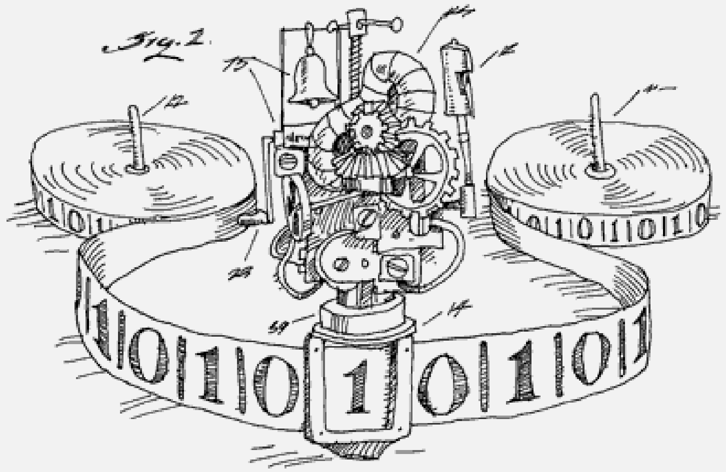
UMB CS420

Turing Machines (TMs)

Monday, March 27, 2023



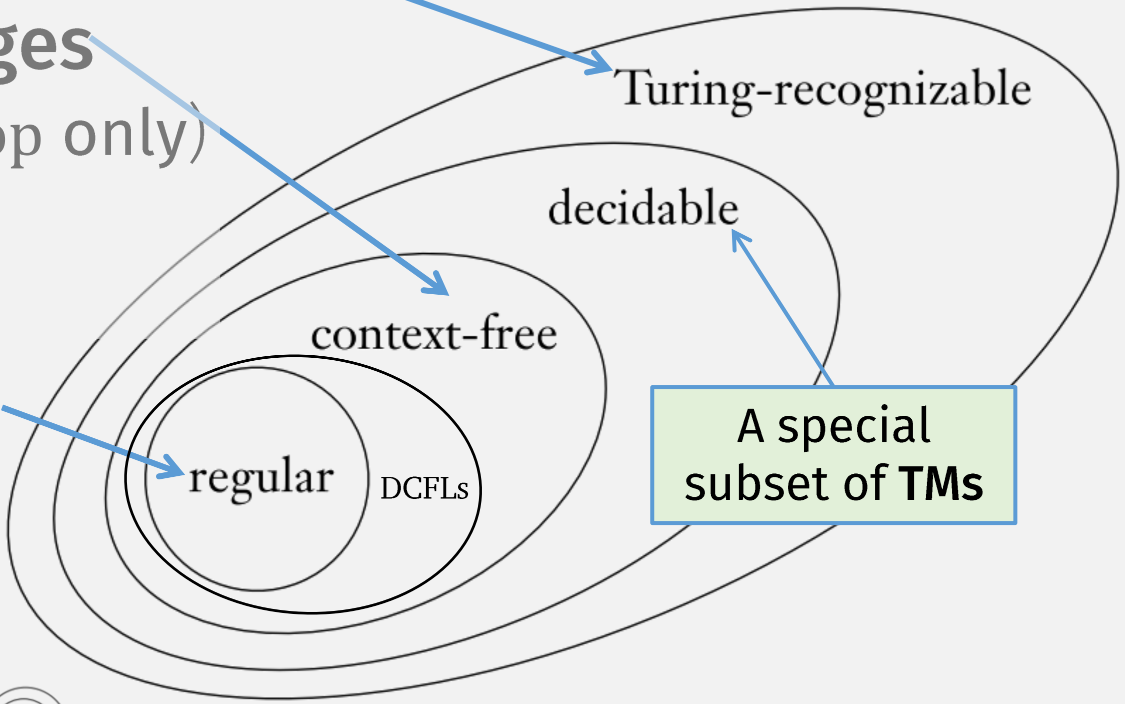
CS 420: Where We've Been, Where We're Going

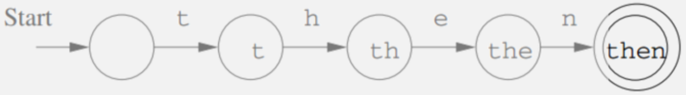
- **Turing Machines** (TMs)

  - Memory: states + infinite **tape**, (arbitrary read/write)
  - Expresses any "computation"
- **PDAs**: recognize **context-free languages**
  - Memory: states + infinite **stack** (push/pop only)
  - Can't express: arbitrary dependency,
    - e.g., $\{ww \mid w \in \{0,1\}^*\}$

$A \rightarrow 0A1$
$A \rightarrow B$
$B \rightarrow \#$

- **DFAs / NFAs**: recognize **regular langs**
  - Memory: finite states
  - Can't express: **dependency**
    e.g., $\{0^n1^n \mid n \geq 0\}$

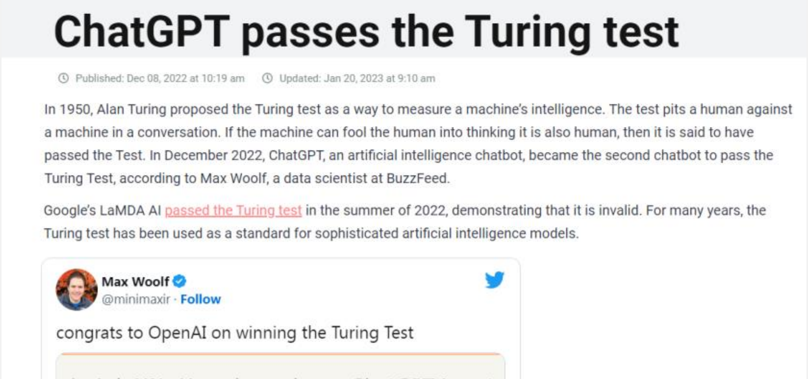




Alan Turing

- First to formalize a model of computation
  - I.e., he invented many of the ideas in this course
- And worked as a codebreaker during WW2

- Also studied Artificial Intelligence
  - The Turing Test



ChatGPT passes the Turing test

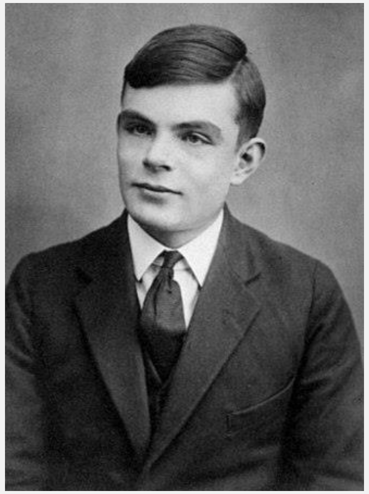
Published: Dec 08, 2022 at 10:19 am Updated: Jan 20, 2023 at 9:10 am

In 1950, Alan Turing proposed the Turing test as a way to measure a machine's intelligence. The test pits a human against a machine in a conversation. If the machine can fool the human into thinking it is also human, then it is said to have passed the Test. In December 2022, ChatGPT, an artificial intelligence chatbot, became the second chatbot to pass the Turing Test, according to Max Woolf, a data scientist at BuzzFeed.

Google's LaMDA AI passed the Turing test in the summer of 2022, demonstrating that it is invalid. For many years, the Turing test has been used as a standard for sophisticated artificial intelligence models.

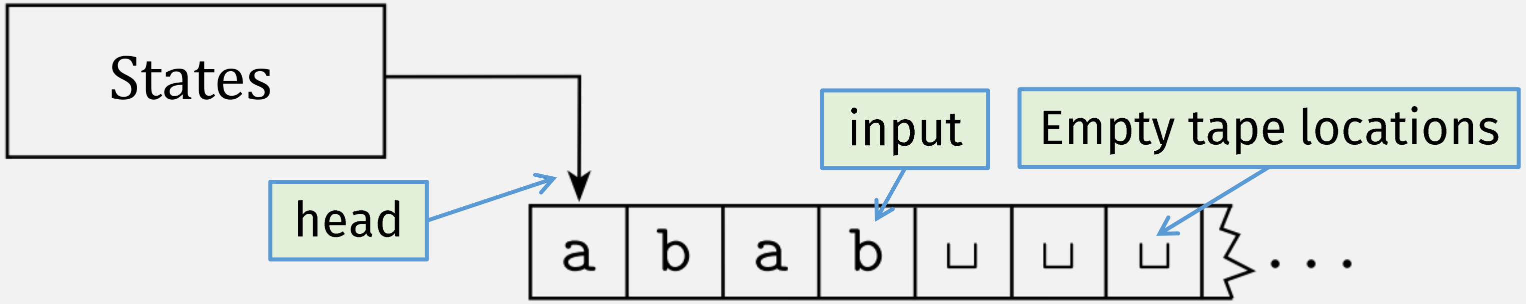
Max Woolf
@minimaxir · Follow

congrats to OpenAI on winning the Turing Test



Finite Automata vs Turing Machines

- **Turing Machines** can <u>read and write</u> to <u>arbitrary</u> "tape" cells
  - Tape initially contains input string



- **Tape is infinite**
  - To the right
- <u>Each step</u>: **"head" can move left or right**
- Turing Machine can **accept** / **reject** at any time

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

Turing Machine Example

Define: TM M_1 accepts inputs in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

Example input

tape

head

0 1 1 0 0 0 # 0 1 1 0 0 0 ␣ . . .

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.

High-level: "Cross off"
Low-level δ: write "x" char

This is a **high-level TM description**

It is **equivalent to** (but more concise than) our typical (low-level) tuple descriptions, i.e., one step = maybe multiple δ transitions

Analogy
"High-level": `Python`
"Low-level": `assembly language`

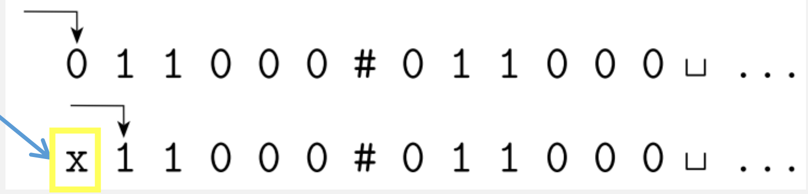
Turing Machine Example

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

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.

"Cross off" = write "x" char



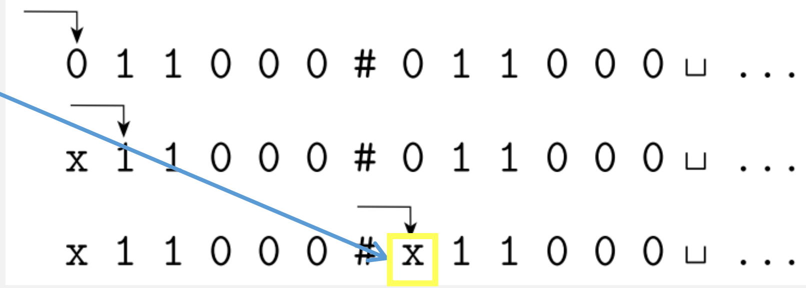
Turing Machine Example

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

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.

"Cross off" = write "x" char



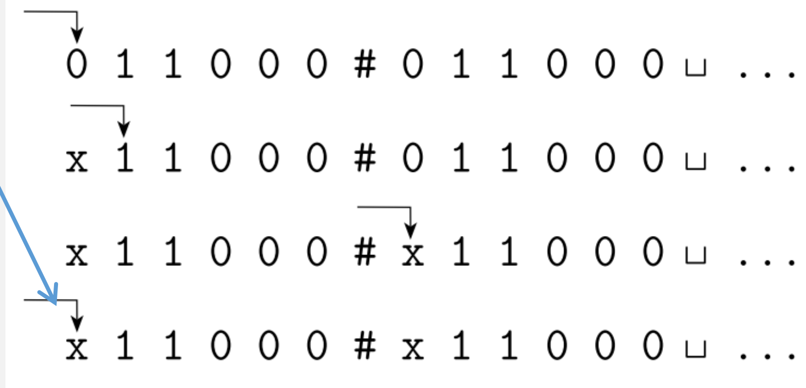
Turing Machine Example

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

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.

Head "zags" back to start



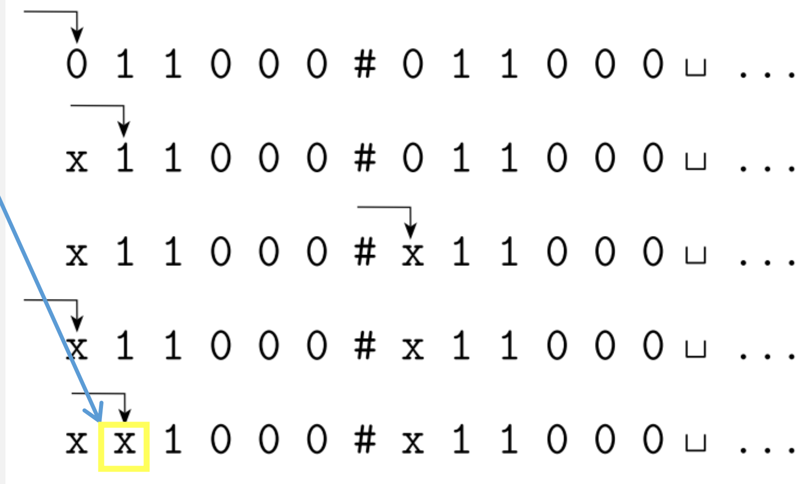
Turing Machine Example

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

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.

Continue crossing off



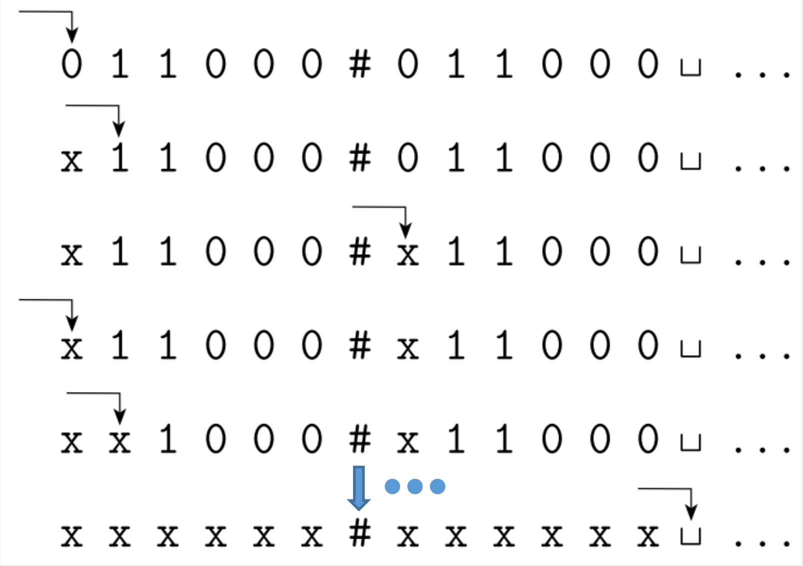
Turing Machine Example

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

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 Example

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

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 Machines: Formal Definition

This is a "**low-level**" TM description

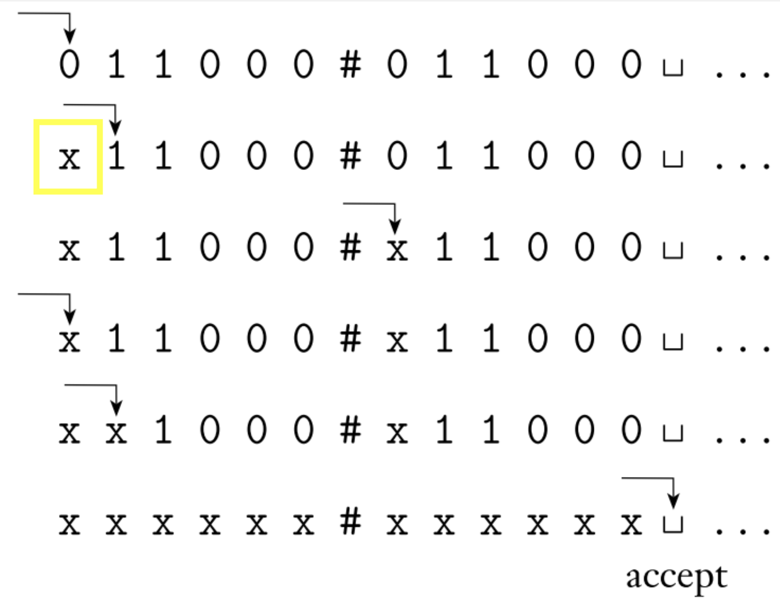
A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

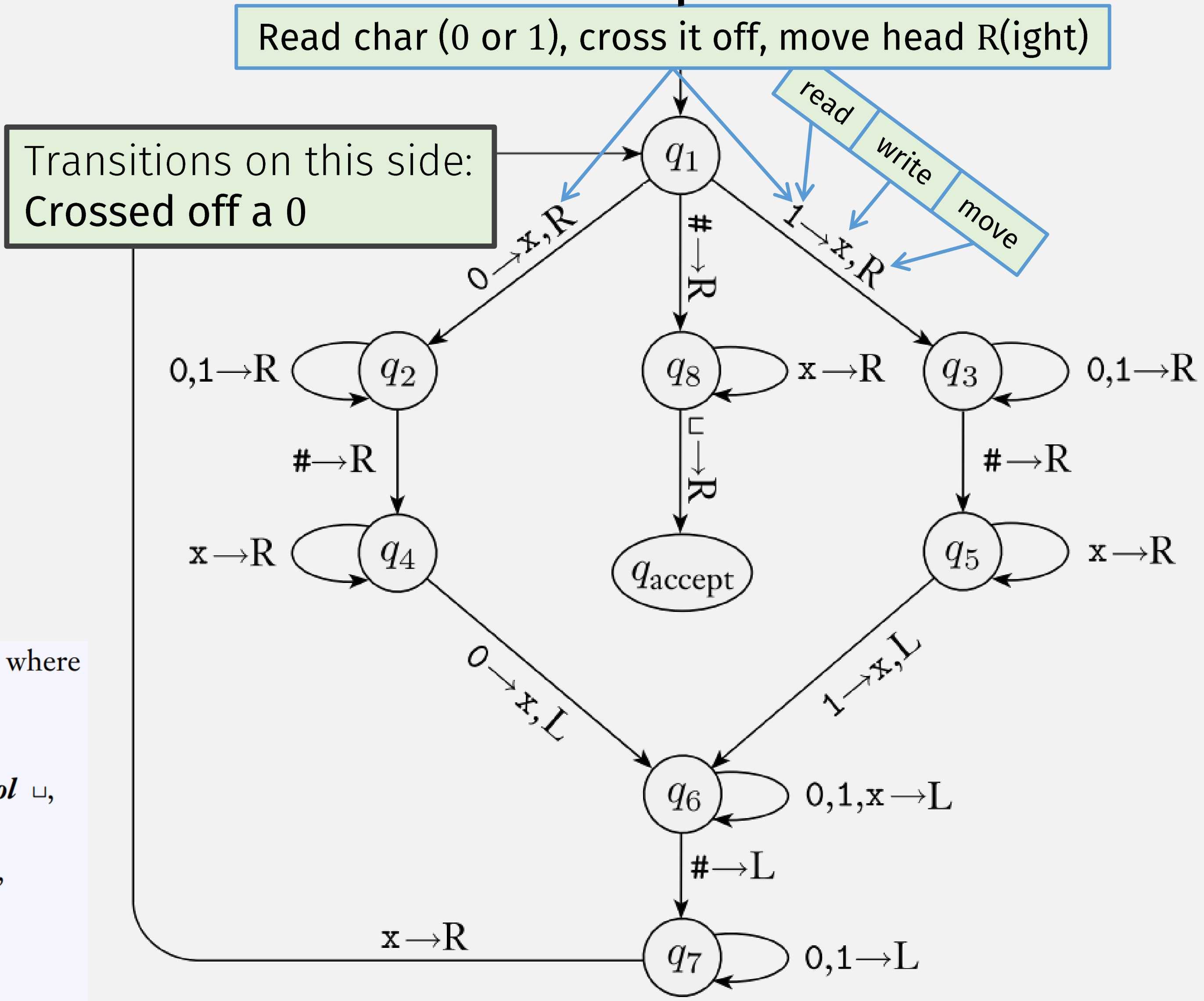
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
   read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

Is this machine **deterministic**?
Or **non-deterministic**?

$$B = \{w\#w \mid w \in \{0,1\}^*\}$$

Formal Turing Machine Example



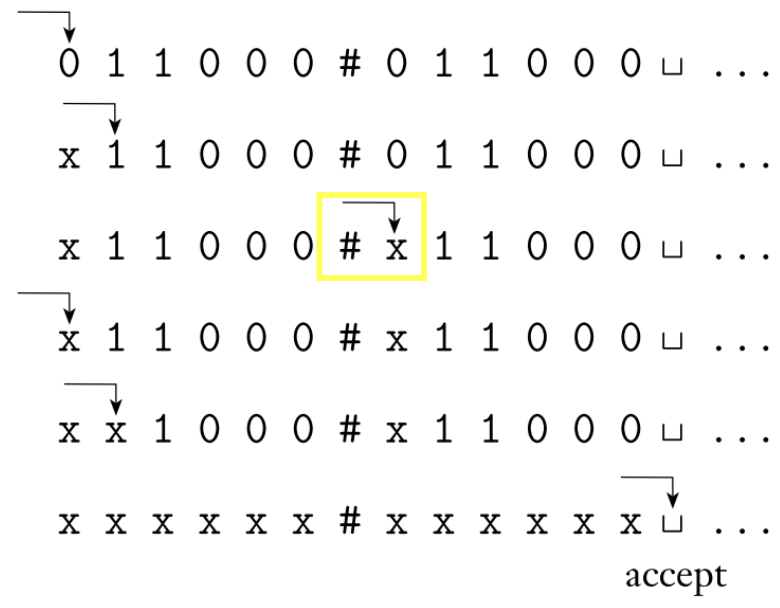


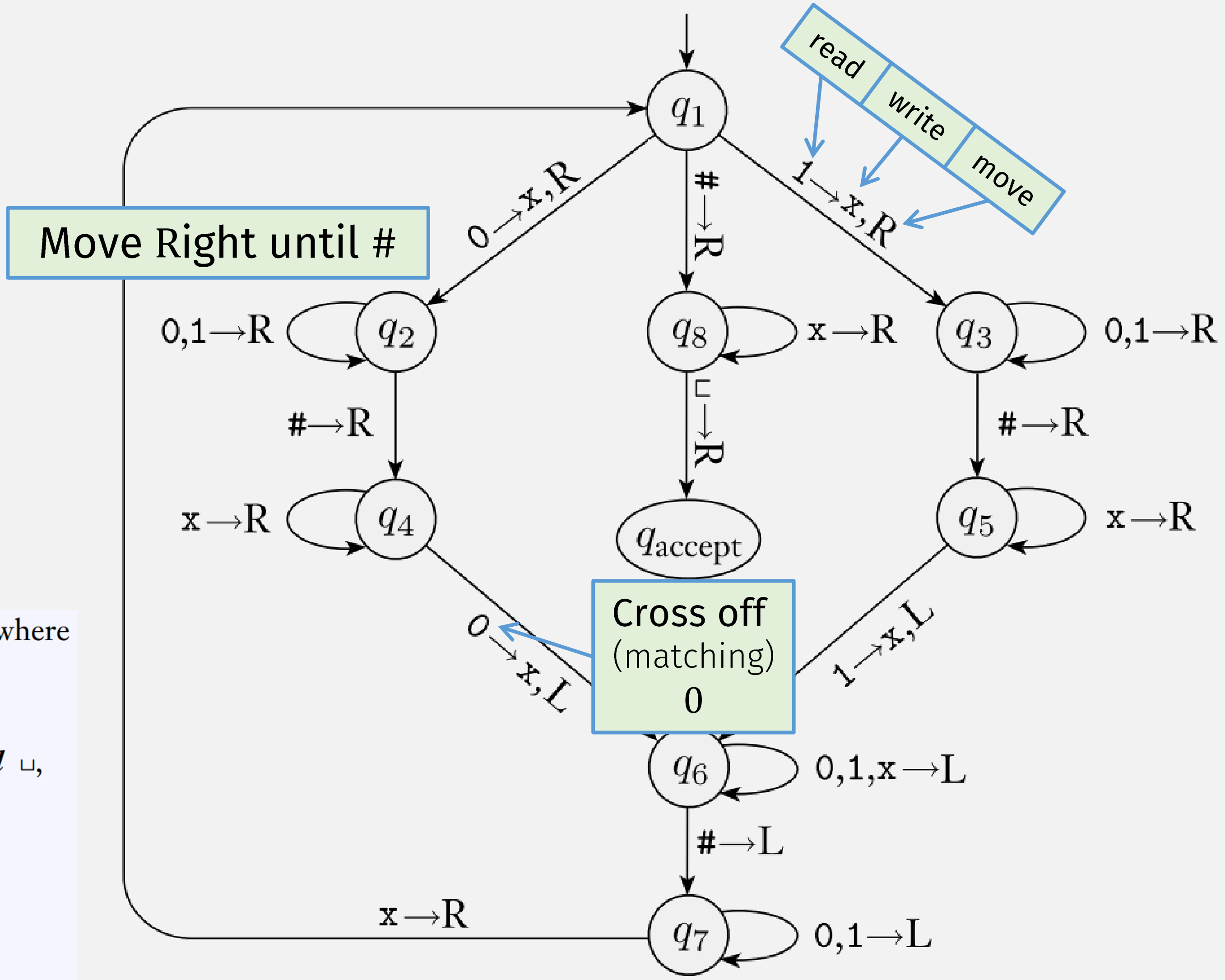
A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

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

$B = \{w\#w \mid w \in \{0,1\}^*\}$

Formal Turing Machine Example



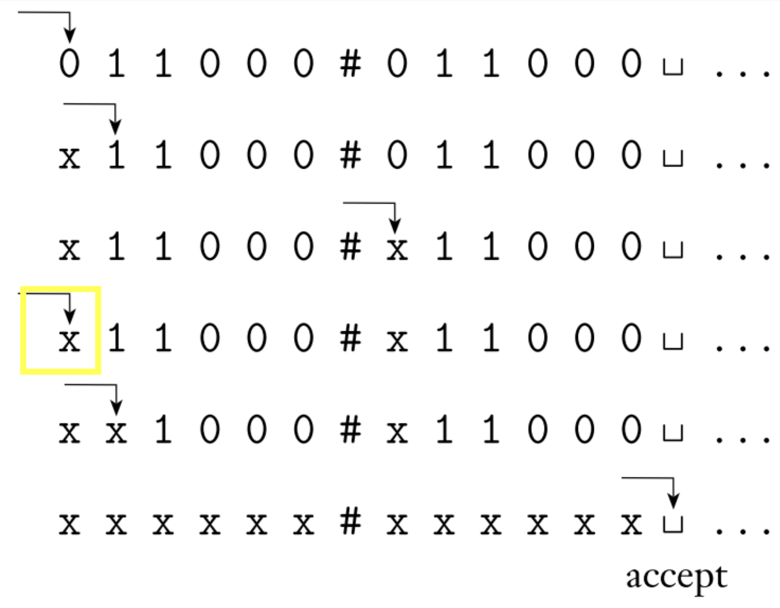


A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

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

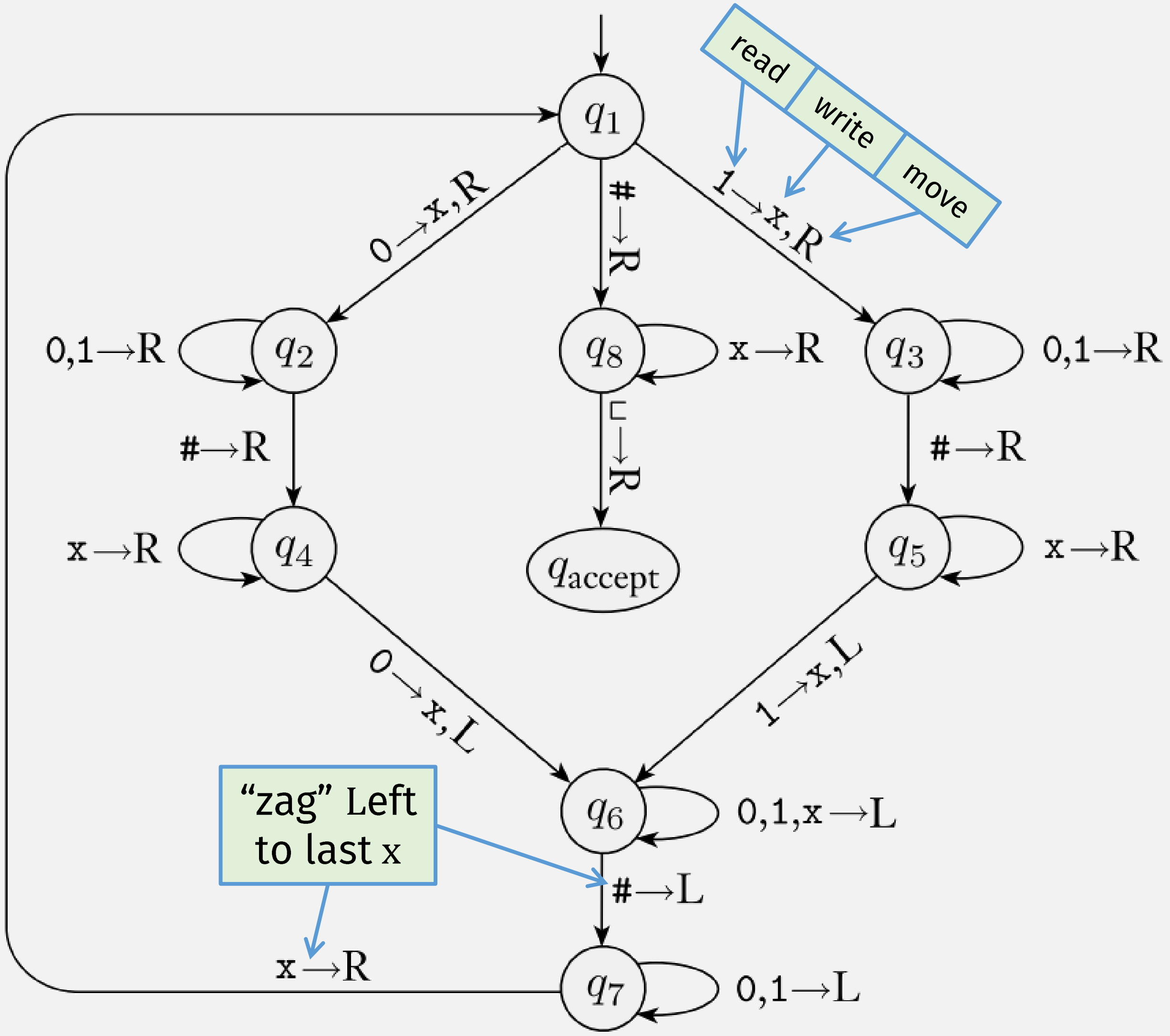
$B = \{w\#w \mid w \in \{0,1\}^*\}$

Formal Turing Machine Example



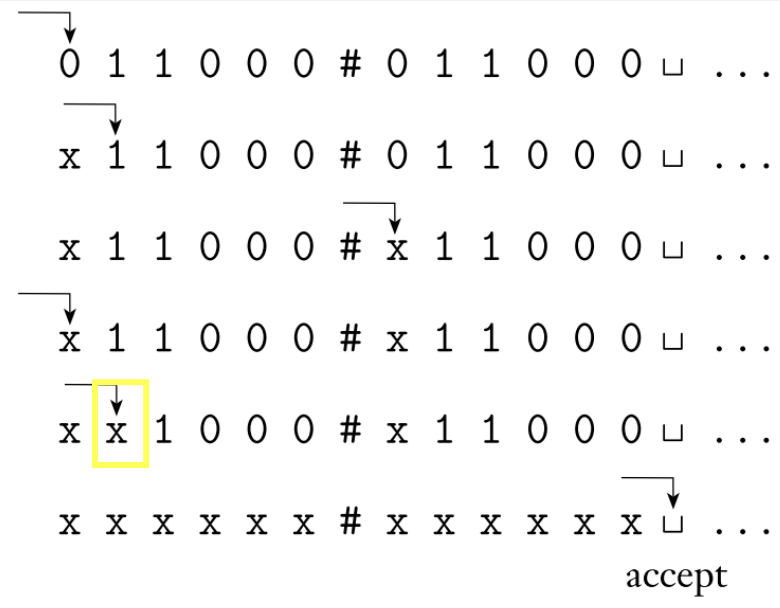
A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

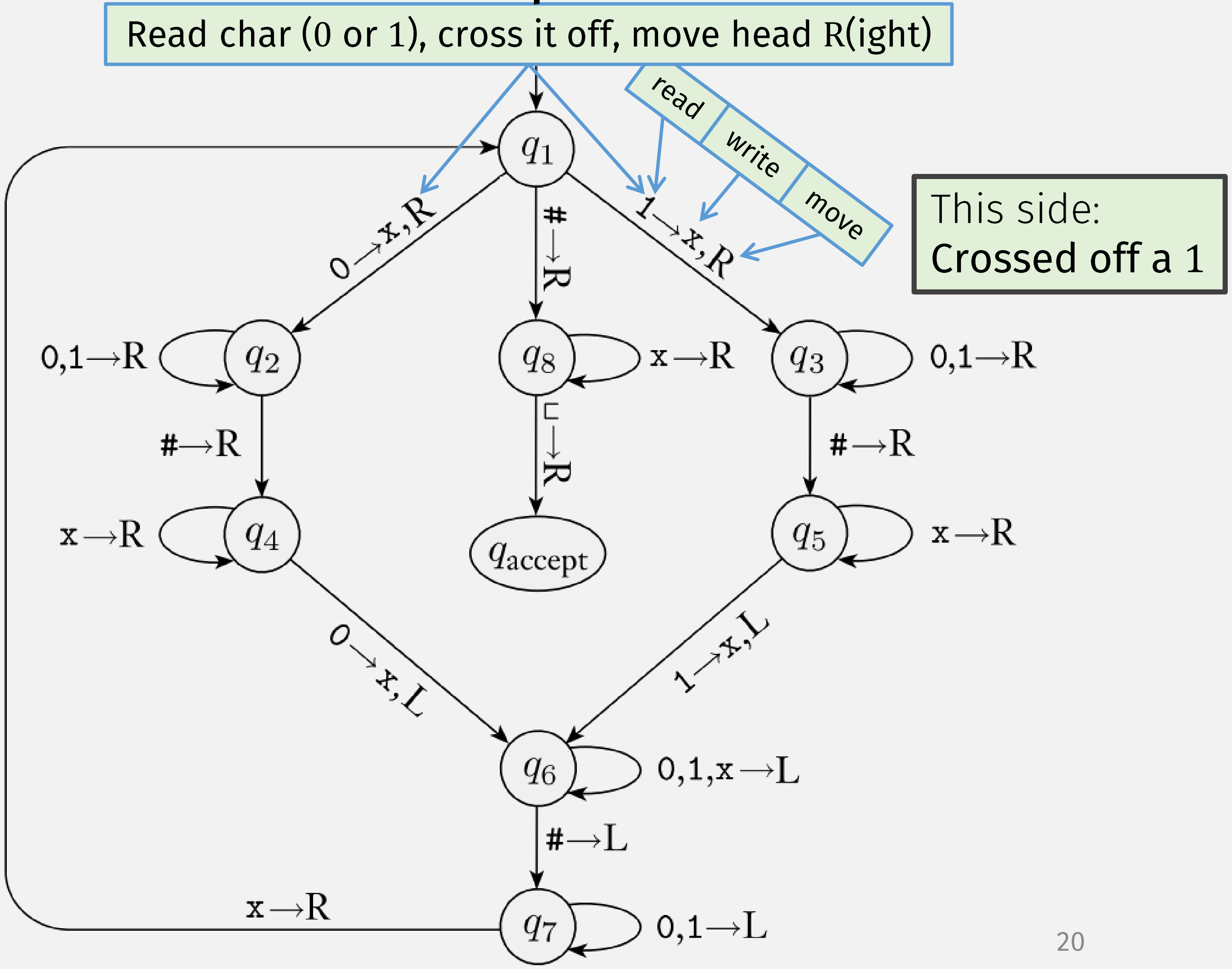
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** ⊔,
3. Γ is the tape alphabet, where $⊔ \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in$ read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



$B = \{w\#w \mid w \in \{0,1\}^*\}$

Formal Turing Machine Example



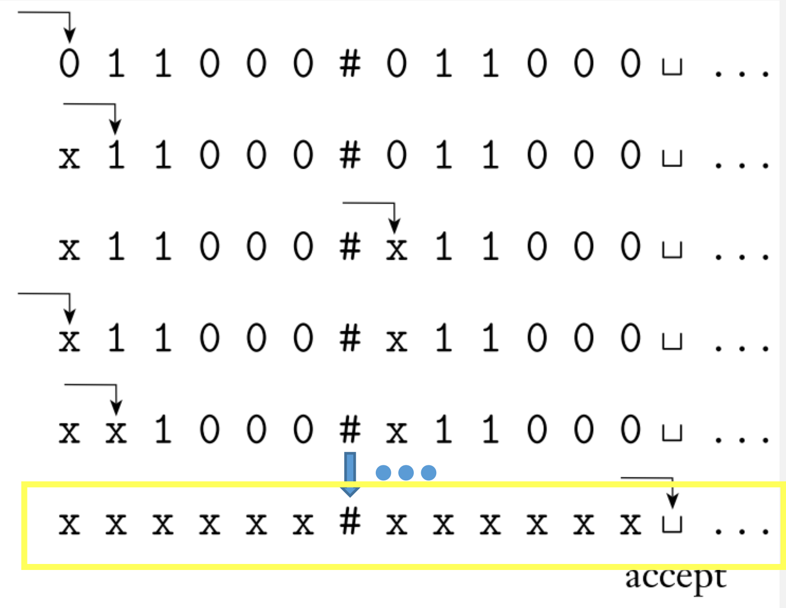


A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

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. $q_0 \in$ read e s write move
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

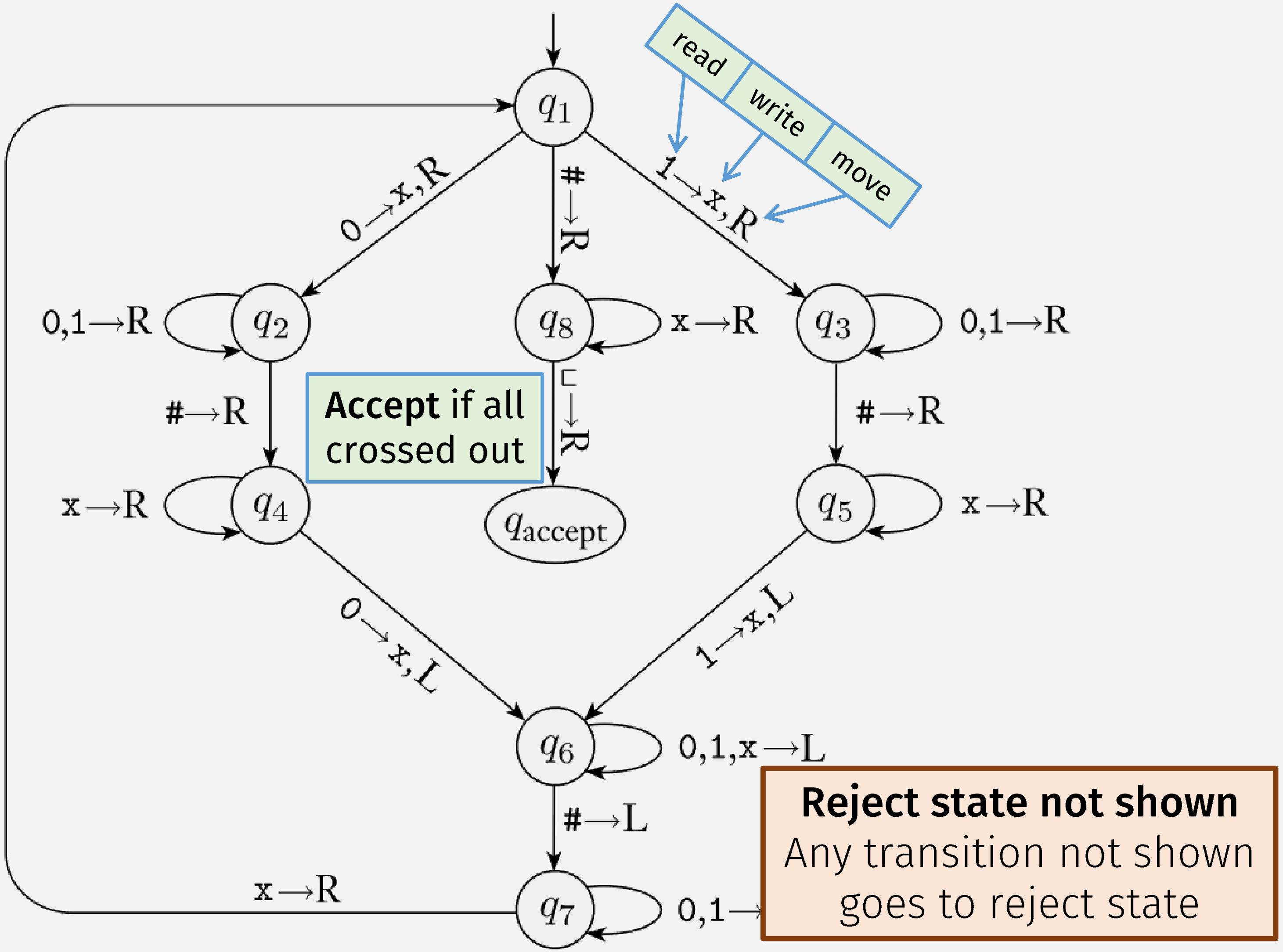
$B = \{w\#w \mid w \in \{0,1\}^*\}$

Formal Turing Machine Example



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

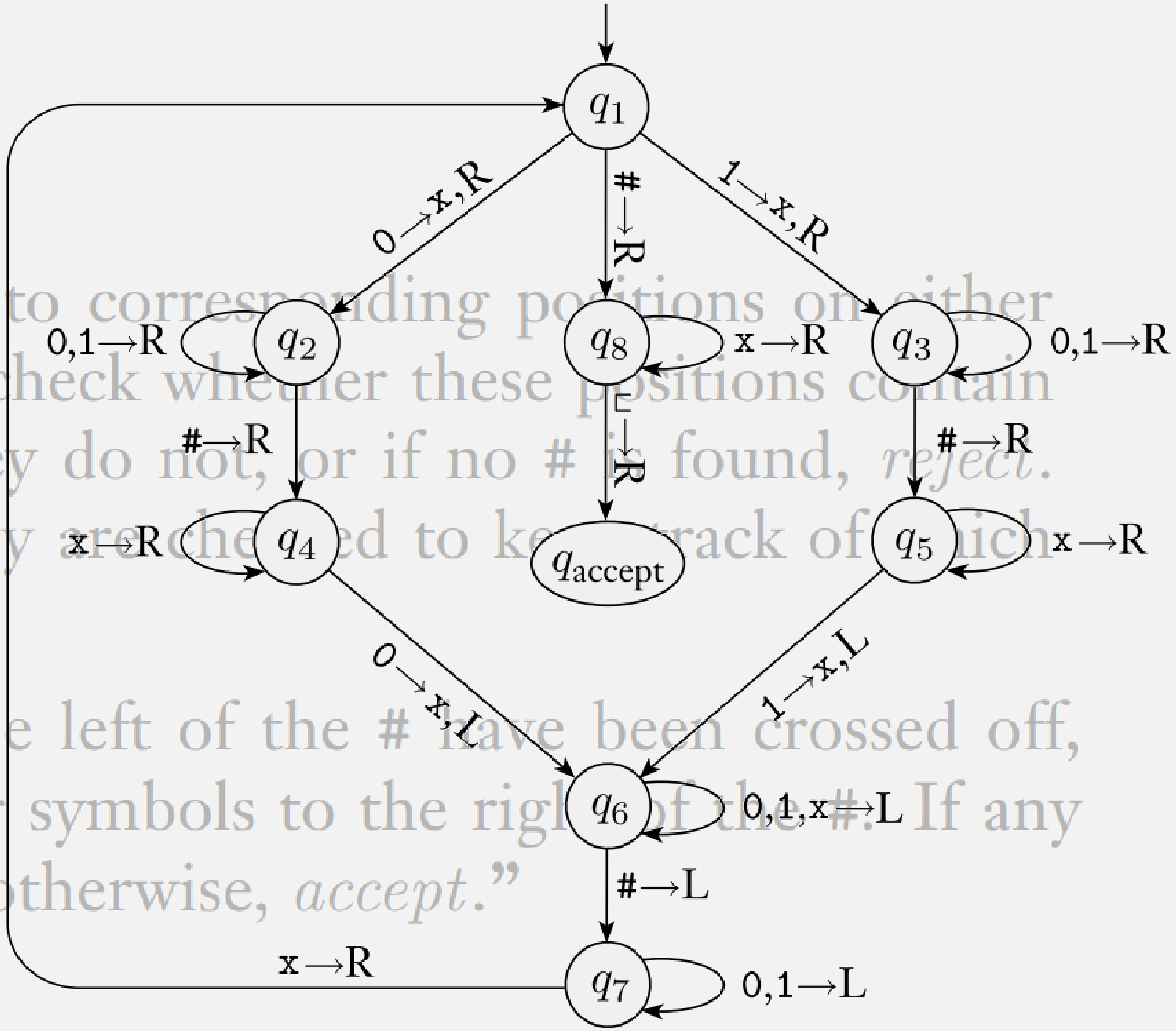
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0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



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 the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do ... if no # is found, *reject*. Cross off symbols as they ... track of which symbols correspond.

2. When all symbols to ... crossed off, check for any remaining ... of the #. If any symbols remain, *reject*; otherwise, ...*ccept*."

We will (mostly) stick to **high-level** descriptions of Turing machines, like this one

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. All TMs must have a <u>name</u>, e.g., M_1
2. <u>Input strings</u> must also be <u>named</u> (like a function parameter), e.g., w
3. TMs can "call" or "simulate" other TMs (if they pass appropriate arguments)
   - e.g., a step for a TM M can say: "call TM M_2 with argument string w, if M_2 accepts w then ..., else ..."
4. Follow typical programming "scoping" rules
   - can assume functions we've already defined are in "global" scope, RE2NFA …
5. Other variables must also be defined (named) before they are used
   - 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 a TM computation has <u>3 possible results</u>:
- Accept
- Reject
- Loop forever

- A Turing Machine can <u>run forever</u>
  - E.g., the 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.

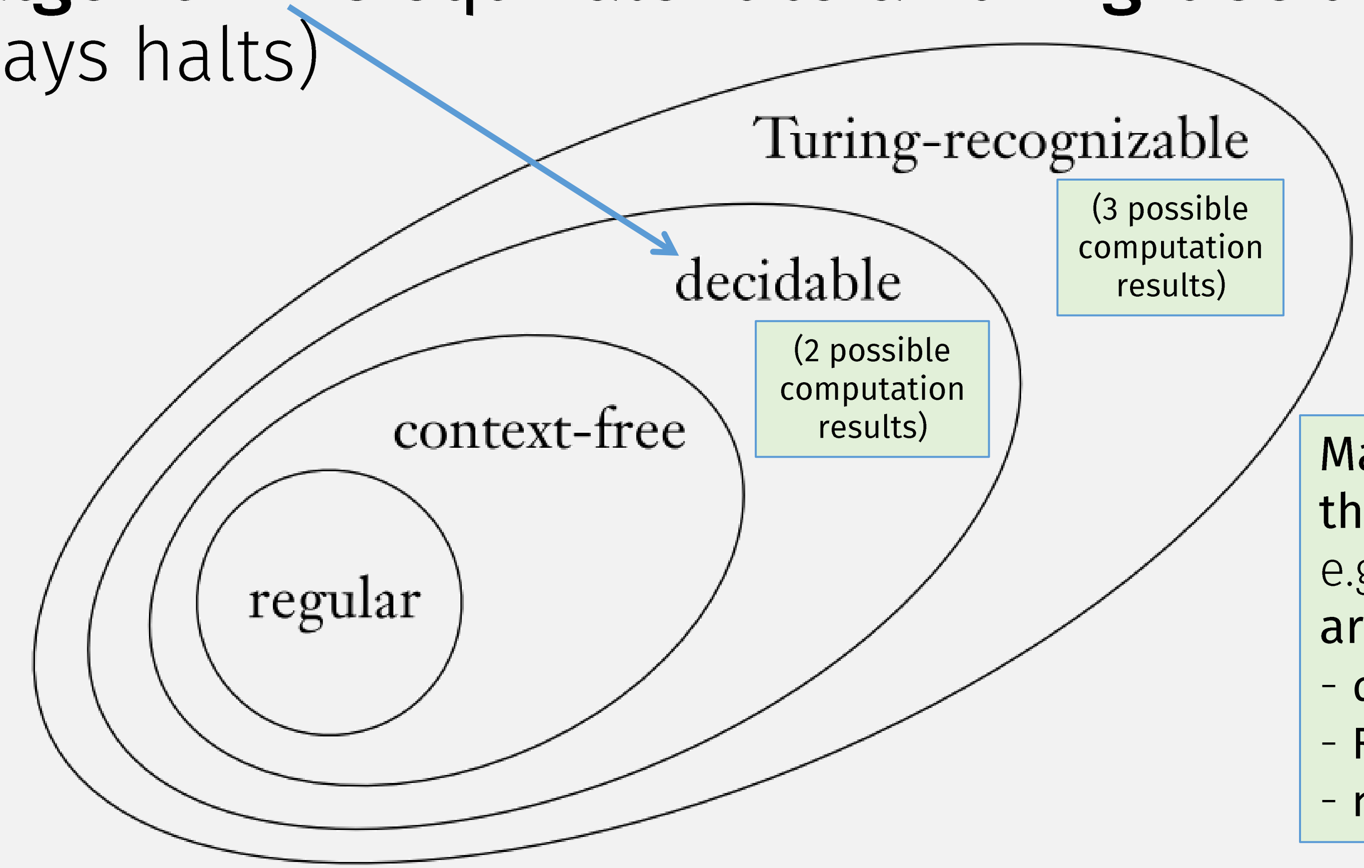
(3 possible computation results)

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

(2 possible computation results)

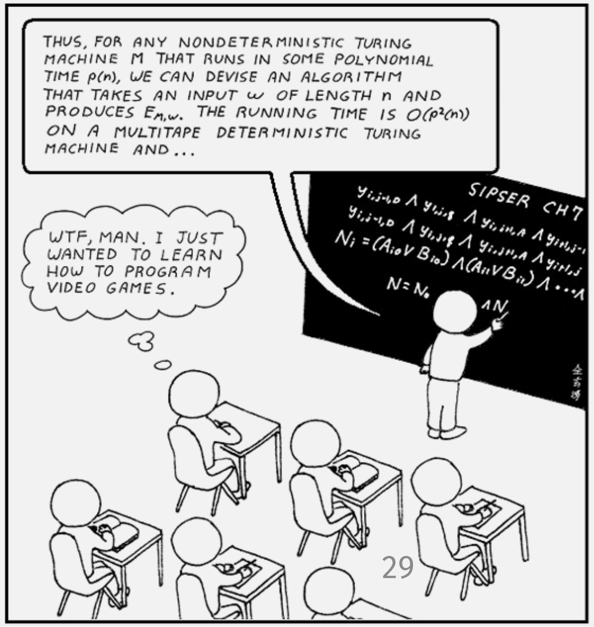
Formal Definition of an "Algorithm"

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

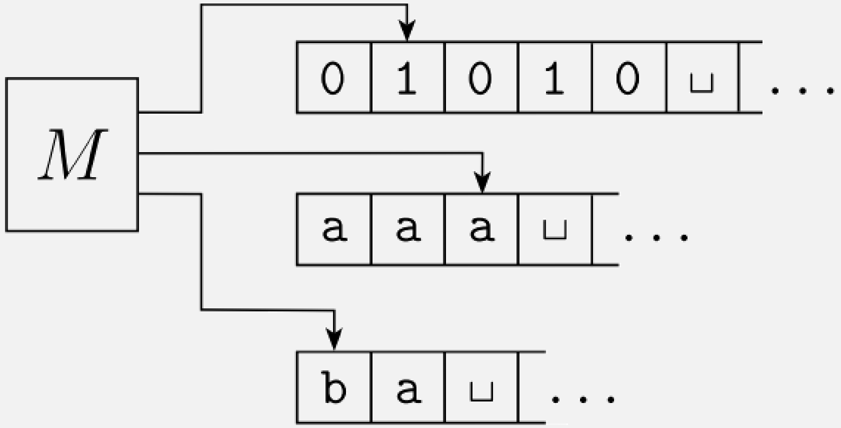


Many <u>functions</u> we have defined this semester are **algorithms**!
e.g., all our conversion functions are **deciders**!!
- d2n
- RE2NFA
- n2p

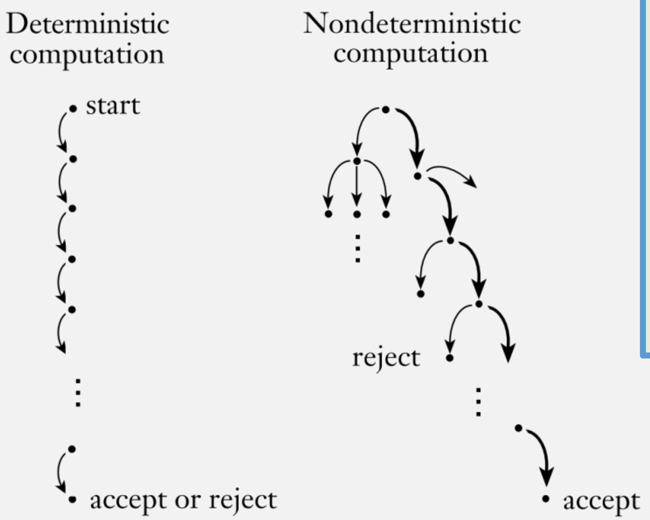
More Turing Machine Variations



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?)

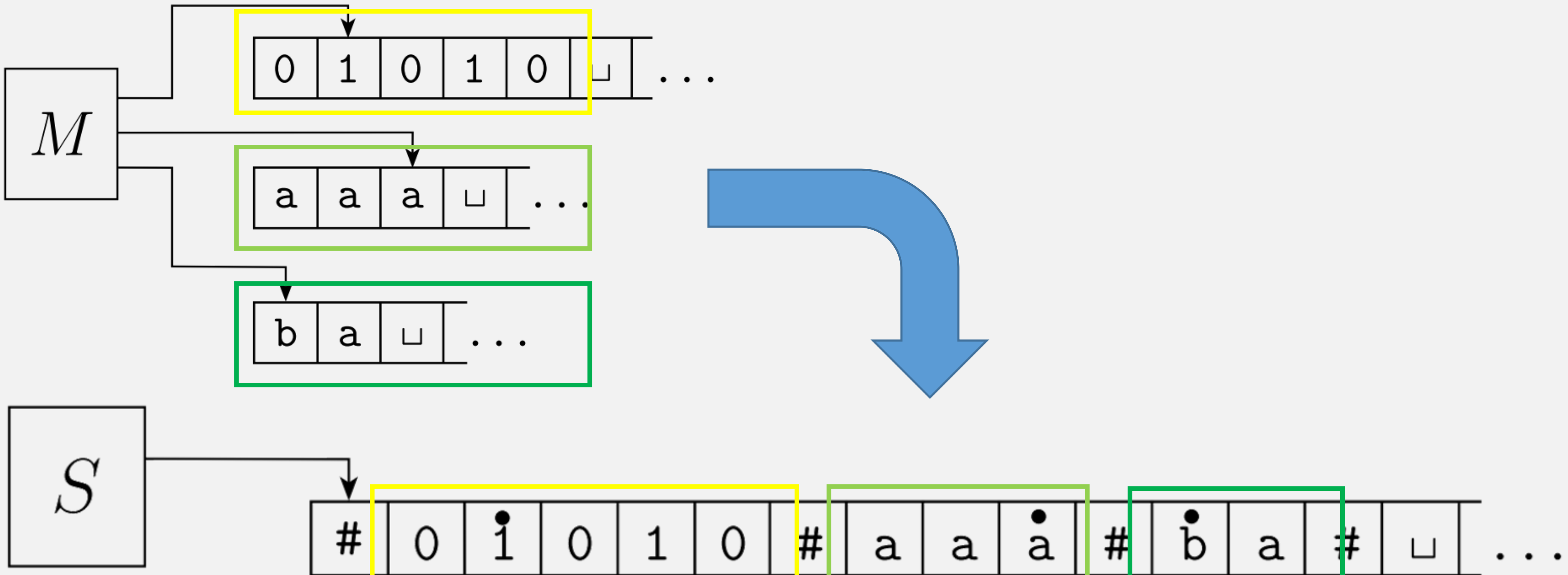
⇐ If a multi-tape TM recognizes a language, then a single-tape TM recognizes the language

- 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

■

Check-in Quiz 3/27

On gradescope