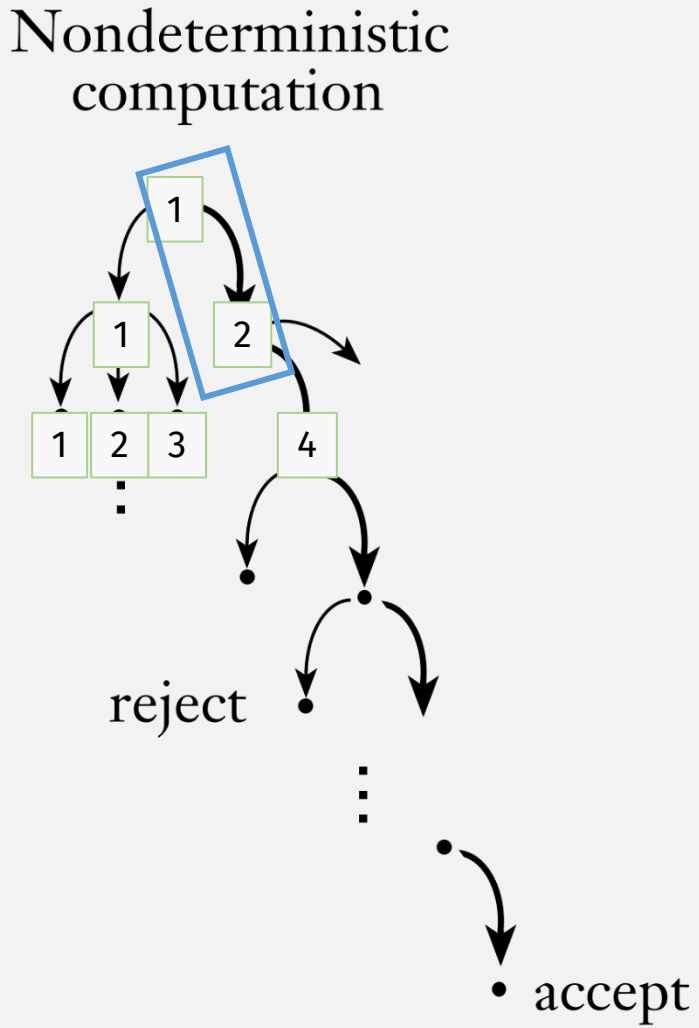
- **Simulate NTM with Det. TM:**
  - **Number the nodes at each step**
  - **Check all tree paths** (in breadth-first order)
    - 1
    - **1-1**



Nondeterministic TM → Deterministic

2nd way (Sipser)

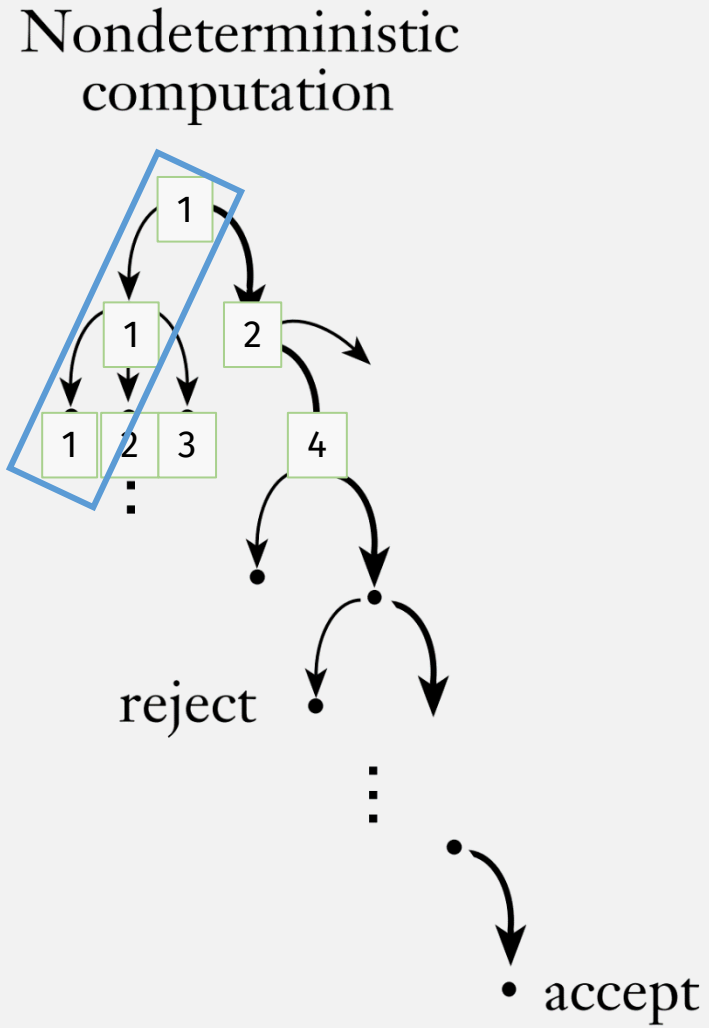
- **Simulate NTM with Det. TM:**
  - **Number the nodes at each step**
  - **Check all tree paths** (in breadth-first order)
    - **1**
    - **1-1**
    - **1-2**



Nondeterministic TM → Deterministic

2nd way (Sipser)

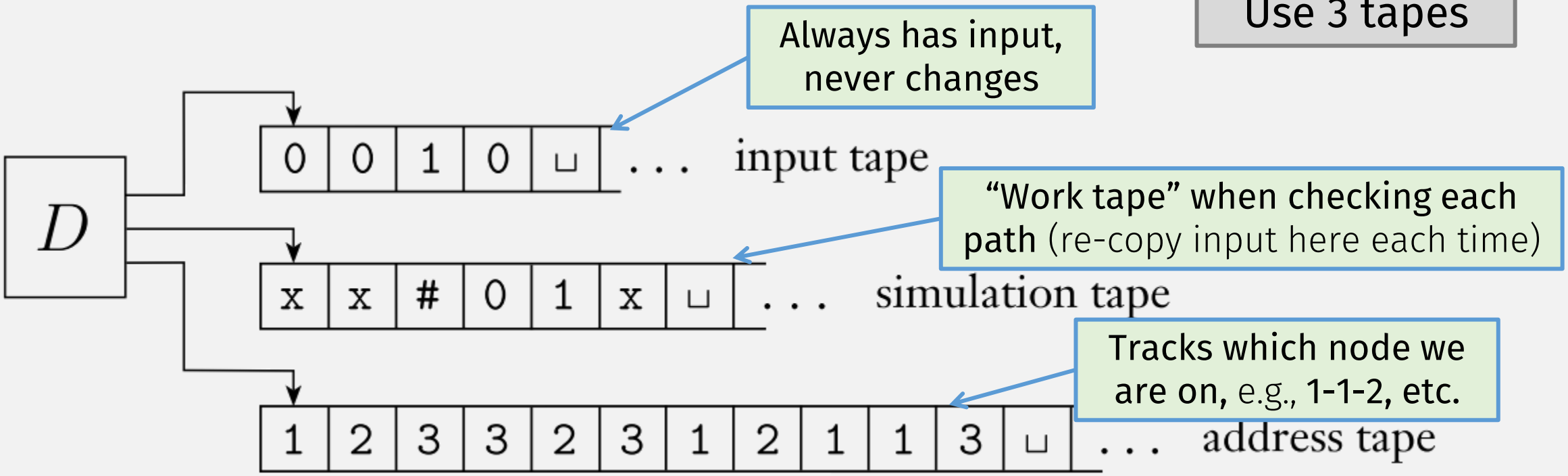
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - 1-2
    - **1-1-1**



Nondeterministic TM → Deterministic

2nd way (Sipser)

Use 3 tapes



Nondeterministic TM ⇔ Deterministic TM

☑ ⇒ IF a **deterministic TM recognizes** a **language**,
THEN a **nondeterministic TM recognizes** the **language**

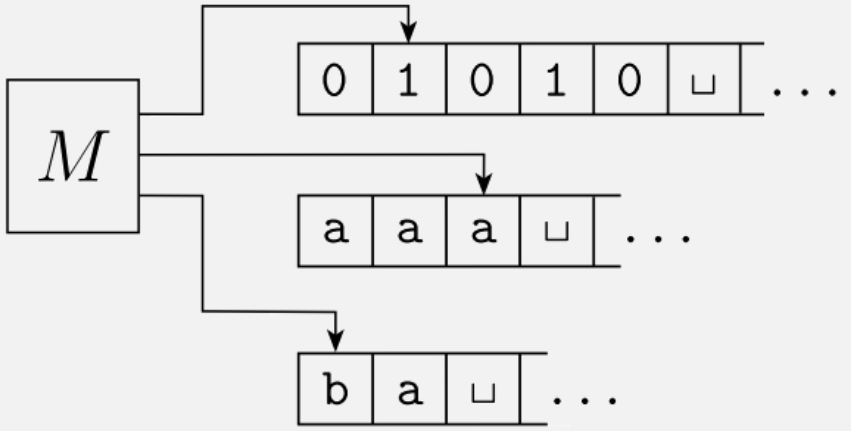
- Convert **Deterministic TM → Non-deterministic TM**

☑ ⇐ IF a **nondeterministic TM recognizes** a **language**,
THEN a **deterministic TM recognizes** the **language**

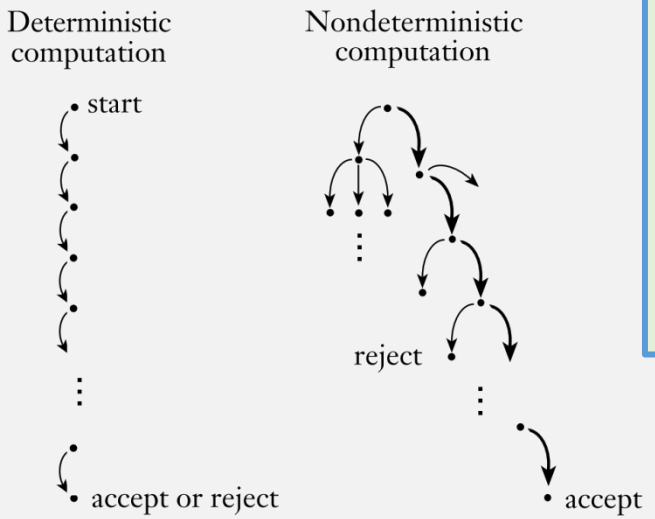
- Convert **Nondeterministic TM → Deterministic TM**

■

☑ 1. **Multi-tape TMs**



☑ 2. **Non-deterministic TMs**



We have proven: these **TM variations** are **equivalent to** deterministic, single-tape machines

Conclusion: These are All Equivalent TMs!

- Single-tape Turing Machine
- Multi-tape Turing Machine
- Non-deterministic Turing Machine