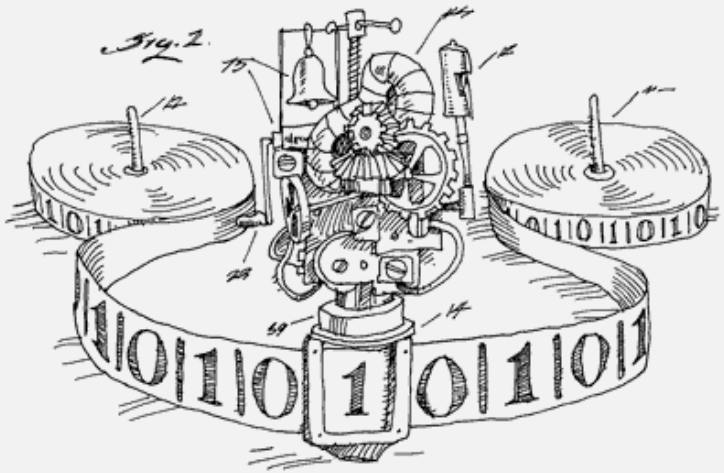
CS 420

Turing Machines (TMs)

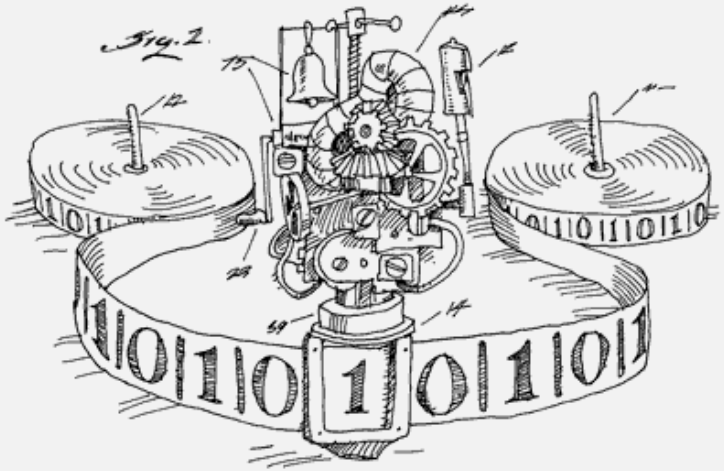
Tuesday, November 3, 2026

UMass Boston Computer Science



Announcements

- HW 7
  - Due: Thurs 11/5 12:30pm EST



In-class questions (in Gradescope)

Q1 TM possible results
2 Points

Q1.1 TM number of possible results
1 Point

When a Turing Machine (TM) starts running with an input string, how many different possible results can there be.

Q1.2 TM computation results
1 Point

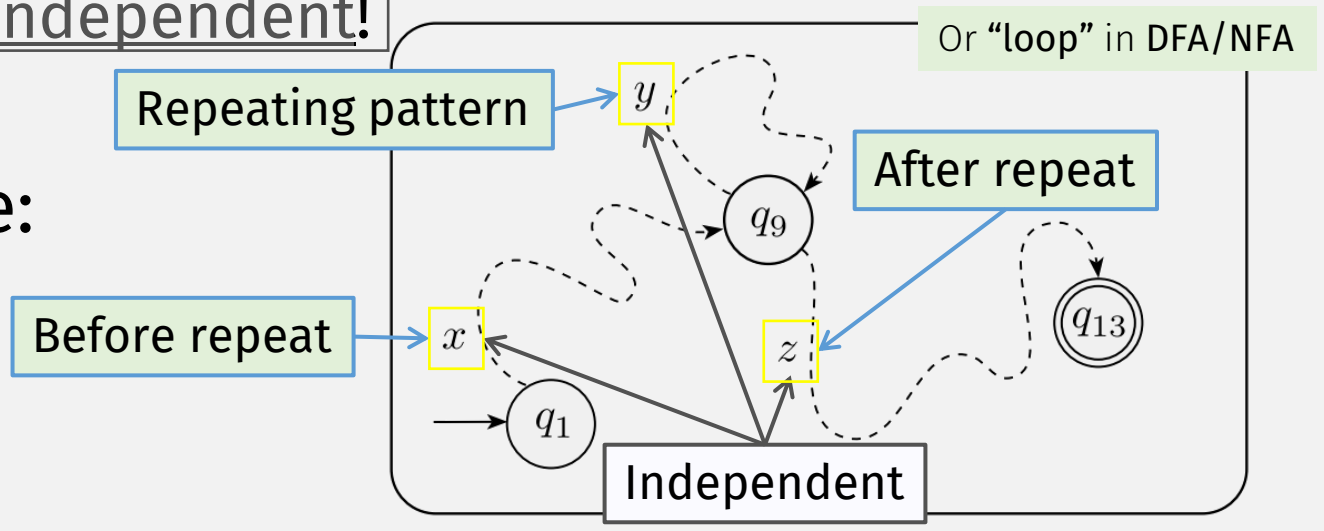
What are the possible results when a TM is run with a string input?

Flashback: Pumping Lemma for Regular Langs

- **Pumping Lemma** explains how strings repeat
- **Regular language** strings repeat using **Kleene star** operation
  - Key: 3 substrings $x\,y\,z$ independent!
- A non-regular language:

$$\{0^n 1^n \mid n \geq 0\}$$

Kleene star can't express this pattern:
2nd part depends on (length of) 1st part



- Q: How do CFLs repeat?

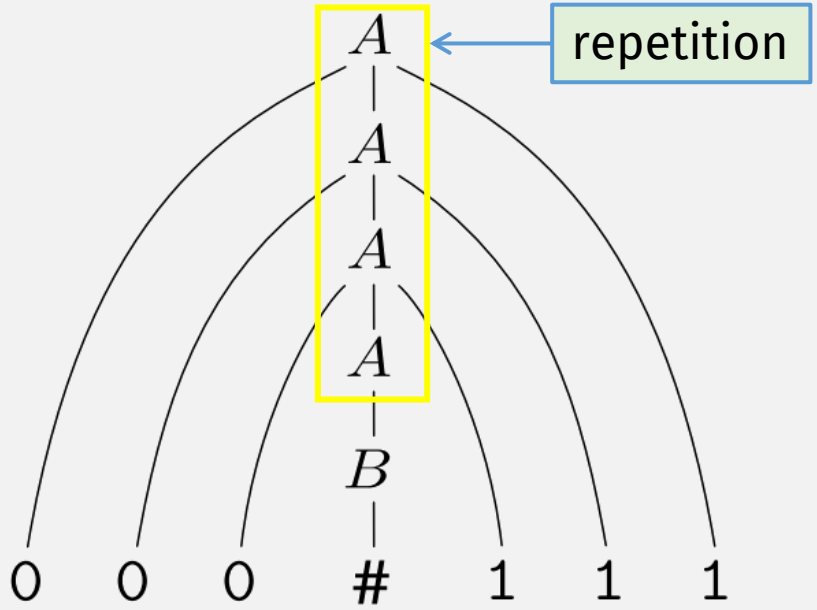
Repetition and Dependency in CFLs

Parts before/after **repetition point** **linked** (not independent)

Repetition

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

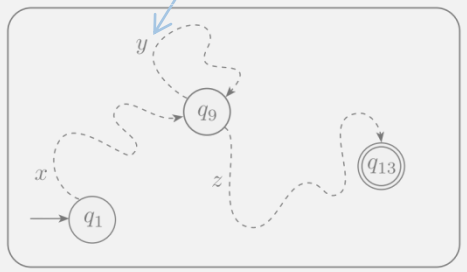
$$\{0^n\#1^n \mid n \geq 0\}$$



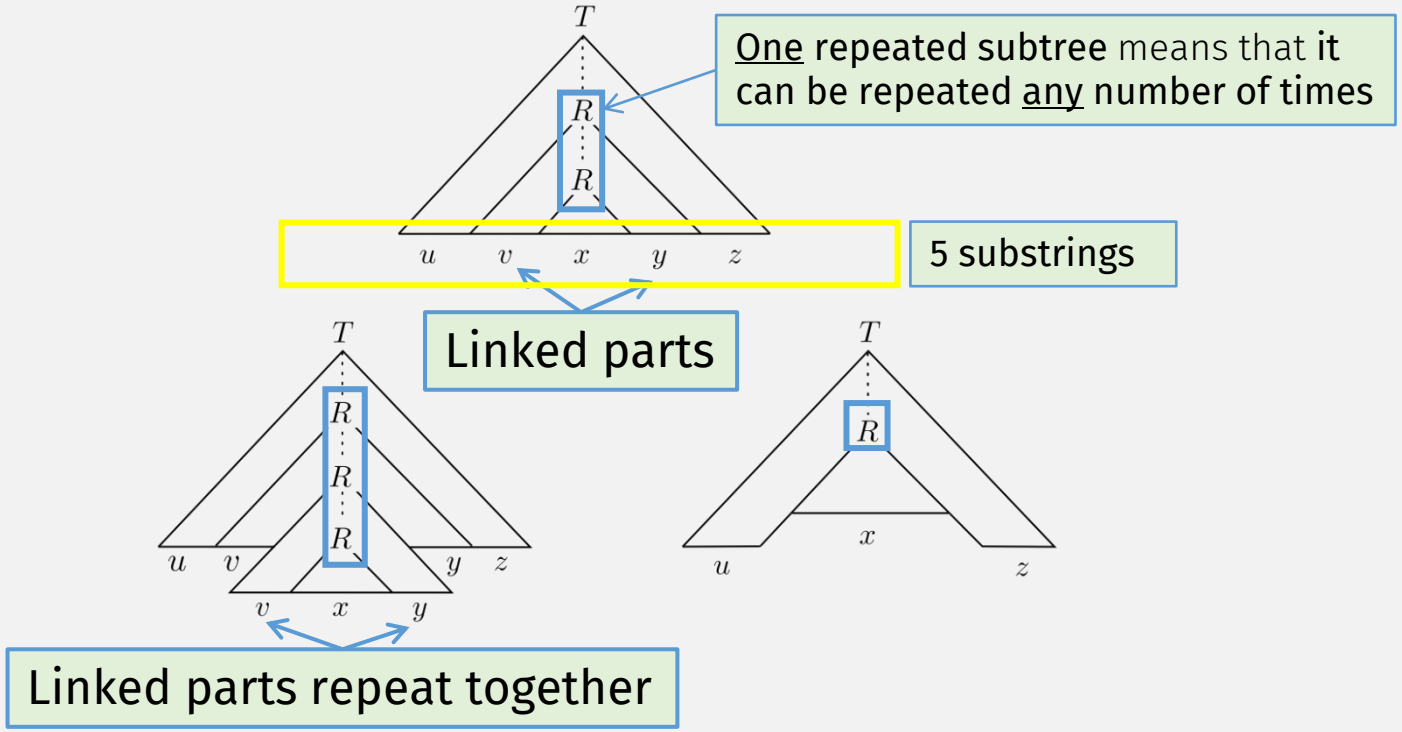
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

How Do Strings in CFLs Repeat?

DFA can take loop transition(s) any number of times, to process repeated y in input

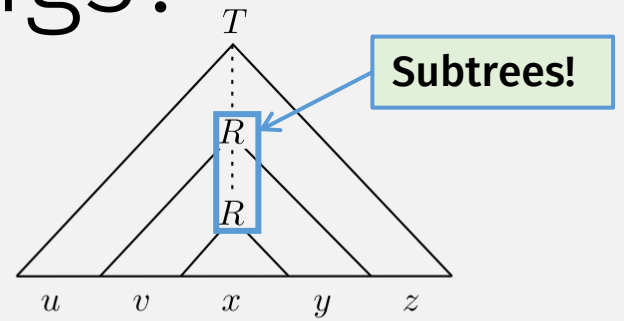


- Strings in regular languages repeat states
- **Strings** in **CFLs repeat subtrees** in the **parse tree**

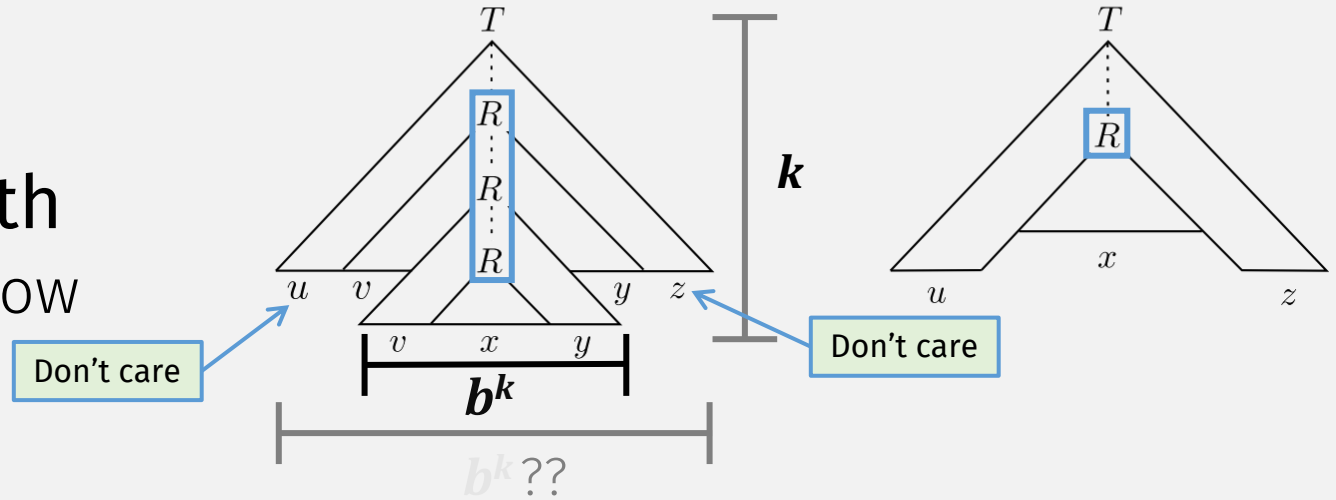


Repeating Pattern in CFL Strings?

- When are we **guaranteed** to have a **repeated subtree?**
  - When **height** of parse tree > # of rules!



- Let k = # of rules
  and b = longest rule RHS length
  - Then **length string** where we know there's a **repeated rule** is … b^k
  - I.e., "**pumping length**" $p = b^k$???



Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

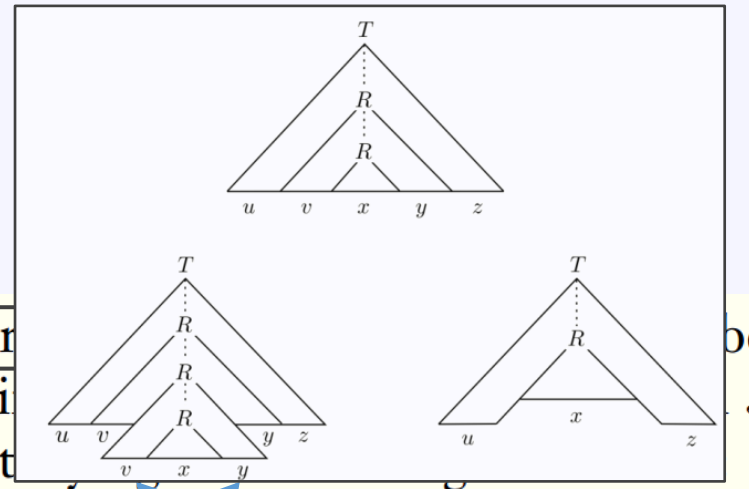
1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Pumping Lemma for CFLS

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

Two pumpable parts.
But they must be pumped together!

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.



Pumping lemma If A is a regular [illegible] ber p (the pumping length) where if s is any stri [illegible] s may be divided into three pieces, $s = xyz$, sat [illegible]

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

One pumpable part

Two pumpable parts, pumped together

Previously

A Non CFL example

$$\text{language } B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\} \text{ is not context free}$$

Intuition

- **Strings in CFLs** can have **two parts** that are **"pumped" together**
- **Language B requires three parts** to be **"pumped" together**
- So **it's not a CFL!**

Proof?

Want to prove: $a^n b^n c^n$ **is not** a CFL

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Now we must find a contradiction …

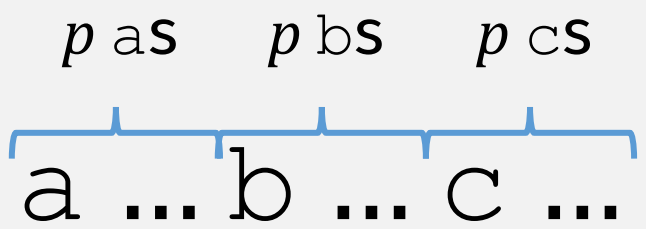
Reminder: CFL Pumping lemma says:
all strings $a^n b^n c^n \geq$ length p are splittable into $uvxyz$ where v and y are pumpable

Proof sketch (by contradiction):

- **Assume:** $a^n b^n c^n$ **is a CFL**
  - So it must satisfy the pumping lemma for CFLs
  - I.e., strings in lang ≥ length p are pumpab
- **Counterexample** = $a^p b^p c^p$

Contradiction if:
- A string in the language ☑
- ≥ length p ☑
- Is **not splittable** into $uvxyz$ where v and y are pumpable

???



Want to prove: $a^n b^n c^n$ **is not** a CFL

Possible Splits

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Reminder: CFL Pumping lemma says:
all strings $a^n b^n c^n \geq$ length p are splittable into $uvxyz$ where v and y are pumpable

Proof sketch (by contradiction):

- Assume: $a^n b^n c^n$ **is** a CFL
  - So it must satisfy the pumping lemma for CFLs
  - I.e., all strings ≥ length p are pumpable
- Counterexample = $a^p b^p c^p$

Assumption is false

contradiction

Contradiction if:
- A string in the language
- ≥ length p
- Is **not splittable** into $uvxyz$ where v and y are pumpable

- Possible Splits (using condition # 3: $|vxy| \leq p$)
  - ✗ vxy is all as
  - ✗ vxy is all bs
  - ✗ vxy is all cs
  - ✗ vxy has as and bs
  - ✗ vxy has bs and cs
  - (vxy cannot have as, bs, and cs)

Not pumpable

p as p bs p cs

a ... b ... c ...

vxy ???

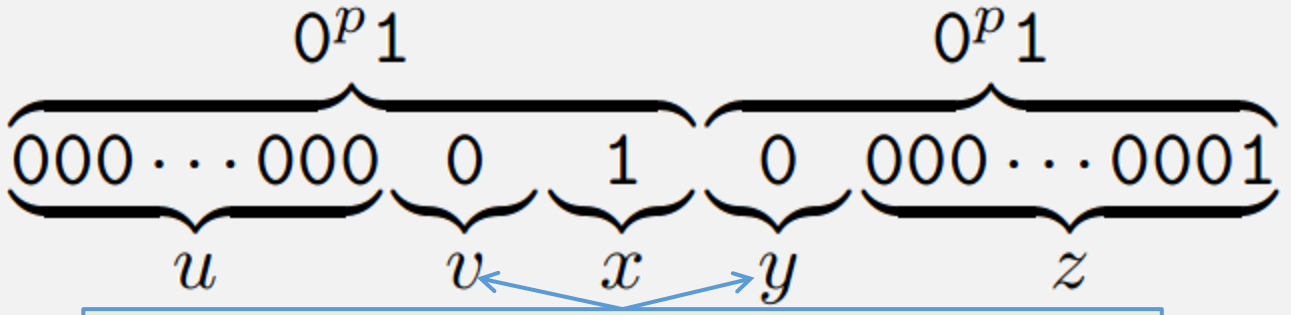
$a^p b^p c^p$ **cannot be split** into $uvxyz$ where v and y are pumpable

So $a^n b^n c^n$ **is not a CFL**

Another Non-CFL $D = \{ww | \ w \in \{0,1\}^*\}$

Be careful when choosing **counterexample** s: 0^p10^p1

This s **can** be pumped according to **CFL pumping lemma:**



Pumping v and y (together) produces string still in D!

- CFL Pumping Lemma conditions:
  - ☑ **1.** for each $i \geq 0$, $uv^ixy^iz \in A$,
  - ☑ **2.** $|vy| > 0$, and
  - ☑ **3.** $|vxy| \leq p$.

No contradiction!

So **this attempt** to **prove** that the **language** is **not** a **CFL failed**.
(It doesn't prove that the language is a CFL!)

Another Non-CFL $D = \{ww | \ w \in \{0,1\}^*\}$

… so this language is **not a CFL!**

- Need another counterexample string s:

If vyx is contained in **first or second half**, then **any pumping will break the match** ✗

$$0^p 1^p 0^p 1^p$$

So vyx must **straddle the middle** ✗

But **any pumping still breaks the match** because **order is wrong**

- CFL Pumping Lemma conditions:

1. for each $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Now we have proven that **the counterexample is not pumpable** …

A Practical Non-CFL

- **XML**
  - ELEMENT → <TAG>CONTENT</TAG>
  - Where TAG **is any string**

- XML also looks like this non-CFL: $D = \{ww | \ w \in \{0,1\}^*\}$

- This means **XML is not context-free!**
  - Note: **HTML** *is* **context-free** because …
  - … there are only a **finite number of tags,**
  - so **they can be embedded** into **a finite number of rules.**

In practice:
  - **XML is parsed** as a **CFL,** with a **CFG**
  - Then **matching tags checked in a 2nd pass with a more powerful machine …**

Next: A More Powerful Machine …

M_1 accepts its input if it 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 not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.

Infinite memory (initial contents are the input string)

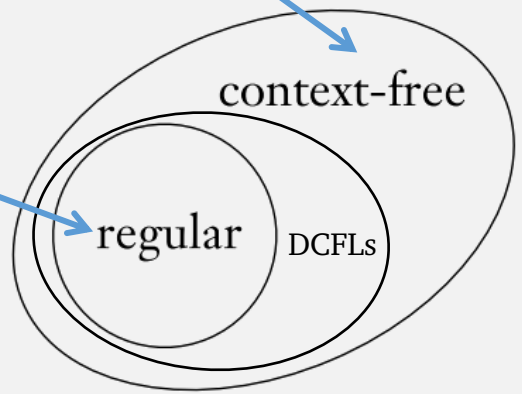
Can move to, and read/write from arbitrary memory locations!

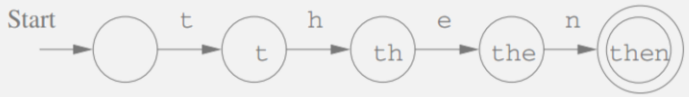
Where We've Been, Where We're Going

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

$A \to 0A1$
$A \to B$
$B \to \#$

- **DFAs / NFAs**: recognize **regular langs**
  - **Memory**: finite states
  - Can't express: dependency
    e.g., $\{0^n 1^n \mid n \geq 0\}$





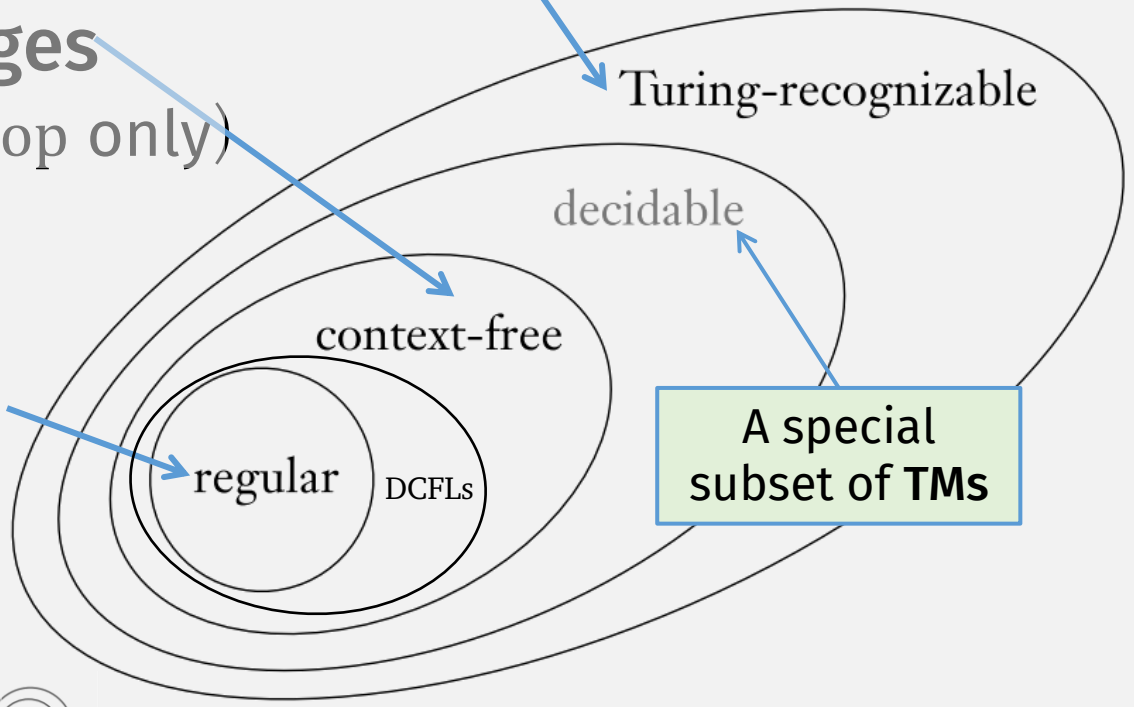
Where We've Been, Where We're Going

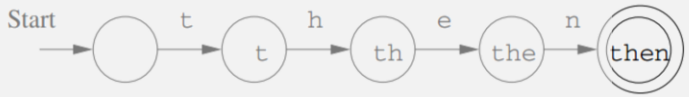
- **Turing Machines** (TMs)
  - **Memory**: states + infinite **tape**, (arbitrary read/write)
  - Expresses any "computation"
- **PDAs**: recognize **context-free languages**
  - **Memory**: states + infinite **stack** (push/pop only)
  - Can't express: arbitrary dependency,
    - e.g., $\{ww \mid w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

$A \to 0A1$
$A \to B$
$B \to \#$

- **DFAs / NFAs**: recognize **regular langs**
  - **Memory**: finite states
  - Can't express: dependency
    e.g., $\{0^n 1^n \mid n \geq 0\}$







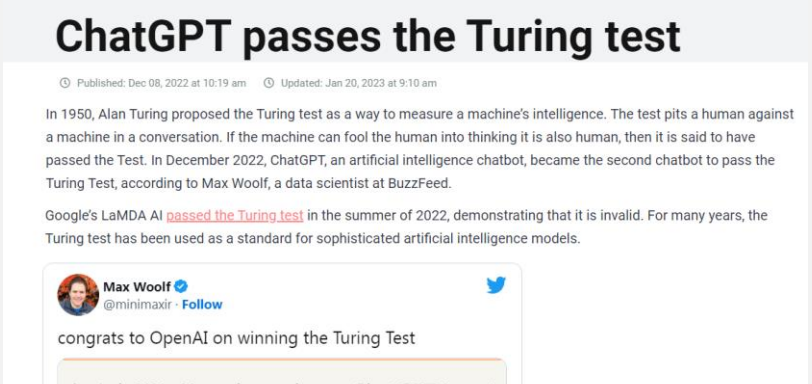
Alan Turing

- First to **formalize a model of computation**
  - I.e., **he invented many of the ideas in this course!**



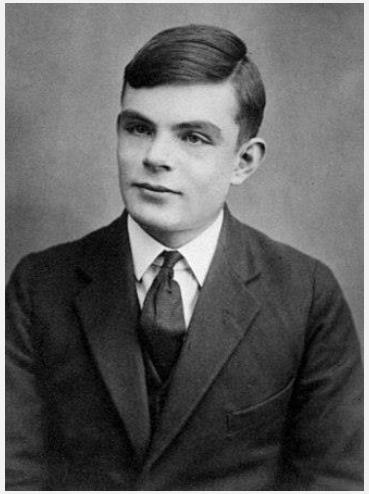
- And **worked as a codebreaker during WW2**
- Also **studied Artificial Intelligence**
  - **The Turing Test**





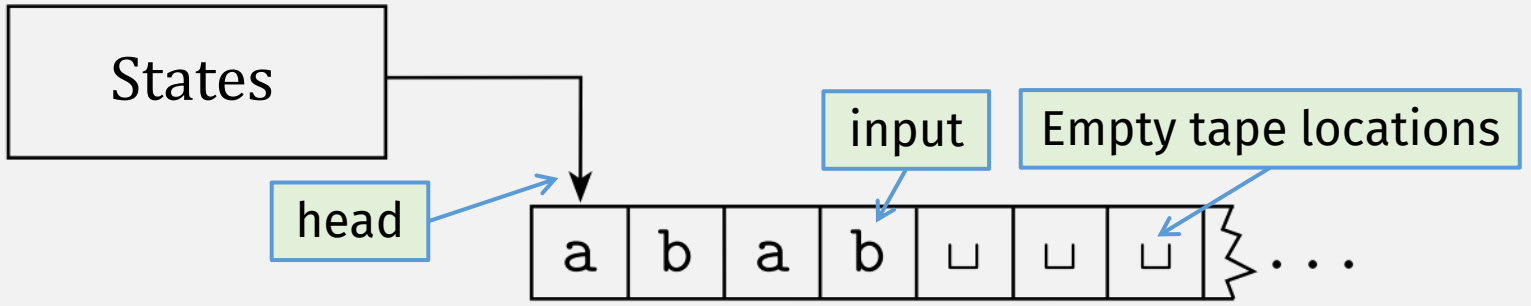






Finite Automata vs Turing Machines

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



- Each step: "**head**" can **move left** or **right**
- **Turing Machine** can **accept** / **reject** at **any time**

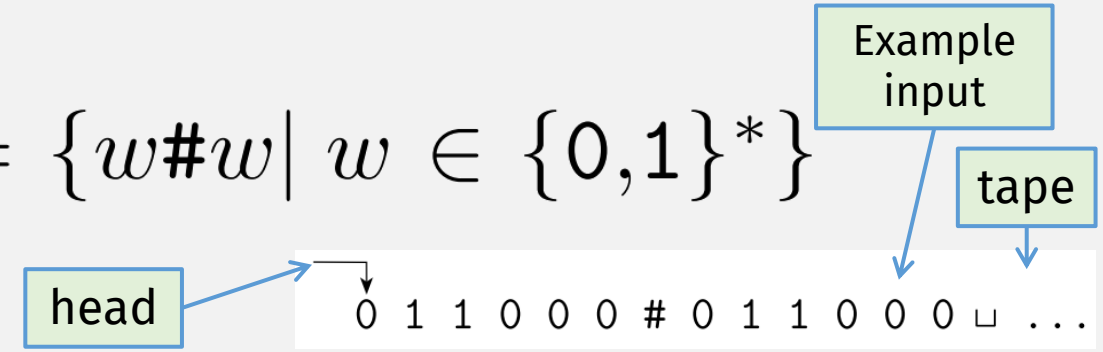
DefRecognizeLang

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

Turing Machine Example

Want:

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



let 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

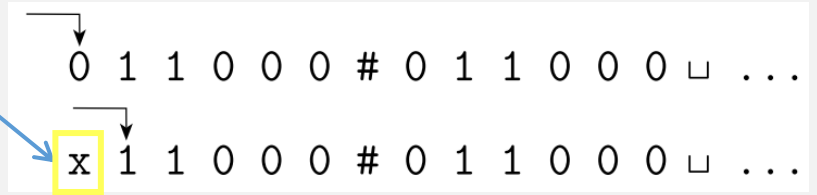
Want:

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

let 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

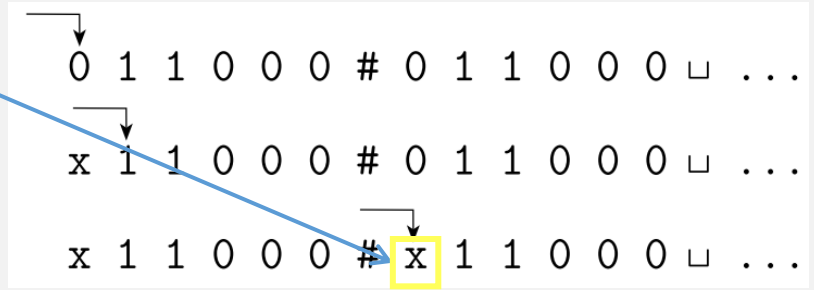
Want:

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

let 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

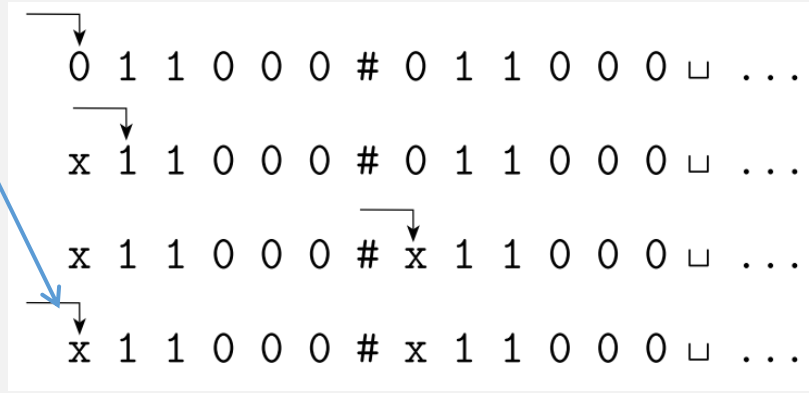
Want:

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

let 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

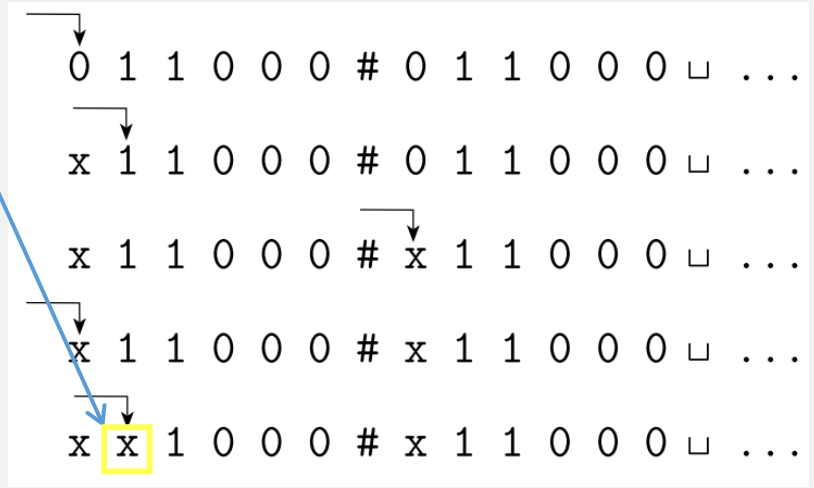
Want:

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

let 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

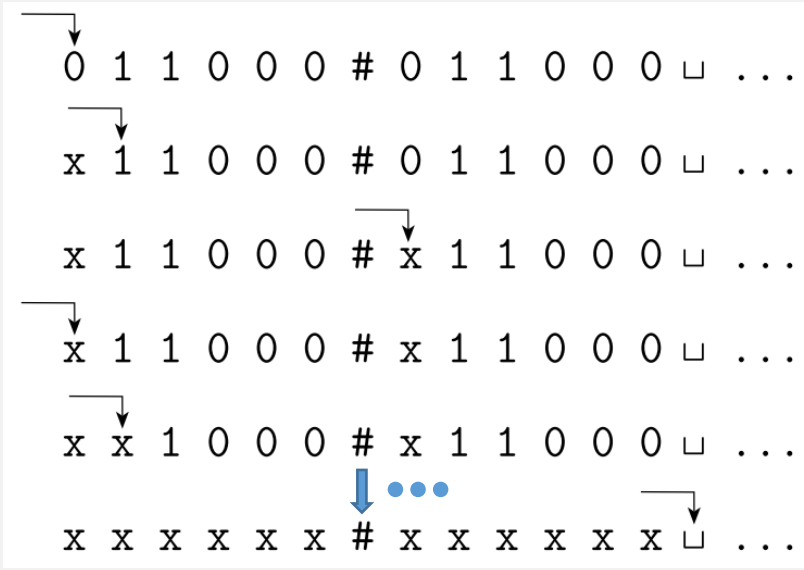
Want:

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

let 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

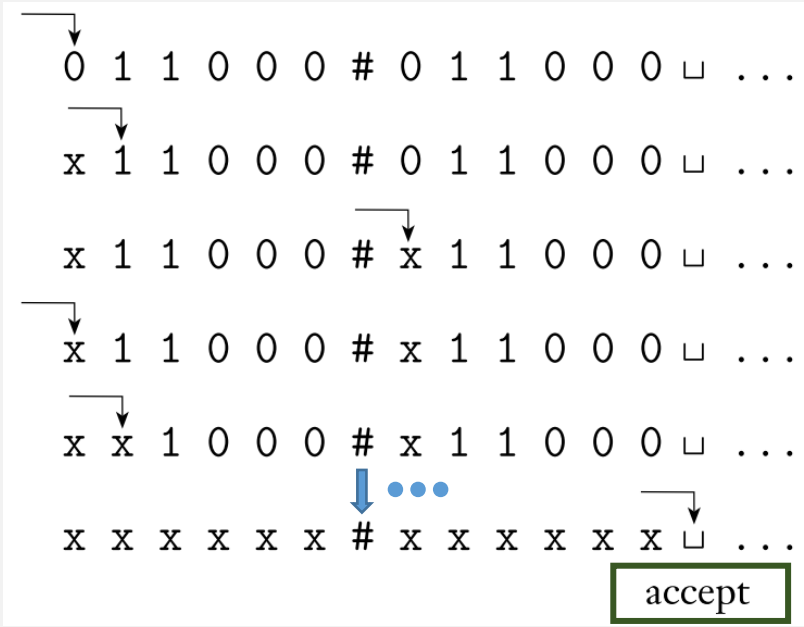
Want:

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

let 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

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

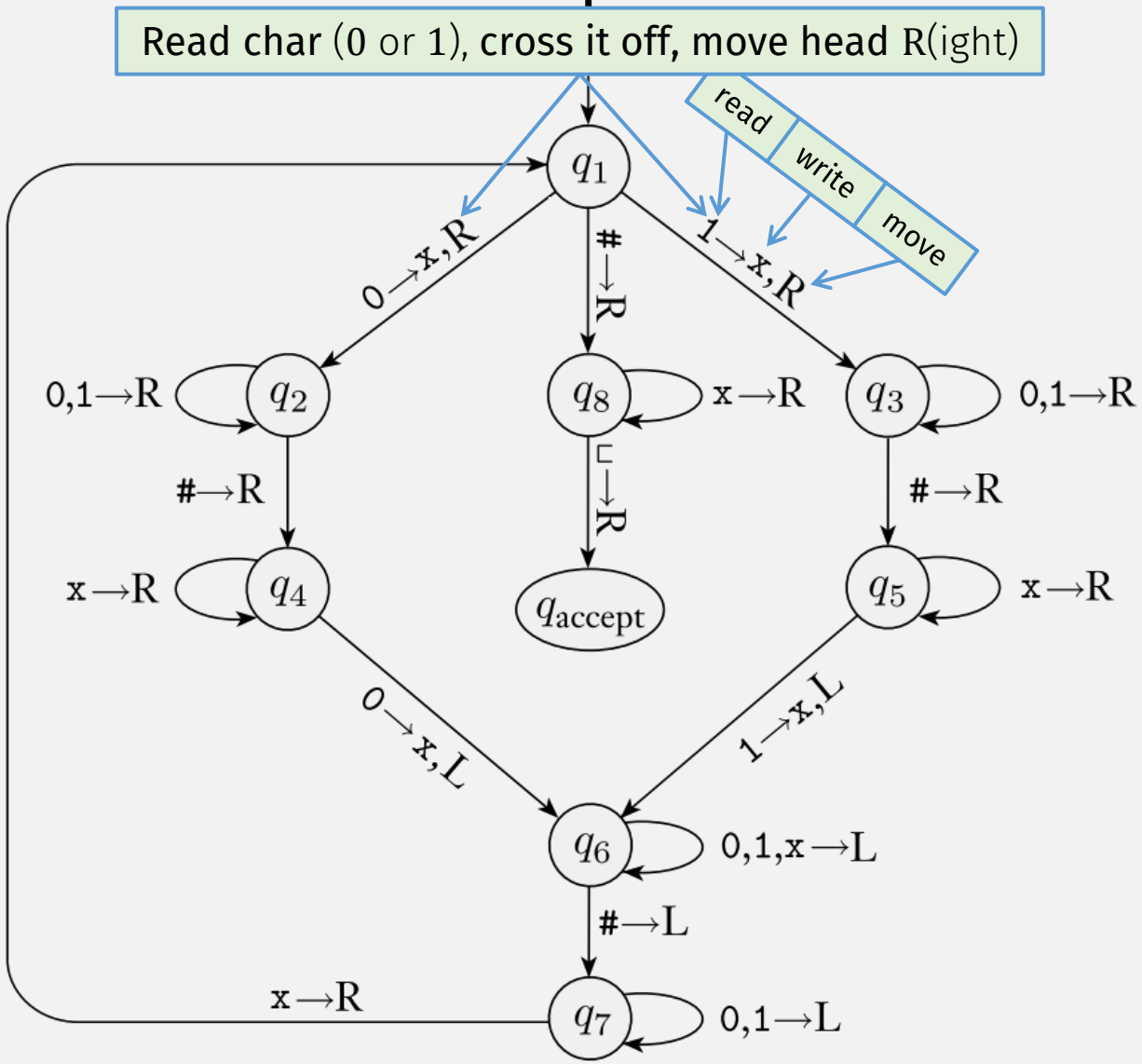
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,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

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

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

Formal Turing Machine Example



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

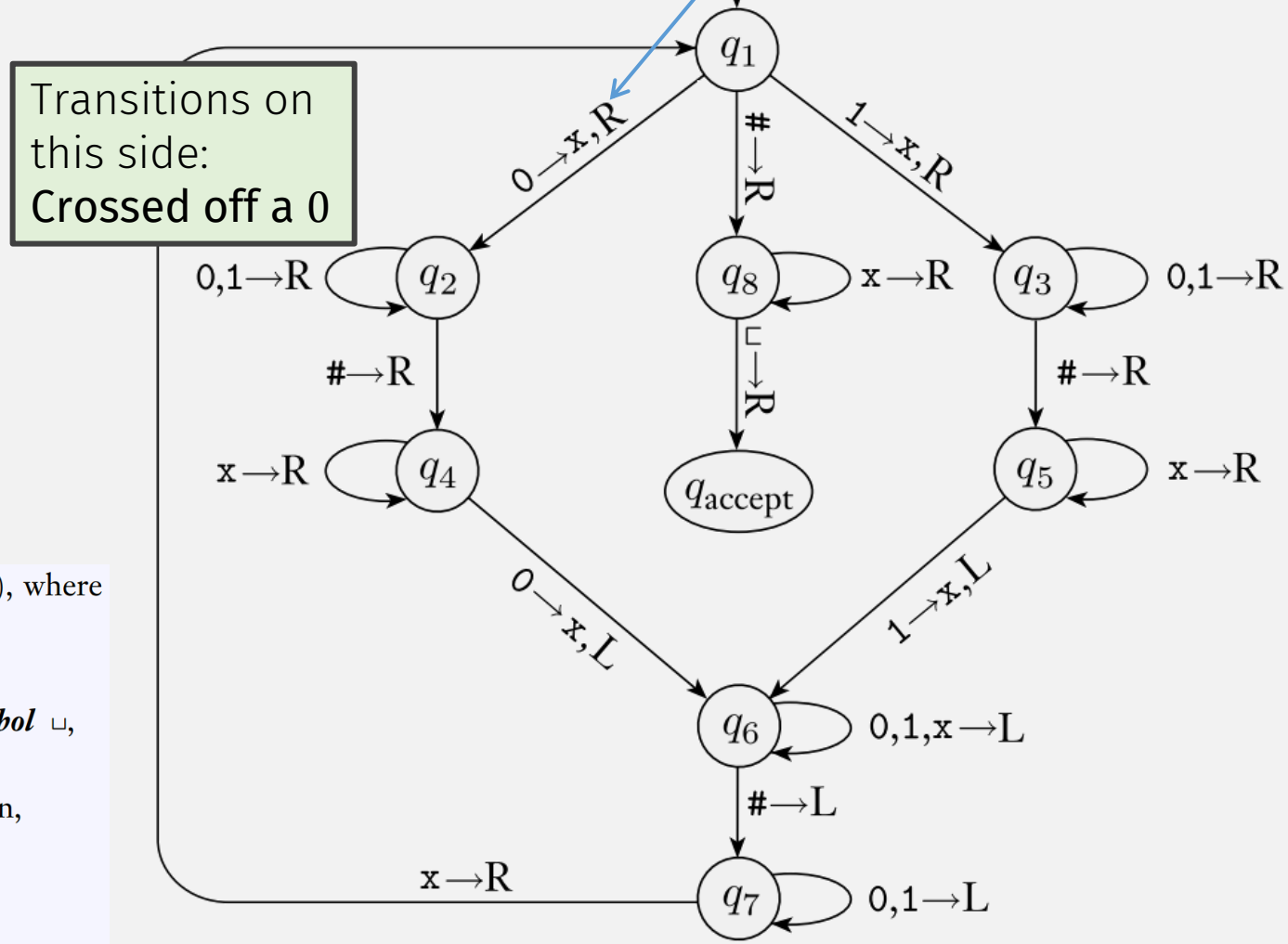
$$B = \{w\#w \mid w \in \{0,1\}^*\}$$

Formal Turing Machine Example



Read char (0 ~~or 1~~), cross it off, move head R(ight)

Transitions on this side:
Crossed off a 0

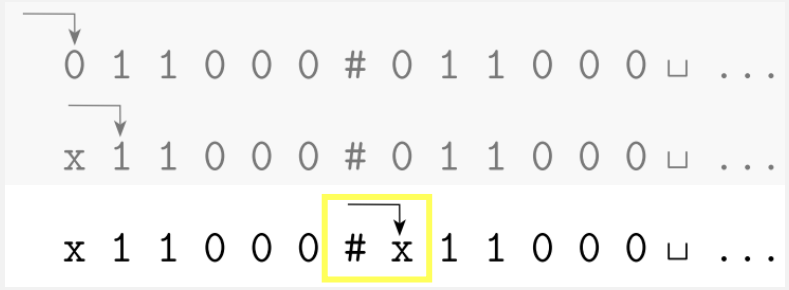


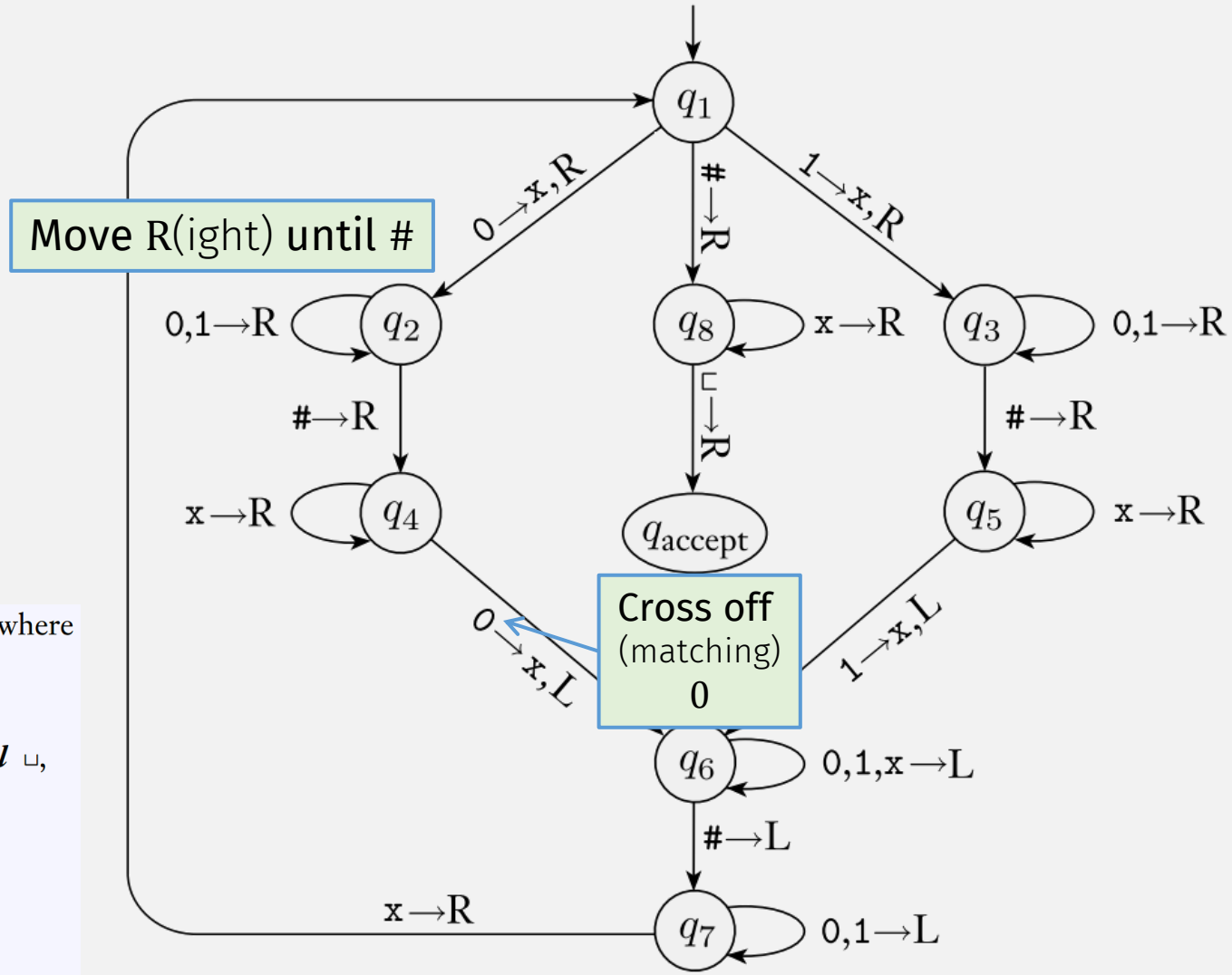
A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

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

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

Formal Turing Machine Example



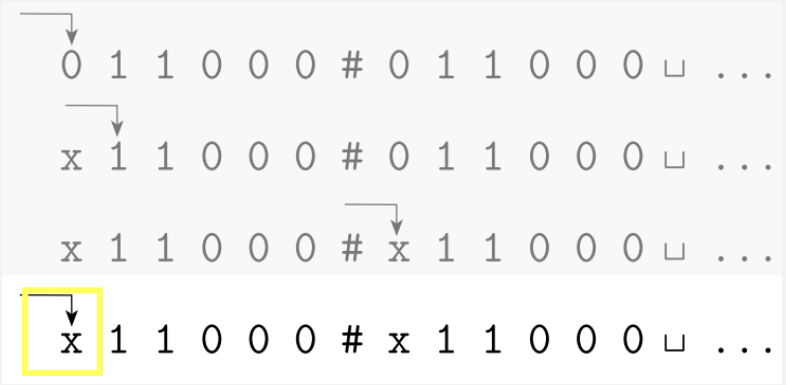


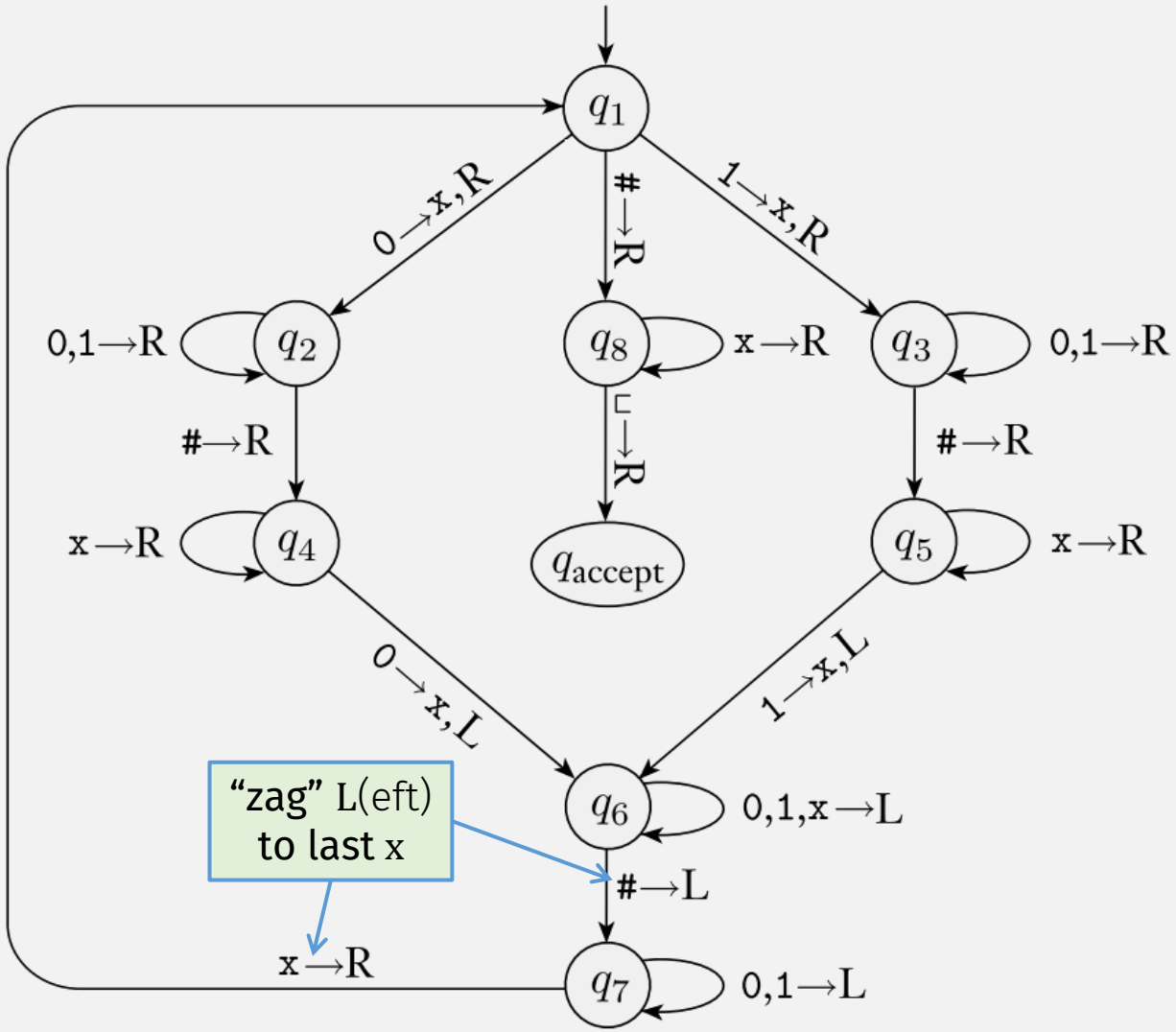
A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

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

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

Formal Turing Machine Example



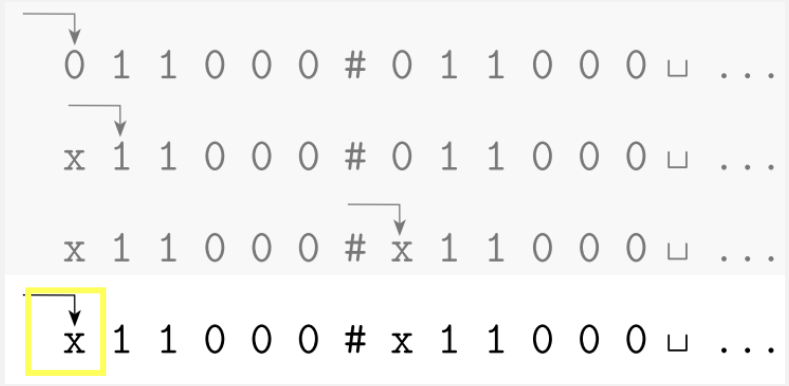


A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

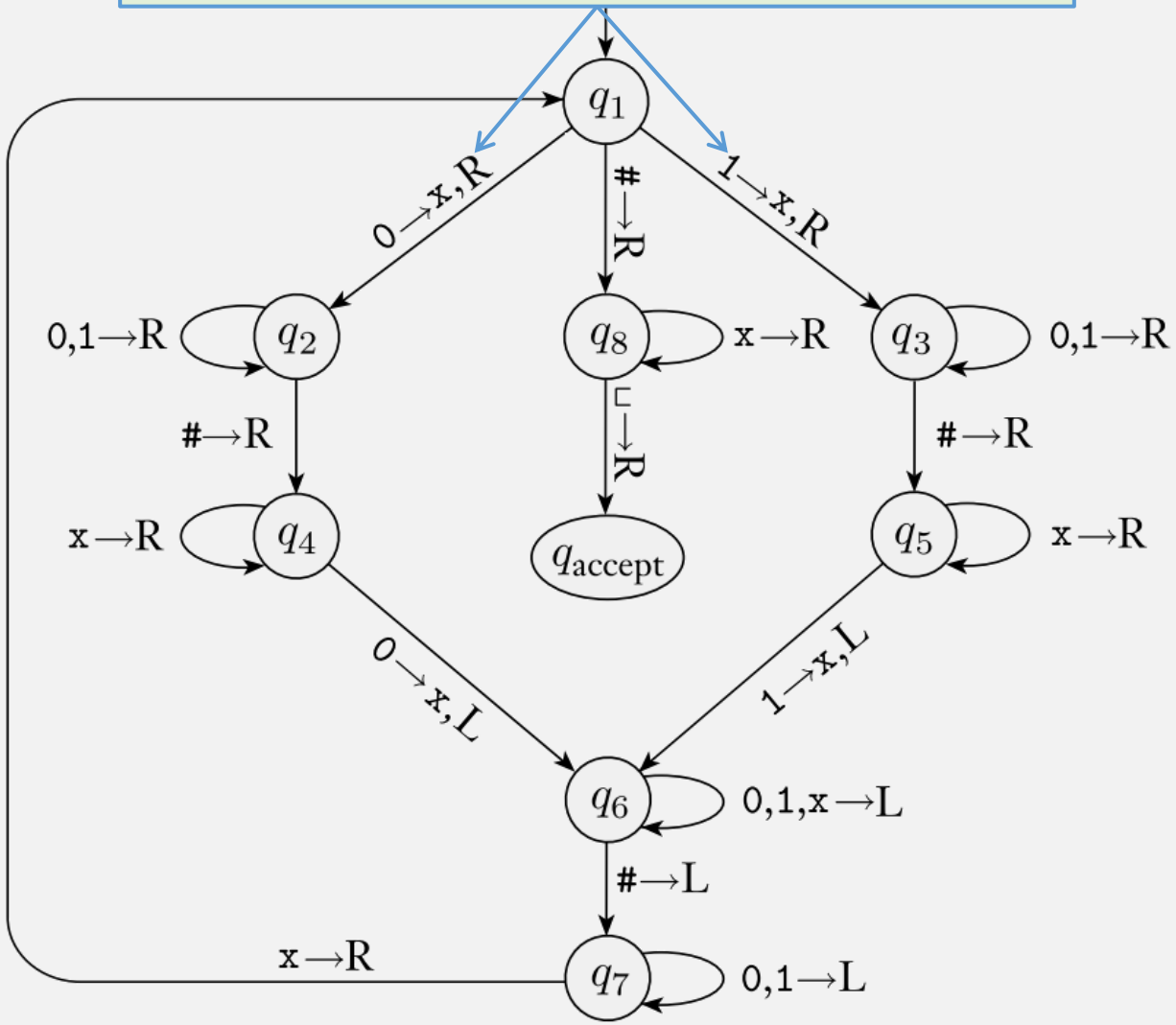
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,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

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

Formal Turing Machine Example



Read char (0 or 1), cross it off, move head R(ight)

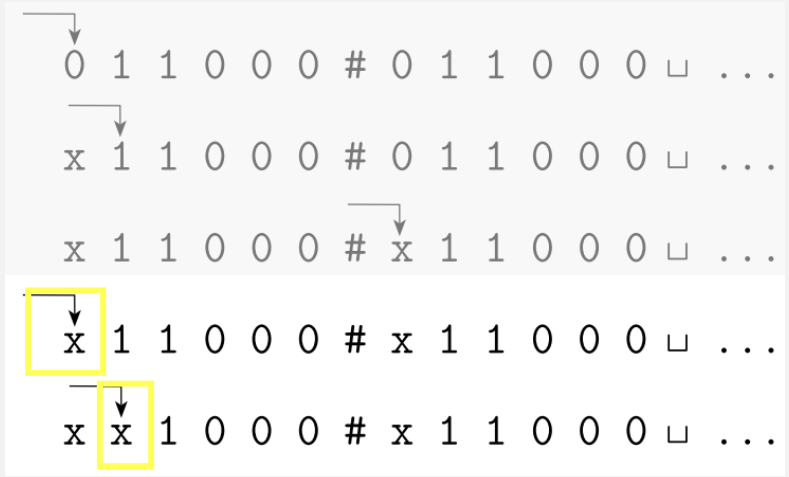


A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

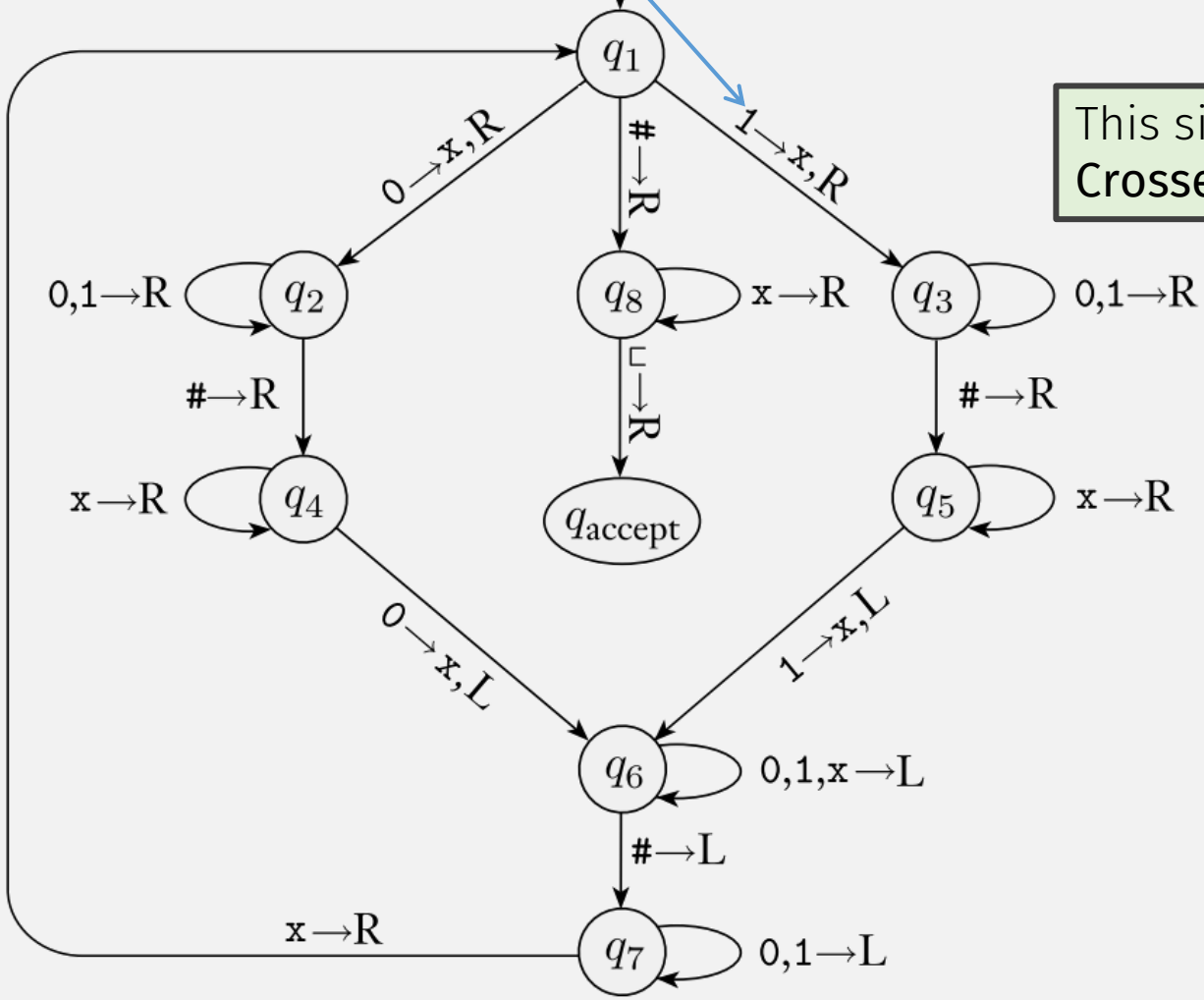
1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,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

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

Formal Turing Machine Example



Read char (~~0 or~~ 1), cross it off, move head R(ight)



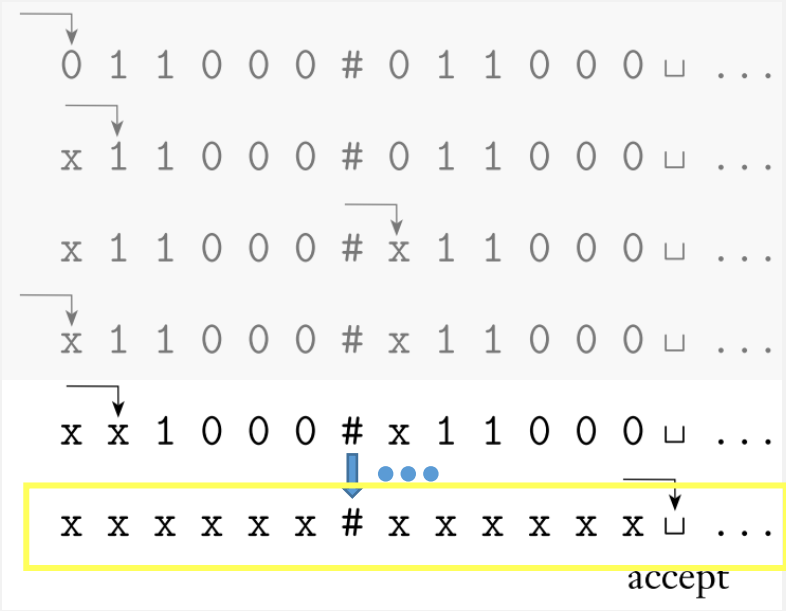
This side:
Crossed off a 1

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

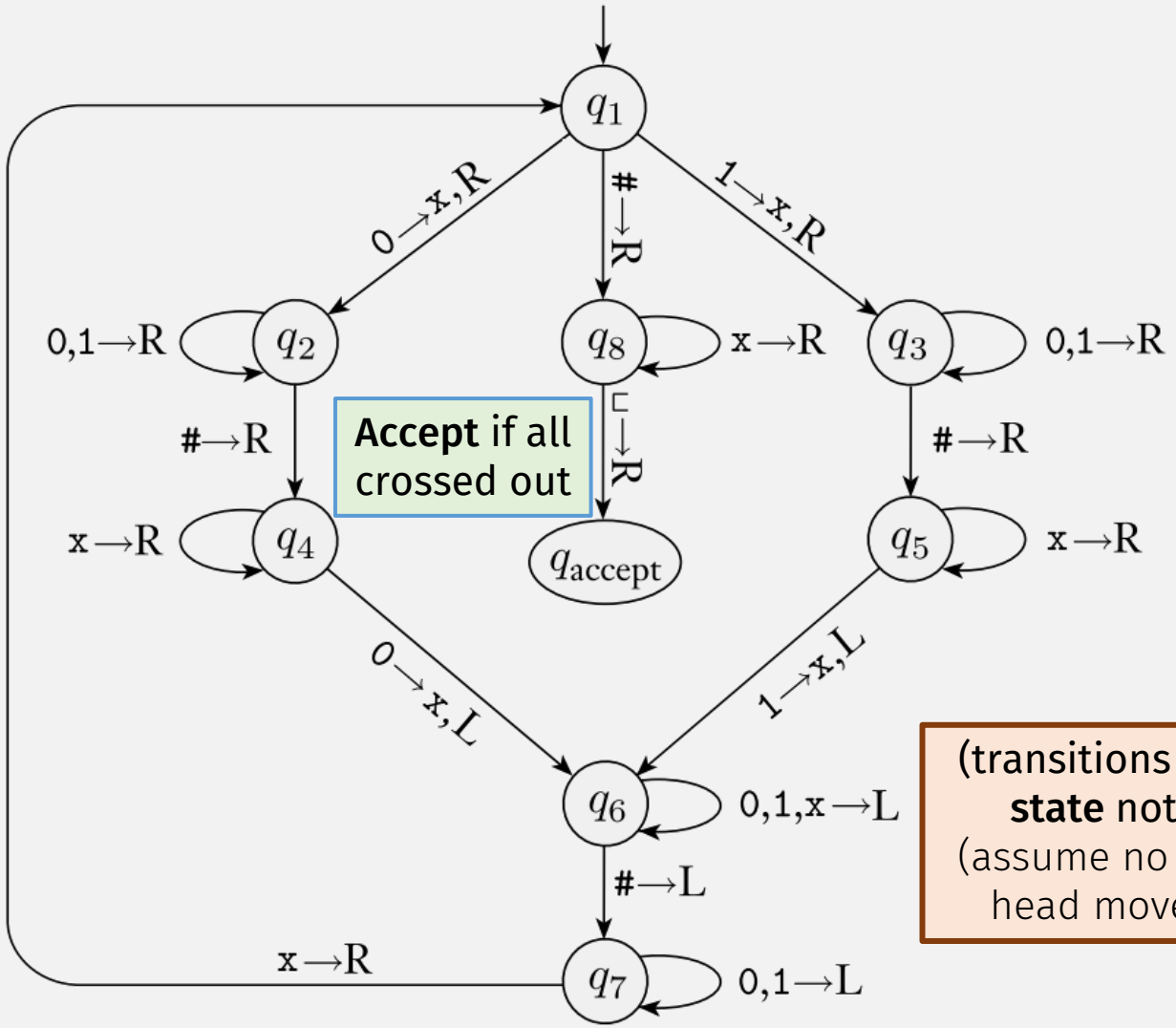
$$B = \{w\#w \mid w \in \{0,1\}^*\}$$

Formal Turing Machine Example



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



(transitions to) **Reject state** not shown (assume no write, and head moves Right)

TM Computation, Formally …

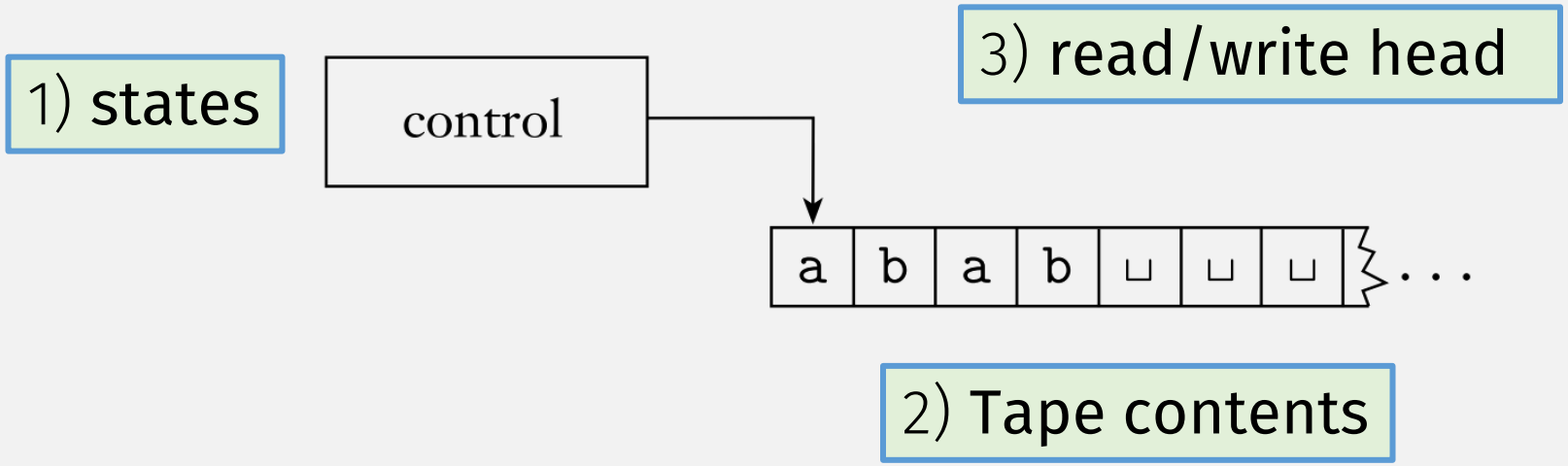
Flashback: PDA Configurations (IDs)

- 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

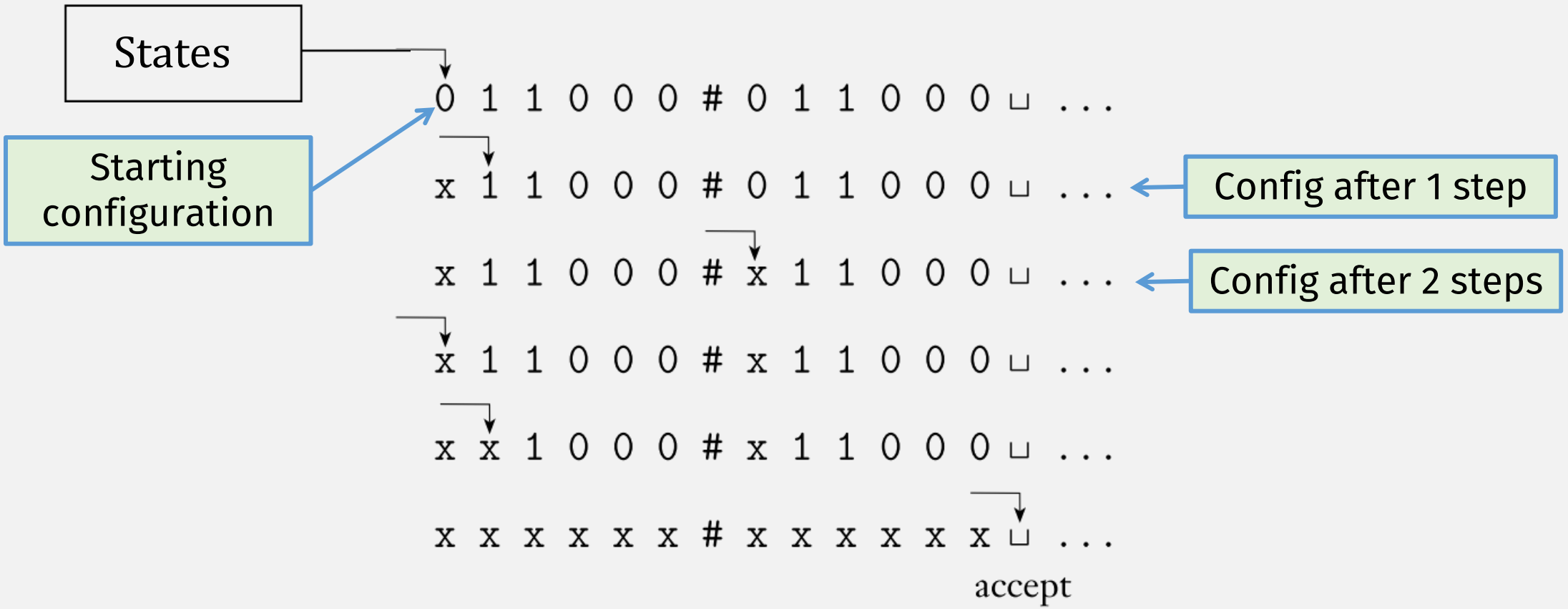
TM Configuration (ID) = ???



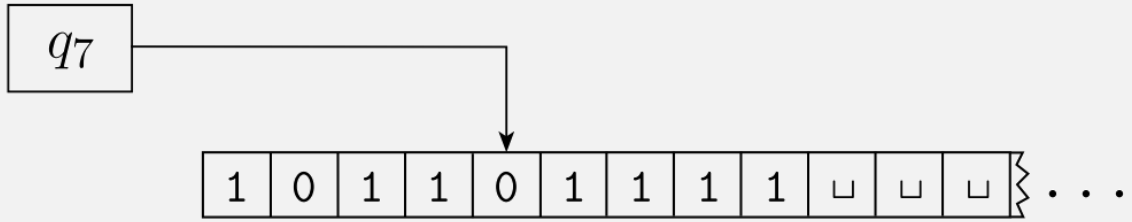
A ***Turing machine*** is a `mk-TM` $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

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:

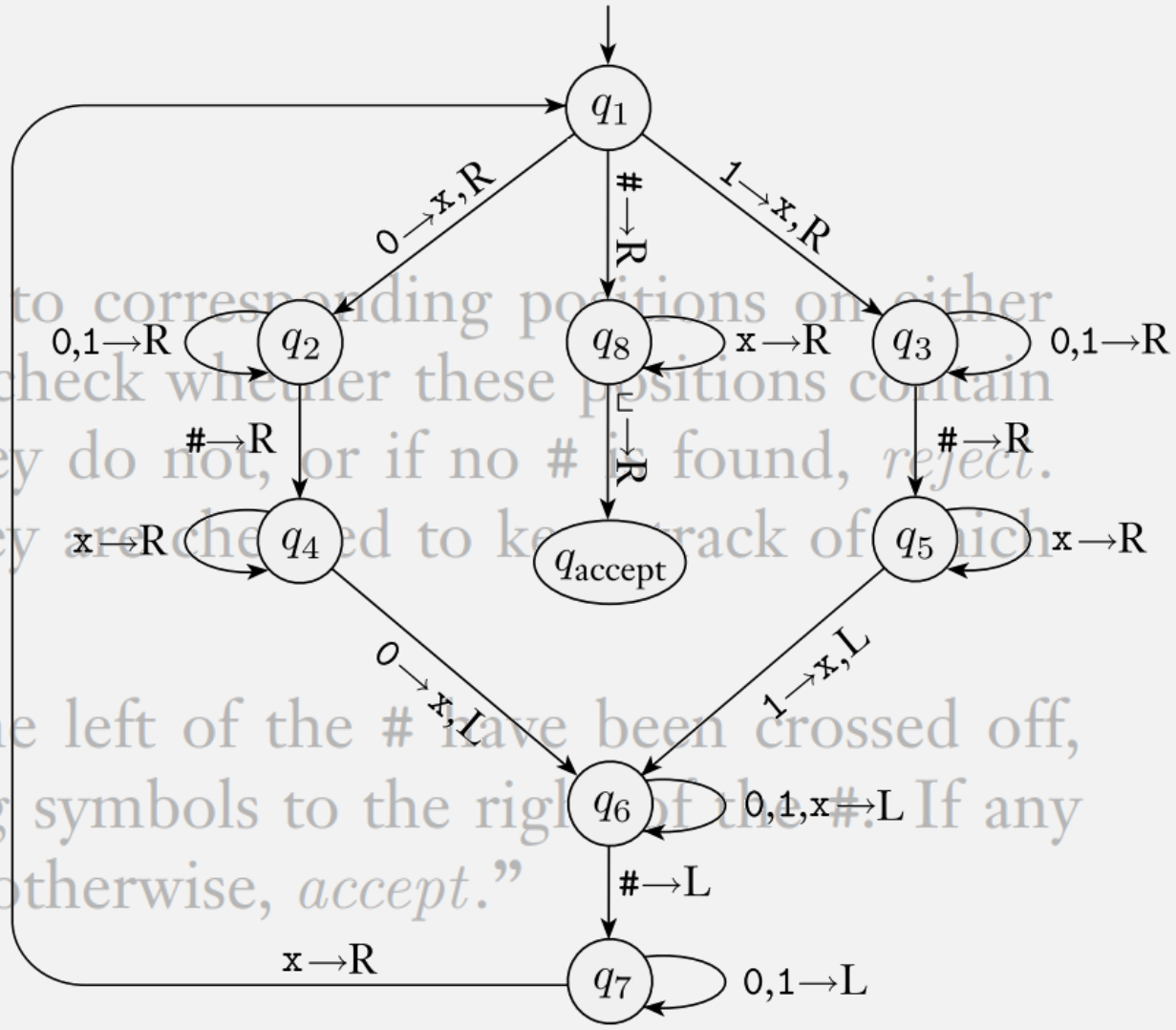
Head stays at leftmost cell — $q_1 \mathbf{a} \beta \vdash q_2 \mathbf{x} \beta$ if $\delta(q_1, \mathbf{a}) = (q_2, \mathbf{x}, \mathrm{L})$ (L move, when already at leftmost cell)

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)

TMs: High-level vs Low-level?

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

let 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 no[illegible] if no # is found, *reject*. Cross off symbols as the[illegible] track of which symbols correspond.
2. When all symbols to[illegible] crossed off, check for any remaining s[illegible] of the #. If any symbols remain, *reject*; otherwise, *accept*."

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 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 an 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 **run forever**
  - E.g., **head** can **move back and forth in a loop**

So: **TM computation** has **3 possible results**:
- Accept
- Reject
- Loop forever

- We will work with **two classes of Turing Machines**:
  - 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

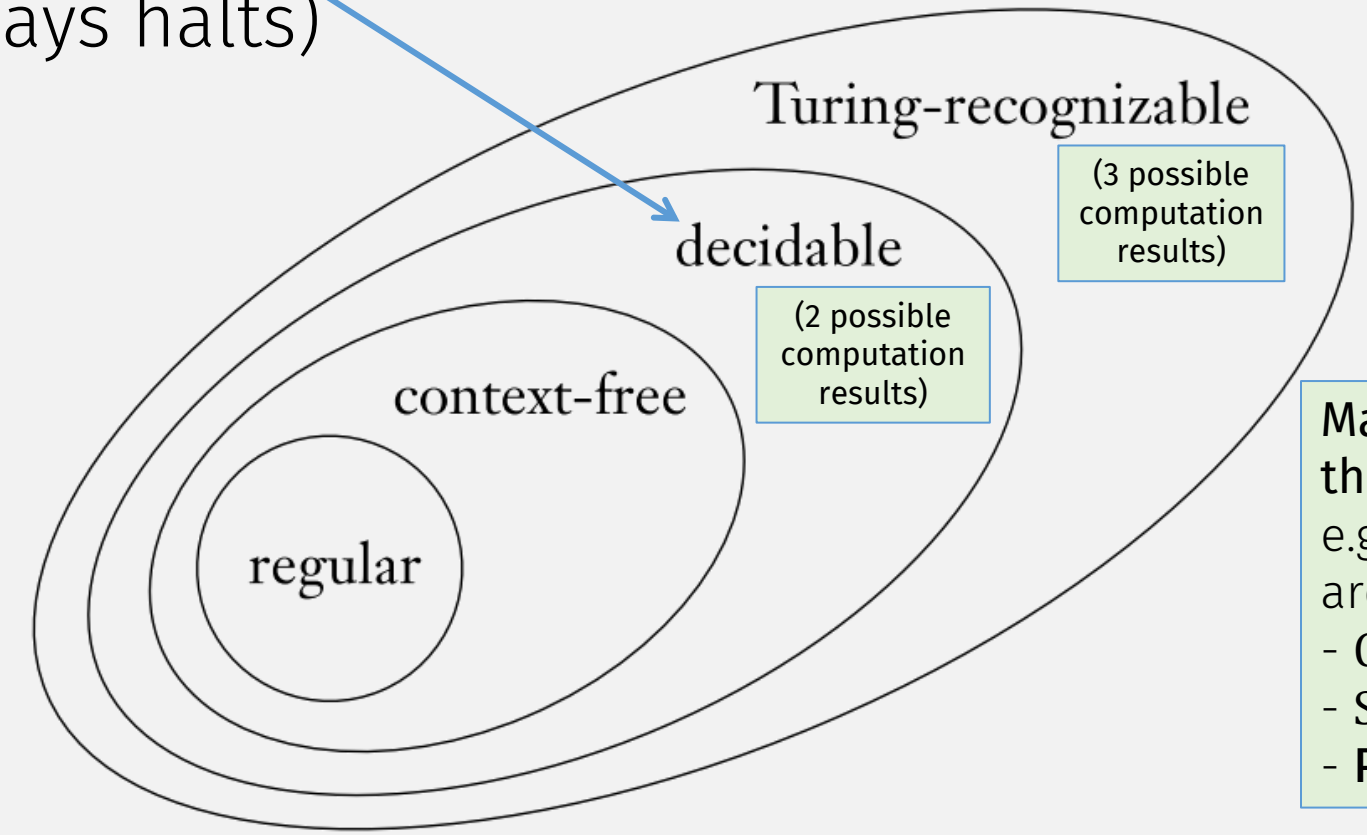
IF M IS-A **Decider** Machine AND M **decides** L
THEN L IS-A **Turing-decidable** Language

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)



Many <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**