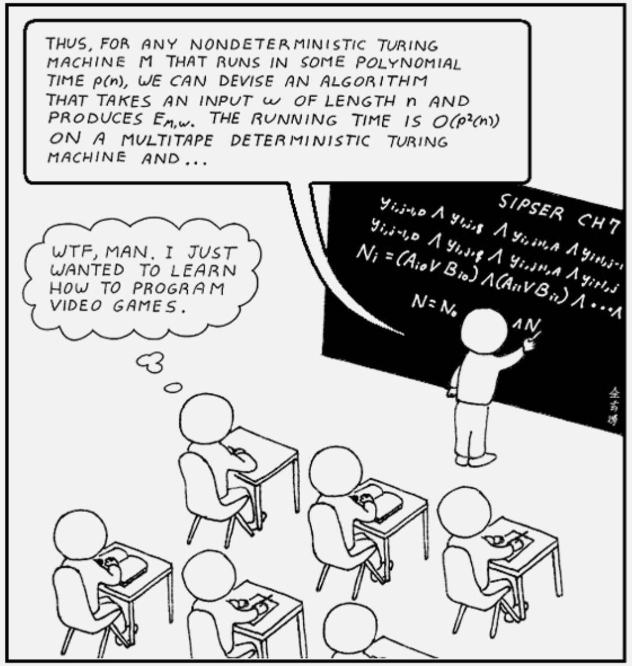
UMB CS622

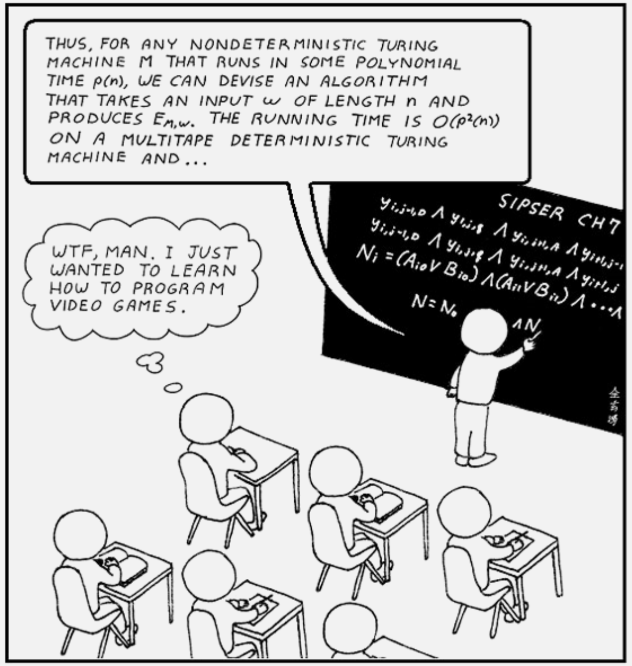
Turing Machine Variations

Wednesday, April 3, 2024



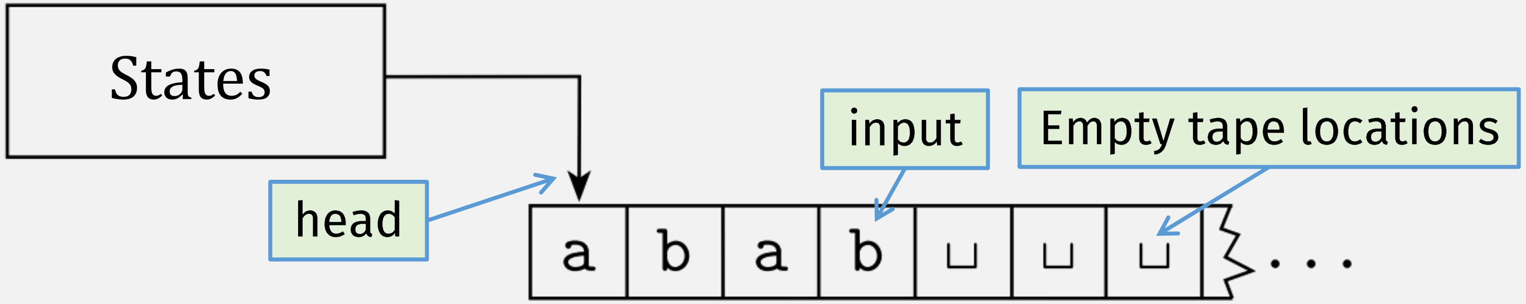
Announcements

- HW 7 out
  - due Mon 4/8 12pm noon EST



Last Time: Turing Machines

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



- On a transition, "head" can move left or right <u>1 step</u>

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

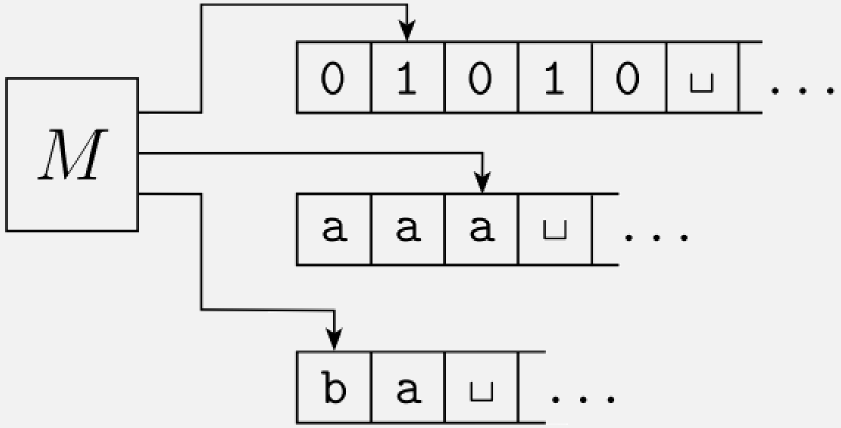
Turing Machines: Formal Tuple Definition

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

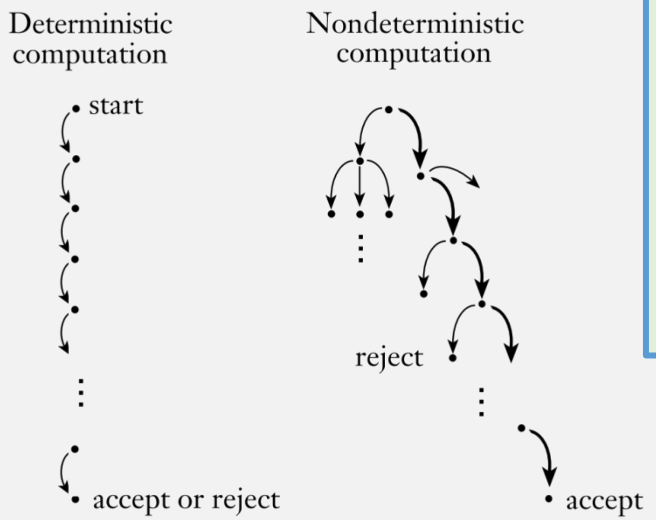
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta \colon Q \times \underbrace{\Gamma}_{\text{read}} \longrightarrow Q \times \underbrace{\Gamma}_{\text{write}} \times \underbrace{\{\mathrm{L}, \mathrm{R}\}}_{\text{move}}$ 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 Variations

1. **Multi-tape TMs**



2. **Non-deterministic TMs**



We will prove that 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?)

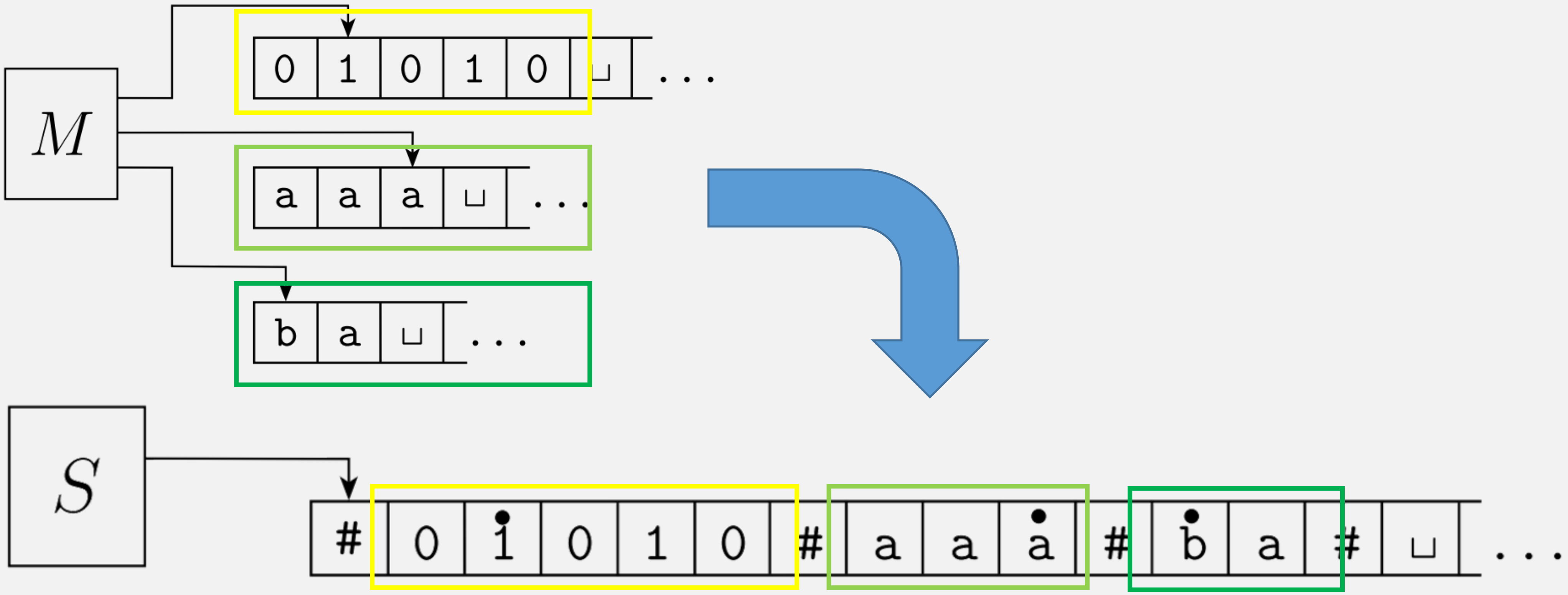
⇐ If a multi-tape TM recognizes a language,
then a single-tape TM recognizes the language

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

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



Theorem: Single-tape TM ⇔ Multi-tape TM

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

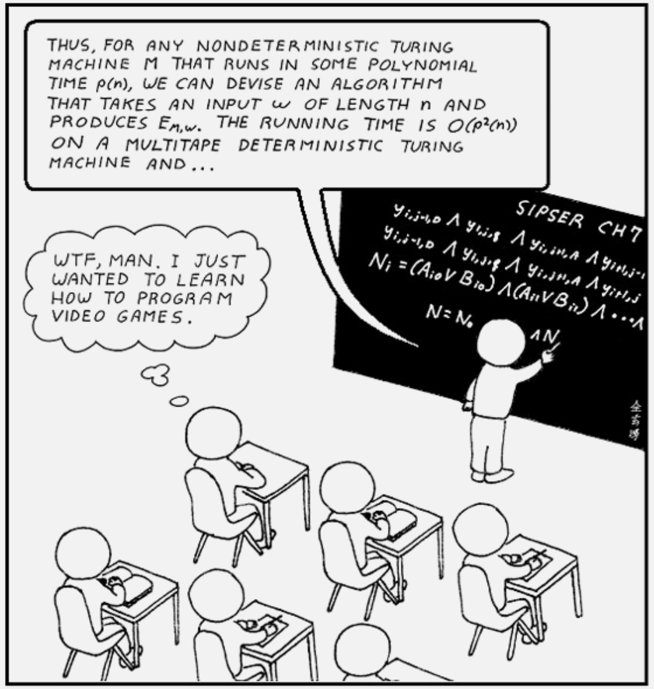
- Single-tape TM is equivalent to …
- … multi-tape TM that only uses one of its tapes

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

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

∎

Nondeterministic TMs



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. $\boxed{\delta : Q \times \Sigma \longrightarrow Q}$ 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 <u>set</u> 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. $\boxed{\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)}$ 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, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, 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 $q_{\text{reject}} \neq q_{\text{accept}}$.

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. $\cancel{\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}}$ $\boxed{\delta\colon Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})}$
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

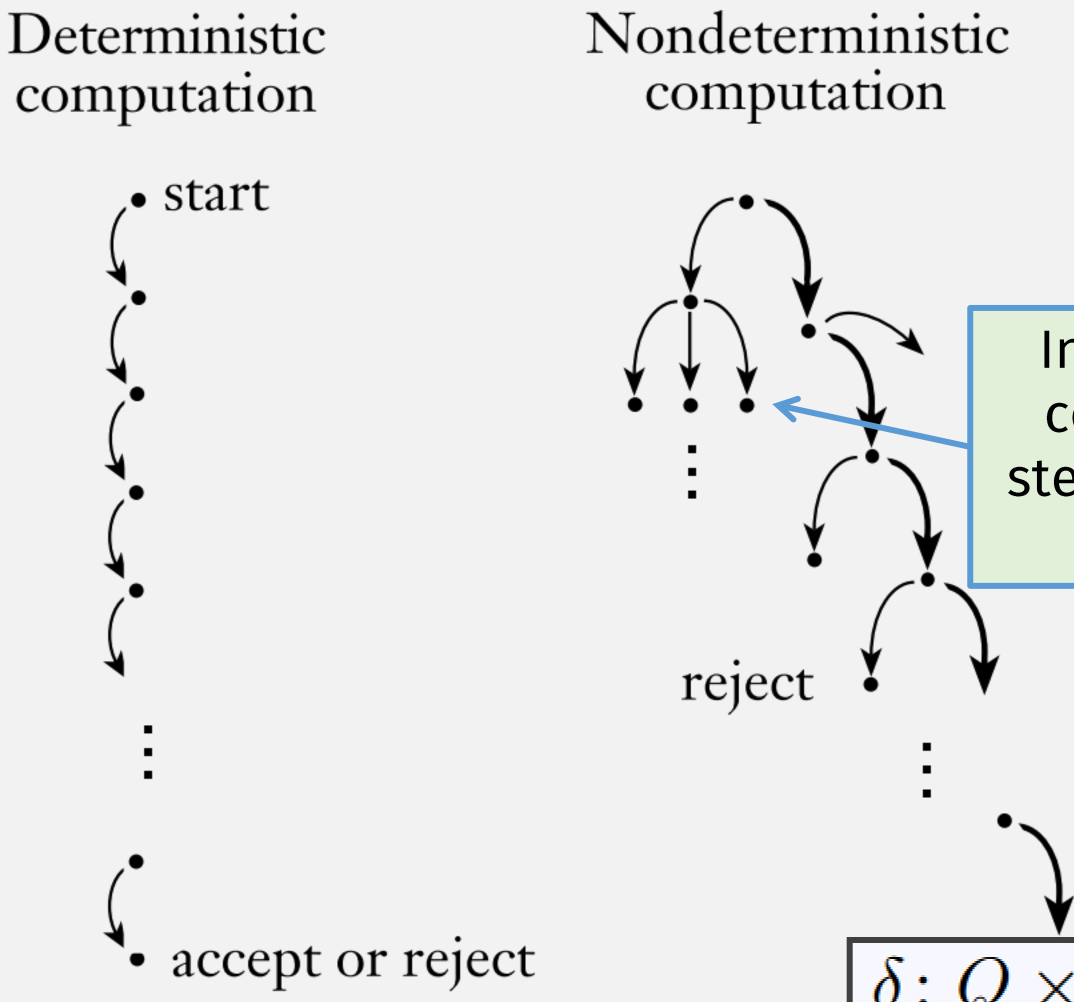
⇒ If a **deterministic TM** recognizes a language,
then a **non-deterministic TM** recognizes the language

- Convert: Deterministic TM → Non-deterministic TM …
- … change **Deterministic TM δ fn output to a one-element set**
  - $\delta_{\mathrm{ntm}}(q, a) = \{\delta_{\mathrm{dtm}}(q, a)\}$
  - (just like conversion of DFA to NFA --- HW 3, Problem 1)
- **DONE!**

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

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

Review: Nondeterminism



In nondeterministic computation, every step can branch into a set of "states"

What is a "state" for a TM?

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

Flashback: PDA Configurations (IDs)

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

- 3 components (q, w, γ) :

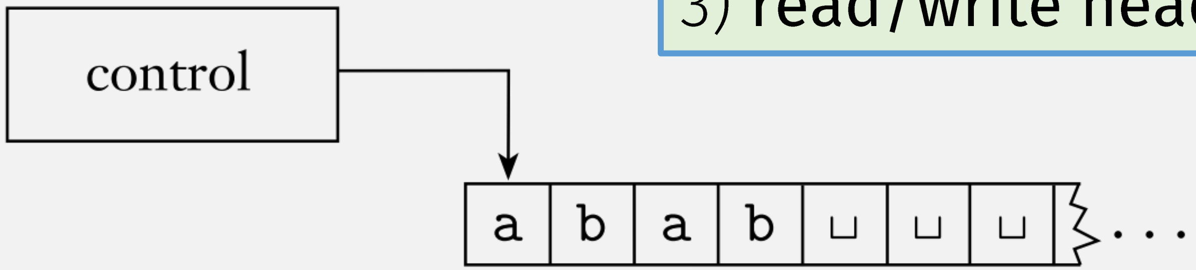
  q = the current state
  w = the remaining input string
  γ = the stack contents

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

TM Configuration (ID) = ???

1) states

3) read/write head

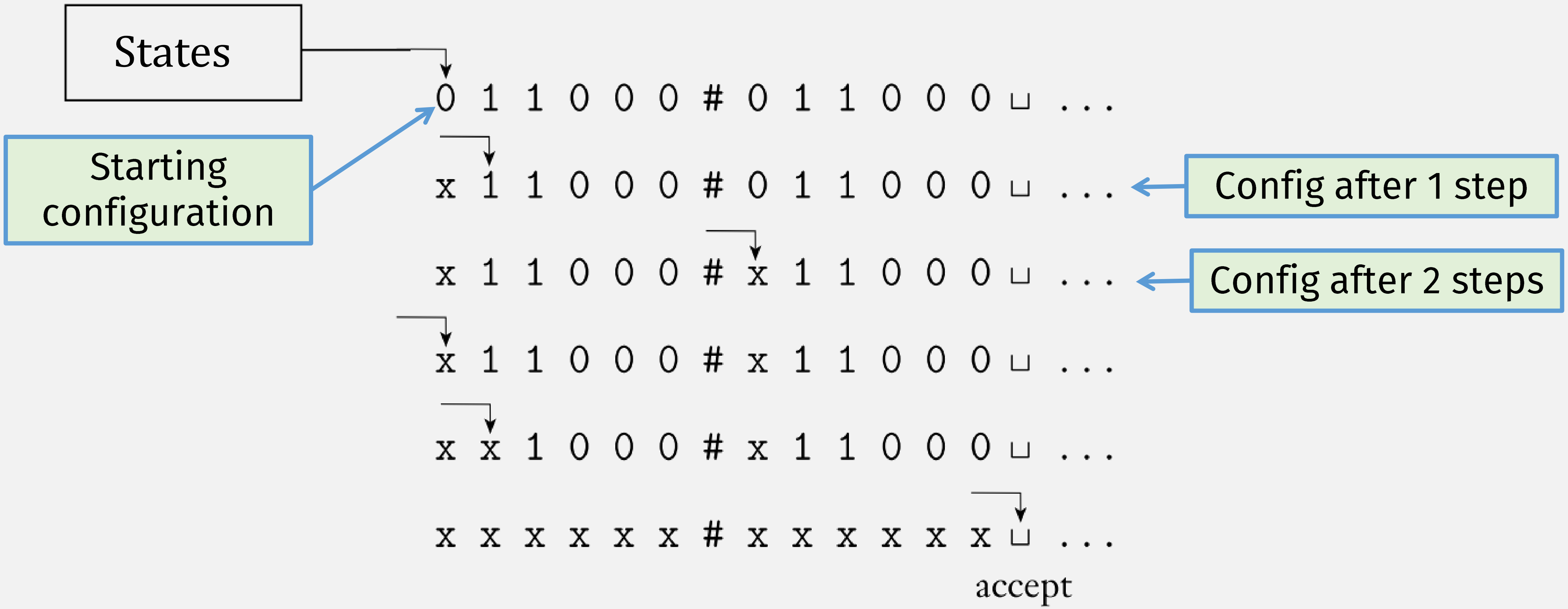


2) Tape contents

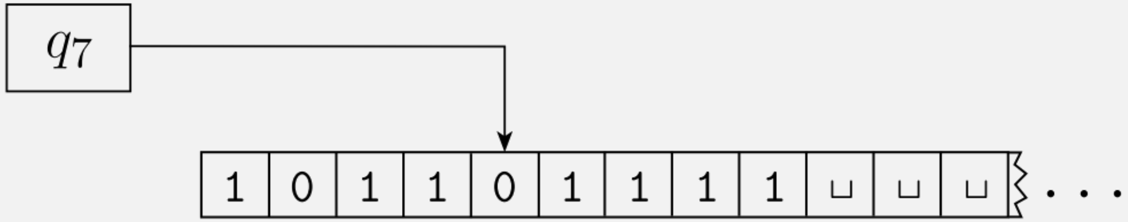
A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

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. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ 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



$$1011q_7 01111$$

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

head

Next config (head moved past written char)

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

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 $\mathbf{x}$ and) head moved left

Multi-step

- Base Case

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

- Recursive Case

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

<u>Edge cases</u>:

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

Head stays at leftmost cell

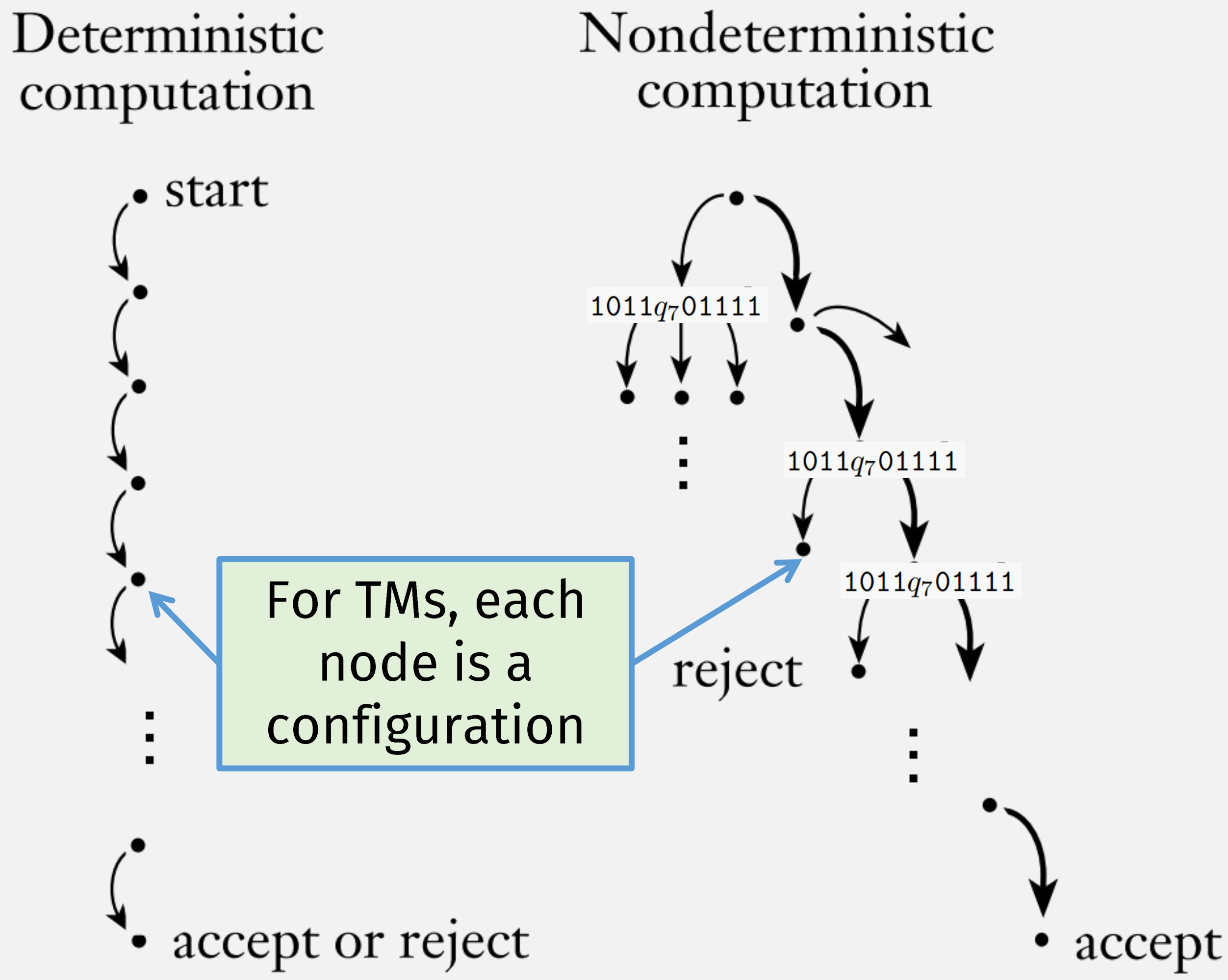
(L move, when already at leftmost cell)

$\alpha q_1 \vdash \alpha \_ q_2$ if $\delta(q_1, \_) = (q_2, \_, \mathrm{R})$

Add blank symbol to config

(R move, when at rightmost filled cell)

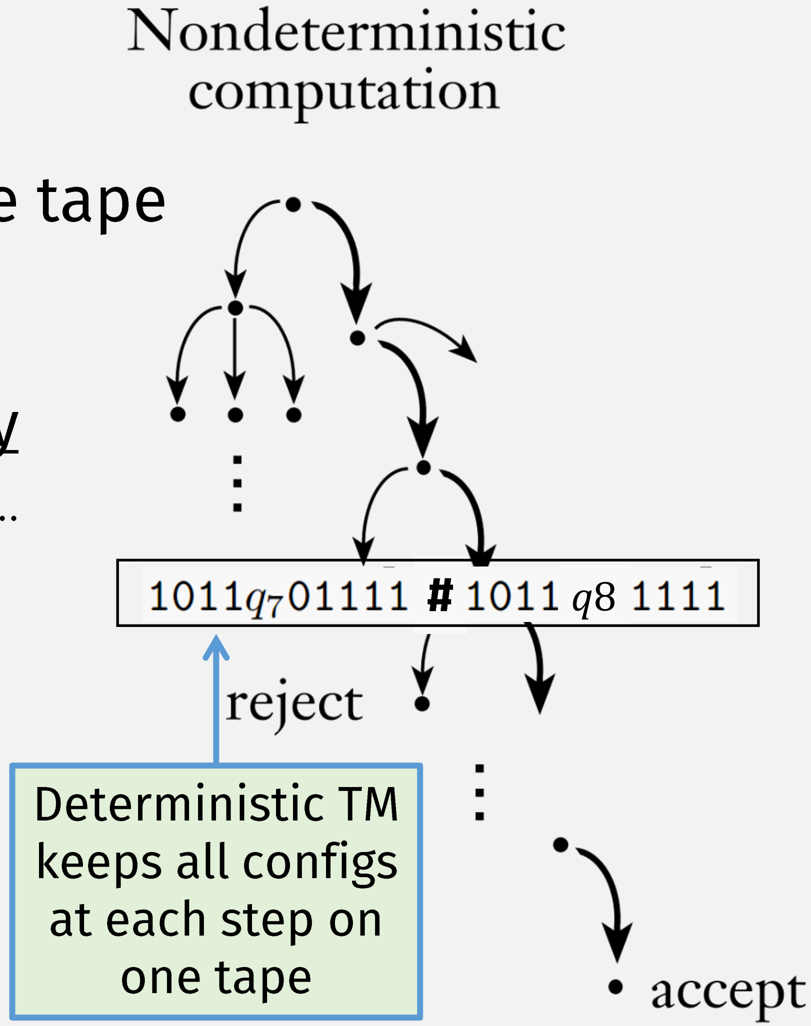
Nondeterminism in TMs



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 Results for M

M_1	M_2	$\boldsymbol{M}$
reject	accept	accept
accept	reject	accept
accept	loops	accept
loops	accept	loops

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

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*" TMs actually means "computing" its computation ...

Exercise:

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

... with concurrency!

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,

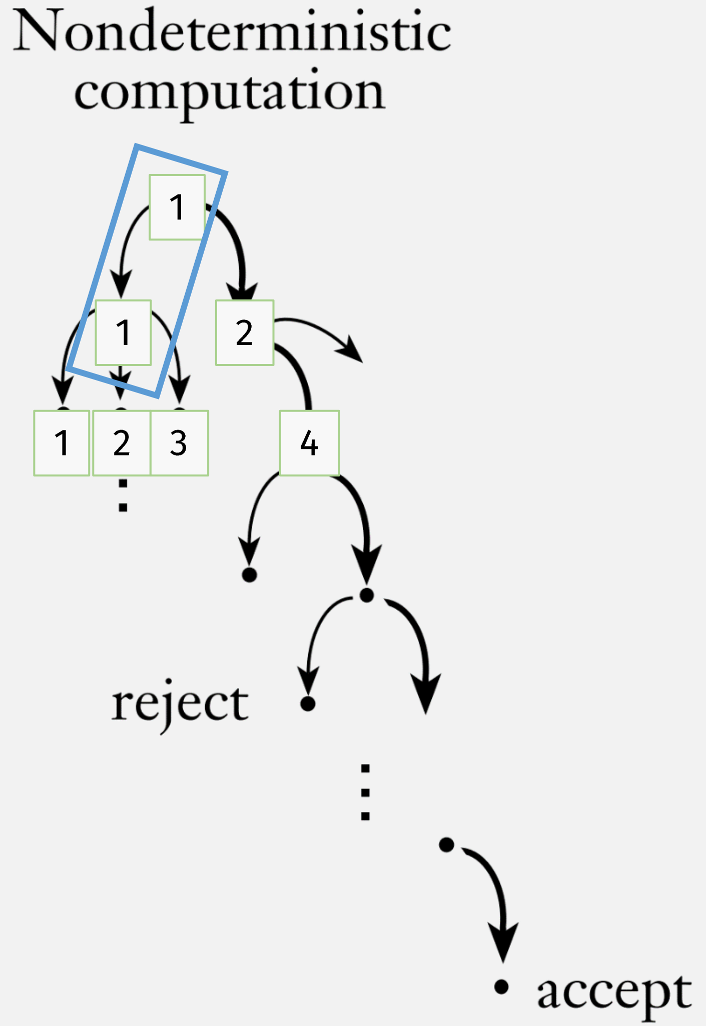
1. `Call` M_1 and M_2, each with x, concurrently, i.e.,
   a) Run M_1 with x for 1 step; **accept** x if M_1 accepts
   b) Run M_2 with x for 1 step; **accept** x if M_2 accepts
   c) Repeat

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

Nondeterministic TM → Deterministic

2nd way
(Sipser)

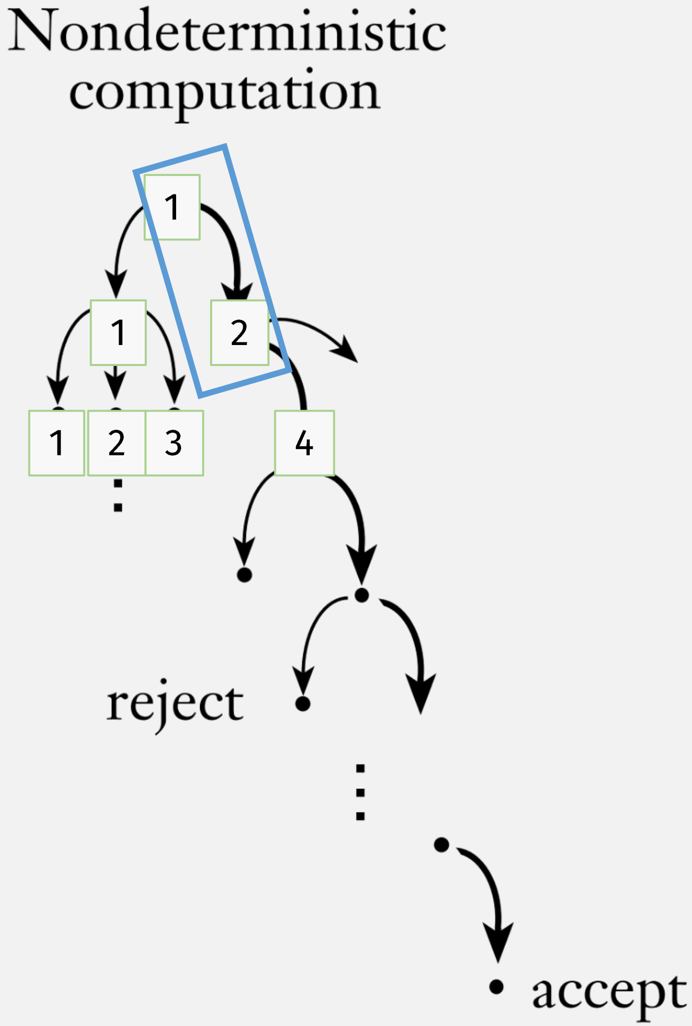
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - **1-1**



Nondeterministic TM → Deterministic

2nd way
(Sipser)

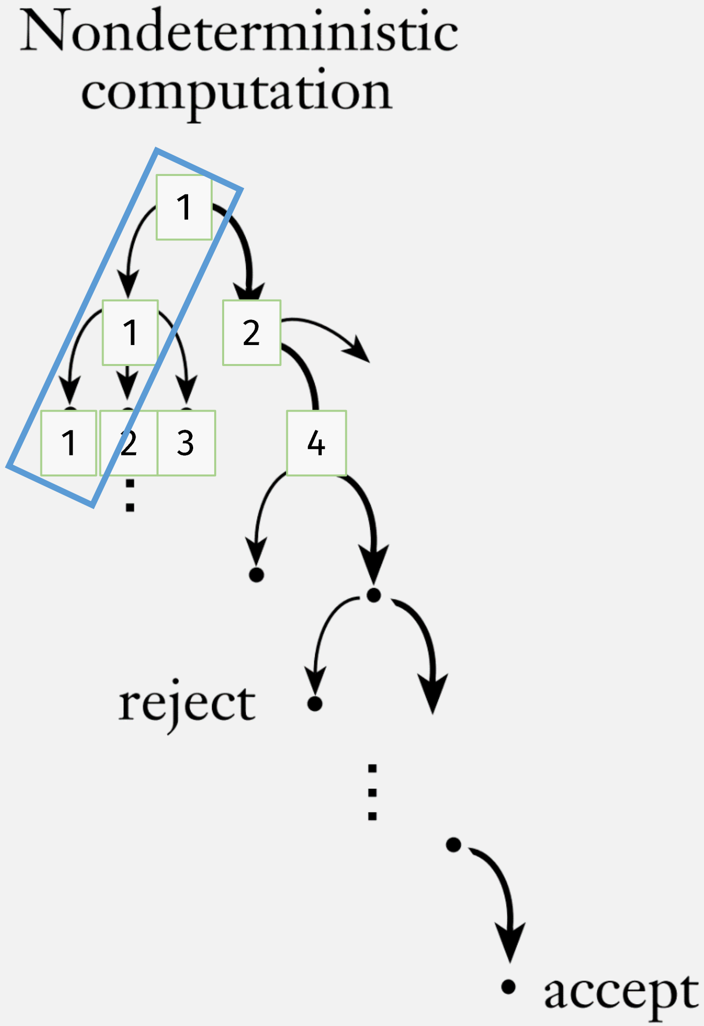
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - **1-2**



Nondeterministic TM → Deterministic

2^nd^ 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

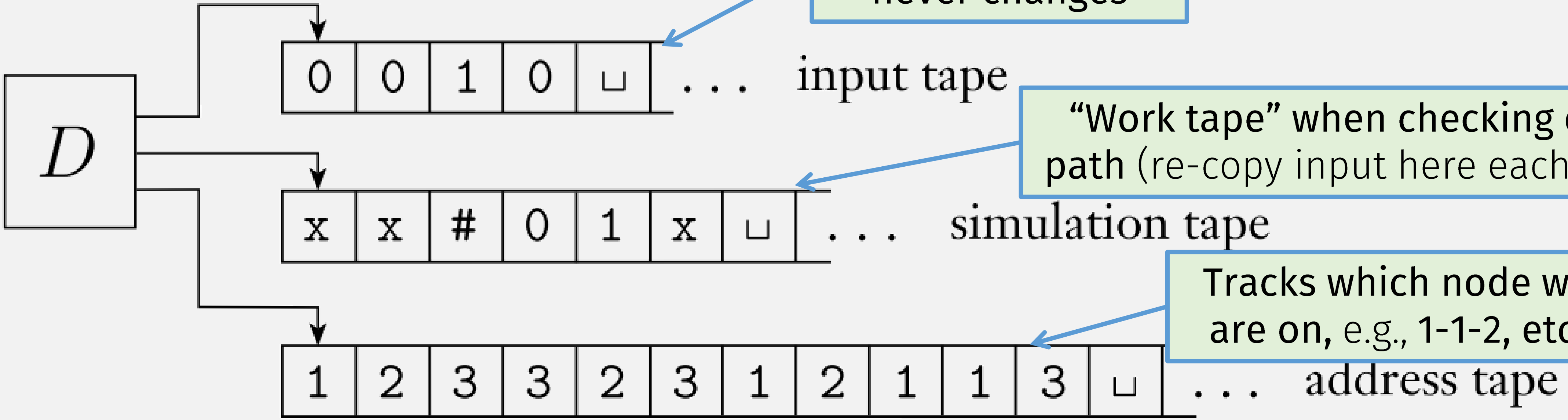
2nd way
(Sipser)

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

■

Conclusion: These are All Equivalent TMs!

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

Interlude: Running TMs inside other TMs

Just an analogy: “***calling***” TMs actually means “computing” its computation …

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

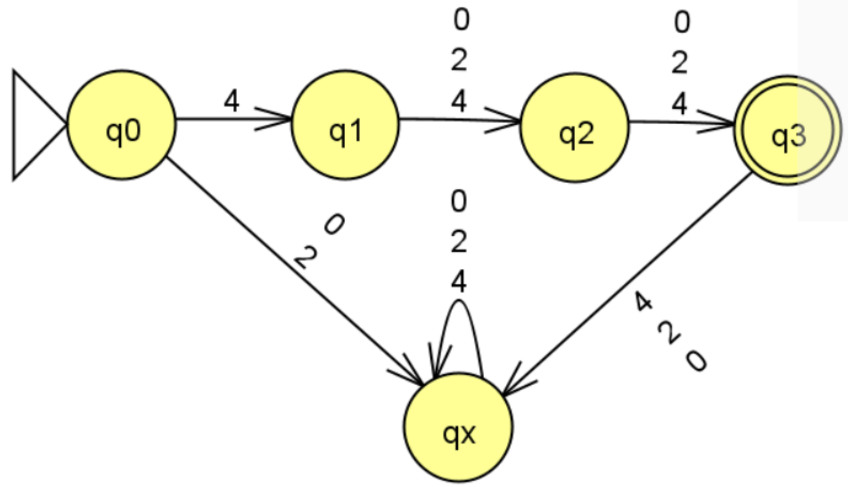
Possible solution #2:

$\boldsymbol{M}$ = 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
reject	accept	**accept**
accept	reject	**accept**
accept	loops	**accept**
loops	accept	**accept**

Flashback: HW 1, Problem 1



1. Come up with 2 strings that are accepted by the DFA. These strings are said to be in the **language** rec

2. Come up with not in the lan

3. Come up with a formal description for this DFA.

Recall that a DFA's formal description is a tuple of five components, e.g. $M = (Q, \Sigma, \delta, q_{start}, F)$.

You may assume that the alphabet contains only the symbols from the diagram.

4. Then for each of the following, say whether the computation represents an **accepting computation** or not (make sure to review the definition of an accepting computation). If the answer is no, explain why not:

a. $\hat{\delta}(q0, 420)$

Remember:
TMs = program (functions)

You had to "compute" how a DFA computes

Figuring out this HW problem about a DFA's computation … is itself (meta) computation!

What kind of ~~computation~~ **language** is it?

Could you write a program (function) to do it?

A function: **`DFAaccepts(B,w)`** returns TRUE if DFA **B** accepts string **w**

1) Define "current" state $q_{current}$ = start state q_0
2) For each input char a_i … in w
   a) Define $q_{next} = \delta_B(q_{current}, a_i)$
   b) Set $q_{current} = q_{next}$
3) Return TRUE if $q_{current}$ is an accept state (of **B**)

This is "computing" the **accepting computation** $\hat{\delta}(q_0, w) \in F$!!

The language of `DFAaccepts`

The set of strings that a **Turing Machine** accepts is a **language** ...

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle |\ B \text{ is a } \mathsf{DFA} \text{ that accepts input string } w\}$$

Is this language a set of strings???

A function: `DFAaccepts(B,w)` returns TRUE if DFA `B` accepts string `w`

Interlude: Encoding Things into Strings

Definition: A Turing machine's input is always a **string**

Problem: We sometimes want TM's (program's) input to be "something else" …

- **set, graph, DFA, …?**

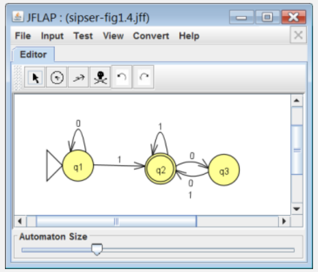
Solution: allow **encoding** other kinds of TM input as a string

Notation: <SOMETHING> = string **encoding** for SOMETHING

- A tuple combines multiple encodings, e.g., <*B*, *w*> (from prev slide)

Details don't matter! (In this class) **Just assume it's possible**

Example: Possible string encoding for a DFA?




```
<automaton>
  <!--The list of states.-->
  <state name="q1"><initial/></state>
  <state name="q2"><final/></state>
  <state name="q3"></state>
  <!--The list of transitions.-->
  <transition>
    <from>0</from>
    <to>0</to>
    <read>0</read>
  </transition>
  <transition>
    <from>1</from>
```

Or:

$$(Q, \Sigma, \delta, q_0, F)$$

(written as string)

Interlude: High-Level TMs and Encodings

A high-level TM description:

1. Needs to say the **type** of its input
   - E.g., graph, DFA, etc.

$M =$ "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

2. Doesn't need to say how input string is encoded

3. Assumes TM knows how to parse and extract parts of input

Description of M can refer to B's $(Q, \Sigma, \delta, q_0, F)$

4. Assumes input is a valid encoding
   - Invalid encodings implicitly rejected

`DFAaccepts` as a TM recognizing A_{DFA}

Remember:
TM ~ program (function)
Creating TM ~ programming

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

A function: `DFAaccepts(B,w)`
returns TRUE if DFA **B** accepts string **w**

1) Define "current" state q_{current} = start state q_0
2) For each input char a_i … in w
   a) Define $q_{\text{next}} = \delta(q_{\text{current}}, a_i)$
   b) Set $q_{\text{current}} = q_{\text{next}}$
3) Return TRUE if q_{current} is an accept state

TM M_{DFA} =

"On input $\langle B, w\rangle$, where B is a DFA and w is a string:

$B = (Q, \Sigma, \delta, q_0, F)$

1) Define "current" state q_{current} = start state q_0
2) For each input char a_i … in w
   a) Define $q_{\text{next}} = \delta(q_{\text{current}}, a_i)$
   b) Set $q_{\text{current}} = q_{\text{next}}$
3) **Accept** if q_{current} is an accept state

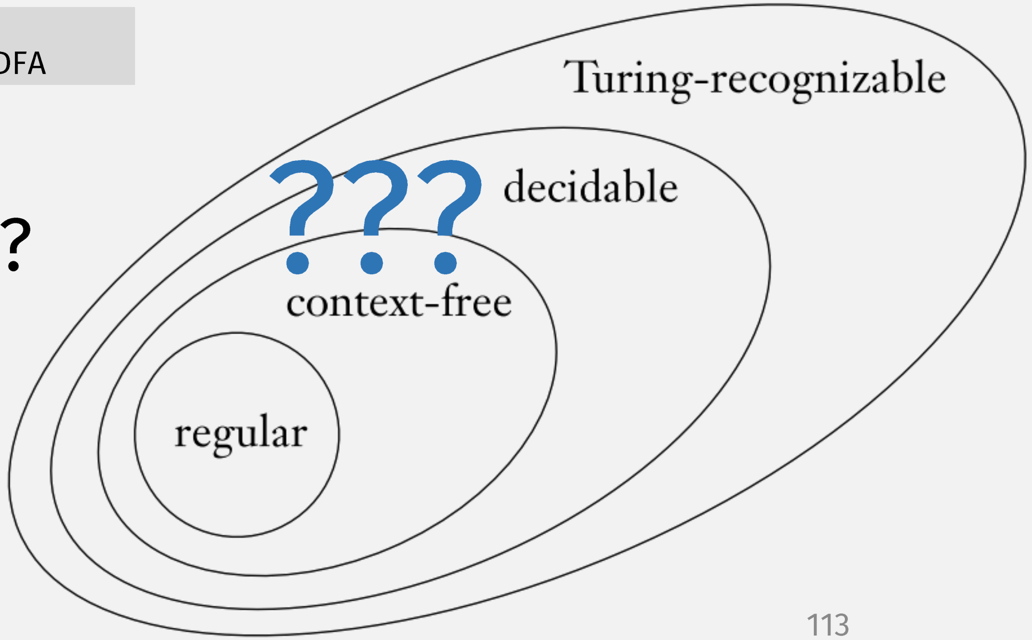
The language of **DFAaccepts**

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a } \mathsf{DFA} \text{ that accepts input string } w\}$$

TM M_{DFA}

- A_{DFA} has a Turing machine
- But **is that TM a decider** or **recognizer?**
  - I.e., **is it an algorithm**?
- **To show it's an algo, need to** <u>prove</u>:

$$A_{\mathsf{DFA}} \text{ is a decidable language}$$



How to prove that a language is decidable?

How to prove that a language is decidable?

Statements	**Justifications**
1. If a **decider** decides a lang L, then L is a **decidable** lang	1. Definition of **decidable** langs
2. Define **decider** M = On input w … , M **decides** L	2. See M def, and Examples Table
3. L is a **decidable** language	3. By statements #1 and #2

Key step

How to Design Deciders

- A **Decider** is a TM ...
  - See previous slides on how to:
    - write a **high-level TM description**
    - Express **encoded** input strings
  - E.g., M = On input <B, w>, where B is a DFA and w is a string: ...

- A **Decider** is a TM ... that must always **halt**
  - Can only **accept** or **reject**
  - Cannot go into an infinite loop

- So a **Decider** definition must include an extra **termination argument**:
  - Explains how every step in the TM halts
  - (Pay special attention to loops)

- Remember our analogy: TMs ~ Programs ... so *Creating* a TM ~ Programm*ing*
  - To design a TM, think of how to write a program (function) that does what you want

Next Time: A_{DFA} is a decidable language

$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B$ is a DFA that accepts input string $w\}$

Decider for A_{DFA} :