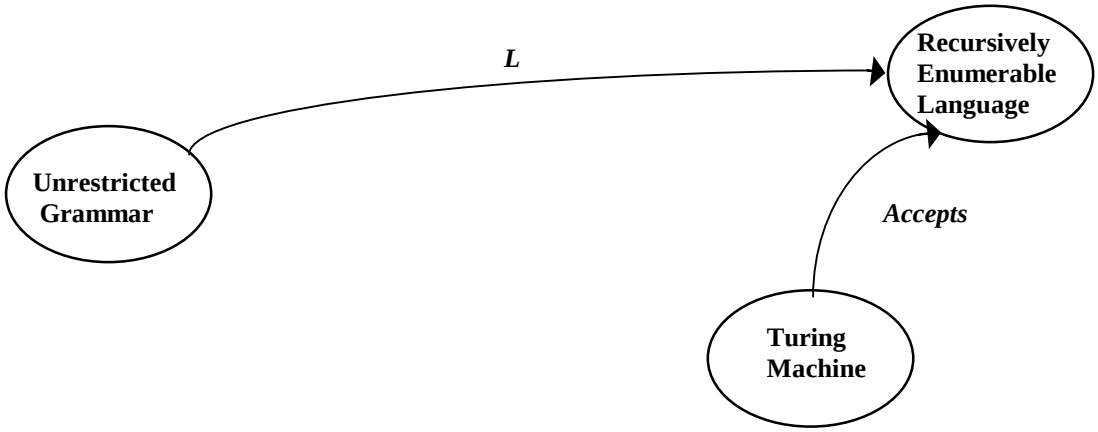
Turing Machines

Read K & S 4.1.
Do Homework 17.

Grammars, Recursively Enumerable Languages, and Turing Machines

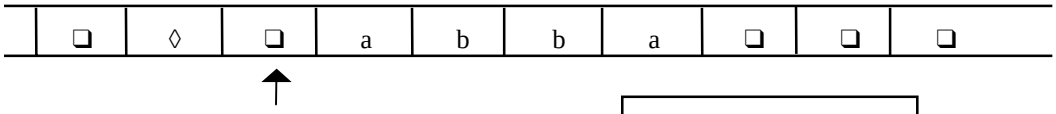


Turing Machines

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

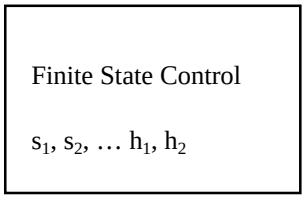
- powerful enough to describe all computable things
  unlike FSMs and PDAs
- simple enough that we can reason formally about it
  like FSMs and PDAs
  unlike real computers

Turing Machines



At each step, the machine may:

- go to a new state, and
- either
  - write on the current square, or
  - move left or right



A Formal Definition

A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$:

K is a finite set of states;
Σ is an alphabet, containing at least ❑ and ◊, but not $\rightarrow$ or $\leftarrow$;
$s \in K$ is the initial state;
$H \subseteq K$ is the set of halting states;
δ is a function from:

$(K - H)$	$\times$	Σ	to	K	$\times$	$(\Sigma \cup \{\rightarrow, \leftarrow\})$
non-halting state	$\times$	input symbol		state	$\times$	action (write or move)

such that

(a) if the input symbol is ◊, the action is $\rightarrow$, and
(b) ◊ can never be written .

Notes on the Definition

1. The input tape is infinite to the right (and full of ❑), but has a wall to the left. Some definitions allow infinite tape in both directions, but it doesn't matter.

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

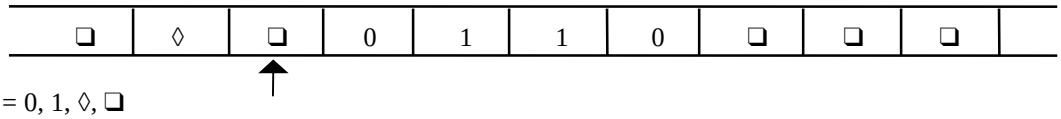
3. δ must be defined for all state, input pairs unless the state is a halt state.

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

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

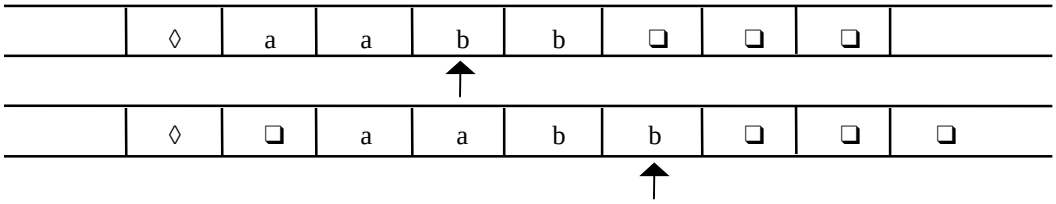
A Simple Example

A Turing Machine Odd Parity Machine:



Σ = 0, 1, ◊, ❑
s =
H =
δ =

Formalizing the Operation



(1)

(2)

A **configuration** of a Turing machine
M = (K, Σ, δ, s, H) is a member of

K	×	◊Σ*	×	(Σ*(Σ - {❑})) ∪ ε
state		input up to scanned square		input after scanned square

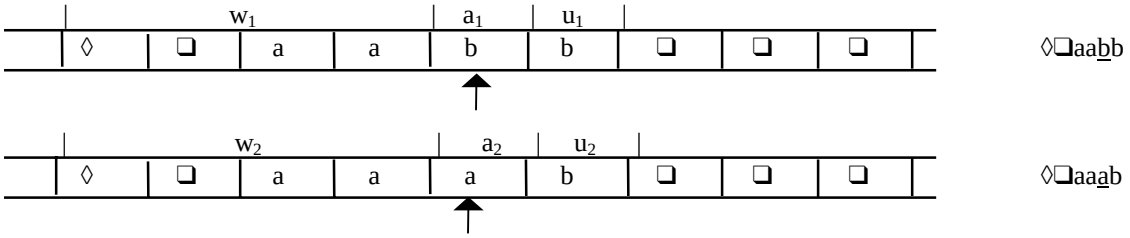
The input after the scanned square may be empty, but it may not end with a blank. We assume the entire tape to the right of the input is filled with blanks.

(1) (q, ◊aab, b) = (q, ◊aa<u>b</u>b)
(2) (h, ◊❑aabb, ε) = (h, ◊❑aab<u>b</u>) a halting configuration

Yields

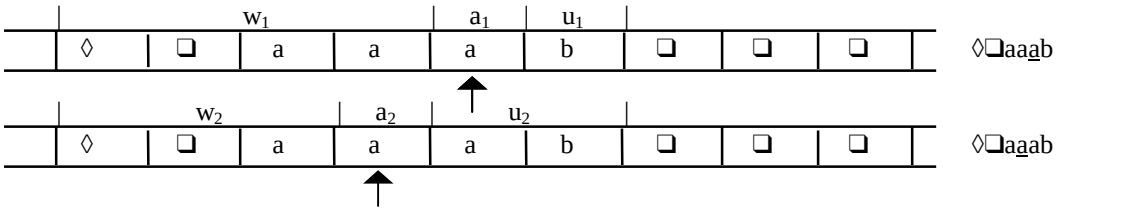
$(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$, a_1 and $a_2 \in \Sigma$, iff $\exists\ b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$ and either:

(1) $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$, and $a_2 = b$ (rewrite without moving the head)

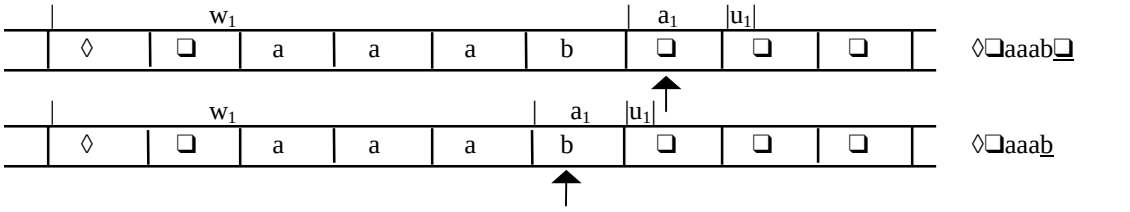


Yields, Continued

(2) $b = \leftarrow$, $w_1 = w_2a_2$, and either
(a) $u_2 = a_1u_1$, if $a_1 \neq ❑$ or $u_1 \neq \varepsilon$,



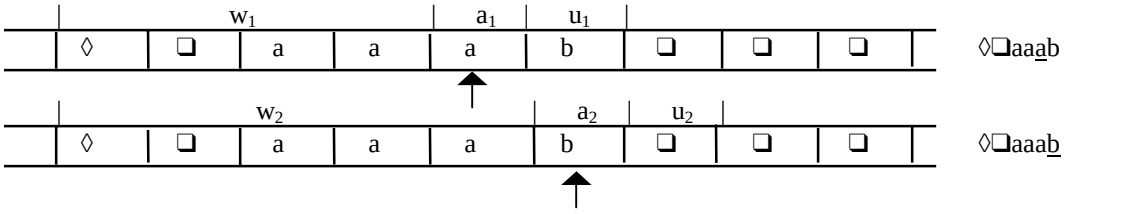
or (b) $u_2 = \varepsilon$, if $a_1 = ❑$ and $u_1 = \varepsilon$



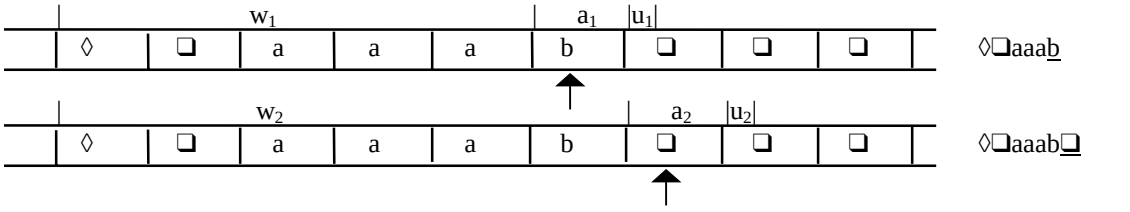
If we scan left off the first square of the blank region, then drop that square from the configuration.

Yields, Continued

(3) $b = \rightarrow$, $w_2 = w_1a_1$, and either
(a) $u_1 = a_2u_2$



or (b) $u_1 = u_2 = \varepsilon$ and $a_2 = ❑$



If we scan right onto the first square of the blank region, then a new blank appears in the configuration.

Yields, Continued

For any Turing machine M, let $|-_M^*$ be the reflexive, transitive closure of $|-_M$.

Configuration C_1 **yields** configuration C_2 if

$C_1 \ |-_M^* \ C_2$.

A **computation** by M is a sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that

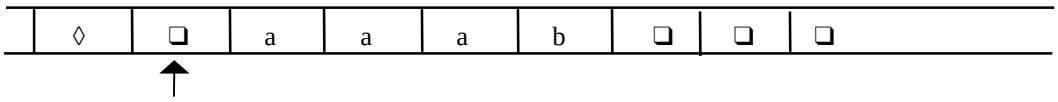
$C_0 \ |-_M \ C_1 \ |-_M \ C_2 \ |-_M \ \ldots \ |-_M \ C_n$.

We say that the computation is of **length** n or that it has n **steps**, and we write

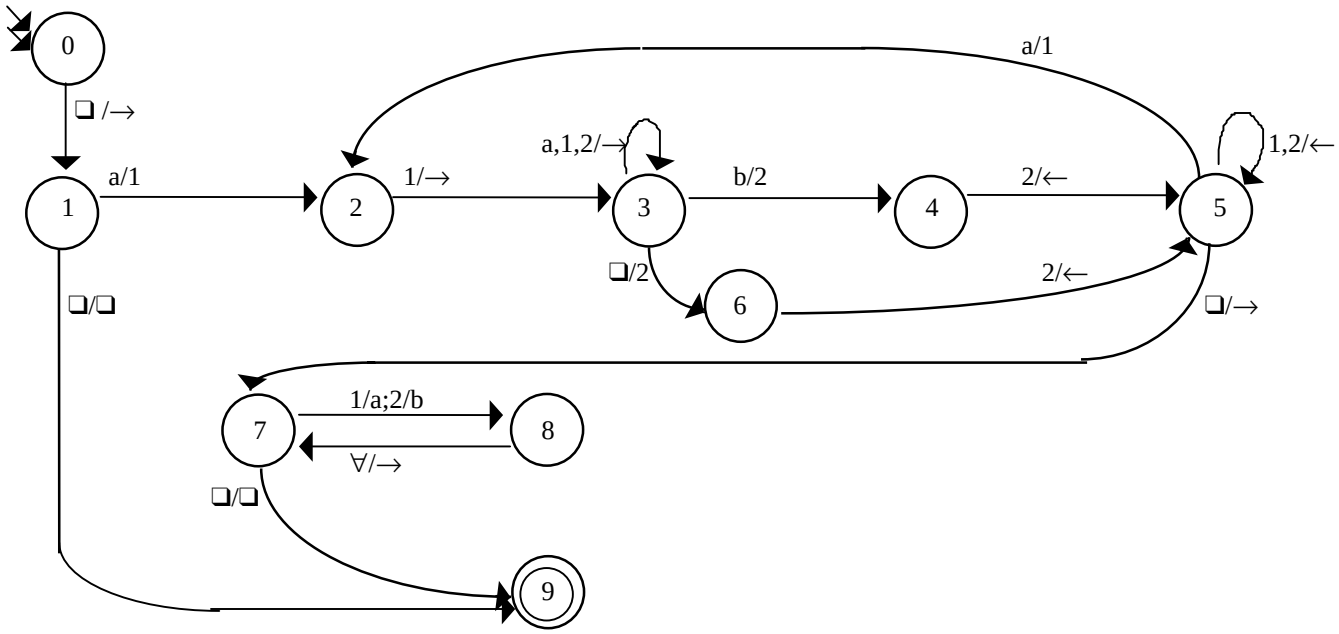
$C_0 \ |-_M^n \ C_n$

A Context-Free Example

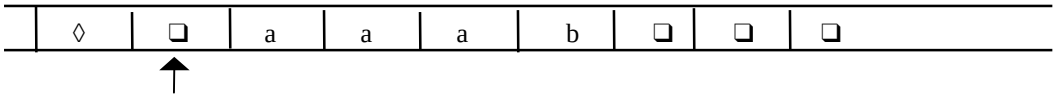
M takes a tape of a's then b's, possibly with more a's, and adds b's as required to make the number of b's equal the number of a's.



K = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Σ = a, b, ◊, ❑, 1, 2
s = 0 H = {9} δ =



An Example Computation



(0, ◊❑aaab) $|-_M$
(1, ◊❑aaab) $|-_M$
(2, ◊❑1aab) $|-_M$
(3, ◊❑1aab) $|-_M$
(3, ◊❑1aab) $|-_M$
(3, ◊❑1aab) $|-_M$
(4, ◊❑1aa2) $|-_M$
…

Notes on Programming

The machine has a strong procedural feel.

It's very common to have state pairs, in which the first writes on the tape and the second move. Some definitions allow both actions at once, and those machines will have fewer states.

There are common idioms, like scan left until you find a blank.

Even a very simple machine is a nuisance to write.

A Notation for Turing Machines

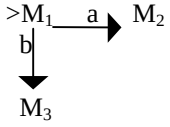
(1) Define some basic machines

- Symbol writing machines
  For each $a \in \Sigma - \{\lozenge\}$, define M_a, written just a, = $(\{s, h\}, \Sigma, \delta, s, \{h\})$,
  for each $b \in \Sigma - \{\lozenge\}$, $\delta(s, b) = (h, a)$
  $\delta(s, \lozenge) = (s, \rightarrow)$
  Example:
  a writes an a

- Head moving machines
  For each $a \in \{\leftarrow, \rightarrow\}$, define M_a, written R($\rightarrow$) and L($\leftarrow$):
  for each $b \in \Sigma - \{\lozenge\}$, $\delta(s, b) = (h, a)$
  $\delta(s, \lozenge) = (s, \rightarrow)$
  Examples:
  R moves one square to the right
  aR writes an a and then moves one square to the right.

A Notation for Turing Machines, Cont'd

(2) The rules for combining machines: as with FSMs



- Start in the start state of M_1.
- 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

M_1 —a→ M_2 (a and b) becomes M_1 —a, b→ M_2

M_1 —all elems of Σ→ M_2 becomes M_1 → M_2 (or M_1M_2)

MM becomes M^2

M_1 —all elems of Σ except a→ M_2 becomes M_1 —x ≠ a→ M_2

and x takes on the value of the current square

M_1 —a, b→ M_2 becomes M_1 —x = a, b→ M_2

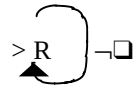
and x takes on the value of the current square

M —x ? y→ M_2

if x = y then take the transition

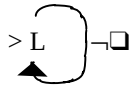
e.g., > —x ≠ ❑→ Rx if the current square is not blank, go right and copy it.

Some Useful Machines



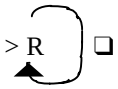
find the first blank square to the right of the current square

$R_{❑}$



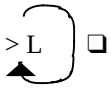
find the first blank square to the left of the current square

$L_{❑}$



find the first nonblank square to the right of the current square

$R_{¬❑}$



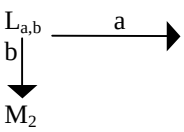
find the first nonblank square to the left of the current square

$L_{¬❑}$

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



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

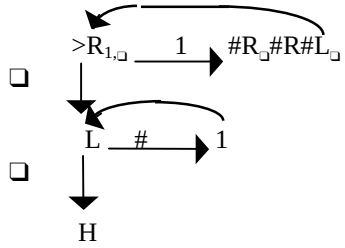
$L_{x=a,b}$ find the first occurrence of a or b to the left of the current square and set x to the value found

$L_{x=a,b}Rx$ 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

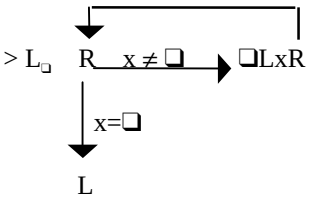
Example: ◊ ❑111❑❑❑❑❑❑❑❑❑❑❑❑



A Shifting Machine $S_{\leftarrow}$

Input: ❑❑w❑
Output: ❑w❑

Example: ❑❑abba❑❑❑❑❑❑❑❑❑❑❑❑



Computing with Turing Machines

Read K & S 4.2.
Do Homework 18.

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, ◊❑w)

A recognizing Turing machine M must have two halting states: y and n
Any configuration of M whose state is:

y is an accepting configuration
n is a rejecting configuration

Let Σ_0, the input alphabet, be a subset of Σ_M-{❑,◊}
Then M **decides** a language $L \subseteq \Sigma_0^*$ iff for any string

$w \in \Sigma_0^*$ it is true that:

if $w \in L$ then M accepts w, and
if $w \notin L$ then M rejects w.

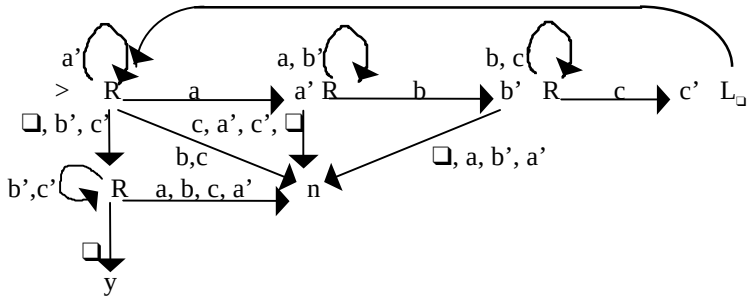
A language L is **recursive** if there is a Turing machine M that decides it.

A Recognition Example

$L = \{a^n b^n c^n : n \geq 0\}$

Example: ◊❑aabbcc❑❑❑❑❑❑❑❑❑

Example: ◊❑aaccb❑❑❑❑❑❑❑❑❑

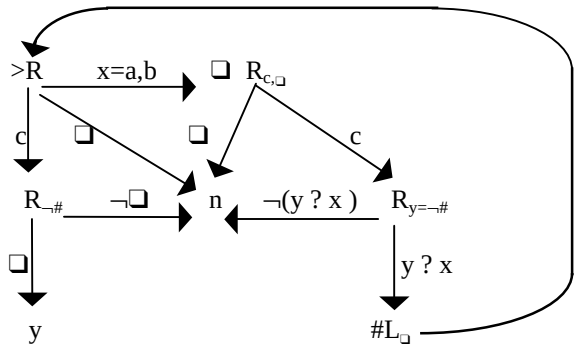


Another Recognition Example

L = {wcw : w ∈ {a, b}*}

Example: ◊❑abbcabb❑❑❑

Example: ◊❑acabb❑❑❑



Do Turing Machines Stop?

FSMs Always halt after n steps, where n is the length of the input. At that point, they either accept or reject.

PDAs Don't always halt, but there is an algorithm to convert any PDA into one that does halt.

Turing machines Can do one of three things:

(1) Halt and accept
(2) Halt and reject
(3) Not halt

And now there is no algorithm to determine whether a given machine always halts.

Computing Functions

Let $\Sigma_0 \subseteq \Sigma - \{◊, ❑\}$ and let $w \in \Sigma_0^*$

Convention: We will write the input on the tape as: ◊❑w❑

The initial configuration of M will then be: (s, ◊❑w)

Define M(w) = y iff:

- M halts if started in the input configuration,
- the tape of M when it halts is ◊❑y❑, and
- $y \in \Sigma_0^*$

Let f be any function from Σ_0^* to Σ_0^*.

We say that M **computes** f if, for all $w \in \Sigma_0^*$, M(w) = f(w)

A function f is **recursive** if there is a Turing machine M that computes it.

Example of Computing a Function

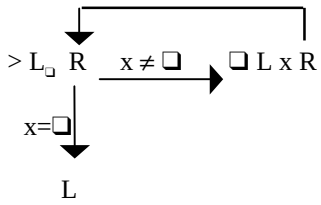
f(w) = ww

Input: ◊❑w❑❑❑❑❑❑ Output: ◊❑ww❑

Define the copy machine C: ◊❑w❑❑❑❑❑❑ → ◊❑w❑w❑

Remember the $S_{\leftarrow}$ machine:

◊❑w❑w❑ → ◊❑ww❑



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

Computing Numeric Functions

We say that a Turing machine M computes a function f from N^k to N provided that

$$num(M(n_1;n_2;\ldots n_k)) = f(num(n_1), \ldots num(n_k))$$

Example: Succ(n) = n + 1

We will represent n in binary. So $n \in 0 \cup 1\{0,1\}^*$

Input: ◊❑n❑❑❑❑❑❑ Output: ◊❑n+1❑
◊❑1111❑❑❑❑ Output: ◊❑10000❑

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.

Recursively Enumerable and Recursive Languages

Read K & S 4.5.

Recursively Enumerable Languages

Let Σ_0, the input alphabet to a Turing machine M, be a subset of $\Sigma_M - \{\square, \lozenge\}$

Let $L \subseteq \Sigma_0^*$.

M **semidecides** L iff

for any string $w \in \Sigma_0^*$,

$w \in L \Rightarrow$ M halts on input w

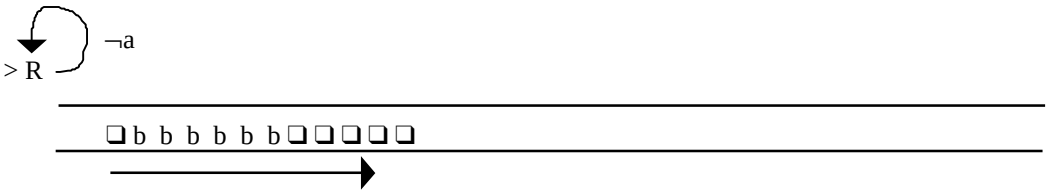
$w \notin L \Rightarrow$ M does not halt on input w

$M(w) = \uparrow$

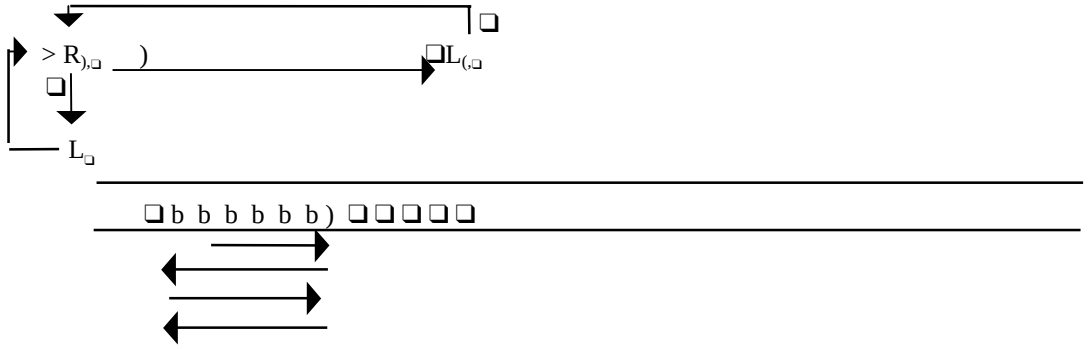
L is **recursively enumerable** iff there is a Turing machine that semidecides it.

Examples of Recursively Enumerable Languages

L = {w ∈ {a, b}* : w contains at least one a}



L = {w ∈ {a, b, (,) }* : w contains at least one set of balanced parentheses}



Recursively Enumerable Languages that Aren't Also Recursive

A Real Life Example:

L = {w ∈ {friends} : w will answer the message you've just sent out}

Theoretical Examples

L = {Turing machines that halt on a blank input tape}

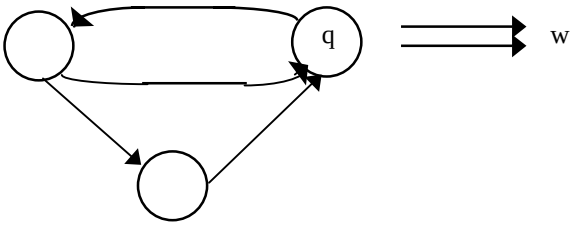
Theorems with valid proofs.

Why Are They Called Recursively Enumerable Languages?

Enumerate means list.

We say that Turing machine M **enumerates** the language L iff, for some fixed state q of M,

$L = \{w : (s, \lozenge\underline{\square}) \mid\!-_M^* (q, \lozenge\square w)\}$



A language is **Turing-enumerable** iff there is a Turing machine that enumerates it.

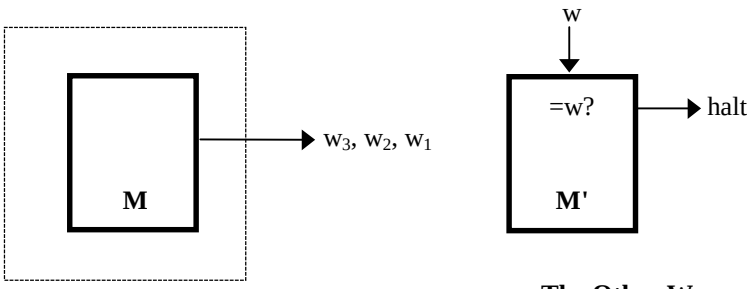
Note that q is not a halting state. It merely signals that the current contents of the tape should be viewed as a member of L.

Recursively Enumerable and Turing Enumerable

Theorem: A language is recursively enumerable iff it is Turing-enumerable.

Proof that Turing-enumerable implies RE: Let M be the Turing machine that enumerates L. We convert M to a machine M' that semidecides L:

1. Save input w.
2. Begin enumerating L. Each time an element of L is enumerated, compare it to w. If they match, accept.



The Other Way

Proof that RE implies Turing-enumerable:

If $L \subseteq \Sigma^*$ is a recursively enumerable language, then there is a Turing machine M that semidecides L.

A procedure to enumerate all elements of L:

Enumerate all $w \in \Sigma^*$ lexicographically.

e.g., ε, a, b, aa, ab, ba, bb, …

As each string w_i is enumerated:

1. Start up a copy of M with w_i as its input.
2. Execute one step of each M_i initiated so far, excluding only those that have previously halted.
3. Whenever an M_i halts, output w_i.

ε [1]					
ε [2]	a [1]				
ε [3]	a [2]	b [1]			
ε [4]	a [3]	b [2]	aa [1]		
ε [5]	a [4]	b [3]	aa [2]	ab [1]	
ε [6]	a [5]		aa [3]	ab [2]	ba [1]

Every Recursive Language is Recursively Enumerable

If L is recursive, then there is a Turing machine that decides it.

From M, we can build a new Turing machine M' that semidecides L:

1. Let n be the reject (and halt) state of M.
2. Then add to δ'
   ((n, a), (n, a)) for all a ∈ Σ

What about the other way around?
Not true. There are recursively enumerable languages that are not recursive.

The Recursive Languages Are Closed Under Complement

Proof: (by construction) If L is recursive, then there is a Turing machine M that decides L.

We construct a machine M' to decide $\overline{L}$ by taking M and swapping the roles of the two halting states y and n.

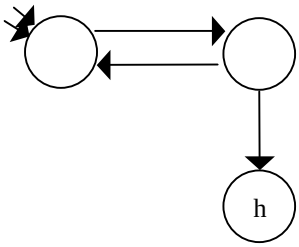
M: M':

This works because, by definition, M is

- deterministic
- complete

Are the Recursively Enumerable Languages Closed Under Complement?

M: M':



Lemma: There exists at least one language L that is recursively enumerable but not recursive.

Proof that M' doesn't exist: Suppose that the RE languages were closed under complement. Then if L is RE, $\overline{L}$ would be RE. If that were true, then $\overline{L}$ would also be recursive because we could construct M to decide it:

1. Let T_1 be the Turing machine that semidecides L.
2. Let T_2 be the Turing machine that semidecides $\overline{L}$.
3. Given a string w, fire up both T_1 and T_2 on w. Since any string in Σ* must be in either L or $\overline{L}$, one of the two machines will eventually halt. If it's T_1, accept; if it's T_2, reject.

But we know that there is at least one RE language that is not recursive. Contradiction.

Recursive and RE Languages

Theorem: A language is recursive iff both it and its complement are recursively enumerable.

Proof:

- L recursive implies L and ¬L are RE: Clearly L is RE. And, since the recursive languages are closed under complement, ¬L is recursive and thus also RE.
- L and ¬L are RE implies L recursive: Suppose L is semidecided by M1 and ¬L is semidecided by M2. We construct M to decide L by using two tapes and simultaneously executing M1 and M2. One (but not both) must eventually halt. If it's M1, we accept; if it's M2 we reject.

Lexicographic Enumeration

We say that M **lexicographically enumerates** L if M enumerates the elements of L in lexicographic order. A language L is **lexicographically Turing-enumerable** iff there is a Turing machine that lexicographically enumerates it.

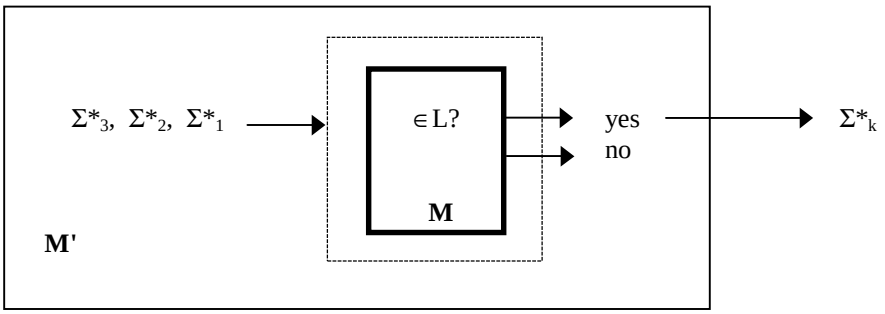
Example: $L = \{a^n b^n c^n\}$

Lexicographic enumeration:

Proof

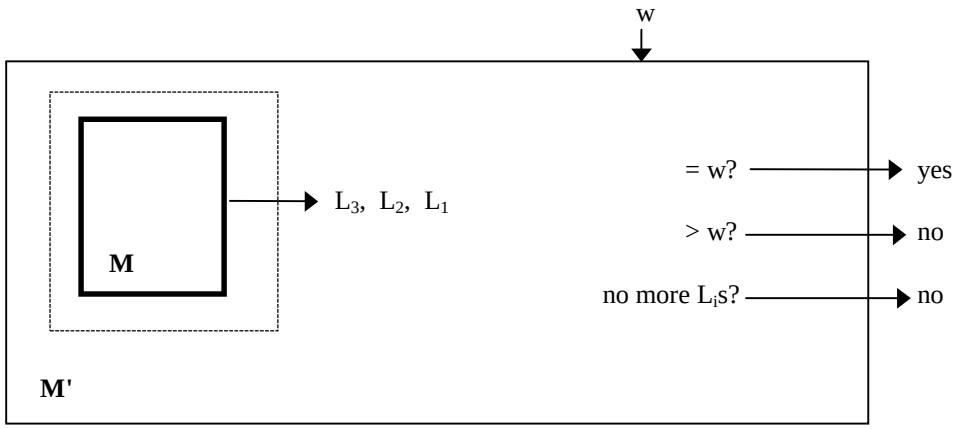
Theorem: A language is recursive iff it is lexicographically Turing-enumerable.

Proof that recursive implies lexicographically Turing enumerable: Let M be a Turing machine that decides L. Then M' lexicographically generates the strings in Σ* and tests each using M. It outputs those that are accepted by M. Thus M' lexicographically enumerates L.



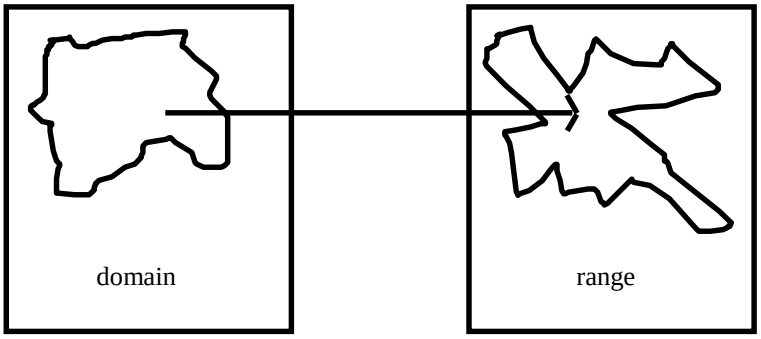
Proof, Continued

Proof that lexicographically Turing enumerable implies recursive: Let M be a Turing machine that lexicographically enumerates L. Then, on input w, M' starts up M and waits until either M generates w (so M' accepts), M generates a string that comes after w (so M' rejects), or M halts (so M' rejects). Thus M' decides L.



Partially Recursive Functions

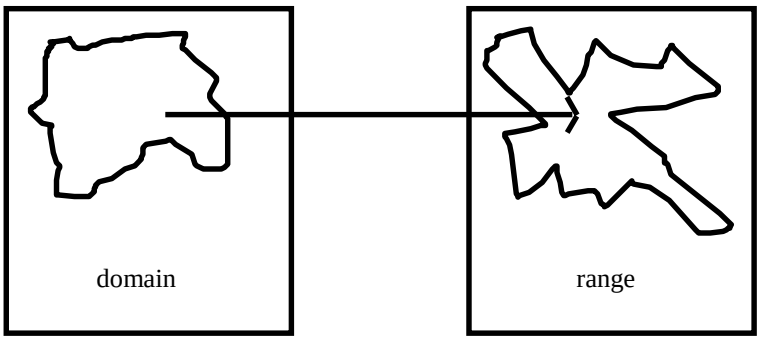
	Languages	Functions
Tm always halts	**recursive**	**recursive**
Tm halts if yes	**recursively enumerable**	**?**



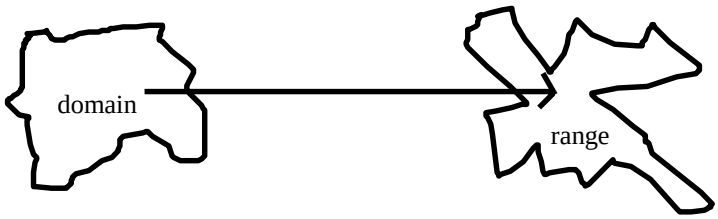
Suppose we have a function that is not defined for all elements of its domain.

Example: f: N $\rightarrow$ N, f(n) = n/2

Partially Recursive Functions



One solution: Redefine the domain to be exactly those elements for which f is defined:



But what if we don't know? What if the domain is not a recursive set (but it is recursively enumerable)? Then we want to define the domain as some larger, recursive set and say that the function is partially recursive. There exists a Turing machine that halts if given an element of the domain but does not halt otherwise.

Language
Summary

IN		OUT
Semidecidable Enumerable Unrestricted grammar	**Recursively Enumerable**	
Decision procedure Lexicicographically enumerable Complement is recursively enumer.	**Recursive**	Diagonalization Reduction
CF grammar PDA Closure	**Context Free**	Pumping Closure
Regular expression FSM Closure	**Regular**	Pumping Closure