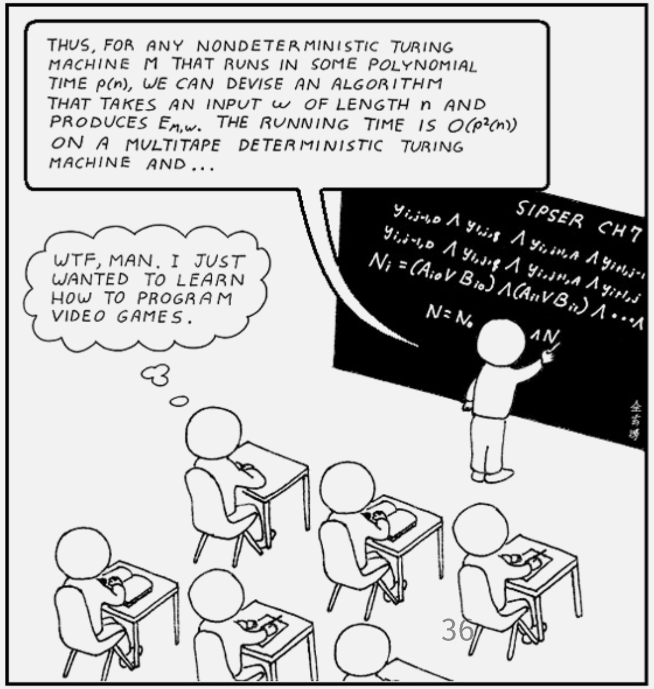
UMB CS420

Nondeterministic TMs

Wednesday, March 29, 2023



Announcements

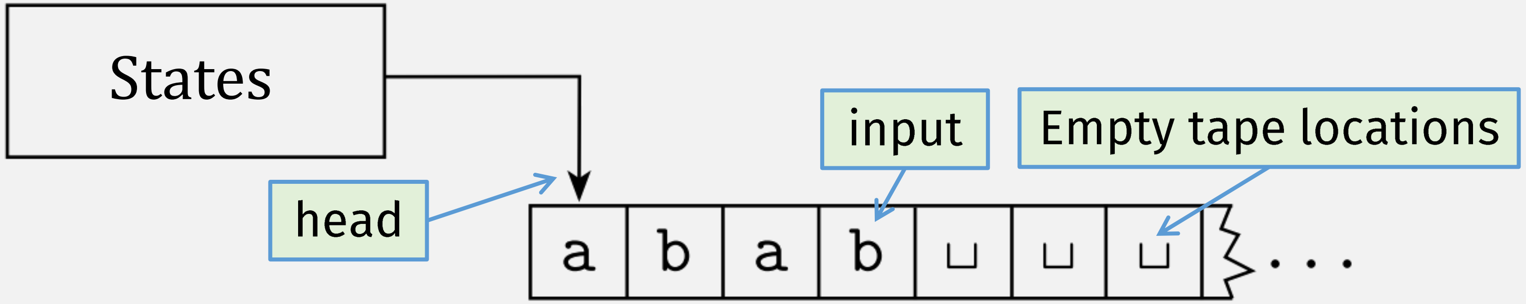
- HW 7 out
  - due Sun 4/2 11:59pm EST

Quiz Preview

- Which of the following are equivalent to a single-tape deterministic TM?

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

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.

Nondeterministic transition produces set of possible next 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. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ 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, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, 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. ~~$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$~~ → $\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$
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 $q_{reject} \neq q_{accept}$.

Thm: Deterministic TM ⇔ Non-det. TM

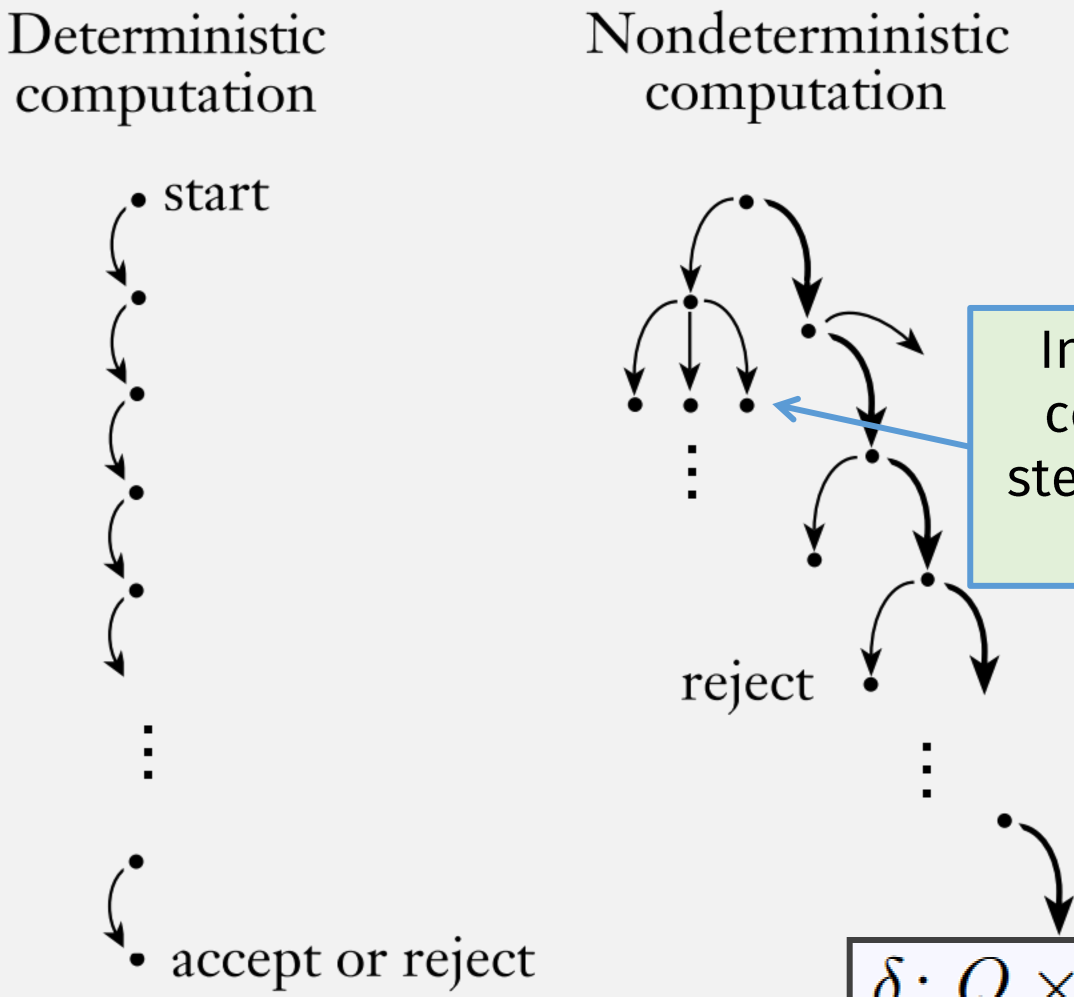
⇒ If a **deterministic TM** recognizes a language,
then a **non-deterministic TM** recognizes the language

- Convert: Deterministic TM → Non-deterministic TM …
- … change **Deterministic TM δ fn output to a one-element set**
  - $\delta_{\text{ntm}}(q, a) = \{\delta_{\text{dtm}}(q, a)\}$
  - (just like **d2n** conversion of DFA to NFA --- HW 2, Problem 2)
- **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?

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

Flashback: PDA Configurations (IDs)

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

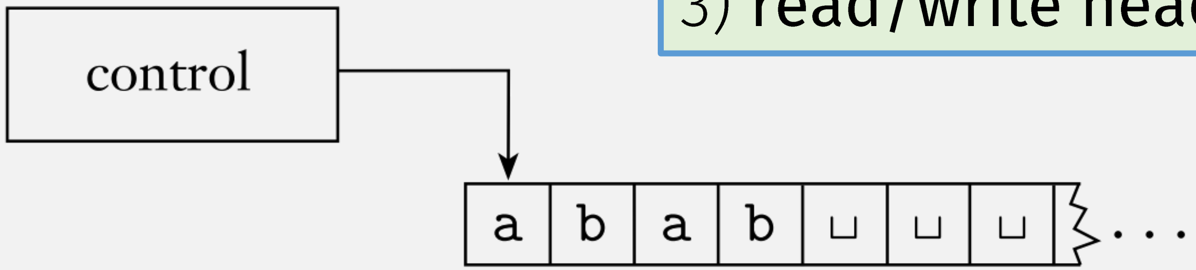
- 3 components (q, w, γ) :
  - q = the current state
  - w = the remaining input string
  - γ = the stack contents

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

TM Configuration (ID) = ???

1) states

3) read/write head

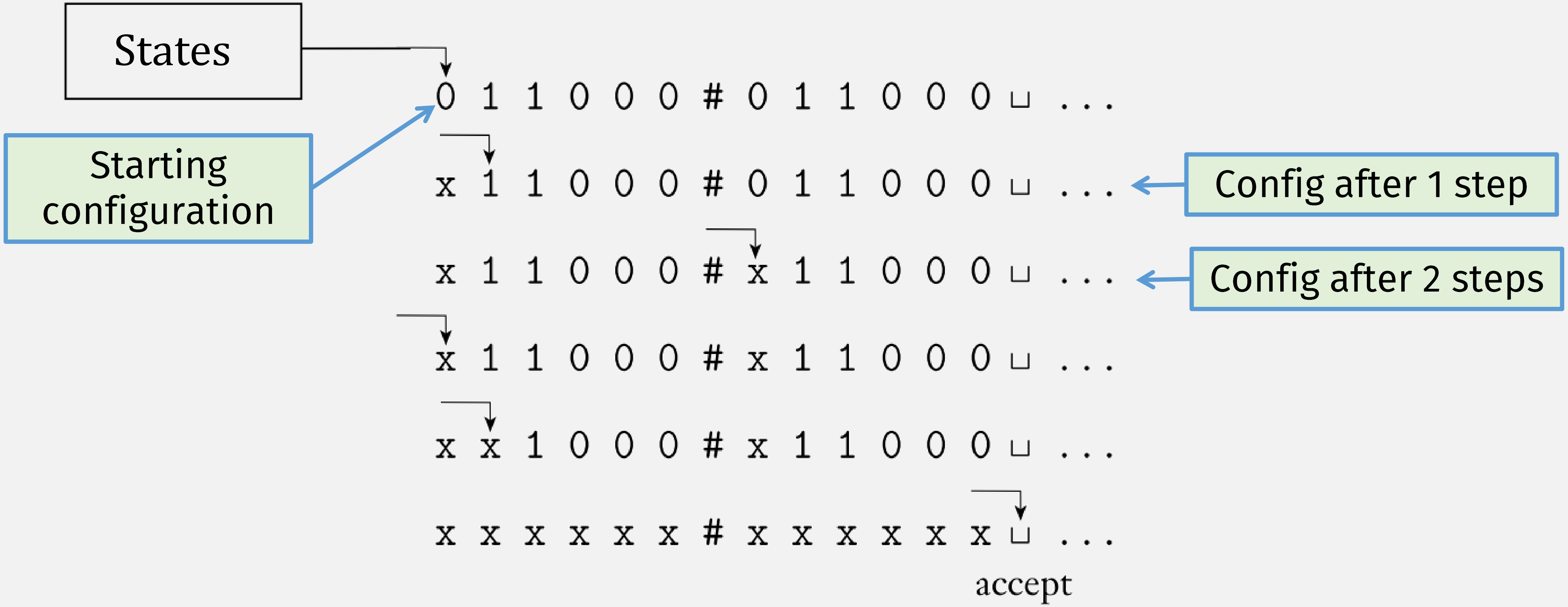


2) Tape contents

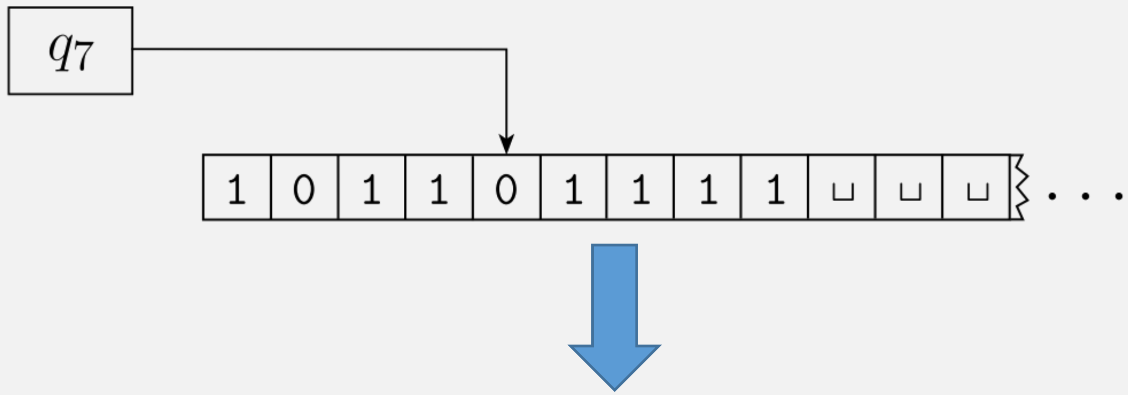
A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

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 $q_{reject} \neq q_{accept}$.

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

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

Single-step

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

head — Next config — write — read

if $q_1, q_2 \in Q$

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

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

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

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

<u>Edge cases</u>:

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

Extended

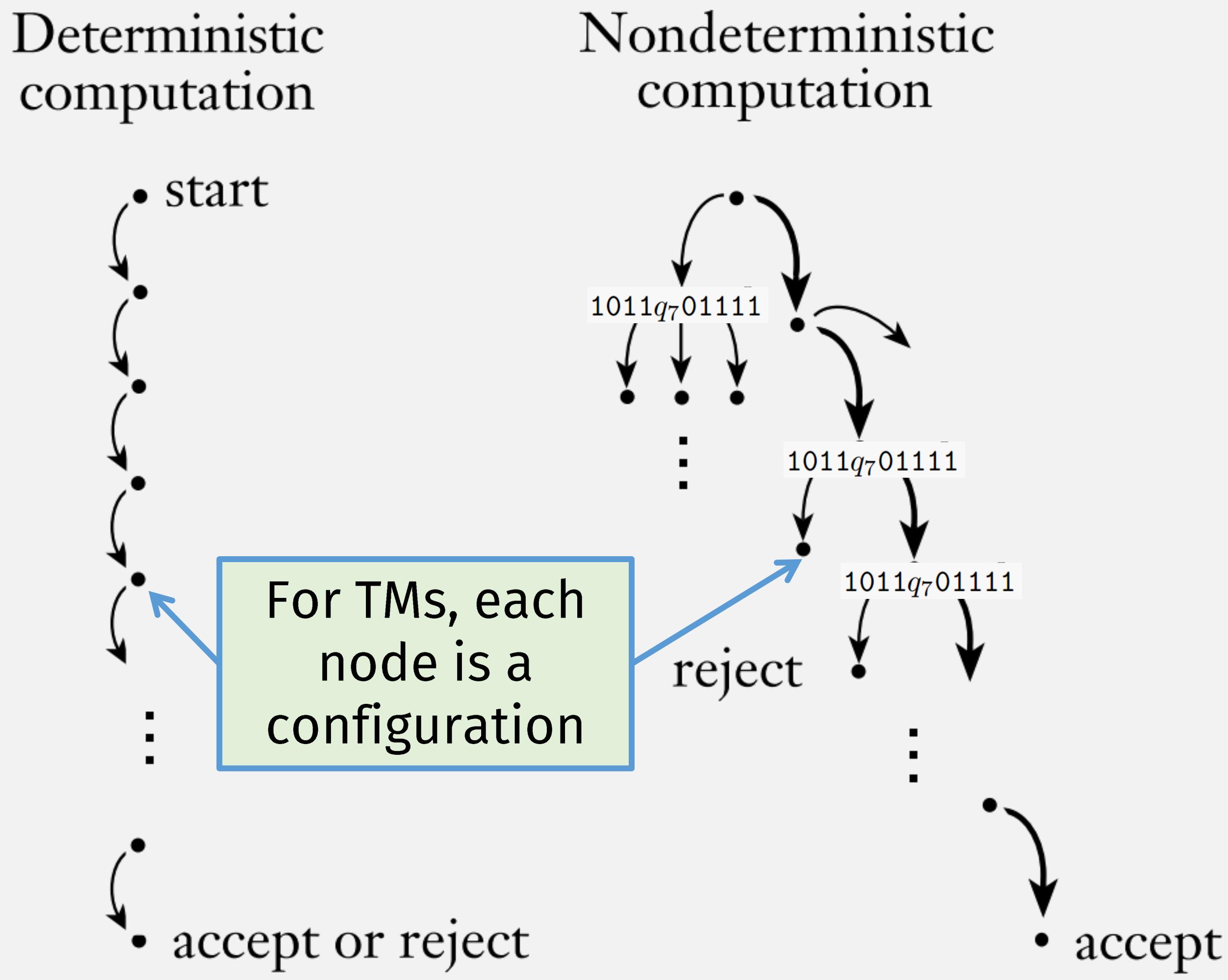
- Base Case

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

- Recursive Case

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

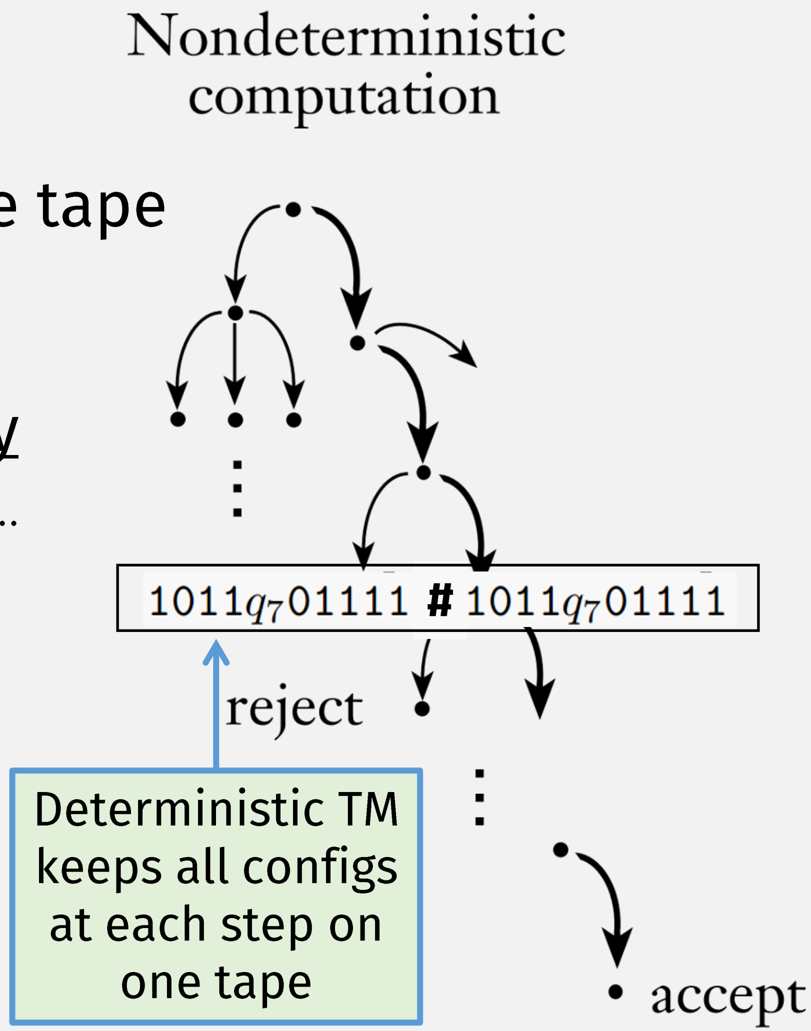
Nondeterminism in TMs



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: If TMs are like function definitions, then 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

"loop" means input string not accepted

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)

Interlude: Running TMs inside other TMs

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,

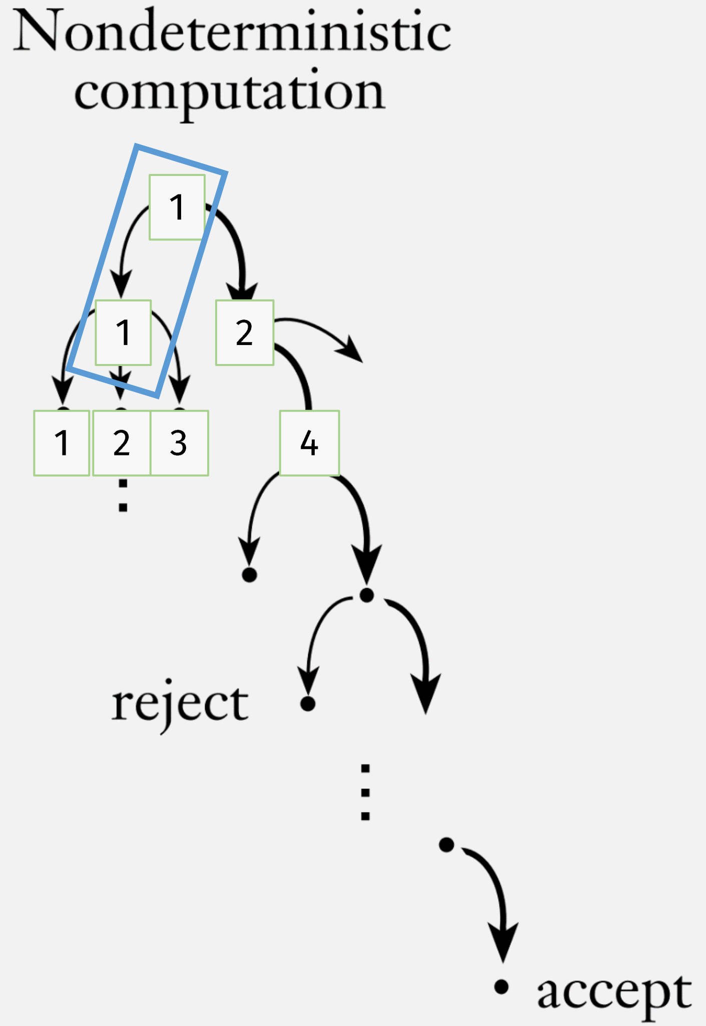
1. `Call` M_1 and M_2, each with x, concurrently, i.e.,
   a) Run M_1 with x for 1 step; **accept** if M_1 accepts
   b) Run M_2 with x for 1 step; **accept** if M_2 accepts
   c) Repeat

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

Nondeterministic TM → Deterministic

2nd way
(Sipser)

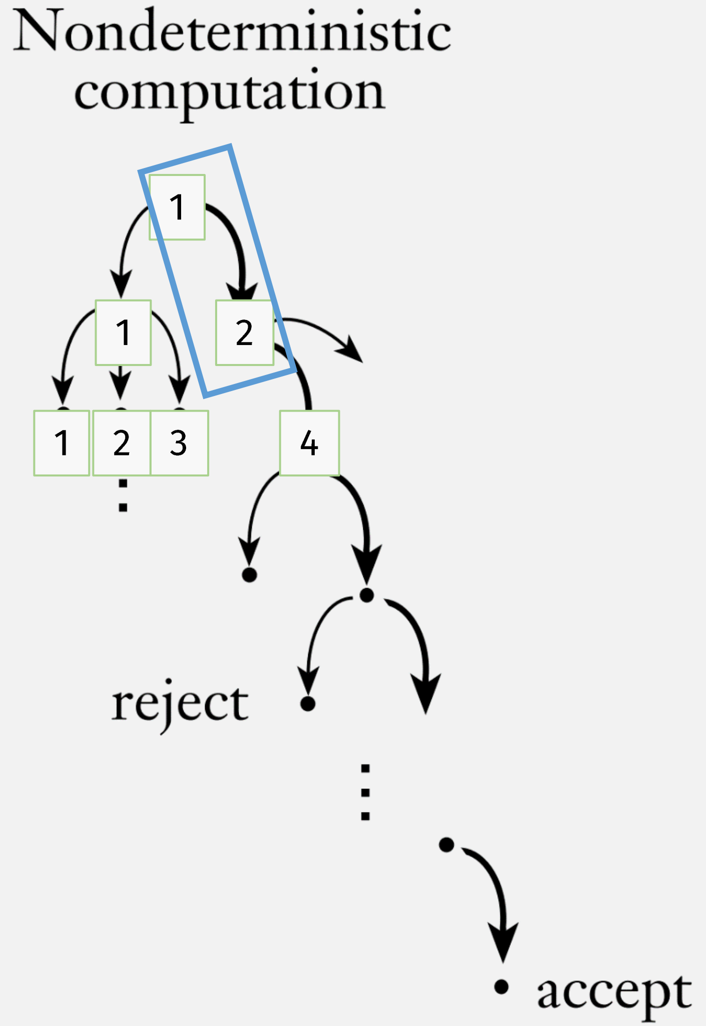
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - **1-1**



Nondeterministic TM → Deterministic

2nd way
(Sipser)

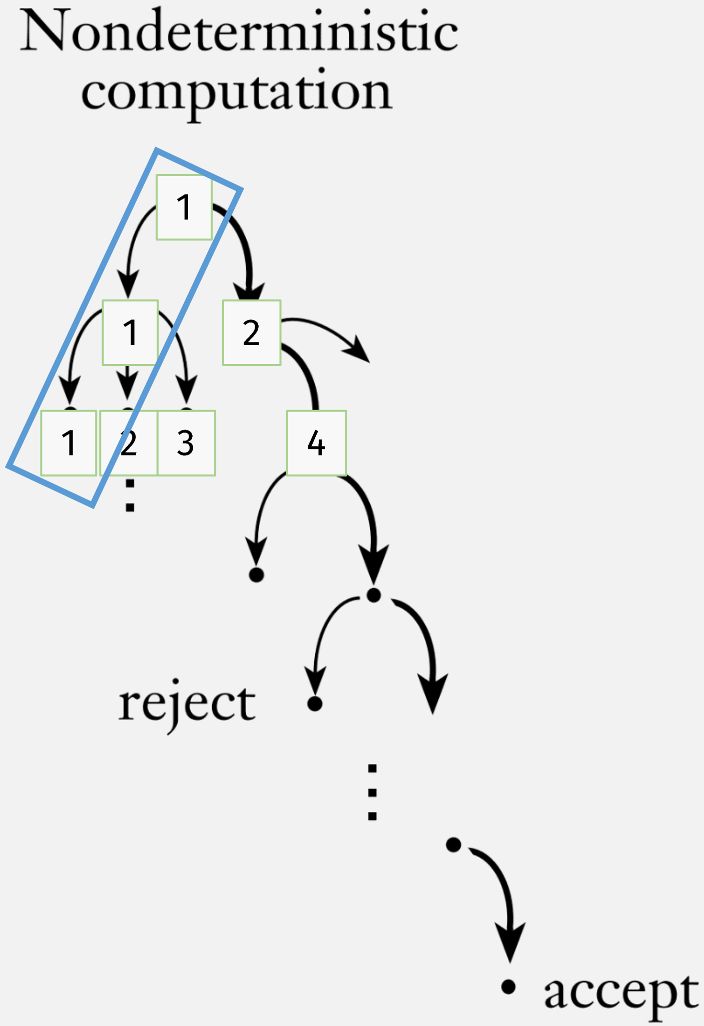
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - **1-2**



Nondeterministic TM → Deterministic

2nd way
(Sipser)

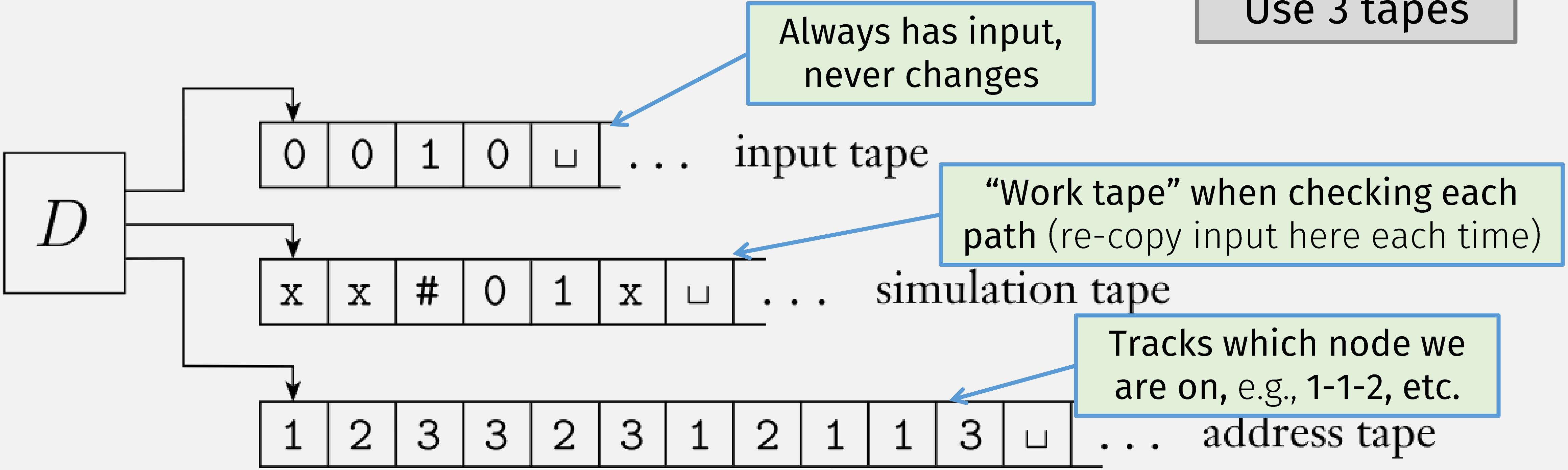
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - 1-2
    - **1-1-1**



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**

■

Conclusion: These are All Equivalent TMs!

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

Turing Machines as Algorithms

Turing Machines and Algorithms

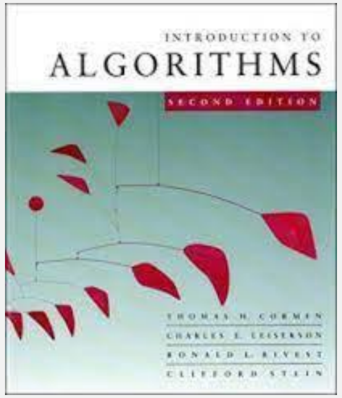
- **Turing Machines** can express any "computation"
  - I.e., a Turing Machine models (`Python`, `Java`) programs (functions)!

- 2 classes of Turing Machines
  - **Recognizers** may loop forever
  - **Deciders** always halt

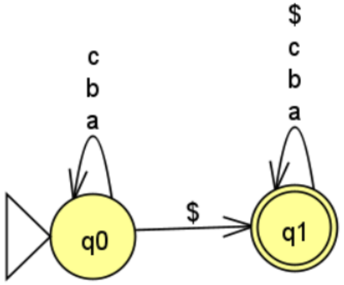
Remember:
TMs = program (functions)

Next

- **Deciders = Algorithms**
  - I.e., an algorithm is any program that always halts



Flashback: HW 1, Problem 1



1. Come up with 2 strings that are accepted by the DFA. These
   language recognized by the DFA.

2. Come up with 2 strings that are not accepted (rejected) by
   not in the language recognized by the DFA.

3. Is the empty string, ε, in the language of the DFA?

4. 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.

5. 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. $\hat{\delta}(q0, \texttt{a\$b})$

   b. $\hat{\delta}(q1, \texttt{a\$b})$

   c. $\hat{\delta}(q0, \texttt{abc})$

   d. $\hat{\delta}(q0, \texttt{cd\$})$

To "figure out" this computation … you had to "do" (meta) computations (e.g., in your head)

This represents computation by a DFA

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

Flashback: DFA Computations

Define the extended transition function: $\hat{\delta} : Q \times \Sigma^* \to Q$

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

Recursive case: $\hat{\delta}(q, a_1 w_{rest}) = \hat{\delta}(\delta(q, a_1), w_{rest})$

First char

Last chars

Single transition step

Remember:
TMs =
program (functions)

Calculating this computation requires (meta) computation!

Could you implement this (meta) computation as an **algorithm?**

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 ...
   a) Define $q_{next} = \delta(q_{current}, a_i)$
   b) Set $q_{current} = q_{next}$
3) Return TRUE if $q_{current}$ is an accept state

The language of `DFAaccepts`

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

But I thought a **language** is defined as 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**

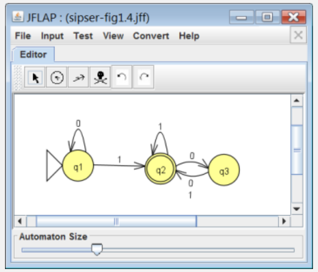
Problem: **A TM's** (program's) **input could also be: list, graph, DFA, …?**

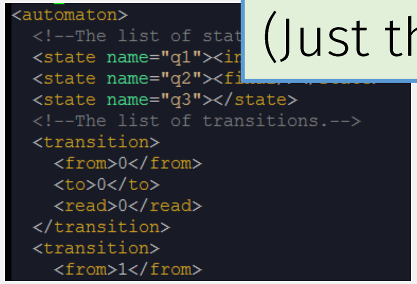
Solution: **encode other kinds of TM input as a string**

Notation: <SOMETHING> = string encoding for SOMETHING

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

Example: Possible string encoding for a DFA?





But in this class, we don't care about what the encoding is!
(Just that there is one)

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

(written as string)

Interlude: High-Level TMs and Encodings

A high-level TM description:

1. Doesn't need to describe exactly how input string is encoded
2. Assumes input is a "valid" encoding
   - Invalid encodings are implicitly rejected

The language of **`DFAaccepts`**

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

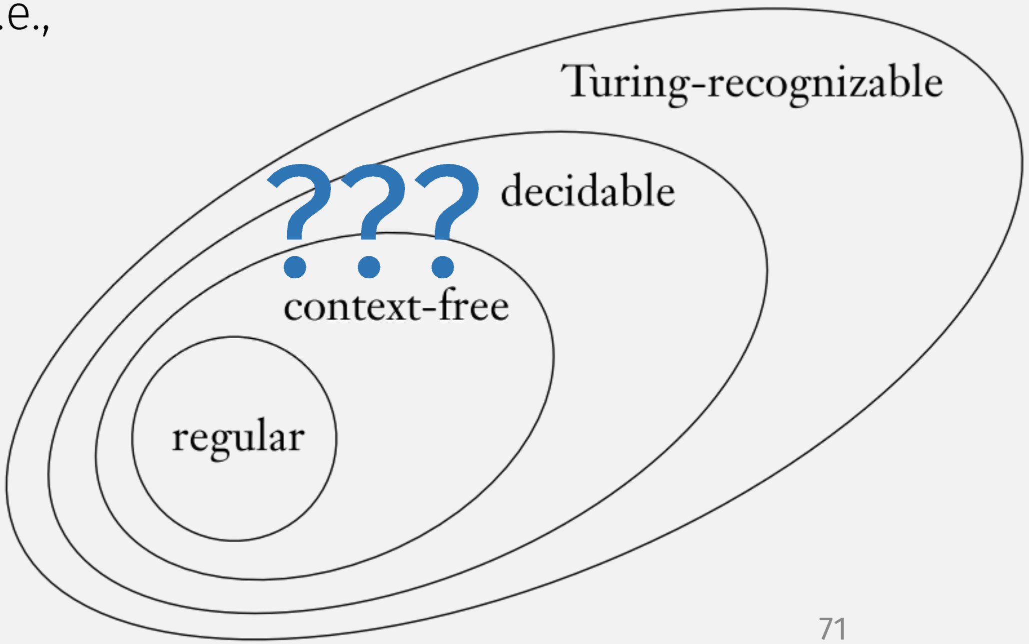
`DFAaccepts` is a Turing machine recognizing language A_{DFA} i.e.,

- its inputs strings look like <*B*, *w*> where
  - *B* is a DFA description
  - *w* is any string
- **`DFAaccepts`** accepts string <*B*, *w*> if
  - DFA *B* would end in accept state if run with input string *w*

But is **`DFAaccepts`** 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?

- Create a Turing machine that **decides** that language!

Remember:

- A **decider** is Turing Machine that always halts
  - I.e., for any input, it either accepts or rejects it.
  - It must never go into an infinite loop

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 examples
3. L is a **decidable** language	3. By statements #1 and #2

How to prove that a language is decidable?

- Create a Turing machine that **decides** that language!

Remember:

- A **decider** is Turing Machine that always halts
  - I.e., for any input, it either accepts or rejects it.
  - It must never go into an infinite loop

- Deciders must also include a **termination argument**:
  - Explains how every step in the TM halts
  - (Pay special attention to loops)

Next Time: A_{DFA} is a decidable language

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

Decider for A_{DFA} :

Check-in Quiz 3/29

On gradescope