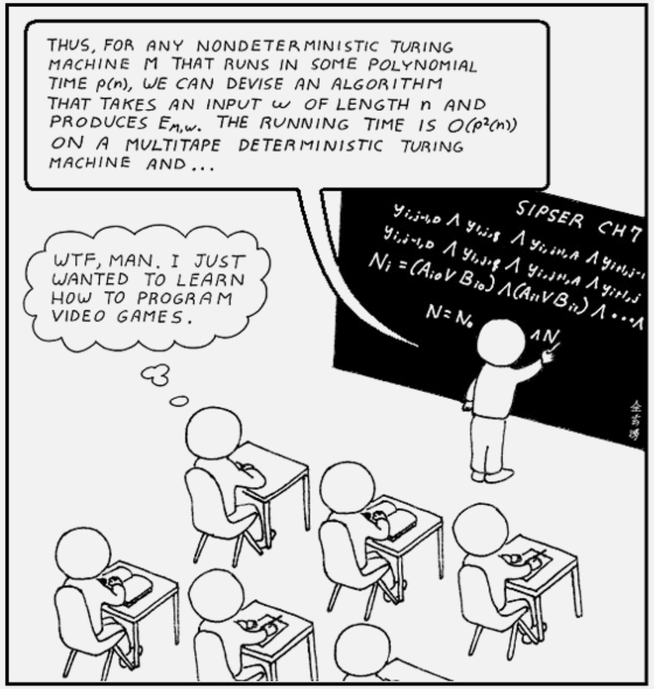
CS 420 / CS 620

Turing Machine Variants

Monday, November 3, 2025

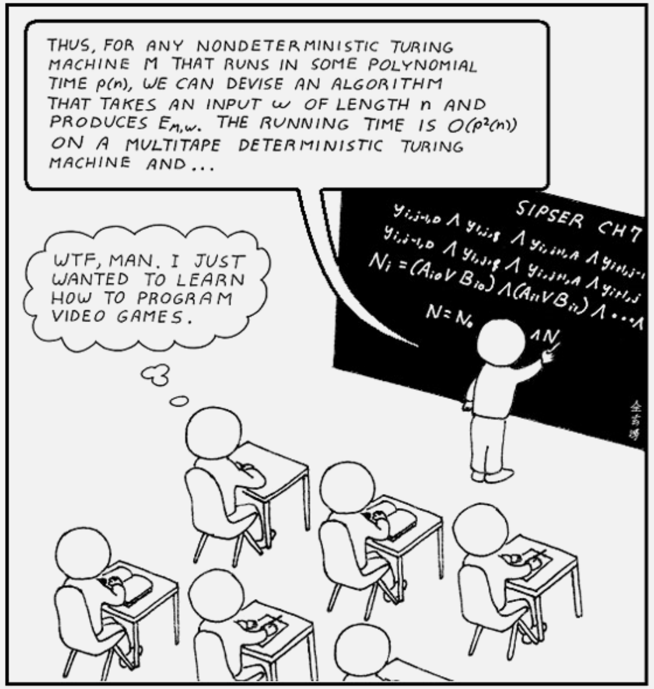
UMass Boston Computer Science



Announcements

- HW 8
  - ~~Due: Mon 11/3 12pm (noon)~~

- HW 9
  - Out: Mon 11/3 12pm (noon)
  - Due: Mon 11/10 12pm (noon)



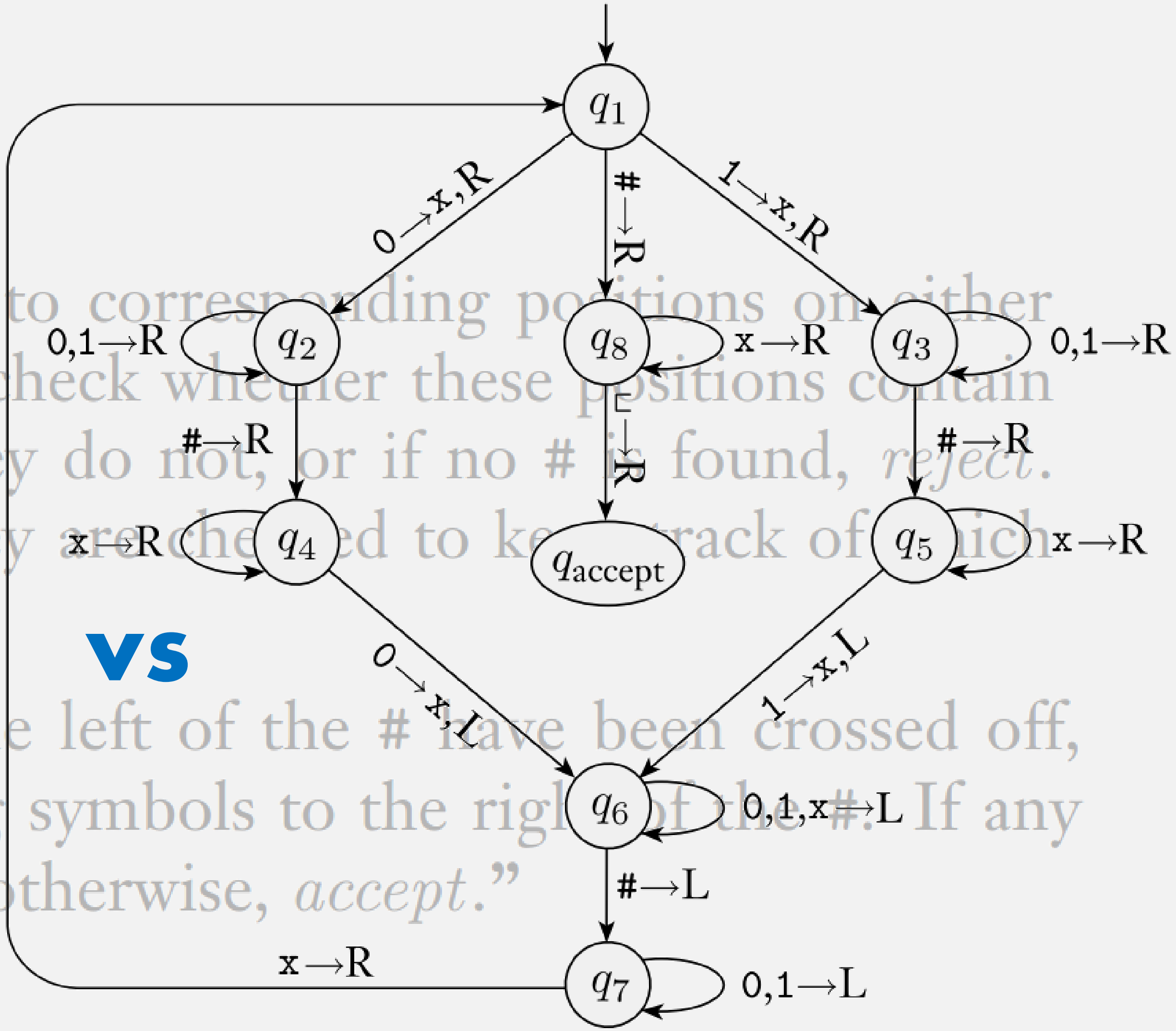
Last Time

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*."

vs



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… …he 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 are <u>not</u> a "do whatever" card, some rules:

1. TMs and input strings must be <u>named</u> (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 ..., else ..."
    - 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 "<u>step</u>" represents a <u>finite</u> # of low-level δ transitions
    - So one step cannot run forever
    - E.g., can't say "try all numbers" as a "step"

$M_1 =$ "On input string w:

$M =$ "On input w
1. Simulate B on input w.
2. If simulation ends in accept state,

$N =$ "On input $\langle B, w\rangle$, where B is an NFA and w is a string:
1. Convert NFA B to an equivalent DFA C, using the procedur this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w\rangle$.

$S =$ "On input w
1. Construct the following TM M_2.

$M_2 =$ "On input x:

Non-halting Turing Machines (TMs)

IMPORTANT SLIDE

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

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

- 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.**

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.

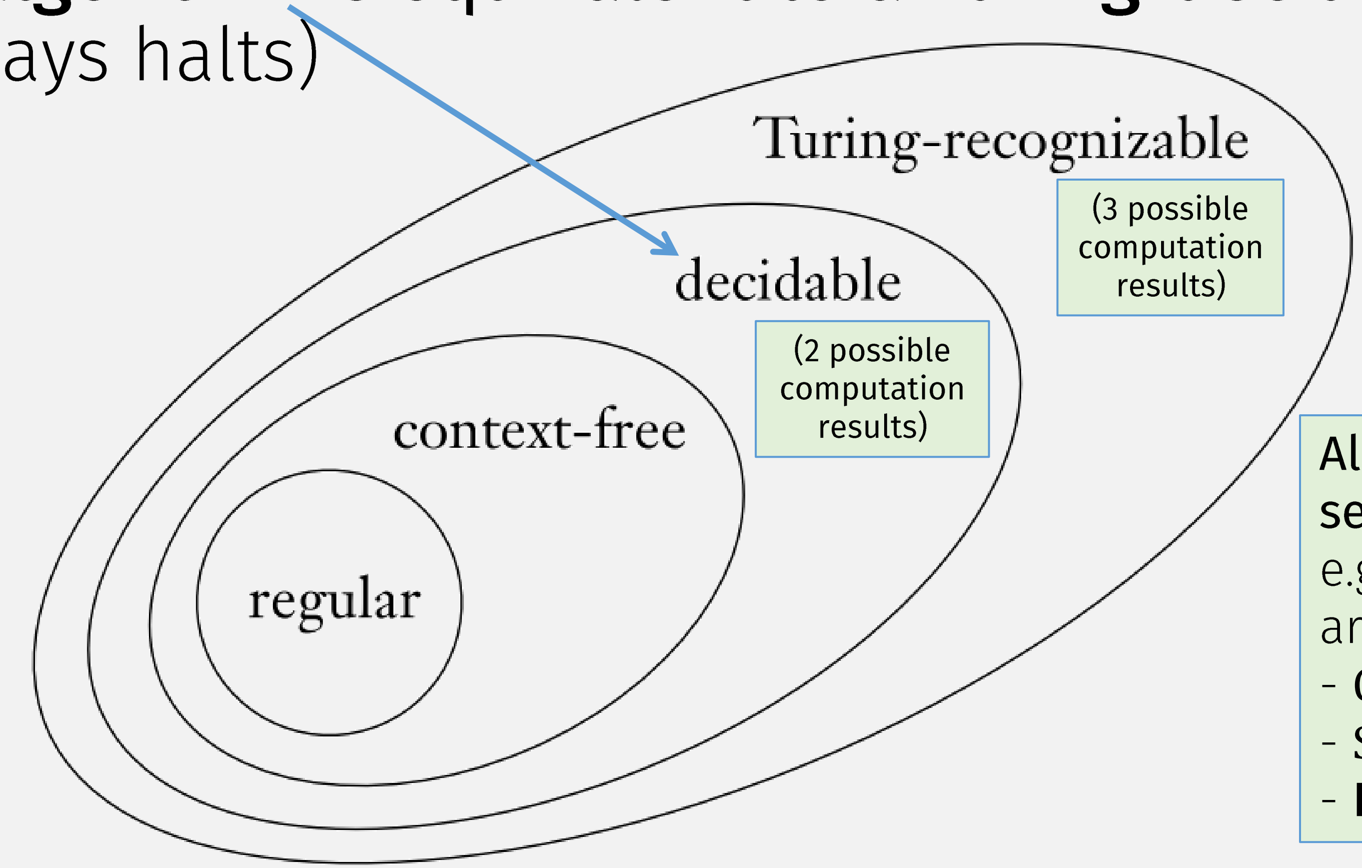
(3 possible computation results)

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it.

(2 possible computation results)

Formal Definition of an "Algorithm"

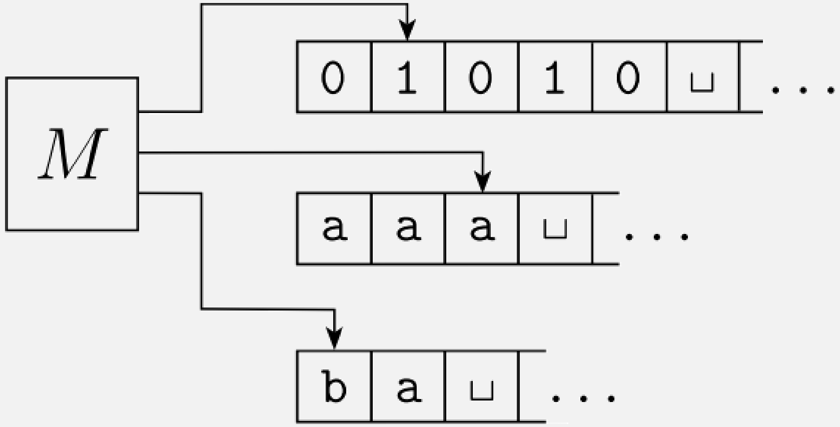
- An **algorithm** is equivalent to a **Turing-decidable** Language (always halts)



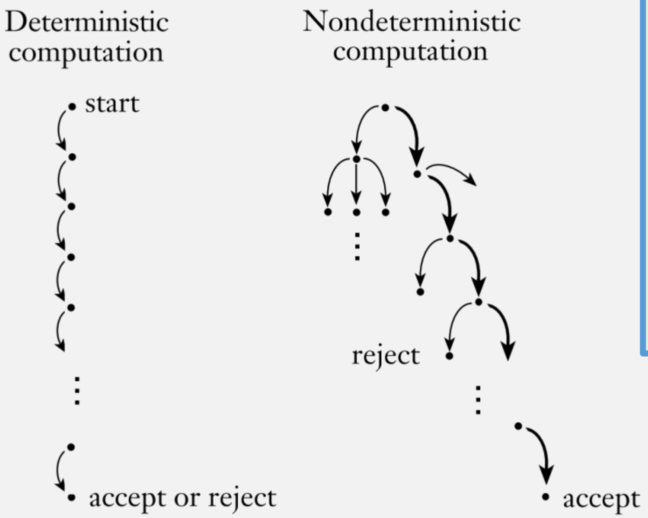
All functions we have defined this semester are **algorithms!**
e.g., all our conversion functions are **decider** TMs!!
- $\text{CONVERT}_{\text{DFA-NFA}}$
- STAR_{NFA}
- **PDA→CFG**

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

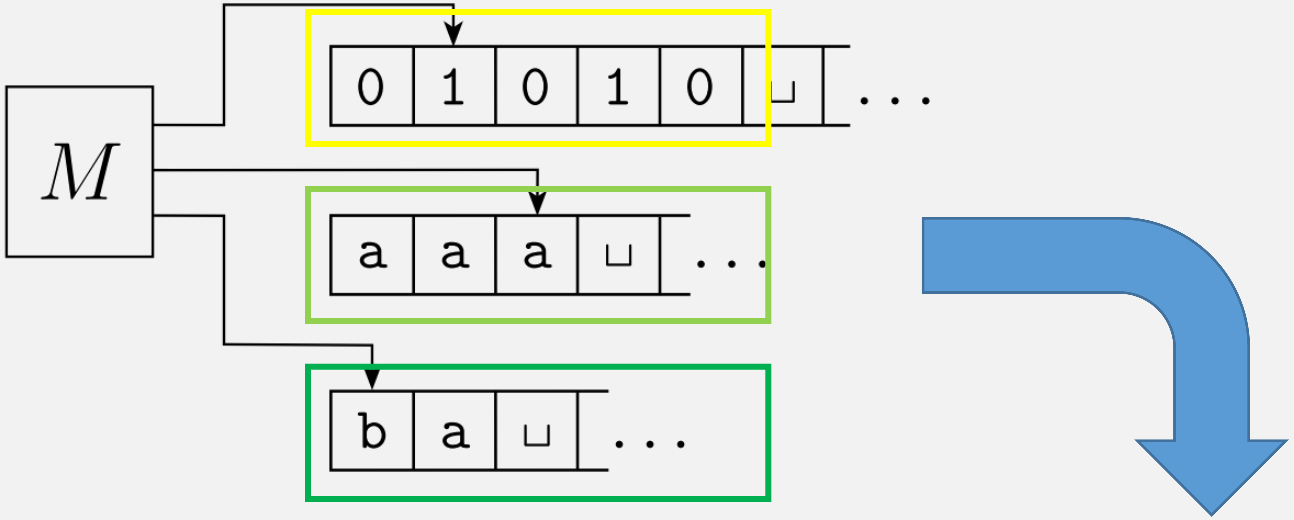
- Convert: multi-tape TM → single-tape TM

Key insight: single-tape is infinite in length!

Multi-tape TM ➔ Single-tape TM

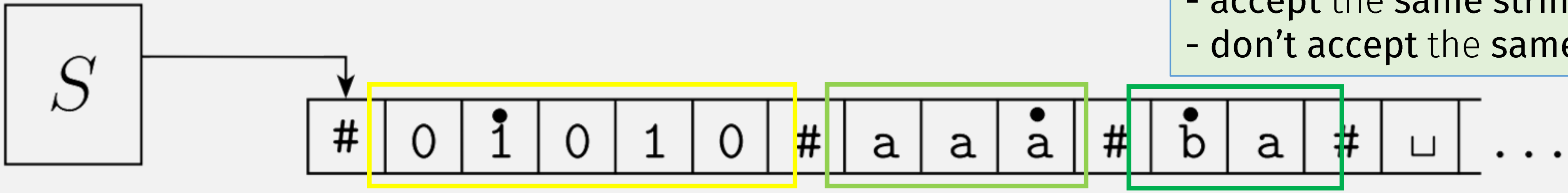
<u>Idea</u>: **Use delimiter (#)** on **single-tape** to **<u>simulate</u>** multiple **<u>tapes</u>**

- **Add “dotted” version of every char** to **<u>simulate</u>** multiple **<u>heads</u>**



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**

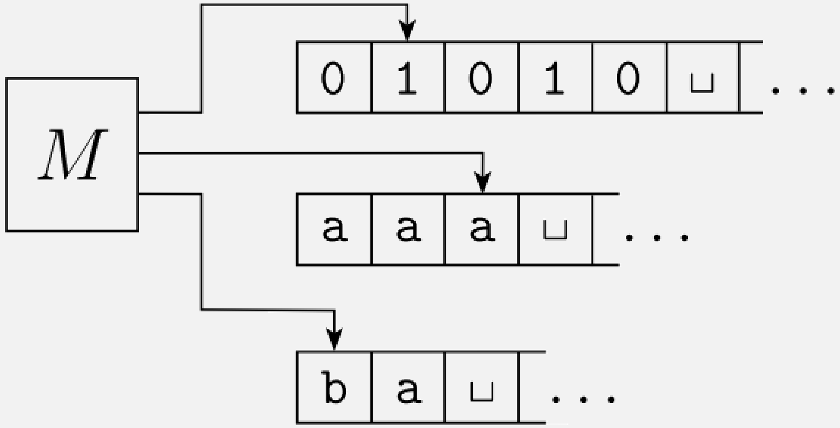
- Single-tape TM is equivalent to …
- … multi-tape TM that only uses one of its tapes

☑ ⇐ If a **multi-tape TM recognizes** a **language**,
then a **single-tape TM recognizes** the **language**

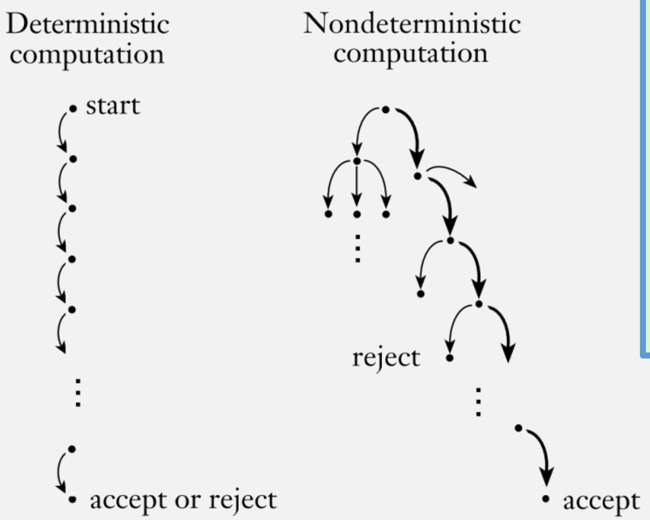
- Convert: multi-tape TM → single-tape TM

■

☑ 1. **Multi-tape TMs**



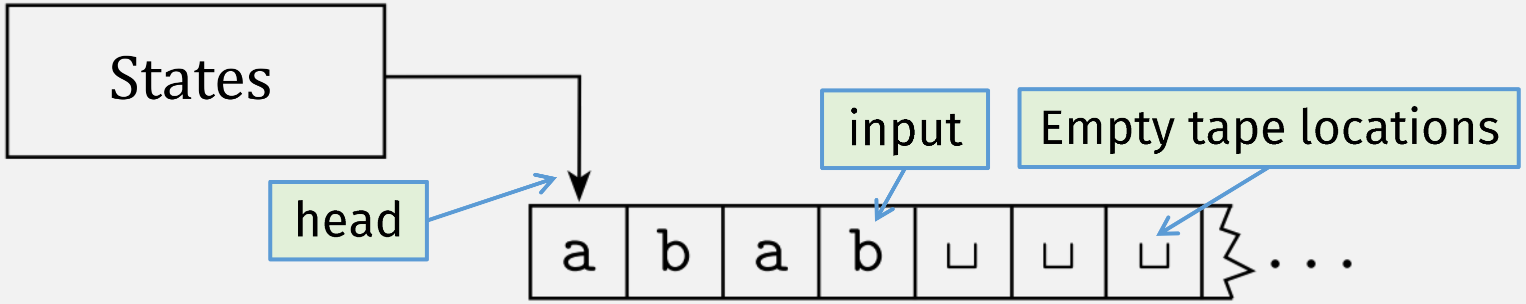
➡ 2. **Non-deterministic TMs**



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

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.

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… …he 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

Turing Machines: Formal Tuple Definition

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, 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. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,

read

write

move

5. $q_0 \in Q$ is the start state,

6. $q_{accept} \in Q$ is the accept state, and

7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Flashback: DFAs vs NFAs

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

Nondeterministic transition produces set of possible next states

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

Remember: Turing Machine Formal Definition

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, 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. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Nondeterministic Remember: Turing Machine Formal Definition

A **_Nondeterministic Turing Machine_** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, 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

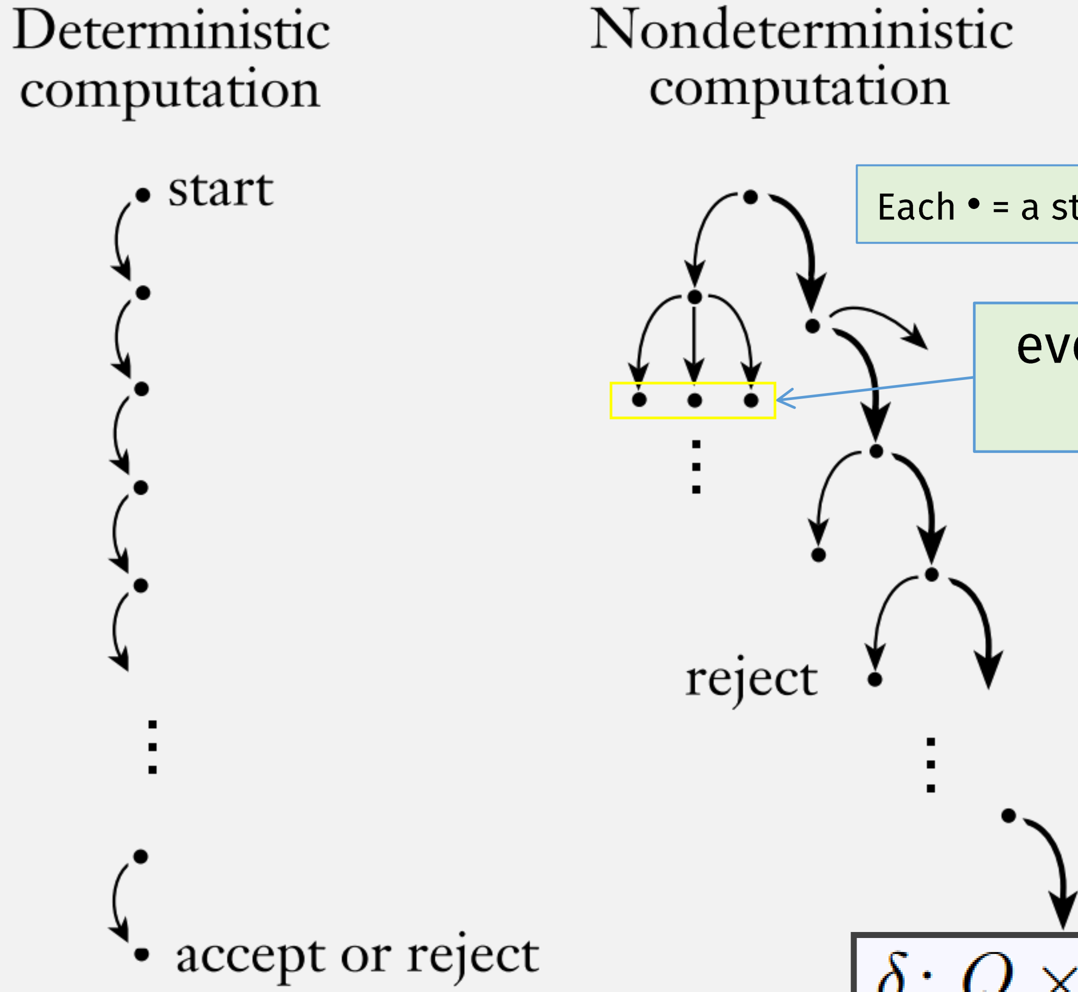
⇒ If a **deterministic TM** recognizes a **language**,
then a **non-deterministic TM** recognizes the **language**

- Convert: Deterministic TM → Non-deterministic TM …
- … change **Deterministic TM** δ **output** to: **one-element set**
  - $\delta_{\mathrm{NTM}}(q, a) = \{\delta_{\mathrm{DTM}}(q, a)\}$
  - (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



Each • = a state (for NFA)

every step can branch to set of states

What is a "state" for a TM?

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

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

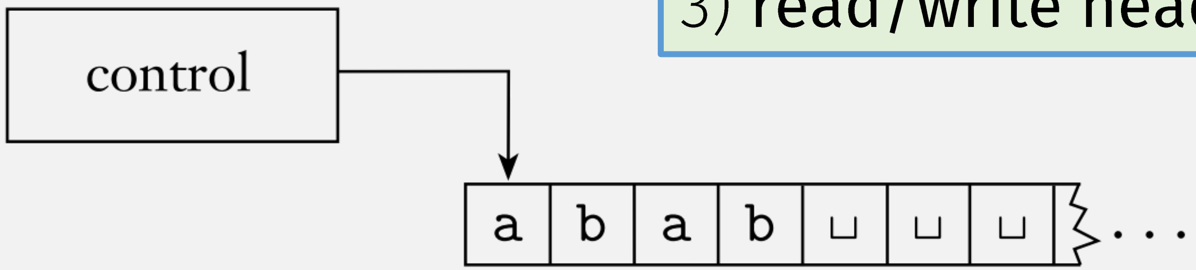
  γ = the stack contents

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

TM Configuration (ID) = ???

1) states

3) read/write head

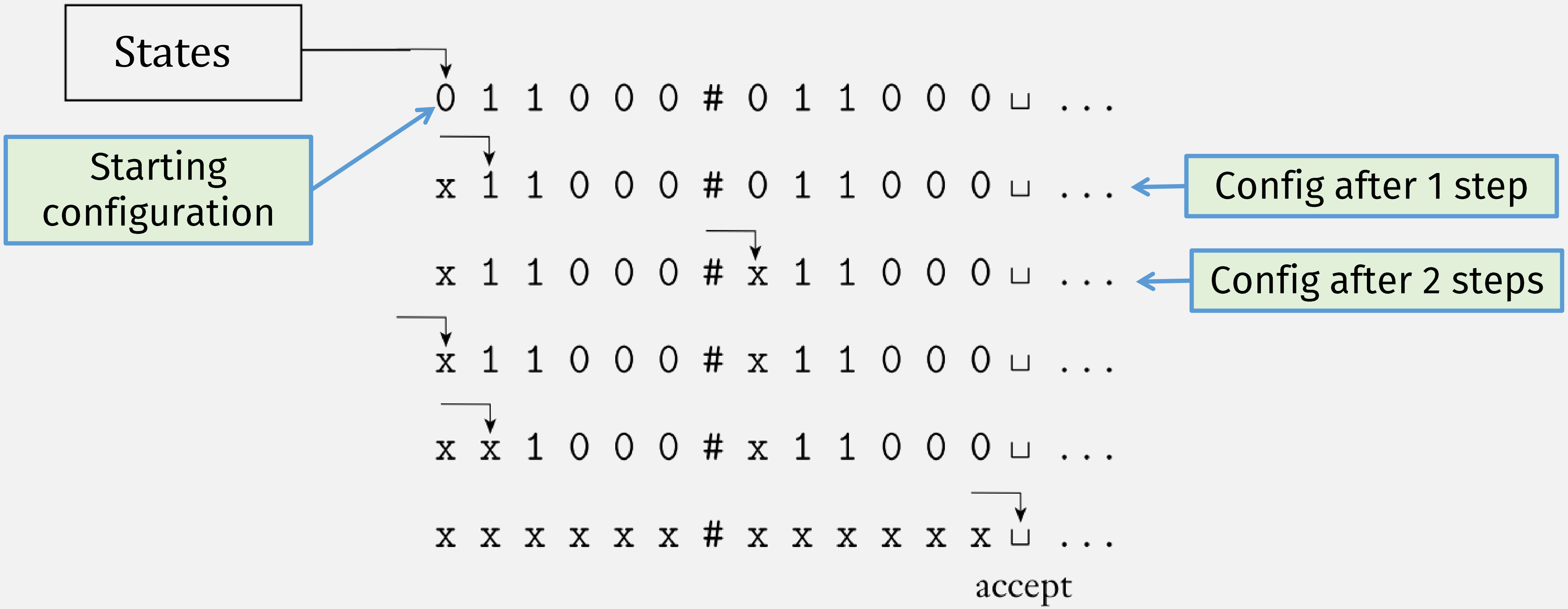


2) Tape contents

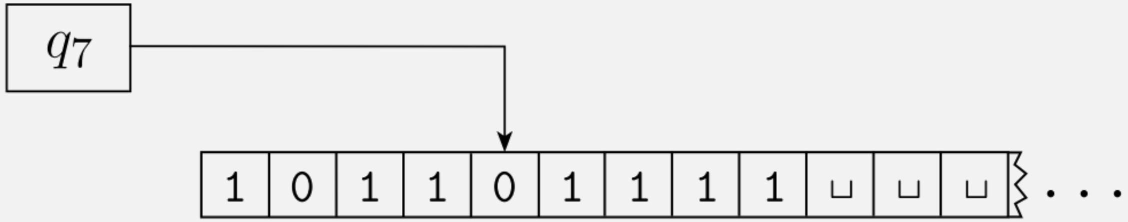
A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

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. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ 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 $q_{\text{reject}} \neq q_{\text{accept}}$.

TM Configuration = State + Head + Tape



TM Configuration = State + Head + Tape



$$1011q_7 01111$$

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

1st char after state is current head position

TM Computation, Formally

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

Single-step

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

[head] [Next config (head moved past written char)]

if $q_1, q_2 \in Q$

$\delta(q_1, \mathbf{a}) = (q_2, \mathbf{x}, \mathrm{R})$ [write] [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] [(wrote $\mathbf{x}$ and) head moved left]

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

Multi-step

- Base Case

$I \overset{*}{\vdash} I$ for any ID I

- Recursive Case

$I \overset{*}{\vdash} J$ if there exists some ID K such that $I \vdash K$ and $K \overset{*}{\vdash} J$

Edge cases:

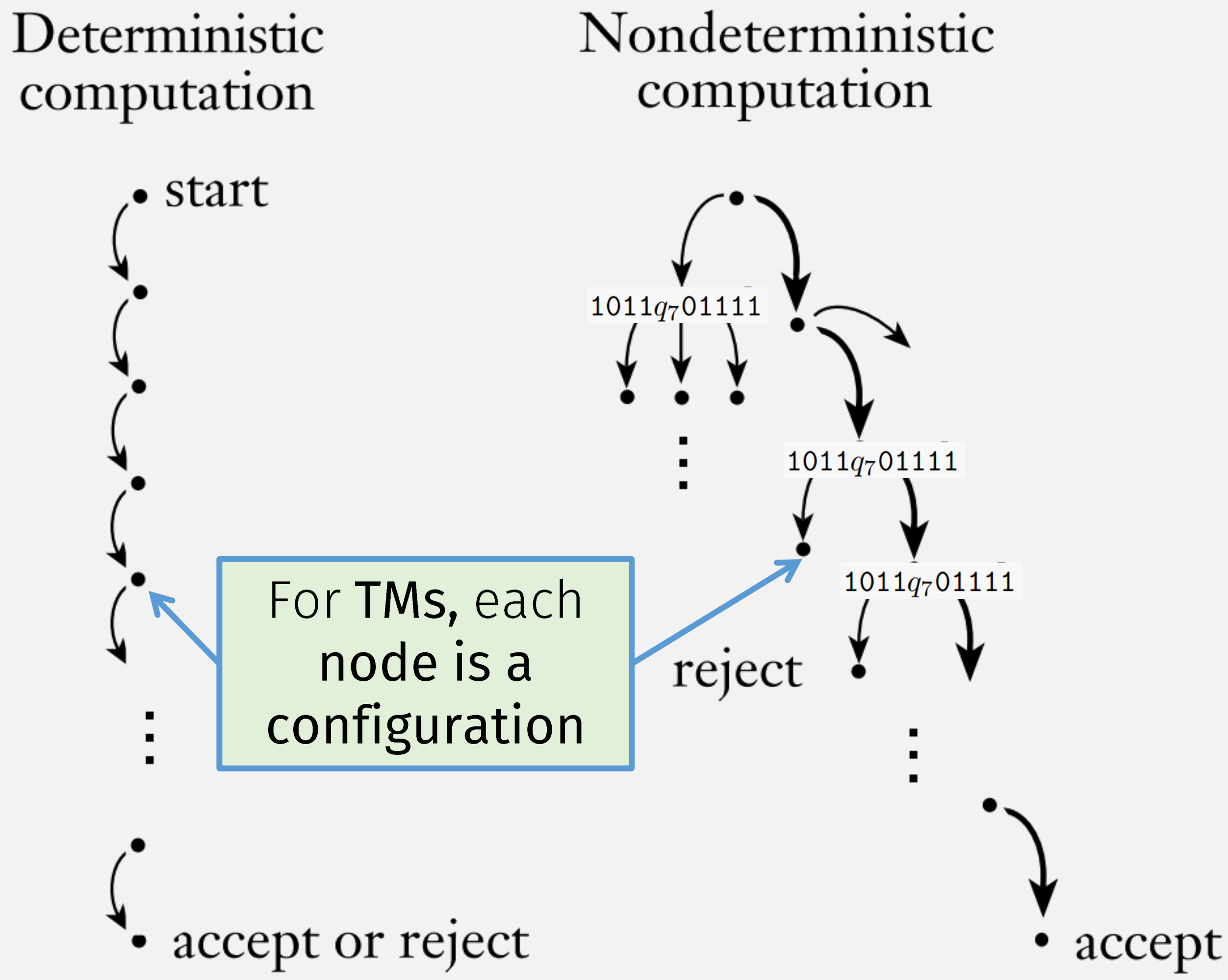
$q_1 \mathbf{a} \beta \vdash q_2 \mathbf{x} \beta$ if $\delta(q_1, \mathbf{a}) = (q_2, \mathbf{x}, \mathrm{L})$ (L move, when already at leftmost cell)

[Head stays at leftmost cell]

$\alpha q_1 \vdash \alpha \sqcup q_2$ if $\delta(q_1, \sqcup) = (q_2, \sqcup, \mathrm{R})$ (R move, when at rightmost filled cell)

[Add blank symbol to config]

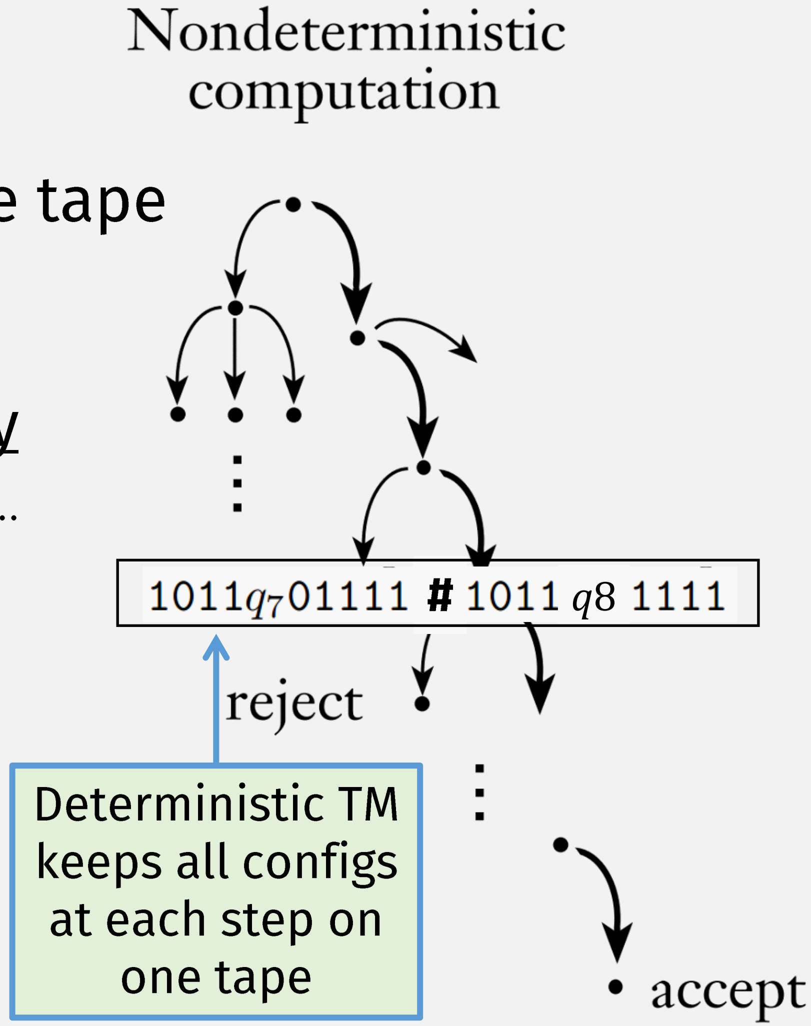
Nondeterminism in TMs



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, concurrently
    - 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 concurrently?

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:

$\boldsymbol{M}$ = 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 $\boldsymbol{M}$

"in the lang" that we want M to recognize

M_1	M_2	$\boldsymbol{M}$	$\boldsymbol{M}$ 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

Just an analogy: "*calling*" a TM actually requires "computing" how it computes … $\alpha q_1 \mathbf{a} \beta \vdash \alpha \mathbf{x} q_2 \beta$

… 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:

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:

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

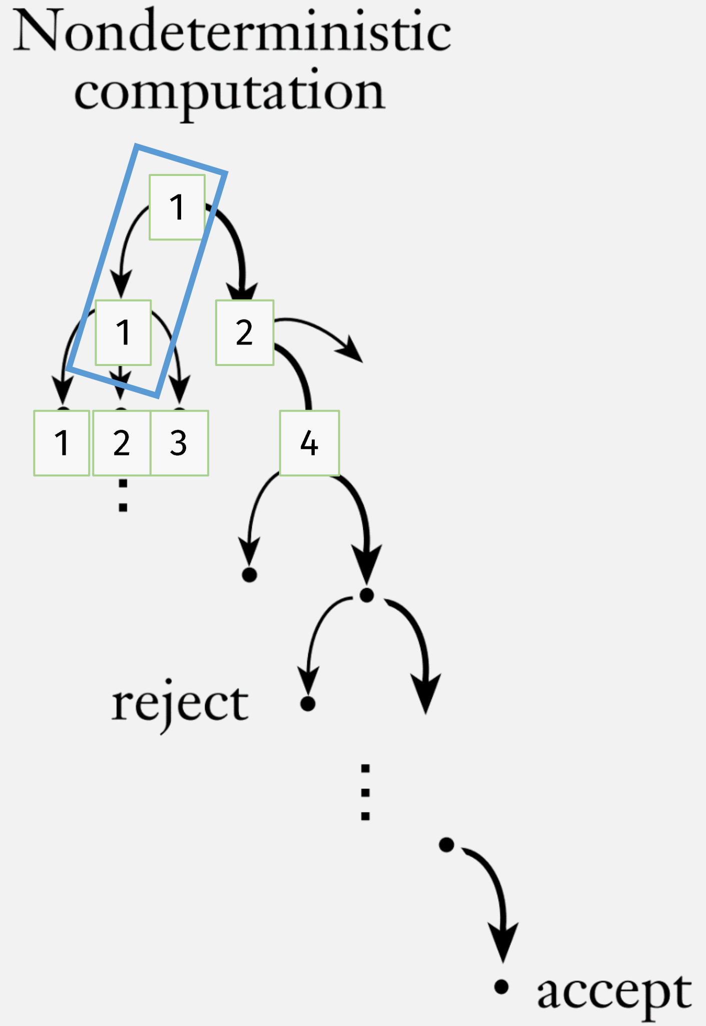
1. Call M_1 and M_2, each with x, concurrently, i.e.,
   a) Run M_1 with x for 1 step; **accept** x if M_1 accepts
   b) Run M_2 with x for 1 step; **accept** x if M_2 accepts
   c) Repeat

M_1	M_2	M	M Expected?
reject	accept	**accept**	**accept**
accept	reject	**accept** ☑	**accept**
accept	loops	**accept**	**accept**
loops	accept	**accept** ☑	**accept**

Nondeterministic TM → Deterministic

2nd way
(Sipser)

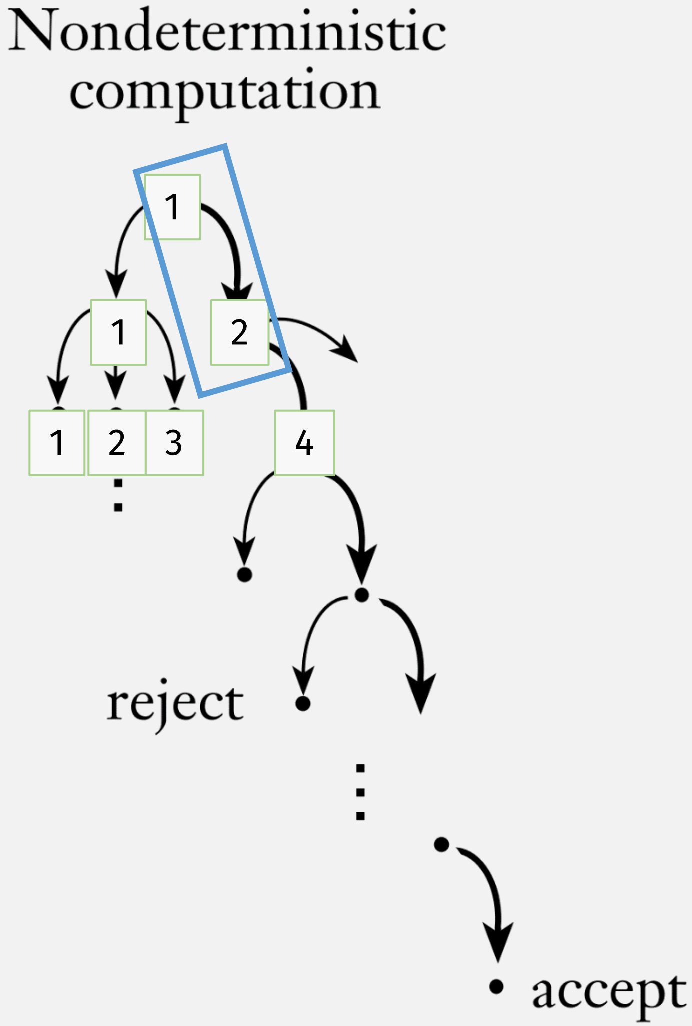
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - **1-1**



Nondeterministic TM → Deterministic

2nd way
(Sipser)

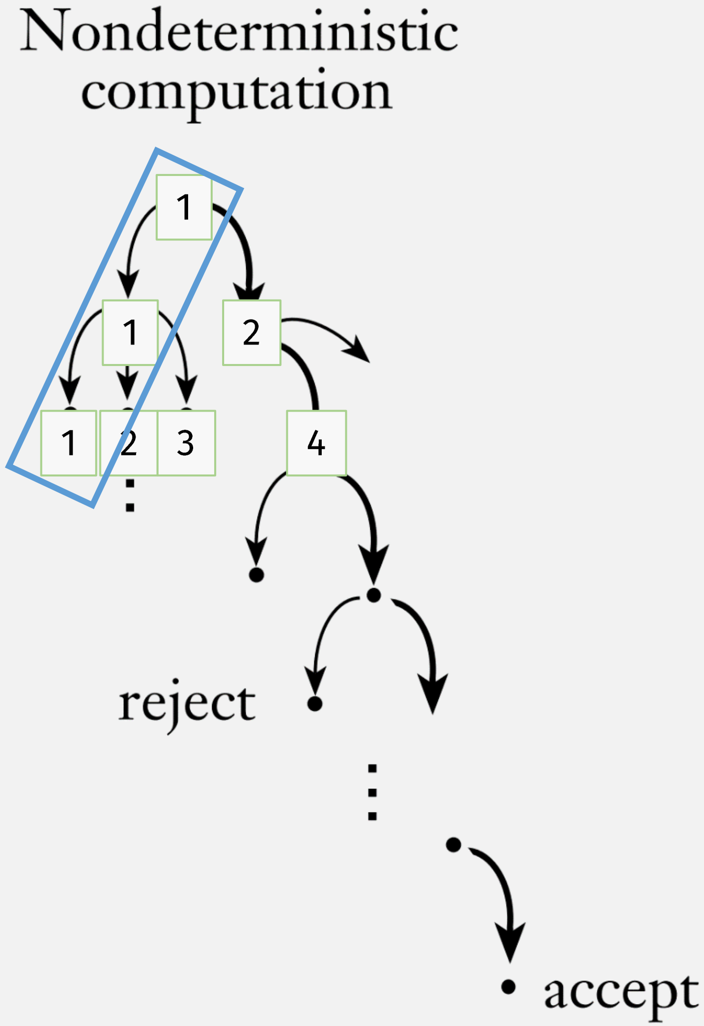
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - **1-2**



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

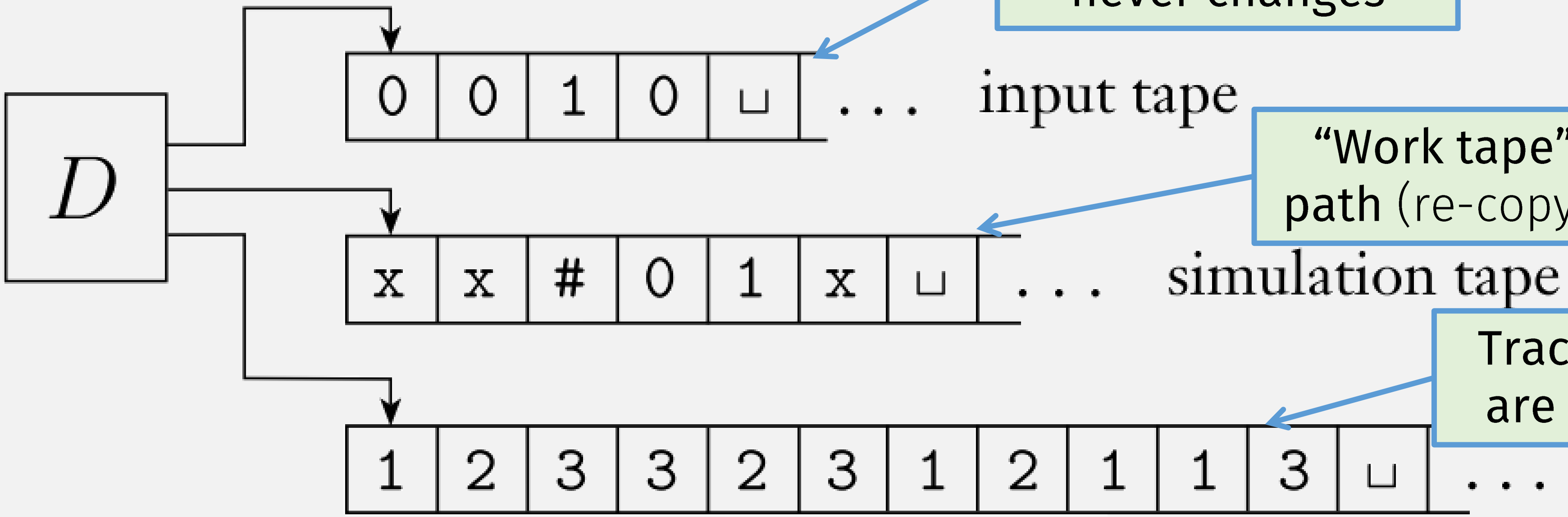
2nd way
(Sipser)

Use 3 tapes

Always has input, never changes

“Work tape” when checking each path (re-copy input here each time)

Tracks which node we are on, e.g., 1-1-2, etc.



Nondeterministic TM ⇔ Deterministic TM

☑ ⇒ If a **deterministic TM recognizes** a **language**, then a **nondeterministic TM recognizes** the **language**

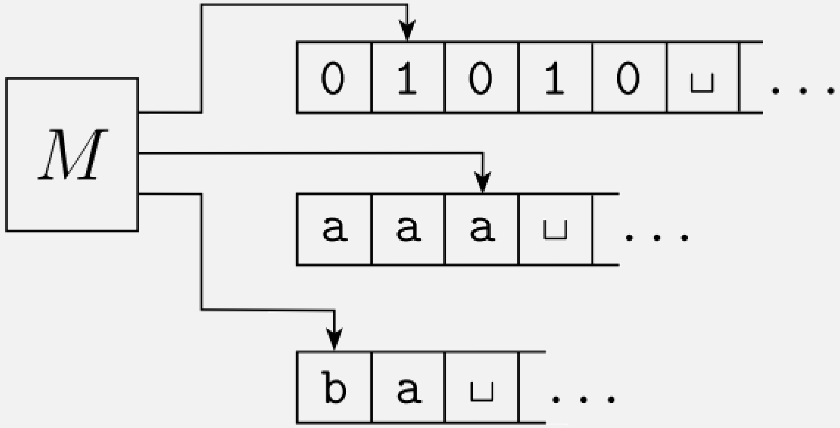
- Convert Deterministic TM → Non-deterministic TM

☑ ⇐ If a **nondeterministic TM recognizes** a **language**, then a **deterministic TM recognizes** the **language**

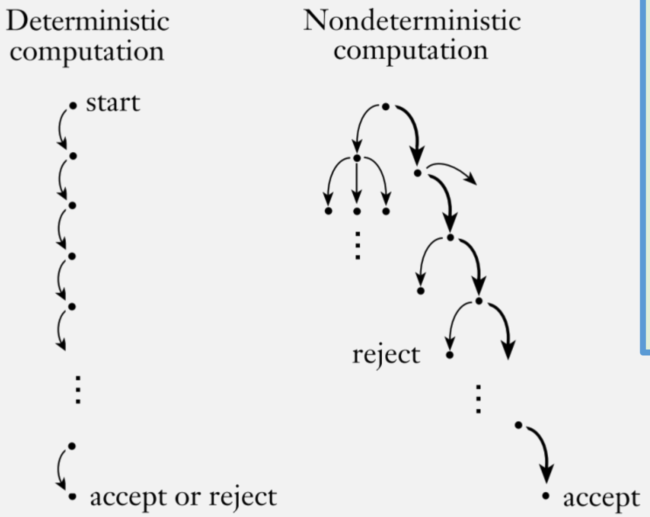
- Convert Nondeterministic TM → Deterministic TM

∎

☑ **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