Turing Machine II & Undecidability

Zeyu Mi

2021-12-17

Adapted From:

IALC 9.1, 9.2

Stanford CS103 Turing Machine

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

Part2. Undecidability.

Computation Capability

- For any certain type of automata (e.g. DFA,TM), the computation capability of it can be regarded as all the languages the type of automata can accept.

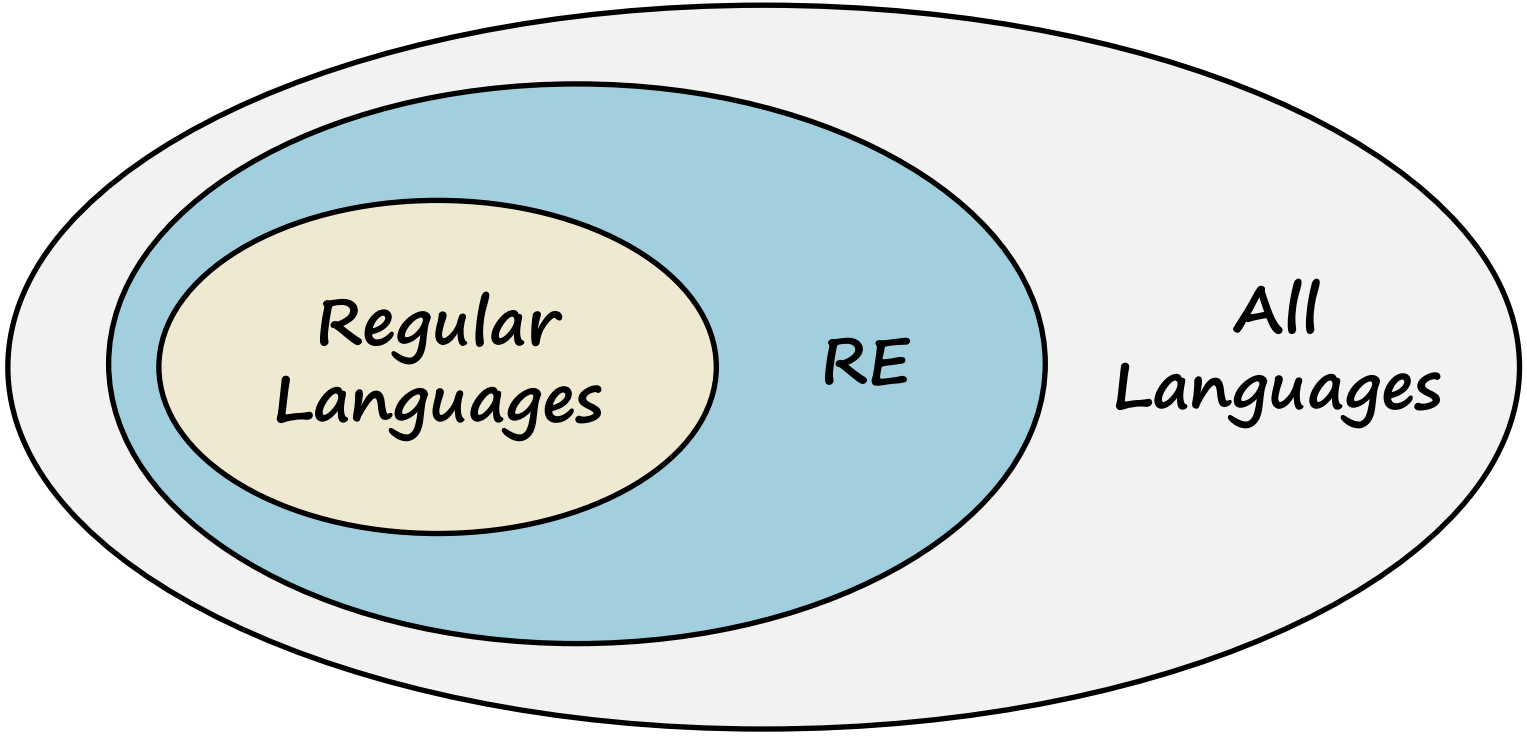
$$P(T) = \{L(A) | A \ is \ a \ T\},$$
where T can be DFA , Turing Machine, etc.

- For DFA, the capability is all the regular languages. For Turing Machine, it is all the *RE* languages.
  - $P(DFA) = \{L | L \ is \ a \ Regular \ Language\}$
  - $P(TM) = \{L | L \ is \ a \ RE \ Language\}$

Computation Capability

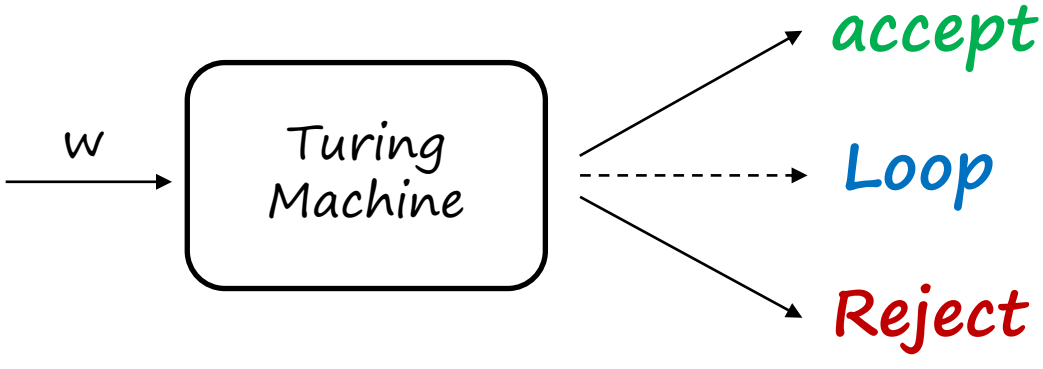
- For two types of automata T_1 and T_2
  - If $P(T_1) = P(T_2)$, we say T_1 and T_2 have same capability.
  - If $P(T_1) \subset P(T_2)$, we say T_2 have greater capability than T_1.
- DFA and TM:
  - For any language $\boldsymbol{L}$ accepted by a DFA ($\boldsymbol{L} \in \boldsymbol{P}(\boldsymbol{DFA})$), there's a TM $\boldsymbol{M}$ such that $L(\boldsymbol{M}) = \boldsymbol{L}$ ($\boldsymbol{L} \in \boldsymbol{P}(\boldsymbol{TM})$), so it means that $\boldsymbol{P}(\boldsymbol{DFA}) \subseteq \boldsymbol{P}(\boldsymbol{TM})$
  - And we know there's some languages can be accepted by TM but can not be accepted by DFA(like $\{1^n2^n\}$), so we have $P(DFA) \subset P(TM)$, which means **TM have greater computation capability than DFA**

Computation Capability



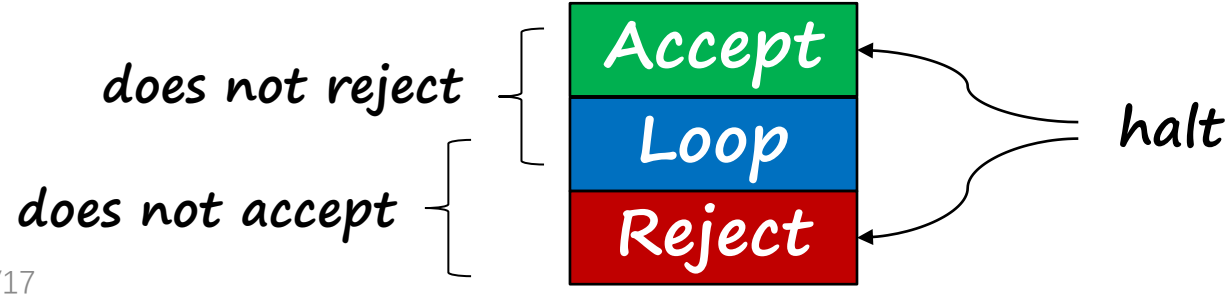
Output of TM

- For a certain input string, the output of a TM can be one in three kinds:
  - The TM **accepts** this string.
  - The TM **rejects** this string.
  - Rather than accepting or rejecting, the TM **loops** forever on the input string and never stops.



Very Important Terminology

- Let M be a Turing machine and let s be a string.
  - M **accepts** s if it enters an accepting state when running on s
  - M **rejects** s if it enters a rejecting state when running on s.
  - M **loops infinitely** on s when running on s if it enters neither an accepting nor a rejecting state.
  - M **does not accept** s if it either rejects s or loops on s.
  - M **does not reject** s if it either accepts s or loops on s.
  - M **halts** on s if it accepts w or rejects s.



More Details of RE

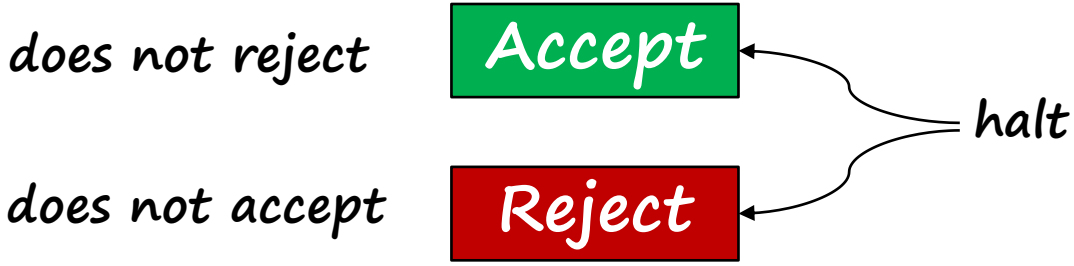
- We've known that for a certain language L, if there exists a TM $\boldsymbol{M}$ such that $L = L(\boldsymbol{M})$, then we call L is a **recursively enumerable language(递归可枚举语言)**, or ***RE***.
- So for any RE $\boldsymbol{L}$, a TM $\boldsymbol{M}$ that $L(M) = L$, and any string $\boldsymbol{w}$
  - If $w \in L, M$ accepts w in finite steps.
  - If $w \notin L, M$ does not accept w, it means M rejects w or loops forever
- We also call the TM $\boldsymbol{M}$ a **recognizer** for the language $\boldsymbol{L}$, and $\boldsymbol{L}$ is **Turing-recognizable(图灵可识别的).**

More Details of RE

- In other words, as an recognizer, $\boldsymbol{M}$ will tell you correct on every correct input, but $\boldsymbol{M}$ is not necessary to tell you wrong on every wrong input.
  - You can not determine the input is correct or wrong until $\boldsymbol{M}$ halts.
- But a problem is solvable(computable) if the computation process finishes in finite steps.

Decider

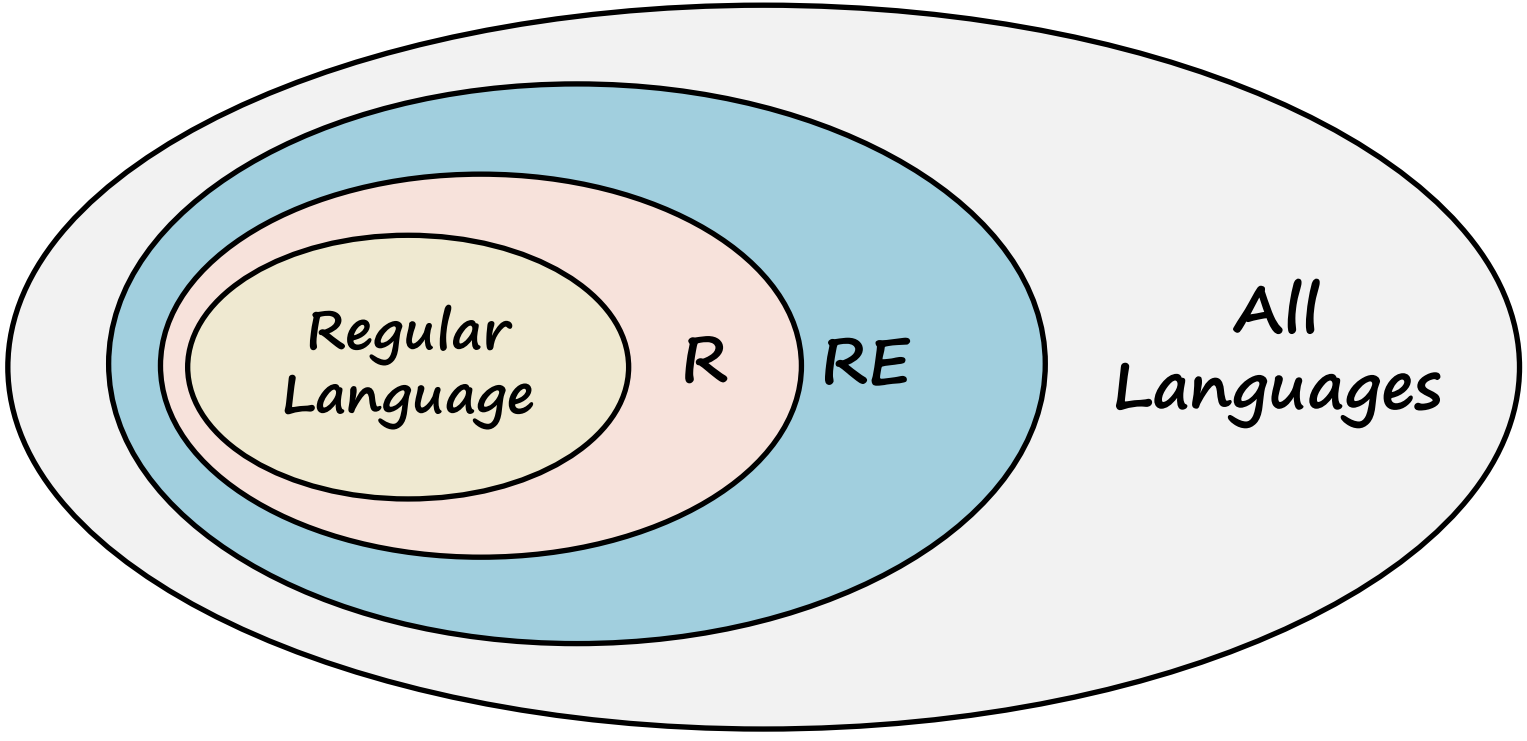
- Some Turing machines always halt; they never go into an infinite loop.
- If M is a TM and M halts on every possible input, then we say that M is a decider.
- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.



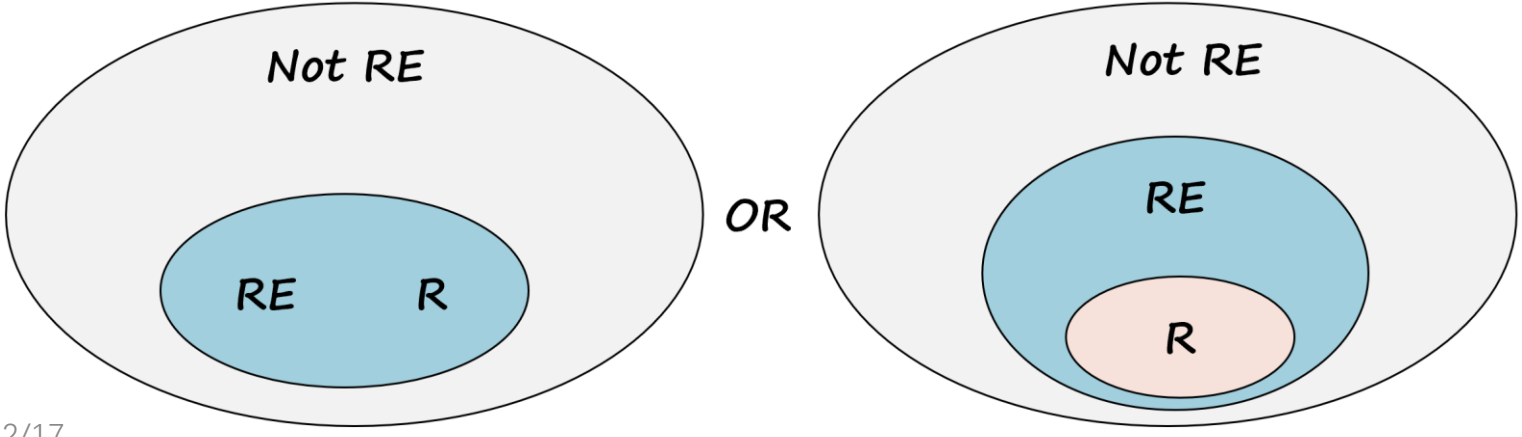
Recursive Language

- A language $\boldsymbol{L}$ is **recursive(递归的)** if there's a TM $\boldsymbol{M}$ such that:
  - If $w \in L, M$ accepts w in finite steps.
  - If $w \notin L, M$ rejects w in finite steps.
- We also call the TM $\boldsymbol{M}$ a **decider** for the language $\boldsymbol{L}$, and $\boldsymbol{L}$ is **Turing-decidable(图灵可判定的).**
- Decidable problems, in some sense, are that can definitely be "solved" by a computer.
- Recursive Language is a subset of RE language

Language Hierarchy



Question:
RE=R?



Universal Turing Machine

通用图灵机

An Observation

- When we've been discussing Turing machines, we've talked about designing specific TMs to solve specific problems.
  - TM for 0^n1^n
- Does this match your real-world experiences? Do you have one computing device for each task you need to perform?
- Or can we make a "**reprogrammable** Turing machine?"

A TM Simulator

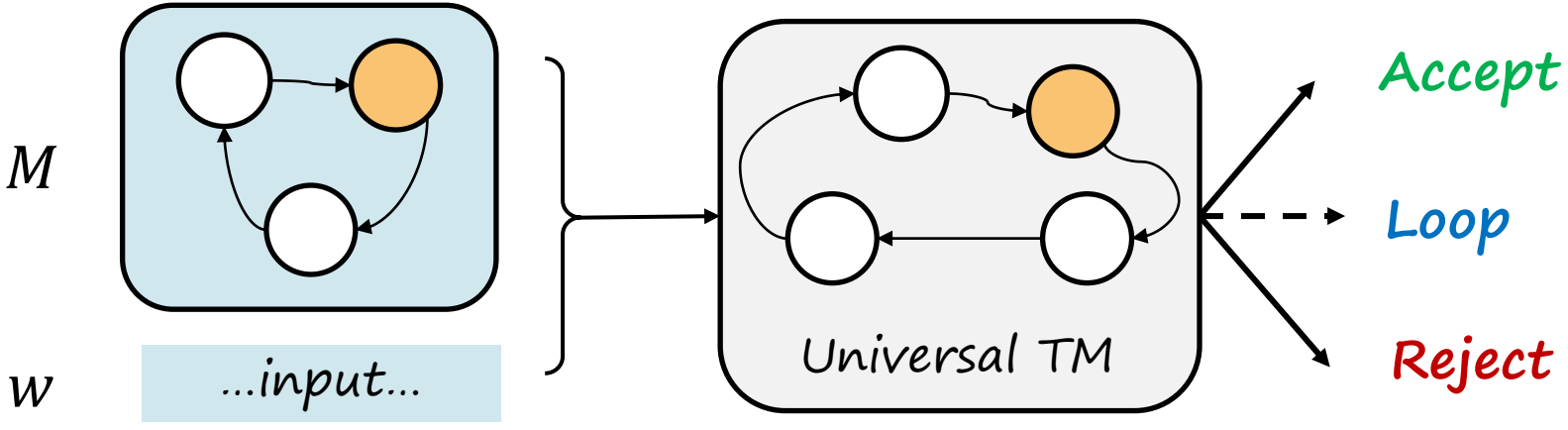
- We've known it is possible to program a TM simulator on an unbounded-memory computer.
- We could imagine it as a method

  boolean simulateTM(TM M, string w)

- with the following behavior:
  - If M accepts w, then *simulateTM(M, w)* returns true.
  - If M rejects w, then *simulateTM(M, w)* returns false.
  - If M loops on w, then *simulateTM(M, w)* loops infinitely.

A TM Simulator

- This means that there may be some TM that has the behavior of this *simulateTM* method.



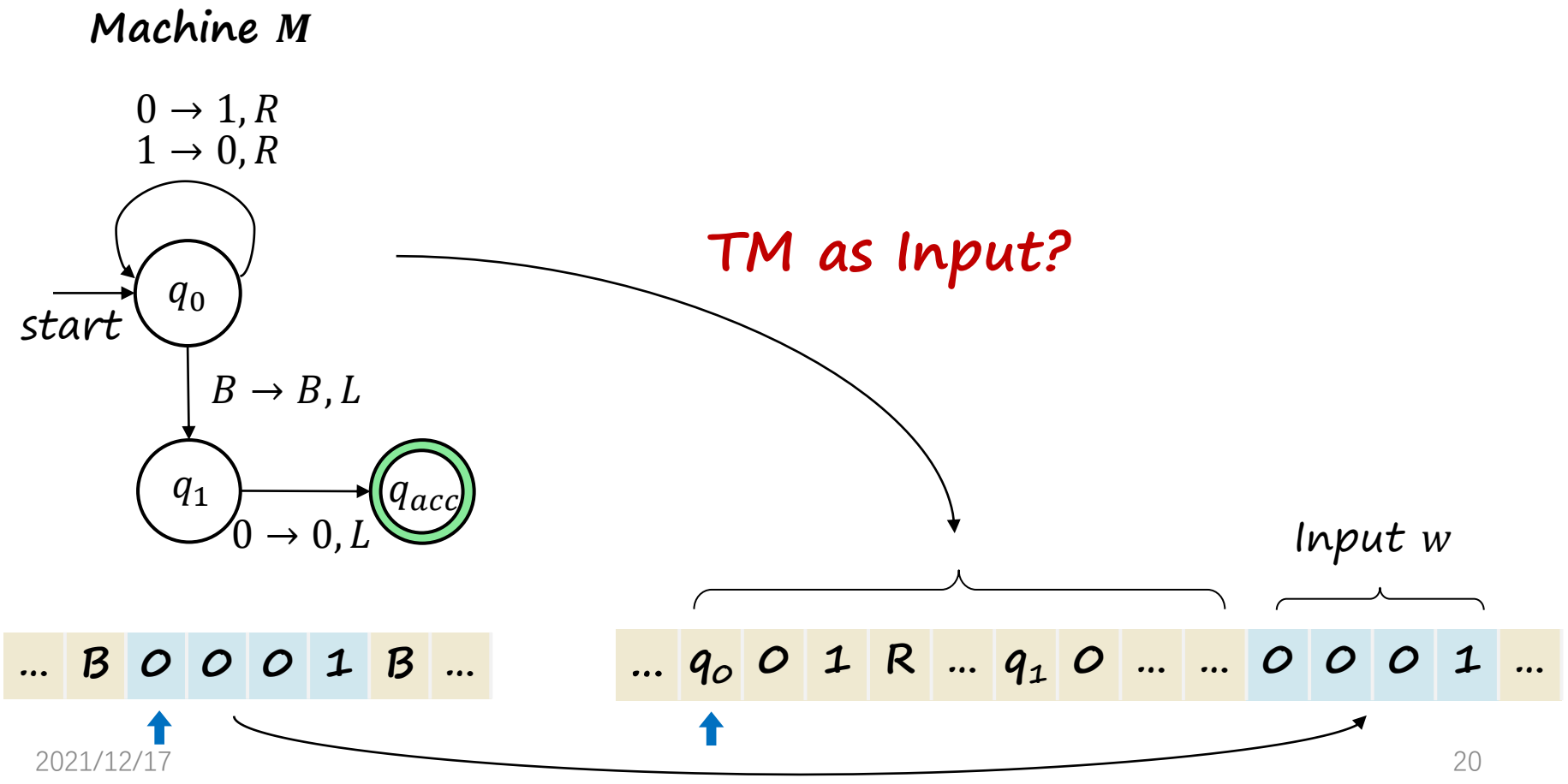
The Universal Turing Machine

- **Theorem (Turing, 1936):** There is a Turing machine $\boldsymbol{U_{TM}}$ called **the Universal Turing Machine** that, when run on an input of the form $\langle M, w \rangle$, where M is a Turing machine and w is a string, simulates M running on w and **does whatever** M does on w (accepts, rejects, or loops).
- U_{TM} behaves as follows**:**
  - If M accepts w, then U_{TM} accepts $\langle M, w \rangle$
  - If M rejects w, then U_{TM} rejects $\langle M, w \rangle$
  - If M loops on w, then U_{TM} loops on $\langle M, w \rangle$

Encoding Input with binary

- Input string may contain any possible character in the input alphabet of M
- But we know everything on your computer is a string over $\{0, 1\}$
- We can let the input alphabet to be $\{0,1\}$
- It not necessary to limit the alphabet as $\{0,1\}$, but only for simplicity.

The Universal Turing Machine



Encoding TM

- In order to take a Turing Machine as an input, we need to encoding the TM. Similarly, we shall encode TM with binary.
- We first assign integers to the states, tape symbols and directions
  - We assume the states are $q_1, q_2, \ldots, q_k$ for some k. $\boldsymbol{q_1}$ is the **input state** and $\boldsymbol{q_2}$ is the **only accept state**.(Is it right?)
  - The tape symbols are $X_1, X_2, \ldots, X_m$ for some m. $\boldsymbol{X_1, X_2, X_3}$ **are** $\boldsymbol{0, 1, B}$ respectively.
  - Refer to direction $\boldsymbol{L}$ **as** $\boldsymbol{D_1}$, $\boldsymbol{R}$ **as** $\boldsymbol{D_2}$

Encoding TM

- After assigning each state, symbol and direction an integer, we can encode the transition function $\boldsymbol{\delta}$.
- Suppose one transition rule is $\boldsymbol{\delta(q_i, X_j) = (q_k, X_l, D_m)}$, we shall code this rule by the string $\boldsymbol{0^i 1 0^j 1 0^k 1 0^l 1 0^m}$.
  - Notice $\boldsymbol{i, j, k, l, m}$ are at least one, so there're no occurrences of two or more consecutive 1's with in the code for a transition
- So let $\boldsymbol{C_i}$ donate the code for the ith transition rule, we can encode the whole TM as:
  - $C_1 11 C_2 11 C_3 11 \ldots 11 C_n$

Example of Code for TM

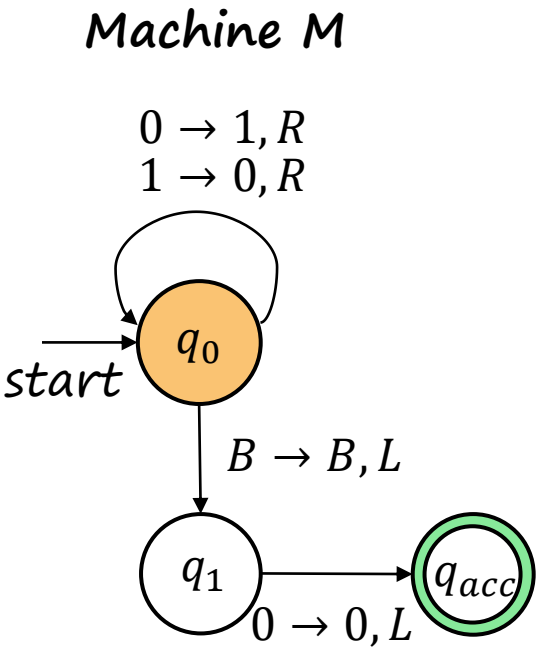
- Let a TM: $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1, B\}, \{\delta\}, q_1, B, q_2)$
  - $X_1 = 0, X_2 = 1, X_3 = B, D_1 = L, D_2 = R$
- Transition Function δ:
  - $\boldsymbol{\delta(q_1, 1) = (q_3, 0, R)}$
  - $\boldsymbol{\delta(q_3, 0) = (q_1, 1, R)}$
  - $\boldsymbol{\delta(q_3, 1) = (q_2, 0, R)}$
  - $\boldsymbol{\delta(q_3, B) = (q_3, 1, L)}$

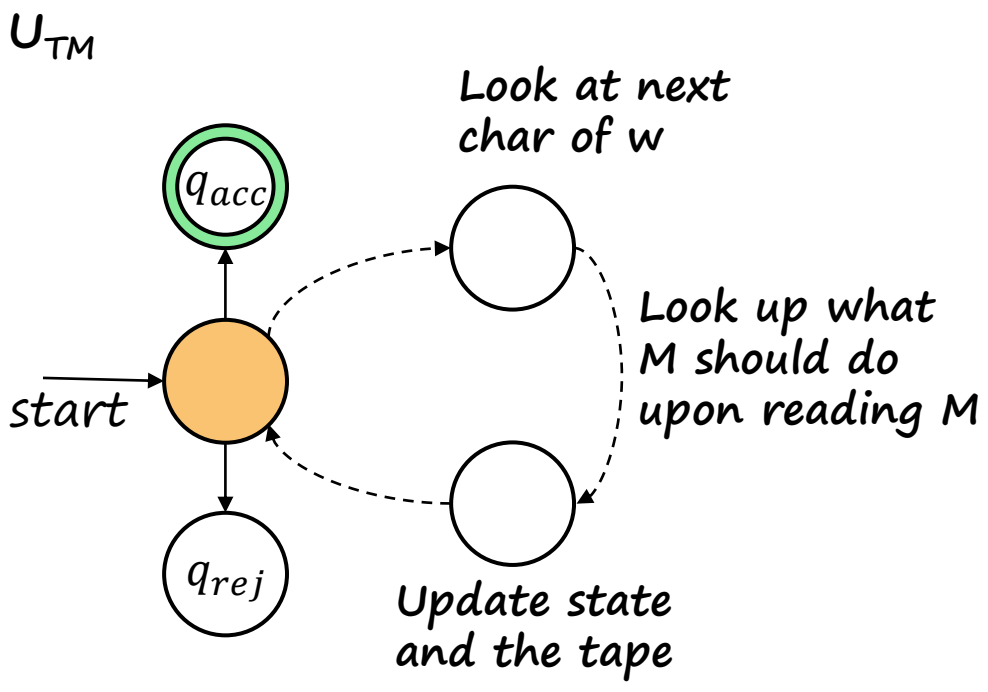
Example of Code for TM

- Let a TM: $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1, B\}, \{\delta\}, q_1, B, q_2)$
  - $X_1 = 0, X_2 = 1, X_3 = B, D_1 = L, D_2 = R$
- Transition Function δ:
  - $\boldsymbol{\delta(q_1, 1) = (q_3, 0, R) \Rightarrow \delta(q_1, X_2) = (q_3, X_1, D_2) \Rightarrow 0100100010100}$
  - $\boldsymbol{\delta(q_3, 0) = (q_1, 1, R) \Rightarrow \delta(q_3, X_1) = (q_1, X_2, D_2) \Rightarrow 0001010100100}$
  - $\boldsymbol{\delta(q_3, 1) = (q_2, 0, R) \Rightarrow \delta(q_3, X_2) = (q_2, X_1, D_2) \Rightarrow 0001001001010 0}$
  - $\boldsymbol{\delta(q_3, B) = (q_3, 1, L) \Rightarrow \delta(q_3, X_3) = (q_3, X_2, D_1) \Rightarrow 0001000100010010}$
- **Code for this TM:**

0100100010100110001010010011000100100101001100010001000100100

The Universal Turing Machine

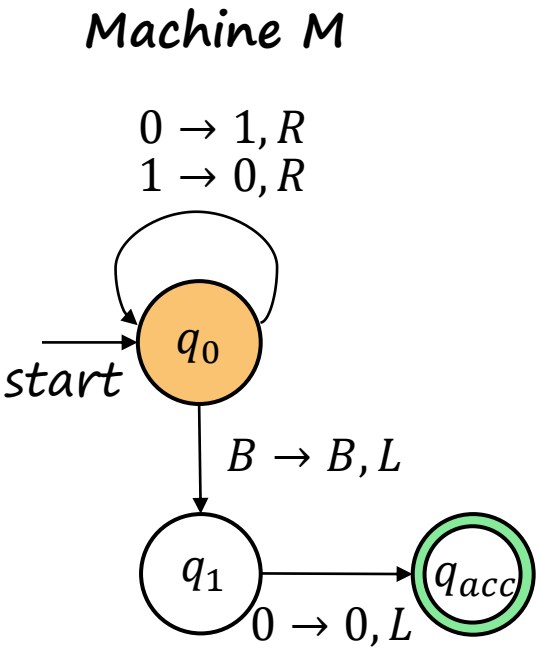


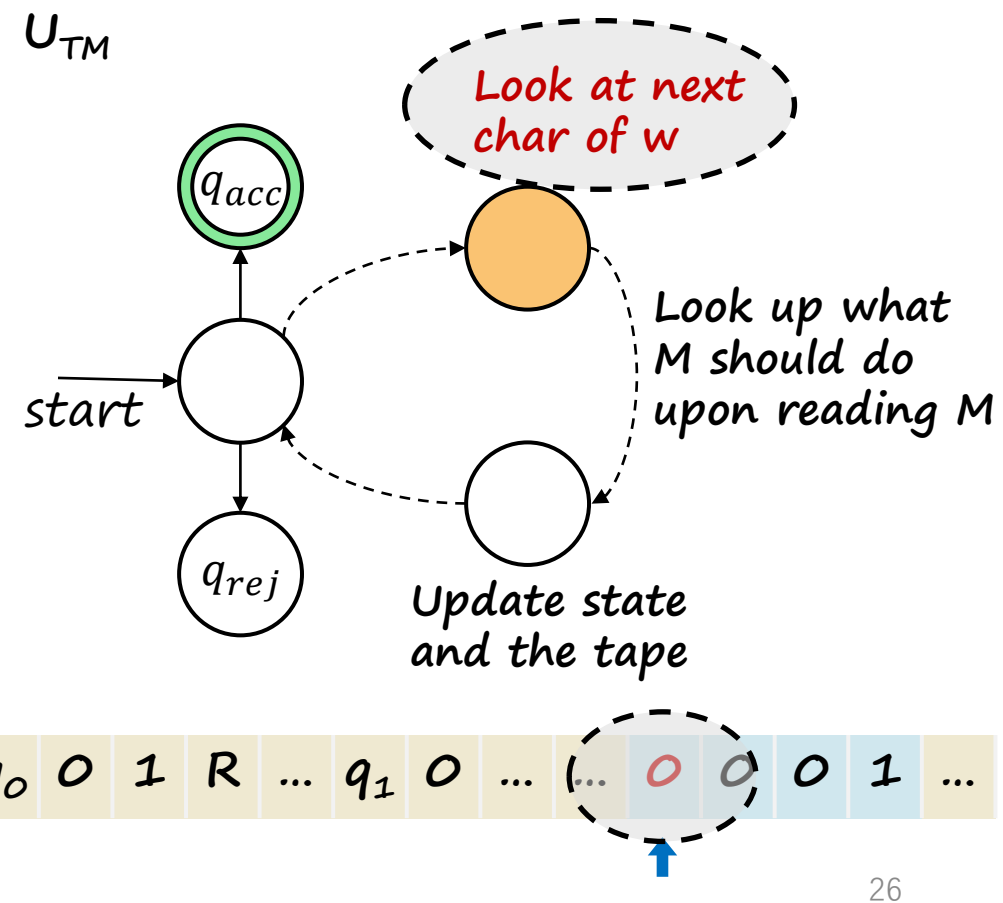


...	B	0	0	0	1	B	...

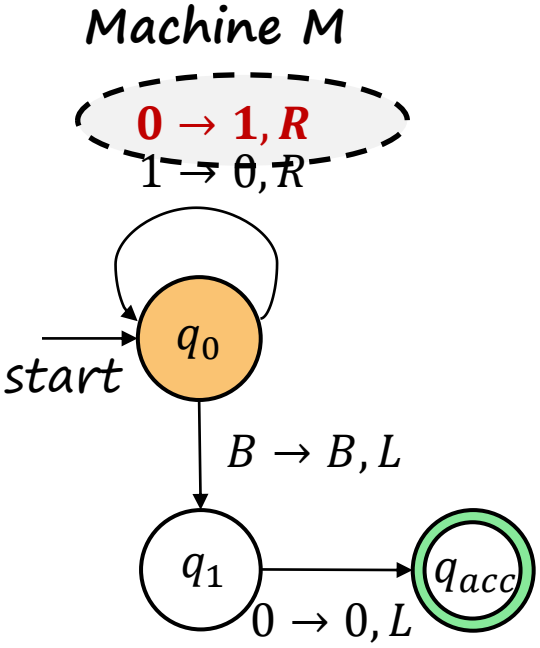
...	q_0	0	1	R	...	q_1	0	...	...	0	0	0	1	...

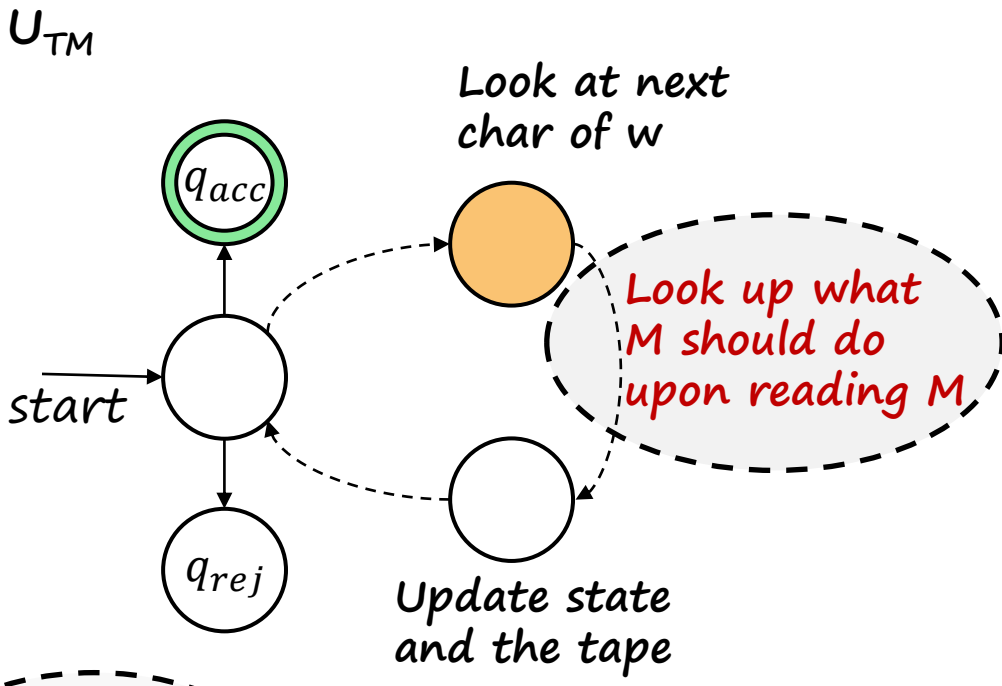
The Universal Turing Machine

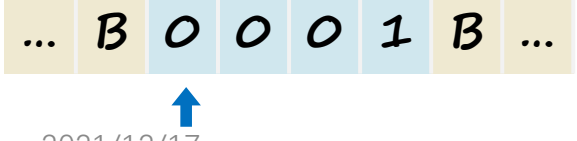


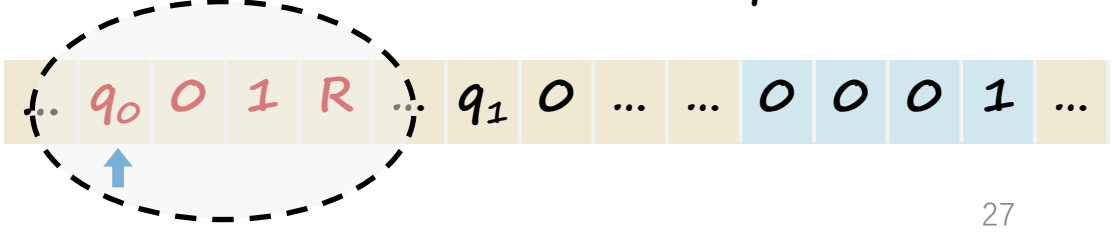


The Universal Turing Machine

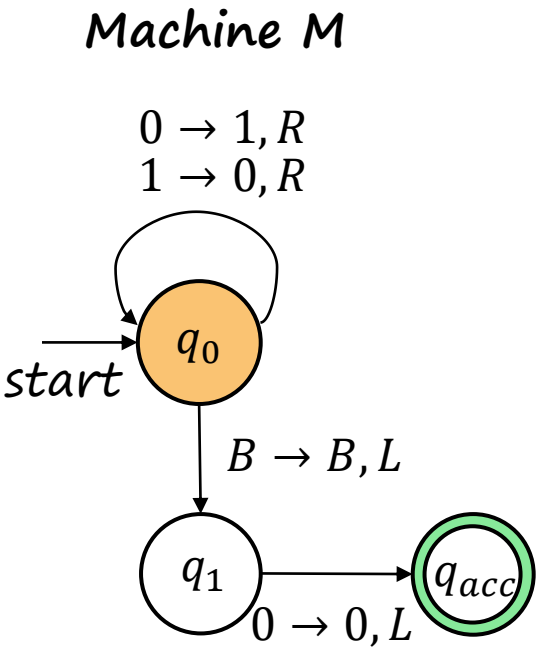


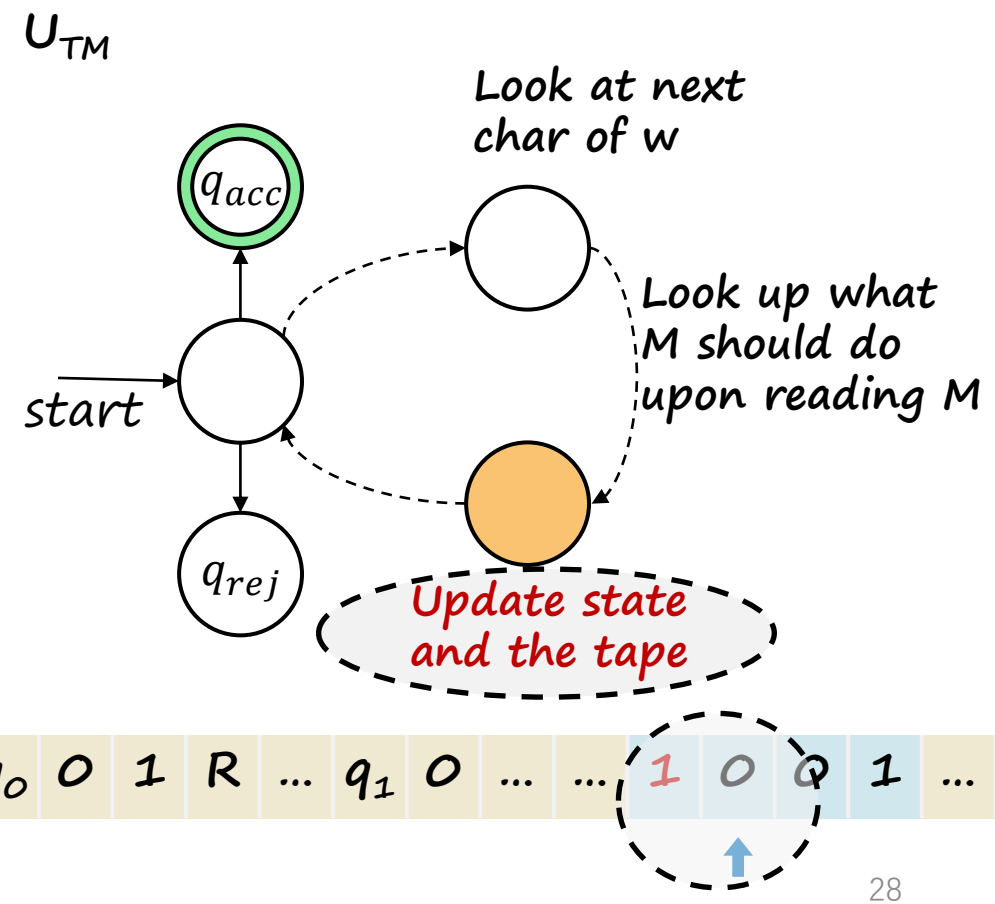




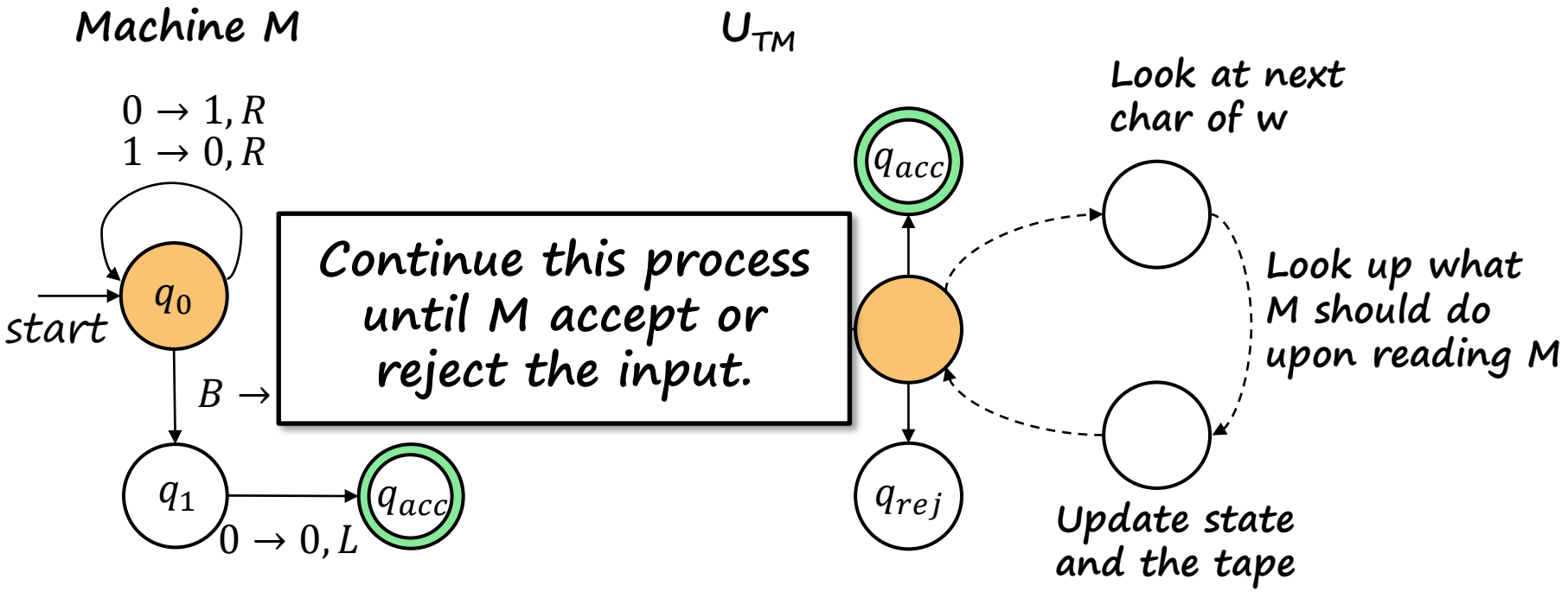


The Universal Turing Machine





The Universal Turing Machine



…	B	1	0	0	1	B	…

…	q_0	0	1	R	…	q_1	0	…	…	1	0	0	1	…

The Language of U_{TM}

- Recall that the language of a TM is the set of all strings that TM accepts.
- $\boldsymbol{U_{TM}}$ when run on a string $\langle \boldsymbol{M}, \boldsymbol{w} \rangle$, where $\boldsymbol{M}$ is a TM and $\boldsymbol{w}$ is a string, will
  - Accept $\langle M, w \rangle$ if M accepts w
  - Reject $\langle M, w \rangle$ if M rejects w
  - Loop on $\langle M, w \rangle$ if M loops on w.

The Language of U_{TM}

- The universal language, donated $\boldsymbol{L_u}$, is the language of the U_{TM}

$$L_u = L(U_{TM}) = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$$

$$= \{\langle M, w\rangle | M \text{ is a TM and } w \in L(M)\}$$

- Useful fact:

$$\boldsymbol{\langle M, w\rangle \in L_u \Leftrightarrow M \text{ accepts } w}$$

- Because $L_u = L(U_{TM})$, we know that $L_u \in RE$

The Language of U_{TM}

- If M accepts w, then we have:
  - U_{TM} accepts $\langle M, w \rangle$
  - U_{TM} accepts $\langle U_{TM}, \langle M, w \rangle \rangle$
  - U_{TM} accepts $\left\langle U_{TM}, \langle U_{TM}, \langle M, w \rangle \rangle \right\rangle$
  - ….

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

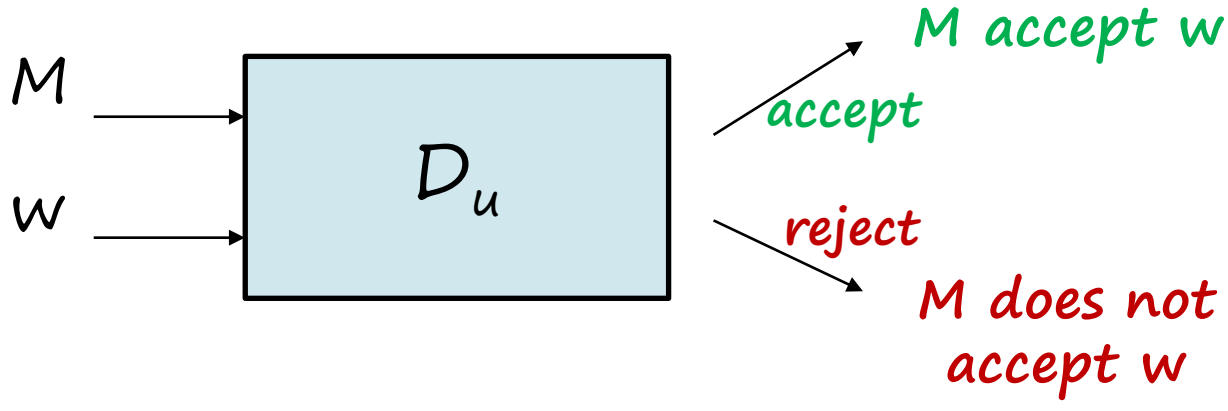
2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

Part2. Undecidability.

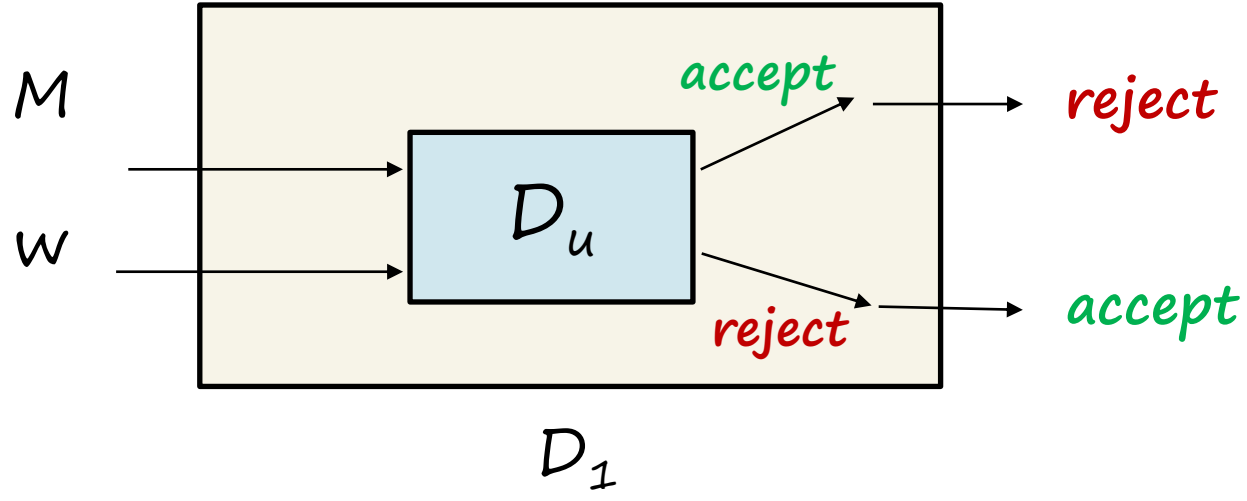
A Decider for L_u

- We know that $L_u \in \mathbf{RE}$, but whether $L_u \in \mathbf{R}$?
- That is, whether there is a decider D_u for L_u?



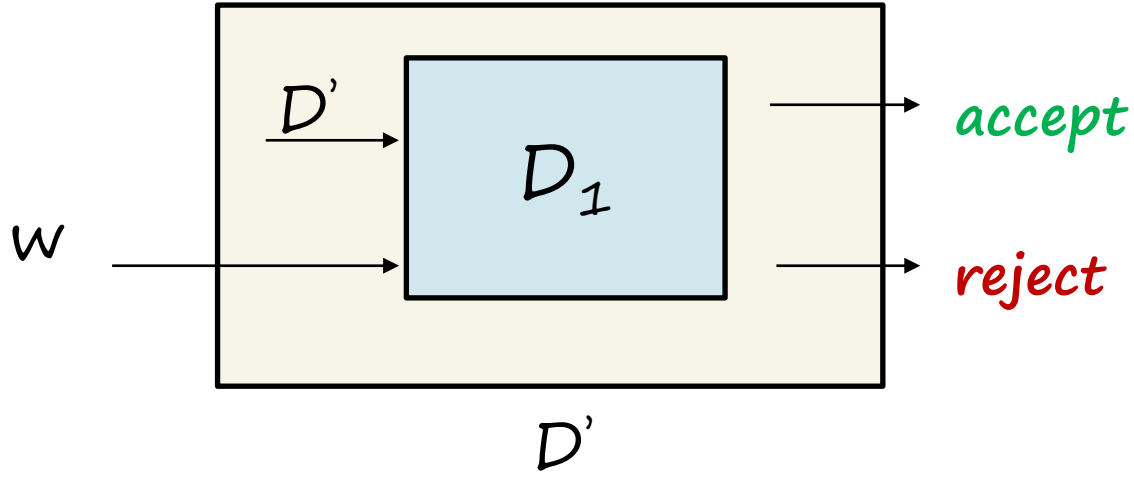
A Decider for L_u

- If D_u exists, we can build a new TM D_1 based on D_u
- D_1 takes a TM M and string w as inputs, and feeds them to D_u, then outputs the **reverse result** of D_u.



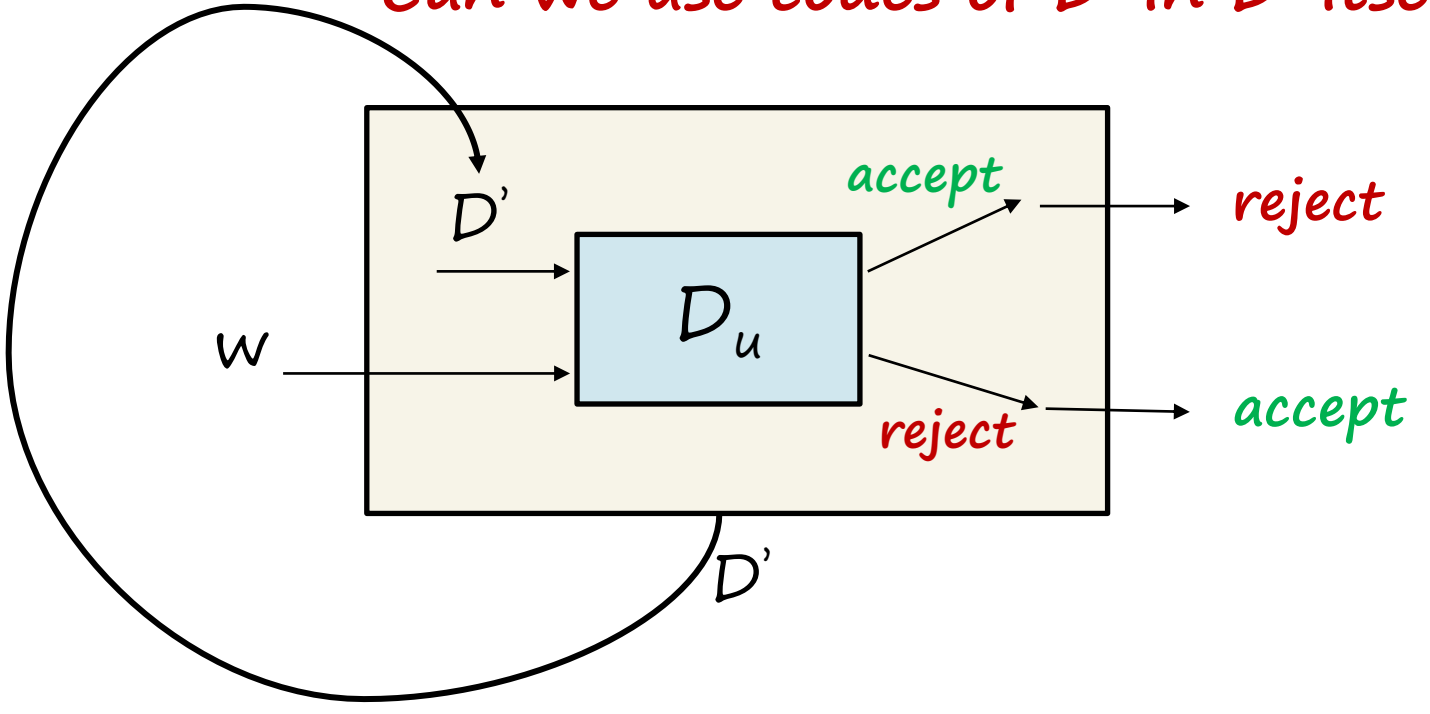
A Decider for L_u

- Then we can build another TM D' based on D_1
- D' only takes a string w as input, and feeds **its own code** and w to D_1, then outputs the result of D_1.

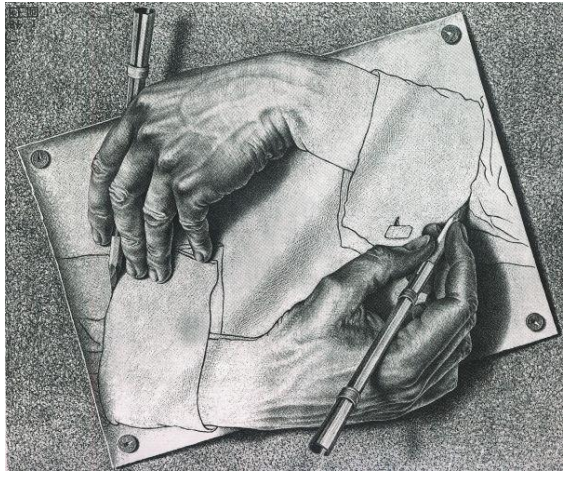


A Decider for L_u

Can we use codes of D' in D' itself?



Self-Reference



Sets that do not contain itself.

"This sentence is wrong."

Self-Referential Software

- A **Quine** is a program that, when running, prints its own source code.
- Quines aren't allowed to just read the file containing their source code and print it out; that's cheating (and technically incorrect if someone changes that file!)
- How would you write such a program?

Example: Quine

```
#include <stdio.h>
char*f="#include <stdio.h>
char*f=%c%s%c;
main(){printf(f,34,f,34,10);}%c";
main(){printf(f,34,f,34,10);}
```

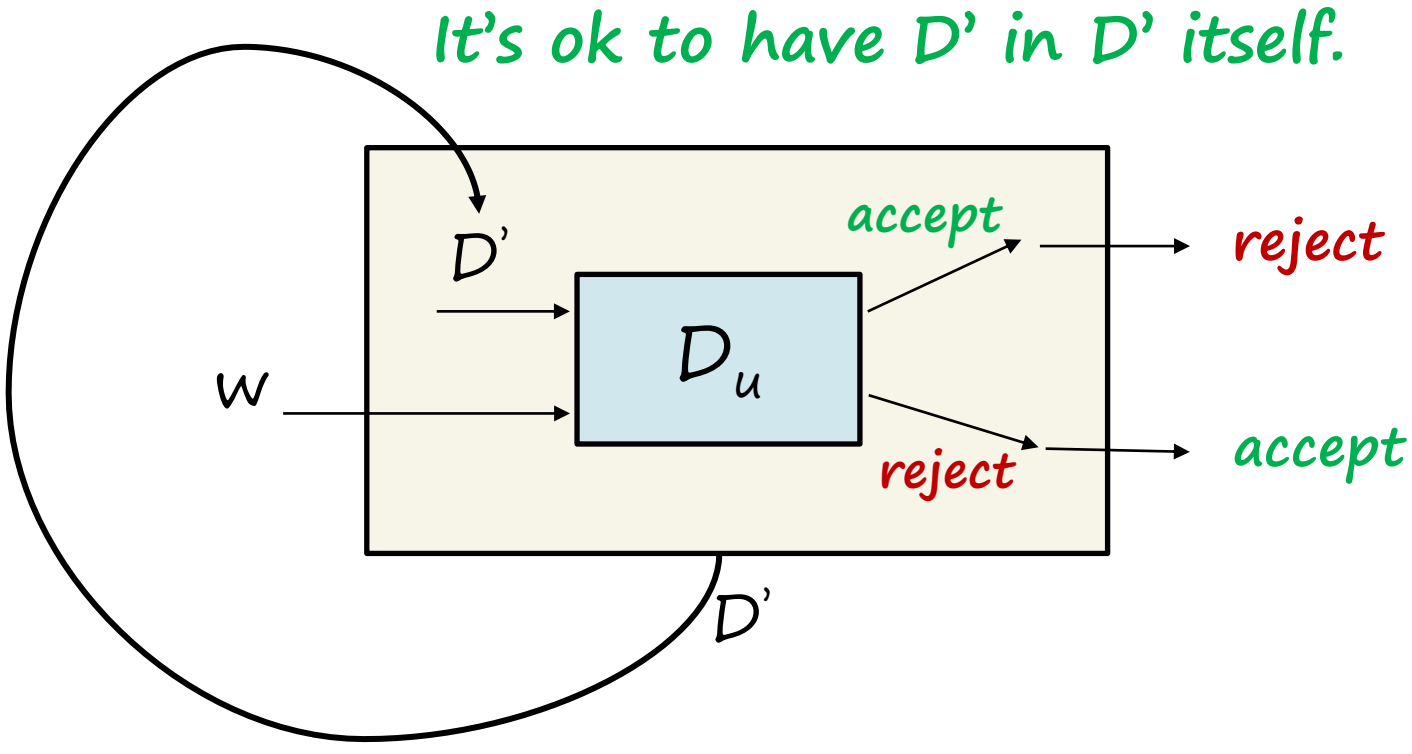
http://www.nyx.net/~gthompso/quine.htm

Self-Referential Software

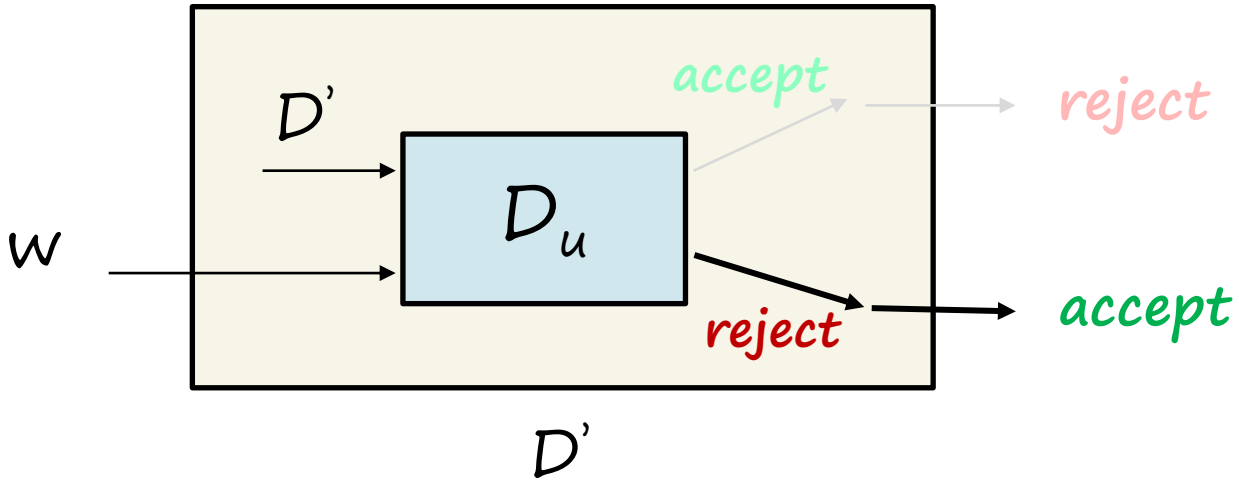
- Going forward, assume that any program can be augmented to include a method called *mySource()* that returns a string representation of its source code.
- Then we can take the whole program codes as an input though we don’t know what the whole program codes are, and even what we write now will change the final code itself.

```
char * mySource(){…}
int main(){
  char * s = mySource();
  ......
}
```

A Decider for L_u

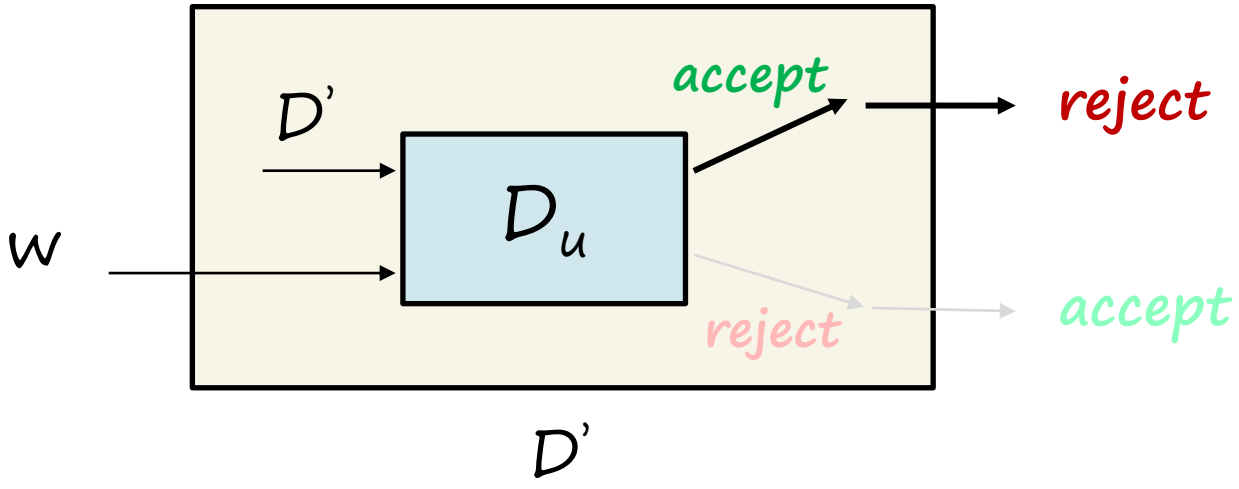


A Decider for L_u



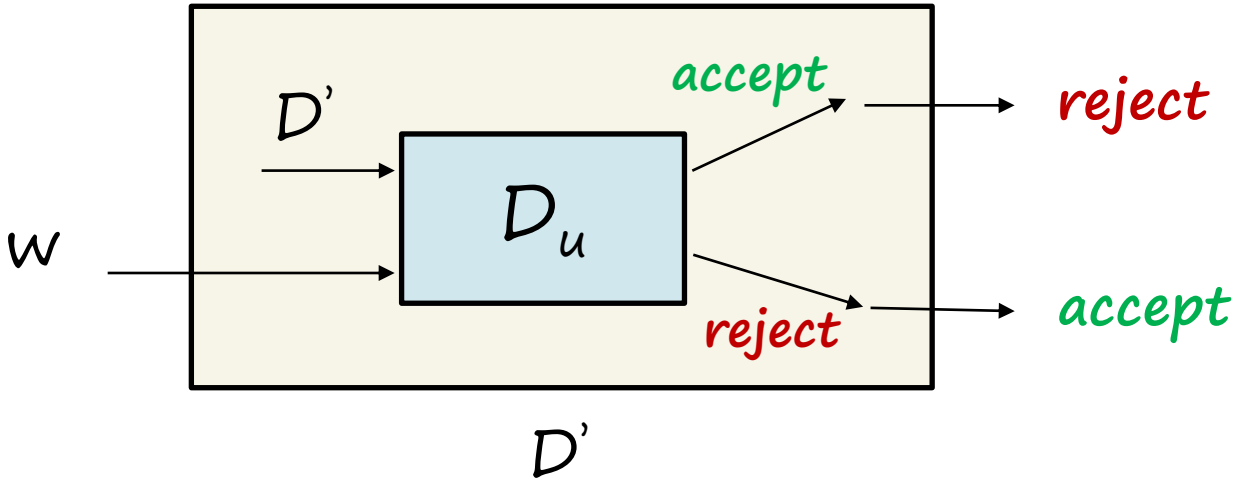
- If D' accepts w.
  - Then D_u rejects $\langle D', w \rangle$, which means that D' rejects w

A Decider for L_u



- If D' accepts w.
  - Then D_u rejects $\langle D', w \rangle$, which means that D' rejects w
- If D' rejects w.
  - Then D_u accepts $\langle D', w \rangle$, which means that D' accepts w

A Decider for L_u



$\boldsymbol{D'}$ accepts $\boldsymbol{w} \Leftrightarrow \boldsymbol{D'}$ rejects $\boldsymbol{w}$

D_u can not exist!

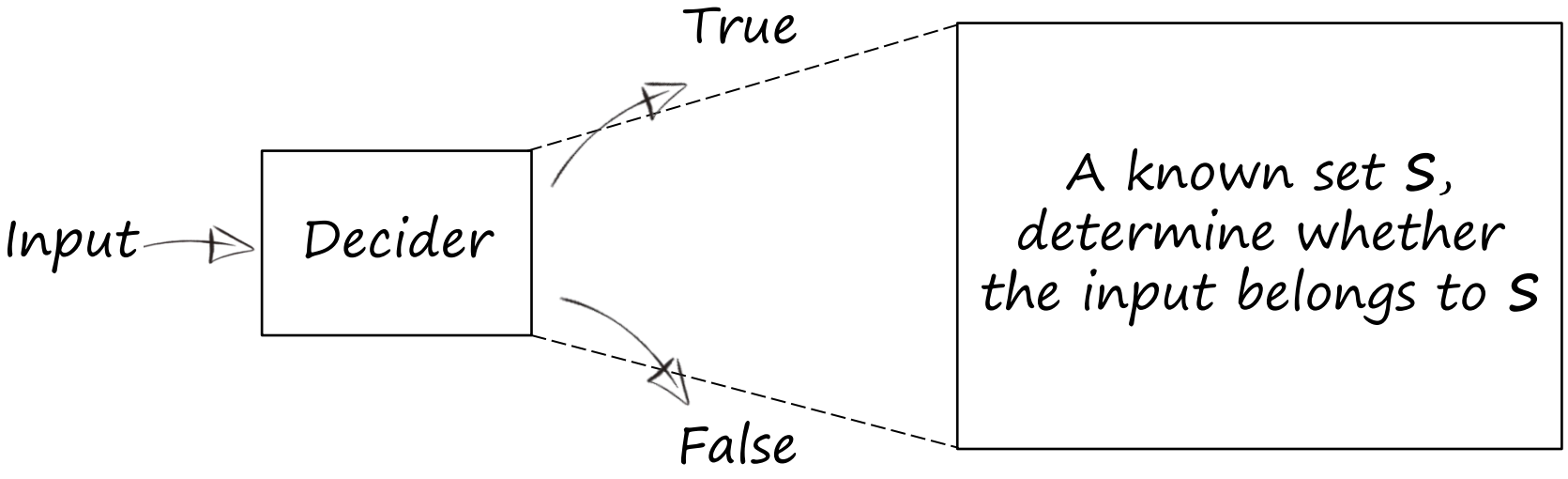
$L_u \notin \mathbf{R}$

- As there are no deciders for L_u, L_u is not in the class of **R.**
- **Undecidable(不可判定):** there is **no algorithm** that can determine whether a program will definitely accept or reject an input.
- Intuition: The only general way to find out what a program will do is to run it.

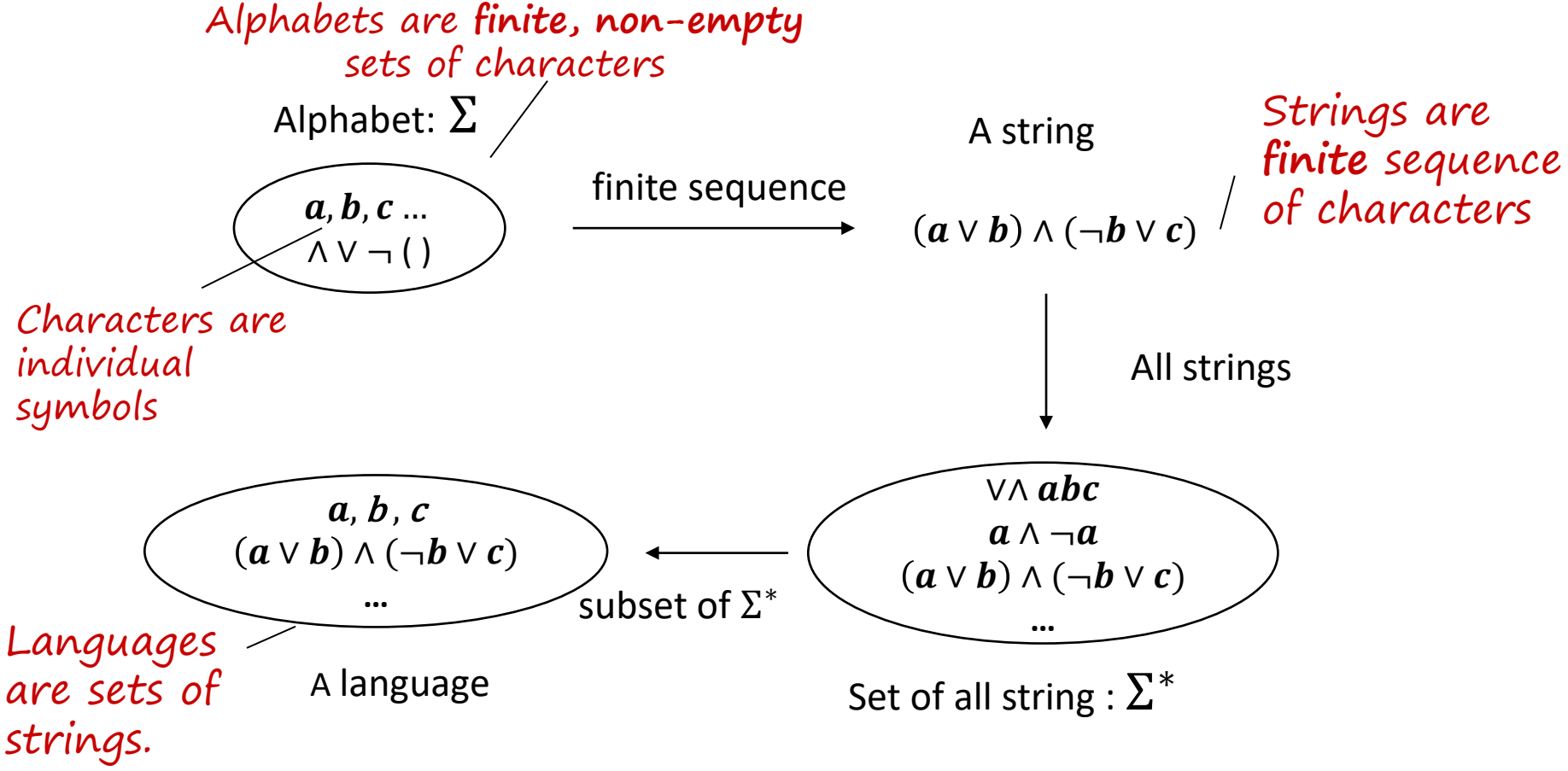
Computable（可计算的）

- **A computational problem is computable if there is a procedure (or algorithm) for it which:**
  - Rigorously follows some finite instructions, requires no ingenuity.
  - Always finishes (terminates) after a finite number of steps.

Solve Decision Problem



Formal Language Theory

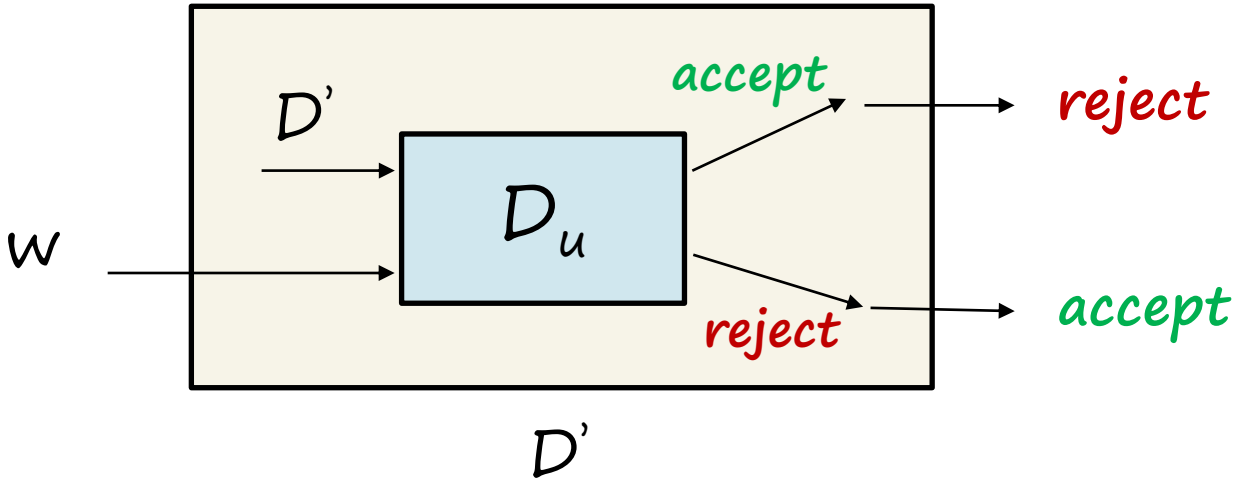


What problems can a computer solve?

What languages can a computer recognize?

What languages can TM recognize?

An Undecidable Problem

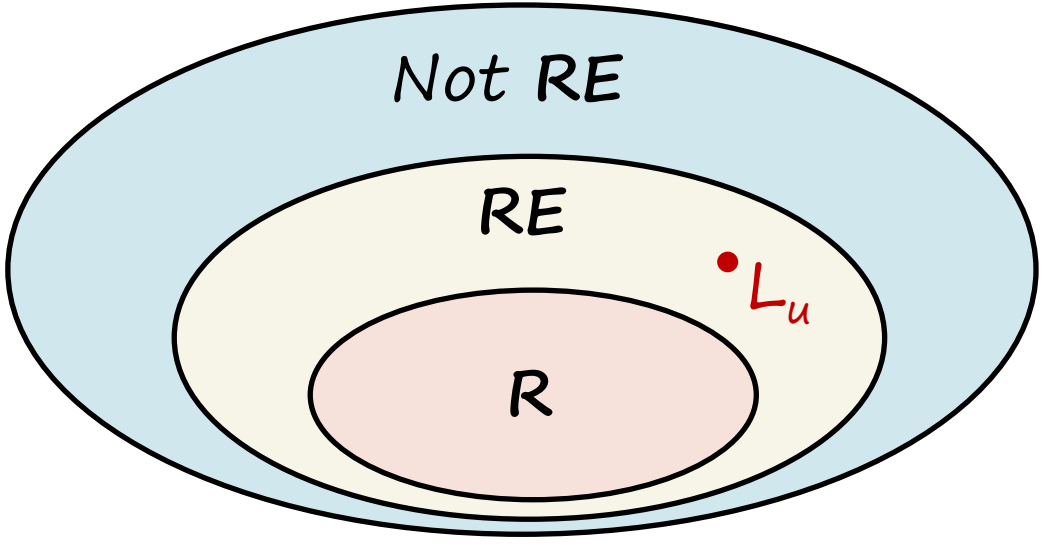


$\boldsymbol{D'}$ accepts $\boldsymbol{w} \Leftrightarrow \boldsymbol{D'}$ rejects $\boldsymbol{w}$

D_u can not exist!

$\mathbf{R} \neq \mathbf{RE}$

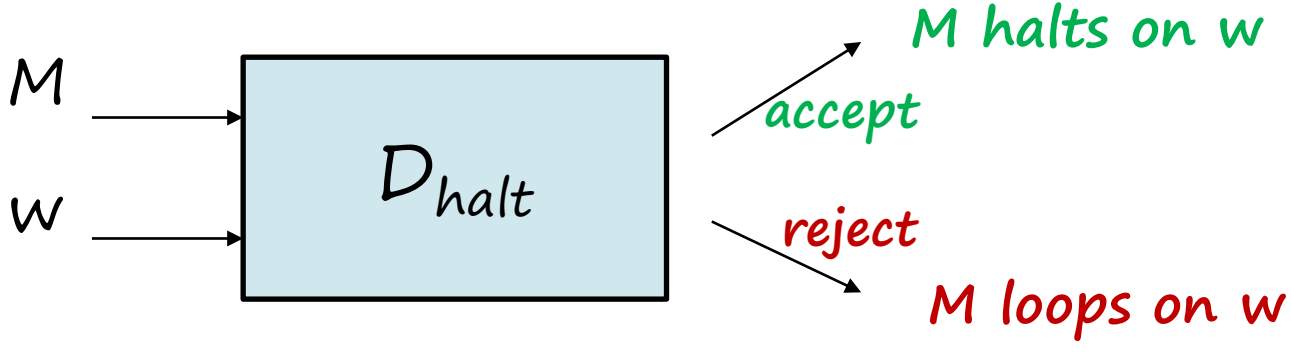
- We found a language $L_u \in \mathbf{RE}$, but $L_u \notin \mathbf{R}$, which means $\mathbf{R} \neq \mathbf{RE}$
- Because R ≠ RE, there is a difference between decidability and recognizability:



Halting Problem:
Determine whether a program will halt.

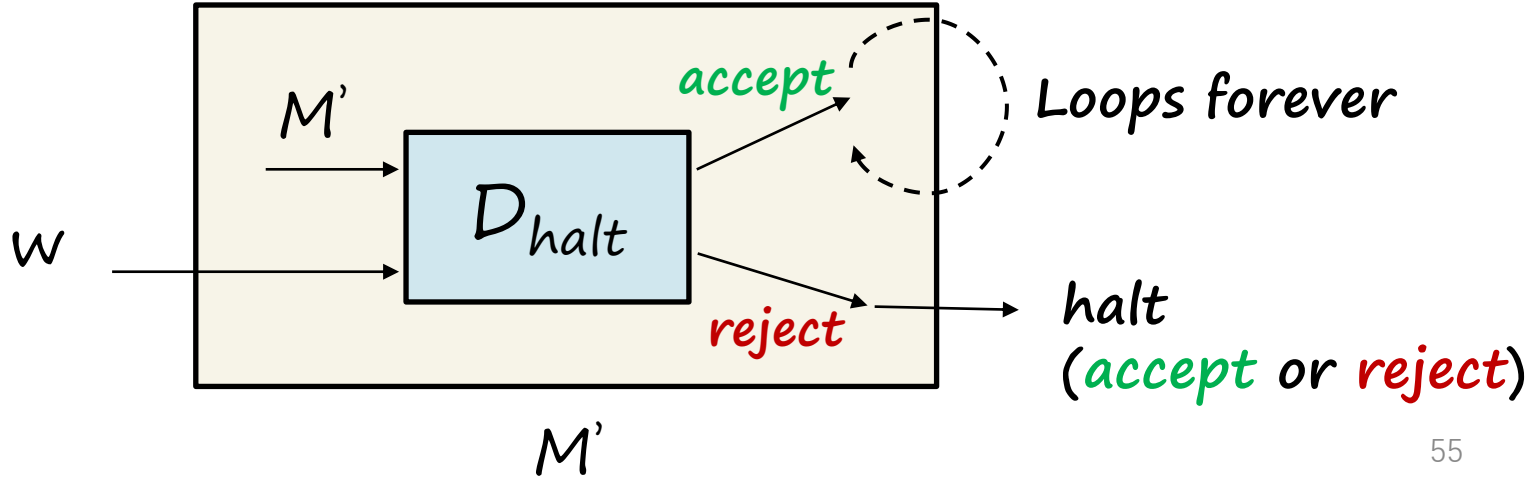
Turing Halting Problem

- Does there exist a TM to **decide** whether a program will finally halt(reject or accept) on a certain input?
- Or whether $L_{halt} = \{\langle M, w\rangle | M\ halts\ on\ w\}$ is in $\mathbf{R}$?

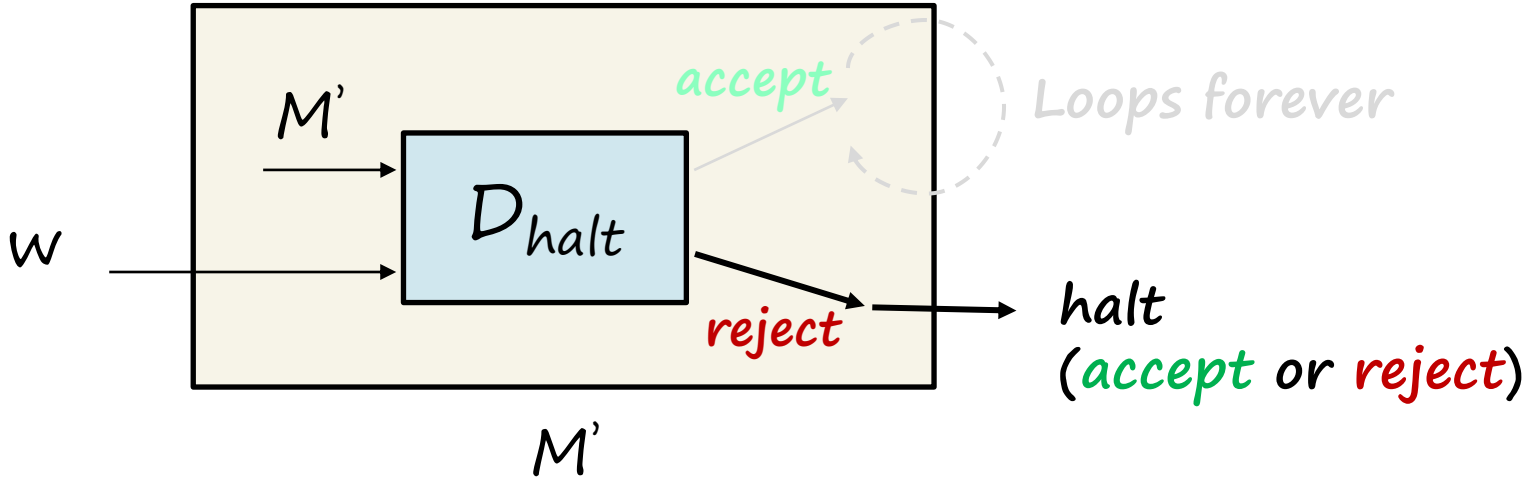


Turing Halting Problem

- Similarly we can build a new TM M' based on D_{halt}.
- M' takes a string w as input, and feed its own code and w to D_{halt}. If D_{halt} rejects the input, then M' halts(accept or reject) on w, otherwise M' loops forever and never stop.

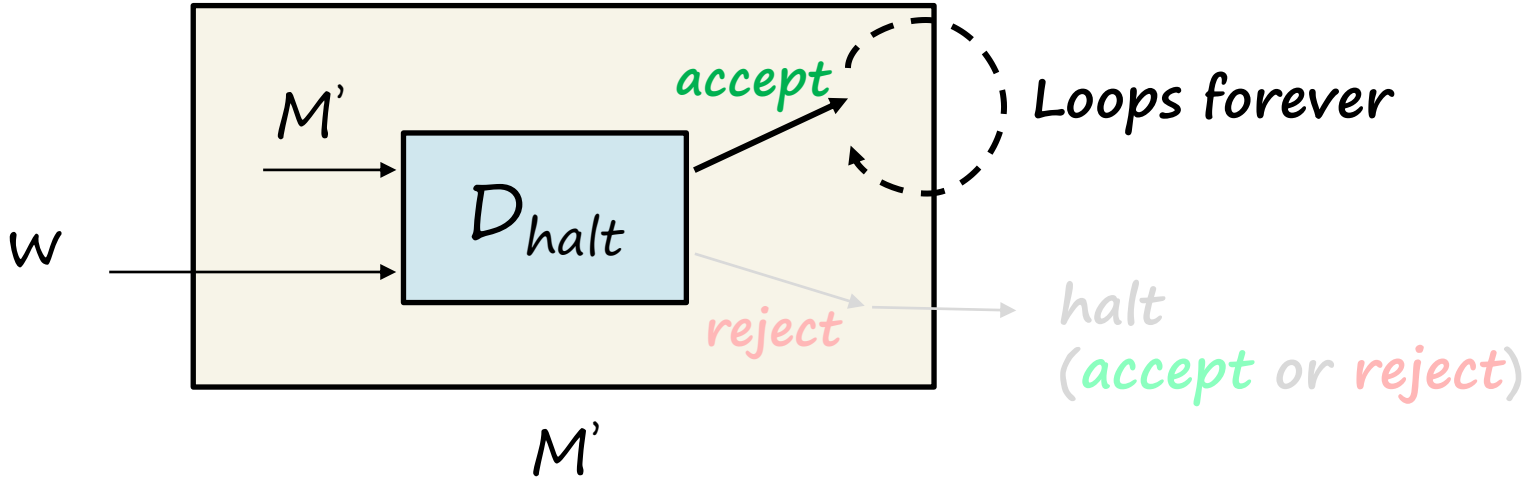


Turing Halting Problem



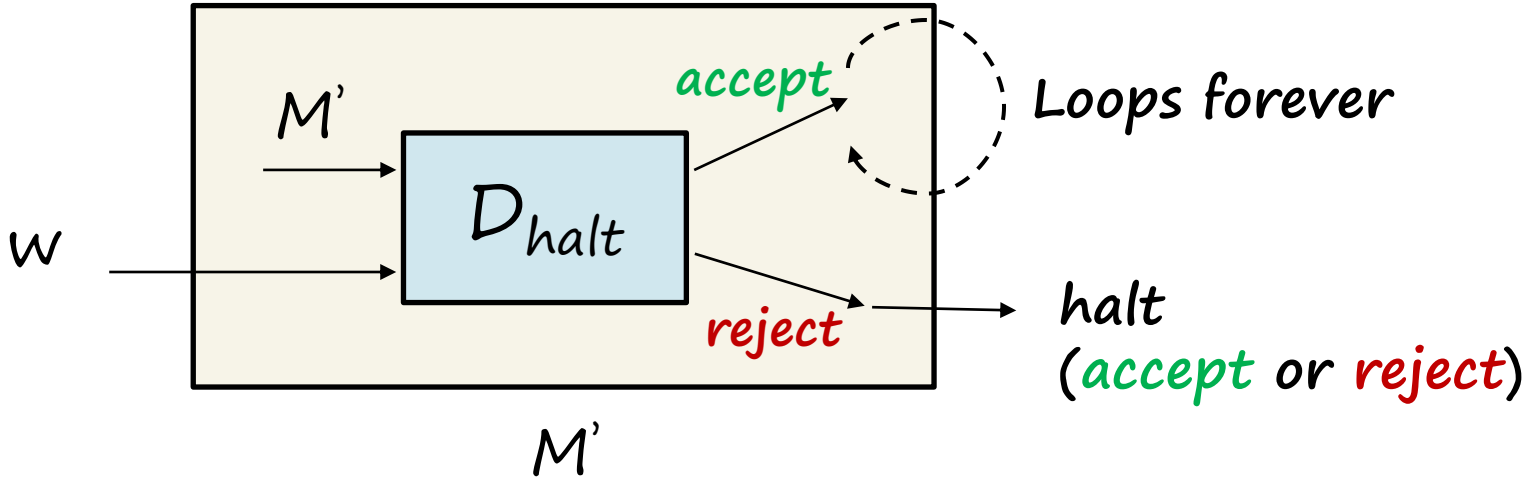
- If M' halts on w
  - Then D_{halt} must reject $\langle M', w \rangle$, which means M' loops on w

Turing Halting Problem



- If M' halts on w
  - Then D_{halt} must reject $\langle M', w \rangle$, which means M' loops on w
- If M' loops on w
  - Then D_{halt} must accept $\langle M', w \rangle$, which means M' halts on w

Turing Halting Problem



M' halts on $w \Leftrightarrow M'$ loops on w

D_{halt} can not exist!

Language that is Non-RE

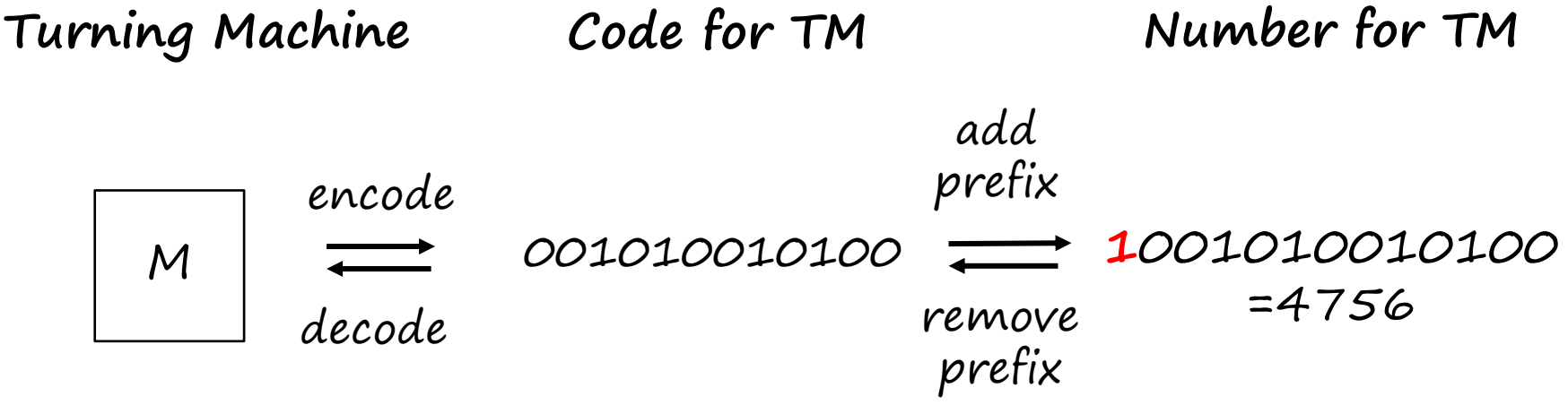
Beyond $\mathbf{RE}$

- The $\mathbf{RE}$ languages represent languages whose membership can be proven
  - If $w \in L$, we can prove that by a TM
  - So what does a non-RE language look like?
- This language cannot be recognized by any Turing machines

Numbering TMs

- Recall that a TM $\boldsymbol{M}$ can be encoded as a binary. In return, a binary can be interpreted as a TM.
  - Some binary may not be interpreted as a TM (illegal format), we interpret that binary as a TM without any transition function and accepting no string
  - TM code begins with 0's, and we can add a prefix 1 to the code of TM as the number of that TM
    - So the numerical values of different TMs are different.

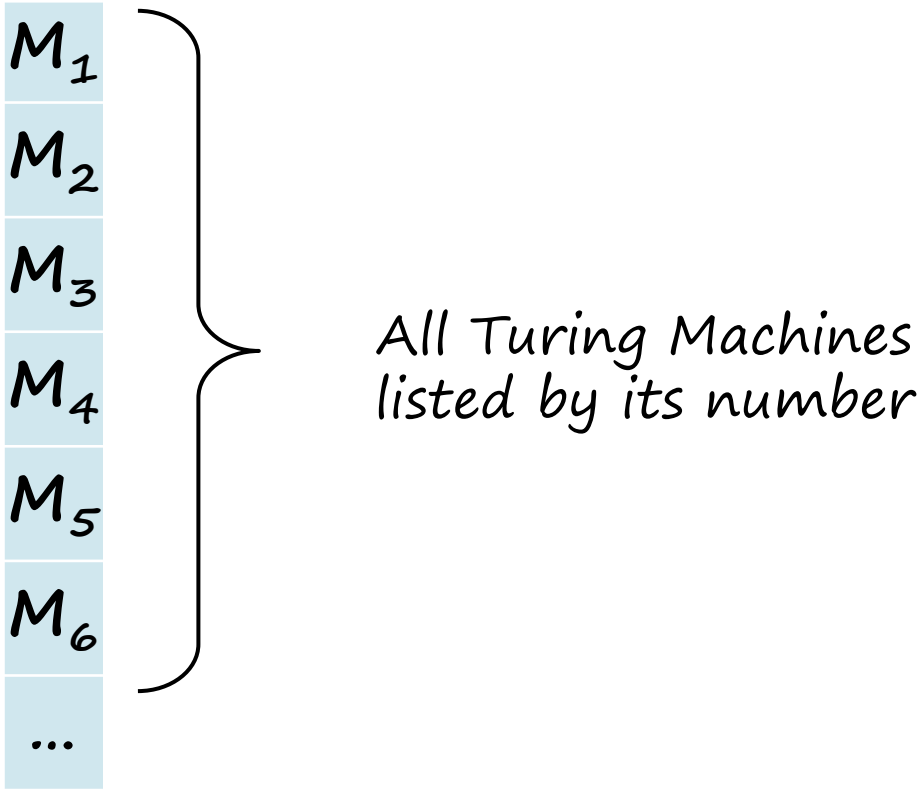
Numbering TMs



M=4756

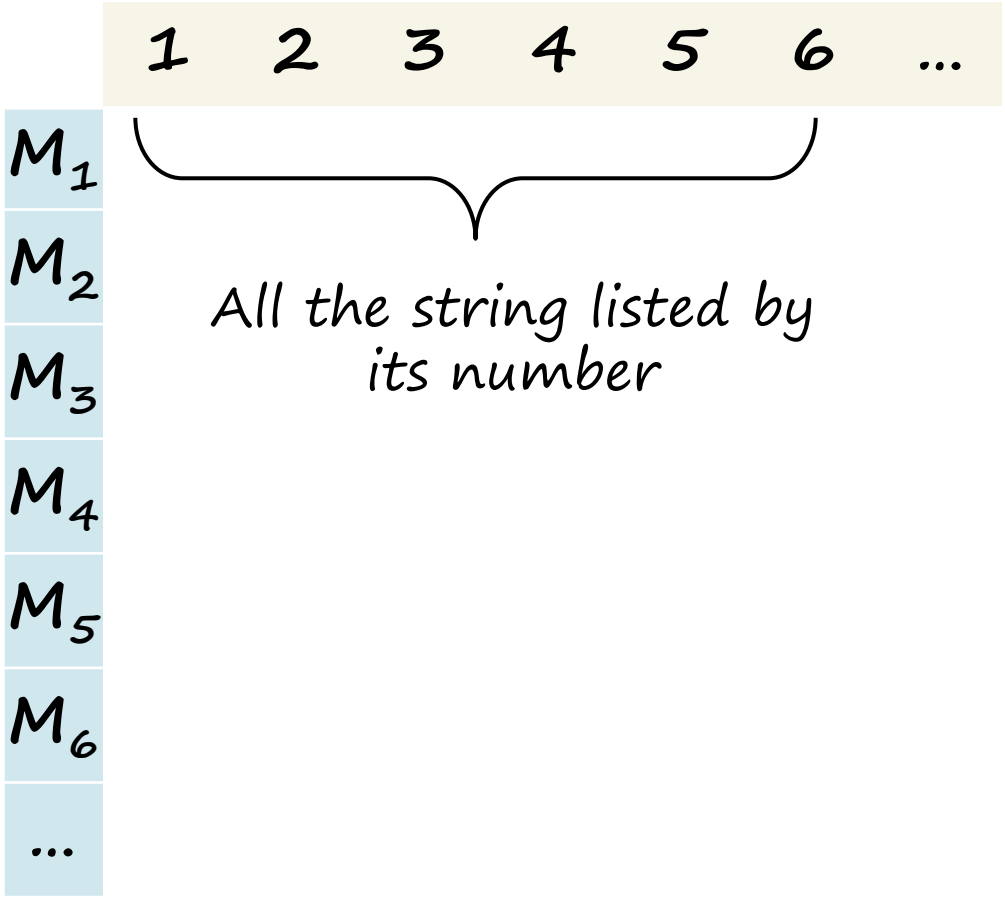
Numbering TMs

- We assign each TM a positive integer, and each positive integer has one TM corresponding to it.
  - We say that the $\boldsymbol{i}$th TM $\boldsymbol{M_i}$ is the TM whose number is $\boldsymbol{i}$.
- So we've build a bijection between the set of all the TMs and the set of positive integer. Then we can list all the TMs one by one.


M_1
M_2
M_3
M_4
M_5
M_6
...
All Turing Machines
listed by its number

Numbering All the String

- Each binary string represents a number
- We add a prefix 1 to each binary string to prevent the following condition
  - “01” and “001” are two different binary strings, but they represent the same number (1)
- w_i denotes the string with number i.
  - $w_1 = \epsilon, w_2 = 0, w_3 = 1, w_4 = 00$


1 2 3 4 5 6 ...
M1
M2
M3
M4
M5
M6
...
All the string listed by its number

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1 in row i column j means that M_i accepts w_j.

0 means "not accept".

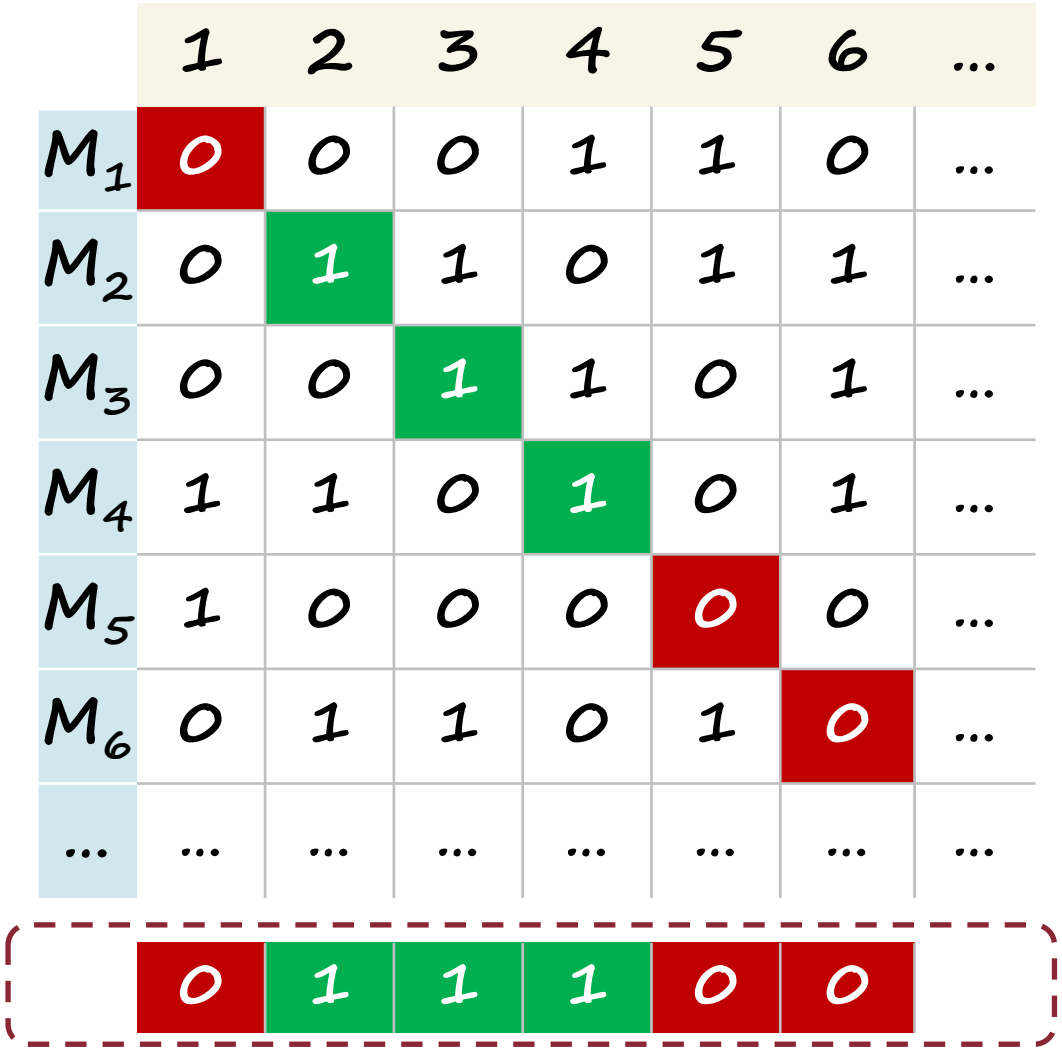
M_3 accepts w_4 (00)

M_6 does not accept w_6(10)

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

Row i represents the language of M_i

$L(M_4)=\{w_1,w_2,w_4,w_6\}$


1 2 3 4 5 6 ...
M1 0 0 0 1 1 0 ...
M2 0 1 1 0 1 1 ...
M3 0 0 1 1 0 1 ...
M4 1 1 0 1 0 1 ...
M5 1 0 0 0 0 0 ...
M6 0 1 1 0 1 0 ...
... ...
0 1 1 1 0 0

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1	0	0	0	1	1	

Flip this language.
Swap what's included
and what's excluded

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1	0	0	0	1	1

No TM has this behavior!

Diagonalization Language

- The language L_d, the diagonalization language, is the set of string w_i such that w_i is not in $L(M_i)$

$$L_d = \{w_i | w_i \notin L(M_i)\}$$

- Notice that the code for M_i is just w_i, so we can write L_d in a more interesting way:

$$L_d = \{M | M \notin L(M)\}$$

- The diagonalization language is all the codes of Turing machine which does not accept its own code.

Where We're Going

- A string is a sequence of characters.
  - There are at most as many programs as there are strings.
  - There are at least as many problems as there are sets of strings.
- **From Cantor's Theorem, we know that there are more sets of strings than strings**

$$|\text{Programs}| \leq |\text{Strings}| < |\text{Set of strings}| \leq |\text{Problems}|$$

Where We Are

- Now can we answer the question:

  What problems can a computer solve?

- Unfortunately we can only tell there exist some unsolvable(undecidable) problems. But we cannot know exactly what problems can be solved.
  - Actually there's little understanding about the sufficient condition for a problem to be computable.

Where We Are

- Knowing some problem is undecidable means:
  - Hold back our ambition of finishing an impossible task. (The universal program verifier does not exist.)
  - Then try to find another decidable problem to solve. (But we can build a verifier for specific programs.)
  - And it is interesting itself.

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

Part2. Undecidability.