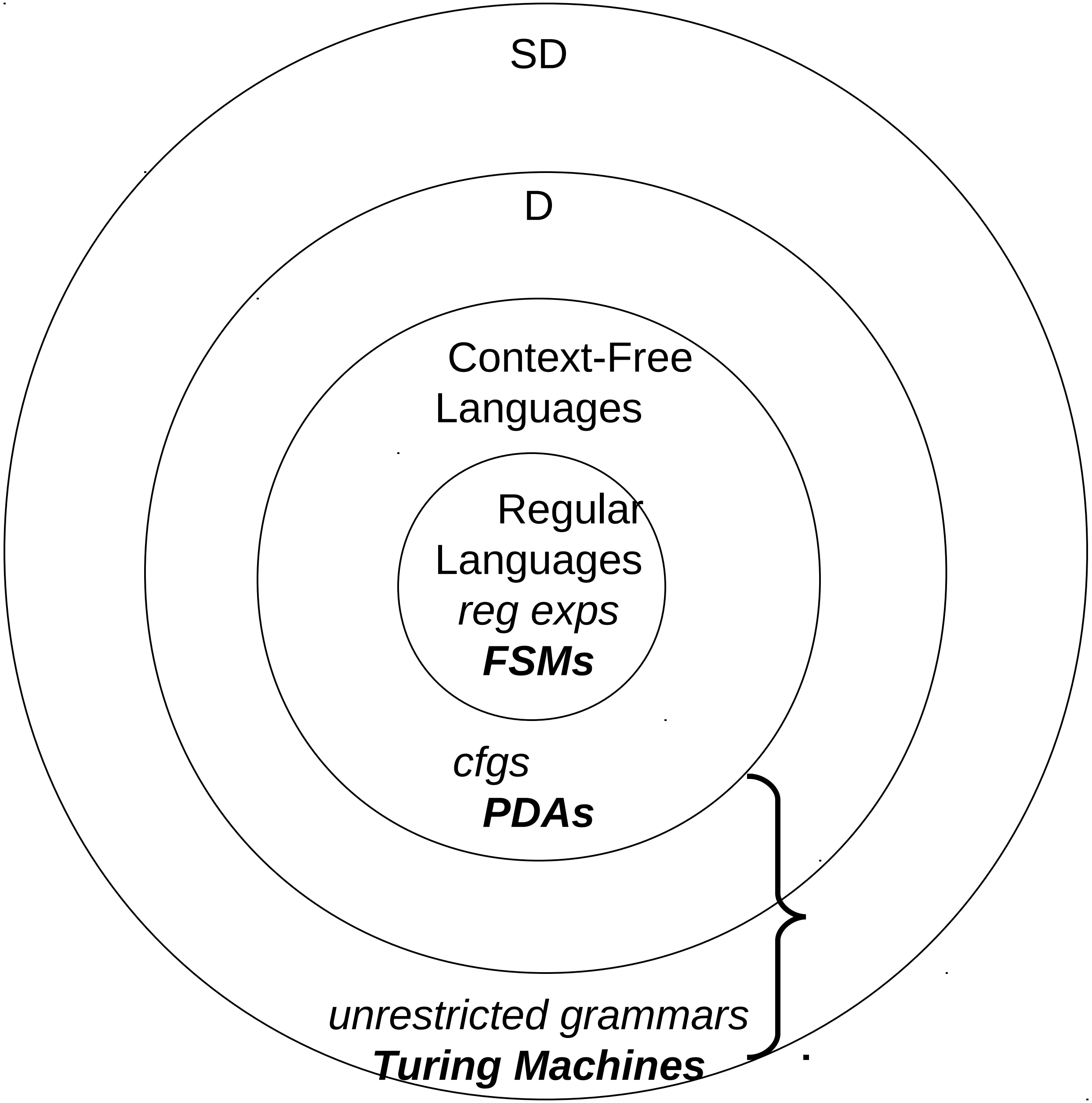
Turing Machines

Chapter 17

Languages and Machines



SD

D

Context-Free Languages

Regular Languages

reg exps

FSMs

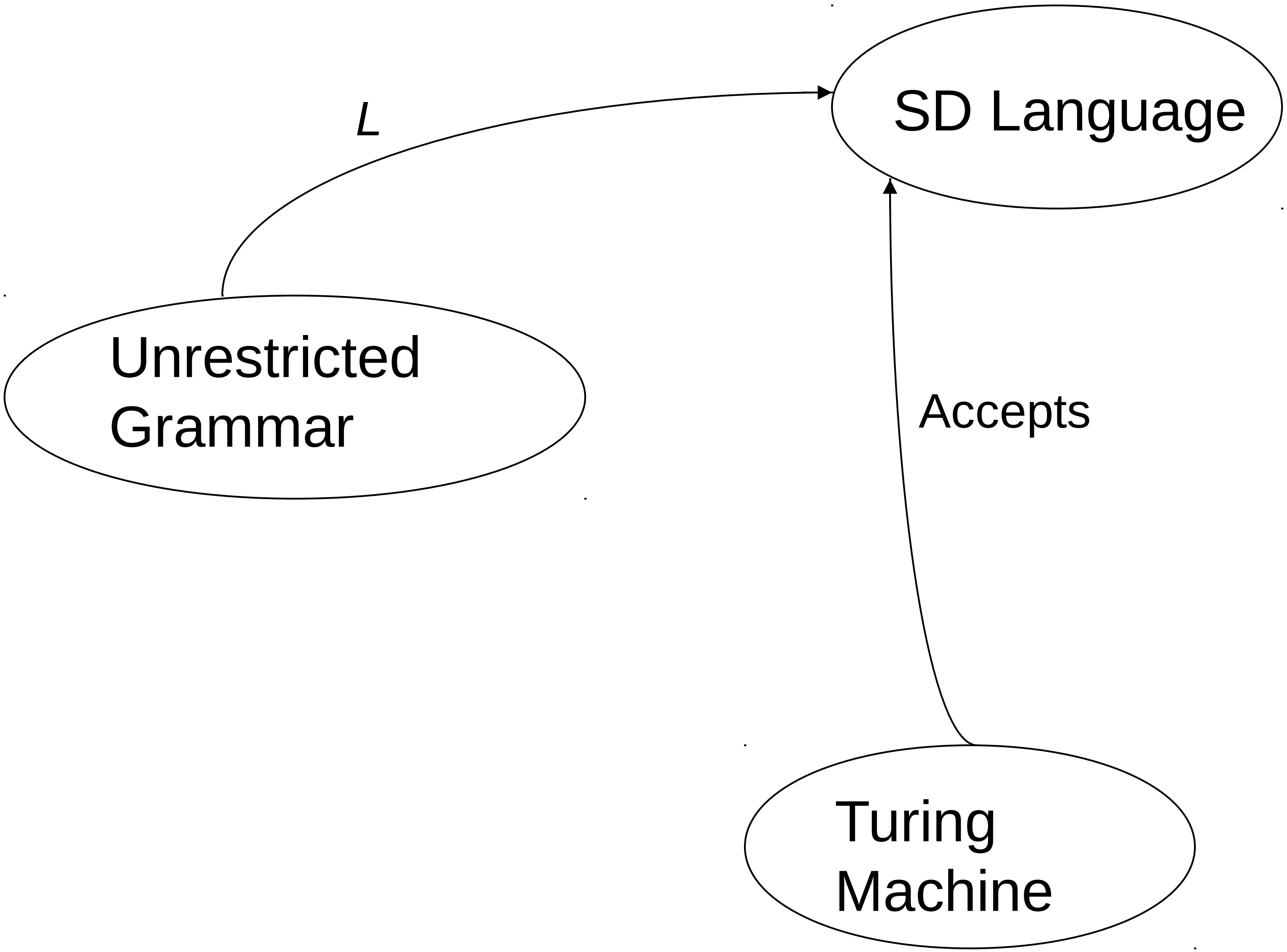
cfgs

PDAs

unrestricted grammars

Turing Machines

Grammars, SD Languages, and Turing Machines



Turing Machines

Can we come up with a new kind of automaton that has two properties:

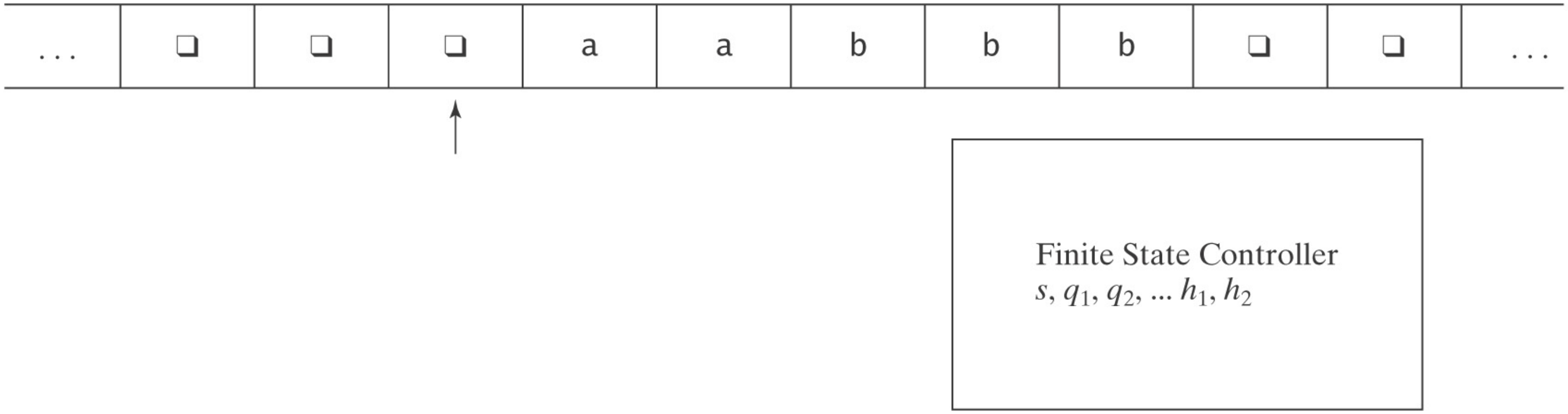
- powerful enough to describe all computable things

  unlike FSMs and PDAs.

- simple enough that we can reason formally about it

  like FSMs and PDAs,
  unlike real computers.

Turing Machines



. . .	❑	❑	❑	a	a	b	b	b	❑	❑	. . .

Finite State Controller
$s, q_1, q_2, \ldots h_1, h_2$

At each step, the machine must:

- choose its next state,
- write on the current square, and
- move left or right.

A Formal Definition

A Turing machine *M* is a sixtuple $(K, \Sigma, \Gamma, \delta, s, H)$:

- *K* is a finite set of states;
- Σ is the input alphabet, which does not contain ❑;
- Γ is the tape alphabet, which must contain ❑ and have Σ as a subset.
- $s \in K$ is the initial state;
- $H \subseteq K$ is the set of halting states;
- δ is the transition function:

$(K - H)$	$\times\ \Gamma$	to	K	$\times$	Γ	$\times$	$\{\rightarrow, \leftarrow\}$
non-halting state	× tape char		state	×	tape char	×	action (R or L)

Notes on the Definition

1. The input tape is infinite in both directions.

2. δ is a function, not a relation. So this is a definition for deterministic Turing machines.

3. δ must be defined for all (state, input) pairs unless the state is a halting state.

4. Turing machines do not necessarily halt (unlike FSM's and PDAs). Why? To halt, they must enter a halting state. Otherwise they loop.

5. Turing machines generate output so they can compute functions.

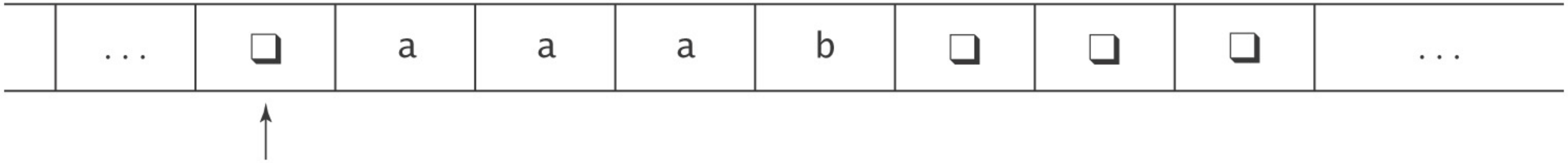
An Example

M takes as input a string in the language:

$\{a^i b^j, 0 \leq j \leq i\}$,

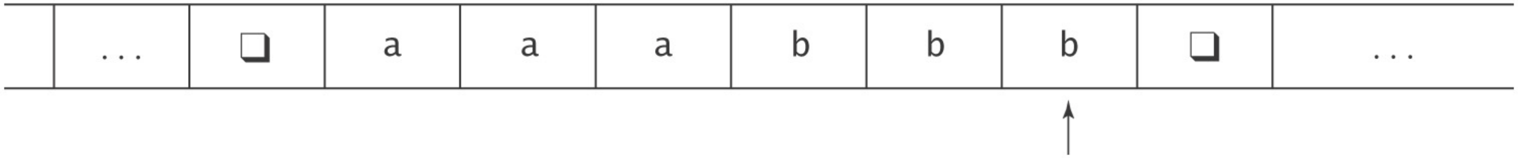
and adds b’s as required to make the number of b’s equal the number of a’s.

The input to *M* will look like this:



. . .	❑	a	a	a	b	❑	❑	❑	. . .

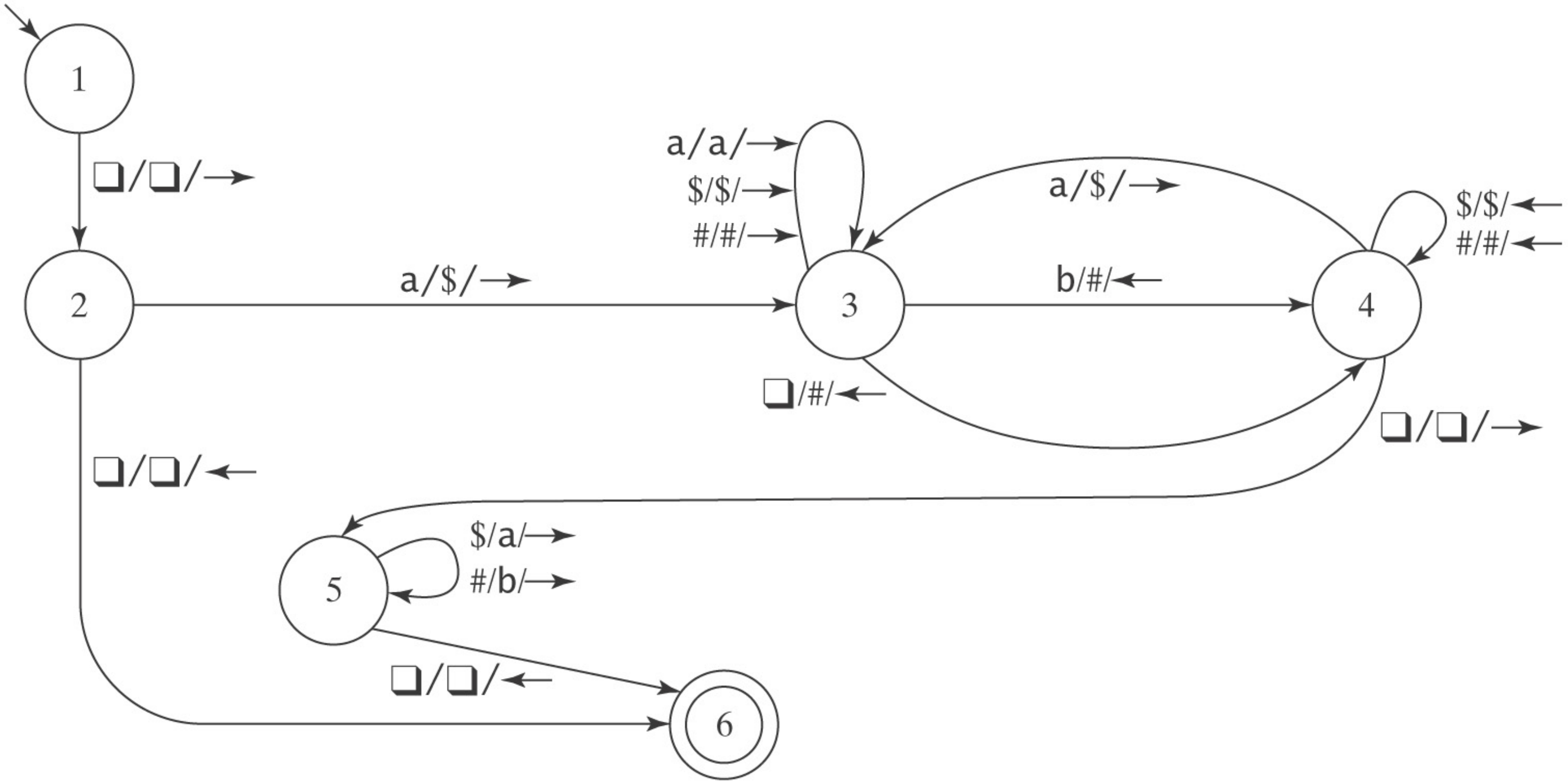
The output should be:



. . .	❑	a	a	a	b	b	b	❑	. . .

The Details

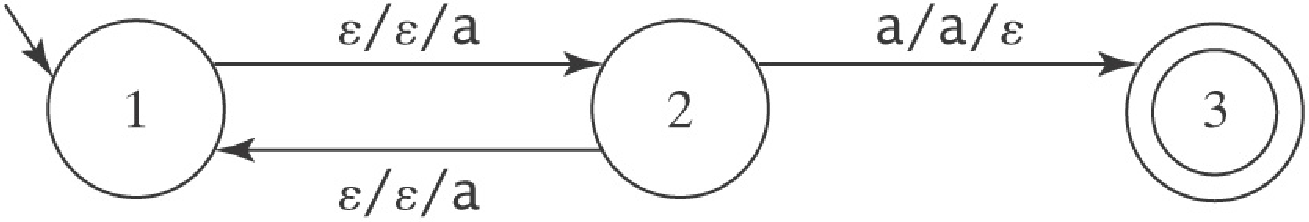
K = {1, 2, 3, 4, 5, 6}, Σ = {a, b}, Γ = {a, b, ❑, $, #},
s = 1, *H* = {6}, δ =



Halting

- A DFSM *M*, on input *w*, is guaranteed to halt in |*w*| steps.

- A PDA *M*, on input *w*, is not guaranteed to halt. To see why, consider again *M* =



But there exists an algorithm to construct an equivalent PDA *M*´ that is guaranteed to halt.

A TM *M*, on input *w*, is not guaranteed to halt. And there exists no algorithm to construct one that is guaranteed to do so.

Formalizing the Operation

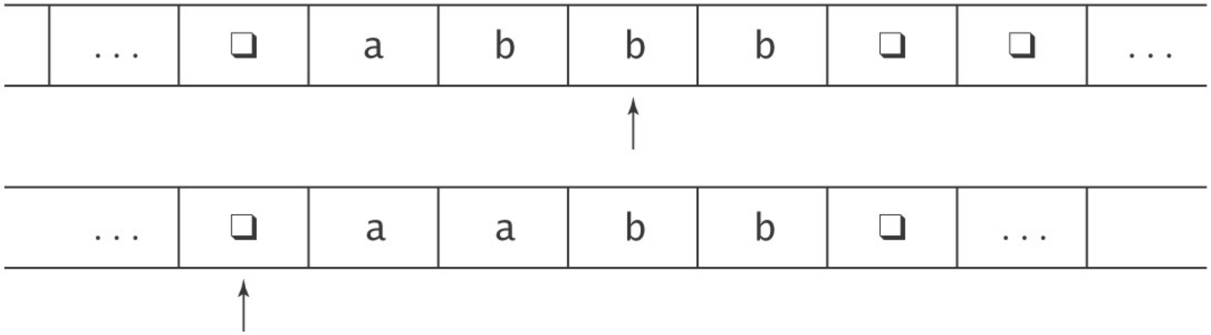
A *configuration* of a Turing machine

$M = (K, \Sigma, \Gamma, s, H)$ is an element of:

$$K \quad \times \quad ((\Gamma - \{\square\})\,\Gamma^*) \cup \{\varepsilon\} \quad \times \quad \Gamma \quad \times \quad (\Gamma^*\,(\Gamma - \{\square\})) \cup \{\varepsilon\}$$

state	up to scanned square	scanned square	after scanned square

Example Configurations



As a 4-tuple	Shorthand
(q, ab, b, b)	(q, ab$\underline{\text{b}}$b)
(q, ε, ❑, aabb)	(q, $\underline{❑}$abbb)

(1)	(q, ab, b, b)	=	(q, ab$\underline{\text{b}}$b)
(2)	(q, ε, ❑, aabb)	=	(q, $\underline{❑}$aabb)

Initial configuration is (s, $\underline{❑}w$).

Yields

(q_1, w_1) **$|\!-_M$** (q_2, w_2) iff (q_2, w_2) is derivable, via δ, in one step.

For any TM M, let **$|\!-_M*$** be the reflexive, transitive closure of $|\!-_M$.

Configuration C_1 ***yields*** configuration C_2 if: C_1 **$|\!-_M*$** C_2.

A ***path*** through M is a sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that C_0 is the initial configuration and:

$$C_0 \; |\!-_M \; C_1 \; |\!-_M \; C_2 \; |\!-_M \; \ldots \; |\!-_M \; C_n.$$

A **computation** by M is a path that halts.

If a computation is of *length n* or has n steps, we write:

$$C_0 \; |\!-_M{}^n \; C_n$$

A Notation for Turing Machines

(1) Define some basic machines

- Symbol writing machines

  For each $x \in \Gamma$, define ***M_x***, written just *x*, to be a machine that writes *x*, then halts.

- Head moving machines

  R: for each $x \in \Gamma$, $\delta(s, x) = (s, x, \rightarrow)$
  L: for each $x \in \Gamma$, $\delta(s, x) = (s, x, \leftarrow)$

- Machines that simply halt:

h, which simply halts.
n, which halts and rejects.
y, which halts and accepts.

Checking Inputs and Combining Machines

Next we need to describe how to:

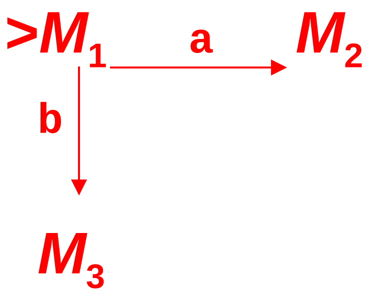
- Check the tape and branch based on what character we see, and
- Combine the basic machines to form larger ones.

To do this, we need two forms:

- ***M_1M_2***
  - Begin in start state of M_1, run M_1 until halts, begin M_2 in start state, run M_2 until halts, then halt. If either fails to halt, then M_1M_2 fails to halt.
- $M_1 \xrightarrow{\textit{<condition>}} M_2$
  - The same, except that <*condition*> is checked to move from M_1 to M_2.

A Notation for Turing Machines, Cont'd

Example:



>M_1

a

M_2

b

M_3

- Start in the start state of M_1. (“>” marks the beginning)
- Compute until M_1 reaches a halt state.
- Examine the tape and take the appropriate transition.
- Start in the start state of the next machine, etc.
- Halt if any component reaches a halt state and has no place to go.
- If any component fails to halt, then the entire machine may fail to halt.

Shorthands

$$\begin{tikzcd} M_1 \arrow[r, shift left, "a"] \arrow[r, shift right, "b"'] & M_2 \end{tikzcd}$$ becomes $M_1 \xrightarrow{a, b} M_2$

$M_1 \xrightarrow{\text{all elems of } \Gamma} M_2$ becomes $M_1 \longrightarrow M_2$

or

M_1M_2

Variables

$M_1 \xrightarrow[\text{except a}]{\text{all elems of } \Gamma} M_2$ becomes $M_1 \xrightarrow{x \leftarrow \neg a} M_2$

and x takes on the value of the current square

$M_1 \xrightarrow{a, b} M_2$ becomes $M_1 \xrightarrow{x \leftarrow a, b} M_2$

and x takes on the value of the current square

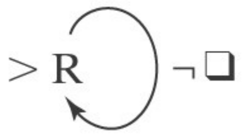
$M_1 \xrightarrow{x = y} M_2$

if $x = y$ then take the transition

e.g., $> \xrightarrow{x \leftarrow \neg \square} Rx$

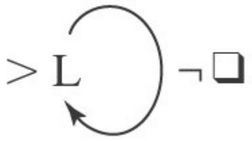
if the current square is not blank, go right and copy it.

Some Useful Machines



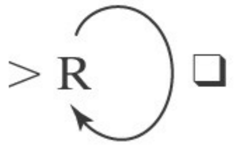
Find the first blank square to the right of the current square.

$R_{\square}$



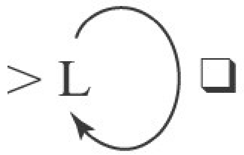
Find the first blank square to the left of the current square.

$L_{\square}$



Find the first nonblank square to the right of the current square.

$R_{\neg\square}$



Find the first nonblank square to the left of the current square

$L_{\neg\square}$

More Useful Machines

L_a

Find the first occurrence of a to the left of the current square.

$R_{a,b}$

Find the first occurrence of a or b to the right of the current square.

$$\begin{tikzcd} L_{a,b} \arrow[r, "a"] \arrow[d, "b"] & M_1 \\ M_2 & \end{tikzcd}$$

Find the first occurrence of a or b to the left of the current square, then go to M_1 if the detected character is a; go to M_2 if the detected character is b.

$L_{x \leftarrow a,b}$

Find the first occurrence of a or b to the left of the current square and set *x* to the value found.

$L_{x \leftarrow a,b}Rx$

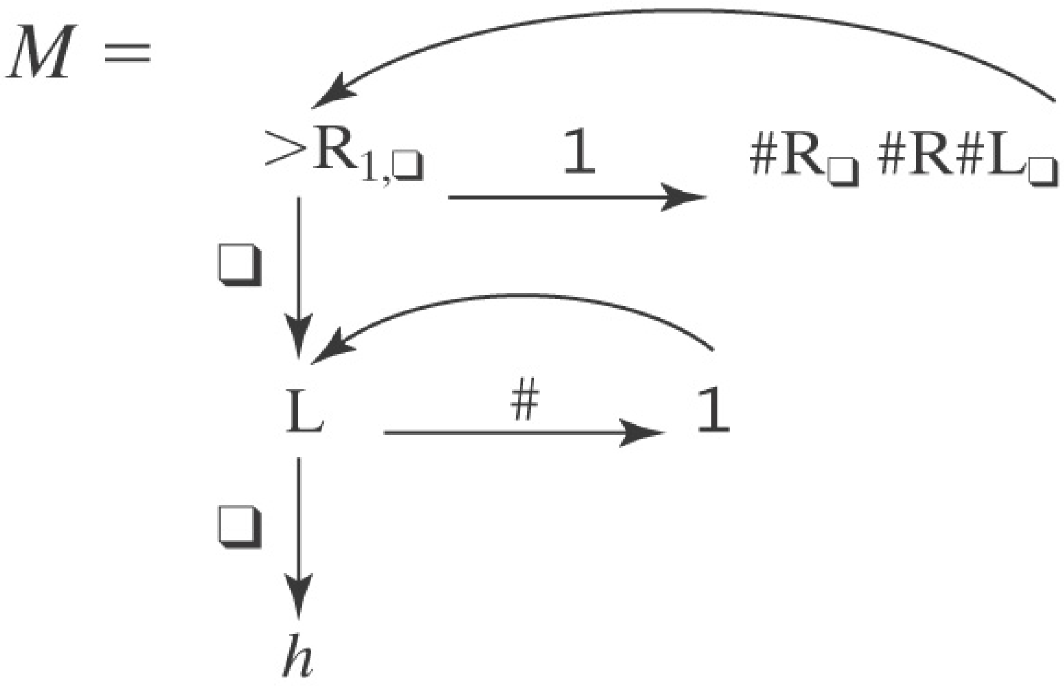
Find the first occurrence of a or b to the left of the current square, set *x* to the value found, move one square to the right, and write *x* (a or b).

An Example

Input: ❑*w* *w* ∈ {1}*

Output: ❑*w*^{3}

Example: ❑111❑❑❑❑❑❑❑❑❑❑❑❑❑❑

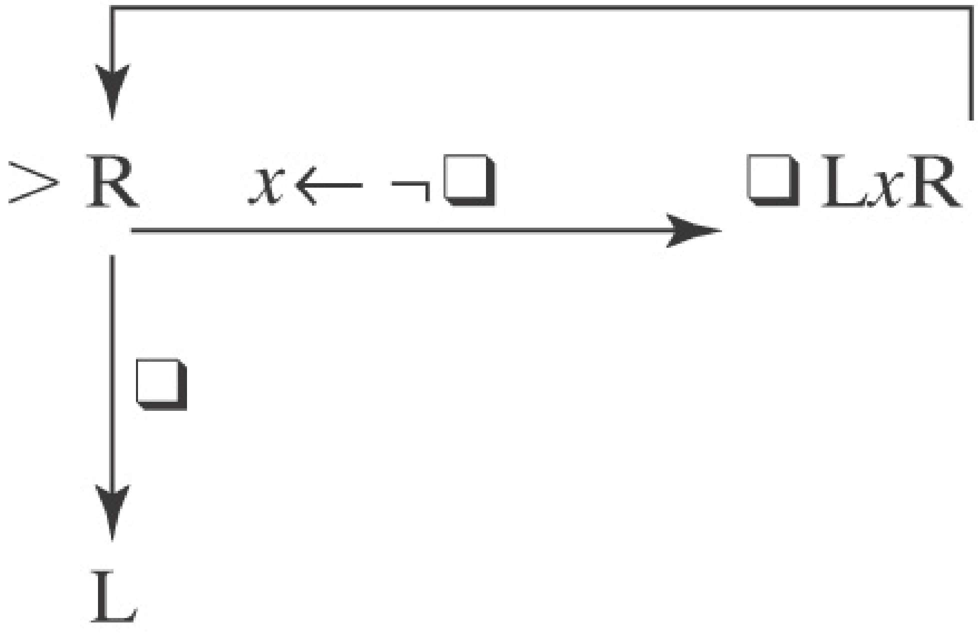


A Shifting Machine $S_{\leftarrow}$

Input: ❑*u*$\underline{❑}$*w*❑

Output: ❑*uw*$\underline{❑}$

Example: ❑ba$\underline{❑}$abba❑❑❑❑❑❑❑❑❑❑❑❑



Turing Machines as Language Recognizers

Convention: We will write the input on the tape as:

❑*w*❑, *w* contains no ❑'s

The initial configuration of *M* will then be:

(s, $\underline{❑}w$)

Let $M = (K, \Sigma, \Gamma, \delta, s, \{y, n\})$.

- *M* ***accepts*** a string *w* iff (s, $\underline{❑}w$) $|\!-_M{}^*$ (y, w') for some string w'.

- *M* ***rejects*** a string *w* iff (s, $\underline{❑}w$) $|\!-_M{}^*$ (n, w') for some string w'.

Turing Machines as Language Recognizers

M ***decides*** a language $L \subseteq \Sigma^*$ iff:
For any string $w \in \Sigma^*$ it is true that:
if $w \in L$ then M accepts w, and
if $w \notin L$ then M rejects w.

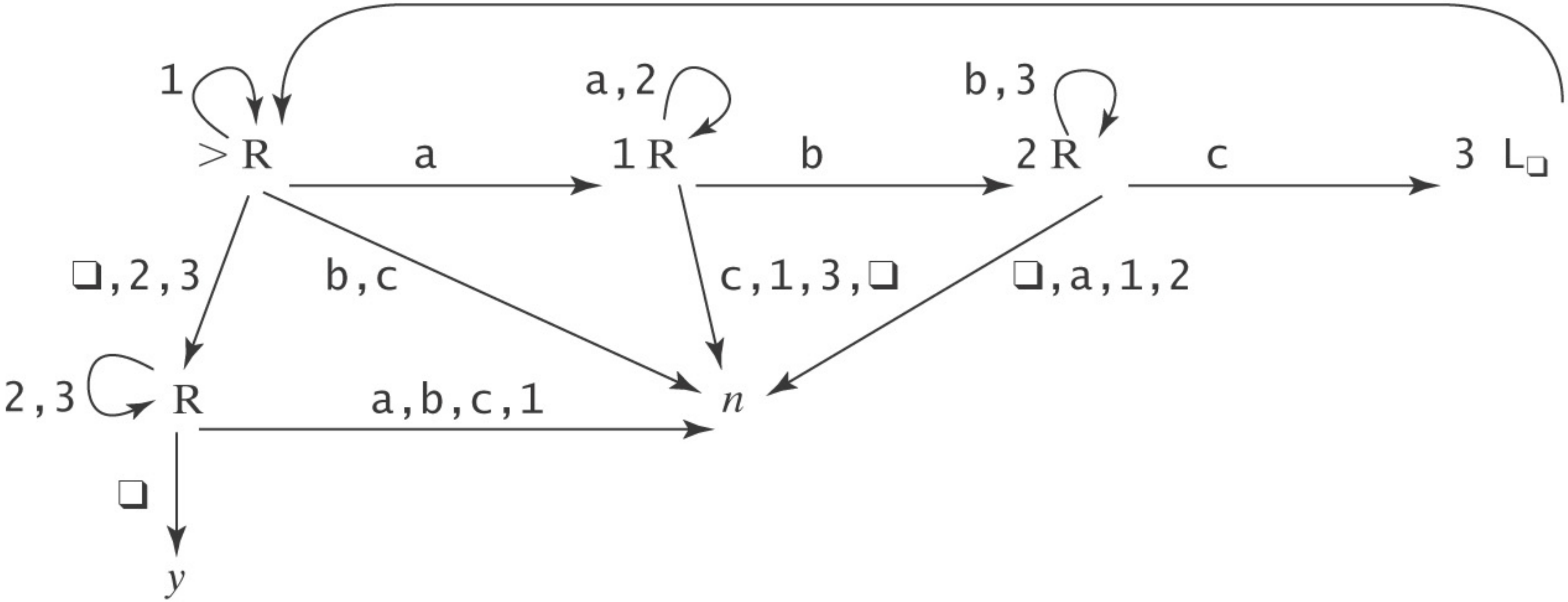
A language L is ***decidable*** iff there is a Turing machine M that decides it. In this case, we will say that L is in ***D***.

A Deciding Example

$A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$

Example: ❑aabbcc❑❑❑❑❑❑❑❑❑

Example: ❑aaccb❑❑❑❑❑❑❑❑❑

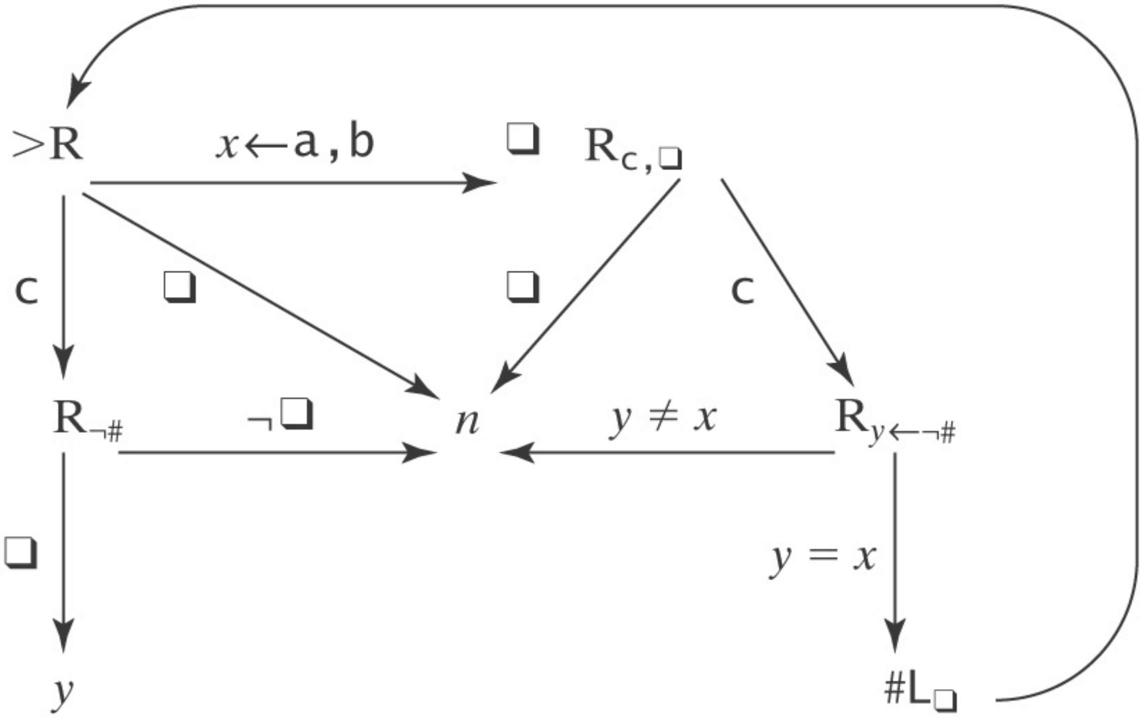


Another Deciding Example

WcW = {*w*c*w* : *w* ∈ {a, b}*}

Example: ❑abbcabb❑❑❑

Example: ❑acabb❑❑❑



Semideciding a Language

Let Σ_M be the input alphabet to a TM M. Let $L \subseteq \Sigma_M^*$.

M ***semidecides*** L iff, for any string $w \in \Sigma_M^*$:

- $w \in L \rightarrow M$ accepts w
- $w \notin L \rightarrow M$ does not accept w.

M may either:
reject or
fail to halt.

A language L is ***semidecidable*** iff there is a Turing machine that semidecides it. We define the set ***SD*** to be the set of all semidecidable languages.

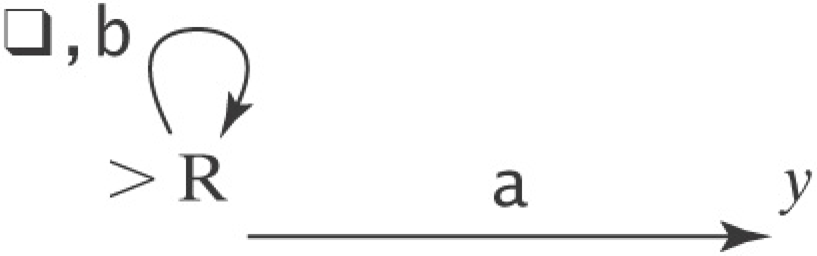
Example of Semideciding

Let *L* = b*a(a ∪ b)*

We can build *M* to semidecide *L*:

1. Loop
    1.1 Move one square to the right. If the character under the read head is an a, halt and accept.

In our macro language, *M* is:

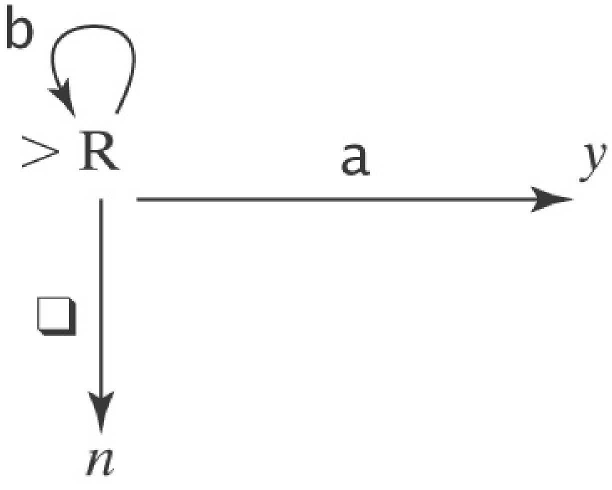


Example of Semideciding

L = b*a(a ∪ b)*. We can also decide *L*:

```
Loop:
        1.1 Move one square to the right.
        1.2 If the character under the read/write head is
            an a, halt and accept.
        1.3 If it is ❑, halt and reject.
```

In our macro language, *M* is:



Computing Functions

Let $M = (K, \Sigma, \Gamma, \delta, s, \{h\})$. Its initial configuration is $(s, \underline{\square}w)$.

Define $M(w) = z$ iff $(s, \underline{\square}w) \vdash_M^* (h, \underline{\square}z)$.

Let $\Sigma' \subseteq \Sigma$ be M's *output alphabet* (i.e., the set of symbols that M may leave on its tape when it halts)

Let f be any function from Σ^* to Σ'^*.

M ***computes f*** iff for all $w \in \Sigma^*$, $M(w) = f(w)$.

A function f is ***recursive*** or ***computable*** iff there is a Turing machine M that computes it.

(Note that M always halts.)

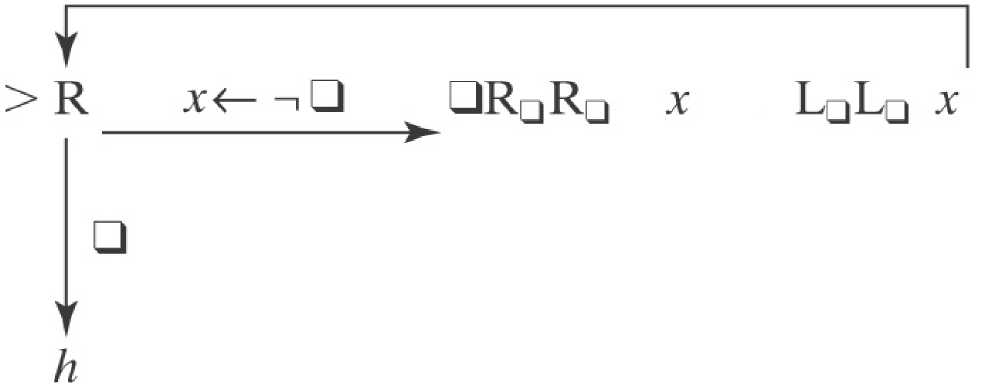
Example of Computing a Function

Let $\Sigma = \{a, b\}$. Let $f(w) = ww$.

Input: $\underline{\square}w\square\square\square\square\square\square$ Output: $\underline{\square}ww\square$

Define the copy machine C:

$\underline{\square}w\square\square\square\square\square\square \rightarrow \square w\underline{\square}w\square$



$> R$ $x \leftarrow \neg\square$ $\square R_{\square} R_{\square}$ x $L_{\square} L_{\square}$ x

$\square$

h

Remember the $S_{\leftarrow}$ machine:

$\square u\underline{\square}w\square \rightarrow \square uw\underline{\square}$

Then the machine to compute f is just $>C\ S_{\leftarrow}\ L_{\square}$

Computing Numeric Functions

For any positive integer k, $\textit{value}_k(n)$ returns the nonnegative integer that is encoded, base k, by the string n.

For example:

- $\textit{value}_2(\texttt{101}) = 5$.
- $\textit{value}_8(\texttt{101}) = 65$.

TM M computes a function f from $\mathbb{N}^m$ to $\mathbb{N}$ iff, for some k:

$$\textit{value}_k(M(n_1;n_2;\ldots n_m)) = f(\textit{value}_k(n_1), \ldots\ \textit{value}_k(n_m)).$$

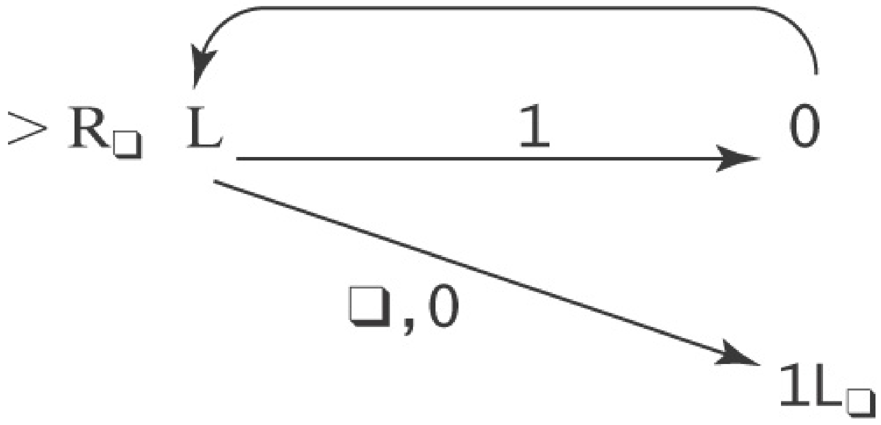
Computing Numeric Functions

Example: *succ*(*n*) = *n* + 1

We will represent *n* in binary. So *n* ∈ 0 ∪ 1{0, 1}*

Input: ❑*n*❑❑❑❑❑❑ Output: ❑*n*+1❑

❑1111❑❑❑❑ Output: ❑10000❑



Not All Functions Are Computable

Let *T* be the set of all TMs that:

- Have tape alphabet Γ = {❑, 1}, and
- Halt on a blank tape.

Define the **busy beaver functions** *S*(*n*) and Σ(*n*):

- *S*(*n*): the maximum number of steps that are executed by any element of *T* with *n*-nonhalting states, when started on a blank tape, before it halts.
- Σ(*n*): the maximum number of 1's left on the tape by any element of *T* with *n*-nonhalting states, when it halts.

n	***S*(*n*)**	**Σ(*n*)**
1	1	1
2	6	4
3	21	6
4	107	13
5	$\geq$ 47,176,870	4098
6	$\geq 3 \cdot 10^{1730}$	$\geq 1.29 \cdot 10^{865}$

Why Are We Working with Our Hands Tied Behind Our Backs?

Turing machines

Are more powerful than any of the other formalisms we have studied so far.

☺

Turing machines

Are a **lot** harder to work with than all the real computers we have available.

☹

Why bother?

The very simplicity that makes it hard to program Turing machines makes it possible to reason formally about what they can do. If we can, once, show that anything a real computer can do can be done (albeit clumsily) on a Turing machine, then we have a way to reason about what real computers can do.