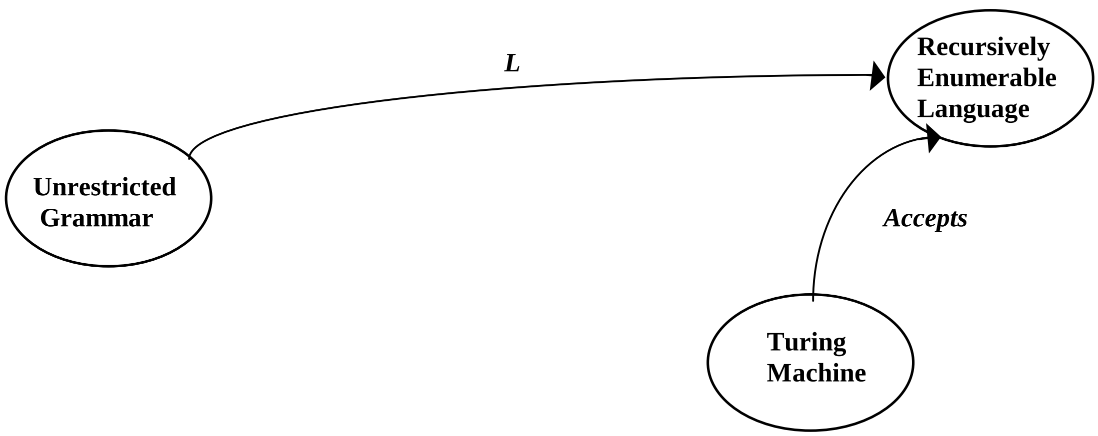
Turing Machines

Read K & S 4.1.
Do Homework 17.

Grammars, Recursively Enumerable Languages, and Turing Machines

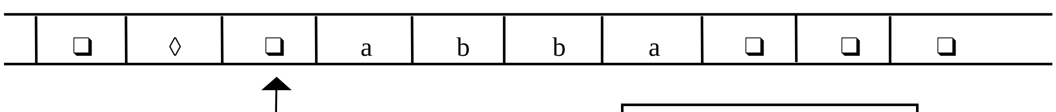


Turing Machines

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

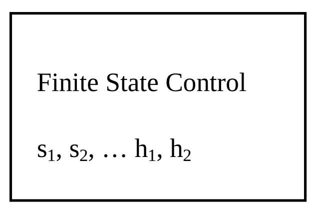
- powerful enough to describe all computable things
  - unlike FSMs and PDAs
- simple enough that we can reason formally about it
  - like FSMs and PDAs
  - unlike real computers

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, Σ, δ, s, H):

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

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

such that

(a) if the input symbol is ◊, the action is →, 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.

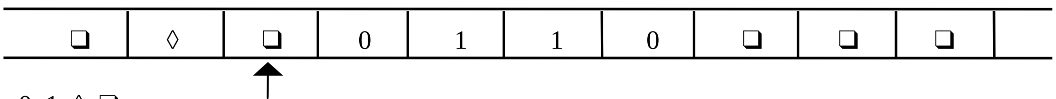
3. δ must be defined for all state, input pairs unless the state is a halt state.

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.

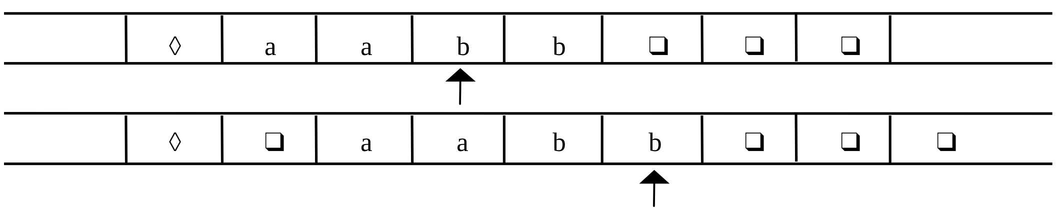
A Simple Example

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

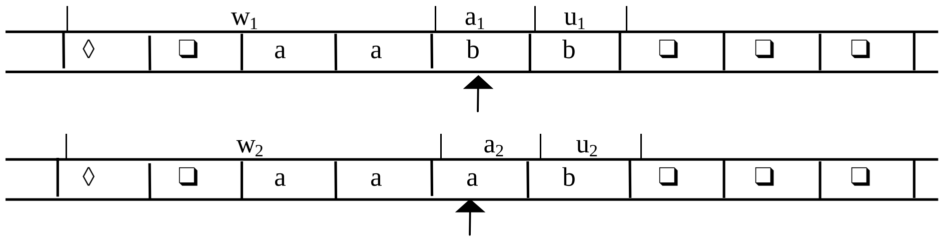
The input after the scanned square may be empty, but it may not end with a blank. We assume the entire tape to the right of the input is filled with blanks.

(1) (q, ◊aab, b) = (q, ◊aa$\underline{\text{b}}$b)
(2) (h, ◊❑aabb, ε) = (h, ◊❑aab$\underline{\text{b}}$) 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)



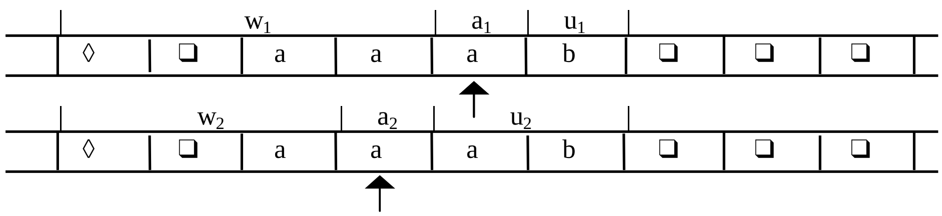
◊❏aa$\underline{\text{b}}$b

◊❏aa$\underline{\text{a}}$b

Yields, Continued

(2) $b = \leftarrow$, $w_1 = w_2a_2$, and either

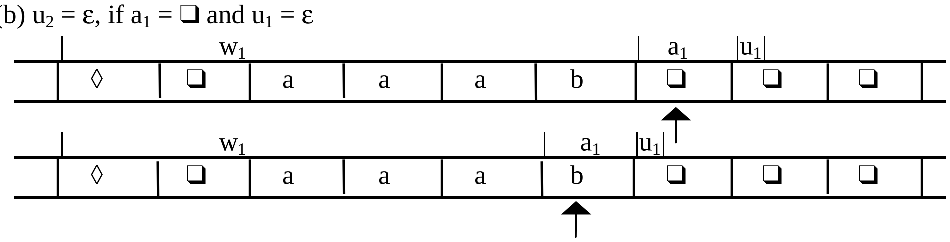
(a) $u_2 = a_1u_1$, if $a_1 \neq$ ❏ or $u_1 \neq \varepsilon$,



◊❏aa$\underline{\text{a}}$b

◊❏a$\underline{\text{a}}$ab

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



◊❏aaab$\underline{❏}$

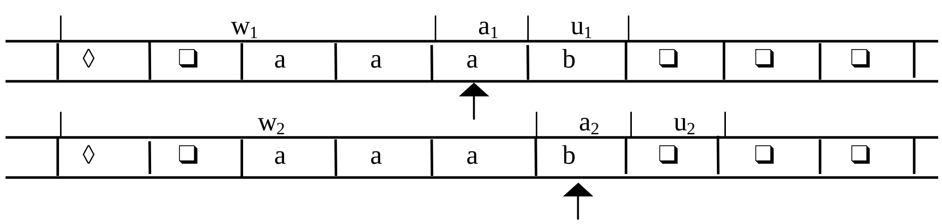
◊❏aaa$\underline{\text{b}}$

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

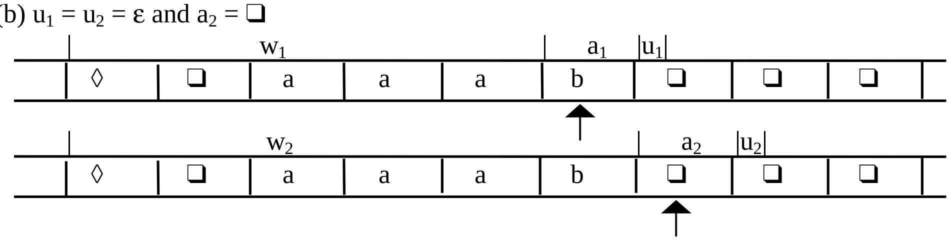
(a) $u_1 = a_2u_2$



◊❏aa$\underline{\text{a}}$b

◊❏aaa$\underline{\text{b}}$

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



◊❏aaa$\underline{\text{b}}$

◊❏aaab$\underline{❏}$

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 $\mathbf{|\!-_M{}^*}$ 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

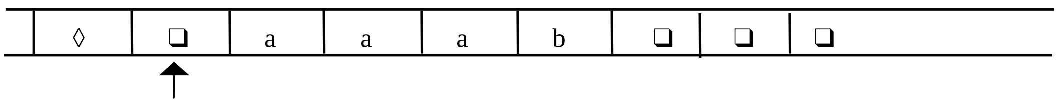
$$C_0 \;|\!-_M\; C_1 \;|\!-_M\; C_2 \;|\!-_M \ldots |\!-_M\; C_n.$$

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

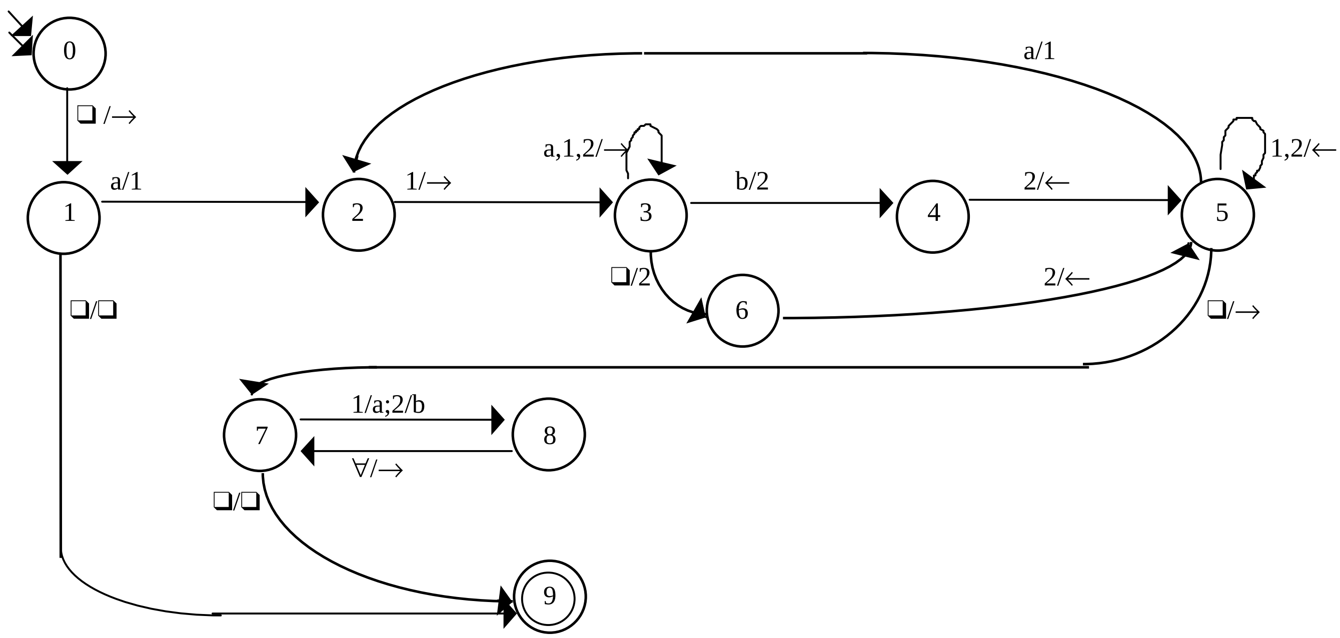
$$C_0 \;|\!-_M{}^n\; C_n$$

A Context-Free Example

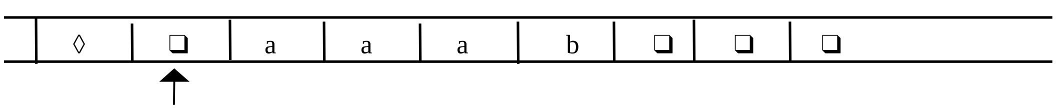
M takes a tape of a's then b's, possibly with more a's, and adds b's as required to make the number of b's equal the number of a's.



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



An Example Computation



$(0, \Diamond\underline{\square}aaab) \;|\!-_M$
$(1, \Diamond\square\underline{a}aab) \;|\!-_M$
$(2, \Diamond\square\underline{1}aab) \;|\!-_M$
$(3, \Diamond\square 1\underline{a}ab) \;|\!-_M$
$(3, \Diamond\square 1a\underline{a}b) \;|\!-_M$
$(3, \Diamond\square 1aa\underline{b)} \;|\!-_M$
$(4, \Diamond\square 1aa\underline{2)} \;|\!-_M$
...

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 $a \in \Sigma - \{\Diamond\}$, define M_a, written just a, $= (\{s, h\}, \Sigma, \delta, s, \{h\})$,

  for each $b \in \Sigma - \{\Diamond\}$, $\delta(s, b) = (h, a)$

  $\delta(s, \Diamond) = (s, \rightarrow)$

  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 $b \in \Sigma - \{\Diamond\}$, $\delta(s, b) = (h, a)$

  $\delta(s, \Diamond) = (s, \rightarrow)$

  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

$$\begin{tikzcd} >M_1 \arrow[r, "a"] \arrow[d, "b"'] & M_2 \\ M_3 & \end{tikzcd}$$

- 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

$M_1 \overset{a}{\underset{b}{\rightrightarrows}} M_2$	becomes	$M_1 \xrightarrow{a,\ b} M_2$
$M_1 \xrightarrow{\text{all elems of } \Sigma} M_2$	becomes	$M_1 \longrightarrow M_2$ or M_1M_2
MM	becomes	M^2
$M_1 \xrightarrow[\text{except a}]{\text{all elems of } \Sigma} M_2$	becomes	$M_1 \xrightarrow{x \neq a} M_2$ and x takes on the value of the current square
$M_1 \xrightarrow{a,\ b} M_2$	becomes	$M_1 \xrightarrow{x = a,\ b} M_2$ and x takes on the value of the current square
		$M \xrightarrow{x\ ?\ y} M_2$ if x = y then take the transition

e.g., $> \xrightarrow{x \neq \square} Rx$ if the current square is not blank, go right and copy it.

Some Useful Machines

$$\begin{tikzcd} > R \arrow[loop right, "\neg\square"] \end{tikzcd}$$

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

$R_{\square}$

$$\begin{tikzcd} > L \arrow[loop right, "\neg\square"] \end{tikzcd}$$

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

$L_{\square}$

$$\begin{tikzcd} > R \arrow[loop right, "\square"] \end{tikzcd}$$

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

$R_{\neg\square}$

$$\begin{tikzcd} > L \arrow[loop right, "\square"] \end{tikzcd}$$

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

$L_{\neg\square}$

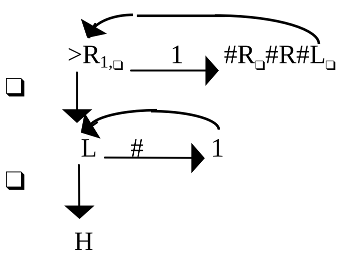
More Useful Machines

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
$\begin{array}{l} L_{a,b} \xrightarrow{\ a\ } M_1 \\ \downarrow b \\ M_2 \end{array}$	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: ◊<u>❑</u>w w ∈ {1}*
Output: ◊<u>❑</u>w^3

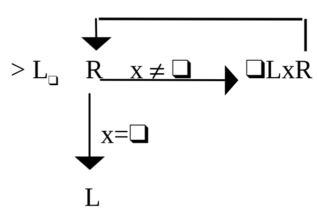
Example: ◊ <u>❑</u>111❑❑❑❑❑❑❑❑❑❑❑❑❑



A Shifting Machine S←

Input: ❑❑w<u>❑</u>
Output: ❑w<u>❑</u>

Example: ❑❑abba<u>❑</u>❑❑❑❑❑❑❑❑❑❑❑❑



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.

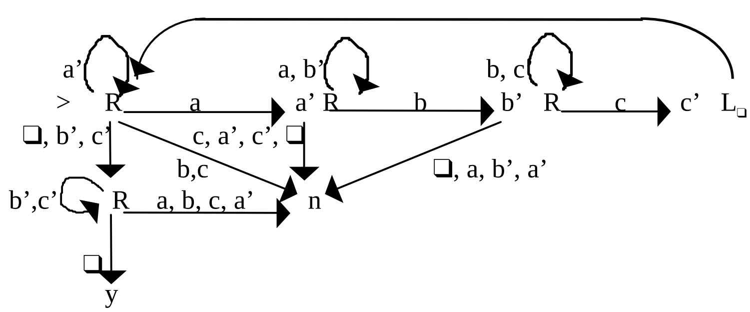
A language L is **recursive** if there is a Turing machine M that decides it.

A Recognition Example

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

Example: ◊❑aabbcc❑❑❑❑❑❑❑❑❑

Example: ◊❑aaccb❑❑❑❑❑❑❑❑❑

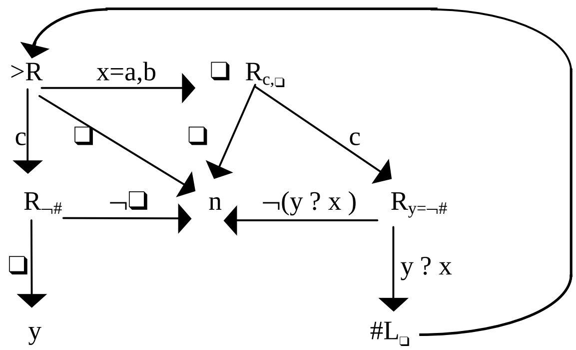


Another Recognition Example

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 ∈ Σ_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 ∈ Σ_0*

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

We say that M **computes** f if, for all w ∈ Σ_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

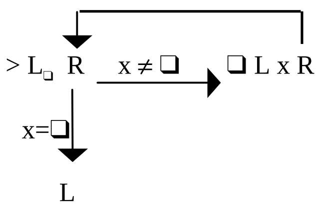
f(w) = ww

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 Σ_M - {❑, ◊}

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

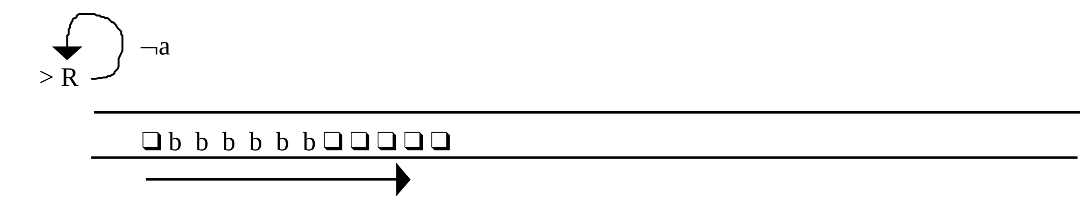
$w \notin L \Rightarrow$ M does not halt on input w

$M(w) = \uparrow$

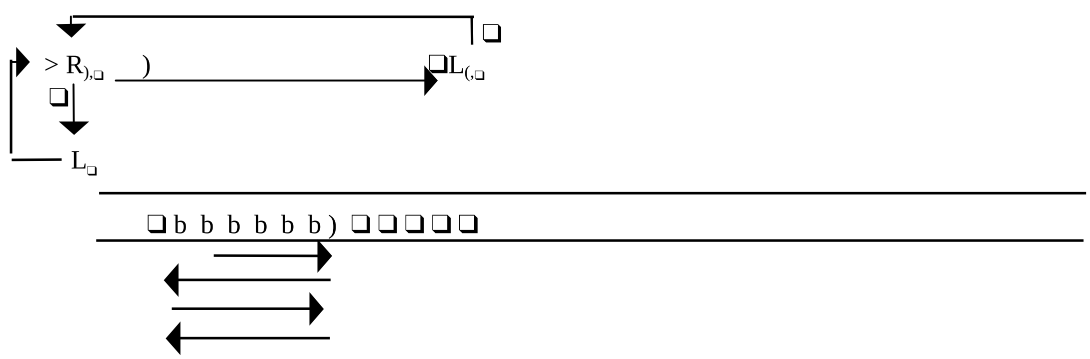
L is **recursively enumerable** iff there is a Turing machine that semidecides it.

Examples of Recursively Enumerable Languages

L = {$w \in$ {a, b}* : w contains at least one a}



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



Recursively Enumerable Languages that Aren't Also Recursive

A Real Life Example:

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

Theoretical Examples

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

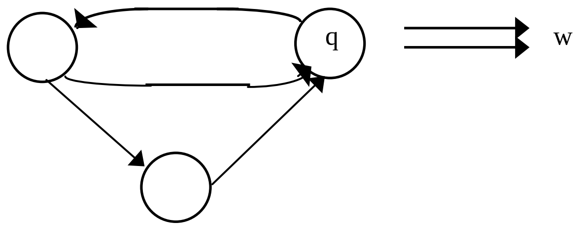
Theorems with valid proofs.

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,

$$L = \{w : (s, \Diamond\underline{\square}) \vdash_M^* (q, \Diamond\underline{\square}w)\}$$



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

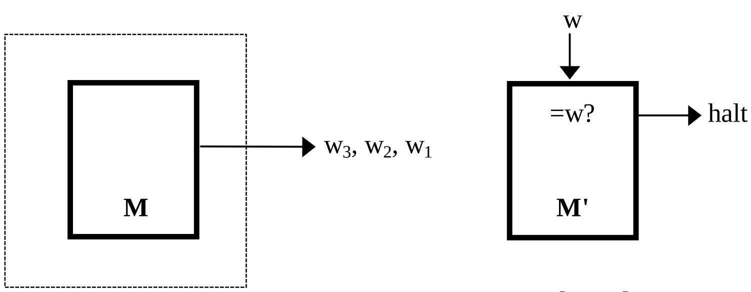
Note that q is not a halting state. It merely signals that the current contents of the tape should be viewed as a member of L.

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]	<u>b [3]</u>	aa [2]	ab [1]	
ε [6]	a [5]		aa [3]	ab [2]	ba [1]

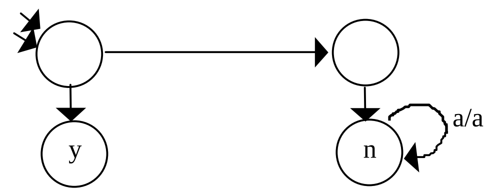
Every Recursive Language is Recursively Enumerable

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



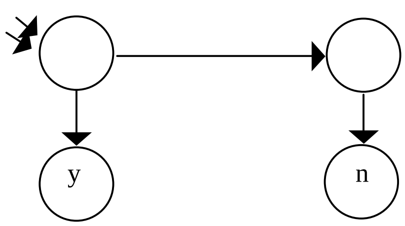
What about the other way around?
Not true. There are recursively enumerable languages that are not recursive.

The Recursive Languages Are Closed Under Complement

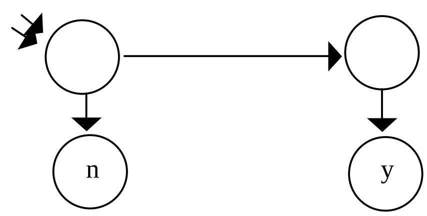
Proof: (by construction) If L is recursive, then there is a Turing machine M that decides L.

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':

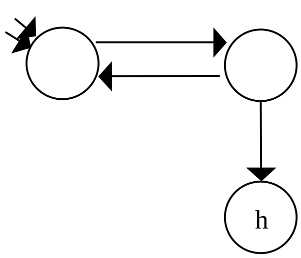


This works because, by definition, M is

- 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.

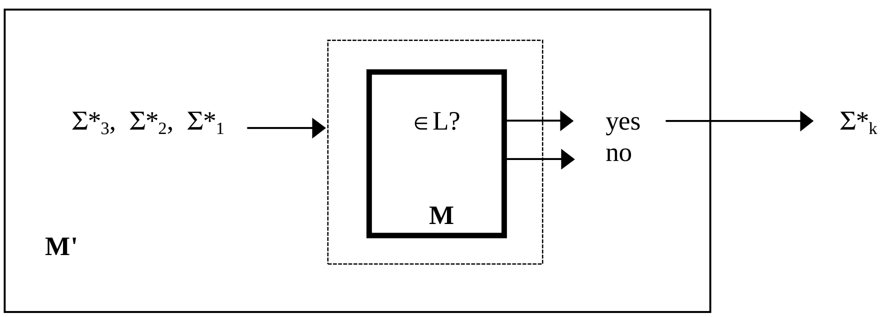
Example: $L = \{a^n b^n c^n\}$

Lexicographic enumeration:

Proof

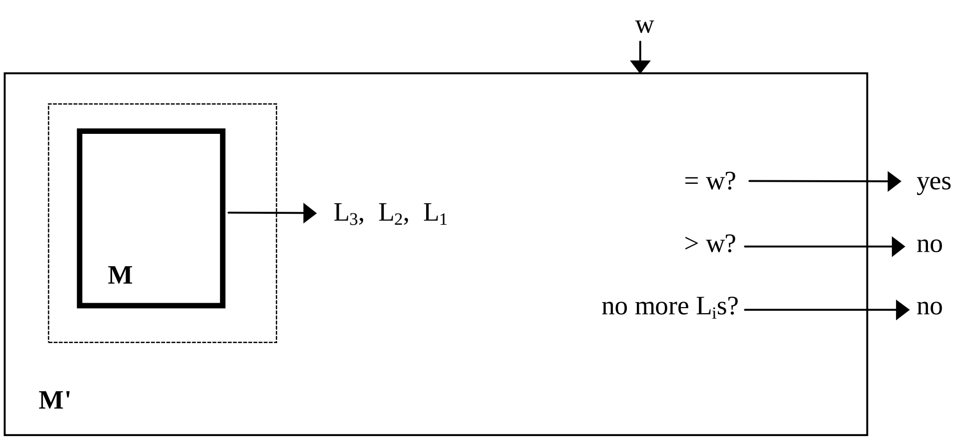
Theorem: A language is recursive iff it is lexicographically Turing-enumerable.

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.



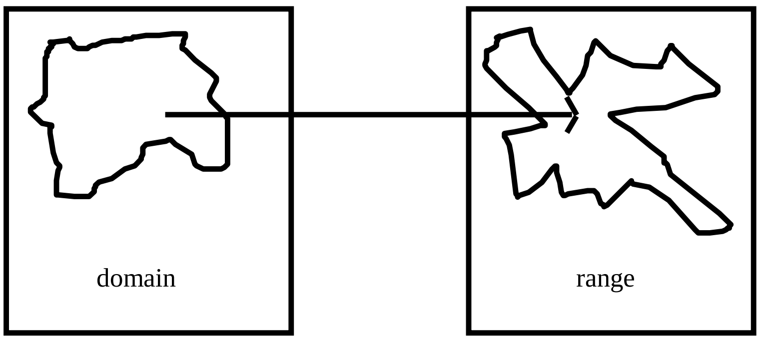
Proof, Continued

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

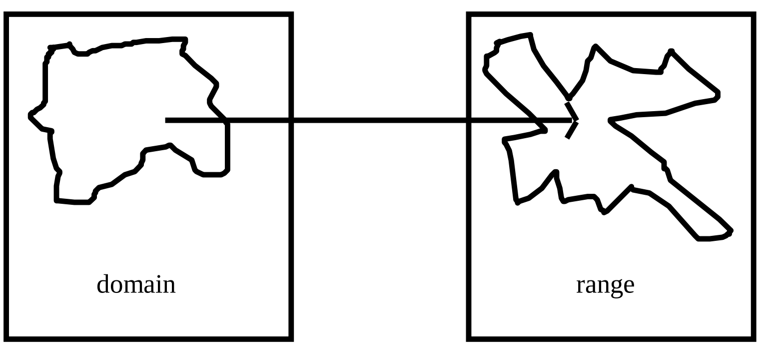
	Languages	Functions
Tm always halts	**recursive**	**recursive**
Tm halts if yes	**recursively enumerable**	**?**



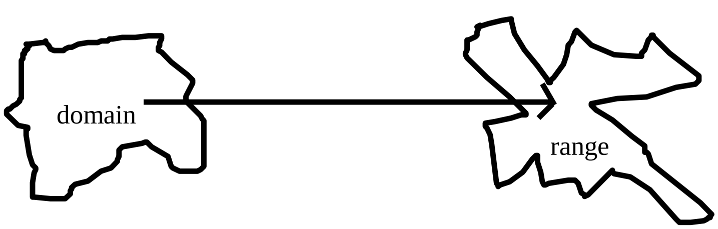
Suppose we have a function that is not defined for all elements of its domain.

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

Partially Recursive Functions

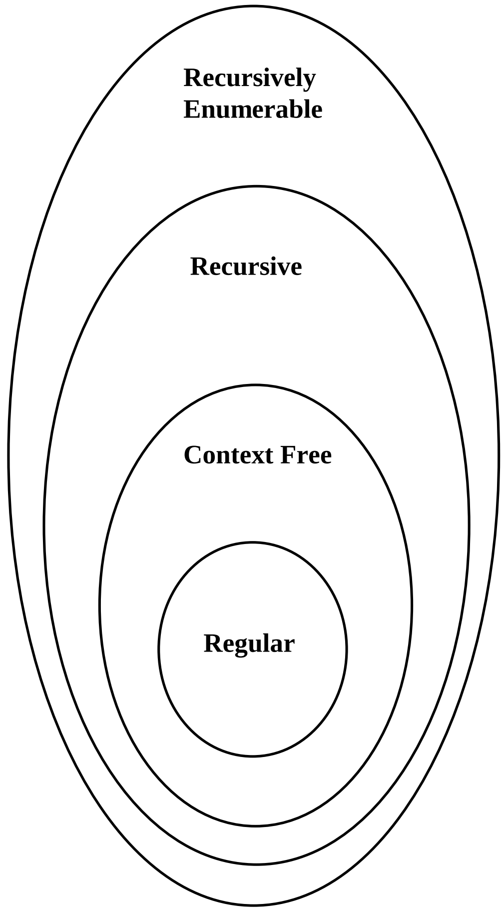


One solution: Redefine the domain to be exactly those elements for which f is defined:



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	
Decision procedure Lexicicographically enumerable Complement is recursively enumer.	Diagonalization Reduction
CF grammar PDA Closure	Pumping Closure
Regular expression FSM Closure	Pumping Closure

Turing Machine Extensions

Read K & S 4.3.1, 4.4.
Do Homework 19.

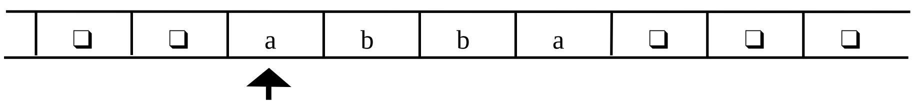
Turing Machine Definitions

An alternative definition of a Turing machine:
$(K, \Sigma, \Gamma, \delta, s, H)$:

Γ is a finite set of allowable tape symbols. One of these is ❑.

Σ is a subset of Γ not including ❑, the input symbols.

δ is a function from:
$K \times \Gamma$ to $K \times (\Gamma - \{❑\}) \times \{\leftarrow, \rightarrow\}$
state, tape symbol, L or R



Example transition: $((s, a), (s, b, \rightarrow))$

Do these Differences Matter?

Remember the goal:

Define a device that is:
- powerful enough to describe all computable things,
- simple enough that we can reason formally about it

Both definitions are simple enough to work with, although details may make specific arguments easier or harder.

But, do they differ in their power?

Answer: No.

Consider the differences:
- One way or two way infinite tape: we're about to show that we can simulate two way infinite with ours.
- Rewrite and move at the same time: just affects (linearly) the number of moves it takes to solve a problem.

Turing Machine Extensions

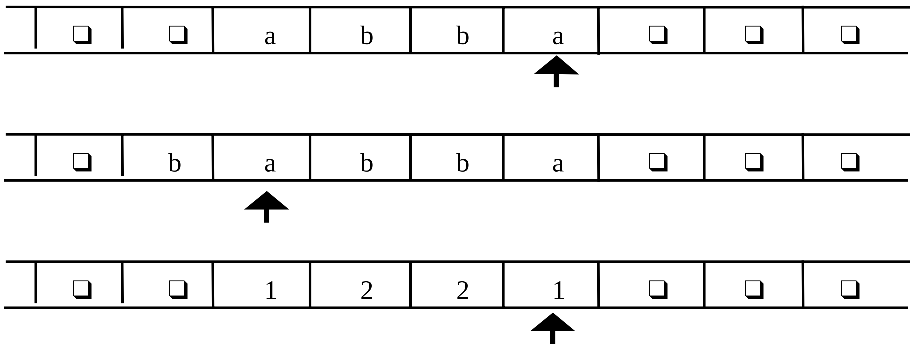
In fact, there are lots of extensions we can make to our basic Turing machine model. They may make it easier to write Turing machine programs, but none of them increase the power of the Turing machine because:

We can show that every extended machine has an equivalent basic machine.

We can also place a bound on any change in the complexity of a solution when we go from an extended machine to a basic machine.

Some possible extensions:
- Multiple tapes
- Two-way infinite tape
- Multiple read heads
- Two dimensional "sheet" instead of a tape
- Random access machine
- Nondeterministic machine

Multiple Tapes



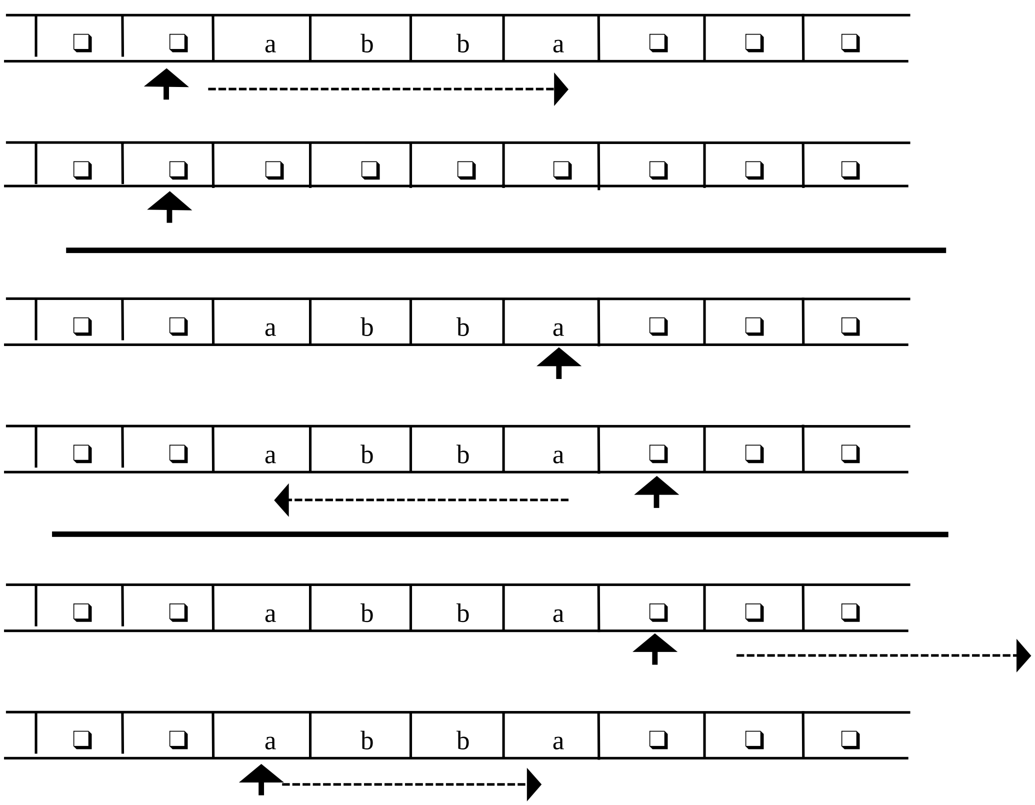
The transition function for a k-tape Turing machine:

$$
\begin{array}{llll}
((K\text{-}H) & , \Sigma_1 & \text{to} & (K, \ \Sigma_{1'} \cup \{\leftarrow, \rightarrow\} \\
 & , \Sigma_2 & & , \Sigma_{2'} \cup \{\leftarrow, \rightarrow\} \\
 & , \ . & & , \ . \\
 & , \ . & & , \ . \\
 & , \ \Sigma_k) & & , \Sigma_{k'} \cup \{\leftarrow, \rightarrow\})
\end{array}
$$

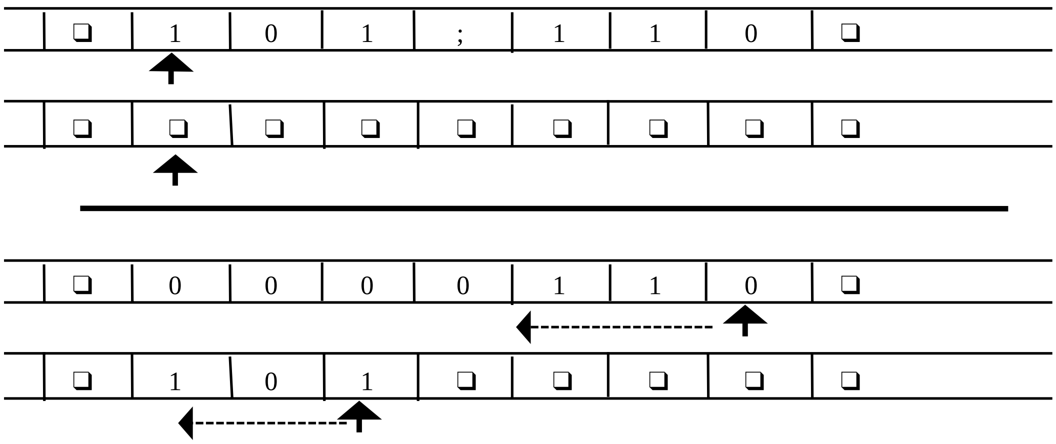
Input: input as before on tape 1, others blank
Output: output as before on tape 1, others ignored

An Example of a Two Tape Machine

Copying a string



Another Two Tape Example - Addition



Adding Tapes Adds No Power

Theorem: Let M be a k-tape Turing machine for some $k \geq 1$. Then there is a standard Turing machine M' where $\Sigma \subseteq \Sigma'$, and such that:

- For any input string x, M on input x halts with output y on the first tape iff M' on input x halts at the same halting state and with the same output on its tape.
- If, on input x, M halts after t steps, then M' halts after a number of steps which is $O(t \cdot (|x| + t))$.

Proof: By construction

<table>
<tr><td rowspan="4">◊</td><td>◊</td><td>❑</td><td>a</td><td>b</td><td>a</td><td>❑</td><td>❑</td><td rowspan="4">❑</td><td rowspan="4">❑</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>◊</td><td>a</td><td>b</td><td>b</td><td>a</td><td>b</td><td>a</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Alphabet (Σ') of M' = $\Sigma \cup (\Sigma \times \{0, 1\})^k$

e.g., ◊, (◊, 0, ◊, 0), (❑, 0, a, 1)

The Operation of M'

<table>
<tr><td rowspan="4">◊</td><td>◊</td><td>❑</td><td>a</td><td>b</td><td>a</td><td>❑</td><td>❑</td><td rowspan="4">❑</td><td rowspan="4">❑</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>◊</td><td>a</td><td>b</td><td>b</td><td>a</td><td>b</td><td>a</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

1. Set up the multitrack tape:
   1) Shift input one square to right, then set up each square appropriately.
2. Simulate the computation of M until (if) M would halt: (start each step to the right of the divided tape)
   1) Scan left and store in the state the k-tuple of characters under the read heads. Move back right.
   2) Scan left and update each track as required by the transitions of M. Move back right.
      i) If necessary, subdivide a new square into tracks.
3. When M would halt, reformat the tape to throw away all but track 1, position the head correctly, then go to M's halt state.

How Many Steps Does M' Take?

Let: x be the input string, and
t be the number of steps it takes M to execute.

Step 1 (initialization) $O(|x|)$

Step 2 (computation)

Number of passes = t

Work at each pass: 2.1 = $2 \cdot$ (length of tape)
$= 2 \cdot (|x| + 2 + t)$
2.2 = $2 \cdot (|x| + 2 + t)$

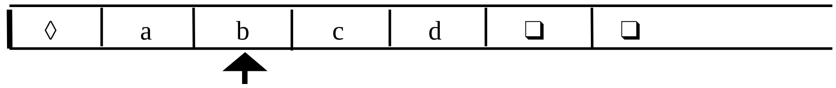
Total = $O(t \cdot (|x| + t))$

Step 3 (clean up) O(length of tape)

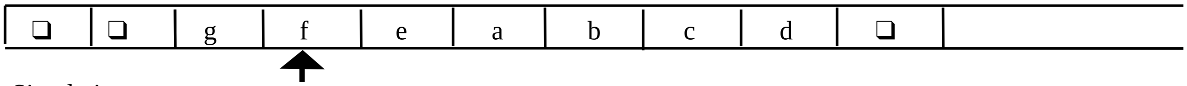
Total = $O(t \cdot (|x| + t))$

Two-Way Infinite Tape

Our current definition:



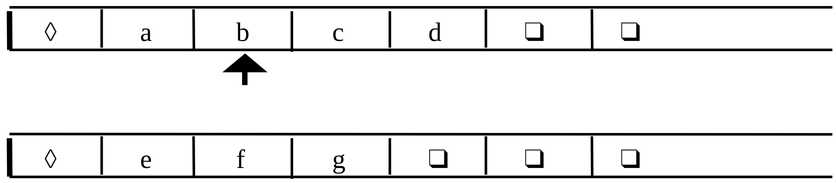
Proposed definition:



Simulation:

Track 1

Track 2



Simulating a PDA

The components of a PDA:

- Finite state controller
- Input tape
- Stack

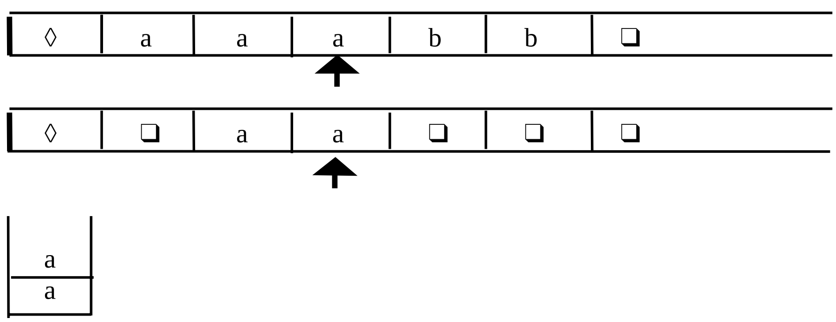
The simulation:

- Finite state controller:
- Input tape:
- Stack:

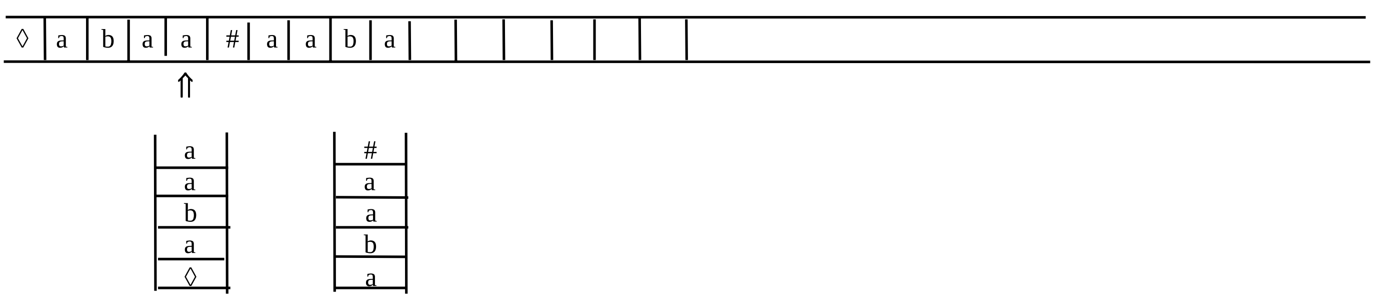
Track 1
(Input)

Track 2

Corresponding to



Simulating a Turing Machine with a PDA with Two Stacks



Random Access Turing Machines

A random access Turing machine has:

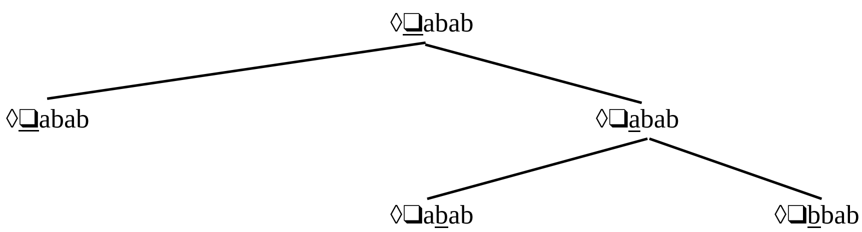
- a fixed number of registers
- a finite length program, composed of instructions with operators such as read, write, load, store, add, sub, jump
- a tape
- a program counter

Theorem: Standard Turing machines and random access Turing machines compute the same things. Furthermore, the number of steps it takes a standard machine is bounded by a polynomial in the number of steps it takes a random access machine.

Nondeterministic Turing Machines

A **nondeterministic** Turing machine is a quintuple (K, Σ, Δ, s, H)
where K, Σ, s, and H are as for standard Turing machines, and Δ is a ***subset*** of

$((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$



What does it mean for a nondeterministic Turing machine to compute something?

- Semidecides - at least one halts.
- Decides - ?
- Computes - ?

Nondeterministic Semideciding

Let M = (K, Σ, Δ, s, H) be a nondeterministic Turing machine. We say that M **accepts** an input

$w \in (\Sigma - \{◊, ❑\})^*$ iff

(s, ◊❑w) yields a least one accepting configuration.

We say that M **semidecides** a language

$L \subseteq (\Sigma - \{◊, ❑\})^*$ iff

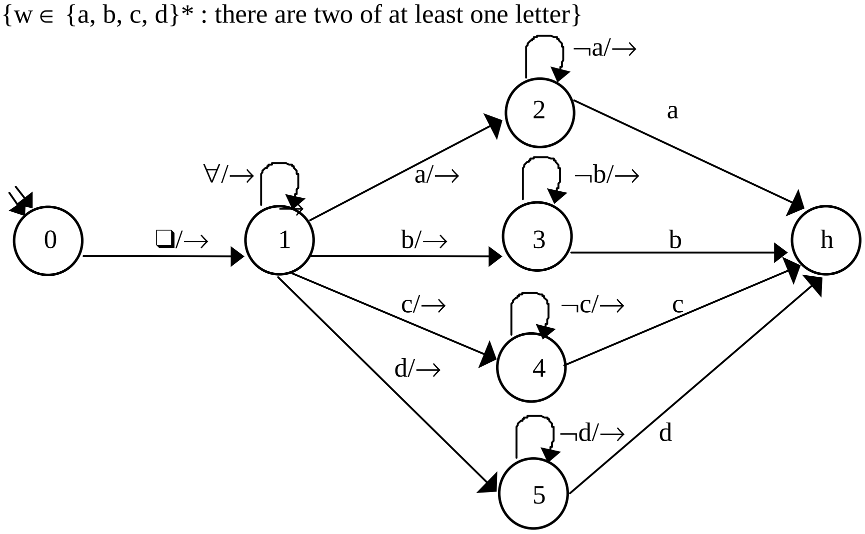
for all $w \in (\Sigma - \{◊, ❑\})^*$:

$w \in L$ iff

(s, ◊❑w) yields a least one halting configuration.

An Example

L = {w ∈ {a, b, c, d}* : there are two of at least one letter}



Nondeterministic Deciding and Computing

M **decides** a language L if, for all $w \in (\Sigma - \{\Diamond, \square\})^*$:
1. all of M's computations on w halt, and
2. $w \in L$ iff ***at least one*** of M's computations accepts.

M **computes** a function f if, for all $w \in (\Sigma - \{\Diamond, \square\})^*$:
1. all of M's computations halt, and
2. ***all*** of M's computations result in f(w)

Note that all of M's computations halt iff:

There is a natural number N, depending on M and w, such that there is no configuration C satisfying

$(s, \Diamond\underline{\square}w) \vdash_M^N C$.

An Example of Nondeterministic Deciding

$L = \{w \in \{0, 1\}^* : w$ is the binary encoding of a composite number$\}$

M decides L by doing the following on input w:

1. Nondeterministically choose two binary numbers $1 < p, q$, where $|p|$ and $|q| \leq |w|$, and write them on the tape, after w, separated by ;.

   $\Diamond\underline{\square}110011;111;1111\square\square$

2. Multiply p and q and put the answer, A, on the tape, in place of p and q.

   $\Diamond\underline{\square}110011;1011111\square\square$

3. Compare A and w. If equal, go to y. Else go to n.

Equivalence of Deterministic and Nondeterministic Turing Machines

Theorem: If a nondeterministic Turing machine M semidecides or decides a language, or computes a function, then there is a standard Turing machine M' semideciding or deciding the same language or computing the same function.

Note that while nondeterminism doesn't change the computational power of a Turing Machine, it can exponentially increase its speed!

Proof: (by construction)
For semideciding: We build M', which runs through all possible computations of M. If one of them halts, M' halts

Recall the way we did this for FSMs: simulate being in a combination of states.

Will this work here?

What about: Try path 1. If it accepts, accept. Else
Try path 2. If it accepts, accept. Else
•
•

The Construction

At any point in the operation of a nondeterministic machine M, the maximum number of branches is

$$r = \underset{\text{states}}{|K|} \cdot \underset{\text{actions}}{(|\Sigma| + 2)}$$

So imagine a table:

	1	2	3		r
(q1,σ1)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)
(q1,σ2)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)
(q1,σn)					
(q2,σ1)					
(q|K|,σn)					

Note that if, in some configuration, there are not r different legal things to do, then some of the entries on that row will repeat.

The Construction, Continued

M_d: (suppose r = 6)

Tape 1: Input

Tape 2: 1 3 2 6 5 4 3 6

M_d chooses its 1st move from column 1
M_d chooses its 2nd move from column 3
M_d chooses its 3rd move from column 2

•

•

until there are no more numbers on Tape 2

M_d either:

- discovers that M would accept, or
- comes to the end of Tape 2.

In either case, it halts.

The Construction, Continued

M' (the machine that simulates M):

Tape 1: Input

Tape 2: Copy of Input

Tape 3: 1 3 2 6 5 4 3 6

M_d

Steps of M':

write ε on Tape 3
until M_d accepts do
(1) copy Input from Tape 1 to Tape 2
(2) run M_d
(3) if M_d accepts, exit
(4) otherwise, generate lexicographically next string on Tape 3.

Pass	1	2	3		7	8	9		
Tape3	ε	1	2	···	6	11	12	···	2635

Nondeterministic Algorithms

Other Turing Machine Extensions

Multiple heads (on one tape)
Emulation strategy: Use tracks to keep track of tape heads. (See book)

Multiple tapes, multiple heads
Emulation strategy: Use tracks to keep track of tapes and tape heads.

Two-dimensional semi-infinite "tape"
Emulation strategy: Use diagonal enumeration of two-dimensional grid. Use second tape to help you keep track of where the tape head is. (See book)

Two-dimensional infinite "tape" (really a sheet)
Emulation strategy: Use modified diagonal enumeration as with the semi-infinite case.

What About Turing Machine Restrictions?

Can we make Turing machines even more limited and still get all the power?

Example:

We allow a tape alphabet of arbitrary size. What happens if we limit it to:

- One character?
- Two characters?
- Three characters?

Problem Encoding, TM Encoding, and the Universal TM

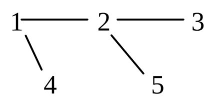
Read K & S 5.1 & 5.2.

Encoding a Problem as a Language

A Turing Machines deciding a language is analogous to the TM solving a **decision problem**.

Problem: Is the number n prime?
Instance of the problem: Is the number 9 prime?
Encoding of the problem, ⟨n⟩: n as a binary number. Example: 1001

Problem: Is an undirected graph G connected?
Instance of the problem: Is the following graph connected?



Encoding of the problem, ⟨G⟩:

1) |V| as a binary number
2) A list of edges represented by pairs of binary numbers being the vertex numbers that the edge connects
3) All such binary numbers are separated by "/".

Example: 101/1/10/10/11/1/100/10/101

Problem View vs. Language View

Problem View: It is *unsolvable* whether a Turing Machine halts on a given input. This is called the **Halting Problem**.

Language View: Let H = {⟨M, w⟩ : TM M halts on input string w}
H is recursively enumerable but not recursive.

The Universal Turing Machine

Problem: All our machines so far are hardwired.

Question: Does it make sense to talk about a programmable Turing machine that accepts as input
program *input string*
executes the program, and outputs
output string

Yes, it's called the Universal Turing Machine.

Notice that the Universal Turing machine semidecides H = {⟨M, w⟩ : TM M halts on input string w} = L(U).

To define the Universal Turing Machine U we need to do two things:

1. Define an encoding operation for Turing machines.
2. Describe the operation of U given an input tape containing two inputs:
   - encoded Turing machine M,
   - encoded input string to be given to M.

Encoding a Turing Machine M

We need to describe M = (K, Σ, δ, s, H) as a string. To do this we must:

1. Encode δ
2. Specify s.
3. Specify H (and y and n, if applicable)

1. To encode δ, we need to:
    1. Encode the states
    2. Encode the tape alphabet
    3. Specify the transitions

1.1 Encode the states as

qs : $s \in \{0, 1\}^+$ and

$|s| = i$ and

i is the smallest integer such that $2^i \geq |K|$

Example: 9 states i = 4

s = q0000,

remaining states: q0001, q0010, q0011,

q0100, q0101, q0110, q0111, q1000

Encoding a Turing Machine M, Continued

1.2 Encode the tape alphabet as

as : $s \in \{0, 1\}^+$ and

$|s| = j$ and

j is the smallest integer such that $2^j \geq |\Sigma| + 2$ (the + 2 allows for ← and →)

Example: Σ = {◊, ❑, a, b} j = 3

❑ = a000

◊ = a001

← = a010

→ = a011

a = a100

b = a101

Encoding a Turing Machine M, Continued

1.3 Specify transitions as (state, input, state, output)

Example: (q00,a000,q11,a000)

2. Specify s as $q0^i$

3. Specify H:

- States with no transitions out are in H.
- If M decides a language, then H = {y, n}, and we will adopt the convention that y is the lexicographically smaller of the two states.

  y = q010 n = q011

Encoding Input Strings

We encode input strings to a machine M using the same character encoding we use for M.
For example, suppose that we are using the following encoding for symbols in M:

symbol	representation
❑	a000
◊	a001
←	a010
→	a011
a	a100

Then we would represent the string s = ◊aa❑a as "s" = ⟨s⟩ = a001a100a100a000a100

An Encoding Example

Consider M = ({s, q, h}, {❑, ◊,a}, δ, s, {h}), where δ =

state	symbol	δ
s	a	(q, ❑)
s	❑	(h, ❑)
s	◊	(s, →)
q	a	(s, a)
q	❑	(s, →)
q	◊	(q, →)

state/symbol	representation
s	q00
q	q01
h	q11
❑	a000
◊	a001
←	a010
→	a011
a	a100

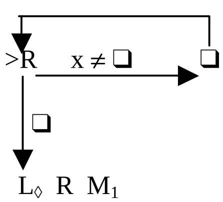
The representation of M, denoted, "M", ⟨M⟩, or sometimes ρ(M) =
(q00,a100,q01,a000), (q00,a000,q11,a000), (q00,a001,q00,a011),
(q01,a100,q00,a100), (q01,a000,q00,a011), (q01,a001,q01,a011)

Another Win of Encoding

One big win of defining a way to encode any Turing machine M:

- It will make sense to talk about operations on programs (Turing machines). In other words, we can talk about some Turing machine T that takes another Turing machine (say M_1) as input and transforms it into a different machine (say M_2) that performs some different, but possibly related task.

Example of a transforming TM T:
Input: a machine M_1 that reads its input tape and performs some operation P on it.
Output: a machine M_2 that performs P on an empty input tape:



The Universal Turing Machine

The specification for U:

U("M" "w") = "M(w)"

	"M ------	------	------	M"	"w------	------	------w"		
	1	0	0	0	0	0	0		
◊	❑	❑	❑	❑	❑	❑	❑	❑	❑
	❑	❑	❑	❑	❑	❑	❑		
	❑	❑	❑	❑	❑	❑	❑□		
	❑	❑	❑	❑	❑	❑	❑		

	"◊	❑"	"w------	------	------w"	❑	❑		
	1	0	0	0	0	0	0		
◊	"M ------	------	------	M"	❑	❑	❑	❑	❑
	1	0	0	0	0	0	0		
	q	0	0	0	❑	❑	❑		
	1	❑	❑	❑	❑	❑	❑		

Initialization of U:

1. Copy "M" onto tape 2
2. Insert "◊❑" at the left edge of tape 1, then shift w over.
3. Look at "M", figure out what i is, and write the encoding of state s on tape 3.

The Operation of U

	a	0	0	1	a	0	0		
	1	0	0	0	0	0	0		
◊	"M ---			M"	❑	❑	❑	❑	❑
	1	0	0	0	0	0	0		
	q	0	0	0	❑	❑	❑		
	1	❑	❑	❑	❑	❑	❑		

Simulate the steps of M:

1. Start with the heads:
   tape 1: the a of the character being scanned,
   tape 2: far left
   tape 3: far left

2. Simulate one step:
   1. Scan tape 2 for a quadruple that matches current state, input pair.
   2. Perform the associated action, by changing tapes 1 and 3. If necessary, extend the tape.
   3. If no quadruple found, halt. Else go back to 2.

An Example

Tape 1: a001a000a100a100a000a100
◊ ❑ a a ❑ a

Tape 2: (q00,a000,q11,a000), (q00,a001,q00,a011),
(q00,a100,q01,a000), (q01,a000,q00,a011),
(q01,a001,q01,a011), (q01,a100,q00,a100)

Tape 3: q01

Result of simulating the next step:

Tape 1: a001a000a100a100a000a100
◊ ❑ a a ❑ a

Tape 3: q00

If A Universal Machine is Such a Good Idea …

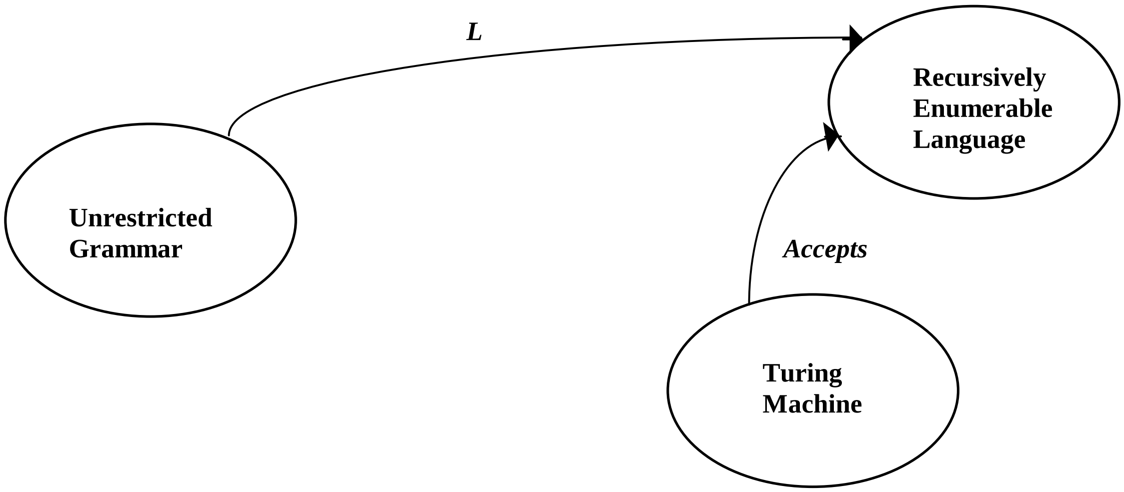
Could we define a Universal Finite State Machine?

Such a FSM would accept the language
L = {"F" "w" : F is a finite state machine, and w ∈ L(F) }

Grammars and Turing Machines

Do Homework 20.

Grammars, Recursively Enumerable Languages, and Turing Machines



Unrestricted Grammars

An unrestricted, or Type 0, or phrase structure grammar G is a quadruple
(V, Σ, R, S), where

- V is an alphabet,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite subset of
  - (V* (V-Σ) V*) × V*,
    context N *context* → result
- S (the start symbol) is an element of V - Σ.

We define derivations just as we did for context-free grammars.
The language generated by G is

$\{w \in \Sigma^* : S \Rightarrow_G^* w\}$

There is no notion of a derivation tree or rightmost/leftmost derivation for unrestricted grammars.

Unrestricted Grammars

Example: $L = a^n b^n c^n$, $n > 0$

S → aBSc
S → aBc
Ba → aB
Bc → bc
Bb → bb

Another Example

$L = \{w \in \{a, b, c\}^+$: number of a's, b's and c's is the same}

S → ABCS
S → ABC
AB → BA
BC → CB
AC → CA
BA → AB
CA → AC
CB → BC
A → a
B → b
C → c

A Strong Procedural Feel

Unrestricted grammars have a procedural feel that is absent from restricted grammars.

Derivations often proceed in phases. We make sure that the phases work properly by using nonterminals as flags that we're in a particular phase.

It's very common to have two main phases:

- Generate the right number of the various symbols.
- Move them around to get them in the right order.

No surprise: unrestricted grammars are general computing devices.

Equivalence of Unrestricted Grammars and Turing Machines

Theorem: A language is generated by an unrestricted grammar if and only if it is recursively enumerable (i.e., it is semidecided by some Turing machine M).

Proof:
Only if (grammar → TM): by construction of a nondeterministic Turing machine.

If (TM → grammar): by construction of a grammar that mimics backward computations of M.

Proof that Grammar → Turing Machine

Given a grammar G, produce a Turing machine M that semidecides L(G).

M will be nondeterministic and will use two tapes:

	◊	❑	a	b	a	❑	❑			
◊	0	1	0	0	0	0	0	❑	❑	
	◊	a	S	T	a	b	❑			
	0	1	0	0	0	0	0			

For each nondeterministic "incarnation":

- Tape 1 holds the input.
- Tape 2 holds the current state of a proposed derivation.

At each step, M nondeterministically chooses a rule to try to apply and a position on tape 2 to start looking for the left hand side of the rule. Or it chooses to check whether tape 2 equals tape 1. If any such machine succeeds, we accept. Otherwise, we keep looking.

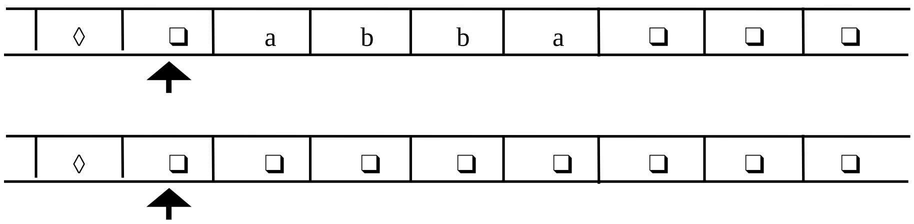
Proof that Turing Machine → Grammar

Suppose that M semidecides a language L (it halts when fed strings in L and loops otherwise). Then we can build M' that halts in the configuration (h, ◊$\underline{❑}$).

We will define G so that it simulates M' backwards.
We will represent the configuration (q, ◊u$\underline{a}$w) as

>uaqw<

M'
goes from



Then, if w ∈ L, we require that our grammar produce a derivation of the form
S $\Rightarrow_G$ >❑h< (produces final state of M')
$\Rightarrow_G$* >❑abq< (some intermediate state of M')
$\Rightarrow_G$* >❑sw< (the initial state of M')
$\Rightarrow_G$ w< (via a special rule to clean up >❑s)
$\Rightarrow_G$ w (via a special rule to clean up <)

The Rules of G

S → >❑h< (the halting configuration)

>❑s → ε (clean-up rules to be applied at the end)
< → ε

Rules that correspond to δ:

If δ(q, a) = (p, b) : bp → aq

If δ(q, a) = (p, →) : abp → aqb ∀b ∈ Σ
a❑p< → aq<

If δ(q, a) = (p, ←), a ≠ ❑ pa → aq

If δ(q, ❑) = (p, ←) p❑b → ❑qb ∀b ∈ Σ
p< → ❑q<

A *REALLY* Simple Example

M' = (K, {a}, δ, s, {h}), where

δ ={	((s, ❑), (q, →)),	1
	((q, a), (q, →)),	2
	((q, ❑), (t, ←)),	3
	((t, a), (p, ❑)),	4
	((t, ❑), (h, ❑)),	5
	((p, ❑), (t, ←))	6

L = a*

	S →>❑h<
	>❑s → ε
	< → ε
(1)	❑❑q→ ❑s❑
	❑aq → ❑sa
	❑❑q< → ❑s<
(2)	a❑q → aq❑
	aaq → aqa
	a❑q< → aq<
(3)	t❑❑ → ❑q❑
	t❑a → ❑qa
	t< → ❑q<
(4)	❑p → at
(5)	❑h → ❑t
(6)	t❑❑ → ❑p❑
	t❑a → ❑pa
	t< → ❑p<

Working It Out

	S →>❑h<	1
	>❑s → ε	2
	< → ε	3
(1)	❑❑q→ ❑s❑	4
	❑aq → ❑sa	5
	❑❑q< → ❑s<	6
(2)	a❑q → aq❑	7
	aaq → aqa	8
	a❑q< → aq<	9
(3)	t❑❑ → ❑q❑	10
	t❑a → ❑qa	11
	t< → ❑q<	12
(4)	❑p → at	13
(5)	❑h → ❑t	14
(6)	t❑❑ → ❑p❑	15
	t❑a → ❑pa	16
	t< → ❑p<	17

>❑saa<	1
>❑aqa<	2
>❑aaq<	2
>❑aa❑q<	3
>❑aat<	4
>❑a❑p<	6
>❑at<	4
>❑❑p<	6
>❑t<	5
>❑h<	

S	⇒ >❑h<	1
	⇒ >❑t<	14
	⇒ >❑❑p<	17
	⇒ >❑at<	13
	⇒ >❑a❑p<	17
	⇒ >❑aat<	13
	⇒ >❑aa❑q<	12
	⇒ >❑aaq<	9
	⇒ >❑aqa<	8
	⇒ >❑saa<	5
	⇒ aa<	2
	⇒ aa	3

An Alternative Proof

An alternative is to build a grammar G that simulates the forward operation of a Turing machine M. It uses alternating symbols to represent two interleaved tapes. One tape remembers the starting string, the other "working" tape simulates the run of the machine.

The first (generate) part of G:
Creates all strings over Σ* of the form

$w = \lozenge \lozenge ❑ ❑ Qs\ a_1\, a_1\, a_2\, a_2\, a_3\, a_3$ ❑ ❑ …

The second (test) part of G simulates the execution of M on a particular string w. An example of a partially derived string:

◊ ◊ ❑ ❑ a 1 b 2 c c b 4 Q3 a 3

Examples of rules:

b b Q 4 → b 4 Q 4 (rewrite b as 4)
b 4 Q 3 → Q 3 b 4 (move left)

The third (cleanup) part of G erases the junk if M ever reaches h.

Example rule:

h a 1 → a # h (sweep # h to the right erasing the working "tape")

Computing with Grammars

We say that **G computes f** if, for all w, v ∈ Σ*,

$SwS \Rightarrow_G^* v$ iff $v = f(w)$

Example:

$S1S \Rightarrow_G^* 11$
$S11S \Rightarrow_G^* 111$ f(x) = succ(x)

A function f is called **grammatically computable** iff there is a grammar G that computes it.

Theorem: A function f is recursive iff it is grammatically computable.
In other words, if a Turing machine can do it, so can a grammar.

Example of Computing with a Grammar

f(x) = 2x, where x is an integer represented in unary

G = ({S, 1}, {1}, R, S), where R =

$S1 \rightarrow 11S$
$SS \rightarrow \varepsilon$

Example:

Input: S111S

Output:

More on Functions: Why Have We Been Using Recursive as a Synonym for Computable?
Primitive Recursive Functions

Define a set of basic functions:

- $zero_k$ $(n_1, n_2, \ldots n_k) = 0$
- $identity_{k,j}$ $(n_1, n_2, \ldots n_k) = n_j$
- successor(n) = n + 1

Combining functions:

- Composition of g with $h_1, h_2, \ldots h_k$ is
  $g(h_1(\ \ ), h_2(\ \ ), \ldots h_k(\ \ ))$
- Primitive recursion of f in terms of g and h:
  $f(n_1,n_2,\ldots n_k, \quad 0) = g(n_1,n_2,\ldots n_k)$
  $f(n_1,n_2,\ldots n_k,m+1) = h(n_1,n_2,\ldots n_k, m, f(n_1, n_2,\ldots n_k,m))$

Example: plus(n, 0) = n
plus(n, m+1) = succ(plus(n, m))

Primitive Recursive Functions and Computability

Trivially true: all primitive recursive functions are Turing computable.
What about the other way: Not all Turing computable functions are primitive recursive.

Proof:
Lexicographically enumerate the unary primitive recursive functions, f_0, f_1, f_2, f_3, ….
Define $g(n) = f_n(n) + 1$.
G is clearly computable, but it is not on the list. Suppose it were f_m for some m. Then
$f_m(m) = f_m(m) + 1$, which is absurd.

	0	1	2	3	4
f_0					
f_1					
f_2					
f_3				27	
f_4					

Suppose g is f_3. Then g(3) = 27 + 1 = 28. Contradiction.

Functions that Aren't Primitive Recursive

Example: Ackermann's function:
A(0, y) = y + 1
A(x + 1, 0) = A(x, 1)
A(x + 1, y + 1) = A(x, A(x + 1, y))

	0	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	2^{65536}-3 *	$2^{2^{65536}}-3$ #	$2^{2^{2^{65536}}}-3$ %

* 19,729 digits
\# 10^{5940} digits
% $10^{10^{5939}}$ digits

10^{17} seconds since big bang
10^{87} protons and neutrons
10^{-23} light seconds = width of proton or neutron

Thus writing digits at the speed of light on all protons and neutrons in the universe (all lined up) starting at the big bang would have produced 10^{127} digits.

Recursive Functions

A function is **μ-recursive** if it can be obtained from the basic functions using the operations of:
- Composition,
- Recursive definition, and
- Minimalization of minimalizable functions:

The **minimalization** of g (of k + 1 arguments) is a function f of k arguments defined as:

$f(n_1,n_2,\ldots n_k)$ =	the least m such at $g(n_1,n_2,\ldots n_k,m)=1$,	if such an m exists,
	0	otherwise

A function g is **minimalizable** iff for every $n_1,n_2,\ldots n_k$, there is an m such that $g(n_1,n_2,\ldots n_k,m)=1$.

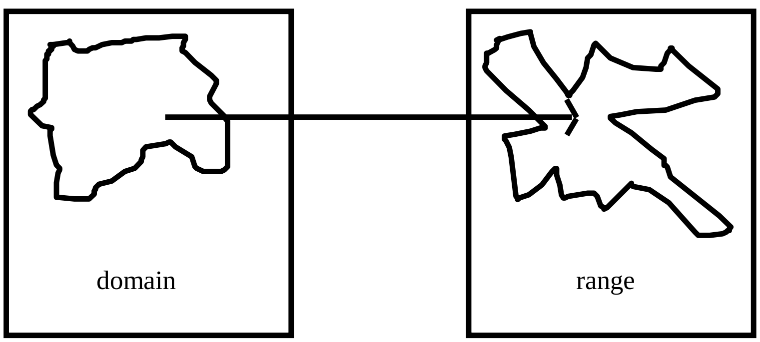
Theorem: A function is μ-recursive iff it is recursive (i.e., computable by a Turing machine).

Partial Recursive Functions

Consider the following function f:

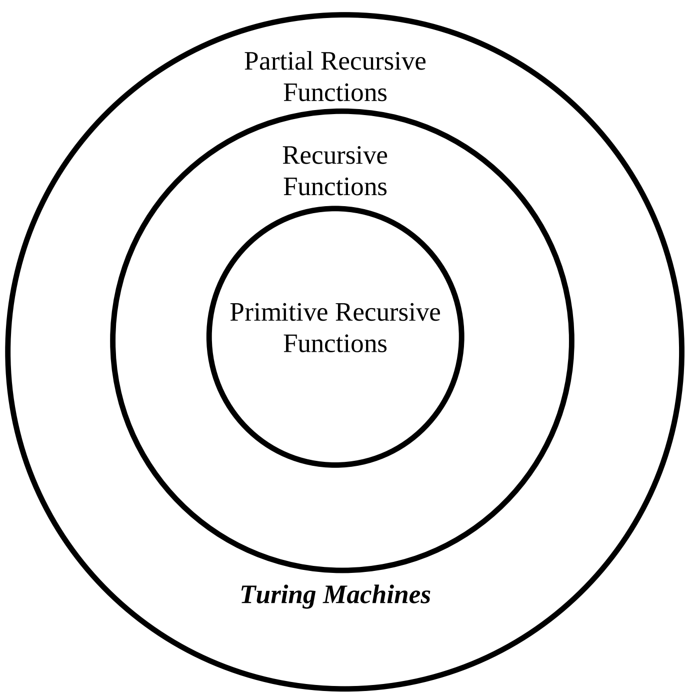
f(n) = 1 if TM(n) halts on a blank tape
0 otherwise

The domain of f is the natural numbers. Is f recursive?

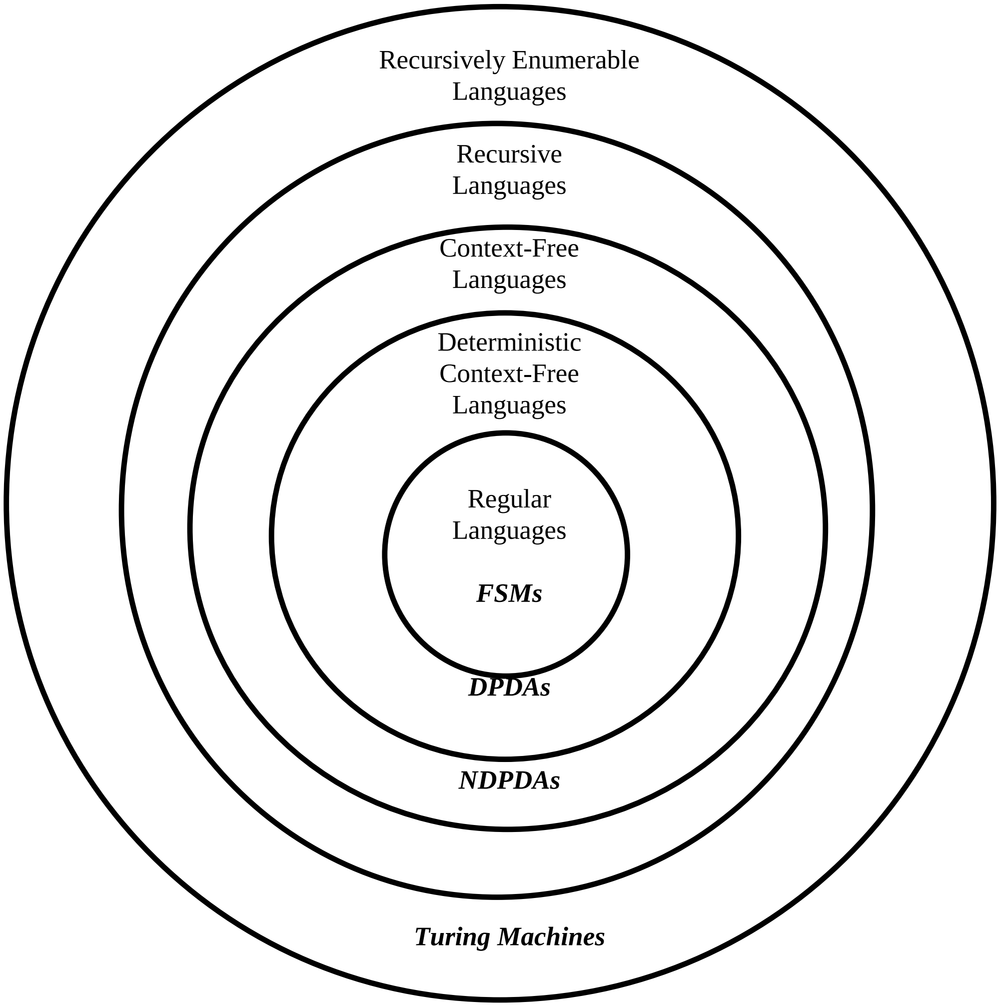


Theorem: There are uncountably many partially recursive functions (but only countably many Turing machines).

Functions and Machines



Languages and Machines



Is There Anything In Between CFGs and Unrestricted Grammars?

Answer: yes, various things have been proposed.

Context-Sensitive Grammars and Languages:

A grammar G is context sensitive if all productions are of the form

$x \rightarrow y$
and $|x| \leq |y|$

In other words, there are no length-reducing rules.

A language is context sensitive if there exists a context-sensitive grammar for it.

Examples:

$L = \{a^n b^n c^n, n > 0\}$
$L = \{w \in \{a, b, c\}^+$: number of a's, b's and c's is the same$\}$

Context-Sensitive Languages are Recursive

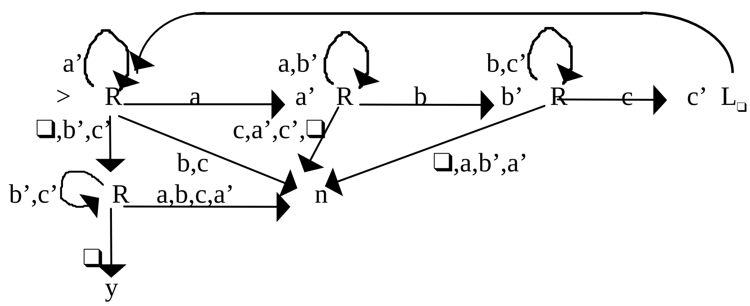
The basic idea: To decide if a string w is in L, start generating strings systematically, shortest first. If you generate w, accept. If you get to strings that are longer than w, reject.

Linear Bounded Automata

A linear bounded automaton is a nondeterministic Turing machine the length of whose tape is bounded by some fixed constant k times the length of the input.

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

◊❑aabbcc❑❑❑❑❑❑❑❑❑



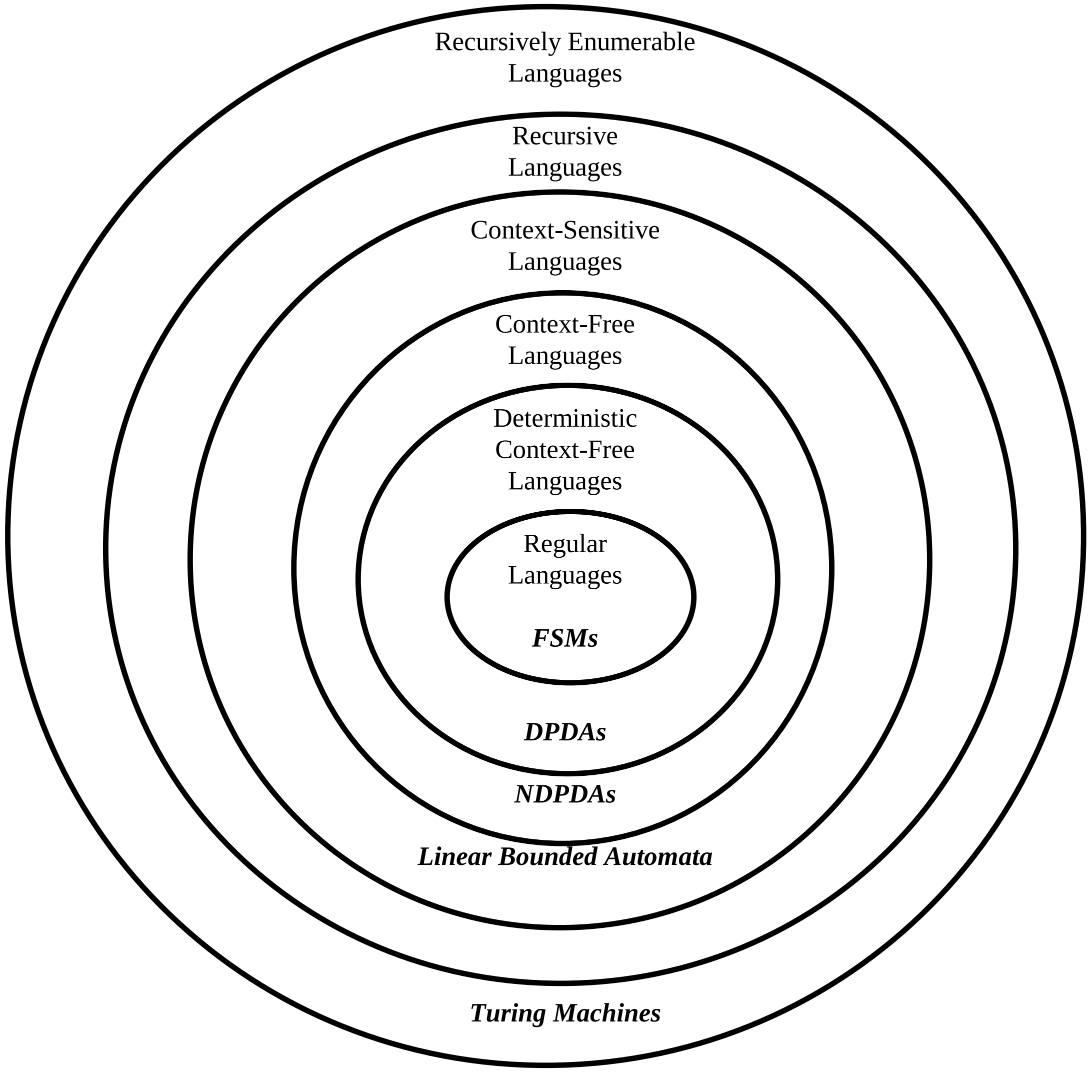
Context-Sensitive Languages and Linear Bounded Automata

Theorem: The set of context-sensitive languages is exactly the set of languages that can be accepted by linear bounded automata.

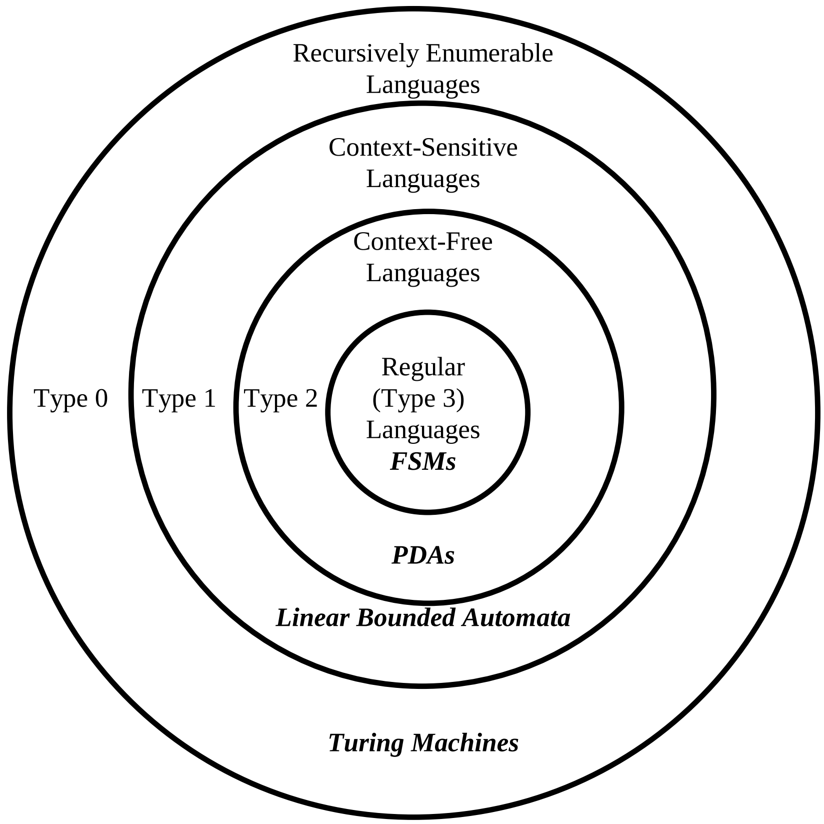
Proof: (sketch) We can construct a linear-bounded automaton B for any context-sensitive language L defined by some grammar G. We build a machine B with a two track tape. On input w, B keeps w on the first tape. On the second tape, it nondeterministically constructs all derivations of G. The key is that as soon as any derivation becomes longer than |w| we stop, since we know it can never get any shorter and thus match w. There is also a proof that from any lba we can construct a context-sensitive grammar, analogous to the one we used for Turing machines and unrestricted grammars.

Theorem: There exist recursive languages that are not context sensitive.

Languages and Machines



The Chomsky Hierarchy



Undecidabilty

Read K & S 5.1, 5.3, & 5.4.
Read Supplementary Materials: Recursively Enumerable Languages, Turing Machines, and Decidability.
Do Homeworks 21 & 22.

Church's Thesis
(Church-Turing Thesis)

An **algorithm** is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine.

This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Examples:

- unrestricted grammars
- lambda calculus
- cellular automata
- DNA computing
- quantum computing (?)

The Unsolvability of the Halting Problem

Suppose we could implement the decision procedure

```
HALTS(M, x)
        M: string representing a Turing Machine
        x: string representing the input for M
        If M(x) halts then True
                      else False
```

Then we could define

```
TROUBLE(x)
        x: string
        If HALTS(x, x) then loop forever
                       else halt
```

So now what happens if we invoke TROUBLE("TROUBLE"), which invokes HALTS("TROUBLE", "TROUBLE")

If HALTS says that TROUBLE halts on itself then TROUBLE loops. IF HALTS says that TROUBLE loops, then TROUBLE halts. Either way, we reach a contradiction, so HALTS(M, x) cannot be made into a decision procedure.

Another View

The Problem View: The halting problem is undecidable.

The Language View: Let H =
{"M" "w" : TM M halts on input string w}
H is recursively enumerable but not recursive.

Why?

H is recursively enumerable because it can be semidecided by U, the Universal Turing Machine.

But *H cannot be recursive*. If it were, then it would be decided by some TM MH. But MH("M" "w") would have to be:
If M is not a syntactically valid TM, then False.
else HALTS("M" "w")

But we know cannot that HALTS cannot exist.

If H were Recursive

H = {"M" "w" : TM M halts on input string w}

Theorem: If H were also recursive, then every recursively enumerable language would be recursive.

Proof: Let L be any RE language. Since L is RE, there exists a TM M that semidecides it.

Suppose H is recursive and thus is decided by some TM O (oracle).

We can build a TM M' from M that decides L:
1. M' transforms its input tape from ◊❑w❑ to ◊❑"M""w"❑.
2. M' invokes O on its tape and returns whatever answer O returns.

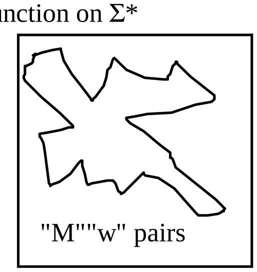
So, if H were recursive, all RE languages would be. **But it isn't.**

Undecidable Problems, Languages that Are Not Recursive, and Partial Functions

The Problem View: The halting problem is undecidable.

The Language View: Let H =
{"M" "w" : TM M halts on input string w}
H is recursively enumerable but not recursive.

The Functional View: Let f (w) = M(w)
f is a partial function on Σ*



Other Undecidable Problems About Turing Machines

- Given a Turing machine M, does M halt on the empty tape?
- Given a Turing machine M, is there any string on which M halts?
- Given a Turing machine M, does M halt on every input string?
- Given two Turing machines M_1 and M_2, do they halt on the same input strings?
- Given a Turing machine M, is the language that M semidecides regular? Is it context-free? Is it recursive?

Post Correspondence Problem

Consider two lists of strings over some alphabet Σ. The lists must be finite and of equal length.

$A = x_1, x_2, x_3, \ldots, x_n$
$B = y_1, y_2, y_3, \ldots, y_n$

Question: Does there exist some finite sequence of integers that can be viewed as indexes of A and B such that, when elements of A are selected as specified and concatenated together, we get the same string we get when elements of B are selected also as specified?

For example, if we assert that 1, 3, 4 is such a sequence, we're asserting that $x_1x_3x_4 = y_1y_3y_4$

Any problem of this form is an instance of the Post Correspondence Problem.

Is the Post Correspondence Problem decidable?

Post Correspondence Problem Examples

i	**A**	**B**
1	1	111
2	10111	10
3	10	0

i	**A**	**B**
1	10	101
2	011	11
3	101	011

Some Languages Aren't Even Recursively Enumerable

A pragmatically non RE language: L_1={ (i, j) : i, j are integers where the low order five digits of i are a street address number and j is the number of houses with that number on which it rained on November 13, 1946 }

An analytically non RE language: L_2={x : x = "M" of a Turing machine M and M("M") does *not* halt}

Why isn't L_2 RE? Suppose it were. Then there would be a TM M* that semidecides L_2. Is "M*" in L_2?
- If it is, then M*("M*") halts (by the definition of M* as a semideciding machine for L_2)
- But, by the definition of L_2, if "M*" $\in$ L_2, then M*("M*") does not halt.

Contradiction. So L_2 is not RE.

Another Non RE Language

$\overline{H}$

Why not?

Reduction

Let $L_1, L_2 \subseteq \Sigma^*$ be languages. A **reduction** from L_1 to L_2 is a recursive function $\tau: \Sigma^* \rightarrow \Sigma^*$ such that

$x \in L_1$ iff $\tau(x) \in L_2$.

Example:

$L_1 = \{a, b : a,b \in N : b = a + 1\}$

$\Downarrow$ τ = Succ

$\Downarrow$ a, b becomes Succ(a), b

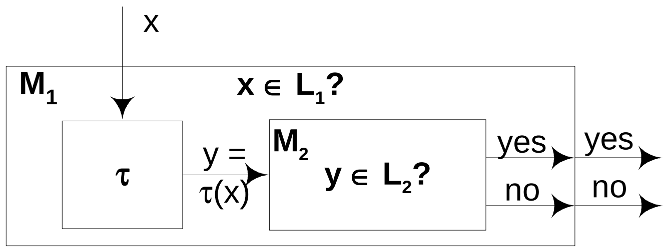
$L_2 = \{a, b : a,b \in N : a = b\}$

If there is a Turing machine M_2 to decide L_2, then I can build a Turing machine M_1 to decide L_1:

1. Take the input and apply Succ to the first number.
2. Invoke M_2 on the result.
3. Return whatever answer M_2 returns.

Reductions and Recursive Languages

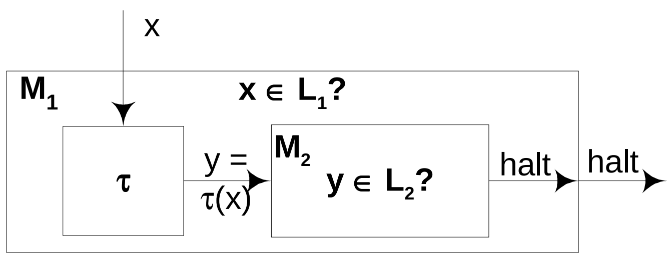
Theorem: If there is a reduction from L_1 to L_2 and L_2 is recursive, then L_1 is recursive.



Theorem: If there is a reduction from L_1 to L_2 and L_1 is not recursive, then L_2 is not recursive.

Reductions and RE Languages

Theorem: If there is a reduction from L_1 to L_2 and L_2 is RE, then L_1 is RE.



Theorem: If there is a reduction from L_1 to L_2 and L_1 is not RE, then L_2 is not RE.

Can it be Decided if M Halts on the Empty Tape?

This is equivalent to, "Is the language L_2 = {"M" : Turing machine M halts on the empty tape} recursive?"

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow$ τ

(?M_2) L_2 = {s = "M" : Turing machine M halts on the empty tape}

Let τ be the function that, from "M" and "w", constructs "M*", which operates as follows on an empty input tape:

1. Write w on the tape.
2. Operate as M would have.

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

A Formal Reduction Proof

Prove that $L_2 = \{\langle M\rangle$: Turing machine M halts on the empty tape$\}$ is not recursive.

Proof that L_2 is not recursive via a reduction from H = $\{\langle M, w\rangle$: Turing machine M halts on input string w$\}$, a non-recursive language. Suppose that there exists a TM, M_2 that decides L_2. Construct a machine to decide H as $M_1(\langle M, w\rangle) = M_2(\tau(\langle M, w\rangle))$. The τ function creates from $\langle M\rangle$ and $\langle w\rangle$ a new machine M*. M* ignores its input and runs M on w, halting exactly when M halts on w.

- $\langle M, w\rangle \in H \Rightarrow$ M halts on w $\Rightarrow$ M* always halts $\Rightarrow \varepsilon \in L(M^*) \Rightarrow \langle M^*\rangle \in L_2 \Rightarrow M_2$ accepts $\Rightarrow M_1$ accepts.
- $\langle M, w\rangle \notin H \Rightarrow$ M does not halt on w $\Rightarrow \varepsilon \notin L(M^*) \Rightarrow \langle M^*\rangle \notin L_2 \Rightarrow M_2$ rejects $\Rightarrow M_1$ rejects.

Thus, if there is a machine M_2 that decides L_2, we could use it to build a machine that decides H. Contradiction. $\therefore L_2$ is not recursive.

Important Elements in a Reduction Proof

- A clear declaration of the reduction “from” and “to” languages and what you’re trying to prove with the reduction.
- A description of how a machine is being constructed for the “from” language based on an assumed machine for the “to” language and a recursive τ function.
- A description of the τ function’s inputs and outputs. If τ is doing anything nontrivial, it is a good idea to argue that it is recursive.
- Note that machine diagrams are not necessary or even sufficient in these proofs. Use them as thought devices, where needed.
- Run through the logic that demonstrates how the “from” language is being decided by your reduction. You must do both accepting and rejecting cases.
- Declare that the reduction proves that your “to” language is not recursive.

The Most Common Mistake: Doing the Reduction Backwards

The right way to use reduction to show that L_2 is not recursive:

1. Given that L_1 is not recursive,
2. Reduce L_1 to L_2, i.e. show how to solve L_1 (the known one) in terms of L_2 (the unknown one)

L_1
$\Downarrow$
L_2

Example: If there exists a machine M_2 that solves L_2, the problem of deciding whether a Turing machine halts on a blank tape, then we could do H (deciding whether M halts on w) as follows:

1. Create M* from M such that M*, given a blank tape, first writes w on its tape, then simulates the behavior of M.
2. Return M_2("M*").

Doing it wrong by reducing L_2 (the unknown one to L_1): If there exists a machine M_1 that solves H, then we could build a machine that solves L_2 as follows:

1. Return (M_1("M", "")).

Why Backwards Doesn't Work

Suppose that we have proved that the following problem L_1 is unsolvable: Determine the number of days that have elapsed since the beginning of the universe.

Now consider the following problem L_2: Determine the number of days that had elapsed between the beginning of the universe and the assassination of Abraham Lincoln.

Reduce L_1 to L_2:
$L_1 = L_2$ + (now - 4/9/1865)

$$L_1 \downarrow L_2$$

Reduce L_2 to L_1:
$L_2 = L_1$ - (now - 4/9/1865)

$$L_2 \downarrow L_1$$

Why Backwards Doesn't Work, Continued

L_1 = days since beginning of universe
L_2 = elapsed days between the beginning of the universe and the assassination of Abraham Lincoln.
L_3 = days between the assassination of Abraham Lincoln and now.

Considering L_2:
Reduce L_1 to L_2:
$L_1 = L_2$ + (now - 4/9/1865)

$$L_1 \downarrow L_2$$

Reduce L_2 to L_1:
$L_2 = L_1$ - (now - 4/9/1865)

$$L_2 \downarrow L_1$$

Considering L_3:
Reduce L_1 to L_3:
L_1 = oops

$$L_1 \downarrow L_3$$

Reduce L_3 to L_1:
$L_3 = L_1$ - 365 - (now - 4/9/1866)

$$L_3 \downarrow L_1$$

Is There Any String on Which M Halts?

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow$ τ

(?M_2) L_2 = {s = "M" : there exists a string on which Turing machine M halts}

Let τ be the function that, from "M" and "w", constructs "M*", which operates as follows:

1. M* examines its input tape.
2. If it is equal to w, then it simulates M.
3. If not, it loops.

Clearly the only input on which M* has a chance of halting is w, which it does iff M would halt on w.

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Does M Halt on All Inputs?

L_1 = {s = "M" : Turing machine M halts on the empty tape}

$\Downarrow$ τ

(?M_2) L_2 = {s = "M" : Turing machine M halts on all inputs}

Let τ be the function that, from "M", constructs "M*", which operates as follows:

1. Erase the input tape.
2. Simulate M.

Clearly M* either halts on all inputs or on none, since it ignores its input.

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Rice's Theorem

Theorem: No nontrivial property of the recursively enumerable languages is decidable.

Alternate statement: Let P: $2^{\Sigma^*} \rightarrow$ {true, false} be a nontrivial property of the recursively enumerable languages. The language {"M": P(L(M)) = True} is not recursive.

By "nontrivial" we mean a property that is not simply true for all languages or false for all languages.

Examples:

- L contains only even length strings.
- L contains an odd number of strings.
- L contains all strings that start with "a".
- L is infinite.
- L is regular.

Note:
Rice's theorem applies to languages, not machines. So, for example, the following properties of machines are decidable:

- M contains an even number of states
- M has an odd number of symbols in its tape alphabet

Of course, we need a way to define a language. We'll use machines to do that, but the properties we'll deal with are properties of L(M), not of M itself.

Proof of Rice's Theorem

Proof: Let P be any nontrivial property of the RE languages.

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow$ τ

(?M_2) L_2 = {s = "M" : P(L(M)) = true}

Either P(∅) = true or P(∅) = false. Assume it is false (a matching proof exists if it is true). Since P is nontrivial, there is some language L_P such that $P(L_P)$ is true. Let M_P be some Turing machine that semidecides L_P.

Let τ construct "M*", which operates as follows:

1. Copy its input y to another track for later.
2. Write w on its input tape and execute M on w.
3. If M halts, put y back on the tape and execute M_P.
4. If M_P halts on y, accept.

Claim: If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Why?

Two cases to consider:

- "M" "w" $\in$ H $\Rightarrow$ M halts on w $\Rightarrow$ M* will halt on all strings that are accepted by $M_P \Rightarrow L(M^*) = L(M_P) = L_P \Rightarrow P(L(M^*)) = P(L_P)$ = true $\Rightarrow M_2$ decides P, so M_2 accepts "M*" $\Rightarrow M_1$ accepts.

- "M" "w" $\notin$ H $\Rightarrow$ M doesn't halt on w $\Rightarrow$ M* will halt on nothing $\Rightarrow L(M^*) = \varnothing \Rightarrow P(L(M^*)) = P(\varnothing)$ = false $\Rightarrow$ M_2 decides P, so M_2 rejects "M*" $\Rightarrow M_1$ rejects.

Using Rice's Theorem

Theorem: No nontrivial property of the recursively enumerable languages is decidable.

To use Rice's Theorem to show that a language L is not recursive we must:

- Specify a language property, P(L)

- Show that the domain of P is the set of recursively enumerable languages.

- Show that P is nontrivial:
  - P is true of at least one language
  - P is false of at least one language

Using Rice's Theorem: An Example

L = {s = "M" : there exists a string on which Turing machine M halts}.
= {s = "M" : $L(M) \neq \varnothing$ }

- Specify a language property, P(L):
  P(L) = True iff $L \neq \varnothing$

- Show that the domain of P is the set of recursively enumerable languages.
  The domain of P is the set of languages semidecided by some TM. This is exactly the set of RE languages.

- Show that P is nontrivial:
  P is true of at least one language: P({ε}) = True
  P is false of at least one language: $P(\varnothing)$ = False

Inappropriate Uses of Rice's Theorem

Example 1:
L = {s = "M" : M writes a 1 within three moves}.

- Specify a language property, P(L)
  P(M?) = True if M writes a 1 within three moves,
  False otherwise
- Show that the domain of P is the set of recursively enumerable languages.
  ??? The domain of P is the set of all TMs, not their languages

Example 2:
L = {s = "M1" "M2": L(M1) = L(M2)}.

- Specify a language property. P(L)
  P(M1?, M2?) = True if L(M1) = L(M2)
  False otherwise
- Show that the domain of P is the set of recursively enumerable languages.
  ??? The domain of P is RE $\times$ RE

Given a Turing Machine M, is L(M) Regular (or Context Free or Recursive)?

Is this problem decidable?

No, by Rice's Theorem, since being regular (or context free or recursive) is a nontrivial property of the recursively enumerable languages.

We can also show this directly (via the same technique we used to prove the more general claim contained in Rice's Theorem):

Given a Turing Machine M, is L(M) Regular (or Context Free or Recursive)?

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow\ \tau$

(?M_2) L_2 = {s = "M" : L(M) is regular}

Let τ be the function that, from "M" and "w", constructs "M*", whose own input is a string
t = "M_*" "w_*"
M*("M_*" "w_*") operates as follows:

1. Copy its input to another track for later.
2. Write w on its input tape and execute M on w.
3. If M halts, invoke U on "M_*" "w_*".
4. If U halts, halt and accept.

If M_2 exists, then $\neg M_2(M^*(s))$ decides L_1 (H).

Why?
If M does not halt on w, then M* accepts $\varnothing$ (which is regular).
If M does halt on w, then M* accepts H (which is not regular).

Undecidable Problems About Unrestricted Grammars

- Given a grammar G and a string w, is $w \in L(G)$?
- Given a grammar G, is $\varepsilon \in L(G)$?
- Given two grammars G_1 and G_2, is $L(G_1) = L(G_2)$?
- Given a grammar G, is $L(G) = \varnothing$?

Given a Grammar G and a String w, Is w ∈ L(G)?

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow$ τ

(?M_2) L_2 = {s = "G" "w" : $w \in L(G)$}

Let τ be the construction that builds a grammar G for the language L that is semidecided by M. Thus
$w \in L(G)$ iff M(w) halts.

Then τ("M" "w") = "G" "w"

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Undecidable Problems About Context-Free Grammars

- Given a context-free grammar G, is $L(G) = \Sigma^*$?
- Given two context-free grammars G_1 and G_2, is $L(G_1) = L(G_2)$?
- Given two context-free grammars G_1 and G_2, is $L(G_1) \cap L(G_2) = \varnothing$?
- Is context-free grammar, G ambiguous?
- Given two pushdown automata M_1 and M_2, do they accept precisely the same language?
- Given a pushdown automaton M, find an equivalent pushdown automaton with as few states as possible.

Given Two Context-Free Grammars G_1 and G_2, Is $L(G_1) = L(G_2)$?

$L_1 =$ {s = "G" a CFG G and $L(G) = \Sigma^*$}

$\Downarrow$ τ

($?M_2$) $L_2 =$ {s = "G_1" "G_2" : G_1 and G_2 are CFGs and $L(G_1) = L(G_2)$}

Let τ append the description of a context free grammar G_{Σ^*} that generates Σ^*.

Then, τ("G") = "G" "G_{Σ^*}"

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Non-RE Languages

There are an uncountable number of non-RE languages, but only a countably infinite number of TM's (hence RE languages). $\therefore$The class of non-RE languages is <u>much</u> bigger than that of RE languages!

Intuition: Non-RE languages usually involve either infinite search or knowing a TM will infinite loop to accept a string.

{$\langle M\rangle$: M is a TM that does not halt on the empty tape}
{$\langle M\rangle$: M is a TM and $L(M) = \Sigma^*$}
{$\langle M\rangle$: M is a TM and there does not exist a string on which M halts}

Proving Languages are not RE

- Diagonalization
- Complement RE, not recursive
- Reduction from a non-RE language
- Rice's theorem for non-RE languages (not covered)

Diagonalization

L={$\langle M\rangle$: M is a TM and $M(\langle M\rangle)$ does not halt} is not RE

Suppose L is RE. There is a TM M* that semidecides L. Is $\langle M^*\rangle$ in L?
- If it is, then $M^*(\langle M^*\rangle)$ halts (by the definition of M* as a semideciding machine for L)
- But, by the definition of L, if $\langle M^*\rangle \in L$, then $M^*(\langle M^*\rangle)$ does not halt.

Contradiction. So L is not RE.

(This is a very "bare-bones" diagonalization proof.)

Diagonalization can only be easily applied to a few non-RE languages.

Complement of an RE, but not Recursive Language

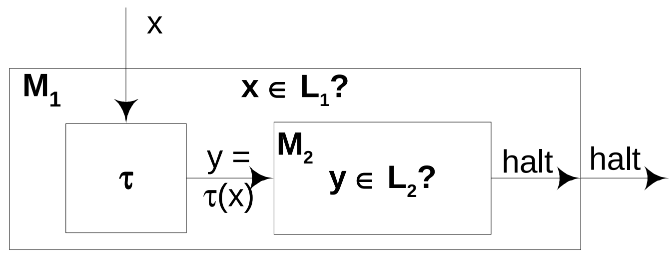
Example: $\overline{H}$ = {⟨M, w⟩: M does not accept w}
Consider H = {⟨M, w⟩: M is a TM that accepts w}:

- H is RE—it is semidecided by U, the Universal Turing Machine.
- H is not recursive—it is equivalent to the halting problem, which is undecidable.

From the theorem, $\overline{H}$ is not RE.

Reductions and RE Languages

Theorem: If there is a reduction from L_1 to L_2 and L_2 is RE, then L_1 is RE.



Theorem: If there is a reduction from L_1 to L_2 and L_1 is not RE, then L_2 is not RE.

Reduction from a known non-RE Language

Using a reduction from a non-RE language:

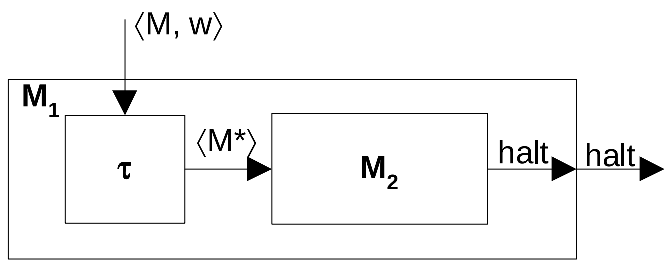
$L_1 = \overline{H}$ = {⟨M, w⟩: Turing machine M does not halt on input string w}

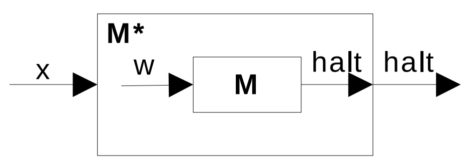
$\Downarrow \tau$

(?M_2) L_2 = {⟨M⟩: there does not exist a string on which Turing machine M halts}

Let τ be the function that, from ⟨M⟩ and ⟨w⟩, constructs ⟨M*⟩, which operates as follows:

1. Erase the input tape (M* ignores its input).
2. Write w on the tape
3. Run M on w.

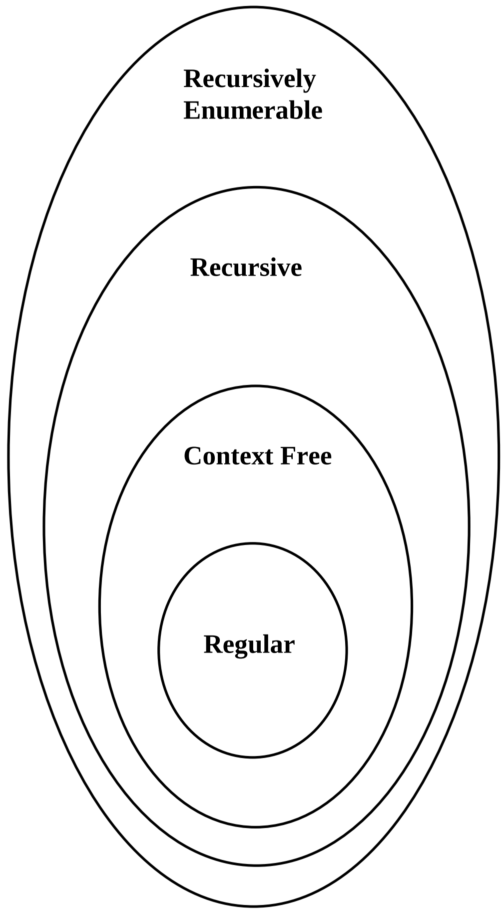




⟨M, w⟩ ∈ $\overline{H}$ ⇒ M does not halt on w ⇒ M* does not halt on any input ⇒ M* halts on nothing ⇒ M_2 accepts (halts).
⟨M, w⟩ ∉ $\overline{H}$ ⇒ M halts on w ⇒ M* halts on everything ⇒ M_2 loops.

If M_2 exists, then M_1(⟨M, w⟩) = $M_2(M_\tau$(⟨M, w⟩)) and M_1 semidecides L_1. Contradiction. L_1 is not RE. ∴ L_2 is not RE.

Language Summary



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

Introduction to Complexity Theory

Read K & S Chapter 6.

Most computational problems you will face your life are solvable (decidable). We have yet to address whether a problem is "easy" or "hard". Complexity theory tries to answer this question.

Recall that a computational problem can be recast as a language recognition problem.

Some "easy" problems:
- Pattern matching
- Parsing
- Database operations (select, join, etc.)
- Sorting

Some "hard" problems:
- Traveling salesman problem
- Boolean satisfiability
- Knapsack problem
- Optimal flight scheduling

"Hard" problems usually involve the examination of a large search space.

Big-O Notation

- Gives a quick-and-dirty measure of function size
- Used for time and space metrics

A function f(n) is O(g(n)) whenever there exists a constant c, such that $|f(n)| \le c \cdot |g(n)|$ for all $n \ge 0$.

(We are usually most interested in the "smallest" and "simplest" function, g.)

Examples:

$2n^3 + 3n^2 \cdot \log(n) + 75n^2 + 7n + 2000$ is $\underline{O(n^3)}$

$75 \cdot 2^n + 200n^5 + 10000$ is $\underline{O(2^n)}$

A function f(n) is *polynomial* if f(n) is O(p(n)) for some polynomial function p.

If a function f(n) is not polynomial, it is considered to be *exponential*, whether or not it is O of some exponential function (e.g. $n^{\log n}$).

In the above two examples, the first is polynomial and the second is exponential.

Comparison of Time Complexities

Speed of various time complexities for different values of n, taken to be a measure of *problem size*. (Assumes 1 step per microsecond.)

f(n)\n	10	20	30	40	50	60
n	.00001 sec.	.00002 sec.	.00003 sec.	.00004 sec.	.00005 sec.	.00006 sec.
$\mathbf{n^2}$	.0001 sec.	.0004 sec.	.0009 sec.	.0016 sec.	.0025 sec.	.0036 sec.
$\mathbf{n^3}$	.001 sec.	.008 sec.	.027 sec.	.064 sec.	.125 sec.	.216 sec.
$\mathbf{n^5}$	.1 sec.	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.	13.0 min.
$\mathbf{2^n}$	.001 sec.	1.0 sec.	17.9 min.	12.7 days	35.7 yr.	366 cent.
$\mathbf{3^n}$	.059 sec.	58 min.	6.5 yr.	3855 cent.	$2\text{x}10^8$ cent.	$1.3\text{x}10^{13}$ cent.

Faster computers don't really help. Even taking into account Moore's Law, algorithms with exponential time complexity are considered *intractable*. $\therefore$ Polynomial time complexities are strongly desired.

Polynomial Land

If $f_1(n)$ and $f_2(n)$ are polynomials, then so are:

- $f_1(n) + f_2(n)$
- $f_1(n) \cdot f_2(n)$
- $f_1(f_2(n))$

This means that we can sequence and compose polynomial-time algorithms with the resulting algorithms remaining polynomial-time.

Computational Model

For formally describing the time (and space) complexities of algorithms, we will use our old friend, the deciding TM (decision procedure).

There are two parts:

- The problem to be solved must be translated into an equivalent language recognition problem.
- A TM to solve the language recognition problem takes an encoded instance of the problem (of size *n* symbols) as input and decides the instance in at most $T_M(n)$ steps.

We will classify the time complexity of an algorithm (TM) to solve it by its big-O bound on $T_M(n)$.

We are most interested in polynomial time complexity algorithms for various types of problems.

Encoding a Problem

Traveling Salesman Problem: Given a set of cities and the distances between them, what is the minimum distance tour a salesman can make that covers all cities and returns him to his starting city?

Stated as a decision question over graphs: Given a graph G = (V, E), a positive distance function for each edge d: $E \rightarrow N+$, and a bound B, is there a circuit that covers all V where $\sum d(e) \leq B$? (Here a *minimization* problem was turned into a *bound* problem.)

A possible encoding the problem:

- Give |V| as an integer.
- Give B as an integer.
- Enumerate all (v_1, v_2, d) as a list of triplets of integers (this gives both E and d).
- All integers are expressed as Boolean numbers.
- Separate these entries with commas.

Note that the sizes of most "reasonable" problem encodings are polynomially related.

What about Turing Machine Extensions?

Most TM extensions are can be simulated by a standard TM in a time polynomially related to the time of the extended machine.

- k-tape TM can be simulated in $O(T^2(n))$
- Random Access Machine can be simulated in $O(T^3(n))$

(Real programming languages can be polynomially related to the RAM.)

BUT… The nondeterminism TM extension is different.

A nondeterministic TM can be simulated by a standard TM in $O(2^{p(n)})$ for some polynomial p(n).
Some faster simulation method might be possible, but we don't know it.

Recall that a nondeterministic TM can use a "guess and test" approach, which is computationally efficient at the expense of many parallel instances.

The Class P

P = { L : there is a polynomial-time deterministic TM, M that decides L }

Roughly speaking, P is the class of problems that can be solved by deterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

The way the problem is encoded or the computational abilities of the machine carrying out the algorithm are not very important.

Example: Given an integer n, is there a positive integer m, such that n = 4m?

Problems in P are considered tractable or “easy”.

The Class NP

NP = { L: there is a polynomial time nondeterministic TM, M that decides L }

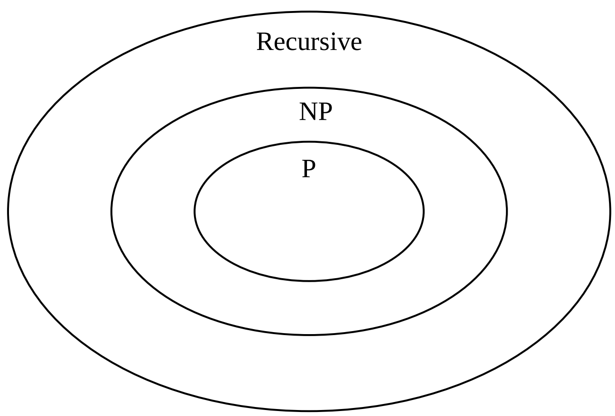
Roughly speaking, NP is the class of problems that can be solved by nondeterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

Many problems in NP are considered “intractable” or “hard”.

Examples:

- **Traveling salesman problem**: Given a graph G = (V, E), a positive distance function for each edge d: $E \rightarrow N+$, and a bound B, is there a circuit that covers all V where $\sum d(e) \leq B$?
- **Subgraph isomorphism problem**: Given two graphs G_1 and G_2, does G_1 contain a subgraph isomorphic to G_2?

The Relationship of P and NP



We’re considering only solvable (decidable) problems.

Clearly $P \subseteq NP$.

P is closed under complement.

NP probably isn’t closed under complement. Why?

Whether P = NP is considered computer science’s greatest unsolved problem.

Why NP is so Interesting

- To date, nearly all decidable problems with polynomial bounds on the size of the solution are in this class.
- Most NP problems have simple nondeterministic solutions.
- The hardest problems in NP have exponential deterministic time complexities.
- Nondeterminism doesn't influence decidability, so maybe it shouldn't have a big impact on complexity.
- Showing that P = NP would dramatically change the computational power of our algorithms.

Stephen Cook's Contribution (1971)

- Emphasized the importance of polynomial time reducibility.
- Pointed out the importance of NP.
- Showed that the Boolean Satisfiability (SAT) problem has the property that every other NP problem can be polynomially reduced to it. Thus, SAT can be considered the hardest problem in NP.
- Suggested that other NP problems may also be among the "hardest problems in NP".

This "hardest problems in NP" class is called the class of "NP-complete" problems.

Further, if any of these NP-complete problems can be solved in deterministic polynomial time, they all can and, by implication, P = NP.

Nearly all of complexity theory relies on the assumption that $P \neq NP$.

Polynomial Time Reducibility

A language L_1 is *polynomial time reducible* to L_2 if there is a polynomial-time recursive function τ such that $\forall x \in L_1$ iff $\tau(x) \in L_2$.

If L_1 is polynomial time reducible to L_2, we say L_1 reduces to L_2 ("polynomial time" is assumed) and we write it as $L_1 \propto L_2$.

Lemma: If $L_1 \propto L_2$, then $(L_2 \in P) \Rightarrow (L_1 \in P)$. And conversely, $(L_1 \notin P) \Rightarrow (L_2 \notin P)$.

Lemma: If $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$.

L_1 and L_2 are *polynomially equivalent* whenever both $L_1 \propto L_2$ and $L_2 \propto L_1$.

Polynomially equivalent languages form an equivalence class. The partitions of this equivalence class are related by the partial order $\propto$.
P is the "least" element in this partial order.

What is the "maximal" element in the partial order?

The Class NP-Complete

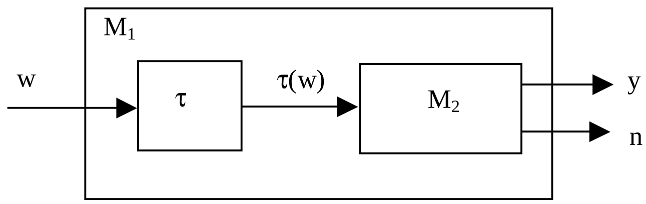
A language L is *NP-complete* if L $\in$ NP and for all other languages L' $\in$ NP, L' $\propto$ L.

NP-Complete problems are the "hardest" problems in NP.

Lemma: If L_1 and L_2 belong to NP, L_1 is NP-complete and $L_1 \propto L_2$, then L_2 is NP-complete.

Thus to prove a language L_2 is NP-complete, you must do the following:

1. Show that $L_2 \in$ NP.
2. Select a known NP-complete language L_1.
3. Construct a reduction τ from L_1 to L_2.
4. Show that τ is polynomial-time function.



How do we get started? Is there a language that is NP-complete?

Boolean Satisfiability (SAT)

Given a set of Boolean variables U = {u_1, u_2, …, u_m} and a Boolean expression in conjunctive normal form (conjunctions of clauses—disjunctions of variables or their negatives), is there a truth assignment to U that makes the Boolean expression true (satisfies the expression)?

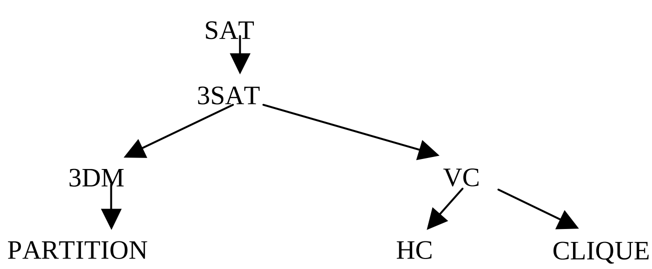
Note: All Boolean expressions can be converted to conjunctive normal form.
Example: $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_4 \vee \neg x_2)$

Cook's Theorem: SAT is NP-complete.

1. Clearly SAT $\in$ NP.
2. The proof constructs a complex Boolean expression that satisfied exactly when a NDTM accepts an input string x where |w| = n. Because the NDTM is in NP, its running time is O(p(n)). The number of variables is polynomially related to p(n).

SAT is *NP-complete* because SAT $\in$ NP and for all other languages L' $\in$ NP, L' $\propto$ SAT.

Reduction Roadmap



The early NP-complete reductions took this structure. Each phrase represents a problem. The arrow represents a reduction from one problem to another.

Today, thousands of diverse problems have been shown to be NP-complete.

Let's now look at these problems.

3SAT (3-satisfiability)

Boolean satisfiability where each clause has exactly 3 terms.

3DM (3-Dimensional Matching)

Consider a set $M \subseteq X \times Y \times Z$ of disjoint sets, X, Y, & Z, such that $|X| = |Y| = |Z| = q$. Does there exist a *matching*, a subset $M' \subseteq M$ such that $|M'| = q$ and M' partitions X, Y, and Z?

This is a generalization of the marriage problem, which has two sets men & women and a relation describing acceptable marriages. Is there a pairing that marries everyone acceptably?

The marriage problem is in P, but this "3-sex version" of the problem is NP-complete.

PARTITION

Given a set A and a positive integer size, $s(a) \in N^+$, for each element, $a \in A$. Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A-A'} s(a) \ ?$$

VC (Vertex Cover)

Given a graph G = (V, E) and an integer K, such that $0 < K \leq |V|$, is there a *vertex cover* of size K or less for G, that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge, $(u, v) \in E$, at least one of u and v belongs to V'?

CLIQUE

Given a graph G = (V, E) and an integer J, such that
$0 < J \leq |V|$, does G contain a *clique* of size J or more, that is a subset $V' \subseteq V$ such that $|V'| \geq J$ and every two vertices in V' are joined by an edge in E?

HC (Hamiltononian Circuit)

Given a graph G = (V, E), does there exist a *Hamiltonian circuit*, that is an ordering $<v_1, v_2, \ldots, v_n>$ of all V such that $(v_{|V|}, v_1) \in E$ and $(v_i, v_{i+1}) \in E$ for all i, $1 \leq i < |V|$?

Traveling Salesman Prob. is NP-complete

Given a graph G = (V, E), a positive distance function for each edge $d: E \rightarrow N+$, and a bound B, is there a circuit that covers all V where $\sum d(e) \leq B$?

To prove a language TSP is NP-complete, you must do the following:

1. Show that $TSP \in NP$.
2. Select a known NP-complete language L_1.
3. Construct a reduction τ from L_1 to TSP.
4. Show that τ is polynomial-time function.

TSP ∈ NP: Guess a set of roads. Verify that the roads form a tour that hits all cities. Answer "yes" if the guess is a tour and the sum of the distances is $\leq B$.

Reduction from HC: Answer the Hamiltonian circuit question on G = (V, E) by constructing a complete graph where "roads" have distance 1 if the edge is in E and 2 otherwise. Pose the TSP problem, is there a tour of length $\leq |V|$?

Notes on NP-complete Proofs

The more NP-complete problems are known, the easier it is to find a NP-complete problem to reduce from.

Most reductions are somewhat complex.

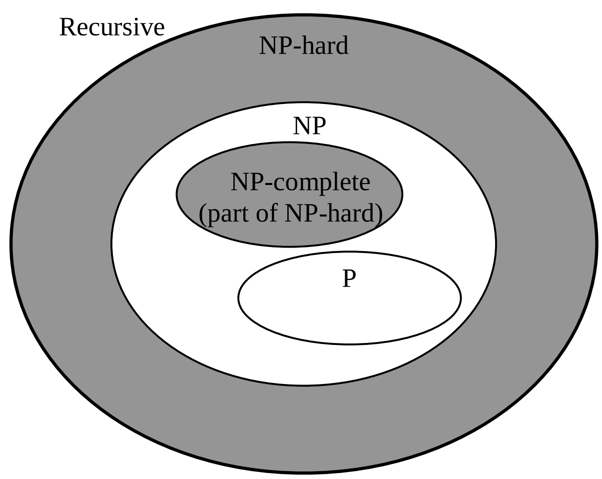
It is sufficient to show that a restricted version of the problem is NP-complete.

More Theory

NP has a rich structure that includes more than just P and NP-complete. This structure is studied in later courses on the theory of computation.

The set of recursive problems outside of NP (and including NP-complete) are called *NP-hard*. There is a proof technique to show that such problems are at least as hard as NP-complete problems.

Space complexity addresses how much tape does a TM use in deciding a language. There is a rich set of theories surrounding space complexity.



Dealing with NP-completeness

You will likely run into NP-complete problems in your career. For example, most optimization problems are NP-complete.

Some techniques for dealing with intractable problems:
- Recognize when there is a tractable special case of the general problem.
- Use other techniques to limit the search space.
- For optimization problems, seek a near-optimal solution.

The field of *linear optimization* springs out of the latter approach. Some linear optimization solutions can be proven to be "near" optimal.

A branch of complexity theory deals with solving problems within some error bound or probability.

For more: Read <u>Computers and Intractability: A Guide to the Theory of NP-Completeness</u> by Michael R. Garey and David S. Johnson, 1979.