Turing Machines

What is a computer?

```
void hello( String name)
{
   print( "Hello, ", name);
}
```

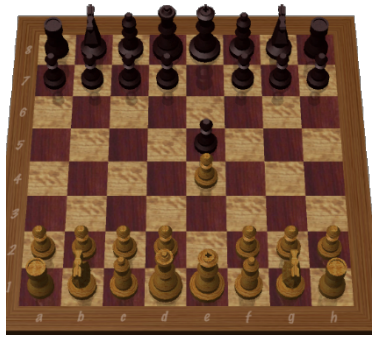
world

?

Hello, world

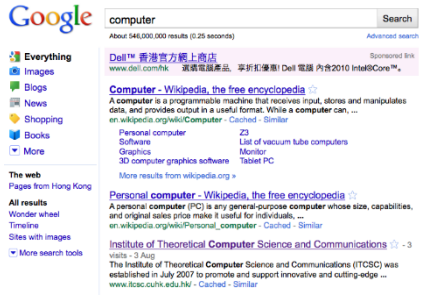
E2-E4

?

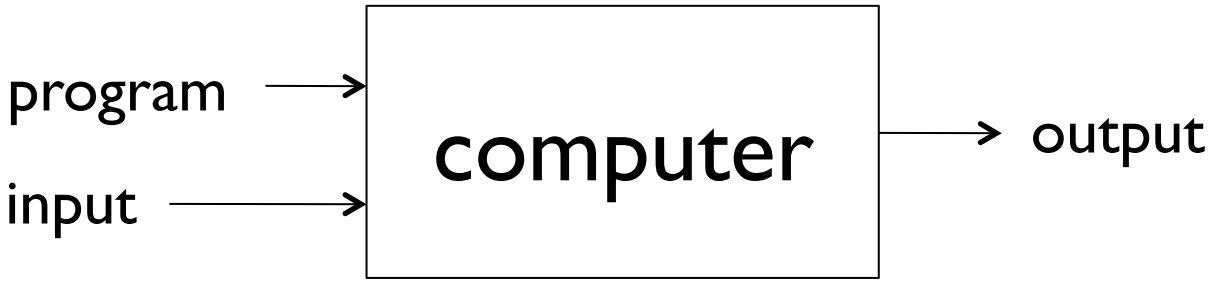


computer

?



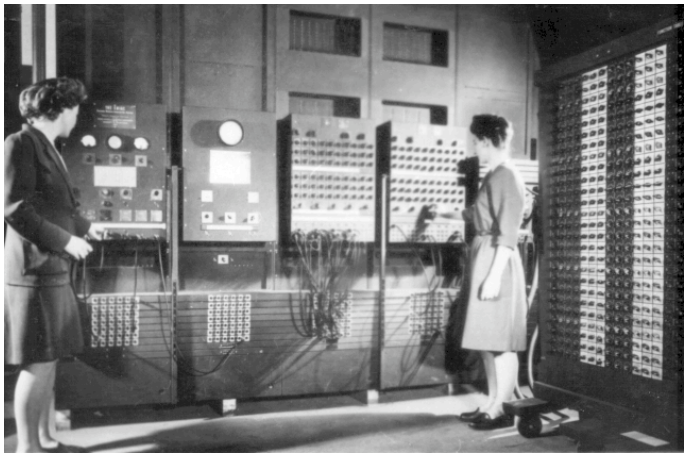
What is a computer?



A computer is a machine that manipulates data according to a list of instructions.

A brief history of computing devices

Z3 (Germany, 1941)

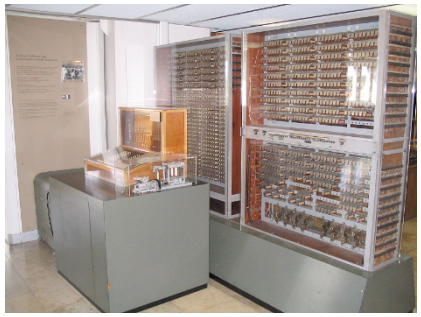


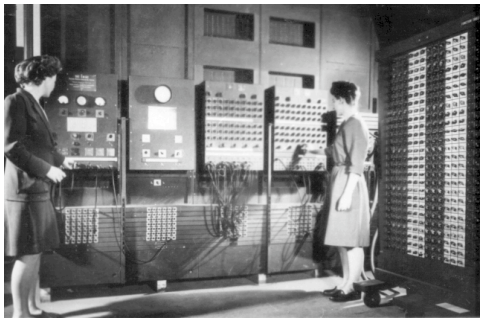
ENIAC (Pennsylvania, 1945)

PC (1980)

MacBook Air (2008)

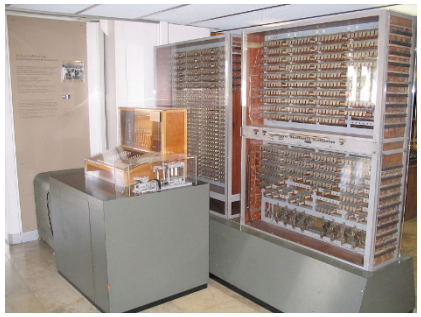
Computation is universal

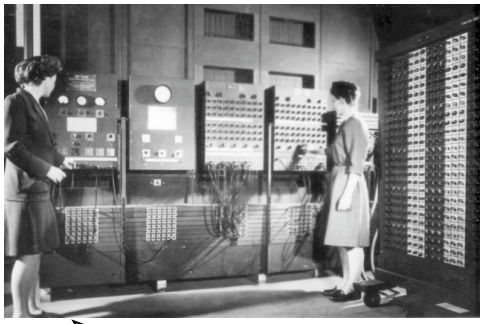




In principle, all these
have the same problem-solving ability

The Church-Turing Thesis

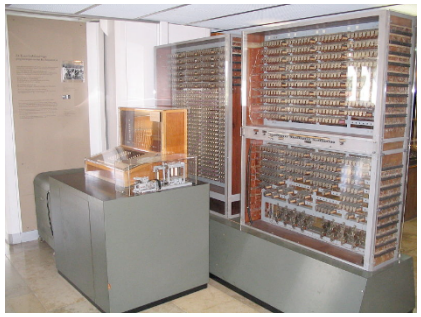


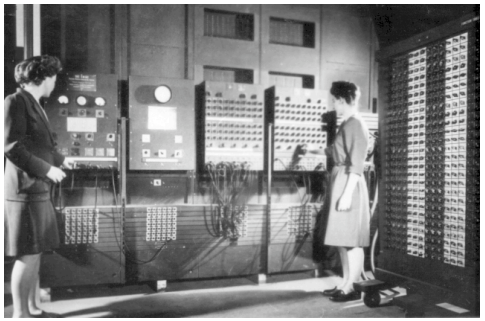


Turing Machine

If an algorithm can be implemented on any realistic computer, then it can be implemented on a Turing Machine.

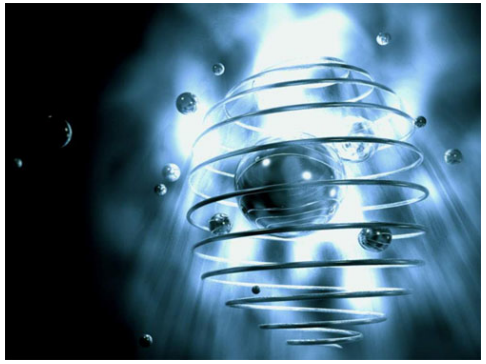
The Church-Turing Thesis





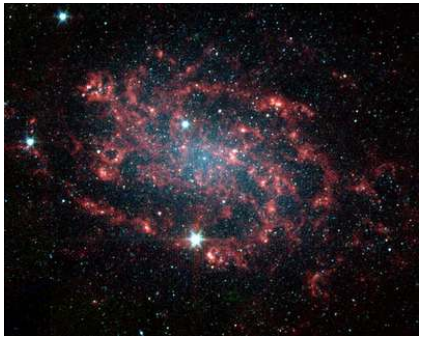
Turing Machine

quantum computing



DNA computing

cosmic computing



Alan Turing

Alan Turing
(1912-1954)

- Invented the Turing Test to tell humans and computers apart
- Helped to break German encryption system (Enigma) in World War II
- The Turing Award is the "Nobel prize of Computer Science"

Turing's motivation: By studying Turing Machines, we can understand the limitations of real computers.

Computer program analysis

```
public static void main(String args[]) {
  System.out.println("Hello World!");
}
```

What does this program do?

```
public static void main(String args[]) {
  int i = 0;
  for (j = 1; j < 10; j++) {
    i += j;
    if (i == 28) {
      System.out.println("Hello World!");
    }
  }
}
```

How about this one?

Computers cannot analyze programs!

No Turing Machine can do this:

input: The code of a java program P

Accept if P prints "hello, world"

Reject if not

Significance: It is impossible for computer to predict what a computer program will do!

How do you argue things like that?

To argue what computers cannot do, we need to have a precise definition of what a computer is.

Turing's answer: *A computer is just a Turing Machine.*

1936: "On Computable Numbers, with an Application to the *Entscheidungsproblem*"

Hilbert's list of 23 problems

David Hilbert
(1862-1943)

- Leading mathematician in the 1900s
- At the 1900 International Congress of Mathematicians, proposed a list of 23 problems for the 20th century
- Hilbert's 10th problem:

Find a method to tell if an equation like
$xyz - 3xy + 6xz + 2 = 0$ has integer solutions

Undecidability

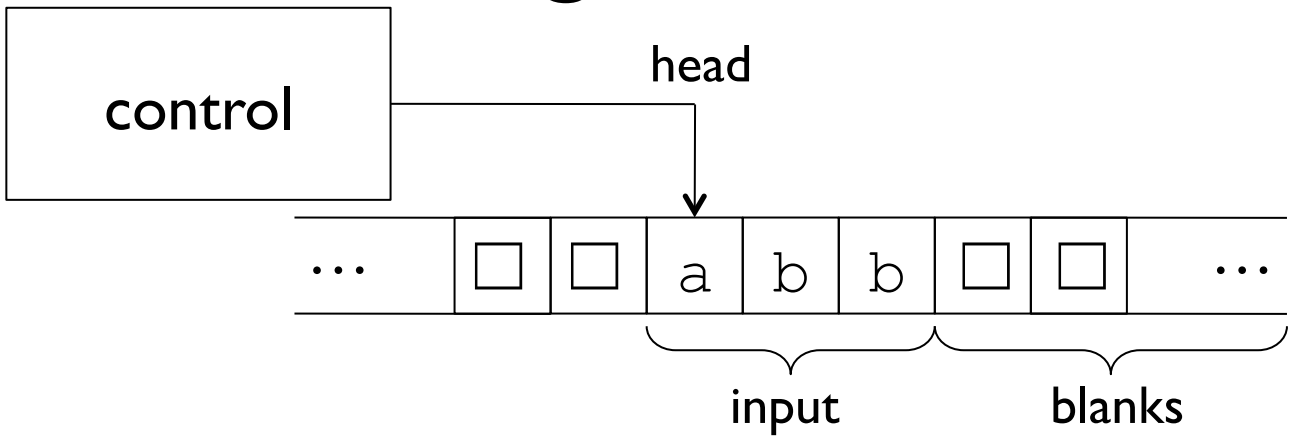
- What Hilbert meant to ask was:

> Find a Turing Machine to tell if an equation like $xyz - 3xy + 6xz + 2 = 0$ has integer solutions

- Building on work of Gödel, Church, Turing, Davis, Putnam, Robinson, in 1970 Matijasievič showed:

> There is no such Turing Machine!

Turing Machines



Can both read from and write to the tape

Head can move both left and right

Tape is infinite

Has two special states accept and reject

Example

$L_1 = \{w\#w: w \in \{\mathtt{a}, \mathtt{b}\}^*\}$

Strategy:

Read and remember the first symbol	abbaa#abbaa
Cross it off (x)	xbbaa#abbaa
Read the first symbol past #	xbbaa#abbaa
If they don't match, reject	
If they do, cross it off	xbbaa#xbbaa

Example

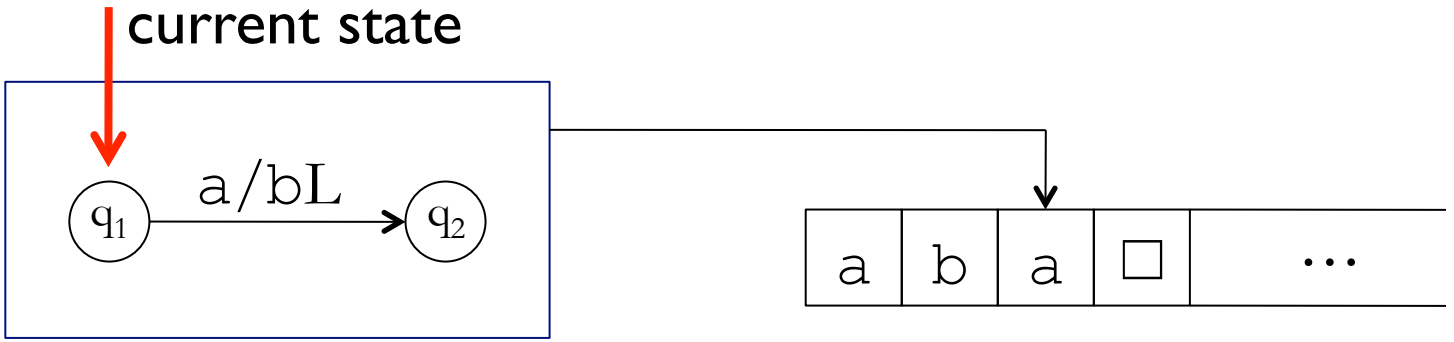
$L_1 = \{w\#w: w \in \{\mathtt{a}, \mathtt{b}\}^*\}$

Strategy:

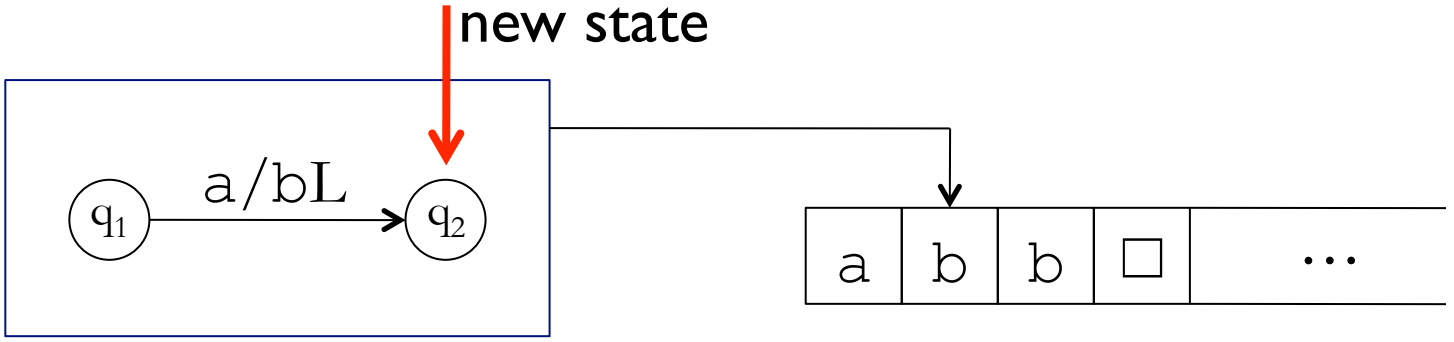
Look for the first uncrossed symbol	x<u>b</u>baa#xbbaa
Cross it off (x)	x<u>x</u>baa#xbbaa
Read the first uncrossed symbol past #	xxbaa#x<u>b</u>baa
If they match, cross it off, else reject	xxbaa#x<u>x</u>baa
At the end, there should be only xs and #s	xxxxx#xxxx<u>x</u>

If not, reject; otherwise, accept.

How Turing Machines operate



Replace a with b, and move head left



Formal Definition

A Turing Machine is $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$:

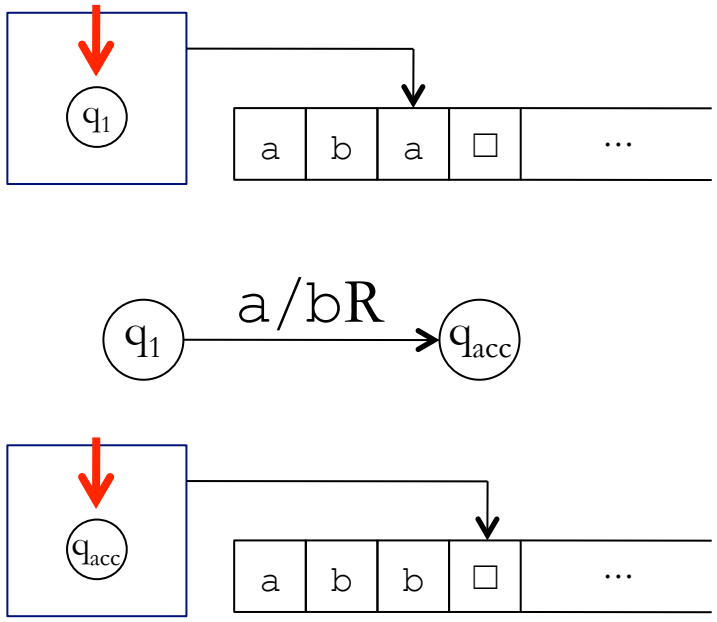
- Q is a finite set of states;
- Σ is the input alphabet not containing the blank symbol $\square$
- Γ is the tape alphabet ($\Sigma \subseteq \Gamma$) including $\square$
- q_0 in Q is the start state;
- q_{acc}, q_{rej} in Q are the accepting and rejecting state
- δ is the transition function

$$\delta: (Q - \{q_{acc}, q_{rej}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

Turing Machines are deterministic

Configurations

- A configuration consists of the current state, the head position, and tape contents



configuration

abq_1a

$abbq_{acc}$

Configurations

- We say configuration C yields C' if the TM can go from C to C' in one step

$$\mathtt{abq_1a} \quad \textbf{yields} \quad \mathtt{abbq_{acc}}$$

- The start configuration of the TM on input w is $q_0 w$

- An accepting configuration is one that contains q_{acc};
  A rejecting configuration is one that contains q_{rej}

The language of a Turing Machine

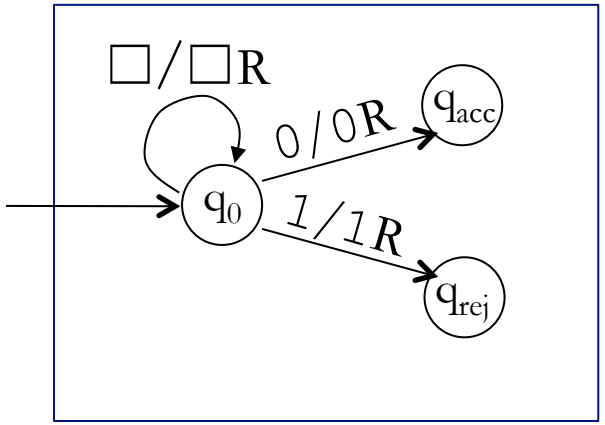
- We say M accepts x if there exists a sequence of configurations $C_0, C_1, ..., C_k$ where

C_0 is starting C_i yields C_{i+1} C_k is accepting

The language recognized by M is the set of all inputs that M accepts

Looping

- Something strange can happen in a Turing Machine:

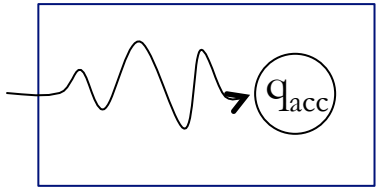


$\Sigma = \{0, 1\}$

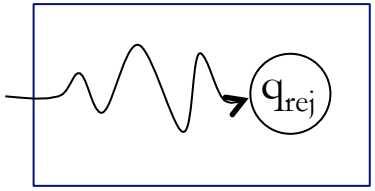
input: ε

This machine never halts

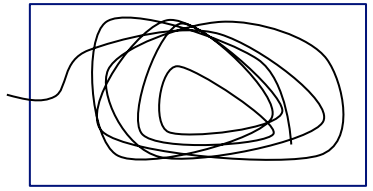
- Inputs can be divided into three types



accept



reject



loop

Halting

- We say M halts on x if there exists a sequence of configurations $C_0, C_1, ..., C_k$ where

C_0 is starting $\quad$ C_i yields C_{i+1} $\quad$ C_k is accepting or rejecting

A TM M is total if it halts on every input.

Language L is recursive (decidable) if it is recognised by a total TM.

Programming Turing Machines

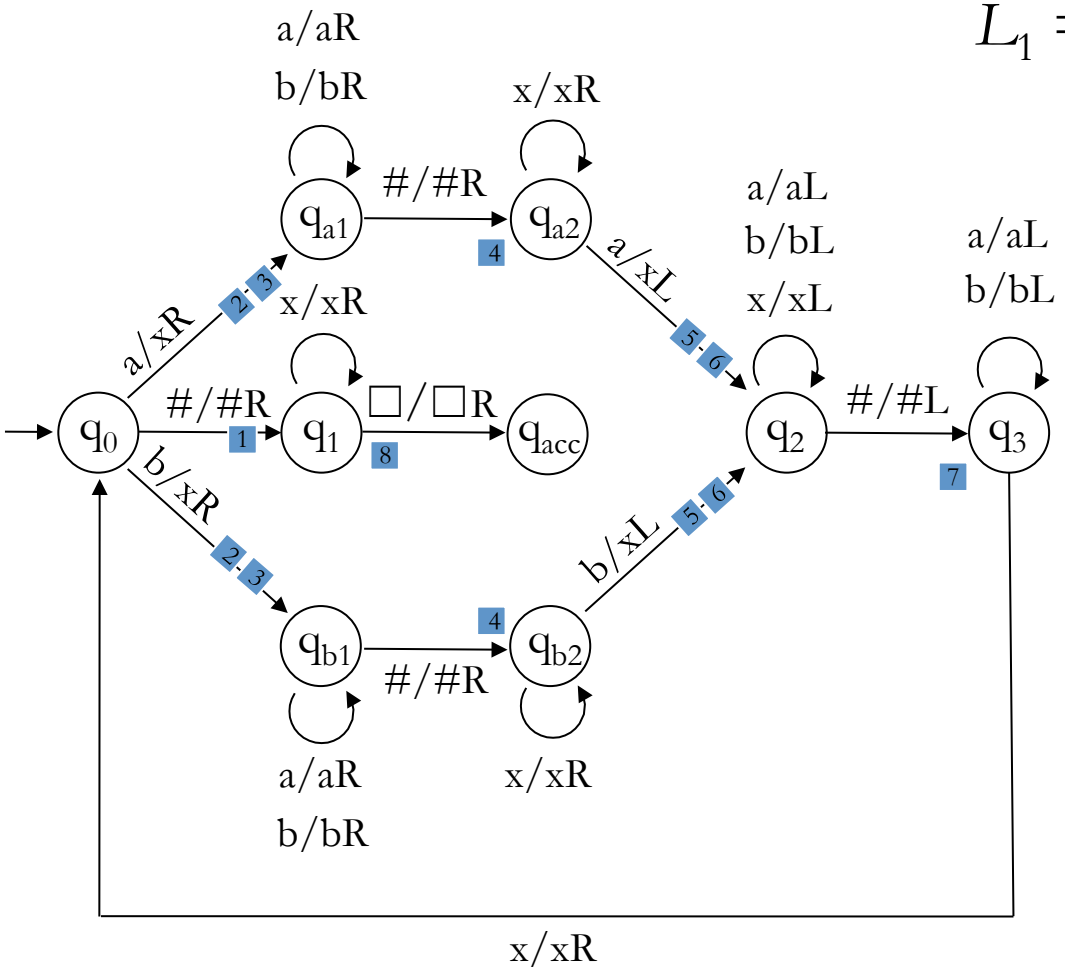
Description of Turing Machine: $L_1 = \{w\#w: w \in \{a, b\}^*\}$

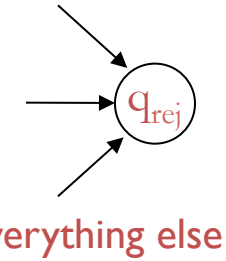
1. Until you reach #
2. Read and remember entry — xbbaa#xbbaa
3. Write x — xxbaa#xbbaa
4. Move right past # and past all xs — xxbaa#xbbaa
5. If this entry is different, reject

   Otherwise
6. Write x — xxbaa#xxbaa
7. Move left past # and to right of first x — xxbaa#xxbaa
8. If you see only xs followed by □, accept

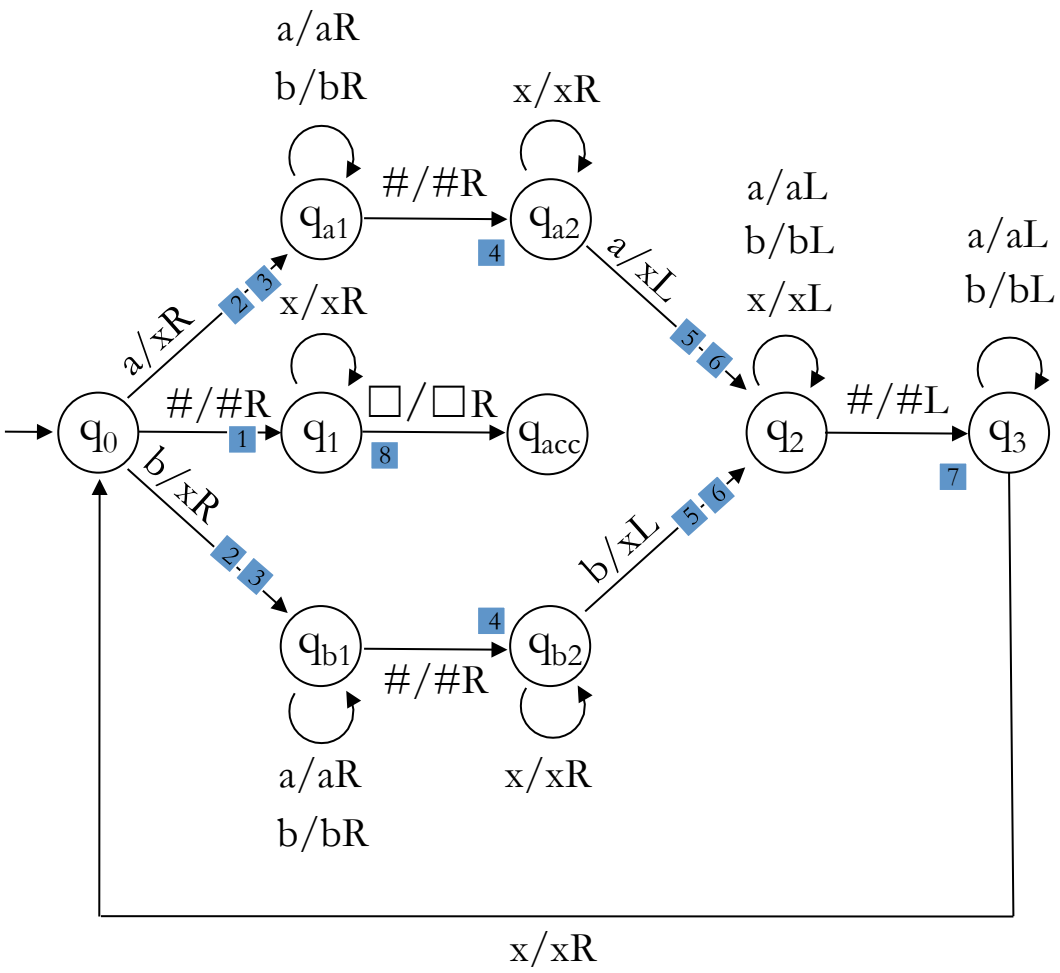
Programming Turing Machines

$L_1 = \{w\#w : w \in \{a, b\}^*\}$





Programming Turing Machines



input: aab#aab

configurations:

$q_0$aab#aab
xq_{a1}ab#aab
xaq_{a1}b#aab
xabq_{a1}#aab
xab#q_{a2}aab
xabq_2#xab
xaq_3b#xab
x$q_3$ab#xab
$q_3$xab#xab
x$q_0$ab#xab

Programming Turing Machines

$L_2 = \{ a^i b^j c^k : i \cdot j = k \text{ and } i, j, k > 0 \}$

High-level description of TM:

For every a:

  Cross off the same number of bs and cs

  Uncross the crossed bs (but not the cs)

  Cross off this a

If all as and cs are crossed off, accept.

$\Sigma = \{a, b, c\}$ $\Gamma = \{a, b, c, \cancel{a}, \cancel{b}, \cancel{c}, \square\}$

aabbcccc

aabbcccc

aa~~bbcc~~cc

aabb~~cc~~cc

~~a~~abb~~cc~~cc

~~a~~a~~bbcccc~~

~~a~~a~~bb~~~~cccc~~

Programming Turing Machines

$L_2 = \{ a^i b^j c^k : i \cdot j = k \text{ and } i, j, k > 0 \}$

Low-level description of TM:

Scan input from left to right to check it looks like `aa*bb*cc*`

Move the head to the first symbol of the tape

For every `a`: how do we know?

Cross off the same number of `b`s and `c`s ← how to do this?

Restore the crossed `b`s (but not the `c`s)

Cross off this `a`

If all `a`s and `c`s are crossed off, accept.

Programming Turing Machines

Implementation details:

aabbcccc

Move the head to the first symbol of the tape

Put a special marker on top of first a

ȧabbcccc

Cross off the same number of bs and cs:

ȧaqbbcccc

Replace b by ~~b~~

ȧa~~b~~qbcccc

Move right until you see a c

ȧa~~b~~bqcccc

Replace c by ~~c~~

ȧa~~b~~qb~~c~~ccc

Move left just past the last ~~b~~

ȧa~~bb~~q~~c~~ccc

If any bs are left, repeat

ȧa~~bbc~~qccc

ȧa~~bb~~q~~cc~~cc

$\Sigma = \{a, b, c\}$ $\Gamma = \{a, b, c, \cancel{a}, \cancel{b}, \cancel{c}, \dot{a}, \dot{\cancel{a}}, \square\}$

Programming Turing Machines

$L_3 = \{\#x_1\#x_2...\#x_l : x_i \in \{0, 1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}$

#01#0011#1

High-level description of TM:

On input w,

For every pair of blocks x_i and x_j in w,

Compare the blocks x_i and x_j

If they are the same, reject.

Otherwise, accept.

Programming Turing Machines

$$L_3 = \{\#x_1\#x_2...\#x_l : x_i \in \{0, 1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}$$

Low-level description: `#01#0011#1`

0. If input is ε, or has exactly one #, accept.

1. Place a mark on the leftmost # `#̇01#0011#1`
   (i.e. replace # by #̇) and move right

2. Place another mark on next unmarked # `#̇01#̇0011#1`
   (If there are no more #, accept)

Programming Turing Machines

$L_3 = \{\#x_1\#x_2...\#x_l : x_i \in \{0, 1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}$

current state: $\dot{\#}01\dot{\#}0011\#1$

3. Compare the two strings to the right of marked $\dot{\#}$. If there are same, reject

$\dot{\#}01\dot{\#}0011\#1$

4. Move the right $\dot{\#}$ to the right

$\dot{\#}01\#0011\dot{\#}1$

If not possible, move the left $\dot{\#}$ to the next # and put the right $\dot{\#}$ on the next

If not possible, accept

5. Repeat Step 3

$\dot{\#}01\#0011\dot{\#}1$

Programming Turing Machines

$L_3 = \{\#x_1\#x_2...\#x_l : x_i \in \{0, 1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}$

3. Compare the two strings to the right of marked $\dot{\#}$. If there are same, reject

4. Move the right $\dot{\#}$ to the right
   If not possible, move the left $\dot{\#}$ to the next # and put the right $\dot{\#}$ on the next
   If not possible, accept

$\dot{\#}01\#0011\dot{\#}1$

$\#01\dot{\#}0011\dot{\#}1$

accept

How to describe Turing Machines?

- Unlike for DFAs, NFAs, PDAs, Turing Machines are rarely given as complete state diagrams
- Instead usually a high-level description is given: A recipe about the workings of the Turing Machine
- However, in the exam and class test state diagrams will be used

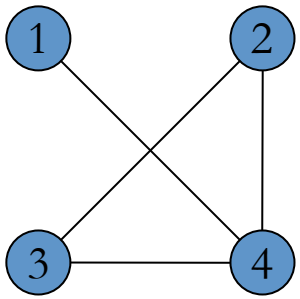
Practice makes perfect!

Programming Turing Machines

$L_4 = \{\langle G \rangle : G \text{ is a connected undirected graph}\}$

Q: How to feed a graph into a Turing Machine?

A: Represent it by a string, e.g.



`(1,2,3,4)((1,4),(2,3),(3,4)(4,2))`

Convention for describing graphs:

(nodes) (edges)

no node must repeat

edges are pairs $(\text{node}_1, \text{node}_2)$

Programming Turing Machines

$L_4 = \{\langle G \rangle : G \text{ is a connected undirected graph}\}$

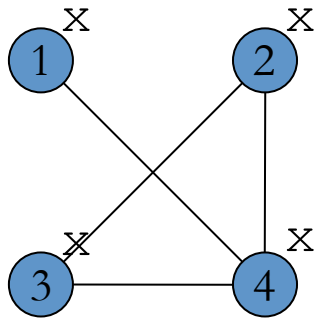
On input $\langle G \rangle$,

0. Verify that $\langle G \rangle$ is the description of a graph
   (no vertex repeats; edges only go between nodes)

1. Mark the first node of G

2. Repeat until no new nodes are marked:

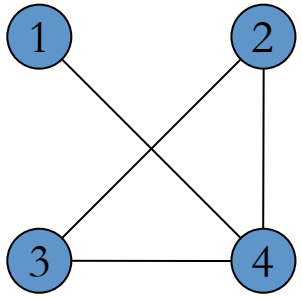
   For each node, mark it if it is attached
   to an already marked node



3. If all nodes are marked accept, otherwise reject.

Programming Turing Machines

$L_4 = \{\langle G \rangle : G \text{ is a connected undirected graph}\}$



(1̇,2,3,4)((1,4)(2,3)(3,4)(4,2))

(1̇,2,3,4)((1̲,4)(2,3)(3,4)(4,2))

(1̇,2,3,4)((1,4̲)(2,3)(3,4)(4,2))

(1̇,2,3,4̇)((1,4)(2,3)(3,4)(4,2))

(1̇,2,3,4̇)((1̲,4)(2,3)(3,4)(4,2))

(1̇,2,3,4̇)((1,4̲)(2,3)(3,4)(4,2))

(1̇,2,3,4̇)((1,4)(2̲,3)(3,4)(4,2))

(1̇,2,3,4̇)((1,4)(2,3̲)(3,4)(4,2))

(1̇,2,3,4̇)((1,4)(2,3)(3̲,4)(4,2))

(1̇,2,3̇,4̇)((1,4)(2,3)(3,4)(4,2)) etc.

Variants of Turing Machines

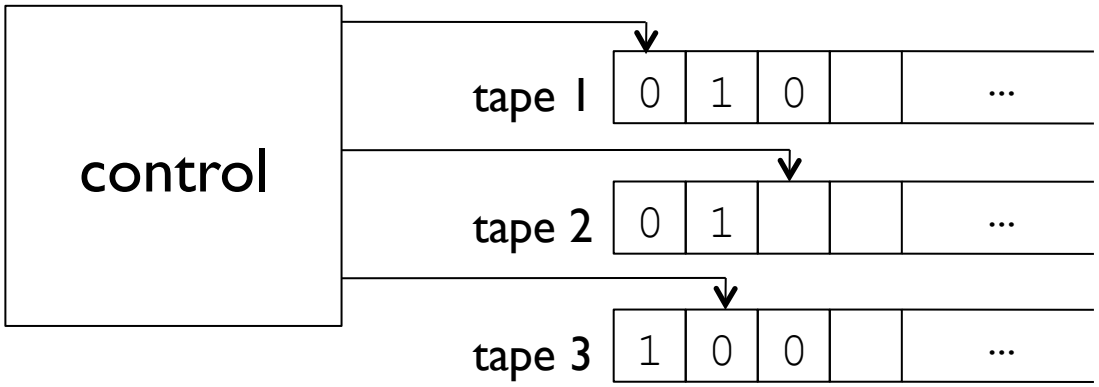
The Church-Turing Thesis

Every effectively calculable function is a computable function.

”effectively calculable” = ”produced by any intuitively
’effective’ means whatsoever”
”effectively computable” = ”produced by a Turing-machine
or equivalent mechanical device”

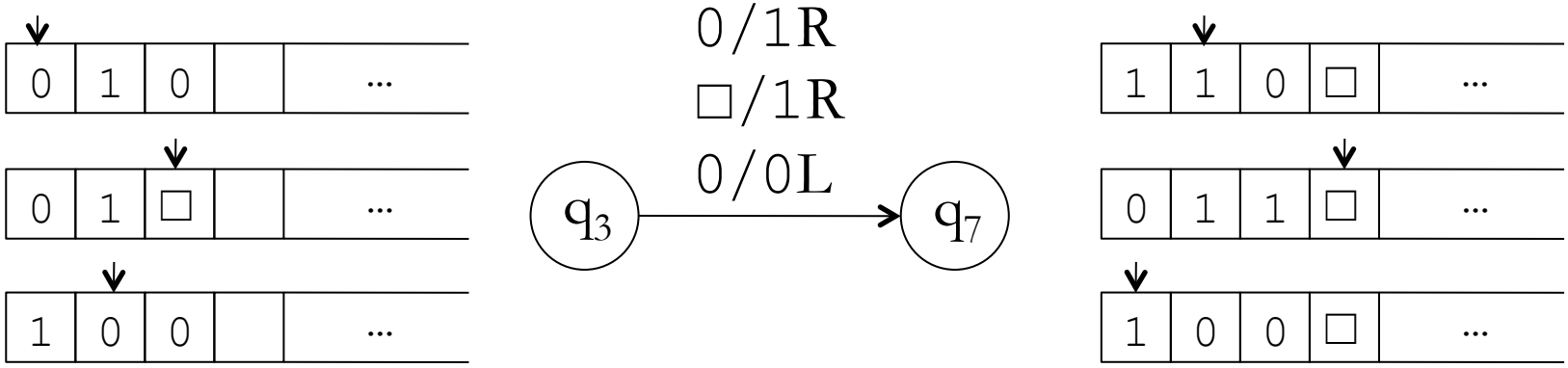
Gödel [1931]: Recursive Functions
Church [1933]: λ -Calculus
Post [1936]: Post model
Turing [1936]: a-Machines (Turing Machines)

The multitape Turing Machine



- The transition may depend on the contents of all the cells
- Different tape heads can be moved independently

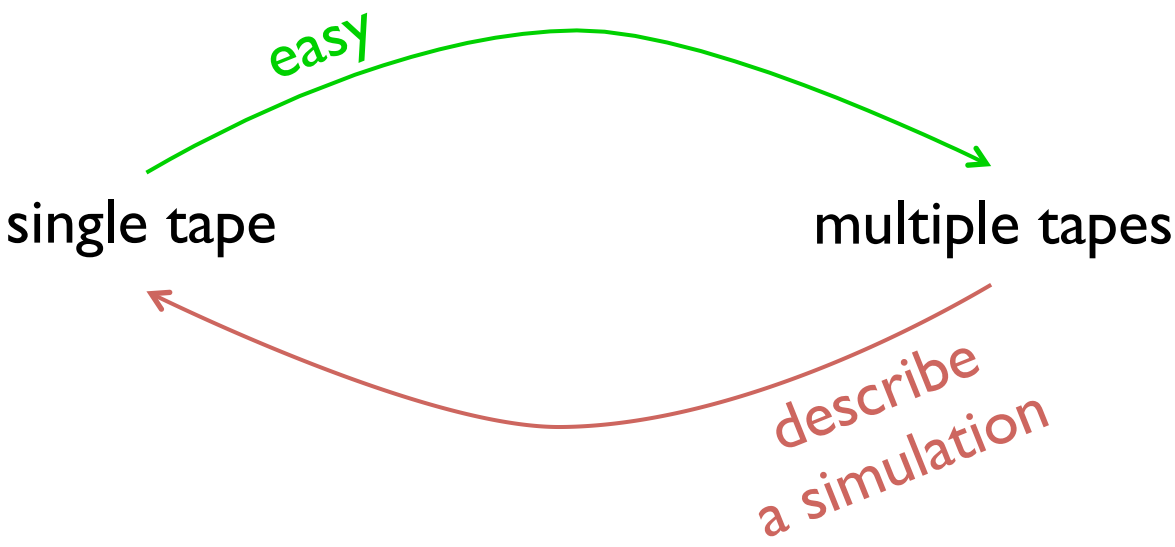
The multitape Turing Machine



- Multiple tapes are convenient,
  e.g. one can serve as temporary storage

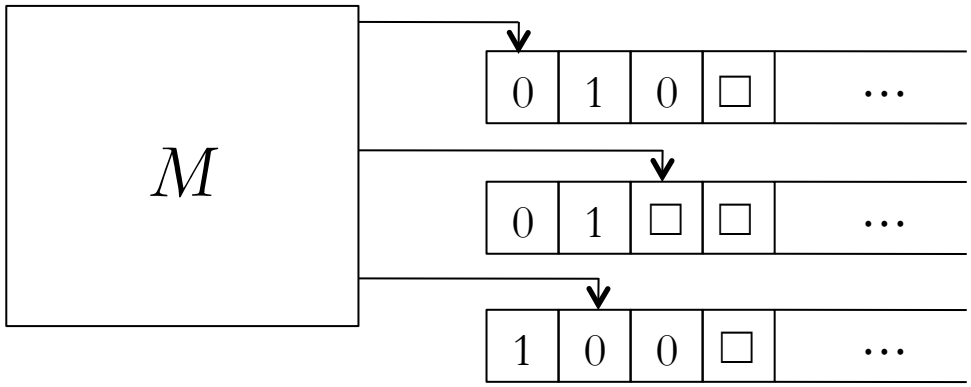
How to argue equivalence

Multitape Turing Machines are equivalent to single-tape Turing Machines

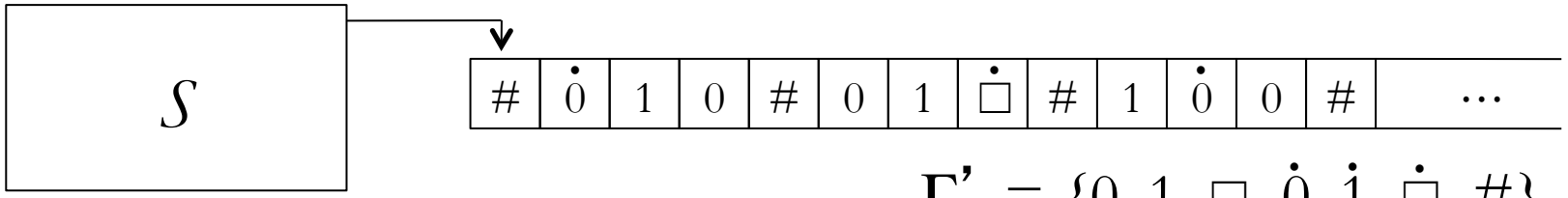


The multitape Turing Machine

Multitape Turing Machines are equivalent to single-tape Turing Machines



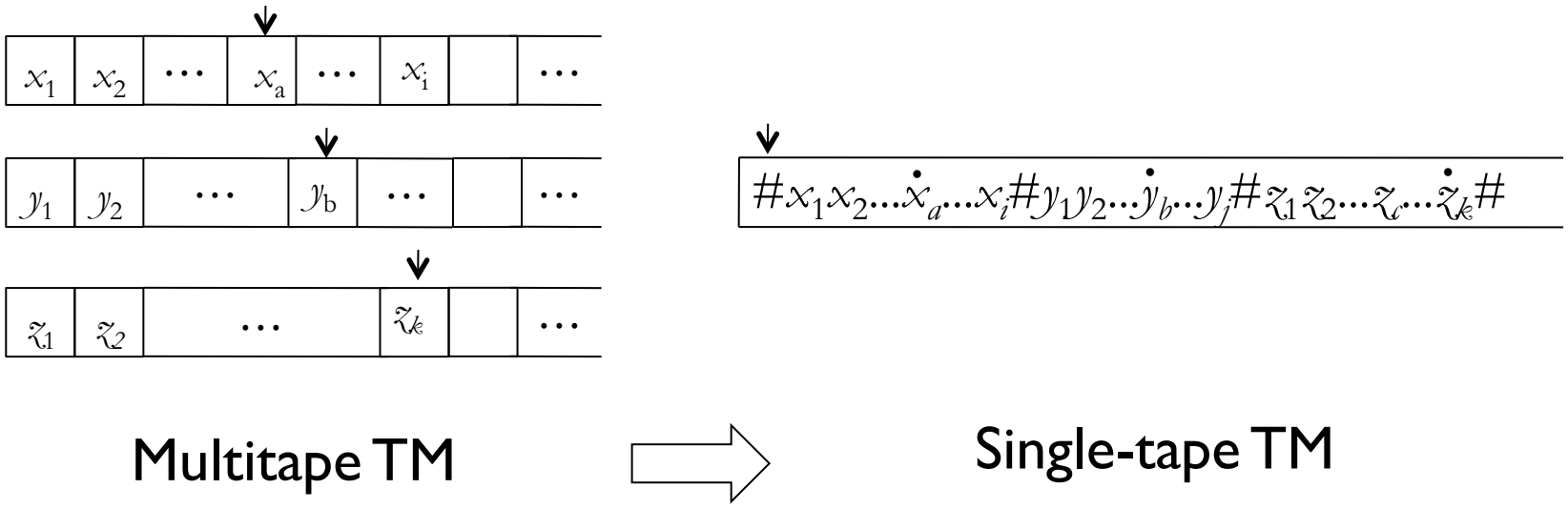
$\Gamma = \{0, 1, \square\}$



$\Gamma' = \{0, 1, \square, \dot{0}, \dot{1}, \dot{\square}, \#\}$

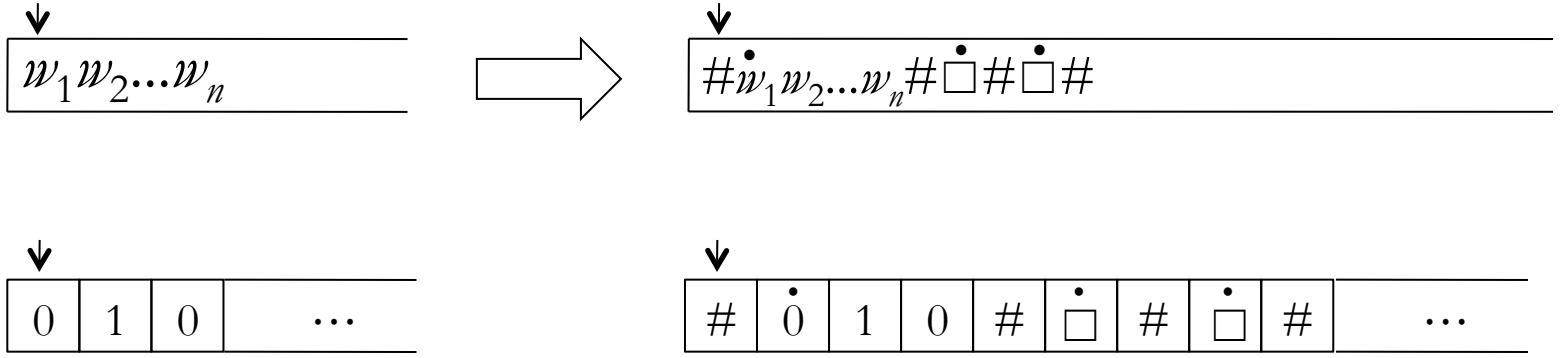
Simulating a multitape TM

- We show how to simulate a multitape TM on an single tape TM
- To be specific, let's do a 3-tape TM



Simulating a multitape TM

Single-tape TM: Initialisation



S: On input $w_1 \ldots w_n$:

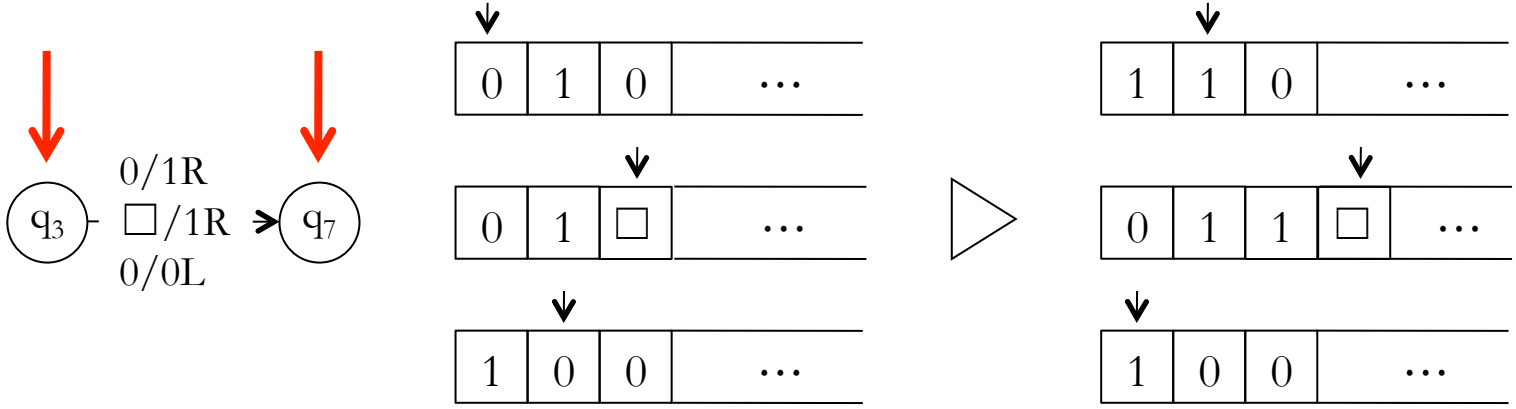
1. Replace tape contents by $\#\dot{w}_1 w_2 \ldots w_n \# \dot{\sqcup} \# \dot{\sqcup} \#$

   Remember that M is in state q_0

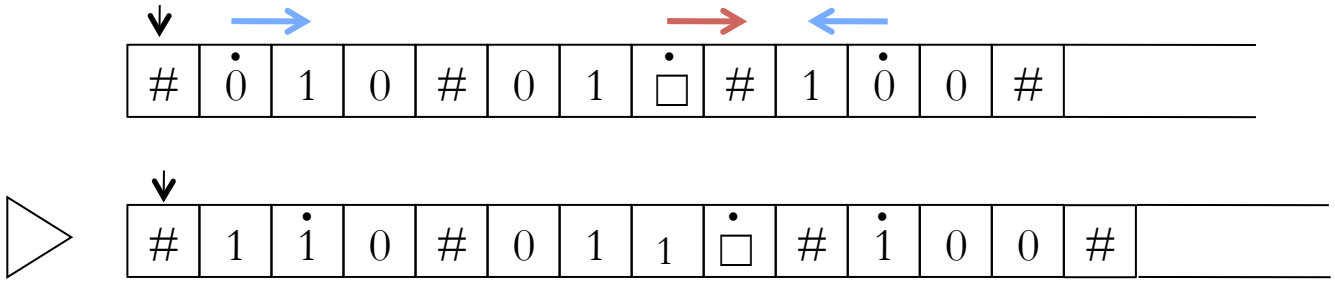
Simulating a multitape TM

Single-tape TM: Simulating Multitape TM moves

Suppose Multitape TM wants to move like this:



We simulate move on single-tape TM like this:

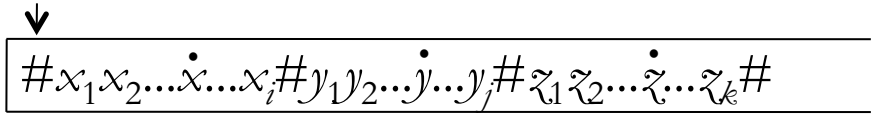


Simulating a multitape TM

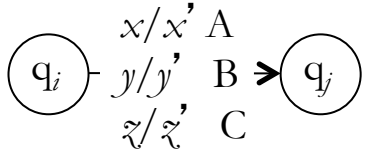
S: On input $w_1 \ldots w_n$:

1. Replace tape contents by $\#\dot{w}_1 w_2 \ldots w_n \# \dot{\sqcup} \# \dot{\sqcup} \#$
   Remember (in state) that M is in state q_0

2. To simulate a step of M:
   Make a pass over tape to find $\dot{x}, \dot{y}, \dot{z}$

   If M is in state q_i and has transition update state / tape accordingly.

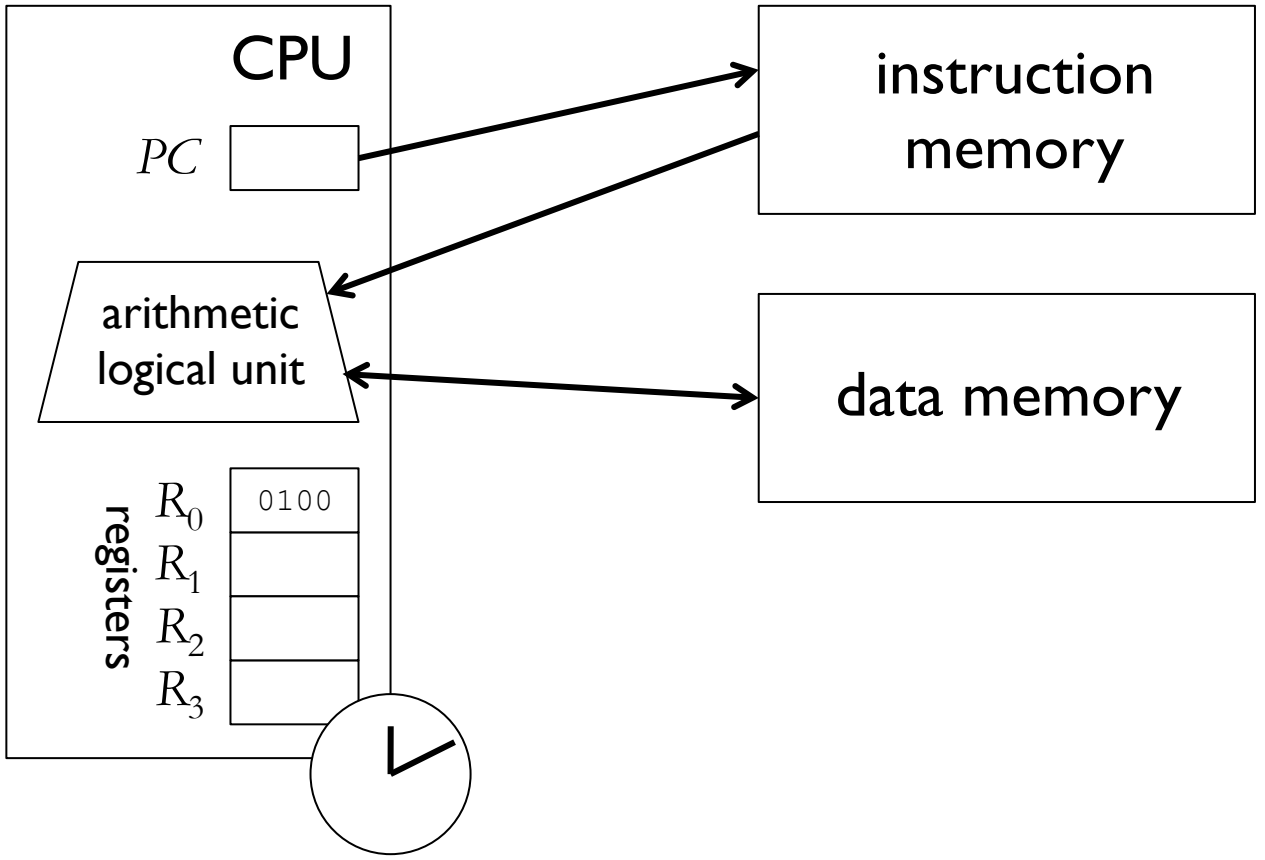
3. If M reaches accept (reject) state, accept (reject).

Doing simulations

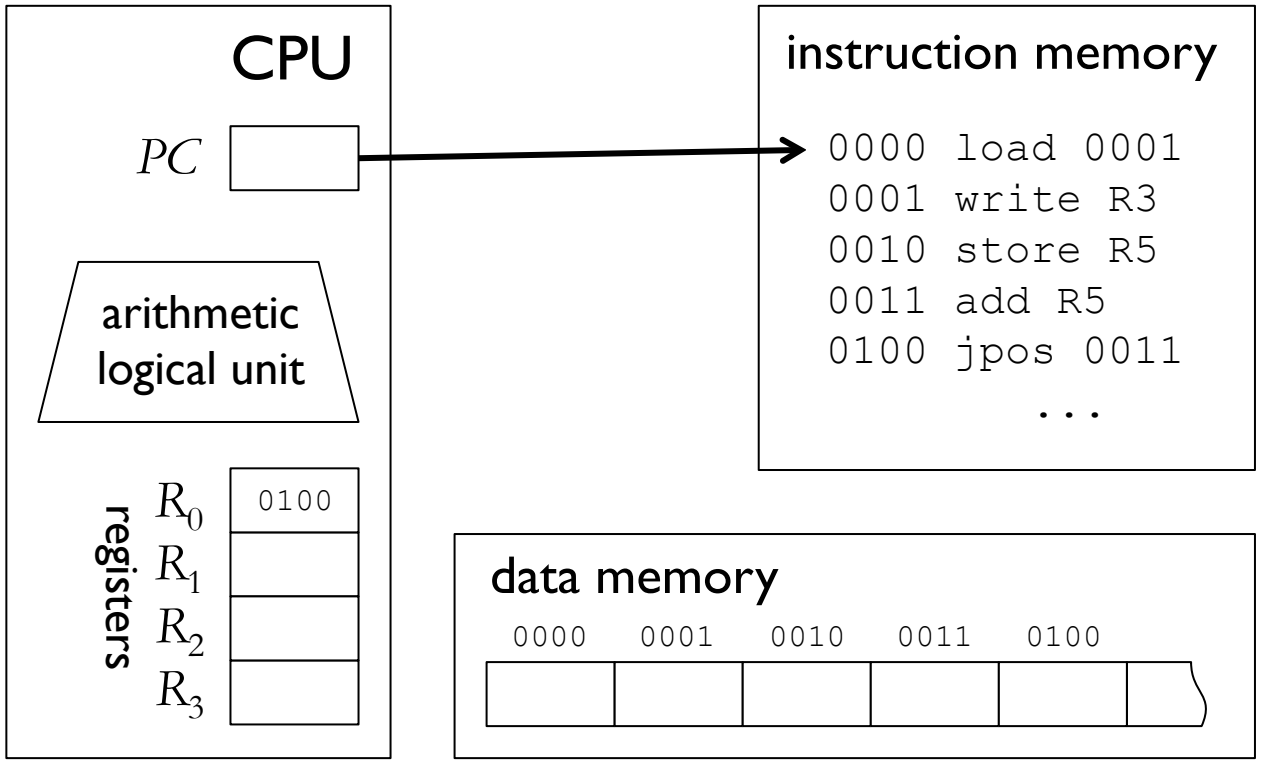
- To simulate a model M by another model N:

  1. Say how the state and storage of N is used to represent the state and storage of M
  2. Say what should be initially done to convert the input of N (if anything)
  3. Say how each transition of M can be implemented by a sequence of transitions of N

What does a computer look like?



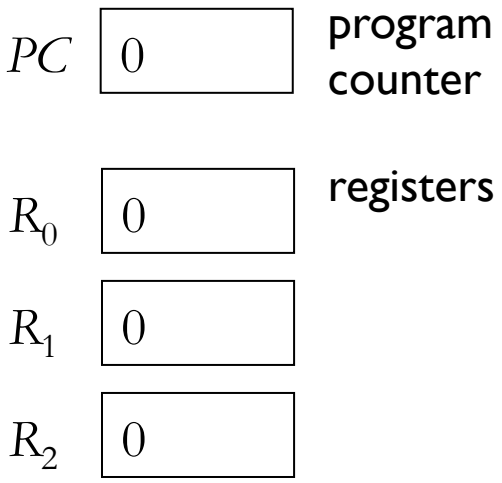
What does a computer look like?



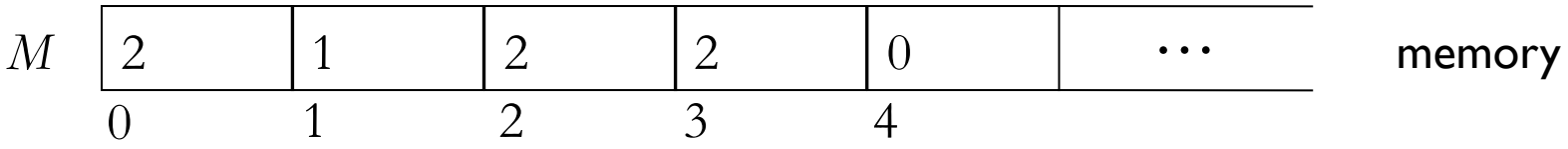
Instruction set

`load` x	Put the value x into R_0
`load R`k	Copy the value of R_k into R_0
`store R`k	Copy the value of R_0 into R_k
`read R`k	Copy the value at memory location R_k into R_0
`write R`k	Copy the value of R_0 into memory location R_k
`add R`k	Add R_0 and R_k, and put result in R_0
`jump` n	Set PC to n
`jzero` n	Set PC to n, if R_0 is zero
`jpos` n	Set PC to n, if R_0 is positive

Random access machines

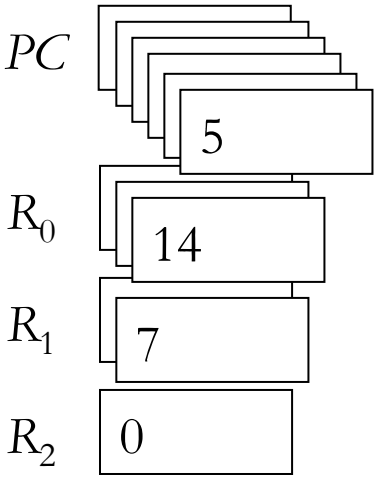


instruction	meaning
0 `load 7`	$R_0 := 7$
1 `write R2`	$M[R_2] := R_0$
2 `store R1`	$R_1 := R_0$
3 `add R1`	$R_0 := R_0 + R_5$
4 `jzero 3`	**if** $R_0 = 0$ **then** $PC := 3$
5 `accept`	

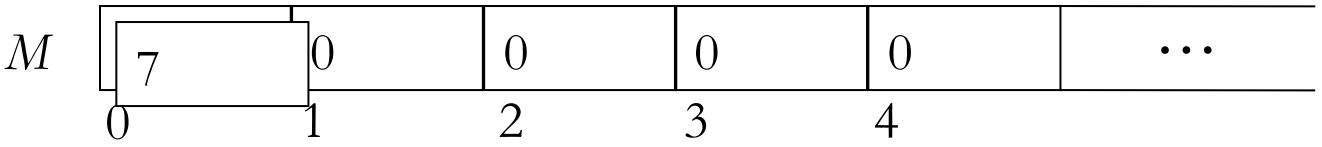


It has registers that can store integer values, a program counter, and a random-access memory

Random access machines



	instruction	meaning
0	`load 7`	$R_0 := 7$
1	`write R2`	$M[R_2] := R_0$
2	`store R1`	$R_1 := R_0$
3	`add R1`	$R_0 := R_0 + R_1$
4	`jzero 3`	**if** $R_0 = 0$ **then** $PC := 3$
5	`accept`	

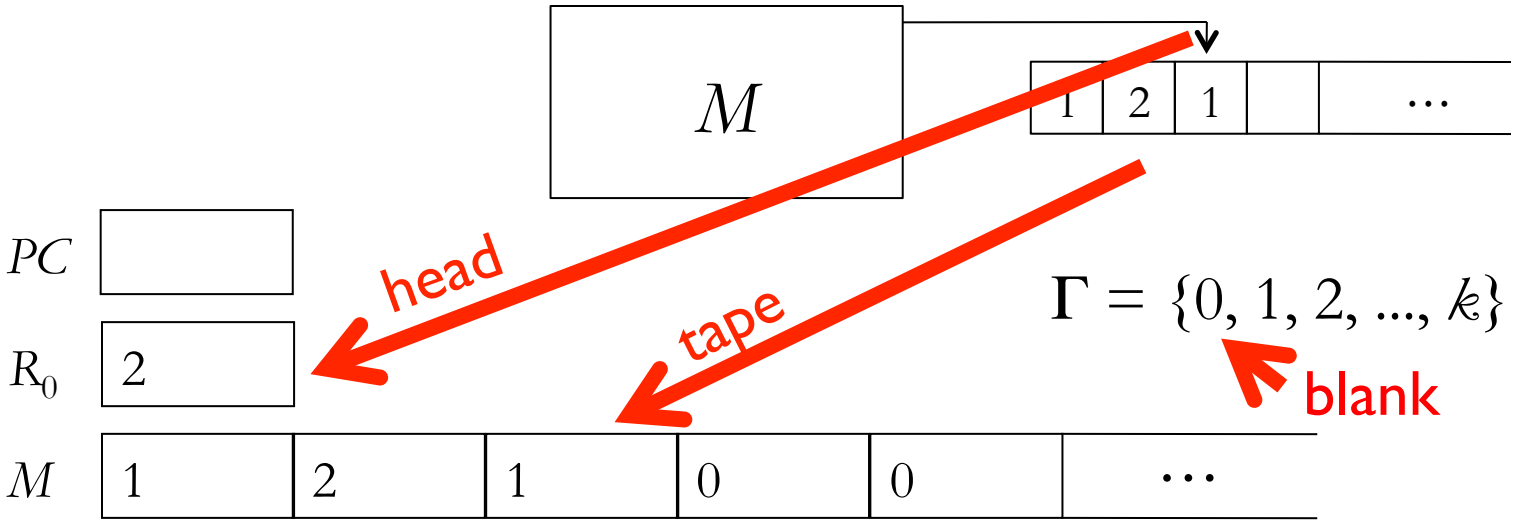


- The instructions are indexed by the program counter
- Initially, the input is in the first k memory cells, all registers and PC are set to 0

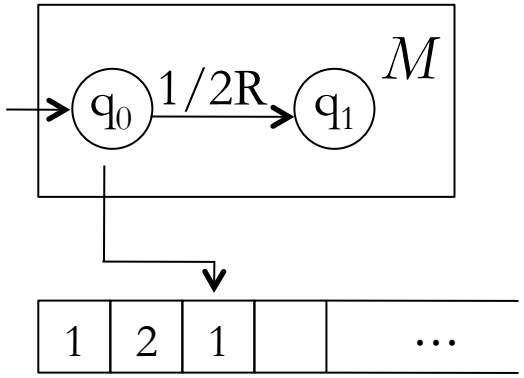
Random access machines

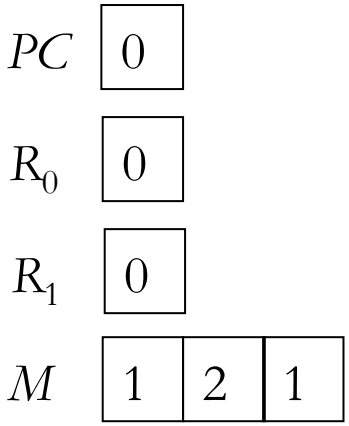
Random access machines are equivalent to Turing Machines

- Simulating a Turing Machine on a RAM:



Simulating a TM on a RAM





program

```
 0 store R1
 1 read R1
 2 add -1
 3 jzero 6

 6 load 2
 7 write R1
 8 load R1
 9 add 1
10 jump 30
   …
30 store R1
   …
200 accept
```

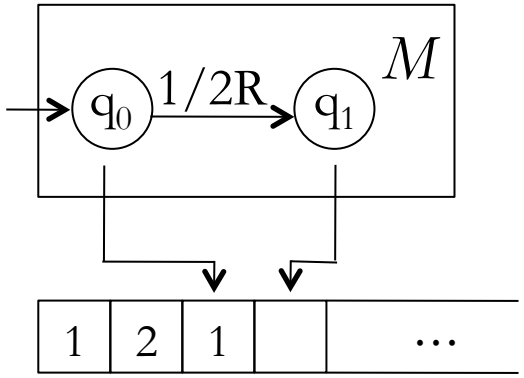
save head position
read tape contents x
if $x = 1$ goto line 6

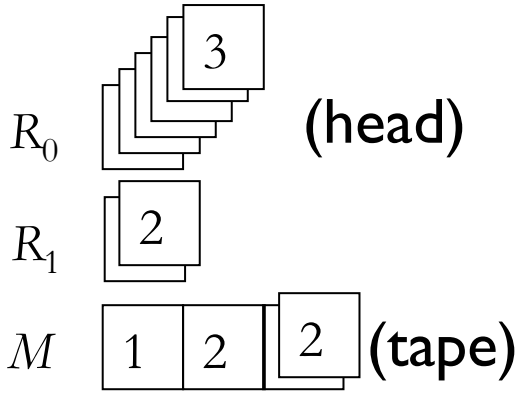
new value of cell
write in memory
recall head position
move head to right
go to state q_1

handle for state q_1

handle for state q_{acc}

Simulating a TM on a RAM





program

```
0 store R1
1 read R1
2 add -1
3 jzero 6

6 load 2
7 write R1
8 load R1
9 add 1
10 jump 30
   …
30 store R1
   …
```

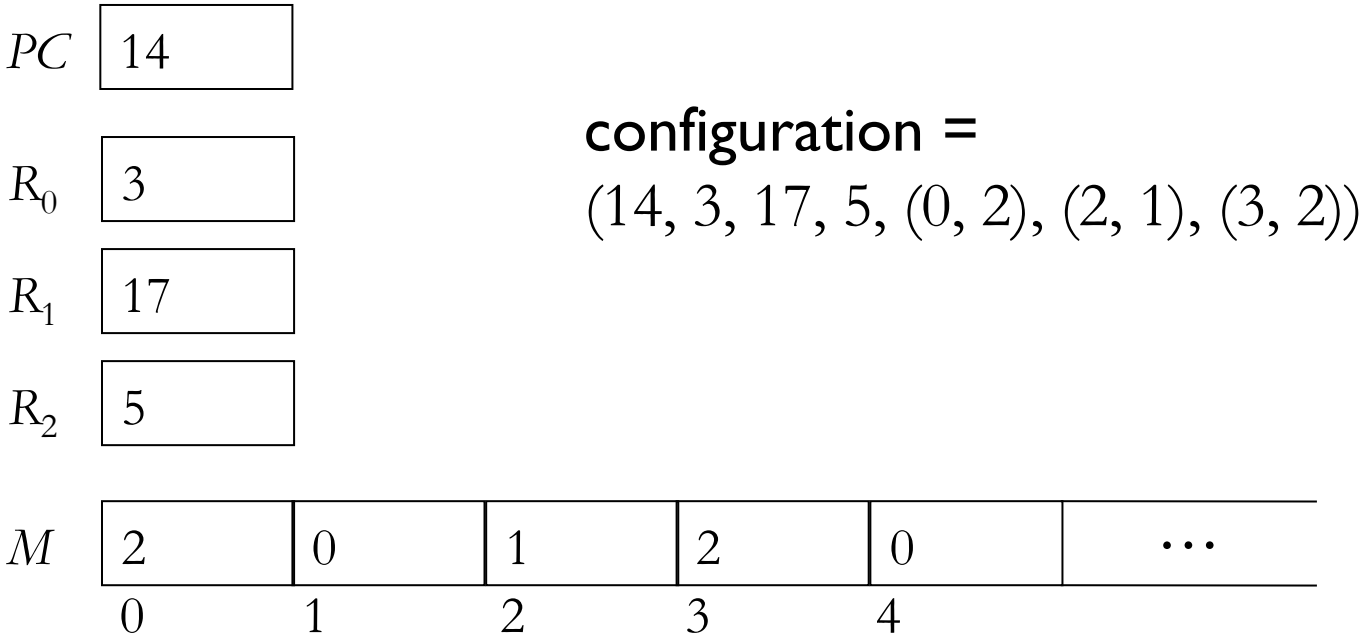
save head position
read tape contents x
if $x = 1$ goto line 6

new value of cell
write in memory
recall head position
move head to right
go to state q_1

handle for state q_1

Simulating a RAM on a Turing Machine

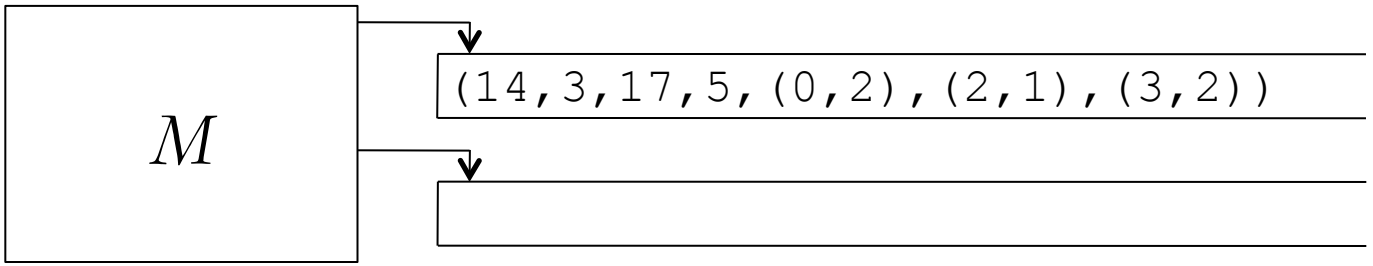
- The configuration of a RAM consists of
  - Program counter
  - Contents of registers
  - Indices and contents of all nonempty memory cells



configuration =
(14, 3, 17, 5, (0, 2), (2, 1), (3, 2))

Simulating a RAM on a 2-tape TM

- The TM has a simulation tape and a scratch tape
- The simulation tape stores RAM configuration



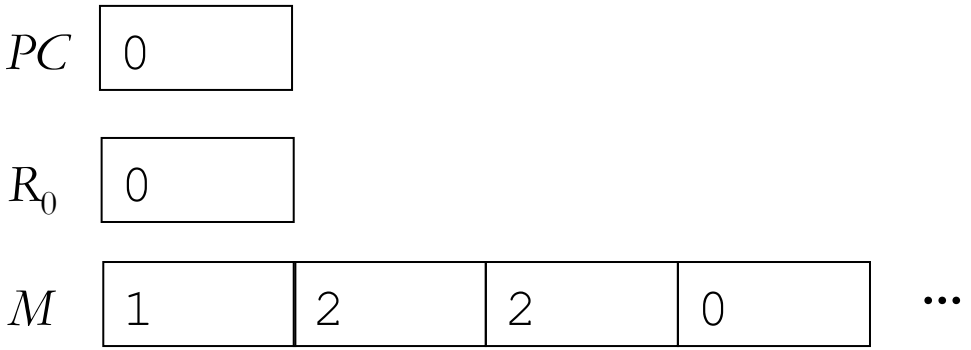
- The TM has a set of states corresponding to each program instruction of the RAM
- The TM tape is updates according to RAM instruction

Simulating a RAM on a 2-tape TM

Initialization

TM input: `122`

RAM initial state:

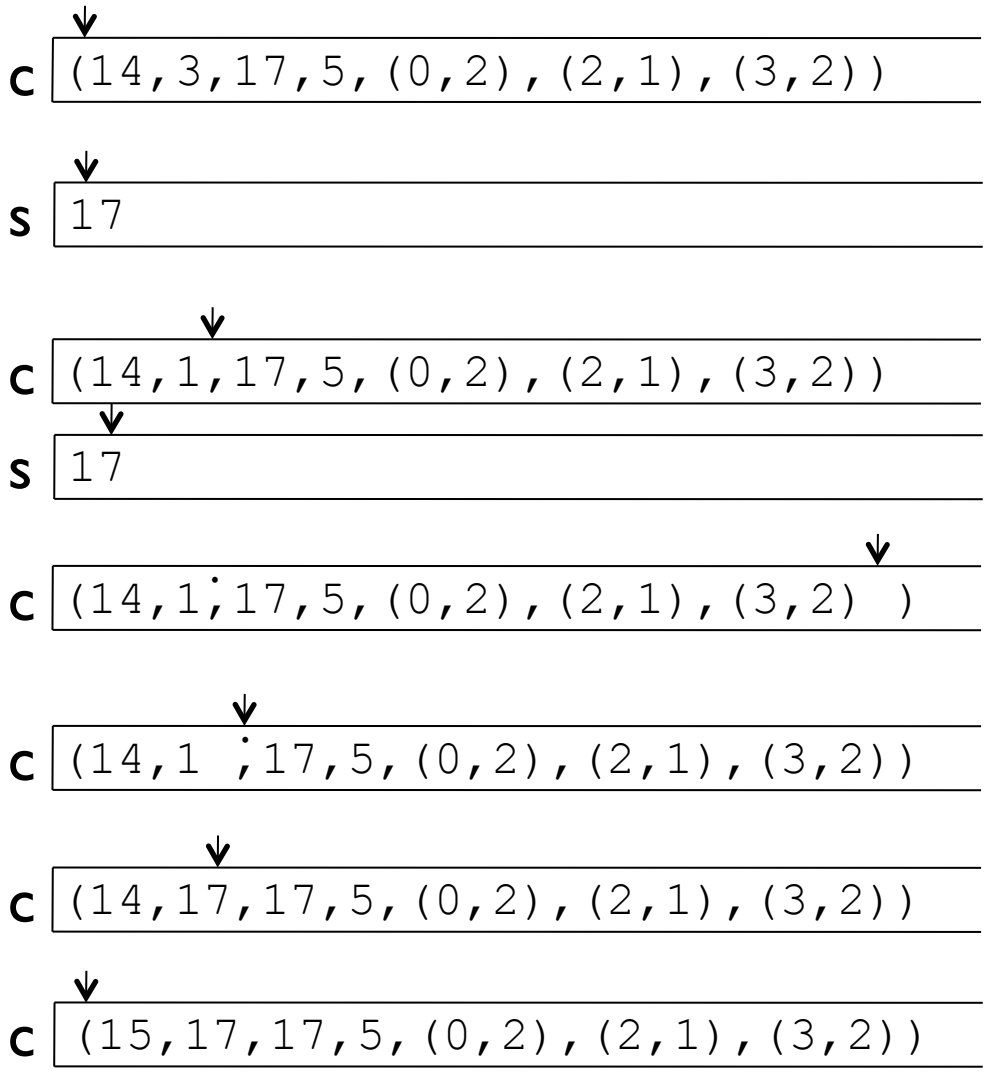


S: On input $w_1 ... w_n$:

1. Replace tape contents by
   $(0, 0, 0, ..., 0, (0, w_1), (1, w_2), ..., (n\text{-}1, w_n))$

Simulating a RAM on a 2-tape TM

- Example: `load R1`



Simulating a RAM on a 2-tape TM

S: On input $w_1 ... w_n$:

1. Replace tape contents by
   $(0, 0, 0, ..., 0, (0, w_1), (1, w_2), ..., (n\text{-}1, w_n))$
2. Simulate instruction in RAM program
   by a sequence of TM transitions
3. If RAM instruction is accept, go to accept state.
   If RAM instruction is reject, go to reject state.

Running time

- The running time of TM M is the function $t_M(n)$:

> $t_M(n)$ = maximum number of steps that M takes on any input of length n

$L = \{w\#w\colon w \in \{\mathtt{a}, \mathtt{b}\}\}$

M:	On input x, until you reach #	$O(n)$ times
	Read and cross off first a or b before #	$O(n)$ steps
	Read and cross off first a or b after #	
	If there is a mismatch, reject	
	If all symbols but # are crossed off, accept	$O(n)$ steps
	running time:	$O(n^2)$

Measuring running time

- What does it mean when we say:

  "This algorithm runs in time T"

- One "time unit" in

java

```
if (x > 0)
  y = 5*y + x;
```

RAM machine

```
write r3;
```

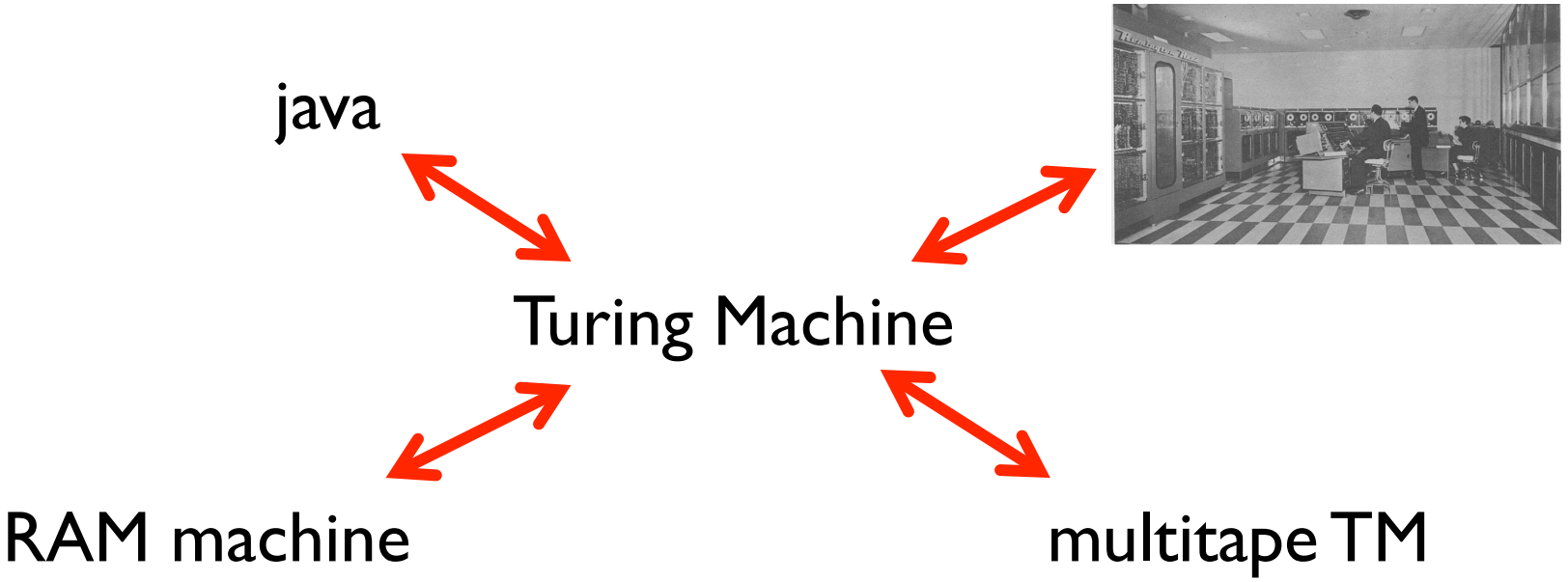
Turing Machine

$\delta(q_3, a) = (q_7, b, R)$

all mean different things!

Efficiency and the Church-Turing thesis

- The Church-Turing thesis says all these are equivalent in computing power…



… but not in running time!

The Cobham-Edmonds thesis

- It states that a computational problem can be feasibly solved on a computational device only if they can be computed in polynomial time, i.e., in time $O(n^c)$ where n is the size of the input and c is some constant.

- This differ between machines considered, e.g., quantum machines are able to factorise any number is polynomial time, but no algorithm for a conventional computer is known.

Millenium prize problems

- Recall that in 1900, Hilbert gave 23 problems that guided mathematics in the 20th century

- In 2000, the Clay Mathematical Institute gave 7 problems for the 21st century

1 P versus NP ← computer science
2 The Hodge conjecture
~~3 The Poincaré conjecture~~ Perelman 2006
4 The Riemann hypothesis ← Hilbert's 8th problem
5 Yang–Mills existence and mass gap
6 Navier–Stokes existence and smoothness
7 The Birch and Swinnerton-Dyer conjecture

P vs NP

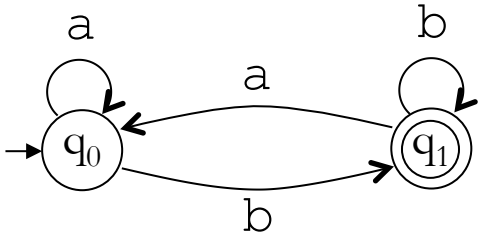
- P is the class of languages that can be decided on a TM with polynomial running time in the input length.

- NP is the class of languages that can be decided on a nondeterministic TM with polynomial running time in the input length.

Is P = NP ? (open since Cook (1971))

Decidable and undecidable languages

Problems about automata

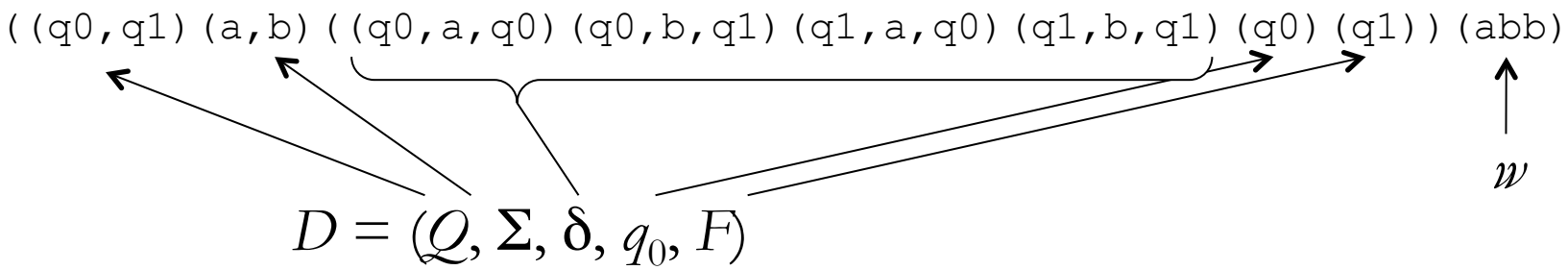
Does 

accept input abb?

- We can formulate this question as a language:

$$A_{\text{DFA}} = \{\langle D, w\rangle : D \text{ is a DFA that accepts input } w\}$$

Is A_{DFA} decidable?



Problems about automata

$A_{\text{DFA}} = \{\langle D, w\rangle : D \text{ is a DFA that accepts input } w\}$

pseudocode:

On input $\langle D, w\rangle$,
where $D = (Q, \Sigma, \delta, q_0, F)$:

Set $q := q_0$

For $i := 1$ to length(w):

$q := \delta(q, w_i)$

If $q \in F$ accept, else reject

TM description:

On input $\langle D, w\rangle$,
where D is a DFA, w is a string

Simulate D on input w

If simulation ends in acc state, *accept*. Otherwise, *reject*.

Problems about automata

$A_{\text{DFA}} = \{\langle D, w\rangle : D \text{ is a DFA that accepts input } w\}$

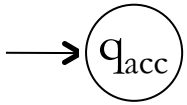
Turing Machine details:

Check input is in correct format.
(Transition function is complete, no duplicate transitions)

Perform simulation:

state input symbol

((q̇0,q̇1)(a,b)((q0,a,q0)(q0,b,q1)(q1,a,q0)(q1,b,q1)(q0)(q1))(ȧḃḃ)



Problems about automata

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

decidable

Turing Machine details:

Check input is in correct format.
(Transition function is complete, no duplicate transitions)

Perform simulation:

Put markers on start state of D and first symbol of w

Until marker for w reaches last symbol:

Update both markers

If state marker is on accepting state, *accept*. Else *reject*.

Acceptance problems about automata

$A_{\text{DFA}} = \{\langle D, w\rangle : D \text{ is a DFA that accepts input } w\}$

decidable

$A_{\text{NFA}} = \{\langle N, w\rangle : N \text{ is an NFA that accepts } w\}$

$A_{\text{REX}} = \{\langle R, w\rangle : R \text{ is a regular expression that generates } w\}$

Which of these is decidable?

Acceptance problems about automata

$A_{\text{NFA}} = \{\langle N, w\rangle : N \text{ is an NFA that accepts input } w\}$

decidable

- The following TM decides A_{NFA}:

$N :=$ On input $\langle N, w\rangle$, where N is an NFA and w is a string

Convert N to a DFA D using the standard conversion procedure.

Run the TM M for A_{DFA} on input $\langle D, w\rangle$

If M accepts, *accept*. Otherwise, *reject*.

Acceptance problems about automata

$A_{\text{REX}} = \{\langle R, w\rangle : R$ is a regular expression that generates $w\}$

decidable

- The following TM decides A_{REX}:

$P :=$ On input $\langle R, w\rangle$, where R is a reg exp and w is a string

Convert R to an NFA N using the standard conversion procedure.

Run the TM N for A_{NFA} on input $\langle N, w\rangle$

If N accepts, *accept*. Otherwise, *reject*.

Other problems about automata

$MIN_{\text{DFA}} = \{\langle D \rangle : D \text{ is a minimal DFA}\}$

decidable

- The following TM decides MIN_{DFA}:

$R :=$ On input $\langle D \rangle$, where D is a DFA

Run the algorithm for checking distinguishability of states of D.

If every pair of states is distinguishable, *accept*.

Otherwise, *reject*.

Other problems about automata

$EQ_{\text{DFA}} = \{\langle D_1, D_2 \rangle : D_1, D_2 \text{ are DFAs and } L(D_1) = L(D_2)\}$

decidable

- The following TM decides EQ_{DFA}:

$S :=$ On input $\langle D_1, D_2 \rangle$, where D_1 and D_2 are DFAs

Run the DFA minimization algorithm on D_1 to obtain a DFA D_1'

Run the DFA minimization algorithm on D_2 to obtain a DFA D_2'

If $D_1' = D_2'$ *accept*, otherwise *reject*.

Other problems about automata

$E_{\text{DFA}} = \{\langle D \rangle \mid D \text{ is a DFA and } L(D) \text{ is empty}\}$

decidable

- The following TM decides E_{DFA}:

$T :=$ On input $\langle D \rangle$, where D is a DFA

Run the TM S for EQ_{DFA} on input $\langle D,$  $\rangle$

If S accepts *accept*, otherwise *reject*.

Problems about context-free grammars

$A_{\text{CFG}} = \{\langle G, w \rangle : G \text{ is a CFG that generates } w\}$

decidable

$V :=$ On input $\langle G, w \rangle$, where G is a CFG and w is a string

Eliminate the nullable and unit productions from G

Convert G to Chomsky Normal Form

Run the Cocke-Younger-Kasami algorithm on $\langle G, w \rangle$

If the CYK algorithm produces a parse tree, *accept*.

Otherwise, *reject*.

Are all problems about CFGs decidable?

$EQ_{\text{CFG}} = \{\langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are context-free grammars that generate the same strings}\}$

undecidable

What's the difference between EQ_{DFA} and EQ_{CFG}?

To decide EQ_{DFA}, we minimized both DFAs

But there is no method that, given a CFG or PDA, produces a unique equivalent minimal CFG or PDA.

Entscheidungsproblem

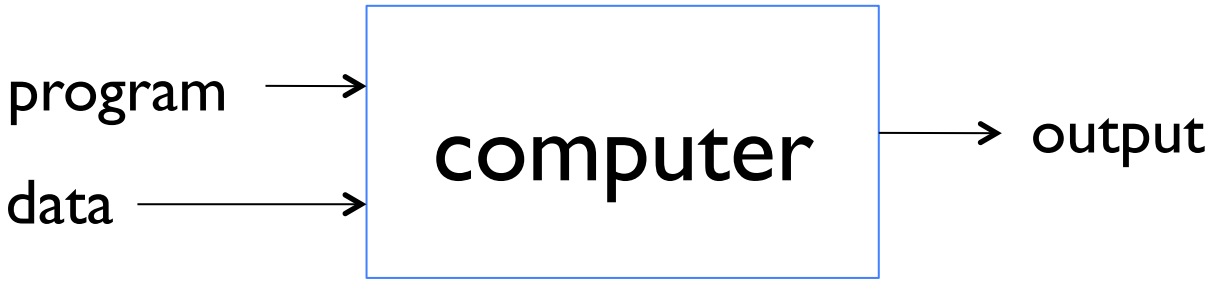
- $V_{\mathrm{F}} = \{\langle F \rangle : F$ is universally valid assuming some axioms$\}$

undecidable

Asks for an algorithm to decide whether a given statement is provable from the axioms using the rules of first-order logic.

The universal Turing Machine and undecidability

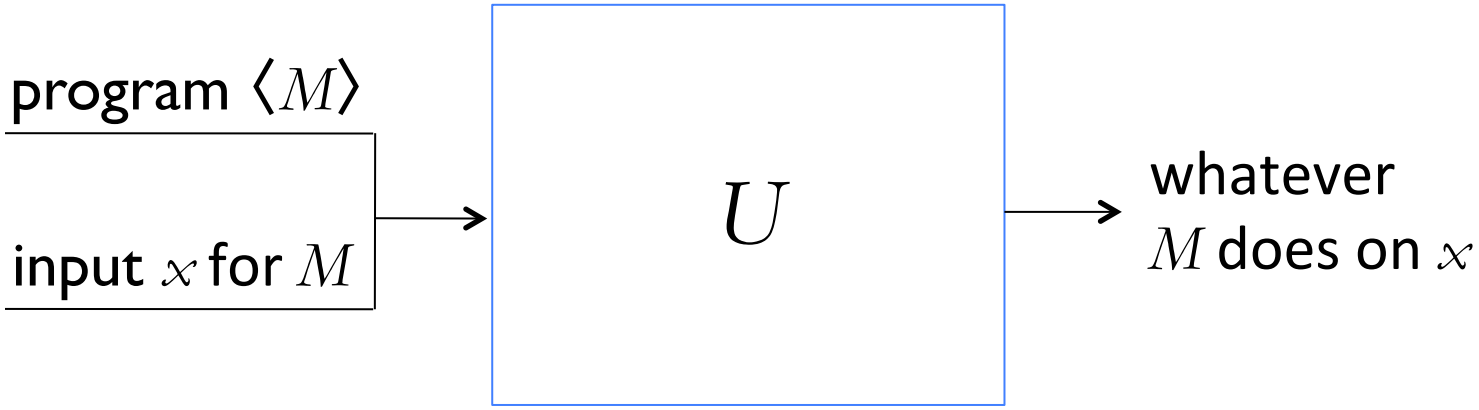
Turing Machines versus computers



A computer is a machine that manipulates data according to a list of instructions.

How does a Turing Machine take a program as part of its input?

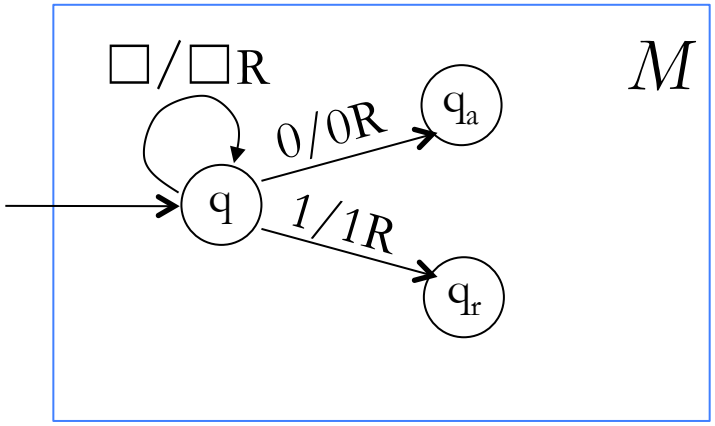
The Universal Turing Machine



The universal TM U takes as inputs a program M and a string x and simulates M on x

The program M itself is specified as a TM!

Turing Machines as strings

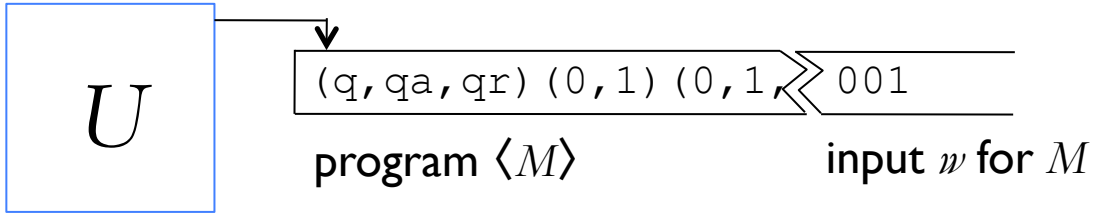


A Turing Machine is
$(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$

- This Turing Machine can be described by the string

$\langle M \rangle$ = `(q,qa,qr)(0,1)(0,1,□)`
`((q,q,□/□R)`
`(q,qa,0/0R)`
`(q,qr,1/1R))`
`(q)(qa)(qr)`

The universal Turing Machine



$U :=$ On input $\langle M, w \rangle$,

Simulate M on input w

If M enters accept state, *accept*.

If M enters reject state, *reject*.

Acceptance of Turing Machines

$A_{\text{TM}} = \{\langle M, w\rangle : M \text{ is a TM that accepts } w\}$

$U :=$ On input $\langle M, w\rangle$,
Simulate M on input w

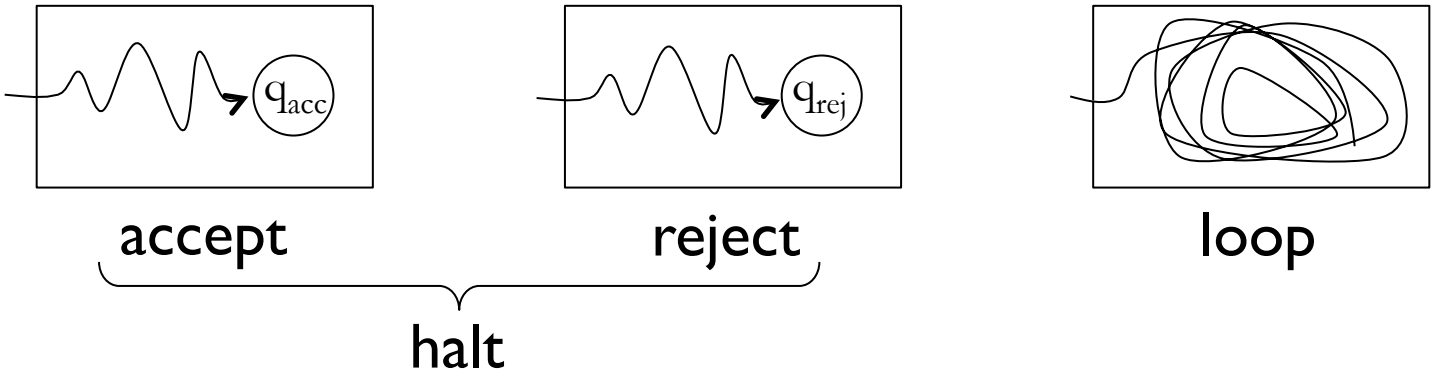
M accepts w ⇩ U accepts $\langle M, w\rangle$

M rejects w ⇩ U rejects $\langle M, w\rangle$

M loops on w ⇩ U loops on $\langle M, w\rangle$

TM U recognises but does not decide A_{TM}

Recognising versus deciding



The language recognised by a TM is the set of all inputs that make it accept.

A TM decides language L if it recognises L and halts (does not loop) on every input.

Undecidability

- Turing's Theorem:

The language A_{TM} is undecidable (semi-decidable).

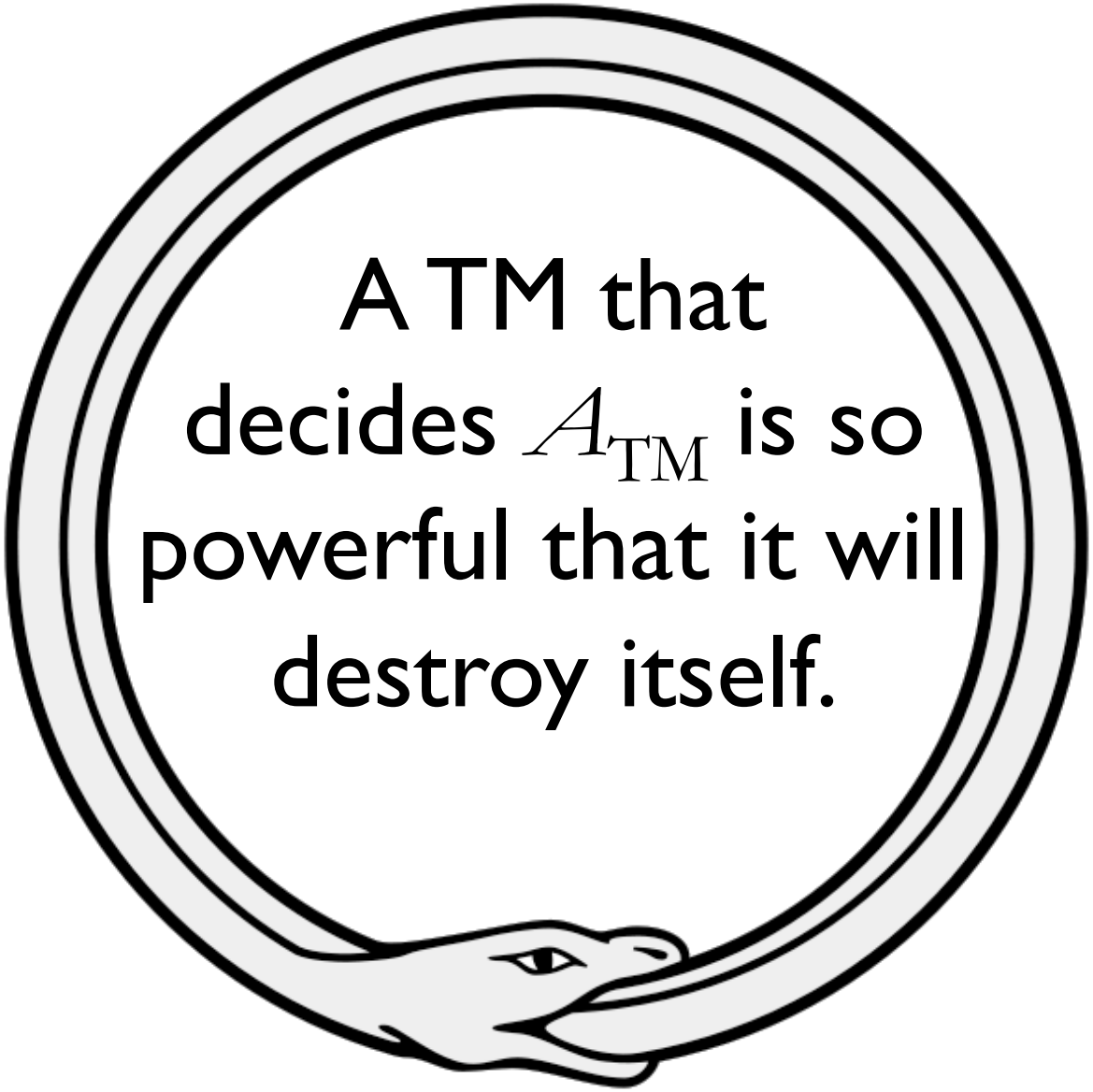
- Before we show this, let's observe one thing:

A Turing Machine M can be given its own description $\langle M \rangle$ as an input!

Terminology

- Recursive language = decidable problem = total TM
- Recursively enumerable language = semi-decidable problem = recognisable language = (undecidable problem)
- Undecidable problem = unrecognisable language

Turing's theorem

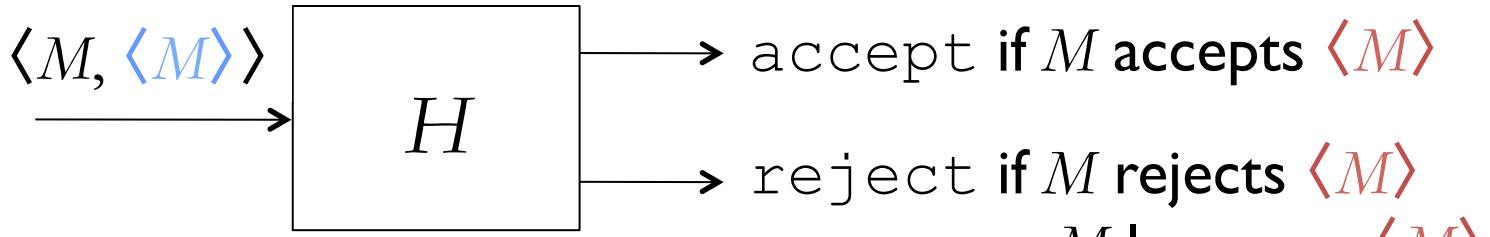


Proof of Turing's Theorem

- Proof by contradiction:

  Suppose A_{TM} is decidable.

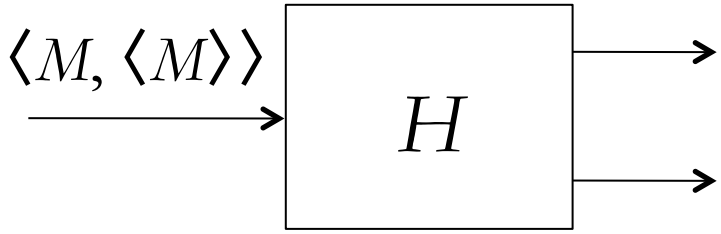
  Then there is a (total) TM H that decides A_{TM}:



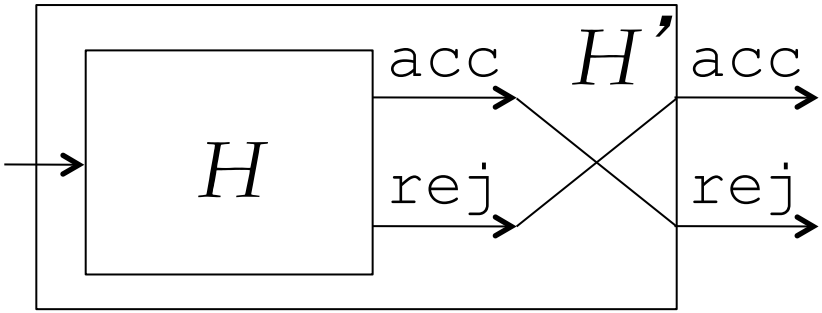
  What happens when $w = \langle M \rangle$?

  H never loops, because it is total

Proof of undecidability

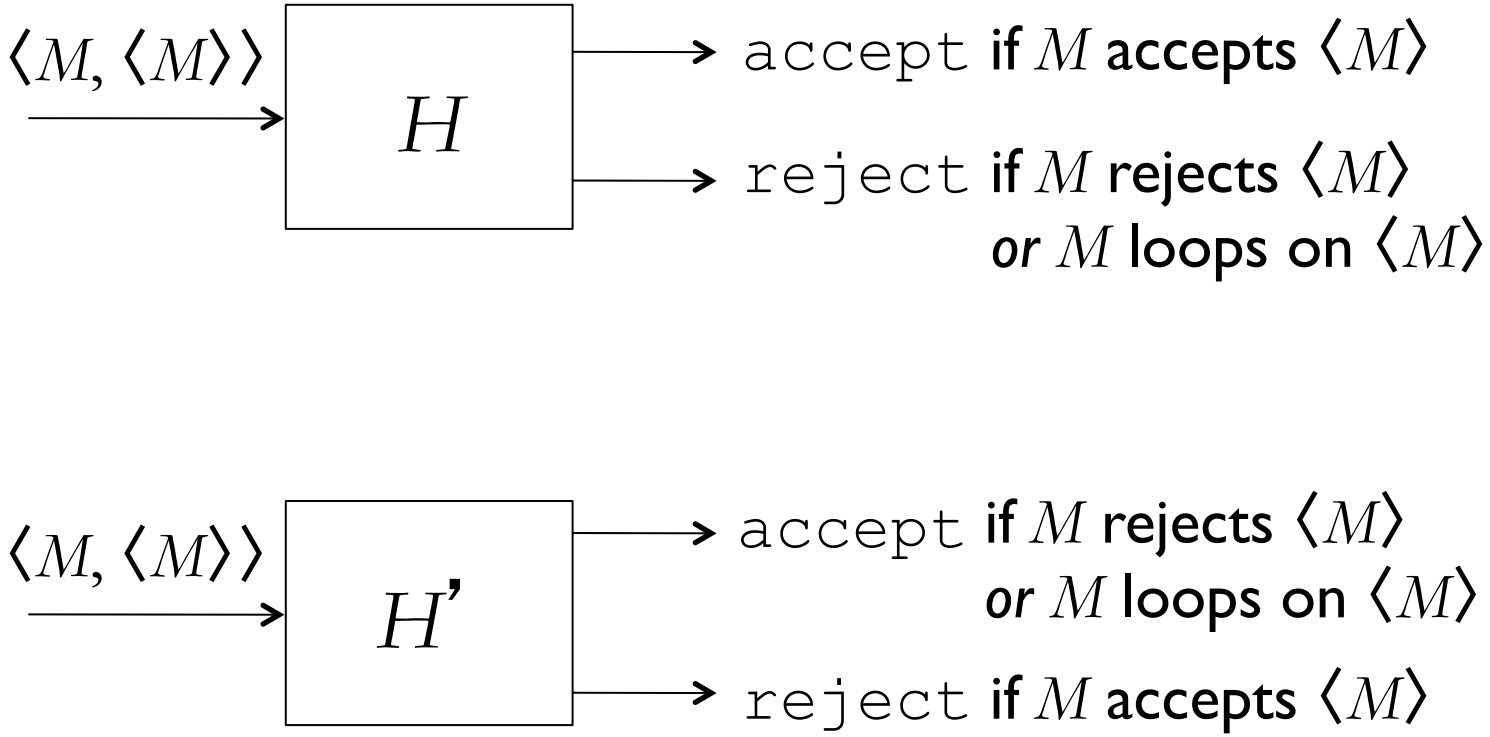


Let H' be a TM that does the opposite of H

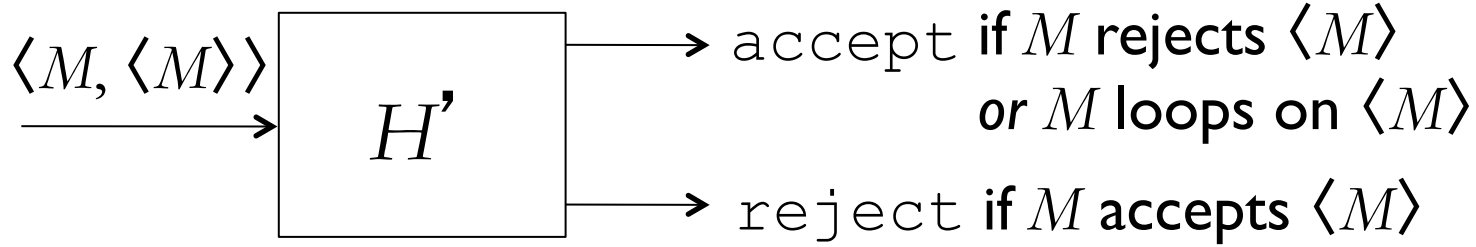


To go from H to H', just switch its *accept* and *reject* states

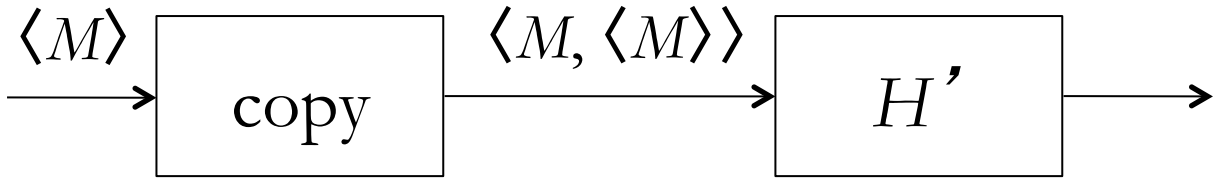
Proof of undecidability



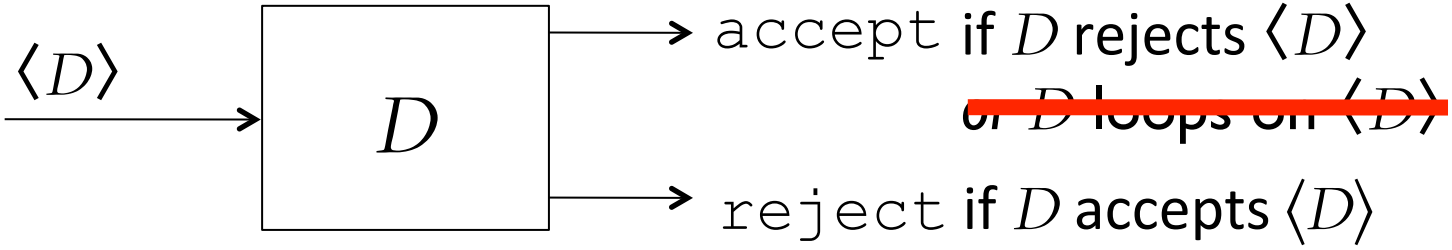
Proof of undecidability



Let D be the following TM:



Proof of undecidability



What happens when $M = D$?

If D accepts $\langle D \rangle$
then D rejects $\langle D \rangle$

If D rejects $\langle D \rangle$
then D accepts $\langle D \rangle$

so D cannot exist!

Proof of undecidability: conclusion

- Proof by contradiction
  - We assumed A_{TM} was decidable
  - Then we built Turing Machines H, H', D
  - But D cannot exist!

- Conclusion

The language A_{TM} is undecidable.

What happened?

all possible inputs w

all possible Turing Machines

	ε	0	1	00	$\cdots$
M_1	acc	rej	rej	acc	
M_2	rej	acc	loop	rej	…
M_3	rej	loop	rej	rej	
$\vdots$		$\vdots$			

We can write an infinite table for every pair (M, w)

What happened?

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	⋯
M_1	`acc`	`loop`	`rej`	`acc`	
M_2	`rej`	`acc`	`rej`	`acc`	⋯
M_3	`loop`	`rej`	`loop`	`rej`	
⋮		⋮			

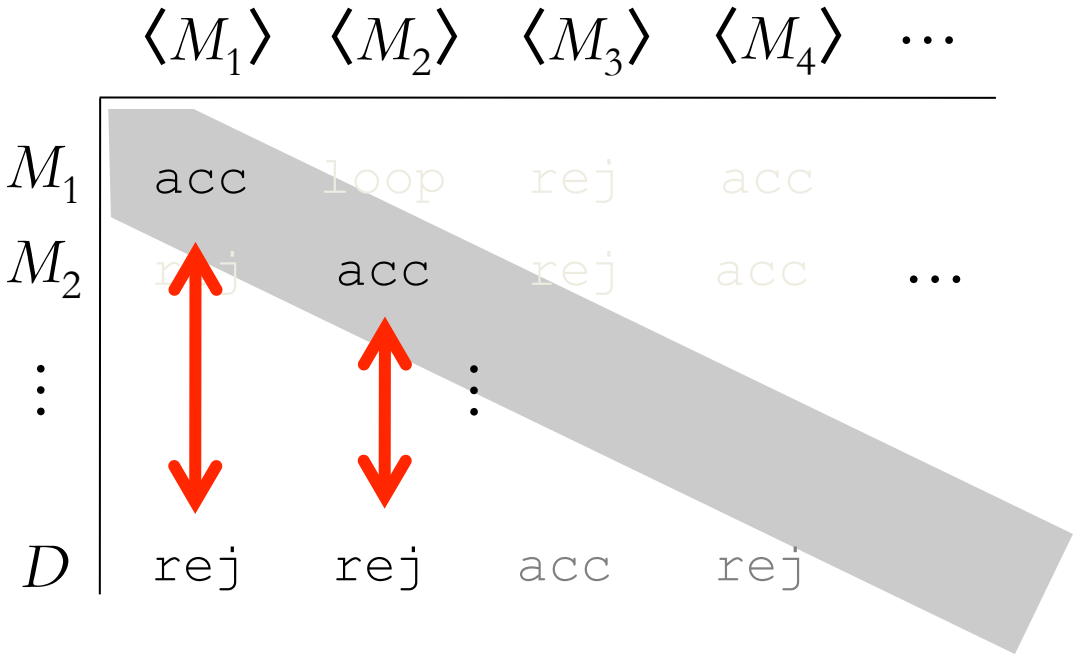
Now let's look only at those w that describe a TM M

What happened?

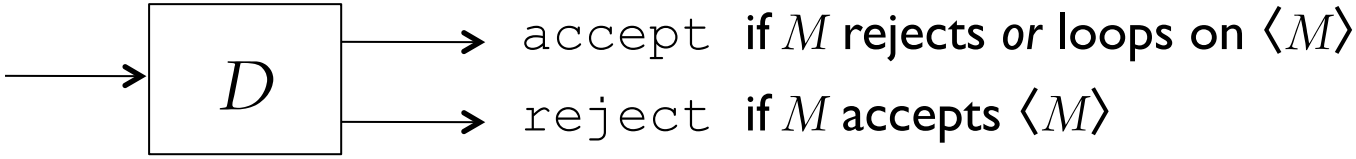
	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	`acc`	`loop`	`rej`	`acc`	
M_2	`rej`	`acc`	`rej`	`acc`	$\cdots$
$\vdots$		$\vdots$			
D	`rej`	`rej`	`acc`	`rej`	
$\vdots$		$\vdots$			

If A_{TM} is decidable, then TM D is in the table

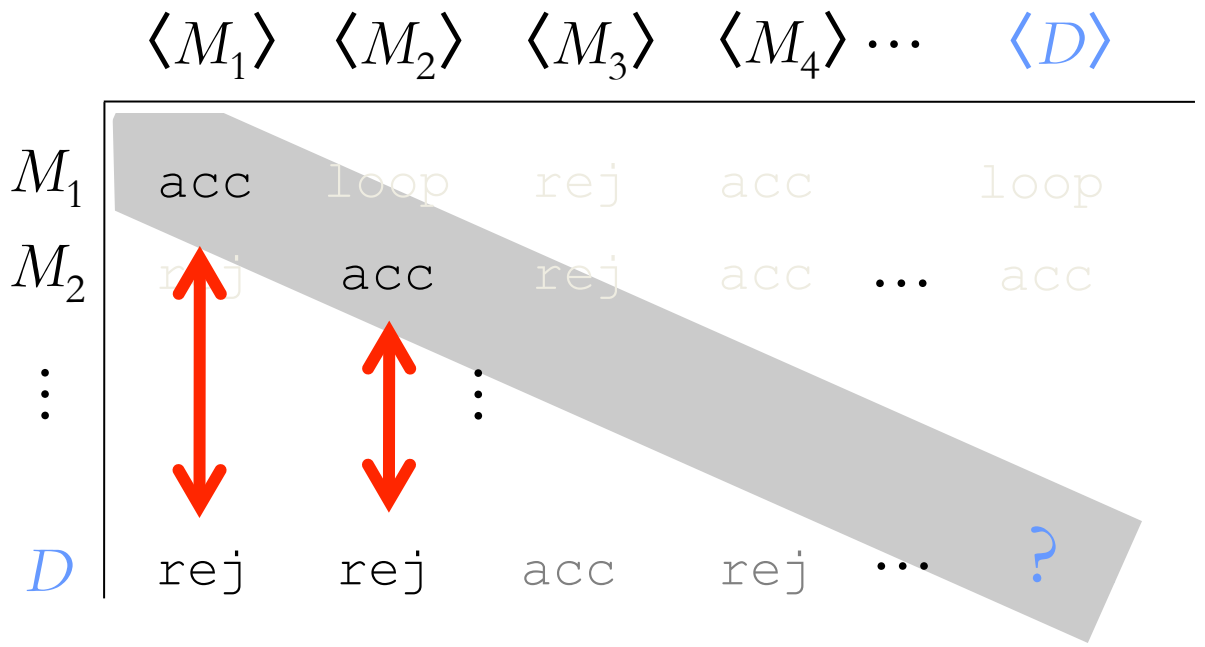
What happened?



D does the opposite of the diagonal entries



What happened?



We run into trouble when we look at $(D, \langle D \rangle)$!

Related to the lair paradox: "This sentence is false."

Unrecognisable languages

The language A_{TM} is recursively enumerable but not recursive.

- How about languages that are not recognisable?

$$\overline{A_{\mathrm{TM}}} = \{\langle M, w\rangle : M \text{ rejects or loops on input } w\}$$
$$\cup \{x : x \text{ does not describe a TM or a valid input}\}$$

The language $\overline{A_{\mathrm{TM}}}$ is not recursively enumerable.

Unrecognisable languages

- Theorem

> If L and $\overline{L}$ are both recursively enumerable, then L is recursive.

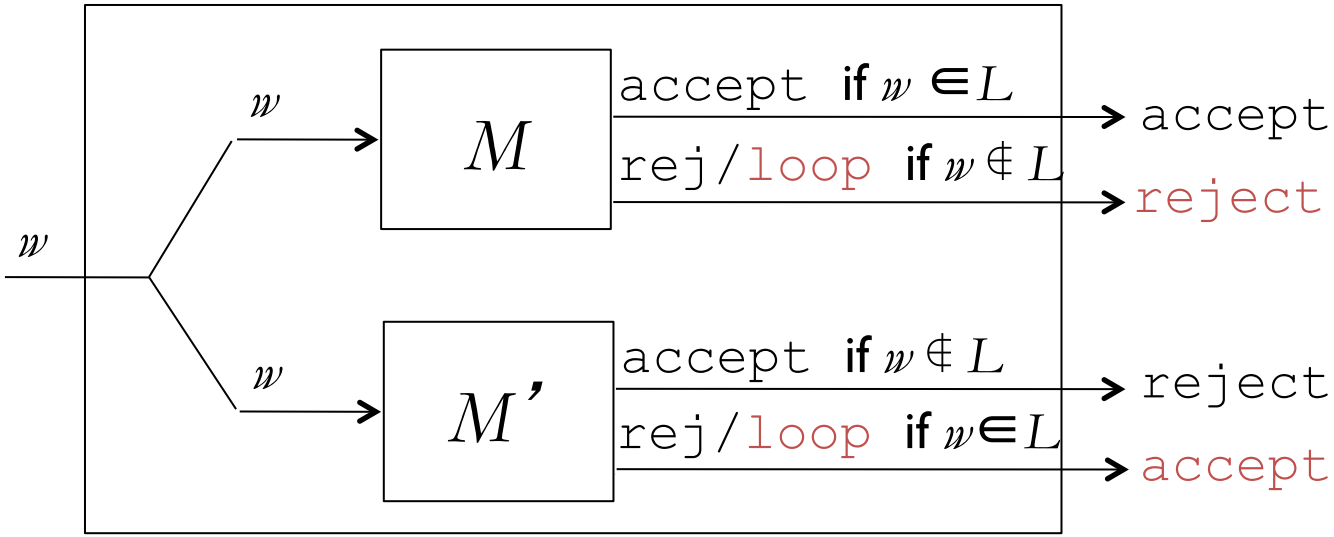
- We know A_{TM} is recursively enumerable, so if $\overline{A_{\mathrm{TM}}}$ was also, then A_{TM} would be recursive.

- But Turing's Theorem says A_{TM} is not recursive.

Unrecognisable languages

If L and $\overline{L}$ are both recursively enumerable, then L is recursive.

- Proof idea



Unrecognisable languages

If L and $\overline{L}$ are both recursively enumerable, then L is recursive.

Let M = TM

On input w,

① Simulate M on input w. If it accepts, *accept*.

② Simulate M' on input w. If it accepts, *reject*.

Problem: If M loops on w, we will never get to step 2!

Bounded simulation

If L and $\overline{L}$ are both recursively enumerable, then L is recursive.

- Turing Machine that decides L:

  Let M = TM for L, M' = TM for $\overline{L}$

  On input w,

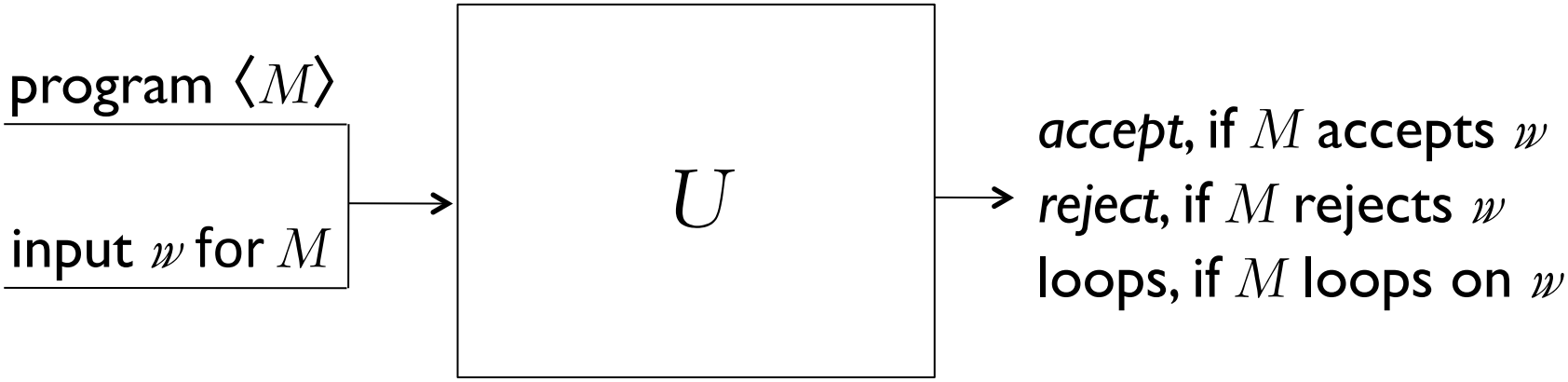
  For $t := 0$ to infinity

    Do t transitions of M on w. If it accepts, *accept*.

    Do t transitions of M' on w. If it accepts, *reject*.

Reducibility

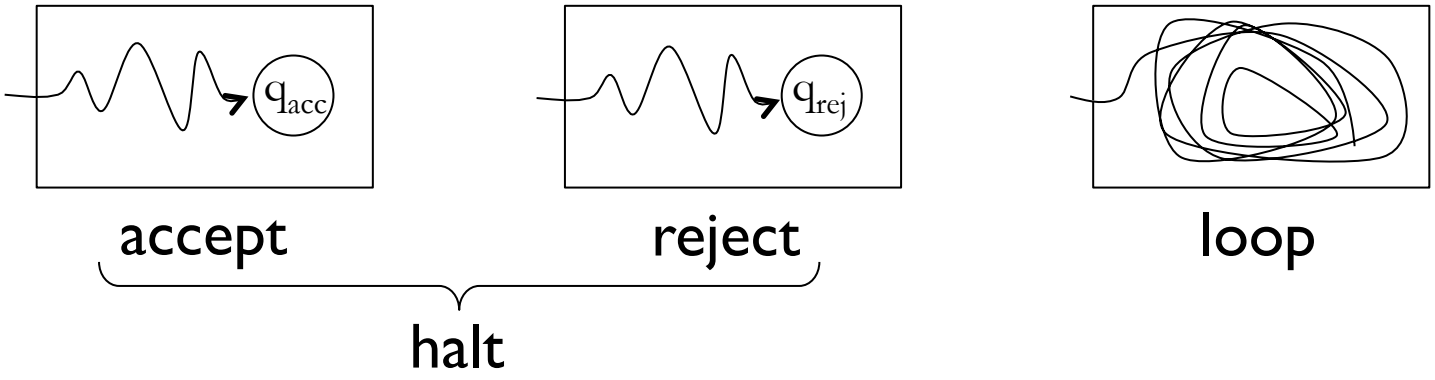
Summary of last lecture



The universal TM U takes as inputs a program M and a string x and simulates M on x

The program M itself is specified as a TM!

Summary of last lecture



$A_{\mathrm{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts input } w\}$

recursively enumerable:	By universal TM U
undecidable (i.e. not recursive):	Turing's Theorem

Another undecidable language

$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$

$HALT_{\text{TM}}$ is an undecidable language.

- We will argue that

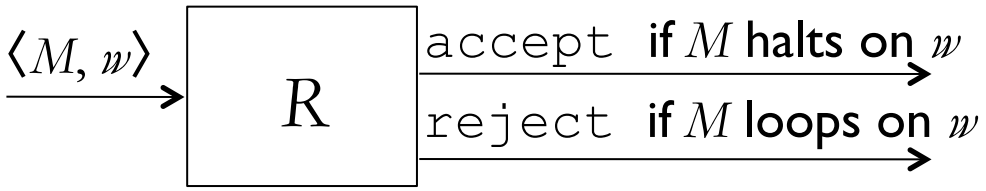
If $HALT_{\text{TM}}$ is recursive, so is A_{TM}.

… but by Turing's Theorem it is not.

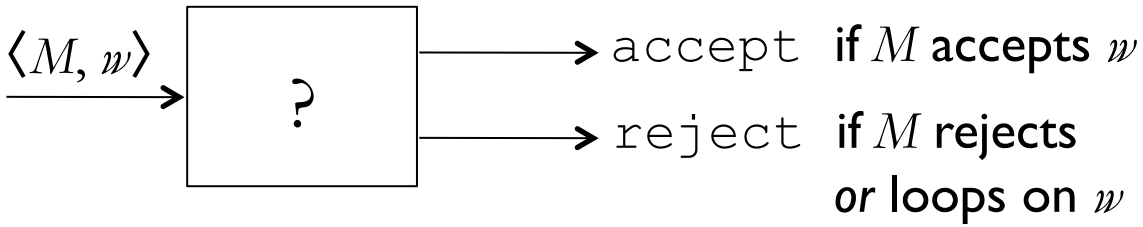
Undecidability of halting

If $HALT_{\text{TM}}$ is recursive, so is A_{TM}.

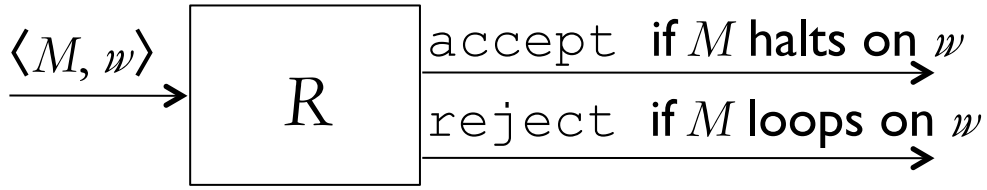
- Suppose R decides

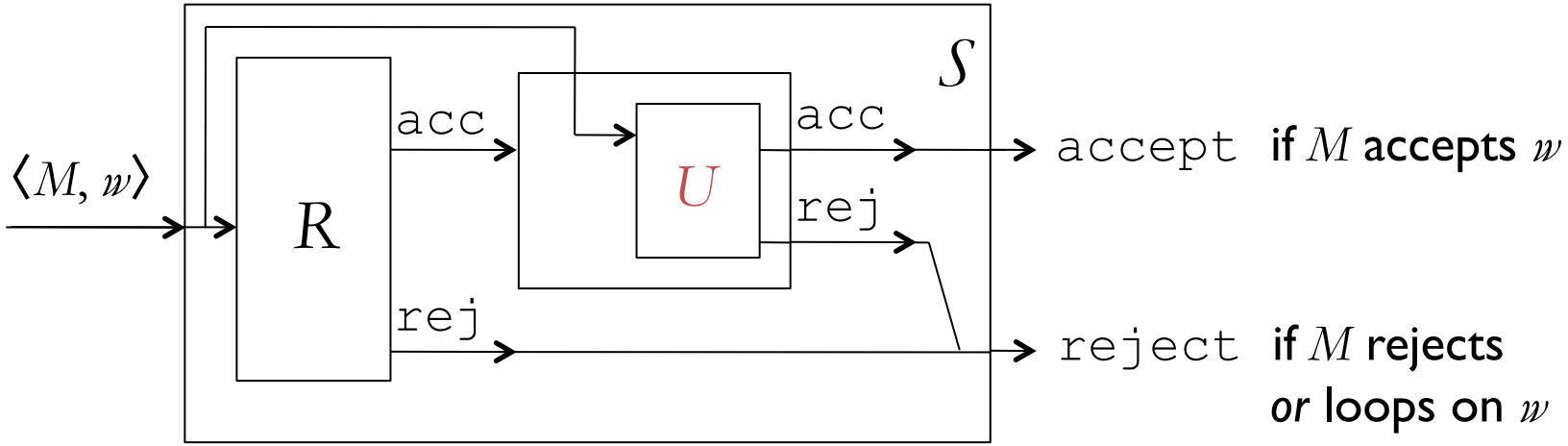


- We want a TM S that decides



Undecidability of halting





Undecidability of halting

$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$

$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts input } w\}$

Suppose that $HALT_{\text{TM}}$ is recursive. Let H be a TM that decides $HALT_{\text{TM}}$. Then this TM decides A_{TM}:

$S :=$ On input $\langle M, w\rangle$,
Run R on input $\langle M, w\rangle$.
If R rejects, *reject*.
If R accepts, run U on input $\langle M, w\rangle$
If U accepts, *accept*. If U rejects, *reject*.

Impossible, because A_{TM} is not recursive.

Reducibility

- Suppose you think L is not recursive.

- How do you know for sure?

Show: If some TM R decides L

then using R, we can build another TM S

so that S decides A_{TM}

...but this is impossible, because A_{TM} is not recursive.

Example 1

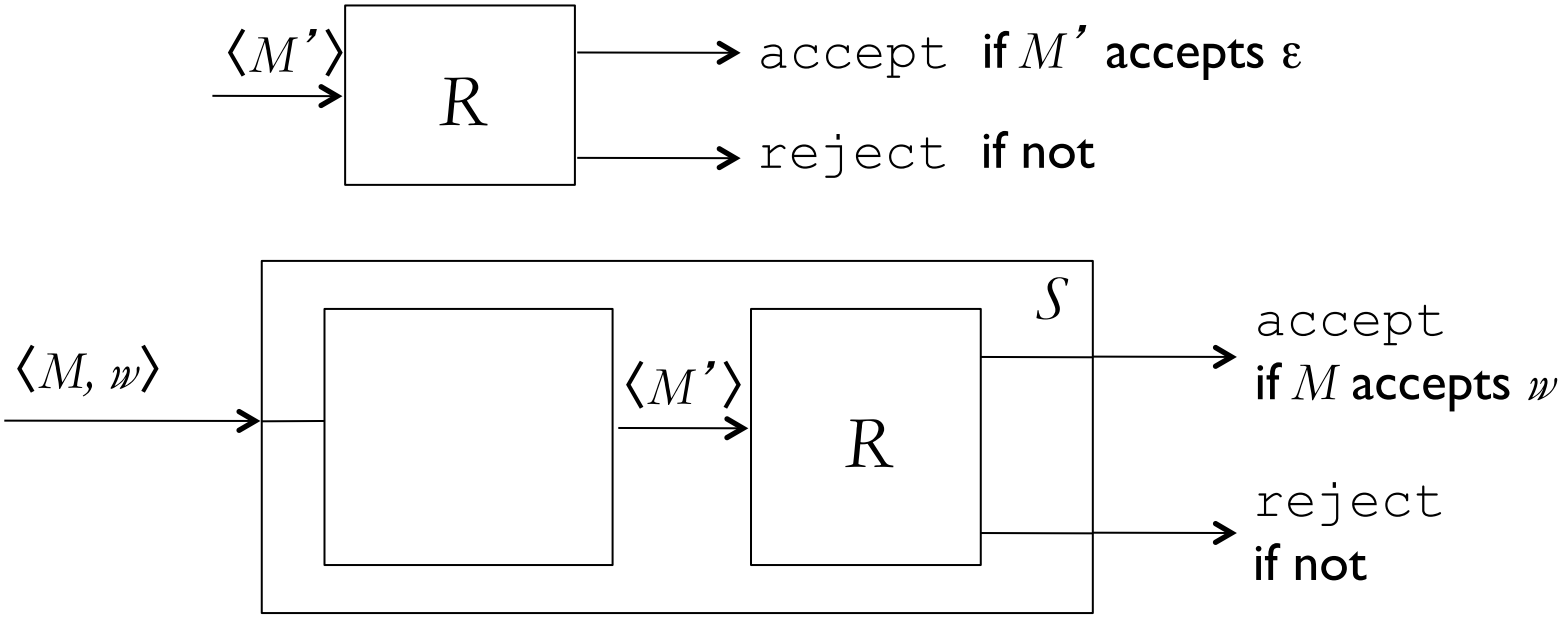
$$AEPS_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts input } \varepsilon\}$$

recursive

undecidable

- Step 1: Take an educated guess
  - To know if M accepts ε, it looks like we have to simulate it
  - But then we might end up in a loop
- Step 2: Prove it!

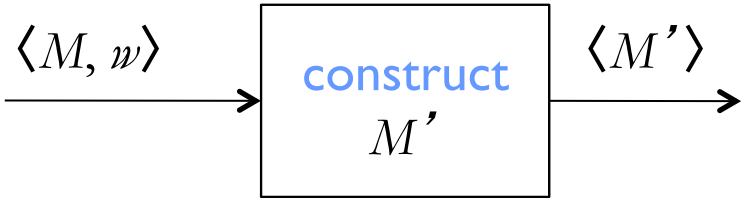
Example 1: Figuring out the reduction



M' should be a Turing Machine such that:

M' on input ε = M on input w

Example 1: Implementing the reduction

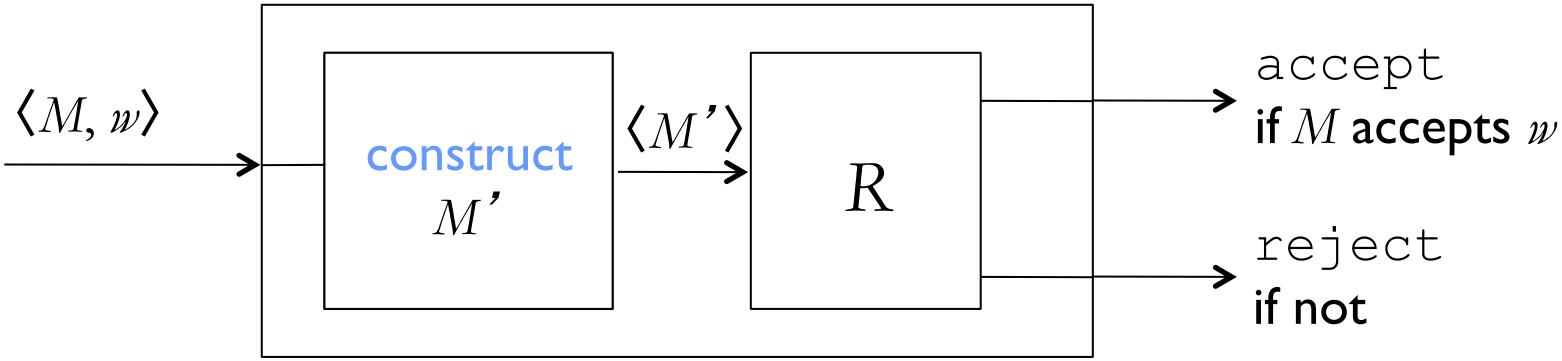


M' should be a Turing Machine such that:

M' on input ε = M on input w

M' := On input z *(ignored)*,
Simulate M on input w
If M accepts w, *accept*
Otherwise, *reject*

Example 1: Impementing the reduction



S := On input $\langle M, w \rangle$:

Construct the following TM M':

> M' := On input z,
> Simulate M on input w and output answer

Run R on input $\langle M' \rangle$ and output its answer.

Example 1: Writing it up

undecidable

$AEPS_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts input } \varepsilon\}$

$A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM that accepts input } w\}$

Suppose $AEPS_{\text{TM}}$ is recursive and R decides it. Let

$S :=$ On input $\langle M, w \rangle$ where M is a TM and w is a string

Construct the following TM M':

$M' :=$ On input z,
Simulate M on input w and output answer

Run R on input $\langle M' \rangle$ and output its answer.

Then S accepts $\langle M, w \rangle$ if and only if M accepts w

So S decides A_{TM}, which is impossible.

Example 2

$SOME_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts some input}\}$

recursive

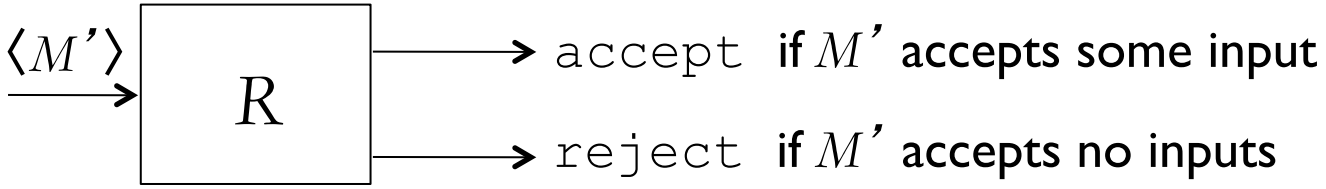
undecidable

- Step 1: Take an educated guess
  - To know if M is in $SOME_{\text{TM}}$, looks like we have to simulate
  - But then we might end up in a loop
- Step 2: Prove it!

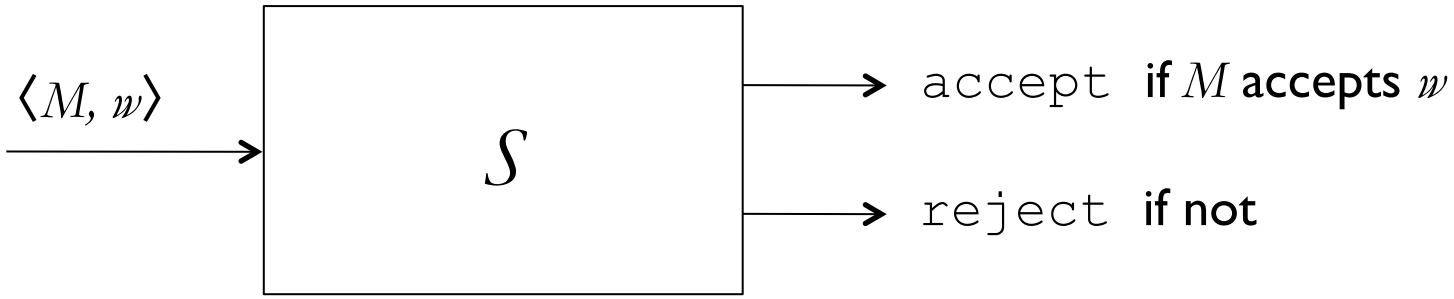
Example 2: Setting up the reduction

$SOME_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts some input}\}$

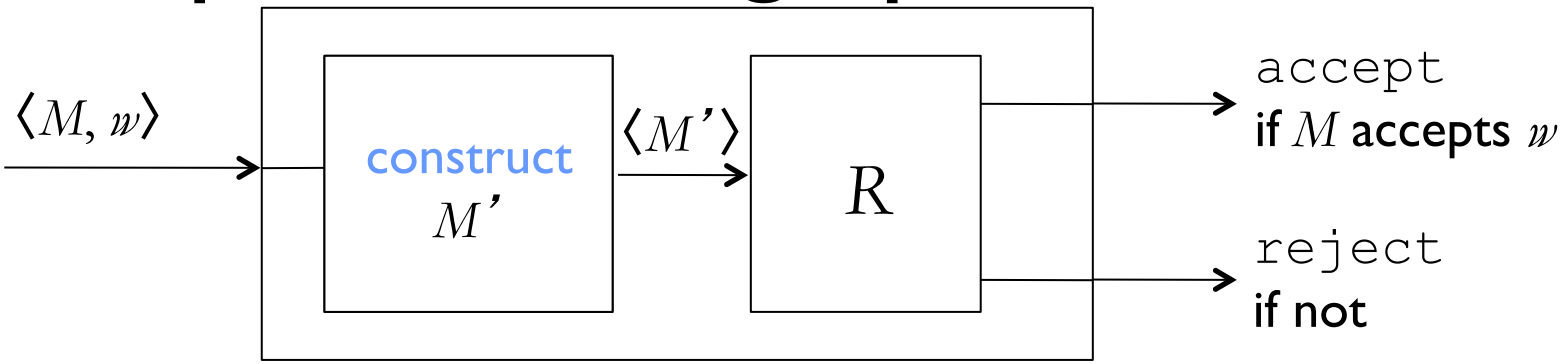
Show: If $SOME_{\text{TM}}$ can be decided by some TM R,



... then A_{TM} can be decided by some other TM S



Example 2: Setting up the reduction



Task Given $\langle M, w \rangle$, construct M' so that:

If M accepts w, then M' accepts some input

If M does not accept w, then M' accepts no inputs

Example 2: Implementing the reduction

Task Given $\langle M, w \rangle$, construct M' so that:

If M accepts w, then M' accepts some input
If M does not accept w, then M' accepts no inputs

M' := On input z,
Simulate M on input w
If M accepts, *accept*
Otherwise, *reject*

Example 2: Writing it up

undecidable

$SOME_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts some input}\}$

$A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM that accepts input } w\}$

Suppose $SOME_{\text{TM}}$ is recursive and R decides it. Let

$S :=$ On input $\langle M, w \rangle$ where M is a TM and w is a string

Construct the following TM M':

> $M' :=$ On input z,
> Simulate M on input w and output answer

Run R on input $\langle M' \rangle$ and output its answer.

Then S accepts $\langle M, w \rangle$ if and only if M accepts w

So S decides A_{TM}, which is impossible.

Example 3

$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts no input}\}$

recursive

undecidable

Show: If E_{TM} can be decided by some TM R,

... then $SOME_{\text{TM}}$ can be decided by another TM S

$SOME_{\text{TM}} = \{\langle M \rangle : M \text{ is a TM that accepts some input}\}$

Example 3

undecidable

$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts no input}\}$

$SOME_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts some input}\}$

Suppose E_{TM} can be decided by some TM R.

Consider the following TM S:

> $S :=$ On input $\langle M \rangle$ where M is a TM
>
> Run R on input $\langle M \rangle$.
>
> If R accepts, *reject*. If R rejects, *accept*.

Then S decides $SOME_{\text{TM}}$, a contradiction.

Example 4

$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ accept the same inputs}\}$

recursive

undecidable

Show: If EQ_{TM} can be decided by some TM R,

... then E_{TM} can be decided by another TM S

Example 4: Setting up the reduction

$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ accept the same inputs}\}$

$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts no input}\}$

Task Given $\langle M \rangle$, construct $\langle M_1, M_2 \rangle$ so that:

If M accepts no input, then M_1, M_2 accept same inputs

If M accepts some input, then M_1, M_2 do not accept same inputs

Idea Make $M_1 = M$

Make M_2 accept nothing

Example 4: Writing it up

undecidable

$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ accept the same inputs}\}$

$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM that accepts no input}\}$

Suppose EQ_{TM} is recursive and R decides it. Let

$S :=$ On input $\langle M \rangle$ where M is a TM

Construct the following TM M_2:

$M_2 :=$ On input x, *reject*.

Run R on input $\langle M, M_2 \rangle$ and output its answer.

Then S accepts $\langle M \rangle$ if and only if M accepts no input.

So S decides E_{TM}, which is impossible.

Undecidable problems for CFGs

Decidable vs. undecidable

decidable	undecidable
"DFA M accepts w"	"TM M accepts w"
"CFG G generates w"	"TM M halts on w"
"DFAs M and M' accept same inputs"	"TM M accepts some input"
	"TM M and M' accept same inputs"

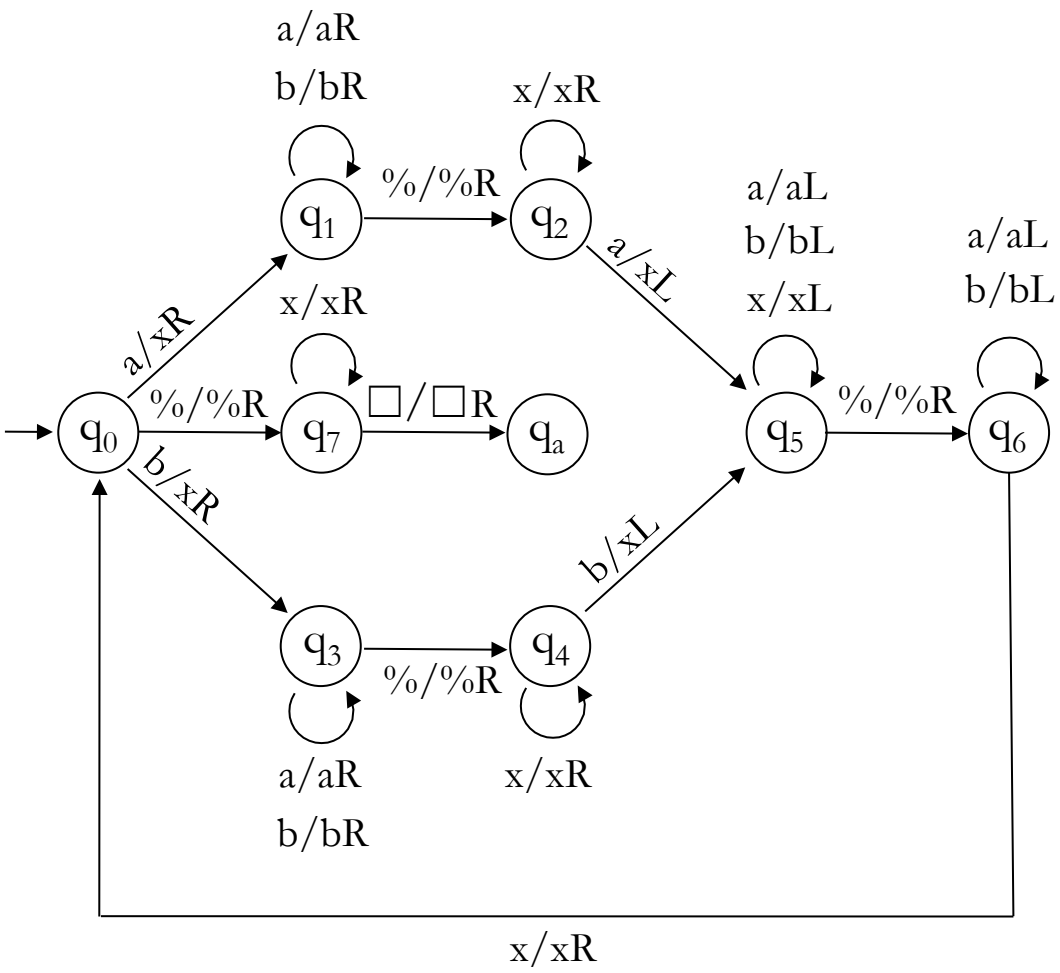
"CFG G generates all in

"CFG G is ambiguous"

more?

Representing computations

$L_1 = \{w\%w : w \in \{a, b\}^*\}$



q_0 <u>a</u>bbaa%abbaa

q_1 x<u>b</u>baa%abbaa

⋮

q_1 xbbaa<u>%</u>abbaa

q_2 xbbaa%<u>a</u>bbaa

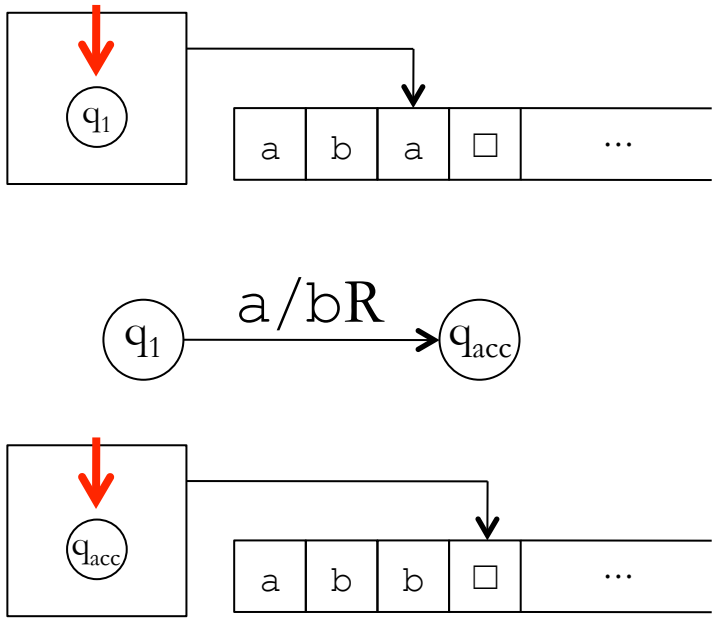
q_5 xbbaa<u>%</u>xbbaa

⋮

q_a xxxxx%xxxx<u>x</u>

Configurations

- A configuration consists of the current state, the head position, and tape contents

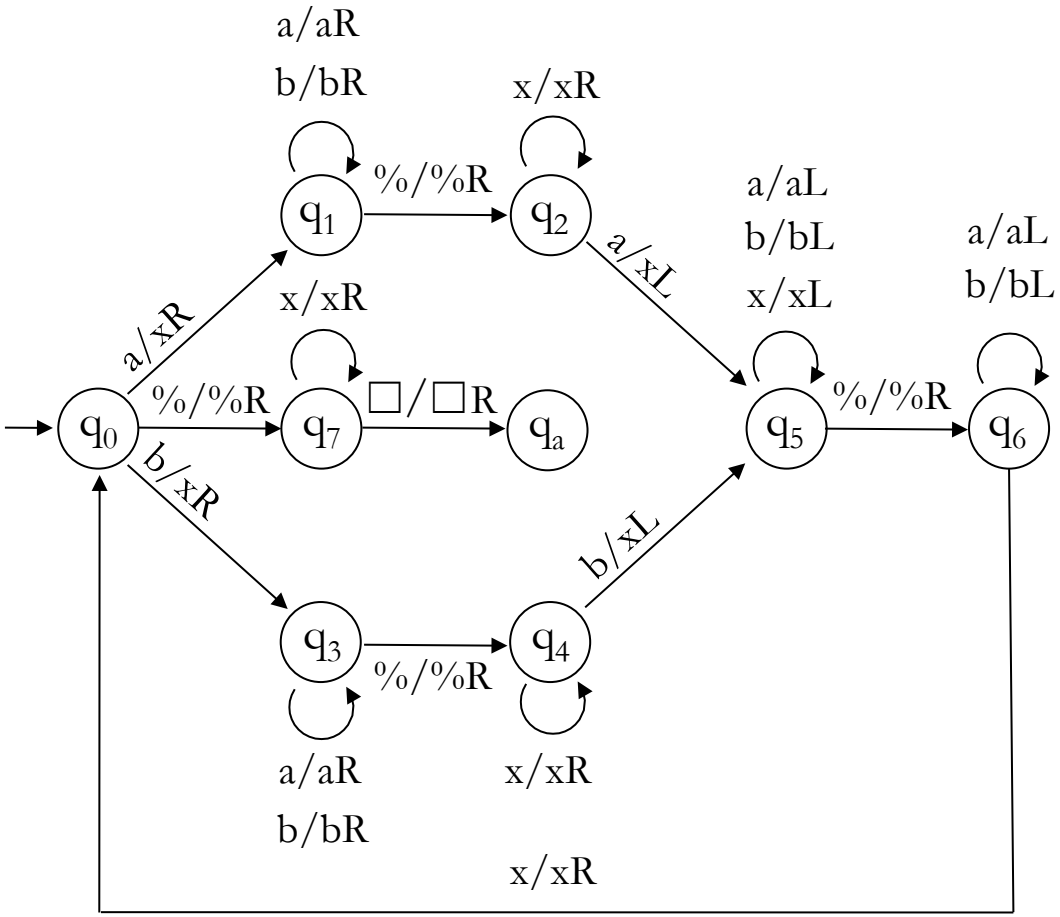


configuration

abq_1a

$abbq_{acc}$

Computation histories



q_0a b % a b
x q_1b % a b
x b q_1% a b
x b % q_2a b
x b q_5% x b
x q_6b % x b
⋮
x x % x x q_7
x x % x x □ q_a

computation history

Computation histories as strings

If M halts on w, the computation history of (M, w) is the sequence of configurations $C_1, ..., C_l$ that M goes through on input w.

q_0a b % a b
x q_1b % a b
⋮
x x % x x q_7
x x % x x q_a

$\underbrace{q_0ab\%ab}\#\underbrace{xq_1b\%ab}\#\ \ldots\ \#\underbrace{xx\%xx\square q_a}$

The computation history can be written as a string *hist* over alphabet $\Gamma \cup Q \cup \{\#\}$

accepting history: M accepts w ⟺ q_{acc} occurs in *hist*

rejecting history: M rejects w ⟺ q_{rej} occurs in *hist*

Undecidable problems for CFGs

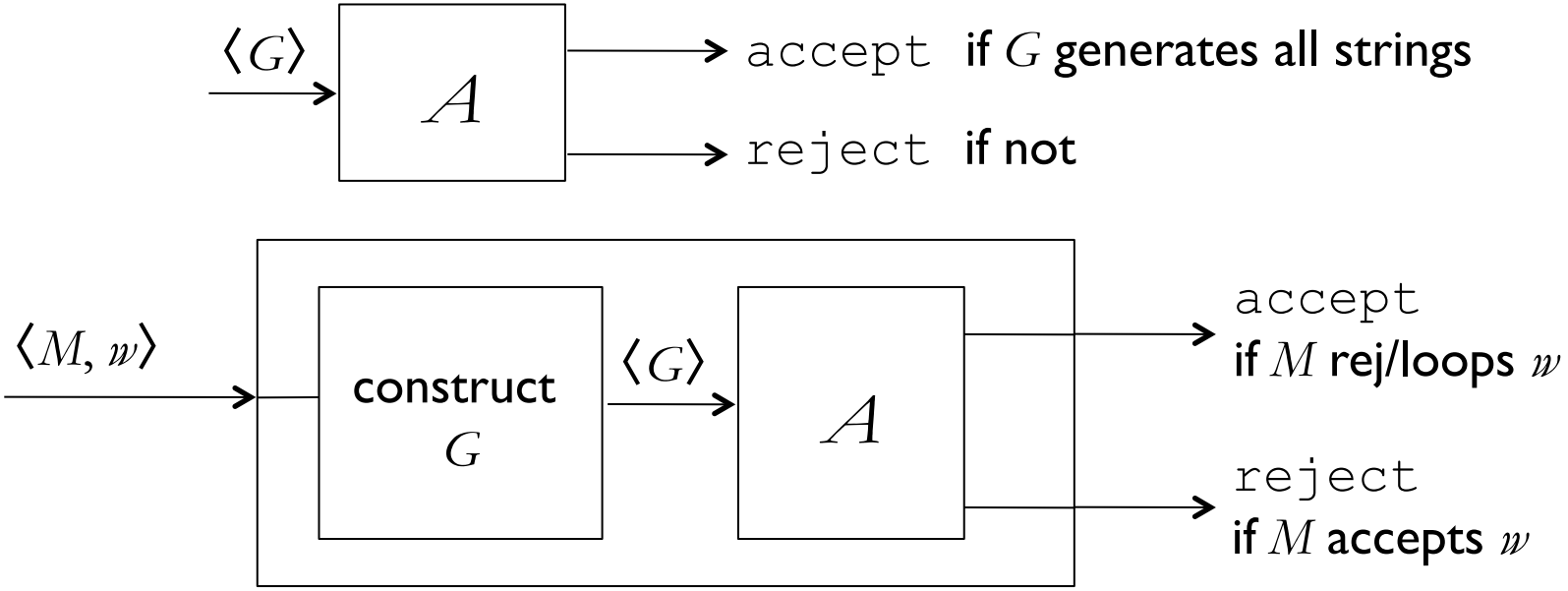
$$ALL_{\text{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG that generates all strings}\}$$

The language ALL_{CFG} is undecidable (and not even r.e.).

- We will argue that

If ALL_{CFG} is recursive, then so is $\overline{A_{\text{TM}}}$.

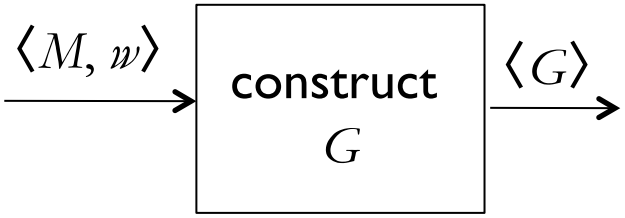
Undecidable problems for CFGs



G generates all strings if M rejects or loops on w

G fails to generate some string if M accepts w

Undecidable problems for CFGs



G fails to generate some string

⇕

M accepts w

The alphabet of G will be $\Gamma \cup Q \cup \{\#\}$

G will generate all strings except the computation history of (M, w), if it is accepting

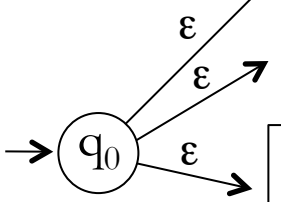
First we construct a PDA P, then convert it to CFG G

Undecidability via computation histories

candidate computation history *hist* of (M, w) → P → accept everything except accepting *hist*

$\#q_0ab\%ab\#xq_1b\%ab\#\ldots\#xx\%xx\square q_a\#$ ⇨ reject

P: On input *hist*, (try to spot a mistake in *hist*)



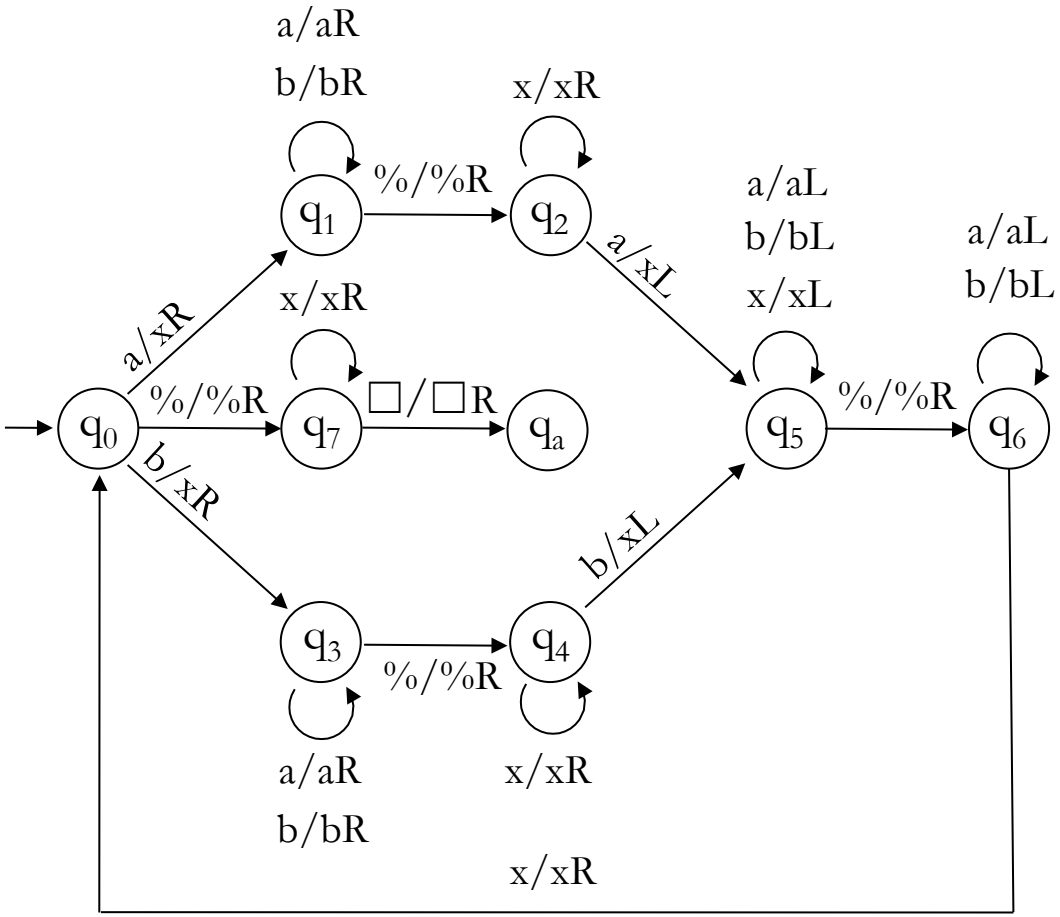
If *hist* is not of the form $\#w_1\#w_2\#\ldots\#w_k\#$, accept.

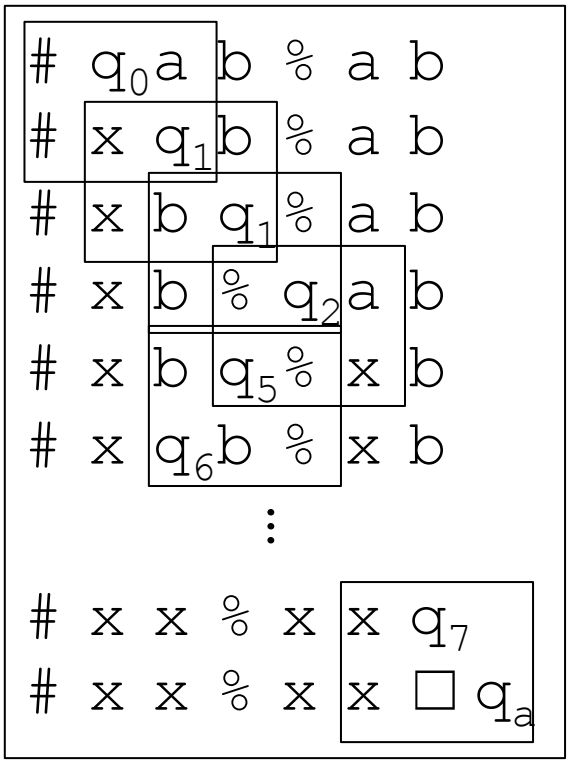
If $w_1 \neq q_0w$ or w_k does not contain q_a, accept.

If two consecutive blocks $w_i\#w_{i+1}$ do not follow from the transitions of M, accept.

Otherwise, *hist* is an accepting history, so reject.

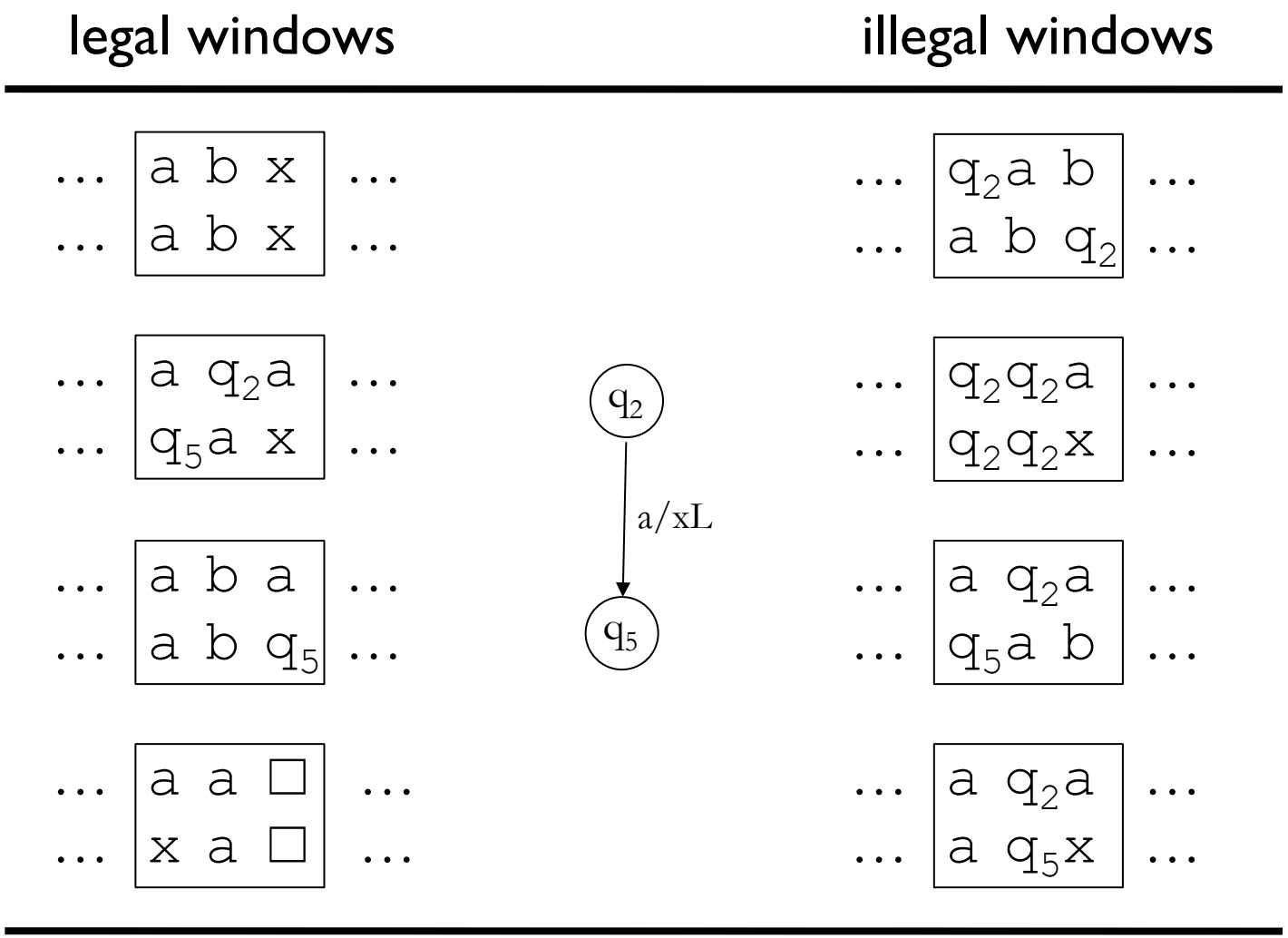
Computation is local





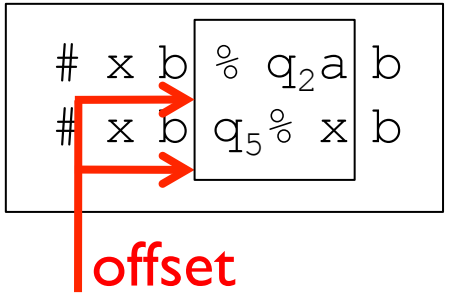
Changes between configurations always occur around the head

Legal and illegal transition windows



Implementing P

If two consecutive blocks $w_i \# w_{i+1}$ do not follow from the transitions of M, accept:



For every position of w_i:

Remember offset from # in w_i on stack

Remember first row of window in state

After reaching the next #:

Pop offset from # from stack as you consume input

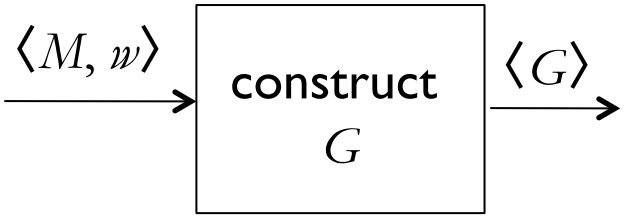
Remember second row of window in state

If window is illegal, accept;
Otherwise reject.

The computation history method

$ALL_{\text{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG that generates all strings}\}$

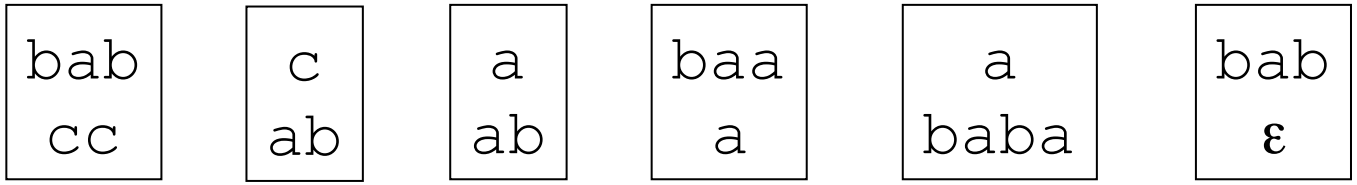
If ALL_{CFG} is recursive, then so is $\overline{A_{\text{TM}}}$.



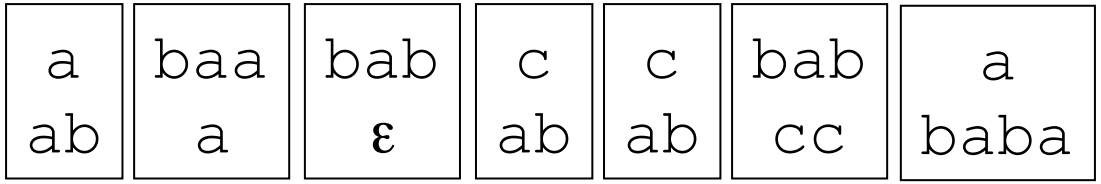
- G accepts all strings except accepting computation histories of (M, w)
- We first construct a PDA P, then convert it to CFG G.

The Post Correspondence Problem

- Input: A set of tiles like this



- Given an infinite supply of such tiles, can you match top and bottom?



Undecidability of PCP

$PCP = \{\langle T \rangle \mid T$ is a collection of tiles that contains a top-bottom match$\}$

The language PCP is undecidable.

Ambiguity of CFGs

$$AMB = \{G \mid G \text{ is an ambiguous CFG}\}$$

The language AMB is undecidable.

- We will argue that

If AMB is recursive, then so is PCP.

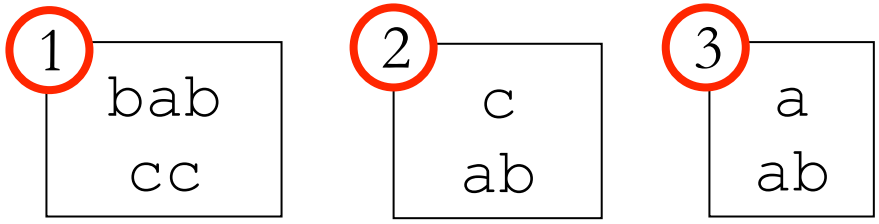
Ambiguity of CFGs

T (collection of tiles) $\longrightarrow$ G (CFG)

If T can be matched, then G is ambiguous

If T cannot be matched, then G is unambiguous

- Step 1: Number the tiles



Ambiguity of CFGs

T (collection of tiles) $\longrightarrow$ G (CFG)

Terminals: a, b, c, 1, 2, 3

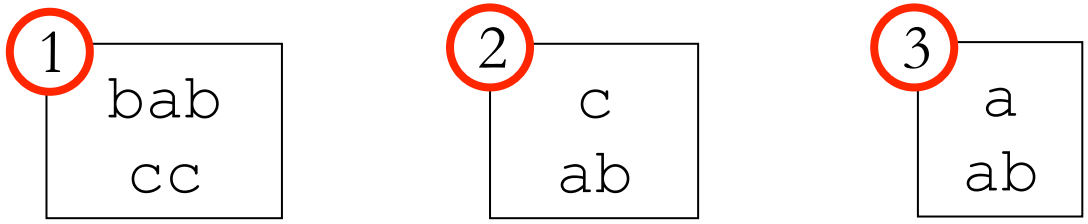
Variables: S, T, B

Productions:

T → babT1 | T → cT2 | T → aT3

B → ccB1 | B → abB2 | B → abB3

S → T | B



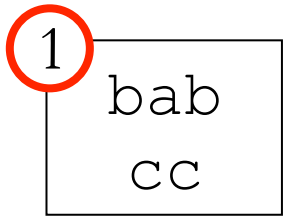
Ambiguity of CFGs

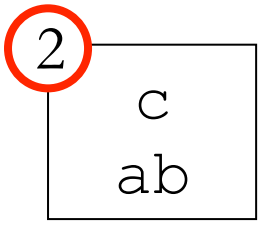
T (collection of tiles) $\longrightarrow$ G (CFG)

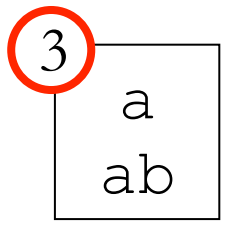
Terminals: a, b, c, 1, 2, 3

Variables: S, T, B

Productions:







S → T | B

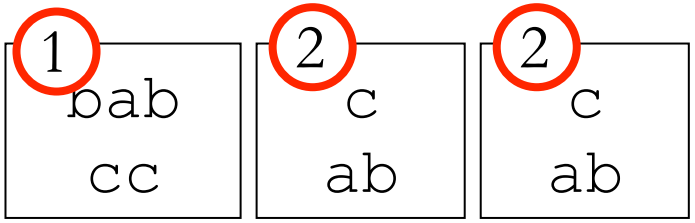
T → babT1
T → bab1
B → ccB1
B → cc1

T → cT2
T → c2
B → abB2
B → ab2

T → aT3
T → a3
B → abB3
B → ab3

Ambiguity of CFGs

- Each sequence of tiles gives two derivations



$S \Rightarrow T \Rightarrow \texttt{babT1} \Rightarrow \texttt{babcT21} \Rightarrow \texttt{babcc221}$

$S \Rightarrow B \Rightarrow \texttt{ccB1} \Rightarrow \texttt{ccabB21} \Rightarrow \texttt{ccabab221}$

- If the tiles match, these two derive the same string

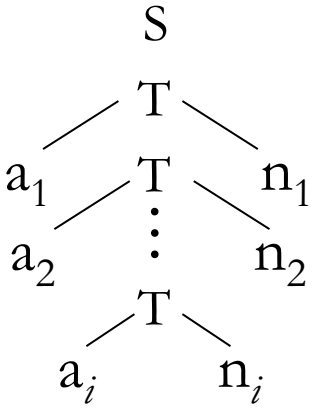
Ambiguity of CFGs

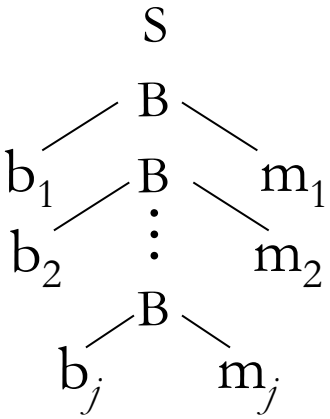
T (collection of tiles) $\longrightarrow$ G (CFG)

If T can be matched, then G is ambiguous ✓

If T cannot be matched, then G is unambiguous ✓

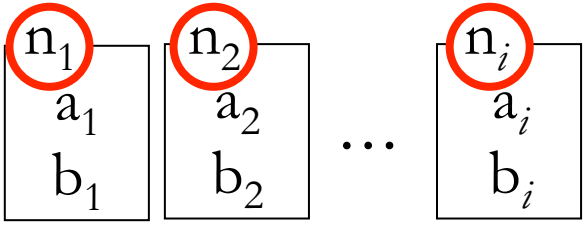
- If G is ambiguous then ambiguity must look like this





Then $n_1 \ldots n_i = m_1 \ldots m_j$

So there is a match



Undecidability of PCP

(optional)

Undecidability of PCP

$PCP = \{\langle T \rangle : T$ is a collection of tiles that contains a top-bottom match$\}$

The language PCP is undecidable.

- We show that

If PCP can be decided, so can A_{TM}.

Undecidability of PCP

$\langle M, w \rangle$ ⟶ T (collection of tiles)

M accepts w ⟺ T contains a match

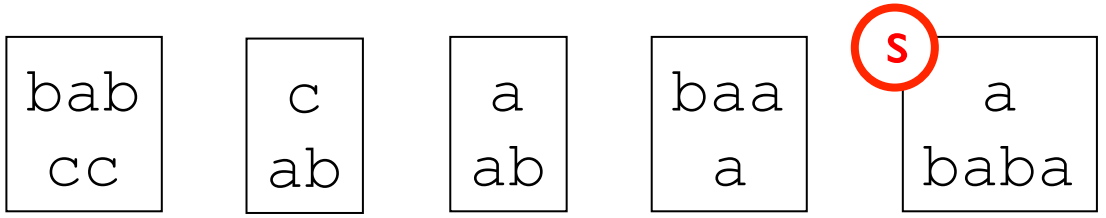
- Idea: Matches represent accepting histories

$\#q_0ab\%ab\#xq_1b\%ab\#...\#xx\%xx\square q_a\#$
$\#q_0ab\%ab\#xq_1b\%ab\#...\#xx\%xx\square q_a\#$

ε $\#q_0ab\%ab$	$\#q_0a$ $\#xq_1$	b b	a a	% %	a a	b b	# #	$xq_1\%$ $x\%q_2$	...

An assumption

- We will assume that one of the PCP tiles is marked as a starting tile



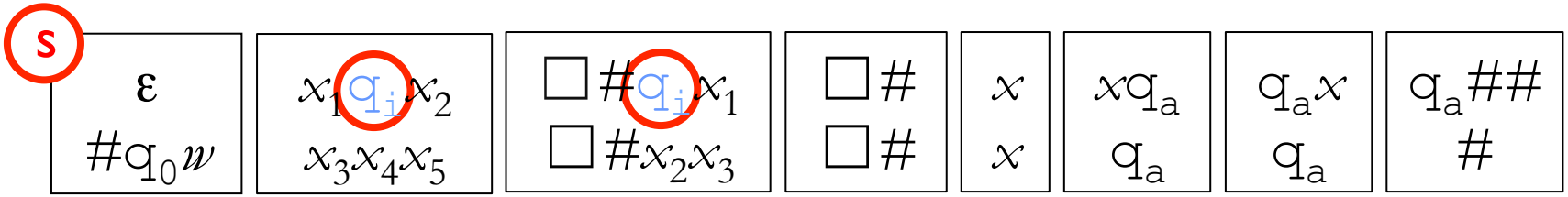
- Later we' ll see how to "simulate" the starting tile by an ordinary tile

Undecidability of PCP

$\langle M, w \rangle$ ⟶ T (collection of tiles)

M accepts w ⟺ T contains a match

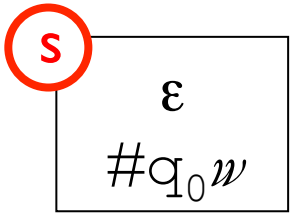
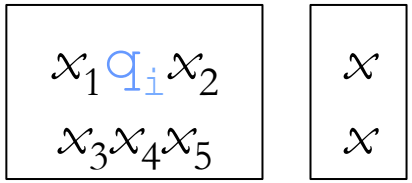
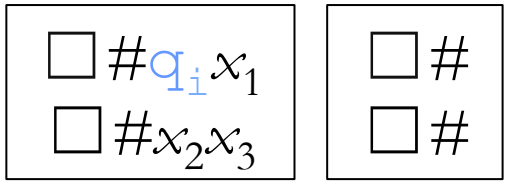
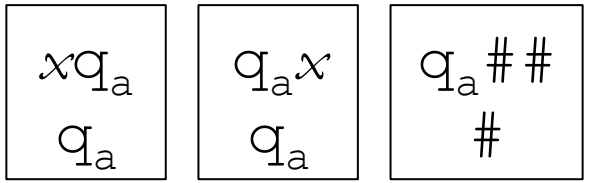
- On input $\langle M, w \rangle$ we construct these tiles for PCP



for each valid window
with state q_i in top middle

for all x in $\Gamma \cup \{\#\}$

Undecidability of PCP

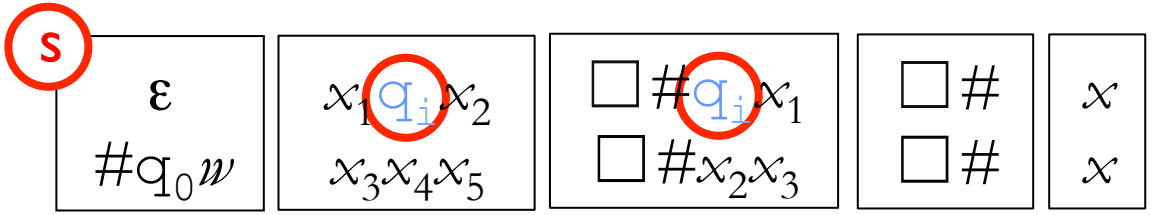
tile type	purpose
	represents initial configuration
	represent valid transitions between configurations
	add blank spaces before # if necessary
	complete match if computation accepts

Undecidability of PCP

q_0a%ab#xq_1%ab#... xx%xxq_7□#xx%xx□q_a

accepting computation history

$q_0$ab%ab#xq_1b%ab#... xx%xxq_7□#



Undecidability of PCP

- Once the accepting state symbol occurs, the last two tiles can "eat up" the rest of the symbols

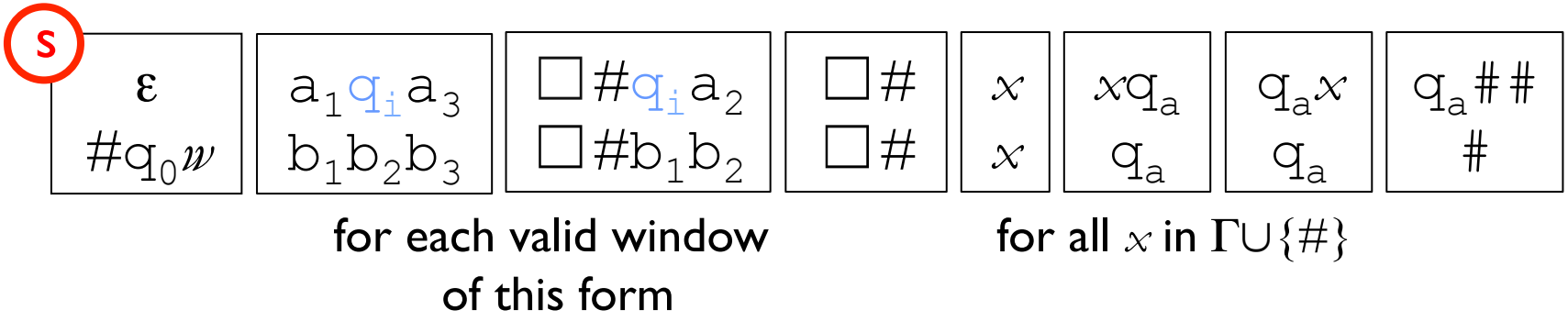
$$\#xx\%xx\square q_a\#xx\%xxq_a\#xx\%xq_a\#...\#q_a\#\#$$
$$\#xx\%xx\square q_a\#xx\%xxq_a\#xx\%xq_a\#...\#q_a\#\#$$

x	xq_a	q_ax	$q_a\#\#$
x	q_a	q_a	$\#$

Undecidability of PCP

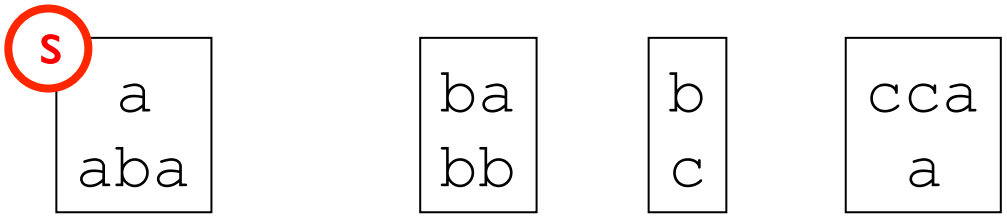
- If M rejects on input w, then q_{rej} appears on bottom at some point, but it cannot be matched on top

- If M loops on w, then matching keeps going forever

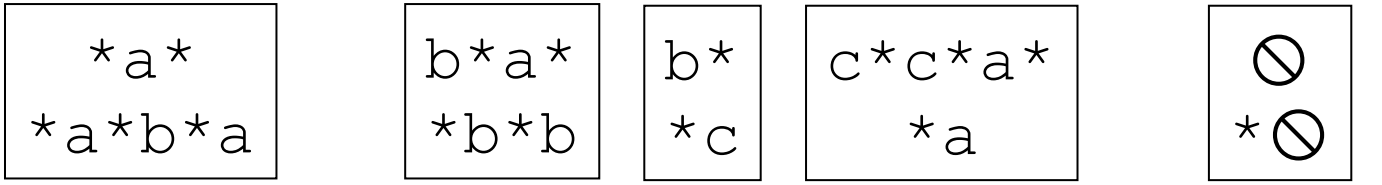


Getting rid of the starting tile

- We assumed that one tile marked as starting tile



- We can remove assumption by changing tiles a bit

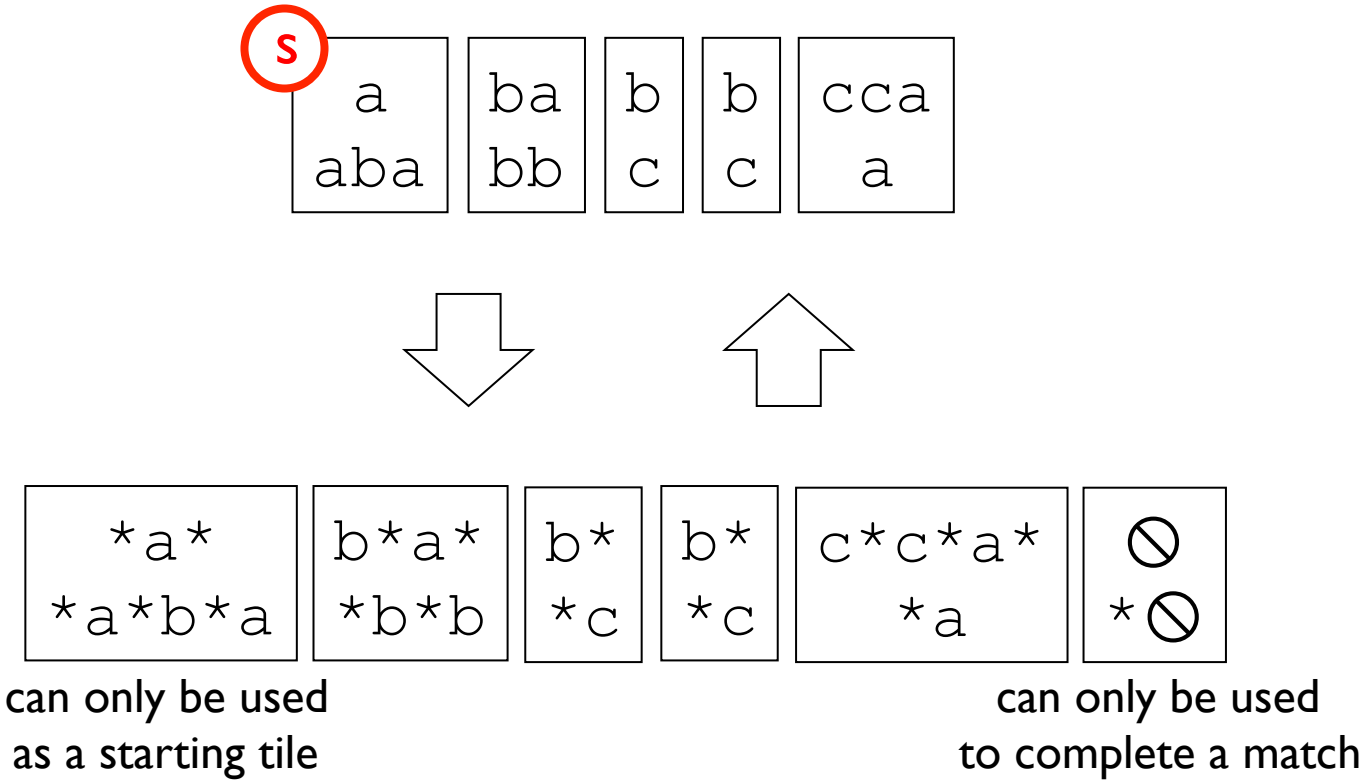


"starting tile"
begins with *

"middle tiles"

"final tile"

Getting rid of the starting tile



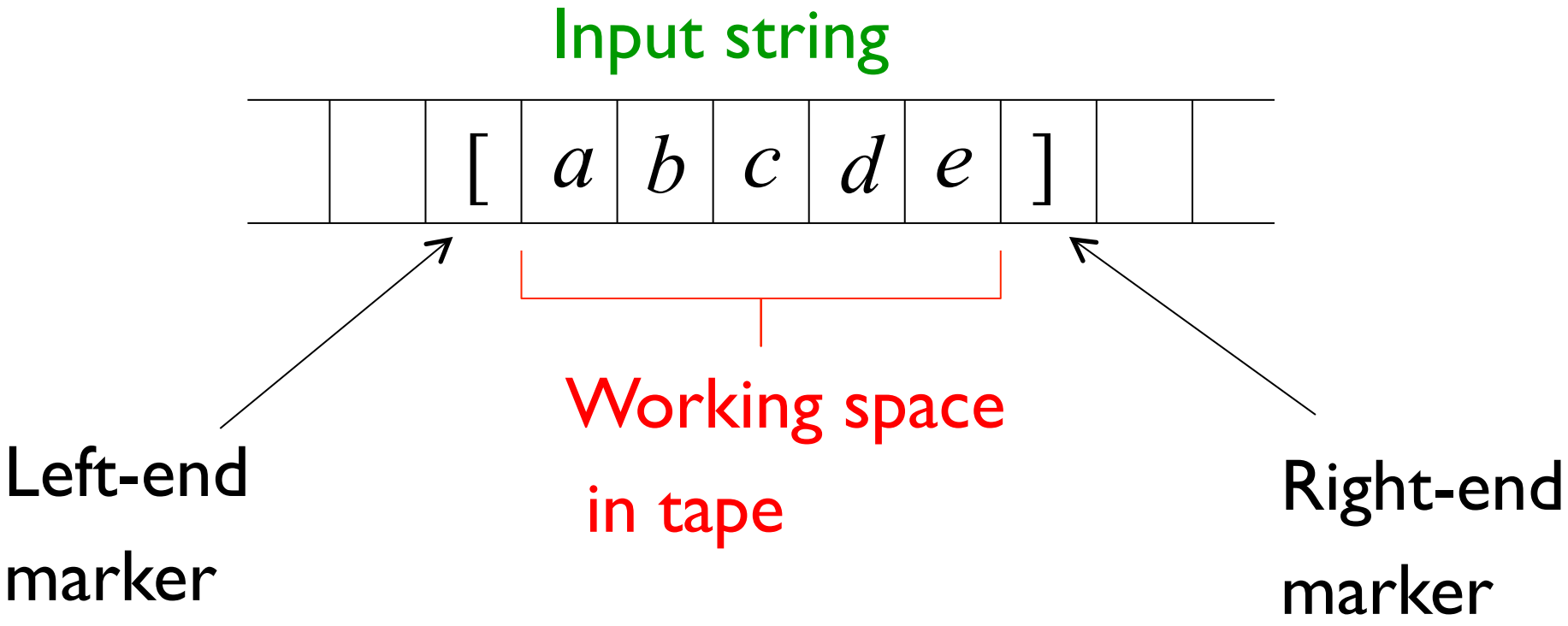
The Chomsky Hierarchy

Linear-Bounded Automata:

Same as Turing Machines with one difference:

the input string tape space
is the only tape space allowed to use

Linear Bounded Automaton (LBA)



All computation is done between end markers

Example languages accepted by LBAs:

$$L = \{a^n b^n c^n\} \qquad\qquad L = \{a^{n!}\}$$

LBAs have more power than PDAs

(pushdown automata)

LBAs have less power than Turing Machines

We define LBAs as non-deterministic.

Open Problem (since 1964):

Do nondeterministic LBAs have the same power as deterministic LBAs ?

LBA vs PDA:

LBAs are closed under complementation (only shown in 1987 by Immerman and Szelepcsényi); PDAs are not and so deterministic PDAs are weaker than PDAs.

Unrestricted Grammars:

Productions

$$u \rightarrow v$$

String of variables and terminals

String of variables and terminals

Example unrestricted grammar:

$$S \rightarrow aBc$$

$$aB \rightarrow cA$$

$$Ac \rightarrow d$$

Theorem:

A language L is Turing-Acceptable if and only if L is generated by an unrestricted grammar

Context-Sensitive Grammars:

Productions

$$u \rightarrow v$$

String of variables and terminals

String of variables and terminals

and: $|u| \leq |v|$

The language $\{a^n b^n c^n\}$

is context-sensitive:

$$S \rightarrow abc \mid aAbc$$

$$Ab \rightarrow bA$$

$$Ac \rightarrow Bbcc$$

$$bB \rightarrow Bb$$

$$aB \rightarrow aa \mid aaA$$

Theorem:

A language L is context sensitive

if and only if

it is accepted by a Linear-Bounded automaton

Observation:

Acceptance of a given input for LBAs is decidable.

(However, emptiness problem is undecidable.)

Fact:

There is a language which is recursive

but not context-sensitive.

$L_d = \{ x \mid x \text{ is LBA and } x \notin \mathcal{L}(x)\}$ is decidable but not context-sensitive

Is decidable by the following program:

1. Check if x is LBA, if not *reject*
2. Run x on input x, if accepts then *reject*, otherwise *accept*

Now, suppose LBA M* recognises L_d, i.e.=$\mathcal{L}$(M*).

If M* $\notin \mathcal{L}$(M*) then M* $\in \mathcal{L}$(M*).

If M* $\in \mathcal{L}$(M*) then M* $\notin \mathcal{L}$(M*).

The Chomsky Hierarchy

All possible languages = 2^{Σ^*}